Malware Analysis with Symbolic Execution and Graph Kernel

Charles-Henry Bertrand Van Ouytsel[1][0000−0001−5720−6569] and Axel Legay[1][0000−0003−2287−8925]

INGI, ICTEAM, Université Catholique de Louvain,
Place Sainte Barbe 2, LG05.02,01, 1348 Louvain-La-Neuve, Belgium
{charles-henry.bertrand,axel.legay}@uclouvain.be

Abstract. Malware analysis techniques are divided into static and dynamic analysis. Both techniques can be bypassed by circumvention techniques such as obfuscation. In a series of works, the authors have promoted the use of symbolic executions combined with machine learning to avoid such traps. Most of those works rely on natural graph-based representations that can then be plugged into graph-based learning algorithms such as Gspan. There are two main problems with this approach. The first one is in the cost of computing the graph. Indeed, working with graphs requires one to compute and representing the entire state-space of the file under analysis. As such computation is too cumbersome, the techniques often rely on developing strategies to compute a representative subgraph of the behaviors. Unfortunately, efficient graph-building strategies remain weakly explored. The second problem is in the classification itself. Graph-based machine learning algorithms rely on comparing the biggest common structures. This sidelines small but specific parts of the malware signature. In addition, it does not allow us to work with efficient algorithms such as support vector machine. We propose a new efficient open source toolchain for machine learning-based classification. We also explore how graph-kernel techniques can be used in the process. We focus on the 1-dimensional Weisfeiler-Lehman kernel, which can capture local similarities between graphs. Our experimental results show that our approach (1) outperforms existing ones by an impressive factor, (2) is resistant to static adversarial attacks.

Keywords: Malware Analysis · Symbolic Execution · Malware Classification

1 Introduction

According to the independent IT security institute AV-Test [5], the number of malware infections has increased significantly over the last ten years, reaching a total of 1287.32 million in 2021. With approximately 450 000 new malware every day, companies spend on average 2.4 millions dollars [1] on defenses against such malicious software. For this reason, effective and automated malware detection

and classification is an important requirement to guarantee system safety and user protection.

Most malware classification approaches are based on the concept of signature and signature detection. A malware signature, which is often built manually, represents the DNA of the malware [13,18,7]. Consequently, deciding whether a binary file contains a specific malware boils down to checking whether the signature of such malware is present in the binary. The simplest type of signature is the syntactic signature, i.e., signatures based on syntactic properties of the malware binaries (length, entropy, number of sections, or presence of certain strings). Alternatively, behavioral signatures are based on the malware's behavioral properties (interaction with the system and its network communications).

Different types of signature give rise to different malware classification approaches. In static malware analysis approaches, the classification boils down to detecting the presence of a given static signature directly in the binary that has been disassembled. This signature often boils down to a sequence of characters[15]. The two main advantages of this approach is that it is fast and does not require executing the malware. On the other hand, static signatures are very sensitive to obfuscation techniques that modify the binary code to change its syntactic properties [22]. An illustration of those limitations is given in [7,25] where the authors show the approach is not robust to variants of the MIRAI malware. Other static approaches use machine learning algorithms (see [34] for an illustration in the context of Android systems). Several works shows that those are not resistant to adversarial examples (e.g:[32]).

Another classification approach is that of dynamic analysis, which executes the malware and observes if its effect on the system corresponds to some behavioral signature [21,38]. This approach is based on the fact that a static obfuscation does not modify the behavior of the malware and therefore has no influence on the classification of a behavioral signature. To avoid infecting the analyst's system and to prevent the malware from spreading, the malware is commonly executed in a sandbox. Unfortunately, malware can implement sandbox detection techniques to determine whether they are being executed in a sandbox. As dynamic analysis is limited to one execution, a malware can pass detection by avoiding exhibiting malicious behavior [3]. More information on static and dynamic malware analysis can be found in the following tutorial [7].

Aware of those limitations, several authors have proposed using some exploration techniques coming from the formal verification areas. This includes symbolic execution [12,14,10], a technique that explores possible execution paths of the binary without either concretizing the values of the variables or dynamically executing the code. As the code exploration progresses, constraints on symbolic variables are built and system calls tracked. A satisfiability-modulo-theory (SMT) checker is in charge of verifying the satisfiability of the collected symbolic constraints and thus the validity of an execution path.

The advent of symbolic execution has led to the development of a new set of machine learning-based fully automatised malware classification methods. Those continue and extend the trend of applying machine learning to malware classi-

fication [20,35,34]. In particular, in [25] the authors have proposed combining symbolic execution with Gspan [37], a machine learning algorithm that allows us to detect the biggest common subgraphs between two graphs. In its training phase, the algorithm collects binary calls via symbolic analysis. Such calls are then connected in a System Call Dependency Graph (SCDG), that is a graph that abstracts the flow of information between those calls. Gspan can compute the biggest common subgraphs between malware of a given family. Those then represent the signature for the family. In its classification phase, the approach extracts the SCDG from the binary and compares it with each family's signature.

Unfortunately, the above-mentioned approach has several limitations. The first one is that it depends on the efficiency of the symbolic analysis engine. The second one is that SCDGs are built as an abstraction of the real behavior of the binary. In particular, the approach will connect two calls that have the same argument even though those calls may be from different function. Such a choice, which is motivated by efficiency reasons, may lead to a crude over-approximation of the file's behavior and hence to misclassification.

Relying on the biggest common subgraphs may exclude important but isolated calls that are specific to the malware. In addition, using graphs poses a particular challenge in the application of traditional data mining and machine learning approaches that rely on vectors. To surmount those limitations, the authors in [24] proposed using a graph kernel [23], which can be intuitively understood as a function measuring the similarity of pairs of graphs. In their work, the authors used the approach in a non-supervised process. However, such kernel can be plugged into a kernel machine, such as a support vector machine. Results in [24] show that the graph kernel outperforms Gspan in terms of accuracy. Unfortunately, the kernel used in [24] still implicitly relies on detecting the biggest commonalities between graphs. Individual important calls are out of its scope.

Our paper makes several contributions to improving symbolic analysis-based malware classification. The first contribution consists in a flexible and open source implementation of a malware analysis toolchain based on [31] (available here [6]). In addition to obtaining better performances, the flexibility of the new implementation allows us to plug in and compare various classification algorithms and symbolic execution strategies. In particular we develop and compare several efficient resource-based strategies that enable us to build compact but more informative SCDGs than those in [25,24]. The approach is able to distinguish more SCDGs and hence obtain a finer grain in both training and classification processes. Another important contribution of this paper is the comparison of the Weisfeiler-Lehman Kernel [30] with other classifiers. Such a graph kernel is capable of comparing the graph's local small structures by a ingenious relabelling of its vertices. Finally, a major contribution of the paper is a series of experimental results showing that our approach outperforms those in [25] and in [24] when being used in a supervised context. We also show that our approach outperform static approaches when facing adversarial examples.

2 On Graph Comparison for Malware Analysis

This section briefly introduces several notions related to graphs. It also outlines the limits of graph-based representation in malware analysis and advantages of graph kernels.

A graph G is defined as a pair (V, E), where V is a set of Vertices and E a set of edges such that $\{\{u, v\} \subseteq V | u \neq v\}$. The set of edges and vertices of G are given by $E(G)$ and $V(G)$, respectively. We also consider labelled graphs where a label function $l : V(G) \to \Sigma$ assigns a label from Σ to each vertex of G. We use $l(v)$ to denote the label of vertices v. A graph G' = (V',E') is a *subgraph* of G=(V,E) if $V' \subseteq V$ and $E' \subseteq E$. We are interested in applying graph comparison to extract and compare malware signatures represented by SCDGs. In particular, graph isomorphism is considered to be a powerful tool that allows us to detect structural similarities between graphs that may not be identical. Two unlabelled graphs G and H are said to be isomorphic ($G \simeq H$) if there exists a bijection $\phi : V(G) \to V(H)$ such that $(u, v) \in E(G)$ if and only if (1) $(\phi(u), \phi(v)) \in E(H)$ (for all $u, v \in V(G)$), and (2) $l(v) = l(v')$ for each $(v, v') \in \phi$. There exists a wide range of graph similarity measures. This includes, e.g., subgraph isomorphism used to compute the largest common subgraph. Checking graph isomorphism is known to be NP. Moreover, reducing the comparison of two graphs to checking their isomorphism is known to be restrictive as it requires both graphs to have same structure. This situation is rarely encountered when comparing (classes of) malware. The situation is illustrated in Figure 1, where two malware from the same family are considered to be different since Vertex *SetFilePointer* cannot be covered by an isomorophic relationship.

To address the problem, authors in [25] proposed an approach based on Gspan. This is a popular algorithm for frequent graph-based pattern mining. Given a set of graphs $\mathbb{G}$ and a desired support *min_supp*, Gspan (whose pseudo-code is given in Appendix A) tries to extract all subgraphs present at least in *min_supp* graphs of $\mathbb{G}$. If $\mathbb{G}$ represents a set of malware from the same family, the set of common subgraphs represents their signatures.

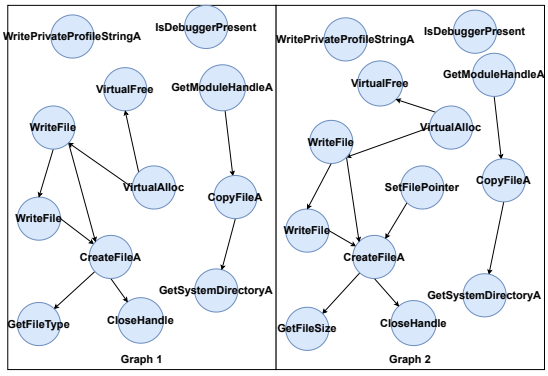


Fig. 1: Example of non-isomorphic graphs with high similarities

Unfortunately, relying on computing the biggest subgraphs may dismiss small but important connected components that do not belong to the biggest subgraphs. The situation is illustrated in Graph 3 of Fig. 2, where important calls such as *IsDebuggerPresent* may be ignored. An inefficient solution could be to extend the number of subgraphs. Unfortunately, when bigger graphs than in our example are involved, this ap-

proach will mostly favor a variant of the biggest connected component, as we will see in Section 4. In order to address this problem, we resort to the concept of Graph Kernels.

2.1 Graph kernels

In machine learning, *kernel methods* are algorithms that allow us to compare different data points with a particular similarity measure. Consider a set of data points X such as $\mathbb{R}^m$ and let $k : X \times X \to \mathbb{R}$ be a function. Function k is a valid kernel on X if there exists a Hilbert space $\mathcal{H}_k$ and a feature map $\phi : X \to \mathcal{H}_k$ such that $k(x, y) = \langle \phi(x), \phi(y) \rangle$ for $x, y \in \mathcal{X}$, where $\langle \cdot, \cdot \rangle$ denotes the inner product of $\mathcal{H}_k$. It is known that ϕ exists only if k is a positive semidefinite function. A well-known kernel is the Gaussian radial basis function (RBF) kernel on $\mathbb{R}^m$, $m \in \mathbb{N}$, defined as:

$$k_{RBF}(x, y) = exp(-\frac{\|x - y\|^2}{2\sigma^2}) \tag{1}$$

with σ, the bandwidth parameter. Observe that RBF kernel gives an explicit definition of ϕ. In practice, this is not always required. Indeed, algorithms such as Support Vectors Machine (SVM) use the data X only through inner products between data points. Having the kernel value $k(x, y)$ between each data point is thus sufficient to build an SVM-based classifier. This approach is known as the *kernel trick* [16]. A Gram matrix K, is defined with respect to a finite set of point $x_1, .., x_n \in X$. Each element $K_{i,j}$ with $i, j \in \{0, .., n\}$ represents the kernel value between pairs of points $k(x_i, x_j)$. If the Gram Matrix K of Kernel k is positive semi-definite for every possible set of data points, then k is a valid kernel.

It is common for kernels to compare data points using differences between data vectors. However, the structures of graphs are invariant to permutations of their representations (i.e., ordering of edges/vertices does not influence structure and distance between graphs). This motivates the need to compare graphs in ways that are permutation invariant. Moreover, to avoid strict comparison (which would be equivalent to isomorphism), it is common to use smoother metrics of comparison, such as *convolutionnal kernels*, for better generalization capabilities. Convolutionnal kernels divide structures (i.e., graphs in our case) into substructures (e.g., edges, subgraphs, paths, etc) and then evaluate a kernel between each pair of such substructures.

In [24], the authors propose a similarity metric for malware behavior graphs based on common vertices and edges. Concretely, they define a similarity σ between two graphs G and H as:

$$\sigma(G, H) = \alpha\sigma_{vertices}(G, H) + (1 - \alpha)\sigma_{edges}(G, H) \tag{2}$$

where α is the vertice-edge factor allowing to adjust weights of vertices and edges in the similarity function (set to 0.25 in the conclusion of their work). The

vertice similarity is defined as:

$$\sigma_{vertices}(G,H) = \frac{|\mathcal{V}(G) \cap \mathcal{V}(H)|}{min(\mathcal{V}(G), \mathcal{V}(H))} \tag{3}$$

and the edge similarity as:

$$\sigma_{edges}(G,H) = \frac{|\mathcal{CC}_{max}(G \cap H)|}{min(|\mathcal{CC}_{max}(G)|, |\mathcal{CC}_{max}(H)|)} \tag{4}$$

where $\mathcal{V}(G)$ are the set of vertices of G and $\mathcal{CC}_{max}(G)$ is the biggest connected component of G.
While this approach adds information related to all nodes labels compared to Gspan, it suffers from similar drawbacks than Gspan. Indeed, it focus on the biggest connected component, neglecting edges in other connected components. This problem is illustrated on Graph 4 of Fig. 2. One can see that the kernel identifies similarities between nodes of Graph 1 and Graph 2. However, it ignores important edge dependencies such as *GetModuleHandle*, *CopyFileA*, and *GetSystemDirectoryA*.

To tackle this problem, a popular approach in graph kernels is the comparison of local structure. In this approach, two vertices of different graphs are considered to be similar if they share the same labels. The two vertices are considered to be more similar if, in addition, they share similar neighborhoods (i.e., vertices with the same labels). Using this approach, Shervashidze et al. [30] introduced graph kernels based on the 1-dimensional Weisfeiler-Lehman (WL). Let G and H be graphs, and $l : V(G) \cup V(H) \to \Sigma$ be a function giving their vertices labels. By several iterations $i = 0, 1, ...$, the 1-WL algorithm computes a new label function $l_i : \mathcal{V}(G) \cup V(H) \to \Sigma$, with each iteration allowing comparison of G and H. Let $N(v)$ be the neighborhood of a vertex $v \in G$ in $V(G)$, i.e., $N(v) = \{u \in V(G) | (v,u) \in E(G)\}$. In the first iteration, $l_0 = l$, and in subsequent iterations,

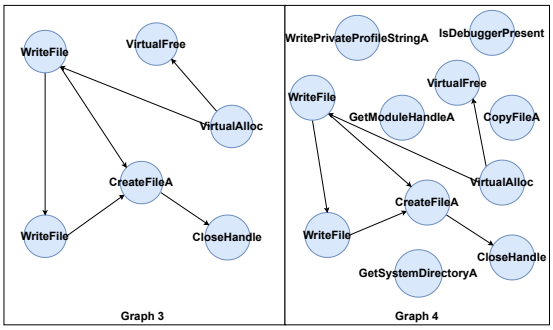


Fig. 2: Graph 3 represents the subgraph extracted with Gspan from graphs of Fig. 1. Graph 4 corresponds to the extraction with the kernel from [24]

$$l_i(v) = relabel(l_{i-1}(v), sort(l_{i-1}(u) | u \in N(v))) \tag{5}$$

with $v \in V(G) \cup V(H)$, sort(S) returning a sorted tuple of S and function $relabel(p)$ maps the pair p to a unique value in Σ which is not already used in previous iterations. When the cardinality of l_i equals the cardinality of l_{i-1},

the algorithm stops. The idea of the WL sub-tree kernel is to compute the previous function for $h \geq 0$ and after each iteration i to compute a feature vector $\phi_i(G) \in \mathcal{R}^{|\Sigma_i|}$ for each graph G, where $\Sigma_i \subseteq \Sigma$ denotes the image of l_i. Each component $\phi_i(G)_{\sigma_j^i}$ counts the number of appearances of vertices labelled with $\sigma_j^i \in \Sigma_i$. The overall feature vector $\phi^{WL}(G)$ is defined as the concatenation of the feature vectors of all h iterations, i.e.,

$$\phi^{WL}(G) = (\phi^0(G)_{\sigma_1^0}, ..., \phi^0(G)_{\sigma^0_{|\Sigma_0|}}, \phi^h(G)_{\sigma_1^h}, ..., \phi^h(G)_{\sigma^h_{|\Sigma_h|}}) \tag{6}$$

Finally, to compute similarity between two different feature vectors, we apply the following formula:

$$k_{WL}(G, G') = \sum_{\phi \in \phi^{WL}(G)} \sum_{\phi' \in \phi^{WL}(G')} \delta(\phi, \phi') \tag{7}$$

where δ is the Dirac kernel, that is, it is 1 when its arguments are equals and 0 otherwise. The more labels the two graphs have in common, the higher this kernel value will be. Compared with Gspan and the kernel from [24], this kernel also targets similarities related to all nodes and edges of the biggest subgraph but also local similarities. This is illustrated in Fig. 3, where dependencies between *GetModuleHandle*, *CopyFileA*, and *GetSystemDirectoryA* are kept in the learning process.

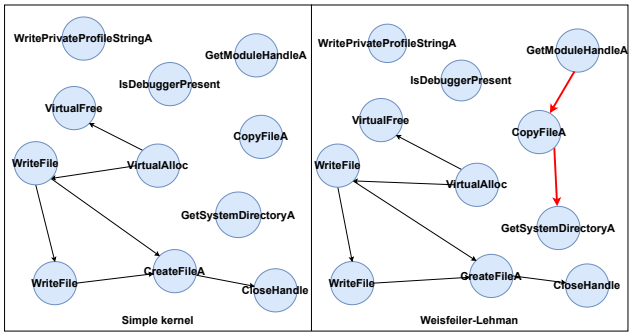


Fig. 3: The figure shows that, contrary to Simple Kernel, the Weisfeiler-Lehman kernel captures all common edges between the graphs of Figure 1

3 Approach: Symbolic execution + Machine Learning for Malware analysis

We propose an open source toolchain for malware analysis that is based on machine learning and SCDGs (available here [6]). The toolchain relies on the following important components: the first component consists in collecting and

labelling a series of binaries from different malware families. Then, **angr** [31], a python framework for symbolic execution, is used to execute those files. The result is used to extract a SCDG for each such binary. One of the contributions of this paper will be to improve and adapt the symbolic engine to malware analysis as well as the construction of SCDGs. Those SCDGs are then used to train machine learning algorithms. If Gspan is used, the training will result in common subgraphs to represent signatures for each family. If SVM is used, a Gram matrix between all the malware programs is created. Finally, the toolchain also contains supervised classifiers. If Gspan is used, the SCDG of the new malware is compared with those of the signature of each family and the classifier retains the one with the closest distance. If SVM is used, a Gram matrix is created between all trained malware and the new malware. This matrix is then used in the SVM classification process. A main contribution of this paper is to compare those two types of classification.

3.1 Extraction of calls

The construction of the SCDG is based on Symbolic Execution. This approach envisages the exploration of all the possible execution paths of the binary without either concretizing the values of the variables or dynamically executing the code (i.e., the binary is analyzed statically). Instead, all the values are represented symbolically. As the code exploration progresses, constraints on symbolic variables are built and system calls tracked. A satisfiability-modulo-theory (SMT) checker is in charge of verifying the satisfiability of the collected symbolic constraints and thus the validity of an execution path. A wide range of tools and techniques have been developed for efficient symbolic execution analysis. Most of those techniques agree on the fact that symbolic execution still suffers from state-space-explosion and, consequently, only a finite set of symbolic paths can be explored in a reasonable amount of time. This is particularly the case with malware analysis where the classification process must be done with very limited resources. As the calls that form the SCDG are collected directly from those symbolic paths, the choice of which paths to follow will have an impact on the machine learning process.In a recent work, authors showed how SMT solving could impact performances [28,8,11]. In this paper, we focus on path selection strategies. The work in [25] implements a Breadth-First Search (BFS) approach, that is, at each execution step all ongoing paths are explored simultaneously. This approach leads to an important growth of states and memory usage. As we have limited resources, we propose to explore one subset of paths at a time. We prioritize states from which one can explore new assembly instruction addresses of the program. Our Custom Breadth-First Search Strategy (**CBFS-Strategy**) is presented in Algorithm 1. The algorithm begins by taking L states for exploration from the set of available states and putting them in the list R of states to explore next (line 4). It then iterates among all other available states. If it finds a state leading to an unexplored part of the code or with a shorter path of execution (line 6), it puts it in R and takes out a state with a lower priority (i.e: state not leading to a new instruction or state with a longer depth). After going

through each state, it returns R to allow angr to perform a new execution step on R' states. In addition to **BFS-Strategy**, we also implemented a Custom Depth-First Search Strategy (**CDFS-Stategy**), which is presented in Algorithm 2 (the main difference with CBFS-Strategy being the condition to select successor state at Line 6). Observe that symbolic execution with this coverage heuristic is not new. However, the implementation and evaluation of restricted versions within a tool for malware classification are.

Algorithm 1 CBFS exploration

```
1: Inputs: A set of states: S
2: Limit of states: L
3: Outputs: A set of L states: R
4: R ← S[: L]
5: for state s ∈ S do
6:     if new(s.next_ip)|{∃state ∈ R|s.depth < state.depth & !new(state.next_ip)} then
7:         Remove state from R
8:         Add s to R
9: Return R
```

Algorithm 2 CDFS exploration

```
1: Inputs: A set of states: S
2: Limit of states: L
3: Outputs: A set of L states: R
4: R ← S[: L]
5: for state s ∈ S do
6:     if new(s.next_ip)|{∃state ∈ R|s.depth > state.depth & !new(state.next_ip)} then
7:         Remove state from R
8:         Add s to R
9: Return R
```

Another important challenge in symbolic execution is that of handling loops. Indeed, the condition of such loops may be symbolic. In addition, the loop may create an infinite repetitive behavior. In those situations, deciding between staying in the loop or exiting the loop remains a tricky choice that has been the subject of several works focusing on the possibilities, which include Loop-extended Symbolic Execution [26], Read-Write set [9], and bit-precise symbolic mapping [36]. As those approaches may be too time-consuming, we propose to reuse two intermediary heuristics from [25]. The first one applies to loops whose condition contains a symbolic value. Such loops may give rise to two states at each iteration: one that exits the loop for those symbolic values that exceed the condition and one that remains within the loop for other values, with this last state being used again to iterate on the loop. We chose to stop such iteration after four steps. For loops that do not contain symbolic values, since such loop may still lead to an unbounded number of behaviors, our approach consists in limiting the execution to a finite, arbitrary fixed, number of steps and then forcing the execution to exit the loop.

3.2 Creating SCDGs

Symbolic execution allows us to obtain several paths representing executions of a given binary. Our next step is to collect the sets of calls present on each such path as well as their addresses and arguments. Those are used to build the SCDG corresponding to this binary. Following [25], SCDGs are graphs where each vertex is labelled with the name of a system call; and the edges correspond to (an abstraction of) information flow between these calls. Concretely, each

SCDG is built from the symbolic representation by merging and linking calls from one or more symbolic paths. Their construction could be influenced by applying different **SCDG-Strategies**.

Consider first the creation of a graph execution from one symbolic path. We consider three types of edge. In the first one, two calls are linked if they both share an argument with identical value (i.e: one-edge strategy). This is, for example, the case of two calls with the same file handler. The second link is established between two calls that both have an argument with the same symbolic value. An example is a symbolic file size returned by a call and passed to a second call added to another value. In addition, we consider that two calls can be linked if an argument of the first call is the calling address of the second one. This situation typically arises in dynamic loading of a library. We also label each edge with the index of the argument in both calls (return value of a call is given index 0). The three-edges strategy is called **SCDG-Strategy 1** and the one-edge strategy is called **SCDG-Strategy 2**. Our experiments shows that **SCDG-Strategy 2** loses important dependency between calls and leads to more isolated nodes in SCDGs. Indeed, this strategy suffers from two types of problem. First, symbolic values may be modified before being passed to another call. Second, some calls used by obfuscation techniques exhibit address-arguments links. A typical example is given by *GetProcAddress*, used to hide real content of the import table of PE files.

An example of an SCDG is given in Fig. 4 with **SCDG-Strategy 1** and **SCDG-Strategy 2**. The program first calls *CreateFile*, which returns a handle to the file with the specified filename. A vertex is thus constructed for *CreateFile*. Then, a call to *SetFilePointer* on the preceding file handle occurs. This leads to the creation of a new vertex (*SetFilePointer*). Since the returned argument of *CreateFile* (index 0) is the same as the first argument of *SetFilePointer* (index 1), an edge is added between them. Vertices *ReadFile* and *WriteFile* are created and linked following similar principles.

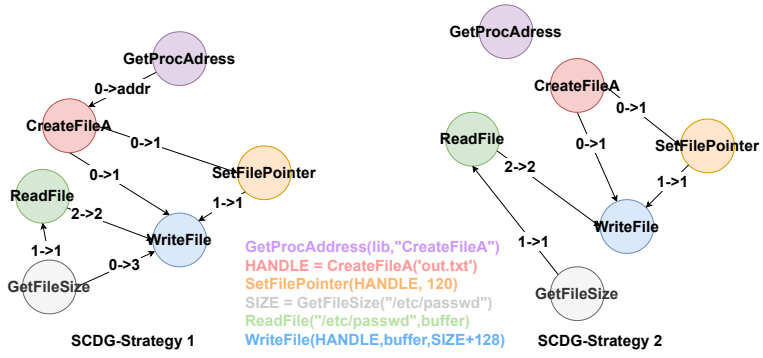


Fig. 4: Illustration of a SCDG built with **SCDG-Strategy 1** and **SCDG-Strategy 2**

There are situations where different calls in the same execution share the same API name and occur at the same instruction address but with distinct arguments. One may decide to merge the two calls into one single vertex. In this case, we conserve the set of arguments of the first call observed in the execution. This merge incurs a loss of precision but leads to a more compact SCDG representation [28]. This may be of importance when one has to train the system with a large number of different types of malware. This merging strategy is called **SCDG-Strategy 3**. Merging calls gives different advantages. First, it decreases the size of the SCDG, which may lead to better classification/detection performances. In addition, it may reduce the impact of some calls in the learning phase. An example is given with the wabot malware, which uses a hundred of calls to *WriteFile* during its execution. If not merged, those calls that are not part of the main actions of the malware will constitute an important part of the signature. This may have a negative impact on the training phase. On the other hand, there are situations where **SCDG-Strategy 3** merges calls with different goals. This situation may result in losing part of the malware behavior.

The above strategies apply to single symbolic paths only. When several symbolic paths are considered, one can decide to produce an SCDG that is composed of the disjoint union of such executions. Such a strategy is referred to as **SCDG-Strategy 5**. On the other hand, **SCDG-Strategy 4** consists in merging successive executions from different symbolic paths. **SCDG-Strategy 5** is simplier to compute, but **SCDG-Strategy 5** gives smaller graphs. According to our experimental results, **SCDG-Strategy 4** may speed up the computation time by an exponential factor for families with high symbolic execution numbers.

3.3 Creating a classification model and evaluate new samples

The toolchain uses SCDGs to train a classifier which is used to detect and classify malware. We have implemented two classifiers. One implementation is based on Gspan and follows the idea from [25]. Another one implements the graph kernel from [24] and the Weisfeiler-Lehman extension we outlined in Section 2.

The classifier that uses Gspan implementation works by extracting signatures from malware families. We obtain the signature of each family by computing the biggest subgraphs between the SCDG of each malware. In the classification phase, we compare the SCDG of new binary with those of each signature. The file belongs to the malware family whose graph is the closest to the binary's.

For the case of graph kernel, the training phase consists in computing the feature vector that corresponds to applying the algorithm in [24] or the Weisfeiler-Lehman extension to each malware of the family. As explained in the background section, the algorithm produces a Gram matrix between all those vectors. This matrix represents an implicit version of the kernel. A support vector machine can then exploit this implicit representation. In the classification phase, we compute a Gram matrix between the feature vector of the binary under classification and the vectors of all malware used in the training set. Observe that, contrary to the Gspan approach, graph kernel does not require us to produce an explicit and hence all-encompassing representation of the signature of each family.

4 Experimental Results

This section describes the methodology used to assess our toolchain's performance in both extracting SCDGs and classifying new binaries. Our evaluation set was composed of 1874 malware divided into 15 families plus 150 cleanware samples. The data set's exact composition is given in Table 1. In terms of origins, 64 percent of the samples were obtained thanks to a direct collaboration with Cisco. The remaining 36 percent were extracted from MalwareBazaar [2]. Samples were labelled using AVClass [27], a python tool to label malware samples. This tool is fed with VirusTotal reports and outputs the most likely family of each sample. To evaluate detection performance, we used 150 open source programs found online [33]. To show the relevance of approach, we also compare our classifier performance with a SVM classifier trained on Ember [4] static features. Those features are known to be representative features for existing static machine learning approach. We then demonstrate how easily it could be fooled with adversarial examples generated with framework such as MAB-malware [32].

In the rest of the section, all experiments were performed on a desktop PC with an Intel Core i7-8665U CPU (1.90GHz x 8) and 16GB RAM running Ubuntu 18.04.5. Our experimental results relied on our ability to extract SCDGs efficiently. In all experiments, we used a timeout of ten minutes for each SCDG. Note that 20 percent of the SCDGs were computed in time while 80 percent never computed entirely. For the case of BFS-Strategy, we used the same parameters as in [28] (loop threshold of 4, unlimited number of states to explore, z3 optimization enabled). However, for CDFS-Strategy and CBFS-Strategy we imposed a limit of 10 states that could be explored simultaneously. All metrics for multi-class classification are weighted average.

Family	#samples	Family	# samples
bancteain	91	remcosRAT	476
delf	78	sfone	32
fickerstealer	44	sillyp2p	269
gandcrab	92	simbot	126
ircbot	36	sodinokibi	75
lamer	61	sytro	115
nitol	71	wabot	134
redlinestealer	35	cleanware	150

Table 1: Composition of the dataset

Environment modelling Proper environment modelling is a major challenge in developing efficient symbolic execution techniques. Indeed, when we apply symbolic execution we avoid exploring/executing API call code. Since performing such an operation would drastically increase the computation time [19]. In angr, when a call to an external library occurs, the call is hooked to a simulated procedure called *simprocedures* that will produce the symbolic outputs for the function. A simple but crude implementation of such procedure is to assume that the external function returns a symbolic value without any constraint. In such a case, *simprocedures* simply returns symbolic values covering the full range of outputs given in the specification. In practice, such a solution gives good results in 26 percent of the analyzed families. However, this solution may generate outputs that are not defined in the specification. In addition, it ignores many

potential effects of the call, which include the modifications of input parameters or the number of its arguments. This may lead to incoherent executions if those parameters impact the rest of the execution (e.g.: in branch choices). We propose several improvements to fix those issues. The first one consists in restricting the ranges of outputs to those given in the specification. As an example, if the output is an integer variable that can take only four values, *simprocedures* would generate those values instead of the full range of integers. Another one concerns the case where an execution is blocked because modifications of some arguments by the external call are not performed. This happens in situations where the external call may modify some of its inputs or even some environment variables. In such case, we emulate several potential modifications with concrete values. Observe that this improvement work must be performed for each call in our dataset that causes problems. That is why, we have constituted a *simprocedures* library that is constantly enriched with experiments and calls.

We first apply Gspan to SCDGs obtained with combinations of different strategies. Signatures are obtained by sampling randomly 30% of the SCDGs of each family; those SCDGs constitute the training set. Other SCDGs are then classified to assess the quality of those signatures; those SCDGs constitute the test set. This process is repeated three times and performance is averaged.

SCDG-strategy					**BFS-Strategy**			**CBFS-Strategy**			**CDFS-Strategy**		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.685	0.566	0.619	0.721	0.604	0.657	0.652	0.609	0.629
x		x		x	0.623	0.552	0.585	0.674	0.568	0.616	0.61	0.587	0.5983
x			x		0.683	0.589	0.632	0.698	0.555	0.619	0.669	0.597	0.631
x				x	0.609	0.534	0.569	0.651	0.534	0.586	0.629	0.54	0.581
	x	x	x		0.684	0.651	0.667	0.736	0.655	0.693	0.693	0.639	0.664
	x	x		x	0.614	0.594	0.614	0.686	0.615	0.648	0.645	0.586	0.614
	x		x		0.679	0.587	0.629	0.623	0.465	0.532	0.68	0.597	0.636
	x			x	0.602	0.556	0.578	0.587	0.448	0.508	0.632	0.578	0.603

Table 2: Results of Gspan classifier with different exploration strategies

In Table 2, we observe that **CDFS-Strategy** generally outperforms **CBFS-Strategy** and **BFS-Strategy**. By inspecting the results, we observed that **BFS-Strategy** ran out of memory for 7 percent of the binaries, thus reducing its performance compared to **CBFS-Strategy**. While **SCDG-strategy 4** showed improvements with **SCDG-strategy 5**, it should be noted that **SCDG-strategy 5** entails significant overhead in SCDG building (up to 100 times slower) and signature size (5 times bigger on average). In general, the best performances were obtained by combining **SCDG-strategy {2,3,4}**. Upon inspecting best classifier in Fig. 6, we see a lot of confusion between different classes. This can be explain by plotting similarities between signatures built with Gspan, as illustrated in Fig. 5. There, different signatures share important similarities, leading to confusion between different malware families, as illustrated in Fig. 6.

This problem is directly linked to a problem exposed in Section 2, that is, Gspan focus on the biggest subgraph while neglecting other components.

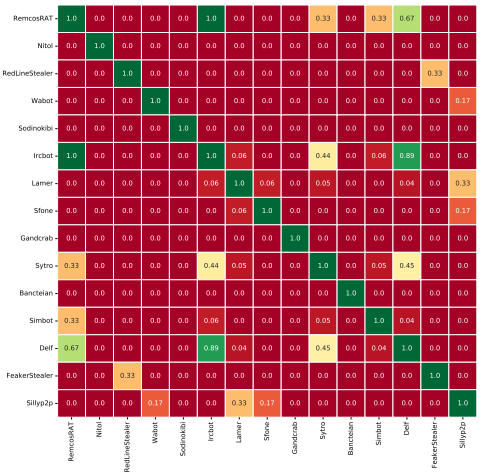


Fig. 5: Similarity matrix between signatures obtained with Gspan

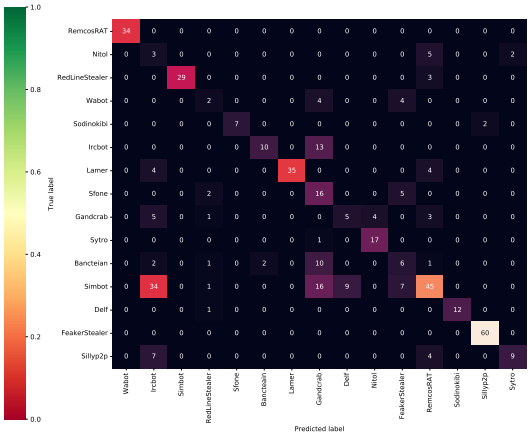


Fig. 6: Confusion matrix obtained for Gspan with **CBFS-strategy** and **SCDG-strategy {2,3,4}**.

We now turn to applying kernel from [24]. Table 3 shows that overall performance increased compared with Gspan. Moreover, **CBFS-Strategy** and **CDFS-Strategy** outperformed both **BFS-Strategy** and **SCDG-strategy {2,3,4}** strategies appeared to be more efficient. However, Fig. 7 shows that several families were still indistinguishable.

SCDG-strategy					**BFS-Strategy**			**CBFS-Strategy**			**CDFS-Strategy**		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.754	0.691	0.721	0.845	0.812	0.828	0.833	0.787	0.8093
x		x		x	0.728	0.673	0.699	0.785	0.736	0.759	0.77	0.714	0.741
x			x		0.742	0.71	0.725	0.827	0.778	0.801	0.82	0.754	0.786
x				x	0.711	0.649	0.678	0.771	0.703	0.735	0.74	0.698	0.718
	x	x	x		0.769	0.723	0.745	0.851	0.826	0.838	0.847	0.813	0.829
	x	x		x	0.738	0.645	0.688	0.813	0.752	0.781	0.802	0.738	0.768
	x		x		0.747	0.632	0.684	0.798	0.747	0.771	0.835	0.772	0.802
	x			x	0.714	0.654	0.682	0.781	0.733	0.756	0.763	0.718	0.739

Table 3: Results of SVM classifier and kernel from [24] with different exploration strategies

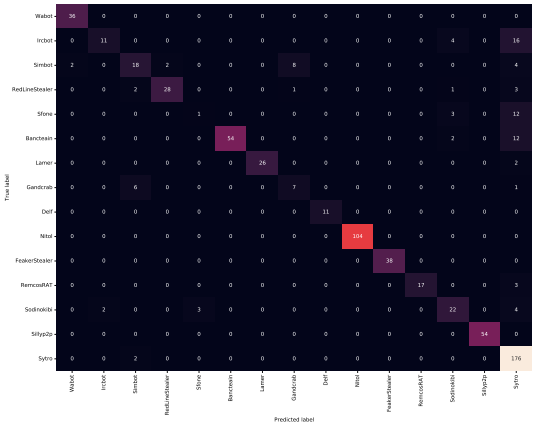


Fig. 7: Confusion matrix obtained for SVM classifier and kernel from [24] with **CBFS-strategy** and **SCDG-strategy {2,3,4}**

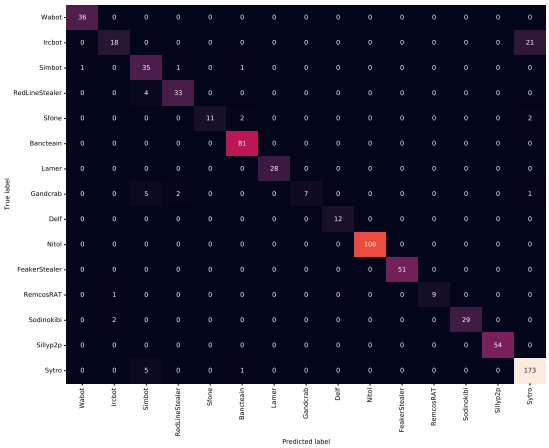


Fig. 8: Confusion matrix obtained for WL kernel with **CDFS-strategy** and **SCDG-strategy {1,4}**.

Weisfeiler-Lehman kernel (WL). Finally, we investigated the SVM classifier with the WL kernel. The results in Table 4 clearly outperformed the others, reaching an F_1-score of 0.929 with **CDFS-Strategy** and **SCDG-strategy {1,3,4}**. Families were better distinguished, as illustrated in Fig. 8. Those results confirm our supposition of Section 2: taking advantage of an SCDG's local structure increases the efficiency of machine learning in malware classification.

Adversarial Examples. Contrary to a static approach, our observed classifiers are resistant to adversarial examples based on static features. This is illustrated in Table 5 where we show that the MAB-malware adversarial framework [32] can easily mutated and corrupt Ember features [4] used in most of static classifiers. This fact highlights that, although offering good performance, those static classifiers are highly vulnerable and there is a need to develop new approaches.

Training time. In general, WL kernel outperforms Gspan by a factor of 15 and Kernel in [24] by a factor of 10 000. We suspect that the overhead is due to the extensive use of pairwise graph mining in the similarity metric presented in Section 2. Compare to the Kernel in [24], Gspan reduces these number of computation since it first create a signature for each family before comparing those signature with the binary to classify.

These experiments permit to draw several conclusions. First, **SCDG-strategy 1** gives overall better results than **SCDG-strategy 2** with WL kernel. That is not the case for the other classifier where these information seems to lead to overfitting and **SCDG-strategy 2** should be preferred. Moreover, the impact of **SCDG-strategy 3** varies. While it improves classification for kernels that

SCDG-strategy					**BFS-Strategy**			**CBFS-Strategy**			**CDFS-Strategy**		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.852	0.847	0.846	0.865	0.864	0.852	0.936	0.931	**0.929**
x		x		x	0.832	0.824	0.827	0.85	0.842	0.846	0.915	0.91	0.912
x			x		0.895	0.891	0.892	0.894	0.881	0.874	0.937	0.933	**0.929**
x				x	0.847	0.836	0.841	0.86	0.851	0.855	0.918	0.911	0.914
	x	x	x		0.86	0.855	0.857	0.897	0.879	0.867	0.929	0.925	0.924
	x	x		x	0.812	0.795	0.803	0.867	0.862	0.864	0.885	0.877	0.881
	x		x		0.895	0.891	0.891	0.895	0.891	0.886	0.939	0.933	**0.929**
	x			x	0.834	0.828	0.831	0.862	0.858	0.859	0.891	0.887	0.888

Table 4: Results of SVM and WL kernel with different exploration strategies

Model	Initial dataset			Mutated dataset		
	Precision	Recall	F_1-score	Precision	Recall	F_1-score
Gspan	0.911	0.914	0.911	0.911	0.914	0.911
SVM kernel from [24]	0.965	0.95	0.957	0.965	0.95	0.957
SVM Weisfeiler-Lehman	0.989	0.975	0.981	0.989	0.975	0.981
SVM Ember features	0.964	0.965	0.964	0.763	0.564	0.502

Table 5: Comparison in the context of adversarial examples (mutated dataset)

are based on the biggest common subgraph, its impact when combined with other strategies varies. Finally, while **SCDG-strategy 5** leads to a considerable overhead, it does not improve performance of any classifier. On the other hand, **SCDG-strategy 4** leads to more compact signatures, better computation times and good classification performances. Regarding exploration strategies, **BFS-strategy** is generally outperformed by **CBFS-strategy** while **CDFS-strategy** outperforms other strategies when used with the WL graph kernel.

5 Future Work

We propose a new efficient approach for malware detection. Directions for future work includes new exploration heuristics, such as concolic executions [29] or smart sampling [17]. Another objective is to apply our kernel in a non-supervised approach like in [24]. We are also interested in implementing a distributed version of the toolchain. In this context, the federated learning paradigm should allow us to combine information from different contributors. We also plan to investigate resistance to adversarial examples based on semantical modifications of malware. In addition, we will continue to improve our toolchain with new *simprocedure* and plugin interfaces.

Acknowledgments. Charles-Henry Bertrand Van Ouytsel is an FRIA grantee of the Belgian Fund for Scientific Research (FNRS-F.R.S.). We would like to thank Cisco for their malware feed, VirusTotal for their API and the CyberExcellence project funded by the Walloon Region under convention 2110186.

A Gspan algorithm

The Gspan algorithm is presented hereunder in Algorithm 3. Given a dataset of graphs $\mathbb{G}$ and a desired support *min_supp*, Gspan tries to extract all subgraphs present at least in *min_supp* graphs of $\mathbb{G}$. If $\mathbb{G}$ represents a set of malware from the same family, the output set of common subgraphs $\mathbb{S}$ represents their signatures. To this purpose, Gspan defines a DFS Code of a graph G as an ordered edge sequence constructed from a DFS exploration of G. Original Gspan paper [37] defines a way to order these DFS codes, allowing to define a unique minimum DFS code for a graph G ($min(.)$ function in Algorithm 3). Given a DFS code s, c is called a child of s if it expands s with a new edge to create a valid DFS code.

Algorithm 3 Gspan algorithm

```
In: A set of graphs:𝔾
Out: A set of common subgraphs 𝕊
Sort labels in 𝔾 by their frequency
Remove infrequent vertices/labels
Relabel remaining vertices and edges
𝕊^1 ← all frequent 1-edge graphs in 𝔾
Sort 𝕊^1 in lexicographical order
𝕊 ← 𝕊^1
for each edge e ∈ 𝕊^1 do
    initialize s with e
    Subgraph_mining(𝔾 ,𝕊, s )
    𝔻 ← 𝔻 − e
    if |𝔾| < min_supp then
        break
Return 𝕊
```

Algorithm 4 Main procedure *Subgrap_mining* of Gspan algorithm

```
if s ≠ min(s) then
    return;
𝕊 ← 𝕊 ∪ {s}
enumerates s in each graph in 𝔾
and count its children;
for each c; c is a child of s do
    if support(c) ≥ min_supp then
        s ← c
        Subgraph_mining(𝔾 ,𝕊, s )
```

References

1. Eighth Annual Cost of Cybercrime Study. https://www.accenture.com/us-en/insights/security/eighth-annual-cost-cybercrime-study, accessed: 2021-10-29
2. MalwareBazaar by abuse.ch, fighting malware and botnets. https://bazaar.abuse.ch/, accessed: 2021-10-29
3. Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D.: Malware dynamic analysis evasion techniques: A survey. ACM Computing Surveys (CSUR) **52**(6), 1–28 (2019)
4. Anderson, H.S., Roth, P.: Ember: an open dataset for training static PE malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
5. AV-Test: AV-Test, the independent IT-Security institute (2021), https://www.av-test.org/en/statistics/malware/
6. Bertrand Van Ouytsel, C.H., Crochet, C., Dam, K., Legay, A.: SEMA : a toolchain using Symbolic Execution for Malware Analysis. To appear in 17th International Conference on Risks and Security of Internet and Systems (CRiSIS) (2022)

7. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: An overview of malware detection and evasion techniques. In: International Symposium on Leveraging Applications of Formal Methods. pp. 565–586. Springer (2018)
8. Biondi, F., Josse, S., Legay, A., Sirvent, T.: Effectiveness of synthesis in concolic deobfuscation. Computers & Security **70**, 500–515 (2017)
9. Boonstoppel, P., Cadar, C., Engler, D.: RWset: Attacking path explosion in constraint-based test generation. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 351–366. Springer (2008)
10. Cadar, C., Ganesh, V., Sasnauskas, R., Sen, K.: Symbolic execution and constraint solving (dagstuhl seminar 14442). Dagstuhl Reports **4**(10), 98–114 (2014)
11. Chen, Z., Chen, Z., Shuai, Z., Zhang, G., Pan, W., Zhang, Y., Wang, J.: Synthesize solving strategy for symbolic execution. In: Cadar, C., Zhang, X. (eds.) ISSTA. pp. 348–360. ACM (2021)
12. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 653–656. IEEE (2016)
13. Faruki, P., Laxmi, V., Bharmal, A., Gaur, M.S., Ganmoor, V.: Androsimilar: Robust signature for detecting variants of android malware. Journal of Information Security and Applications **22**, 66–80 (2015)
14. Godefroid, P.: Test generation using symbolic execution. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
15. Griffin, K., Schneider, S., Hu, X., Chiueh, T.c.: Automatic generation of string signatures for malware detection. In: International workshop on recent advances in intrusion detection. pp. 101–120. Springer (2009)
16. Hofmann, M.: Support vector machines-kernels and the kernel trick. Notes **26**(3), 1–16 (2006)
17. Jegourel, C., Legay, A., Sedwards, S.: Importance splitting for statistical model checking rare properties. In: International Conference on Computer Aided Verification. pp. 576–591. Springer (2013)
18. Kirat, D., Vigna, G.: Malgene: Automatic extraction of malware analysis evasion signature. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 769–780 (2015)
19. Lin, Y.: Symbolic Execution with Over-approximation. Ph.D. thesis, University of Melbourne, Parkville, Victoria, Australia (2017)
20. Macedo, H.D., Touili, T.: Mining malware specifications through static reachability analysis. In: European Symposium on Research in Computer Security. pp. 517–535. Springer (2013)
21. Massicotte, F., Couture, M., Normandin, H., Michaud, F.: A testing model for dynamic malware analysis systems. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST). pp. 826–833. IEEE (2012)
22. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-third annual computer security applications conference (ACSAC). pp. 421–430. IEEE (2007)
23. Nikolentzos, G., Vazirgiannis, M.: Learning structural node representations using graph kernels. IEEE transactions on knowledge and data engineering **33**(5), 2045–2056 (2019)

24. Puodzius, C., Zendra, O., Heuser, A., Noureddine, L.: Accurate and robust malware analysis through similarity of external calls dependency graphs (ecdg). In: The 16th international conference on availability, reliability and security (ARES). pp. 1–12 (2021)
25. Said, N.B., Biondi, F., Bontchev, V., Decourbe, O., Given-Wilson, T., Legay, A., Quilbeuf, J.: Detection of mirai by syntactic and behavioral analysis. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). pp. 224–235. IEEE (2018)
26. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ICST). pp. 225–236 (2009)
27. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: Avclass: A tool for massive malware labeling. In: International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). pp. 230–253. Springer (2016)
28. Sebastio, S., Baranov, E., Biondi, F., Decourbe, O., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Optimizing symbolic execution for malware behavior classification. Computers & Security p. 101775 (2020)
29. Sen, K.: Concolic testing: a decade later (keynote). In: Xu, H., Binder, W. (eds.) WODA@SPLASH. ACM (2015)
30. Shervashidze, N., Schweitzer, P., Van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-Lehman graph kernels. Journal of Machine Learning Research **12**(9) (2011)
31. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157. IEEE (2016)
32. Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., Yin, H.: Mab-malware: A reinforcement learning framework for blackbox generation of adversarial malware. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (AsiaCCS). pp. 990–1003 (2022)
33. Sébastien, C.: Portable freeware dataset (Jan 2019)
34. Taheri, R., Ghahramani, M., Javidan, R., Shojafar, M., Pooranian, Z., Conti, M.: Similarity-based android malware detection using hamming distance of static binary features. Future Generation Computer Systems **105**, 230–247 (2020)
35. Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Computers & Security **81**, 123–147 (2019)
36. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 921–937 (2017)
37. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: 2002 IEEE International Conference on Data Mining, 2002. pp. 721–724. IEEE (2002)
38. Zhang, Z., Qi, P., Wang, W.: Dynamic malware analysis with feature engineering and feature learning. In: Proceedings of the AAAI conference on artificial intelligence. pp. 1210–1217. AAAI (2020)