Accurate and Robust Malware Analysis through Similarity of External Calls Dependency Graphs (ECDG)

Cassius Puodzius, Olivier Zendra, Annelie Heuser, Lamine Noureddine

Accurate and Robust Malware Analysis through Similarity of External Calls Dependency Graphs (ECDG)

Cassius Puodzius
INRIA
Rennes, France
cassius.puodzius@inria.fr

Olivier Zendra
INRIA
Rennes, France
olivier.zendra@inria.fr

Annelie Heuser
IRISA
Rennes, France
annelie.heuser@irisa.fr

Lamine Noureddine
INRIA
Rennes, France
lamine.noureddine.inria.fr

ABSTRACT

Malware is a primary concern in cybersecurity, being one of the attacker's favorite cyberweapons. Over time, malware evolves not only in complexity but also in diversity and quantity. *Malware analysis automation* is thus crucial. In this paper we present ECDGs, a shorter call graph representation, and a new similarity function that is *accurate* and *robust*. Toward this goal, we revisit some principles of malware analysis research to define basic primitives and an evaluation paradigm addressed for the setup of more reliable experiments. Our benchmark shows that our similarity function is very efficient in practice, achieving speedup rates of 3.30x and 354, 11x wrt. *radiff2* for the standard and the cache-enhanced implementations, respectively. Our evaluations generate clusters that produce almost unerring results - homogeneity score of 0.983 for the accuracy phase - and marginal information loss for a highly polluted dataset - NMI score of 0.974 between initial and final clusters of the robustness phase. Overall, ECDGs and our similarity function enable autonomous frameworks for malware search and clustering that can assist human-based analysis or improve classification models for malware analysis.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

binary code analysis, similarity, call graph, malware

ACM Reference Format:
Cassius Puodzius, Olivier Zendra, Annelie Heuser, and Lamine Noureddine. 2021. Accurate and Robust Malware Analysis through Similarity of External Calls Dependency Graphs (ECDG). In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3465481.3470115

IWCC'21, August 17–20, 2021, All-Digital Conference

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3470115

1 INTRODUCTION

Malware (MW) attacks are among the most common ways to compromise IT systems, and are growing in complexity and quantity over time. The number of known MW exceeded one billion instances in 2019 and 1.21B in Apr. 2021 [1]. Each day, over 350K new MW are discovered, a much larger number of programs being analyzed. Historically, MW analysis has heavily resorted to manual creation of signatures for detection and classification. This human-action-based method is very costly and time consuming, thus unable to handle the exponential growth in number of unique MW instances[15]. A solution is to widely *automate* MW analysis.

Overall, this work aims to improve the computational cost of existing (and possibly future) MW *search* and MW *clustering* algorithms. Nonetheless, creating new algorithms for these problems is outside the scope of this work. Instead, we address the backbone question "How to compute similarity of unknown programs with high accuracy while being friendly to search and clustering algorithms for MW analysis?", this way:

RQ1 How to get more precise structural representations of programs wrt. state-of-the-art (with no information loss)?
RQ2 How to exploit this structural representation to define a similarity function that is friendly to binary code search/clustering schemes?
RQ3 Establishing ground truth for MW analysis is an undecidable problem, so how to evaluate this similarity function?

We propose *External Calls Dependency Graphs* (ECDGs) as new call graph representation, and a *similarity function* (σ^{ECDG}) that is reliably *accurate* and *robust*[1]. We show that they lead to an *efficient computation* of binary code similarity able to underpin the construction of frameworks for MW search and clustering. In our experiments, σ^{ECDG} provides highly descriptive *cluster prototypes* that can help to scale up clustering, assist human-based analysis and improve classification models for MW analysis. We devote special attention to the *evaluation methodology*, an intricate issue that directly influences research validity but is often overlooked. For this, we propose the *Accuracy and Robustness (AnR) paradigm* as guideline to create more *reliable* experiments.

As main contributions, we:

- revisit the foundations of MW analysis, defining basic (MW analysis) primitives, and proposing the Accuracy and Robustness

[1]Robustness relates to the ability to resist to semantic transformation.

(AnR) paradigm for more reliable evaluation methodologies; [addresses **RQ3**]
- propose ECDGs, as a more a compact call (dependency) graph enabling more efficient binary similarity computation. [addresses; **RQ1**]
- propose a new similarity function for ECDGs that is efficient, accurate and robust. [addresses **RQ2**]

Our implementation also provides practical contributions, namely the study of symbolic execution to trace external calls, the evaluation of gSpan as a practical algorithm for sub-graph isomorphism, and the evaluation of cluster prototypes extraction to represent MW families. Our whole evaluation is based on experiments with MW samples collected in the wild from real-world dataset feeds.

This paper is organized as follows. Section 2 presents background. Section 3 states foundations of MW analysis with principles guiding our work. Section 4 proposes ECDGs and similarity function σ^{ECDG}. Section 5 shows experimental validation. Section 6 discusses results. Section 7 presents related works. Section 8 concludes.

2 BACKGROUND

This section sets the notations used in this work (2.1) and recalls notions for binary analysis concepts (2.2), frequent subgraph mining (2.3) and clustering (2.4).

2.1 Notations and Definitions

Let $\sim$ denote an equivalence relation on A and $x \in A$. The **equivalence class** of x is the set of all elements of A that are related to x, i.e. $[x]_\sim = \{y \in A | x \sim y\}$.

Given a set S, its **indexed family** $\mathcal{I}_S$ consists of an index set defined by a surjective function $x : \mathcal{I}_S \rightarrow S$ such that $i \rightarrow x_i = x(i), \forall i \in \{1, \cdots, |S|\}$.

A **labeled graph** is notated $G(V, E, L_V, L_E, \varphi)$, where V is a multiset of nodes (also notated as $\mathcal{N}(G)$), $E \subseteq V \times V$ is a set of edges, L_V and L_E are sets of node and edge labels respectively, and φ is a label function that defines the mapping $V \rightarrow V_L$ and $E \rightarrow L_E$.

A **directed graph** is a graph where edges E are ordered pairs of elements of V. A **directed acyclic graph** is a directed graph with no directed cycle.

Given two graphs $G_1(V_1, E_1, L_{V_1}, L_{E_1}, \varphi_1)$ and $G_2(V_2, E_2, L_{V_2}, L_{E_2}, \varphi_2)$, G_1 is **a subgraph** of G_2 if G_1 satisfies: (i) $V_1 \subseteq V_2$, and $\forall v \in V_1$, $\varphi_1(v) = \varphi_2(v)$, (ii) $E_1 \subseteq E_2$, and $\forall (u, v) \in E_1$, $\varphi_1(u, v) = \varphi_2(u, v)$. This relationship is notated $G_1 \subseteq G_2$.

G is a **connected graph** if it contains a path for every pair of vertices in it. G is **disconnected** (or **disjoint**) otherwise.

A subgraph of G is a **connected component** iff there exists a path between any pair of vertices in it. We notate the set of all **connected components** of G as $\mathcal{CC}(G)$. The **largest connected component** (in number of edges) is noted as $\mathcal{CC}_{max}(G)$.

Given the graphs G_1 and G_2, G' is a **common subgraph** of G_1 and G_2 iff $G' \subseteq G_1 \wedge G' \subseteq G_2$. We note the **largest common subgraph** (which can be disjoint) as $G_1 \cap G_2$.

More information on graphs can be found in [41].

2.2 Binary Analysis: Syntax, Semantics and Structural Representations

The primary goal of binary analysis in the context of MW analysis is to determine whether an unknown file embodies malicious behavior. To this end, MW analysts extract and interpret different properties, called *features*, related to the file syntax or semantics. *Syntactical features* characterize file code representation, with properties like strings, opcodes, header information, etc. *Semantic features* relate to the file functionalities. Some representations lie between syntax and semantics, being very tied to syntax, yet able to capture semantics. This is the case for graph representations such as *control flow graphs* and *call graphs*, which are globally called *structure representations*.

When binary diffing is performed for MW analysis, it is useful to determine whether two binary codes originate from the same source code. Two files are *identical* if they have the same syntax, *equivalent* if they have the same semantics and *similar* if they have similar syntax, structure or semantics [17].

Identical or nearly identical files have similar codes, so meaningfully comparing syntactical features generally results in few false positives. Yara rules [2] are the *de facto* standard for syntactical signature writing. However, *robustness* relates to the ability to resist to semantic transformation [17], so syntactical features are considered less *robust*. MW often use obfuscation techniques to change their code representation without changing their functionalities.

2.3 Frequent Subgraph Mining (FSM)

Given a graph dataset $D = \{G_0, G_1, \ldots, G_n\}$, its *support* $\mathcal{S}_D(g)$ is the number of graphs in D of which g is a subgraph. Frequent subgraph mining (FSM) is finding all the subgraphs in a given graph that appear more than a given number of times, i.e. finding any subgraph g s.t. $\mathcal{S}_D(g) \geq \tau_{min}$, where τ_{min} is a minimum support threshold.

An overview of common subgraph algorithms is given in [18], where gSpan (a graph-based substructure pattern mining) [47] is one of the most frequently cited FSM algorithms. gSpan builds a new lexicographic order and maps each graph to a unique minimum depth-first search code which represents a canonical label. Based on this order, gSpan adopts the depth-first search strategy to efficiently mine frequent connected subgraphs. gSpan uses less memory than algorithms based on embedding lists. Experiments show that gSpan outperforms related works by an order of magnitude [19].

2.4 Clustering

Algorithms. Clustering is an unsupervised learning method: no labels are provided to the algorithm. It is often used to find meaningful structures, explanations of underlying processes, generative features, and inherent groupings in a set of samples. Clustering divides data points into groups, so that points from a same cluster are similar, and dissimilar to other clusters points.

In our work we consider algorithms belonging to three main clustering types and that can use custom similarity measures: In **Density-based** clustering, clusters are areas of higher density compared to the remaining data set, and clusters can have arbitrary shapes. Data points in lower density areas are usually considered as noise and border points. Ex.: DBSCAN and OPTICS. **Hierarchical-based** clustering builds (bottom-up or top-down) a hierarchy of

clusters, with no prior information about the number of clusters. In the Agglomerative algorithm each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. Merges and splits are performed greedily. **Hybrid** HDBSCAN extends DBSCAN, converting it into a hierarchical clustering algorithm, then extracting a flat clustering based on clusters stability.

Metrics. Several metrics exist to quantify the quality of clusters. **Homogeneity score** [32] describes the amount of different classes within one cluster. A clustering achieves high homogeneity if all clusters contain only data points which are members of a single class. **Completeness** [32] describes the spread of one class over clusters. A clustering satisfies completeness if all data points that are members of a given class are elements of the same cluster. **Silhouette coefficient** [35] measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation). **Normalized Mutual Information (NMI)** [43] measures the statistical information shared between two clusters, normalizing it in $[0, 1]$. Additionally, we define the **Mean Similarity** of a cluster: let C be the set of points in a cluster and let $d(P_i, P_j)$ be the distance between P_i and P_j. The mean similarity is $\frac{1}{2|C|}\sum_{P_i \in C}\sum_{P_j \in C} d(i, j)$ where the factor $\frac{1}{2}$ accounts for the double counting of pairwise distances between two points.

3 MALWARE ANALYSIS: PRINCIPLES

In this section we discuss high-level principles related to MW analysis, used as guidelines in our work. First we present primitives for MW analysis, then our Accuracy-and-Robustness paradigm.

3.1 Primitives

A plethora of frameworks have been proposed for MW analysis. However, they are in general presented as full-fledged systems designed to fulfill specific tasks related to MW analysis (e.g. MW detection, MW clustering) without any standard regarding their fundamental building blocks, which hinders re-use and impairs analysis and comparison of component conceived in different works.

We thus propose *primitives* recurrently found in MW analysis. They can be seen as basic building blocks to design MW analysis frameworks without dealing with implementation details. Let S represent the set of elements, or *samples* (e.g. files, scripts, network traffic...), on which MW analysis is performed. Let $\sim$ denote an equivalence relation on S. A **family** F is defined as $F = [s]_\sim$, with $s \in S$ and we say that F is a **MW family** iff $\mathcal{F}_{Detection}(s) = 1, \forall s \in F$. **MW analysis primitives** are:

Detection: $\mathcal{F}_{Detection}$ inputs $s \in S$. It outputs a boolean indicating whether the input is malicious. $\mathcal{F}_{detection}(s) \rightarrow \{0, 1\}$.

Code similarity: $\mathcal{F}_{Sim}$ inputs $s_1, s_2 \in S$. It outputs their similarity. $\mathcal{F}_{Sim}(s_1, s_2) \rightarrow [0, 1]$.

Search: $\mathcal{F}_{Search}$ inputs $s \in S$. It outputs $M_f \subseteq S$, such that $f(x) = 1 \iff x \in M_f$ for a given matching function $f : S \rightarrow \{0, 1\}$. $\mathcal{F}_{Search}(s, f) \rightarrow M_f \subseteq S$.

Classification: $\mathcal{F}_{Classification}$ inputs $s \in S$ and a set of MW families F_i. It outputs family F_i, $s \in F_i$ (or $\emptyset$ if such family does not exist). $\mathcal{F}_{classification}(s, \{F_i\}) \rightarrow \{F_i, \emptyset\}$.

Clustering: $\mathcal{F}_{Clustering}$ inputs S. It outputs a set of families $\{F_i\}$, $\bigcup F_i = S$. $\mathcal{F}_{clustering}(S) \rightarrow \{F_i\}$.

Now we show that *search* and *clustering* schemes - a scheme is a specific instance of a primitive - can be derived from *code similarity*.

Lemma 3.1. *Any **code similarity** scheme can be extended to a **search** scheme.*

Proof. Given $s, s_1, s_2 \in S$ and a *code similarity* scheme $\mathcal{F}_{Sim}$, a *search* scheme $\mathcal{F}_{Search}(s, f_\tau)$ can be defined with

$$f_\tau(s_1, s_2) = \begin{cases} 1 & \text{if } \mathcal{F}_{Sim}(s_1, s_2) \geq \tau \\ 0 & \text{if } \mathcal{F}_{Sim}(s_1, s_2) < \tau \end{cases},$$

where τ is a pre-defined similarity threshold. □

Lemma 3.2. *Any **code similarity** scheme can be extended to a **clustering** scheme.*

Proof. Given a *code similarity* scheme $\mathcal{F}_{Sim}$, let $\mathcal{I}_S$ denote the indexed family of S. $\mathcal{M}_{Sim} = (m_{ij})$ is the *similarity matrix*, where $m_{ij} = \mathcal{F}_{Sim}(s_i, s_j)$, with $s_i, s_j \in \mathcal{I}_S \wedge i, j \in \{1, \cdots, |S|\}$. A *distance matrix* can also be defined as $1 - \mathcal{M}_{Sim}$. A *clustering* scheme $\mathcal{F}_{Clustering}$ is obtained by using any of the numerous algorithms working on similarity/distance matrix with predefined metrics. □

3.2 Accuracy-and-Robustness (AnR) Paradigm

Frameworks evaluation methodologies are key to MW analysis research, but systematic discussions about this topic are often neglected. Evaluations are based on measurements in experiments performed on datasets; however, different datasets may bring different levels of difficulty to different experiments. This issue becomes prominent when the evaluation methodology requires the definition of (MW) families to provide *labels* for performance measurement.

Formal studies on computer viruses (MW) show that defining a perfect detector is an undecidable problem [38], conflicting with the ambition of the detection primitive (see section 3.1). The *Template Matching Problem*, which decides whether a given piece of code matches a template behavior, to build semantic-aware detectors [11], is also *undecidable* [38]. Both results pinpoint that defining perfect (MW) families is theoretically *impossible*.

Li et al. [25] address this issue in MW clustering, studying whether performance results are biased towards high accuracy depending on the methodology followed to select ground truth. They first compare clustering results of prior work [7] against clustering frameworks from another domain (plagiarism detection), replicating the same experiment on the same dataset, plus a new one using all frameworks. As in [7], ground truth for the F-measure evaluation is set by selecting only samples for which different antivirus tools agree on labeling (antivirus consensus). All frameworks attained good scores on the first dataset (F-measure from 0.943 to 0.960), but performed poorly on the new one (F-measure from 0.609 to 0.630). The authors hypothesize this may come from the datasets difficulties (i.e. easy-to-cluster vs difficult-to-cluster), leaving the elaboration of a methodology to close this gap as open problem.

Towards this goal, **we propose the Accuracy-and-Robustness (AnR) paradigm as guideline in the evaluation of MW analysis frameworks**. It consists in conducting evaluation as a two-phased experiment, in which one phase assesses the *accuracy* attained by the framework, and the other its *robustness*.

The **accuracy phase** assesses the framework on a dataset likely to be *easy* to evaluate. The main goal is to verify whether the framework is able to generate results that are similar to the ground truth, which is assumed accurate by design. This means that the dataset composition should follow more *stringent methodologies* likely to provide less intrinsic disparity within samples, e.g.:

Outsource consensus: given a sample, this strategy establishes ground truth (MW family) by running a MW analysis on multiple anti-MW engines and selecting the label that comes out as consensus (if any). This strategy benefits from platforms like VirusTotal [3]. This approach weaknesses are inconsistencies among analysis results and lack of universal standards to generate labels [37]. As pointed by *Li et al.* [25], by enforcing consensus among results, the diversity representation within the dataset is drastically reduced.

Very specific string matching (VSSM) works on strings[2] that are strong indicators for a given sample and are used in very specific syntactical signatures, incurring extremely low false positive rates. This idea [34] is applied in YarGen [33].

Multiple stringent methodologies may be used to establish different ground truths on a same dataset, enabling cross-validation for further validation of the results. Furthermore, a balanced dataset should be privileged to improve the result's *significance* [25].

The **robustness phase** assesses the framework on a very diversified dataset able to represent a *real world scenario*. For this, the dataset should be built with *looser methodologies*, e.g.:

Manual labeling of the whole dataset, ideally following a unique guideline to assign labels to samples.

Direct outsourcing that takes as ground truth the output of one single anti-MW. Samples for which anti-MWs do not agree are kept, unlike in the consensus case.

Public, acknowledged signatures (PAS) employed to detect targeted MW families into the wild (ITW) are used. These rules are often tuned to avoid false positives, while being fairly general to maximize variant detection.

In the robustness phase it is essential to purposefully and gradually include *noise*[3] in the dataset, to measure the impact on evaluation metrics. This acts as a control group in the evaluation. Therefore, unlike in the accuracy phase, our primarily concern is to observe the inner differences of metric values as noise is inserted, not to seek straightforward correspondence with the ground truth.

4 EXTERNAL CALLS DEPENDENCY GRAPH (ECDG)

Here we introduce (4.1) *External Calls Dependency Graphs* (ECDGs) and an associated similarity function (4.2). ECDGs are compact, semantic-descriptive and robust *structural representations* (see section 2.2) of binary codes. Associated to our efficient similarity function, ECDGs can be used in practical search and clustering frameworks (see lemmas 3.1 and 3.2).

[2]Not only "text strings", but generic sequence of bytes, much like in Yara rules.
[3]Samples that do not correspond to any MW family.

4.1 ECDG Definition

A call graph is a direct graph whose nodes represent functions and edges represent one or more invocations of these functions [36]. An ECDG is a call graph whose nodes are restricted to *external calls*, meaning calls to external functions, i.e. system or library calls [21, 22][4]. ECDGs are modeled in form of a dependency graph whose edges represent shared arguments between external calls.

The ECDG definition resembles those of *malspec* [10] and *System-call Dependency Graphs (ScD-Graphs)* [29]. ECDGs may be considered as an instance of *malspec* in which the program main object code defines the boundaries of the *trusted computing base*, instead of the whole program as in the original work. For *ScD-Graphs*, the main differences are our extended scope (library calls not being overseen), and the use of labeled edges providing higher precision to our similarity function. Formally an **External Call Dependency Graph (ECDG)** is a directed acyclic graph $G(V, E, L_V, L_E, \varphi)$ whose node labels L_V are symbolic names of external functions, edges labels L_E are *def-use* dependencies [10] between these functions and φ is the labeling function.

4.2 Similarity Function Definition

Our similarity function definition targets the *largest common connected components* of the graphs as well as their *common nodes*. The common connected components capture common sub-behaviors, while the common nodes spot some degree of implementation resemblance. Our similarity function thus combines a *localized* behavioral view of the graphs, encompassed in the edges, with a *holistic* behavioral view, encompassed in the nodes. Frameworks based on features computed from plain call tracing are homologous to node-only analysis of ECDGs [4, 9, 20]. Formally:
The **similarity of two call graphs** G_1 and G_2 is defined as:

$$\sigma_\alpha(G_1, G_2) = \alpha\ \sigma_{nodes}(G_1, G_2) + (1 - \alpha)\ \sigma_{edges}(G_1, G_2),$$

where $\alpha \in [0, 1]$ is the **node-edge factor (nef)**.
Their **node similarity** σ_{nodes} is defined as:

$$\sigma_{nodes}(G_1, G_2) = \frac{|\mathcal{N}(G_1) \cap \mathcal{N}(G_2)|}{min(|\mathcal{N}(G_1)|, |\mathcal{N}(G_2)|)}$$

Their **edge similarity** σ_{edges} is defined as:

$$\sigma_{edges}(G_1, G_2) = \frac{|\mathcal{CC}_{max}(G_1 \cap G_2)|}{min(|\mathcal{CC}_{max}(G_1)|, |\mathcal{CC}_{max}(G_2)|)}$$

Thus, σ_α combines localized and holistic views of the graph, allowing efficient algorithms to independently compute σ_{edges} and σ_{nodes}. This would be impaired by greedier definitions that take many common disjoint components into account. We use σ^{ECDG} to denote the computation of our similarity function on ECDGs.

5 EVALUATION

This section evaluates ECDGs and σ^{ECDG} for *accuracy* and *robustness* with a data-driven approach following our AnR Paradigm (section 3.2). We thus measure and analyze different partitions of our dataset, under the premise that each partition is a "corrupted" version of a true partition but their combination converges to the truth. Since we obtain these partitions through unsupervised learning

[4]Unlike the references, we foresee the loading of new (external) functions at runtime.

(i.e. clustering) we hinge on external labels only for performance measurements - also under the premise that these labels represent a corrupted version of the truth. Furthermore, to optimize the computation of clusters, we choose algorithms that can work with precomputed pairwise similarities/distances (computed as $1-\sigma^{ECDG}$). This incurs a huge initial cost (i.e. $\mathcal{O}(n^2)$) to compute all pairwise distances in order to build the similarity/distance matrix. However, since it can be reused by different algorithms and parameterizations, it fits our goal of producing different partitions of our dataset.

In the remainder of this section we present the evaluation dataset (5.1), the evaluation framework (5.2) and experiments (5.3).

5.1 Evaluation Dataset

We rely on *syntactical signatures* (Yara rules) for dataset composition, because they are typically very precise, with very low false positive rates, and stable in terms of reproducibility across datasets.

To build our evaluation dataset, to avoid defective call traces (see section 5.2.1) we only consider files whose ECDGs have at least 100 edges. Furthermore, this dataset is divided in three groups: **Group I** contains 600 files matching with manually crafted Yara rules following the VSSM rationale, equally balanced into four families with no overlap between families. **Group II** contains 1,001 files from 16 different families defined by public, real-world Yara rules of public repositories (i.e. Yara-Rules, InQuest and McAfee ATR Team), following the PAS rationale, where families are unbalanced and may overlap. **Group III** contains 499 randomly chosen benign, cleanware (CW) files, with no prior knowledge about them. Table 1 shows the number of samples per family and their correponding class.

Table 1 Evaluation dataset

family	class	source (*"rare string"* or *[repository] yar file*)	#samples
Mira	I	"Mira.h"	150
Shohdi	I	"USR_Shohdi_Photo_USR"	150
Bogy	I	"BOGY'S GAME ENGINE"	150
TwarBot	I	"TwarBot"	150
spyeye	II	[Yara-rules] MALWMiscelanea.yar	162
Wabot	II	[Yara-rules] MALW_Wabot.yar	162
IceID_Bank_trojan	II	[Yara-rules] MALW_IcedID.yar	149
shylock	II	[Yara-rules] MALW_Miscelanea.yar	109
Bublik	II	[Yara-rules] MALW_Bublik.yar	88
sakula_v1_3	II	[Yara-rules] RAT_Sakula.yar	80
Njrat	II	[Yara-rules] RAT_Njrat.yar	58
njrat1	II	[Yara-rules] RAT_Njrat.yar	58
win_exe_njRAT	II	[Yara-rules] RAT_Njrat.yar	58
Cerberus	II	[Yara-rules] RAT_Cerberus.yar	50
Glasses	II	[Yara-rules] MALW_Glasses.yar	45
Mirage_APT	II	[Yara-rules] APT_Mirage.yar	41
ClamAV_Emotet_String_Aggregate	II	[InQuest] ClamAV_Emotet_String_Aggregrate.rule	36
Monero_Mining_Detection	II	[McAfee ATR Team] MINER_Monero.yar	27
Warp	II	[Yara-rules] MALW_Warp.yar	9
Cleanware	III	-	499

5.2 Evaluation Framework

Our MW analysis and clustering evaluation framework follows the basic design of call graph analysis frameworks in literature [12, 24], i.e. multiple stages in which 1) binary code is analyzed (*code analyzer stage*), 2) call graphs are generated (*graph generator stage*) and 3) call graphs are analyzed (*call graphs analyzer stage*).

5.2.1 Implementation. The **code analyzer stage** traces the calls invoked by the main binary object[5] to any shared object or system call (i.e. external call). We do this with *symbolic execution* using *angr* [40] *version 7.8.8.1* with *z3 version 4.5.1.0.post2*. Instead of setting breakpoints at the addresses of loaded functions, like plain tracing implementations (e.g *ltrace* [8]), our symbolic tracer steps through code, inspecting all active states whose execution addresses are outside the main object. When these states are found, hinting that the execution of some external procedure may be going on, an attempt to resolve the call is performed. To do it, the current address is looked up in *angr* internal mapping of addresses and *SimProcedures*[6], working as a generic hook to any external call.

To provide higher isolation for the analysis environment, we set angr to perform pure symbolic execution (*auto_load_libs* option false). Missing *SimProcedures* are replaced by stubs that only return an unconstrained symbolic variable, allowing to virtually resolve any runtime dependency. As a side effect these *stub SimProcedures* may alter the real execution of the file. Additionally, notwithstanding *angr*'s fine heuristics, unknown call conventions and unknown function signatures can also incur defective symbolic execution. Therefore, we extended *angr* with 24052 stub *SimProcedures* of commonly used functions with their correct call conventions and signature definitions, which proved to be a very effective tactics to improve the quality of symbolic executions. Our symbolic execution implementation is parameterized with the *step timeout*, *maximal number of active states*, *loop threshold*, and the underlying *z3 tactics*.

The **graph generator stage** builds ECDGs from call traces generated in the previous stage. Each call trace contains all external calls found during code analysis, with arguments and return value (if any). Note that symbolic execution may traverse multiple execution paths during analysis, so instead of generating a single (linked) list of calls it may produce a set of call lists. This opens new possibilities for the definition of call graphs, which may combine calls found in different execution traces.

Our implementation takes parameters to un/set *disjoint union*, *merge calls* and *merge trace*. When *disjoint union* is set, functions comprised in different call traces are not merged, but their interdependencies create edges between functions of all traces. When *merge calls* is set, similar functions are grouped into a single node. When *merge trace* is set, common call trace prefixes are combined into a single subtrace, thus transforming a set of call traces into a tree-like structure. All options can be combined, except *disjoint union* and *merge traces* that are mutually exclusive.

The **call graphs analyzer stage** uses an optimized implementation of *gSpan* [47] in C to compute *edge similarity* and Python code to compute the *node similarity* as well as global *similarity*.

Parametrization. Our evaluation framework comprises multiple stages, with specific parameters. Due to page limits, we only sketch the setup process and its implications.

Initially, the *code analyzer* and *graph generator* stages are optimized to maximize the quality of ECDGs. For this we analyze the correlation of different parameters for both stages using the *Analysis of Variance (ANOVA)*[42].

In this analysis we notice that *merging calls*, either directly or through *trace merging*, can undesirably suppress dependency edges,

[5] Memory segment corresponding to the program's binary code, without considering any shared object (i.e. dynamic loaded libraries)

[6] SimProcedures are symbolic summaries, implemented as Python functions, that mimic library functions.

Table 2 Benchmark evaluation

Stage	Dataset group	job time MEAN (SD, MED)	% common subgraph
Code analysis (CA) Graph generation (CG)	I + II + III	640.32s (419s, 478s)	
σ^{ECDG}	I	10.27s (25s, 3ms)	54,84%
σ^{ECDG}	II	8.67s (20,6s, 4ms)	57,45%
σ^{ECDG}	II + III	7.21s (19.5s, 4ms)	51,60%
radiff2	*Benchmark-DS*	4181.35s (6580s, 2031s)	
CA + GC + σ^{ECDG}	*Benchmark-DS*	1266.78s (570s, 1104s)	
cached-(CA + GC) + σ^{ECDG}	*Benchmark-DS*	11.81s (24s, 2.5s)	

which negatively impacts subsequent analysis. The *maximal number of active states* is positively correlated with ECDGs quality, but should be limited to reduce the analysis memory footprint. The consequences of *loop detection* is quite lurching and demands further investigation. *Z3 tactics* optimization has mild positive impact on ECDGs quality, while the *step timeout* only has minor effect. Thus, for our experiments we chose: code analysis stage with 8GB memory limit, 1h global timeout, 8s step timeout and up to 8 active states; graph generation stage with trace merging, calls merging disabled, disjoint union disabled.

5.2.2 Benchmark. To evaluate σ^{ECDG} time efficiency in a practical scenario, we benchmark our approach against *radiff2*, an open-source tool dedicated to binary comparison in radare2 suite [45]. For this, we tried to compute all pairwise similarities with the files of group I. However since radiff2 can take an overwhelming time for the computation of all pairwise similarities, we stopped the process after three weeks, obtaining $\sim 175k$ pairwise similarities (for each of radiff2 and σ^{ECDG}) that we named *Benchmark-DS*.

Our framework keeps a cache of the already computed ECDGs, which greatly speeds up the whole process. To measure the storage cost of this trade-off, we plotted the graph for *file size* vs. *#edges* taking into account over 10K files - omitted here for the sake of space. The scattered points have an almost linear profile and their linear regression results in a slope of 44.81(±0.02) edges by Kb of storage, where the biggest cache file takes 113Kb for 4806 edges. Table 2 notates this optimization by *cached*-(CA + GC) + σ^{ECDG}.

Table 2 shows that σ^{ECDG} largely outperforms (in time) *radiff2*, achieving a speedup gain of 3.30x and 354.11x for the standard and cache-enhanced implementations wrt. *radiff*.

It also shows that individual jobs in the *code analysis* and *graph generation* stage are much more expensive than the *graph analysis* (i.e. σ^{ECDG}) jobs. However the latter largely dominates overall calculation time because it requires a quadratic number of computations (to build the similarity/distance matrix), while the former ones are linear. So our modular approach allowed devoting more effort to the gSpan implementation, reducing memory footprint more than 100x, achieving 35x speedup with multi-threading and up to 6x speedup with a single thread wrt. original implementation [47].

To compare the similarity results of σ^{ECDG} and *radiff2*, their values were split into four categories: *strong-dissimilarity* ($\in [0, 0.25[$), *weak-dissimilarity* ($\in [0.25, 0.5[$), *weak-similarity* ($\in [0.5, 0.75[$) and *strong-similarity* ($\in [0.75, 0.1]$).

Figure 1 shows the contingency matrix of σ^{ECDG}/*radiff2*, where each cell contains the percentage of pairs in *Benchmark-DS* whose similarity felt into weak/strong similarity/dissimilarity (notated as SIM/~SIM) for σ^{ECDG} and *radiff2*. Ideally, supposing that σ^{ECDG} and

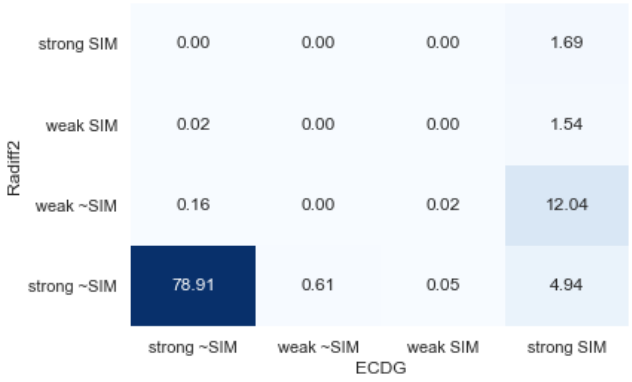

Figure 1: Contingency matrix of σ^{ECDG}/*radiff2*

radiff2 were both flawless, all values would be placed in the matrix main skew diagonal. However, we note that in general σ^{ECDG} is able to find stronger similarities than *radiff2* for the same pair of files. In particular, σ^{ECDG} found strong similarities where *radiff2* found weak and strong dissimilarities for 12.04% and 4.94% of the file pairs, respectively. A hypothesis to explain the differences in the similarity values obtained with σ^{ECDG} and *radiff2* is the fact that σ^{ECDG} targets semantic similarity whereas *radiff2* relies on comparisons of the programs' control-flow graph.

5.2.3 NEF Selection. We do an exploratory analysis to select a value for parameter *nef*, as required by the setup of σ^{ECDG} (section 4.2). Since *group I* contains files corresponding to the most trustworthy ground truth in the dataset, we use this group for the analysis with homogeneity score as our main metric of interest.

In this exploration, we build various similarity/distance matrices containing all pairwise computations of σ^{ECDG} with different *nef* values. For each matrix we compute clustering using different algorithms: Agglomerative, DBSCAN, HDBSCAN and OPTICS. For algorithms requiring input parameters (i.e. DBSCAN and Agglomerative), we performed a hyper-parameter tuning, which adds up to four more clustering instances in each *nef* iteration.

Figure 2 shows the *nef* exploration for HDBSCAN with homogeneity score as target metric - other curves are omitted here for the sake of space. Overall, the profile of all metric curves for OPTICS and HDBSCAN are fairly similar, whereas HBSCAN and Agglomerative have significantly different performances depending on the metric chosen as target. In all cases, the best scores are achieved with *homogeneity* or *silhouette* as target metric, while the worst are obtained with the *completeness* score. When tuned by silhouette score, both HDBSCAN and Agglomerative clustering displayed a decay in completeness score for $nef \geq 0.5$.

As outcome of the whole exploratory analysis, we selected the value 0.25 for *nef*, because it provides positive results for all metrics, in particular homogeneity score. It also takes both nodes and edges components of σ^{ECDG} into account, thus balancing both localized and holistic structures of the graphs.

We present and discuss only the clustering results of HDBSCAN in the following experiments, since the exploratory analysis shows very stable and good results for homogeneity and completeness.

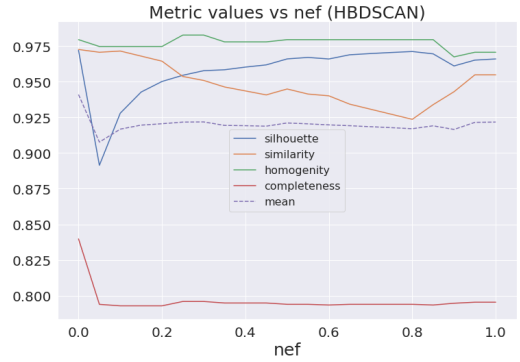

Figure 2: *nef* exploration for HDBSCAN

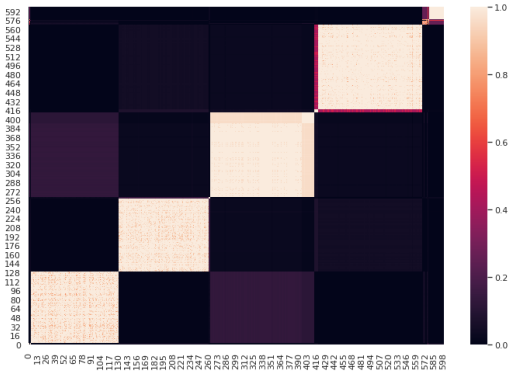

Figure 3: Heatmap of clusters in the *accuracy phase*

Table 3 Clusterings for $nef = 0.25$

	DBSCAN	HDBSCAN	OPTICS	Aggl.
#clusters	6	7	7	13
#noise	5	4	2	3
silhouette	0.969	0.955	0.956	0.978
similarity	0.946	0.954	0.949	0.991
homogeneity	0.995	0.983	0.978	0.995
completeness	0.846	0.796	0.796	0.828

5.3 Accuracy-and-Robustness (AnR) Analysis

Our experiments aim to evaluate σ^{ECDG} accuracy, robustness and practical efficiency, all prerequisites to enable functional application like MW search and MW clustering.

5.3.1 Accuracy phase. We evaluate σ^{ECDG} accuracy, on *group I* files only. This phase results directly derive from the *nef* selection exploratory analysis. Here we detail the corresponding experiment, i.e. clustering of *group I* files with HDBSCAN and $nef = 0.25$.

Table 4 *Accuracy phase* clusters

Cluster	#samples	Similarity	Yara Rule	#samples (per rule)
0	127	0.979	Shohdi	127
1	132	0.96	Shohdi	2
			TWarBot	130
2	132	0.998	Mira	132
3	18	1.000	Mira	18
4	6	0.998	Shohdi	6
5	150	0.994	Bogy	150
6	10	0.701	Shohdi	10
7	21	1.0	Shohdi	1
			TWarBot	20

Clustering results for this phase (see also table 3, column HDBSCAN) are shown in Table 4, and illustrated in Figure 3 heatmap. They contain 8 clusters and 4 singletons, with 0.955 silhouette, 0.954 mean similarity, 0.983 homogeneity and 0.796 completeness, the latter two computed with Yara rules created for *group I* files.

The clusters are very well discriminated. Only 3 samples are found in mixed clusters according to ground truth: two Shohdi samples in cluster #1 and one in cluster #7, both composed predominantely of TWarBot samples. *All other clusters are pure* and cluster #5 is complete, including all 150 Bogy samples.

5.3.2 Robustness phase. We evaluate σ^{ECDG} robustness, starting from *real world* samples (*group II* files), and assessing how clustering degrades as *noise* (*group III* files) is gradually inserted. Table 5 shows the clusterings (HDBSCAN with $nef = 0.25$) of the initial state (*group II* files) and of the final state (*group II* and *group III* files).

The initial clustering results in 20 clusters with 54 singletons (figure 5), with scores of 0.789 silhouette, 0.974 mean similarity, 0.746 homogeneity and 0.712 completeness, the latter two computed with public Yara rules as ground truth.

This experiment creates a majority (11 of 20) of pure clusters, especially for smaller clusters, which suggests that our σ^{ECDG} can discriminate MW families at variant level. For instance, *spyeye* produces 7 pure clusters and one almost pure (25 of 27 samples). In addition, some bigger clusters are completely or almost pure. Cluster #3 is pure, with 158 *Wabot* samples and cluster #8 includes 58 *Njrat* samples from a total of 59.

Cluster #1 (187 samples) has the greatest number of different families (6), dominated by *Bublik* (88 samples) and *sakula_v1_3* (80 samples). Furthermore, 13 samples are detected by both Yara rules, which suggests some commonality between both families. Similarly, cluster #17 includes all 109 *shylock* samples along with 26 *IceID_Bank_trojan*, two samples being detected by both Yara rules.

Our clustering also exposes lower quality Yara rules like *ClamAV_Emotet_String_Aggregrate* that perform poorly. Investigation reveals it was heuristically generated, as "a pruned aggregate of all Emotet related strings extracted from ClamAV on 2019-07-03". Yet, out of 36 samples detected by this rule, 9 sit together in pure cluster #0 and 13 others appear scattered in 5 other clusters. This result reinforces our claim that σ^{ECDG} is *accurate*.

To evaluate the robustness of our behavioral clustering, **we gradually introduce CW samples of *group III*** to act as noise in the dataset and we measure the disturbance. Figure 5 shows the initial clustering, before inserting CW samples in the dataset, and figure 6 shows the final clustering, after inserting the CW samples. Figure 4 shows the impact of *group III* files on clustering metrics.

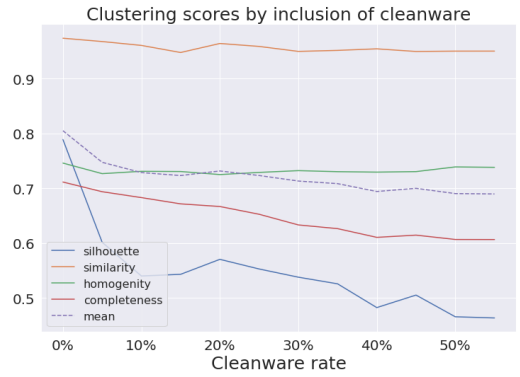

Figure 4: **Noise evaluation in the *robustness phase* [HDBSCAN (*nef*=0.25)]**

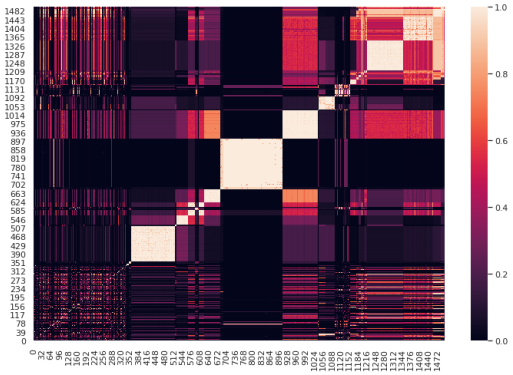

Figure 6: **Final clusterings: *group II* + *group III* files**

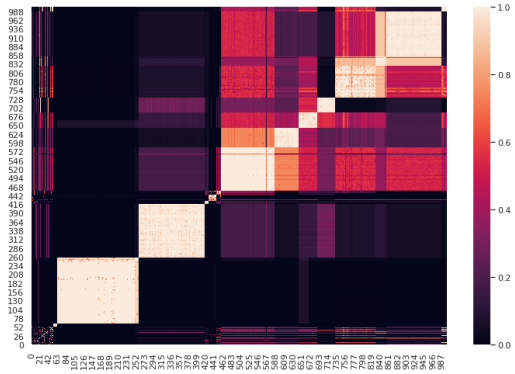

Figure 5: **Initial clusterings: *group II* files**

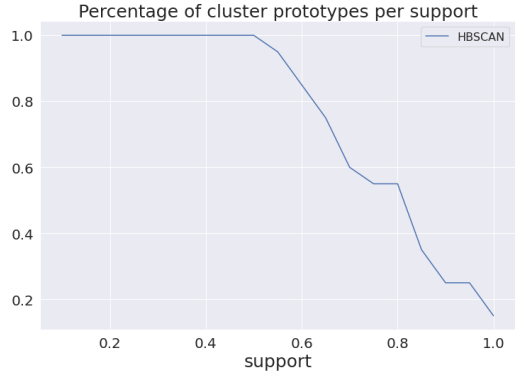

Figure 7: **Prototype exploration for the *robustness phase***

Homogeneity slightly increases after inserting 50% benign files (1/3 of the whole final dataset). Silhouette is the most affected metric, being impacted by new, rather small, spurious clusters. The final clustering achieved scores of 0.464 silhouette, 0.950 mean similarity, 0.738 homogeneity and 0.607 completeness. The NMI measured between the initial and final clusterings was 0.974, meaning that the information loss due to insertion of noise was only marginal.

Table 5 displays the clusters after complete insertion of *group III* files in the final clustering. As expected, most CWs are not included in any cluster, increasing the number of singletons (noise) from 30 in the initial clustering to 340 in the final one. Most original clusters remain unaffected (grey rows in table 5), only 4 being altered by inclusion of a few CWs (red rows). A few clusters end up with less or replaced files, which increases the clusters mean average similarity (green rows). Ten new clusters appear (yellow rows), four of them comprising only CWs, the others including a few MW samples but with relatively low average similarity.

This confirms σ^{ECDG} is *robust* wrt. highly polluted datasets.

5.4 Prototype Analysis

A cluster prototype is defined as *a data object that is representative of the other objects in the cluster* [44]. Here, for a given the cluster, we define the *cluster prototype* $\mathcal{P}_\tau(C)$ as the greatest common connected subgraph with minimum support threshold τ (see section. 2.3) for the entire set of graphs in this cluster. We explore *cluster prototypes* by incrementing τ from 0.05 to 1.0 with 0.05 steps, trying to compute prototypes for all clusters at each iteration. We repeat this with the clusterings of the *accuracy phase* and the initial clustering of the *robustness phase*.

For the *accuracy phase* clustering, all support values below 1 successfully produce prototypes for *all clusters*. Only one cluster fails to provide a prototype for support value equal 1. For the initial clustering of the *robustness phase*, when $nef \leq 0.5$ all clusters successfully produce prototypes, i.e. at least one common subgraph can be found in half of samples for each individual cluster. For greater *nef* values, the rate of prototypes decreases almost linearly. This means the clusters generally have a core of nearly half of their samples that are more similar, while the other less similar half gets gradually incorporated. Figure 7 shows the percentage of clusters that successfully produce prototypes as function of the support.

This behavior is convenient to unveil hidden similarities between MW families and variants, which is impossible with the all-or-nothing approach of syntactical signatures. Figure 8 shows an example of *cluster prototype* obtained for cluster#2.

6 DISCUSSION

This work proposes an efficient structural representation of programs that is reliably accurate and precise. We first tackled general research issues, namely MW primitives and MW analysis evaluation (see section 3). After showing that code similarity schemes can

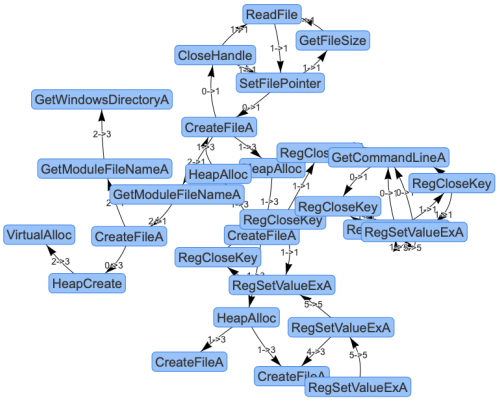

Figure 8: **Prototype of cluster #2 ($\tau_{min} = 1$), *robustness phase***

be extended to address search and clustering, we focused on analyzing ECDGs and σ^{ECDG} in terms of properties desired to implement practical frameworks.

ECDGs are obtained from dependencies between external calls, therefore they are beyond the reach of any purely syntactic obfuscation (e.g. instruction replacement, opaque predicates, etc.), although syntax obfuscation can still be effective in thwarting the underlying analysis for the call tracing (e.g. our symbolic execution). Some obfuscation techniques, such as packing, can also insert new code (e.g. unpacking stub) with new external calls; however as long as this code does not interfere with the original one, their new calls will ensue as disjoint components added to the original ECDG, without breaking our method. To be effective against ECDG itself, the obfuscation needs to actually replace external calls or alter their argument dependencies; this procedure is much more complex than syntactic obfuscation and is not observed in the wild.

Furthermore, although unfit for large datasets, our evaluation clustering is able to assess the efficiency, accuracy and robustness of σ^{ECDG}. This construction can be practical with clustering strategies like those proposed by [22] and [48], keeping the same properties verified here.

Table 2 shows σ^{ECDG} calculation throughout the whole dataset has median time in the order of milliseconds, and that inclusion of noise results in lower average time to compute similarity. Indeed, gSpan typically finishes very quickly when two (or more) input graphs do not have any common subgraph. Assuming that samples of a same family represent just a small part of an entire dataset, this behavior is suitable to address the search problem, which makes σ^{ECDG} a good practical alternative for this problem.

Our prototype analysis (section 5.4) shows many cluster prototypes are naturally produced, even when bigger support values are used. This is a positive consequence of using *subgraph isomorphism* as basis of σ^{ECDG}, which opens the possibility for optimization strategies like *scalable clustering* [31]. Furthermore, the cluster prototypes obtained are very descriptive (see figure 8), which allows assisting human analysis or training models for specific MW families.

Limitations. Our call extraction module extends *Angr* with 24052 new stub *SimProcedures* of commonly used functions, highly improving code coverage, hence ECDGs quality. However, our experiments have limits related to state explosion in symbolic executions [6]: the main reason for job termination (96%) during *code analysis* was memory exhaustion. Additionally, the limitations of symbolic execution to handle packed binaries are well known. To address this, we tested *Angr*'s option to enable self-modifying code support[7], but with no noticeable gain. Although several samples of the evaluation dataset are being flagged by packing detection tools, packed samples often resulted in the extraction of few calls of the unpacking routines (e.g. *LoadLibraryA* and *GetProcAddress*). Therefore, due to the 100 edges threshold (see section 5.1) and the use of syntactical signatures - that are likely to fail with packed samples - this case was not specifically evaluated in our work.

A possible mitigation for both issues is a smarter combination of *concrete* and *symbolic* execution (i.e. *concolic execution*) instead of pure symbolic execution, to bypass the unpacking routine of (simple) packers and to perform expensive analyses (i.e. CPU and memory intensive) only in a few parts of the code. To further improve the quality of call dependencies for edges in ECDGs, instead of plain comparison of (symbolic) values, future work also includes implementing a taint analysis, where arguments and returns of functions are more finely tracked.

The computation of our similarity/distance matrix for the evaluation clearly does not scale for very big datasets. So our *evaluation* dataset for this paper is restricted in size, but this does not affect our method itself nor its practical usability. Indeed, it is only used to evaluate the accuracy and robustness of σ^{ECDG}, as well as to produce benchmarks that assess its efficiency in practice. Overall, the problem of clustering scalability (i.e. a scheme that requires asymptotically less pairwise computations) is out of the scope of this work and an ongoing research topic. Nonetheless, such clustering can of course benefit from more efficient as well as accurate and robust pairwise similarity computations.

7 RELATED WORK

Our focus is on the representation of programs through *call graphs* (ECDGs), benefiting from their *structural similarities* (see section 2.2) to introduce our new similarity function σ^{ECDG}. For this reason, our work places itself closer to researches that study the similarity of binary codes based on structural similarities, especially when they involve some form of call graph.

The concept of call graphs dates back as early as 1979 [36], and as of 2004 [13] a plethora of studies targeting structural similarities for MW analysis have been made. Initially the focus was on the quality of disassemblers, graph formats (i.e. creating labels or grouping nodes) and graph matching algorithms, as feature extraction was done statically [23, 24, 39, 46]. Several *similarity functions* have been proposed and served as basis for code diffing tools, like *BinDiff* [49] and *radiff2* - used in this work as benchmark. The main drawback of these approaches is that they operate on hefty graph representations, that though detailed are very expensive to handle in practice - especially considering that graph matching

[7]Parameter that enables the emulation of code from the current state instead of the original memory, notwithstanding memory protections.

problems fall into the NP class. Furthermore, pure static analysis is more susceptible to syntactical mutations, which can impact the result of the whole similarity analysis.

As of 2007 [5], researches related to *dependency graphs* emerge as dynamic analysis gains traction on MW analysis domain. There too, different graph formats and graph similarity functions, based on various graph matching algorithms, have been proposed [10, 12, 16, 28, 30]. In this case, the main drawback is that the tracing of calls (and OS resources) is done through filters placed as kernel modules/drivers, which is unable to distinguish traces of interest, i.e. those generated by the program's main object, from spurious traces generated by shared modules (i.e. library code). This mixture in the traces muddles the subsequent analysis and, once again, produce bigger graph representations that are expensive to process.

As for *Symbolic execution* for MW analysis, it was introduced in 2008 with BinHunt [14], using *symbolic formulas* to represent basic blocks and *theorem provers* to verify whether they are equivalent or embody *semantic differences*. Most works using symbolic execution still follow this approach (e.g. iBinHunt [26]). BinSim [27], which uses symbolic execution to trace call arguments in order to verify whether system calls in two programs are aligned (i.e. conditionally equivalent), is the scheme closest to ours.

Unlike the above categories of approaches, our work leverages the fine control of *symbolic execution* over the whole execution state (i.e. registers and memory), thereby focusing the analysis solely on the code main object. This allows us to concentrate the information present in a large series of calls into a much smaller subset, thus providing smaller graphs that enable more efficient analysis without any loss of information.

8 CONCLUSION

We defined ECDG, a new call graph to optimize the representation of argument dependencies between calls, that allows more concise structural representation with no information loss. It has a major positive impact on performance, because graph matching algorithms, which are very expensive, can deal with much smaller graphs. We proposed a new similarity function σ^{ECDG} that is efficient - achieving a speedup gain of 3.30x and 354.11x for the standard and cache-enhanced implementations wrt. *radiff*. - as well as reliably *accurate* and *precise*. To support this, we built an evaluation framework to cluster samples with σ^{ECDG} and we evaluated it under the AnR paradigm.

The *accuracy phase* produced almost unerring results, with homogeneity score of 0.983, which shows that our evaluation framework manages to autonomously describe MW families as accurately as a set of strict handcrafted Yara rules. The *robustness phase* verified that the clustering was robust to noise insertion, having almost no impact on homogeneity and mean similarity, and only mild effect on completeness and silhouette scores due to creation of many singletons (see figure 4). Indeed, the the NMI score between the initial and final (highly polluted) clusterings of this phase was 0.974, indicating that the information loss due to noise insertion was only marginal. In addition, our evaluation framework produced a high rate of descriptive cluster prototypes that represent behaviors and can be used to scale up clustering, assist manual analysis or enhance classification models for MW detection.

As main contributions (i) we revisited the foundations of MW analysis research, defining basic MW analysis primitives, and proposing the *AnR paradigm* for reliable evaluation methodologies; (ii) we proposed ECDGs, a compact call dependency graph enabling more efficient binary similarity computation; and (iii) we proposed a new similarity function for ECDGs that is efficient, accurate and robust. Contributions also come from our concrete implementation, namely the study of symbolic execution to trace external calls, the evaluation of gSpan as a practical algorithm for sub-graph isomorphism, and the evaluation of cluster prototypes extraction to represent MW families. Ultimately, our experiments show σ^{ECDG} can reliably work as cornerstone of multiple types of frameworks, from those who autonomously produce descriptions of MW families as accurately as manually created syntactical signatures to those who target MW search and MW clustering.

ACKNOWLEDGMENTS

The authors want to thank Cisco for providing the samples for the MW and CW datasets and also for the stimulating discussions; Nicolas Rougier and Veronica Valeros for the remarkable insights, specially related to the composition of evaluation datasets using cleanware, which set the basis for the AnR paradigm.

REFERENCES

[1] [n. d.]. AV Test - malware statistics. https://www.av-test.org/en/statistics/malware/.
[2] [n. d.]. Yara - VirusTotal. https://virustotal.github.io/yara/.
[3] 2012. VirusTotal. https://www.virustotal.com.
[4] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. 2016. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*. 183–194.
[5] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 178–197.
[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[7] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering.. In *NDSS*, Vol. 9. Citeseer, 8–11.
[8] Rodrigo Rubira Branco. 2007. Ltrace internals. In *Proceedings of the Linux Symposium*, Vol. 1. Ottawa, ON, Canada, June, 41–52.
[9] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang. 2013. An information retrieval approach for malware classification based on Windows API calls. In *2013 International conference on machine learning and cybernetics*, Vol. 4. IEEE, 1678–1683.
[10] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the ESEC and the ACM SIGSOFT SFE (ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 5–14.
[11] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. 2005. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP '05)*. IEEE Computer Society, USA, 32–46.
[12] Ammar Elhadi, Mohd Maarof, Bazara Barry, and Hentabli Hamza. 2014. Enhancing the Detection of Metamorphic Malware using Call Graphs. *Computers & Security* 46 (10 2014), 62–78.
[13] Halvar Flake. 2004. Structural Comparison of Executable Objects. In *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*. 161–173.
[14] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 10th ICICS (ICICS '08)*. Springer-Verlag, Berlin, Heidelberg, 238–255.
[15] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. 2009. Automatic Generation of String Signatures for Malware Detection. In *Recent Advances in Intrusion Detection*, Engin Kirda, Somesh Jha, and Davide Balzarotti (Eds.).

Springer Berlin Heidelberg, Berlin, Heidelberg, 101–120.
[16] Ibai Gurrutxaga, Olatz Arbelaitz, Jesús M. Pérez, Javier Muguerza, José Ignacio Martín, and Iñigo Perona. 2008. Evaluation of Malware clustering based on its dynamic behaviour. In *Proceedings of the Seventh Australasian Data Mining Conference (AusDM 2008) (CRPIT)*, John F. Roddick, Jiuyong Li, Peter Christen, and Paul J. Kennedy (Eds.), Vol. 87. Australian Computer Society, 163–170.
[17] Irfan Ul Haq and Juan Caballero. 2019. A Survey of Binary Code Similarity. *CoRR* abs/1909.11424 (2019). arXiv:1909.11424 http://arxiv.org/abs/1909.11424
[18] Chuntao Jiang, Frans Coenen, and Michele Zito. 2004. A Survey of Frequent Subgraph Mining Algorithms.
[19] Chuntao Jiang, Frans Coenen, and Michele Zito. 2010. Finding Frequent Subgraphs in Longitudinal Social Network Data Using a Weighted Graph Mining Approach. In *Advanced Data Mining and Applications - 6th International Conference, ADMA 2010 (Lecture Notes in Computer Science)*, Vol. 6440. Springer, 405–416.
[20] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. 2015. A novel approach to detect malware based on API call sequence analysis. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 659101.
[21] Joris Kinable and Orestis Kostakis. 2011. Malware classification based on call graph clustering. *Journal in computer virology* 7, 4 (2011), 233–245.
[22] Orestis Kostakis. 2014. Classy: Fast Clustering Streams of Call-Graphs. *Data Mining and Knowledge Discovery* 28, 5–6 (Sept. 2014), 1554–1585.
[23] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2006. Polymorphic Worm Detection Using Structural Information of Executables. In *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 207–226.
[24] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. 2010. Detecting Metamorphic Malwares Using Code Graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. Association for Computing Machinery, New York, NY, USA, 1970–1977.
[25] Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter. 2010. On Challenges in Evaluating Malware Clustering. In *RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6307. Springer, 238–255.
[26] Jiang Ming, Meng Pan, and Debin Gao. 2013. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proc. of the 15th Int'l Conf. on Information Security and Cryptology*. 92–109.
[27] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. 253–270.
[28] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. 2015. Amal: High-fidelity, behavior-based automated malware analysis and classification. *computers & security* 52 (2015), 251–266.
[29] Stavros D. Nikolopoulos and Iosif Polenakis. 2017. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques* 13, 1 (Feb 2017), 29–46.
[30] Younghee Park, D.s Reeves, and Mark Stamp. 2013. Deriving common malware behavior through graph clustering. *Computers & Security* 39 (11 2013), 419–430.
[31] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 19, 4 (2011), 639–668.
[32] Andrew Rosenberg and Julia Hirschberg. 2007. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*. 410–420.
[33] Florian Roth. 2013. HyarGen. https://github.com/Neo23x0/yarGen
[34] Florian Roth. 2015. How to Write Simple but Sound Yara Rules. https://www.nextron-systems.com/2015/02/16/write-simple-sound-yara-rules/
[35] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.
[36] Barbara G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Trans. Softw. Eng.* 5, 3 (May 1979), 216–226.
[37] Aleieldin Salem, Sebastian Banescu, and Alexander Pretschner. 2020. Maat: Automatically Analyzing VirusTotal for Accurate Labeling and Effective Malware Detection. *arXiv preprint arXiv:2007.00510* (2020).
[38] A. A. Selçuk, F. Orhan, and B. Batur. 2017. Undecidable problems in malware analysis. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. 494–497.
[39] Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. 2010. Detecting malware variants via function-call graph similarity. 113 – 120.
[40] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[41] Steven Skiena. 1990. *Implementing discrete mathematics - combinatorics and graph theory with Mathematica.* Addison-Wesley. I–VIII pages.
[42] Lars Sthle and Svante Wold. 1989. Analysis of variance (ANOVA). *Chemometrics and Intelligent Laboratory Systems* 6, 4 (1989), 259–272.
[43] Alexander Strehl and Joydeep Ghosh. 2003. Cluster Ensembles — a Knowledge Reuse Framework for Combining Multiple Partitions. 3 (2003), 583–617. https://doi.org/10.1162/153244303321897735
[44] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to data mining*. Pearson Education India.
[45] Radare Team. 2020. Radare2 github repository. https://github.com/radareorg/radare2
[46] Lingfei Wu, Ming Xu, Jian Xu, Ning Zheng, and Haiping Zhang. 2013. A novel malware variants detection method based On function-call graph. 1–5.
[47] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining.. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*. IEEE Computer Society, Maebashi City, Japan, 721–724.
[48] Y. Zhang, C. Rong, Q. Huang, Y. Wu, Z. Yang, and J. Jiang. 2017. Based on Multi-features and Clustering Ensemble Method for Automatic Malware Categorization. In *2017 IEEE Trustcom/BigDataSE/ICESS*. 73–82.
[49] zynamics. [n. d.]. BinDiff. https://www.zynamics.com/bindiff.html.

Table 5 *Robustness phase* clusters: initial (*group II*) and final (*group II + group III*)

Cluster index		Initial				Final			
Initial	Final	#samples	Similarity	*Yara rule*	#samples (per rule)	#samples	Similarity	*Yara rule*	#samples (per rule)
0	0	9	0.978	ClamAV_Emotet_String_Aggregate	9	9	0.978	ClamAV_Emotet_String_Aggregate	9
4	1	10	0.951	spyeye Glasses	2 8	10	0.951	spyeye Glasses	2 8
3	2	158	0.987	Wabot	158	158	0.987	Wabot	158
-	3					5	0.98	ClamAV_Emotet_String_Aggregate spyeye Warp	1 2 2
12	4	7	1.000	spyeye	7	7	1.000	spyeye	7
13	5	19	0.993	spyeye	19	19	0.993	spyeye	19
14	6	18	0.998	spyeye	18	18	0.998	spyeye	18
9	7	23	0.995	spyeye	23	23	0.995	spyeye	23
6	8	11	0.819	spyeye	11	14	0.605	spyeye cleanware	11 3
10	9	15	0.993	spyeye	15	15	0.993	spyeye	15
11	10	6	1.000	spyeye	6	6	1.000	spyeye	6
8	11	59	0.992	ClamAV_Emotet_String_Aggregate Njrat njrat1 win_exe_njRAT	1 58 58 58	59	0.992	ClamAV_Emotet_String_Aggregate Njrat njrat1 win_exe_njRAT	1 58 58 58
2	12	10	0.995	IceID_Bank_trojan	10	10	0.995	IceID_Bank_trojan	10
-	13					24	0.997	Glasses IceID_Bank_trojan cleanware	1 2 21
1	14	187	0.997	Warp Wabot Glasses IceID_Bank_trojan sakula_v1_3 Bublik	1 4 1 26 80 88	193	0.998	Warp Wabot cleanware IceID_Bank_trojan sakula_v1_3 Bublik	1 4 10 23 80 88
7	15	128	0.994	ClamAV_Emotet_String_Aggregate Glasses Mirage_APT Cerberus spyeye	5 32 41 49 1	127	0.998	ClamAV_Emotet_String_Aggregate Glasses Mirage_APT Cerberus	5 32 41 49
-	16					5	0.934	cleanware	5
-	17					24	0.998	cleanware	24
-	18					34	0.990	cleanware	34
5	19	19	0.877	ClamAV_Emotet_String_Aggregate Glasses spyeye	5 2 12	55	0.516	ClamAV_Emotet_String_Aggregate Glasses spyeye cleanware	1 2 12 40
19	20	7	0.983	ClamAV_Emotet_String_Aggregate Monero_Mining_Detection	1 6	16	0.988	ClamAV_Emotet_String_Aggregate Monero_Mining_Detection cleanware	1 7 8
18	21	5	1.000	Monero_Mining_Detection	5	6	0.998	Monero_Mining_Detection	6
-	22					10	0.959	Warp cleanware	2 8
-	23					8	0.985	ClamAV_Emotet_String_Aggregate cleanware	3 5
-	24					12	0.867	IceID_Bank_trojan cleanware	1 11
-	25					10	0.927	IceID_Bank_trojan cleanware	1 9
17	26	133	0.995	IceID_Bank_trojan shylock	26 109	133	0.995	IceID_Bank_trojan shylock	26 109
15	27	96	0.935	spyeye Monero_Mining_Detection IceID_Bank_trojan	1 11 84	108		spyeye Monero_Mining_Detection IceID_Bank_trojan cleanware	1 11 83 13
16	28	27	0.995	Glasses spyeye ClamAV_Emotet_String_Aggregate	1 25 1	29	0.997	Glasses spyeye cleanware	1 25 3
-	29					13	0.954	cleanware	13
noise				54				340	