IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 31, 2018, accepted September 21, 2018, date of publication October 8, 2018, date of current version October 31, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2874502

Countering Android Malware: A Scalable Semi-Supervised Approach for Family-Signature Generation

ANDREA ATZENI[1], FERNANDO DÍAZ[2], ANDREA MARCELLI[1], ANTONIO SÁNCHEZ[2], GIOVANNI SQUILLERO[1], (Senior Member, IEEE), AND ALBERTO TONDA[3]

[1]Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, 10129 Turin, Italy
[2]Hispasec Sistemas S.L., 29001 Málaga, Spain
[3]INRA, UMR 782 GMPA, 78850 Thiverval-Grignon, France

Corresponding author: Andrea Marcelli (andrea.marcelli@polito.it)

The work (Ph.D. program at Politecnico di Torino) of A. Marcelli was supported by the fellowship through the TIM (Telecom Italia Group).

ABSTRACT Reducing the effort required by humans in countering malware is of utmost practical value. We describe a scalable, semi-supervised framework to dig into massive data sets of Android applications and identify new malware families. Until 2010, the industrial standard for the detection of malicious applications has been mainly based on signatures; as each tiny alteration in malware makes them ineffective, new signatures are frequently created –- a task that requires a considerable amount of time and resources from skilled experts. The framework we propose is able to automatically cluster applications in families and suggest formal rules for identifying them with 100% recall and quite high precision. The families are used either to safely extend experts' knowledge on new samples or to reduce the number of applications requiring thorough analyses. We demonstrated the effectiveness and the scalability of the approach running experiments on a database of 1.5 million Android applications. In 2018, the framework has been successfully deployed on Koodous, a collaborative anti-malware platform.

INDEX TERMS Semi-supervised learning, clustering, android, malware, automatic signature generation.

I. INTRODUCTION

Android's first malware, *FakePlayer*, was released in August 2010 [1] and, since then, the number of new malware steadily increases [2]. After only seven years, malware programs are hundreds of times bigger than the old *FakePlayer*, hide their presence and activities, and they can even communicate secretly through complex anonymous networks.

Android offers an open market model, where millions of applications are downloaded by users every day. While applications from the official Google Play store undergo a review process to confirm that they comply with Google policies [3] other third-party markets do not. Hence, a typical pattern among malware developers is to repack popular applications from Google Play by adding malicious features and distribute them to third-party app-stores, leveraging apps popularity to accelerate malware propagation.

In the personal-computer ecosystem, malware developers commonly exploit executable packing and other code obfuscation techniques to generate a large number of polymorphic variants of the same malicious application [4], [5]. As a consequence antivirus (AV) software are struggling to keep their signature database up-to-date, and AV scanners suffer from a considerable quantity of false negatives [6]. Moreover, the malicious code is often reused and customized to fit different needs. For example, a developer may reuse the rootkit installation code, while replacing the modules that provide network connectivity to a Command-and-Control server.

By the end of 2010s, the Android ecosystem is facing a similar scenario, although the situation is exacerbated by the simplicity of malicious *repackaging* [7]. That is an alteration of the original application installation package (i.e., the `APK` file), where legitimate applications are reverse engineered, modified to include malicious code, signed with a new signature, and eventually distributed for download. Since applications consist of bytecode, changes are relatively easy to implement and ad-hoc tools assist the procedure [8], [9].

The growth of Android malware created a major challenge for AV vendors to efficiently handle new samples and accurately label them. Due to the practical impossibility of manually analyzing the thousands of suspicious samples received every day, a large fraction is left unlabeled, delaying the signature generation.

While malware variants can be generated at a high pace, they are likely to perform similar malicious activities when executed. One possible solution would be to automatically cluster such applications in a family and focus the manual analysis on few archetypal samples, with the underlying assumption that malware bearing significant similarities are likely to derive from the same code base [10]. Furthermore, the label of a new sample of a known family could be automatically derived, and existing signatures or other mitigation techniques could be more easily extended to cover the new threats.

Eventually, if a large number of malware belonging to the same family is identified, it may become possible to define a generic behavioral signature able to detect future variants with reduced false positives and false negatives [11]. Therefore, a sharp clustering is crucial to help AV companies categorizing the large amount of samples, avoiding duplicate work, and allowing analysts to prioritize their limited resources on novel and representative samples [12], [13].

In this article, we describe a semi-supervised system for the analysis of massive datasets of malicious applications. We created a platform able to suggest new families of applications to human experts; the platform also generates an intelligible YARA rule [14] to identify family members with high precision. We explicitly minimize false positives, a business hazard and a reputation blow for AV vendors. The approach alleviates human experts from the burden of manually inspecting thousands of Android applications, while letting them take critical decisions. The main contributions of this article can be summarized as:

- We introduce a scalable system for the analysis of massive Android malware datasets based on careful feature engineering, and a standard clustering algorithm. The mechanism is demonstrated to be robust and able to overcome the well-known limitations of traditional signature-matching mechanisms.
- We propose an algorithm that, starting from a cluster of samples, generates its *family signature* as a YARA rule. Thanks to exact and heuristic evaluations, such rules are *intelligible* and appear *reasonable* to human experts. Moreover, the algorithm guarantees zero false positives in the existing dataset, and limits the possibility of false positives in the future.
- We report experiments on a dataset of about 1.5 million Android applications, and results show the scalability of the approach. We use a set of *internal* and *external* indicators to demonstrate that the proposed system performs an accurate and efficient automatic identification of groups of similar applications. By exploiting limited data, the framework is able to propose insightful extensions to the rule detecting suspicious applications.
- Finally, our framework has been deployed and it is used on Koodous,[1] the mobile AV platform from Hispasec, since the January 2018.

[1] https://koodous.com/

The rest of the paper is organized as follows: Section II illustrates problem statement and motivation, and Section III introduces Koodous; Section IV describes in detail the proposed approach, while experimental results and performance evaluations are presented in Section V; Section VI surveys related work about Android malware and automated analysis procedures; limitations and future works are discussed in Section VII and VIII; Section IX concludes the paper.

II. PROBLEM STATEMENT AND MOTIVATION

Since the 2000s, academia proposed approaches based on machine learning aiming at completely replacing humans in the malware analysis process. In most cases, such proposals fell back into mere classification, that is, *supervised* machine learning. The drawbacks included the need of large amount of accurately labeled, i.e., already analyzed, data, and the lack of control over the false positives eventually produced, a major cause of concern for all AV vendors. As a result, AV companies developed systems mostly based on the reliable signature-detection mechanism. Even though signatures suffer from the so-called ''specificity'' problem, and new ones need to be frequently generated, they have been demonstrated effective, scalable, and almost unaffected by false positives.

The proposed framework is *semi-supervised* and introduces essential improvements in the identification of similar applications and the generation of family signatures. It combines the scalability of fully automatic techniques for clustering and the optimization of new family signatures, while it exploits manual analysis, inherently more flexible and accurate, in few crucial steps, such as the validation of newly discovered malware families.

Traditionally, the effort of automatically classifying and analyzing malware focuses on *content-based* signatures that specify binary sequences. Indeed, content-based signatures are inherently vulnerable to malware obfuscation: even if all variants of a malicious application share the same functionalities and exhibit the same behavior, they can have tiny different syntactic representations. As a consequence, a huge number of signatures needs to be created and distributed by AV companies.

On the other hand, a rule that automatically identifies the behavior of a family of samples would be the first step towards the creation of true *family signatures*. Such a signature would match all samples of a family, and would significantly help to reduce the number of signatures required to cover it. Moreover, as new samples could be mapped to a family behavior already known, the time and effort required to analyze and reverse engineer new samples would be reduced.

Differently from the previous approaches, the proposed system generates effective, precise and descriptive rules using the properties directly extracted from both static and dynamic analyses. While aiming at reducing false positives and false negatives, it also exploits an heuristic measure to emulate how expert analysts write existing signatures.

III. KOODOUS

Koodous is a collaborative platform for researching on Android malware that combines online analysis tools with social interactions between the analysts. Started in 2014, in 4 years it collected one of the largest repositories of Android applications: its databases contain more than 30 millions of applications, among which 7 millions have already been identified as malicious. Fig. 1 illustrates the trend of application submission and detection from October 2014, until March 2017.

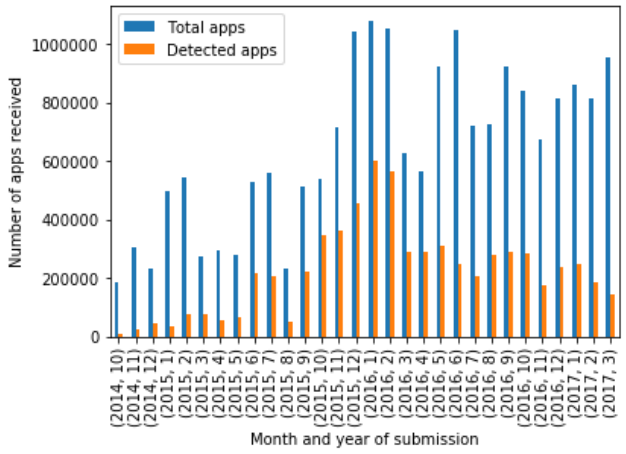


FIGURE 1. Monthly trend of application submission and detection in Koodous from October 2014 until March 2017.

Koodous provides both analysis service and end-point protection: upon submission, each application is analyzed both statically and dynamically, and the final report is accessible through a web interface specifically designed to help analysts detect new malware threats. Analysis tools include a custom version of Androguard [15], CuckooDroid[2] and DroidBox [16].

Instead of relying on a closed group of expert malware analysts, Koodous takes advantage of an open community to identify malicious applications. Furthermore, in order to guarantee high quality results, manual detections are subject to reputation-based checking. Moreover, protection is guaranteed through an Android application, which backs to the cloud platform to detect most recent threats.[3]

Koodous uses YARA to describe patterns for detecting malware application: since the creation of high-quality YARA rules requires a considerable effort, the platform also offer the possibility to identify malware through a simpler voting mechanism –- an operation referred to as "triage." As of July 1, 2018, more than 2.5 millions applications are detected by triage.[4]

[2]https://github.com/idanr1986/cuckoo-droid

[3]https://play.google.com/store/apps/details?id=com.koodous.android

[4]For the up-to-date figure, visit `https://koodous.com/apks?search=rating:%3C-1%20%26%20detected:1`.

IV. PROPOSED FRAMEWORK

Our framework operates through three main steps, detailed in sections IV-A, IV-B and IV-C.

1) Similarities among Android samples are discovered through an iterative clustering process, offering a new point of view and valuable information to malware analysts.
2) Families of suspicious applications are identified taking advantage of the knowledge already available in Koodous, and extensions to the current detection rules are proposed.
3) Signatures are generated to identify the malware families with an acceptable generalization capability, yet a reduced risk of false positives in the future.

A. ITERATIVE CLUSTERING

Clustering provides a mechanism to automatically categorize applications into groups that reflect their similarity, both in source code and runtime behavior. We exploit *HDBSCAN*, a density-based algorithm, as it fits most of our requirements.

Density-based clustering algorithms locate high-density regions in the feature space; DBSCAN (density-based spatial clustering of applications with noise) is probably the best known among them [17]. Density-based algorithms can effectively discover clusters of arbitrary shape and filter out outliers, eventually increasing cluster homogeneity. Additionally, the number of expected clusters to be found in the data is not required: our aim is to discover groups of similar applications without any prior knowledge about their composition, otherwise the number of clusters is hard to guess a priori.

In 2013, Campello *et al.* [18] proposed HDBSCAN, a new density-based algorithm that converts the original DBSCAN into a hierarchical clustering algorithm. As a matter of fact, HDBSCAN find clusters of varying densities, and is more robust to parameter selection. Moreover, it supports the GLOSH (global-local outlier score from hierarchies) outlier detection algorithm: during the fitting phase, each data point is associated to a score that represents its likelihood of being an outlier; at the end of the process, outliers are selected via upper quantiles [19].

In low-dimensional spaces, HDBSCAN has an average complexity of approximately $\mathcal{O}(n \log n)$, while its space requirement is $\mathcal{O}(n)$, making it applicable to moderately large datasets [20].

As the number of samples in malware datasets is in the tens of millions, we exploit an *iterative* process where the original dataset D is divided into m chunks d_i of fixed size N ($m = \left\lceil \frac{|D|}{N} \right\rceil$).

$$D = \bigcup_{i=0}^{m-1} d_i \tag{1}$$

The parameter N balances the quality of the results with the time required for the analysis, and can be set experimentally according to the available resources.

HDBSCAN is applied to each chunk of data d_i finding, at each step, a set of clusters c_i and a set of outliers o_i. Finally, all the outliers $O = \bigcup_{i=0}^{m-1} o_i$, are clustered together in order to find even those small groups of applications whose samples are spread through several chunks of data. In the end, the total number of required iterations is equal to $m + 1$.

Since HDBSCAN could be executed in parallel on the first m chunks, the benefit of the iterative approach is the huge reduction in the time required for the analysis. On the other hand, few applications could be misclassified as outliers and the same group of similar applications could be found multiple times, although, as shown in section IV-B, those corner cases do not limit the framework efficacy.

1) FEATURES SELECTION

An accurate features selection is a crucial step in every machine learning approach. As suggested in [11], we exploit aggregate information: from the analysis result of each application, we extract a subset of ''statistical'' properties, meant as quantitative measure of a malware behavior. Indeed, we experimentally found that exploiting statistical similarities among applications, rather than ''structural'' properties which exactly describe the malicious behavior, does not effectively alter the results, while at the same time, significantly reduces the amount of data to process.

Starting from a set of n analysis reports r_i provided by Koodous, each report r_i is translated into a feature vector $v_i = (f_0, \ldots, f_{34})$ containing the 35 statistical properties.

Table 1 summarizes the features extracted from the results of the static and dynamic analysis.

In more detail, the static analysis performed by Androguard extracts the features from the Manifest file (i.e., number of activities, permissions, receivers, filters), and the source code analysis. The former allows to unveil similarities among applications based on the software architecture used to develop the application, while the latter models each application extracting portions of code related to suspicious API call (e.g., number of calls to SMS API, or IMEI, or other network related methods). On the other hand, the dynamic analysis extracts features that model the application interaction with the surrounding operating system both at file system and network level extracted by DroidBox (e.g., files written, usage of cryptographic methods, SMS sent), and the network information extracted by CuckooDroid (e.g., number of DNS resolved, HTTP requests). These are the standard type of information extracted in the field of malware analysis.

Because the range of each feature is quite different, the dataset is firstly normalized so that the features have mean equal to zero and variance equal to one. Since the choice of the distance to use during cluster analysis is tied to the type and the dimension of selected features, we experimentally found that the combination with the Euclidean distance delivered the best performances.

B. EXTENDING MALWARE DETECTION

Starting from millions of samples, the iterative clustering (Section IV-A) identifies a smaller number of *families* composed of strongly related applications. In some cases, by combining this result with the information already available in Koodous, these families may be automatically labeled, as they extend either known threats or legitimate software. In the other cases, experts are required to manually evaluate the family, but they need to analyze only few representative samples of the group and not all applications, therefore drastically reducing the time required by the analysis. This process exploits the ''clustering assumption'' of the semi-supervised learning algorithms, which states that two points which are in the same cluster (i.e., which are linked by a high density path) are likely to share same label. In such a way, the partial information of few labels extracted from each cluster can be used to increase the knowledge of all the applications within the same group.

The set of all applications in Koodous $\mathbf{K}$ may be partitioned into three subsets $\mathbf{K} = \{\mathbf{S} \cup \mathbf{T} \cup \mathbf{U}\}$ corresponding to the applications detected by signatures ($\mathbf{S}$), detected by triage only ($\mathbf{T}$), and undetected ($\mathbf{U}$); applications detected both by

TABLE 1. List of the 35 statistical properties extracted from the analysis result of each APK file. Features are grouped according to the type of analysis. Static features are extracted using Androguard both parsing the Manifest file and looking for interesting API calls in the decompiled source code. Dynamic features are extracted using DroidBox and CuckooDroid from the dynamic analysis of the application.

Analysis method	Sofware	Statistical property
Parsing Manifest file	Androguard	Filters Activities Receivers Services Permissions
Statically from APK	Androguard	Accounts Advertisement Browser history Camera Crypto functions Dynamic broadcast receiver Installed applications Run binary MCC ICCID IMEI IMSI SMS MMS Phone call Phone number Sensor Serial number Socket SSL
Dynamically	DroidBox	Files written Crypto usage Files read Send SMS Send network Recv Network
	CuckooDroid	HTTP request Hosts Domains DNS

signatures and in the triage phase belong to the **S** set. Such a partition does not reflect a peculiarity of Koodous, as the usage of a *staging area* **T** where samples are pointed out waiting further analysis is common in AV laboratories.

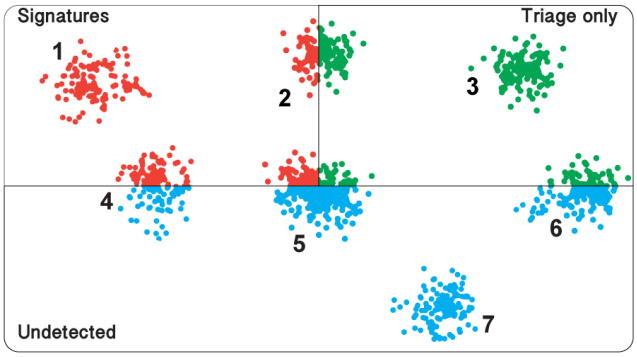


FIGURE 2. The figure illustrates the subdivision of the applications in database and the seven type of families (i.e., clusters) that can be automatically inferred by the proposed approach. The database is divided in three macro areas according to the type of detection: applications detected by signatures, by triage only, and undetected. Each point in the figure represents an application, and based on the detection status of the applications within each cluster, the proposed approach identifies seven possible cases.

It is possible to classify a family according to the different subsets its applications belong to (Fig. 2). The resulting seven different types of family correspond to the powerset P(**K**), excluding the empty set: { {**S**}, {**T**}, {**U**}, {**S**, **T**}, {**S**, **U**}, {**T**, **U**}, {**S**, **T**, **U**} }

- **Type 1** $\{\forall s \in F^{(1)} \mid s \in S\}$. The family is composed of applications that have been already detected by YARA signatures. No further action is required, although the generated family rule may still be effective to generalize the detection.
- **Type 2** $\{\forall s \in F^{(2)} \mid s \in S \cup T\}$. The family includes applications already identified as malicious either by YARA, or during the triage process. The correctness of the detection is either guaranteed by the existing signatures, or by the triage process (i.e., the community votes); thus a new YARA rule matching all the applications in the family can be automatically generated and added to the detection system without further manual check.
- **Type 3** $\{\forall s \in F^{(3)} \mid s \in T\}$. The family is composed of applications that have been detected through the triage process only. The correctness of the detection is guaranteed by the triage process, and as in the previous case, a new YARA rule can be automatically generated and added to the detection system without manual intervention.
- **Type 4** $\{\forall s \in F^{(4)} \mid s \in S \cup U\}$. The family combines applications detected by existing signatures with undetected ones. In order to avoid false positives, the correctness of the family must be manually validated before generating a family signature.
- **Type 5** $\{\forall s \in F^{(5)} \mid s \in S \cup T \cup U\}$. The family combines applications detected both by signatures and by triage only, with undetected ones. As in the previous case, in order to guarantee complete correctness the family must be manually validated before generating a signature.
- **Type 6** $\{\forall s \in F^{(6)} \mid s \in T \cup U\}$. The family combines applications either detected by the triage process only with undetected ones. As in the two previous cases, the family must be manually validated before generating a family signature.
- **Type 7** $\{\forall s \in F^{(7)} \mid s \in U\}$. The family is composed of undetected applications, hence no classification can be automatically inferred. However, as all the applications within the cluster show strong similarities, the analysis of few representative samples shall be sufficient to classify the whole cluster as malware or goodware.

Such an approach offers apparent benefits: the need for human intervention is often limited to the simple validation of the discovered family, while the need for full analysis is reduced to few representative samples. The identification of families with only partially detected applications, either by signature or during the triage process, allows to discover false negative and new 0-day malware.

In Koodous, the triage process makes it possible to quickly identify threats without the burden of creating signatures, although it has the drawback of potentially leaving others similar applications undetected. Our frameworks may automatically convert all the knowledge about single, unrelated threats into more reliable signature, potentially able to discover newer variants as well.

Finally, among **Type 7** families, the system is able to identify groups of legitimate software, for example finding applications written by the same developer or using the same framework. This result was proved to be of practical importance to limit and correct false positive detections.

C. FAMILY SIGNATURES GENERATION

In the last step of our framework, a signature is generated for each family that has been identified as malicious.[5] We developed an automatic procedure that starts from a set of applications, and eventually produces a YARA rule describing them. The program has no requirements on the origin of the set: it could be the result of automatic clustering or manual selection, although the more the applications in the set are related, the better the result.

Authoring an effective signature requires a considerable effort and experience. Good signatures are compact, and they have the ability to generalize, that is, to identify all known variants of the malware and even possible new ones. Moreover, they do not yield false-positive results by detecting non-family members, and finally, they appear intelligible to human experts and are almost self-explanatory.

[5]Our system could generate family signatures of legitimate applications as well, but they would be of no use for Koodous

More formally, a signature S is the disjunction of n clauses $S = \bigvee_{i=0}^{n} c_i$. A clause c_i is a finite conjunction of m_i literals $c_i = \bigwedge_{k=0}^{m_i} l_{k_i}$. In the present context, a literal l_{k_i} is a single feature specified in the report resulting from the analysis of the application.

Traditionally, a YARA rule is defined on unique strings found in malware but not present in legitimate programs; quite differently, we generate precise, descriptive rules using the structural properties extracted by the static and dynamic analyses. Our program identifies an optimal set of clauses for matching all target applications while yielding no false positive in the current database; moreover, thanks to some heuristics, the rule has a good ability to generalize, a low risk of detecting false positives in the future, and it appears *reasonable* to the eye of the human experts.

An example of an automatically generated YARA rule for the *Syringe* malware family is shown below.[6]. It may be noted that the statistical features exploited during clustering (Section IV-A) are not used in the in the rule, as they would result in over-complicated rules hardly understandable by humans.

```
rule YaYaSyringe {
condition:
androguard.filter(''action.BATTERYCHECK'')
and androguard.number_of_services == 3
and androguard.permission(''SYSTEM_ALERT_WINDOW'')
and androguard.url(''http://s.adslinkup.com/v2'')
...
}
```

The process is performed in three steps: a reasonable signature composed of a small number of clauses is generated; the signature is checked against the full database of applications, and false positives are identified; the generation procedure is run again, but explicitly taking into consideration the false positives discovered in the second step.

Fig. 3 exemplifies the idea of the process of generation of a signature for two malware m_a and m_b, and two legitimate applications g_a and g_b. In the first phase, the algorithm defines a signature $Y = r$, where r is a single clause composed by the common features between the two malware: $r = m_a \cap m_b$. Indeed, a rule Y detects an application m only if Y is a subset of m: $Y \subseteq m$.

During the second phase, the rule Y is checked against the complete database, where it generates two false positives matching two legitimate applications g_a and g_b. The clause r is therefore *too generic* to be used as a signature.

As it is not possible to find features common to malware that do not matches legitimate applications $(m_a \cap m_b) \setminus g_a = \varnothing$ and $(m_a \cap m_b) \setminus g_b = \varnothing$, the third step generates a signature with the disjunction of two clauses $Y^* = (r \wedge r_a) \vee (r \wedge r_b)$.

The pseudocode of the algorithm is reported in Algorithm 1: at first it determines a suitable set of clauses (function *Clauses*), then picks a subset of them of variable size to build an optimal family signature (function *Clot*).

[6]The complete version of the rules is available on Koodous at https://koodous.com/rulesets/3243

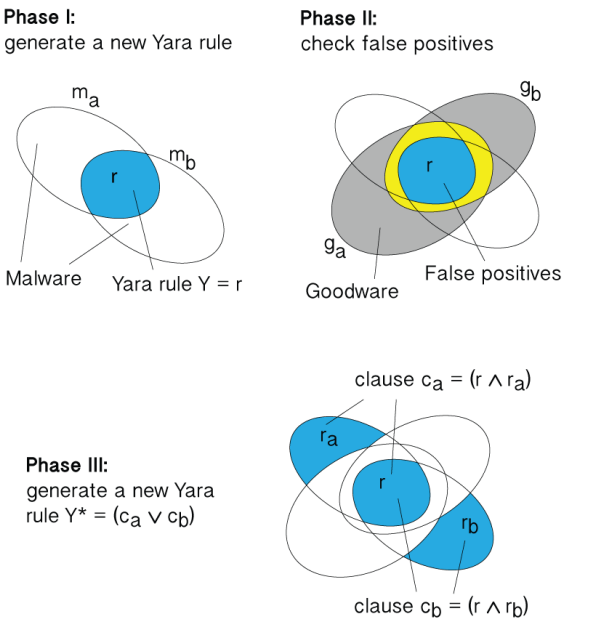


FIGURE 3. Schema of the process of generation of a YARA rule. In the first phase a signature $Y = r$ is defined for malware m_a and m_b. In the second phase Y is checked against a dataset of goodware (g_a and g_b). Finally, in the third phase, a new signatures $Y^* = (r \wedge r_a) \vee (r \wedge r_b)$ is created to avoid false positive detection of g_a and g_b.

Algorithm 1 Automatic YARA Rule Generation

```
1: procedure generateSignature(R)
2:     C ← Clauses(R, ∅)
3:     Y ← Clot(R, C)
4:     G ← GetFalsePositives(Y)
5:     C* ← Clauses(R, G)
6:     Y* ← Clot(R, C*)
7:     DumpAsYARARule(Y*)
```

Lines 2 and 3 correspond to the first phase of Fig. 3; line 4, to the second; lines 5 and 6, to the third.

Algorithms 2 and 3 add more details about the procedure: the function *Clauses* extracts the clauses that can be used to build the signature, and is based on a heuristic algorithm. First, each malware application r_i in the target set R is transformed into a single clause y_i able to detect it using all available literals. Such clauses are not directly usable, but are the starting point of the interactive procedure for building the set of optimal clauses H: in each step, the least generic y_i is selected and compared against all clauses in H calculating the common features z_i; the least generic of these z_i is eventually considered for inclusion in H.

The rationale is to build Y by adding clauses progressively less specific (i.e., checking fewer features), but still usable in signatures. Line 10 computes the set F of application from G detected by the candidate clause; as G is the set of all potential false positives, if F is not null the clause is too generic to be usable. Additionally, the function Quality(·) performs a heuristic evaluation of the clause: if the quality

Algorithm 2 Clauses Extraction

```
function Clauses(R, G)
    Y ← {Features(r) ∀r ∈ R}
    for all r ∈ R do
        Y ← Y ∪ SelectedClauses(r)
    H ← {Features(r) ∀r ∈ R}
    while |H| > 0 do
        h ← LeastGeneric(H)
        Z ← {CommonFeatures(h, y) ∀y ∈ Y}
        z ← LeastGeneric(Z)
        F = {r ∈ G | Det(z, r) = True}
        if F = ∅ and z ∉ Y and Quality(z) > T_q then
            H ← H ∪ {z}
            Y ← Y ∪ {z}
        H ← H \ {h}
    return Y
```

Algorithm 3 Clauses Selection

```
procedure Clot(R, C)
    Y ← ∅
    D ← ∅
    while R ≠ C do
        if ∃r ∈ R \ D : Critical(r) = True then
            r̄ ← GetCritical(R \ D)
            Z = {z ∈ C | Det(z, r̄) = True}
        else
            Z = {z ∈ C | ∃r ∈ R \ D : Det(z, r) = True}
        Y ← Y ∪ {MostUseful(Z)}
        D ← {r ∈ R | ∄y ∈ Y : Det(y, r) = True}
    return Y
```

is below a certain threshold T_q, the rule is so generic that it is likely to create false positives in a near future — see IV-C.1 for more details. For each application, few *not-too-generic*, heuristically selected clauses are also included (i.e., r_a and r_b in the example shown in 3).

The function *Clot* (Algorithm 3) implements a dynamic greedy algorithm for building the signature as a disjunction of clauses. It iteratively adds one clause to Y from a set C until all applications in R are detected by at least on clause in Y.

In an iterative way, *Clot* first picks out all clauses that detect at least an application not yet detected by any rule, with the only exception that, if an application can be detected by only one clause, that clause is the only one picked. Then the algorithm selects among this group the clause that is able to detect more application in the original target set R.

1) RULE QUALITY

A heuristic evaluation is used to reduce the risk of false positives in the future and to increase the perceived quality of the rule. We defined a heuristic score $\mathcal{S}(\cdot)$ for a rule, inversely related to its generality. More formally, let associate each literal l to a score $\mathcal{S}^*(l)$ that measures how specific the literal is. The score of a clause c_i is the sum of the scores of the n_i literals composing it: $\mathcal{S}(c_i) = \sum_{k=0}^{n_i} \mathcal{S}^*(l_{i_k})$. The sore of a rule r is the minimum among the scores of its clauses: $\mathcal{S}(r) = \min_{\forall i} \mathcal{S}(c_i)$.

The higher the score, the more a rule is specific and less susceptible to generate false positives. On the other hand, the lower the score, the more a rule will be able to generalize, while being more prone to generate false positive in the future. High quality signatures require an optimal balance between generality and specificity, and this is one of the main challenges in automatic signature generation. We use two threshold $T_{\min}$ and $T_{\max}$, where the lowest is the minimum score that a rule needs to be valid, and the highest is used in the optimization process to avoid overly-specific rulesets.

All the clauses in YARA rules created by expert analysts are valid, that is, the score assigned to literals must guarantee that $\forall r \in \mathbf{R}_{\text{expert}} : T_{\min} \leq \mathcal{S}(r) \leq T_{\max}$. We consider invalid the rules containing a clause mentioning only Android official permissions and intent filters, or containing a clause composed of a single literal, with the exception of accessing an URL that have been detected as malicious by VirusTotal or similar services. Then, we exploit the simplex method as a mean to automatically define $\mathcal{S}^*(\cdot)$ starting from the existing ruleset.

The simplex method is a linear programming technique, which refers to the problem of optimizing a linear *objective function* ζ of m variables x_i subject to a set of n linear inequality constraints. In *standard form*, the problem of finding an optimal set of weights for m literals can be expressed as:

$$\min \zeta = \mathbf{c}^{\mathrm{T}} \times \mathbf{x}$$
$$\text{s.t} \ -\mathbf{A} \times \mathbf{x} \geq -\mathbf{b}, \ \mathbf{x} \geq 0$$

where $c_i = 1, \forall i = 1 \ldots m$, since the objective function ζ minimize the number of literals in each clause, $\mathbf{x} \in R^m$ is a vector of m unknown weights, and $b_i = T_{min}, \forall i = 1 \ldots n$, as we want each existing literal combination to satisfy the minimum score of all existing rulesets.

Finally $\mathbf{A}$ is a $n \times m$ matrix that put into relation each clause with their own literals:

$$\mathbf{A} = \begin{bmatrix} l_{11} & l_{12} & l_{13} & \ldots & l_{1m} \\ l_{21} & l_{22} & l_{23} & \ldots & l_{2m} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ l_{n1} & l_{n2} & l_{n3} & \ldots & l_{nm} \end{bmatrix}$$

where $l_{nm} = 1$ if l_{nm} is a literal of the clause c_n, otherwise $l_{nm} = 0$. In order to get the list of all the n existing clauses c_i, we firstly reduced all the available YARA ruleset in the Disjunctive normal form (DNF).

We set almost arbitrarily the values $T_{min} = 400$ and $T_{max} = 700$ for the two thresholds. Table 2 reports the details about the rules, clauses, and unique clauses that have been analyzed, using the YARA rules from both Koodous[7] and the Yara-Rules repository on GitHub.[8] Table 3 show the final result, where each literal is assigned a distinct weight.

[7] https://koodous.com/rulesets
[8] https://github.com/Yara-Rules/rules/tree/master/Mobile_Malware

TABLE 2. Details about the number of rules, clauses, and unique clauses analyzed to find the optimal score for each literal.

	Num. of YARA rules	Num. of DNF clauses	
		Total	Unique
Koodous public rules	348	788	104
Yara-Rules on GitHub	348	697	48

TABLE 3. Weights assigned to each type of literal as a result of the simplex method optimization. Weights are used by the automatic procedure to generate new YARA rulesets.

Module Name	Literal type	$\mathcal{S}^*(\cdot)$
Androguard	App name	100
	Package Name	100
	Certificate SHA1	150
	Certificate Subject	100
	Certificate Issuer	100
	Main Activity	50
	Activity	150
	Service	150
	Broadcast Receiver	100
	Intent Filter	150
	Content Provider	80
	Functionality	15
	URL	400
	Permission Normal	7
	Permission Dangerous	80
	Permission Not third party	50
	Permission System	80
	Permission with Typos	150
	Permission non standard	50
Cuckoo	DNS lookup	400
	HTTP request	400

V. CASE STUDY

As a case study we used a dataset of 1.5 million Android applications collected over the 2016. The dataset is recent and diverse in the set of attack vectors it represents: in order to have the same ratio between detected and undetected applications as in Koodous, we sampled a subset of 1 million apps.[9] As result, the dataset under analysis is composed by 65% undetected applications, 31% detected by signatures, and 4% detected through triage only.

A. EVALUATING CLUSTERING RESULTS

HDBSCAN has two parameters that mostly influence the results of the clustering: *min cluster size* (*mss*) determines the smallest size of a cluster, while *min samples* (*ms*) how conservative are the results. A higher value of *min samples* restricts clusters to more dense areas, but it also increases the number of outliers. We use $mss = 3$ and $ms = 1$; in other words, we considered only malware clusters containing a minimum of three samples as representative of a malware family.

We used a high-performance, open-source implementation of HDBSCAN in Python from Leland McInnes [21]. All experiments were performed on a 6-core Intel Xeon (CPU E5-1650 v2 @ 3.50GHz), with 128 GB of RAM, although HDBSCAN only used up to four cores and 6 GB of RAM in each run.

The quality of the clustering results is evaluated as a measure of the ability of correctly extending malware detection to undetected applications. However, given the difficulty of establishing a reliable ground truth in the field of malware analysis, evaluating the results was challenging. Finally, for the clustering validation we used all the available information: detection results and AVs labels extracted from VirusTotal reports, and signature labels extracted from existing YARA rules in Koodous.

Since clustering exploits the relationship between statistical similarities among applications, in contrast to the structural properties commonly used in AVs signatures, no one-to-one correspondence between clusters and AV labels is expected, however by combining several indexes we deliver a trustworthy quality measures of clustering performances. In order to estimate cluster assignment, we adopt the Adjusted Rand Index in combination with other external indexes as proposed by Rosemberg and Hirschberg [22]:

- *Adjusted Rand Index* (ARI) is defined as the number of pairs of items that are either both in the same cluster or both in different clusters in the two partitions, normalized over the total number of pairs of items. The index lies between 0 and 1: when two partitions agree perfectly, the Rand index achieves the maximum value 1, and more in general a larger adjusted Rand index means a higher agreement between two partitions. Moreover, ARI supports the measure of the agreements even when the compared partitions have different numbers of clusters
- *Homogeneity* (Hom.), which measures whether its clusters contain only data points which are members of a single class
- *Completeness* (Comp.), which measures whether all the data points that are members of a given class are elements of the same cluster
- *V-measure* (V-ms.), measured as the weighted harmonic mean of homogeneity and completeness; this is useful since homogeneity and completeness of a clustering solution run roughly in opposition: increasing the homogeneity of a clustering solution often results in decreasing its completeness.

Table 4 compares homogeneity and completeness index values between the families (i.e., clusters) inferred during clustering process, and the families labels extracted from Koodous signature names and VirusTotal AV labels.[10]

[9] In order to ensure the quality of the results and avoid artifacts, the sampling of 1 million applications have been repeated three times: in all the cases the proposed techniques showed coherent results.

[10] The comparison with VirusTotal AV labels is limited to 100,000 randomly selected applications.

TABLE 4. Comparison of Homogeneity (Hom.) and Completeness (Comp.) index values between the families inferred by the clustering process (using both the iterative clustering with different chunk sizes *N*, and the *non*-iterative version), and the families labels extracted from Koodous and VirusTotal.

	Koodous labels		VirusTotal labels	
N	Hom.	Comp.	Hom.	Comp.
50k	0.96	0.36	0.85	0.49
100k	0.96	0.35	0.85	0.49
200k	0.96	0.35	0.85	0.50
non-iterative	0.92	0.36	0.78	0.50

Results are compared using both the iterative clustering, with different chunk size N, and the *non*-iterative version.

Since AVs listed in VirusTotal commonly use different names to identify the same type of threat, we took advantage of AVclass [23], an automated labeling tool that, given the labels of multiple antivirus engines, returns the most likely family names for each sample, focusing on normalization, removal of generic tokes and alias detection. The implementation is open-source, available on GitHub [24], and natively provides VirusTotal integration.

Interestingly, all the cases reported in Table 4 show very high homogeneity value, which indicates that malware families identified by AVs signatures are further split in finer partitions during the clustering process. Moreover, precise clusters increase the effectiveness of the following automatically generaged signatures.

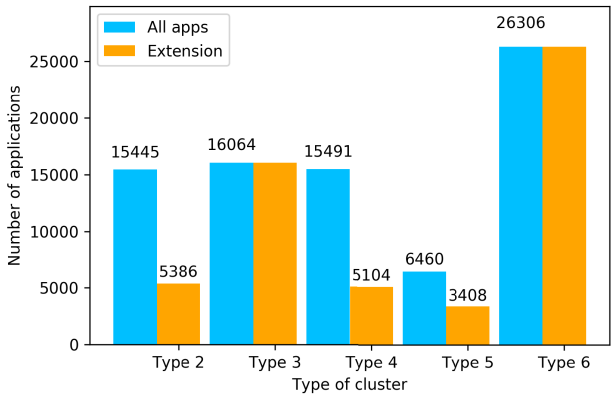


FIGURE 4. Number of total applications, and newly automatically inferred detections, for each type of malware family (Type 2...6). Results refer to the iterative clustering approach, using chunk size $N = 100k$, over a dataset of 1 million applications.

1) EXTENDING MALWARE DETECTION

Fig. 4 illustrates the result of the automatic detection extension for the 1 million applications under analysis: each bar in the plot is related to a family type (refer to Section IV-B for an accurate description of each type of malware family), illustrating both the total number of applications, and the number of those automatically identified as malicious. Results are obtained using the iterative clustering approach, with chunk size $N = 100k$. Note that Type 1 and 7 families are not shown, as the first consist of application that are already completely detected by signature, while the latter include families found within unknown applications, hence no direct information about their composition can be automatically inferred.

TABLE 5. Number of families automatically inferred by the clustering algorithm (using both the iterative clustering with different chunk sizes *N*, and the *non*-iterative version), using dataset of 1 million applications. Results are gathered for each type of malware family (Type 2...6).

N	Type 2	Type 3	Type 4	Type 5	Type 6
50k	1,890	2,949	1,467	463	2,846
100k	1,477	2,439	1,519	500	3,385
200k	1,193	2,203	1,436	536	3,133
non-iterative	435	1,046	2,126	536	4,629

Table 5 is complementary to Fig. 4, as it compares the number of families, for each family Type, using both the iterative clustering with different chunk sizes N, and the *non*-iterative version.

Among the clusters of Type 2 and 3, the system automatically identifies a total of *21,450* new malicious applications that will be automatically covered by new signatures, without requiring any human intervention. In more detail, *5,386* applications (Type 2) are found within clusters with other apps already detected by YARA signatures; while *16,064* applications (Type 3) are assigned to clusters purely made of applications detected during the triage phase only. As matter of fact, generating new family signatures for these applications allows to transform the knowledge of existing threats into a more reliable and scalable form of detection, without affecting the precision of the results: all those applications have been already identified as malicious by the community of malware experts.

On the other hand, *34,818* applications are assigned to families Type 4, 5 and 6: *20,464* are the newly identified potential threats, since previously marked as undetected. In this case, the proposed framework allows an easy identification of hard to find potential threats, reducing the human intervention from the manual analysis of thousands of applications to the validation of a very fewer number of families where applications reflect a similar behavior, eventually speeding up the procedure of new malware discovery. For example, the system identified a total of 500 families for the Type 4 (refer to Table 5, second row) reducing of an order of magnitude the need of manual analysis, as a detail analysis of a malicious application could take few hours, this approach results in a huge time saving.

2) EVALUATION OF MALWARE DETECTION EXTENSION

Aiming at evaluating the detection extension performance in a real-world case, we evaluate how the proposed system is accurate in relationship to the information of the detections availavable in VirusTotal. We choose VirusTotal as a well-known and trustworthy source of information about existing threats since it collects the detection results from

tens of independent AV companies. Moreover, recently other researches used the same metric [25].

TABLE 6. Comparison of the detection results between VirusTotal and two datasets of 50,000 applications, respectively undetected (und.) and detected (det.) by Koodous. Columns indicate the number of applications unknown (unk.), undetected (und.), detected by at least one AV (det.), and detected by more than three AVs, as reported by VirusTotal.

	VT unk.	VT und.	VT det.	VT det. >3
Koodous det.	18	72	49,910	49,717
Koodous und.	3,449	12,508	34,043	28,166

In order to evaluate the detection extension results, we firstly assessed how precisely Koodous detects malware samples, and how effectively covers all the malware variants. Starting from two randomly sampled subsets of 50,000 applications, respectively originally undetected and detected in Koodous, we cross-checked their maliciousness using VirusTotal. Results are illustrated in Table 6. The first line of the table (Koodous det.) shows that among detected applications, Koodous has 100% of precision, and very high recall (99.8%), as almost all Koodous detected applications are completely identified as malware by traditional AVs too, while only 100 applications (the 0.2% of the dataset) are unknown or undetected by VirusTotal. However, the second line of the table (Koodous und.) shows a very low accuracy (27.8%), as a consequence of a major diversity in the detection ratio among the applications undetected by Koodous and VirusTotal. Although such a difference could be partially explained by the different policies that traditional AVs use in identifying a malicious application, particularly regarding adware, this result further motivates the need of an automatic mechanism to increase the number of correct detections in Koodous.

With the awareness that VT detection results are not completely reliable, we only considered those clusters for which the VT information is available for all the applications. In order to calculate the accuracy of the proposed system, we adopted the following metrics:

- if the system proposes an extension to a malware family where all the applications are detected by VT, we consider the extension as *correct*;
- if the system proposes an extension to a family where all the applications are undetected by VT, the extensions is considered as *incorrect*;
- if the system proposes an extension to a cluster that mixes applications partially detected and undetected by VT, the result is considered *unknown*.

Table 7 illustrates the results. For each clustering experiment, each line of the table reports the number of applications that have been correctly or wrongly classified, according to the type of the cluster to which they were assigned. Without any human intervention, the system scores a minimum accuracy that ranges from 86.04% to 91.23%, and it has a worst case error of the 6.18%. A further manual inspection of the results revealed that several families completely undetected by VT are mostly related to aggressive adware samples, whose classification is subject to different considerations. Furthermore, results show that a smaller chunk size increase the precision the detection, reducing the error, although the absolute number of applications automatically extended is smaller. Accordingly, the chunk size can be set in accordance with the needs of the system.

TABLE 7. Evaluation of the accuracy of the clustering system to automatically identify groups of malicious applications, by comparing the detection of the new applications with VirusTotal. Columns *Correct* and *Incorrect* respectively reports the number of applications correctly or wrongly classified, while *Min* and *Error* illustrate the minimum precision and the maxim error of the proposed approach. Results are reported using both the iterative clustering with different chunk sizes *N*, and the *non*-iterative version.

N	Correct	Incorrect	Min %	Error %
50k	7,493	254	91.23	2.91
100k	12,502	877	86.04	6.03
200k	13,628	917	89.54	6.18
non-iterative	14,619	1,109	87.93	6.67

TABLE 8. Example of a Type 4 malware family. As the first two samples are already detected in Koodous by the YARA rule *Xynyin.Trojan*, the system identifies other applications within the cluster as potentially malicious too. The comparison with VirusTotal (the number of detection is reported) and a manual analysis confirm the accuracy of the system.

MD5	Detected Koodous	Detected VT
998faf5e7a0d45f6ad60903bc5d60817	Yes	12
5a8dd85a5707f520563069bf536f9d5f	Yes	19
695d6b9f97a9e992f8e321d36509c080	No	0
304754e9f8f95228af0e7118d62e999f	No	12
805d8770d6314f5adad266ddaba610e1	No	10
23863ddba21b96aea3e8b2cc120bb2b2	No	12

3) EXAMPLE OF MANUAL ANALYSIS OF A MALWARE FAMILY

Table 8 shows an example of a Type 4 malware family. As the first two samples are already detected by the signature *Xynyin.Trojan*[11] in Koodous, the system proposes to extend the detection to the other applications of the same cluster. The comparison of the detection results with VirusTotal[12] shows that all but one application are already detected, while a manual analysis of *Leagueoftankheroes3D*[13] confirm its affinity to the Xynyin malware family.[14]

One of the major benefit of a semi-supervised system is to limit the detection of false positives, and the operation is further simplified since the analysts should only focus on groups of similar applications, without considering single samples. As useful side effect, the system could be also used to improve the precision of the results, by reducing false

[11] https://koodous.com/rulesets/1225

[12] Detection results refer to 15 Nov 2016

[13] MD5: 695d6b9f97a9e992f8e321d36509c080

[14] On 24 August 2017 VirusTotal updated the detection, identifying the applications as malicious too.

positive detections for those families of applications that have been partially miss-classified by existing signatures.

TABLE 9. Comparison of the clustering results using using both the iterative version with different chunk sizes *N*, and the *non*-iterative one. Column *Time* indicates the time (in seconds) required by the clustering process, while column "Outliers" reports the number of outliers found at the end of the iterations.

N	Time (s)	Outliers
50k	5,746	64,553
100k	6,408	65,685
200k	10,573	67,081
non-iterative	16,592	119,919

4) THE ITERATIVE ALGORITHM

The adoption of the iterative approach brings a number of benefits: it proved to be essential in order to analyze millions of applications, and the resulting number of outliers, as illustrated in Table 9, is much lower than what was obtained by clustering all applications together. The time required by the clustering phase is proportional to the chunk size and it is up to one order of magnitude lower than in the *non*-iterative case.

The adoption of the iterative approach does not affect the quality of the results, even though using a bigger chunk size results in a greater number of new detections.

TABLE 10. Indexes comparison of the clustering label inferred by the iterative approach (with different chunk sizes *N*) using the assignment produced by the *non*-iterative version as a reference.

N	ARI	Homogeneity	Completeness	V-Score
50k	0.26	0.92	0.78	0.85
100k	0.27	0.93	0.81	0.86
200k	0.29	0.94	0.84	0.89

Table 10 compares the iterative approach using as a reference clustering assignment the one produced by the *non*-iterative version. A relatively low ARI value indicates a difference in the clustering assignment between the two approaches, while a very high homogeneity value, compared to completeness, is a clear sign of a finer cluster partitioning. In other words, using the iterative approach the quality of the information is not compromised, although the resulting clusters are smaller, hence less likely to contain enough applications that span different detection areas, finally resulting in a lower extension. A bigger chunk size lowers the differences between the iterative and the *non*-iterative assignment, as shown by an increasing *V-score* value. Eventually, if a large enough chunk size is used, the iterative approach produces almost the same results as the *non*-iterative one, while generally finding a higher number of clusters, as illustrated in Table 5, and a less outliers, Table 9.

Finally, in order to further test the scalability of the proposed method, we successfully applied the algorithm on a very large dataset of 10 million applications, using a chunk size $N = 500k$.

B. AUTOMATIC FAMILY SIGNATURES GENERATION

In order to evaluate the effectiveness of the automatic signature generator, we compare the detection results of several YARA rules automatically generated by the proposed algorithm with existing rulesets created by expert analysts.

TABLE 11. Comparison of detection performances of human authored YARA rules (Original) with automated generated ones (Auto). Last column reports the improvement (in percentage) for the newly generated rules. Detections are tested on a dataset of 1.5 million applications.

Rule name	Detections		
	Original	Auto	Improvement
SmsSender	539	1,004	+86.3%
Syringe	220	315	+43.2%
HummingBad2	136	257	+89.0%
Marcher2	559	652	+16.6%
SMSReg	159	172	+8.2%
VolcmanDropper	186	430	+131.2%
FakeGoogleChrome	516	822	+59.3%

Table 11 reports the results of the rules detections on a dataset of 1.5 million applications: in all the cases, the automated generated rules[15] performed better than the one authored by humans, increasing the detection from the 8.2% up to 131.2%, without generating any false positives.

Referring to Section IV-C, in all the cases the rule generation process stopped at the second step, as none of the new rules produced any false positives in the current dataset of applications. A further manual analysis of the detected applications, confirmed that no false positive was generated.

As shown in Table 12, the time required to generate a rule for few hundreds malware is always less than a minute, although when the target increases to a few thousands applications, the time required grows up to several minutes, as the most expensive part of the process is the check for false positives against a reference dataset. This is not considered a limitation, since all the process is automatic, and given the goodness of the results, it is of invaluable support for the family signature generation process.

TABLE 12. Number of literals, score and time (in seconds) required to generate each YARA rule.

Rule name	Literals	Score	Time (s)
SmsSender	15	412	43
Syringe	19	574	48
HummingBad2	12	599	52
Marcher2	20	686	49
SMSReg	34	537	42
VolcmanDropper	10	439	13
FakeGoogleChrome	15	407	43

Table 12 reports the number of literals (i.e., application features) and the final score for each generated YARA rule: referring to Section IV-C, each score is higher than the minimum threshold $T_{min} = 400$, satisfying the minimum

[15]Example rulesets could be found at the following address: `https://koodous.com/analysts/YaYaGen/rulesets`

requirement for acceptability in order to avoid false positive detections, and lower than the maximum threshold $T_{max} = 700$, as a result of the optimization process to increase the rule generality and therefor the ability to catch future malware variants.

As shown in the example of Section IV-C, in order to increase the effectiveness of a rule, urls are included only if are known to be malicious, like in case of `http://s.adslinkup.com/v2` for the *Syringe* malware family. Moreover, aiming at identifying malware with very high precision and avoiding false positives, whenever available, the automatic signature generator includes those attributes extracted from the application analysis that contains a typing mistake. For instance, the rule *YaYaMetasploit1*[16] includes a wrong permission `ACCESS_COURSE_LOCATION` instead of the correct one `ACCESS_COARSE_LOCATION`. Given the difficulty of reproducing such an uncommon mistake, we consider this feature as a hard indicator of the maliciousness of a sample.

VI. RELATED WORK

A. CLUSTERING APPLIED TO MALWARE ANALYSIS

The first attempt to automatically group computer malware based on their behavior dates back to Lee and Mody [26], who use a sequence of runtime events (e.g., registry and file system modifications) to cluster similar programs. As a similarity measure, they choose a variant of the *edit* distance, resulting demanding in term of computational resources, since it has a computational complexity $\mathcal{O}(n^2)$ in the number n of features.

Later, Bailey *et al.* [27] propose a system for automated malware classification and analysis as a remedy for the inconsistent and incomplete labeling that commonly affect traditional antivirus. By applying *single-linkage* Hierarchical agglomerative clustering (HAC) with Normalized Compression Distance (NCD) and using *inconsistency measure* as a cutting criteria, Bailey *et al.* are able to automatically categorize malware profiles into groups that reflect similar classes of behaviors in terms of system state changes. While results are generally affected by the restriction of dynamic analysis, for the first time they introduce the idea of ''detection through clustering,'' exploited in our proposed framework.

In their work, Apel *et al.* [5] study which combination of metrics (i.e., *Edit Distance*, *Approximated Edit Distance with Blockwise Hashing*, *NCD* and *Manhattan Distance*) and n-gram features are mostly appropriate for determining relations between malware samples. They define three different criteria to support their evaluation (i.e., appropriateness, computable efficiency and local sensitiveness), using *single-linkage* HAC as clustering algorithm. Experimental results show that Manhattan Distance along with 3-grams deliver the best results, while NCD and Edit Distance generally perform poorly.

[16]https://koodous.com/my_rulesets/3466

Neither Lee and Mody [26], nor Bailey *et al.* [27] have any specific solution to large-scale clustering. On the other hand, Bayer *et al.* [13], Rieck *et al.* [28], and Jang *et al.* [12] directly address the problem of managing large datasets, developing methods to scale the clustering process.

Bayer *et al.* [13] propose a scalable malware clustering approach using a combination of approximate and hierarchical clustering with Local Sensitive Hashing (LSH) [29] to significantly reduce the number of distance computations. By extending Anubis [30], they are able to extract detailed behavioral-reports based on taint tracking results and network captures from malware execution. In particular, the taint engine allows them to map low-level operations (e.g., system calls) to operating system objects (e.g., registry keys and files). By deploying LSH, Bayer *et al.* are capable of clustering 75,000 samples in less than 3 hours. By contrast, Rieck *et al.* [28], [31] proposes an incremental approach, where they alternate a prototype-based clustering algorithm with a classification step, eventually reducing the runtime complexity by performing clustering only on representative samples.

Jang *et al.* [12] develop BitShred as remedy to the problem of clustering large data sets with high-dimensional feature sets. They propose to use feature hashing to reduce the dimensionality of high-scale feature sets, while reducing the computational cost of the calculation of the Jaccard index using an approximated version that exploits bit-vector arithmetic. However, since BitShred simply relies on a static analysis approach, results are susceptible to binary level obfuscation.

In 2010, Perdisci *et al.* [11] propose a network-based version of a behavioral malware clustering system, relying on a three-step clustering refinement process, starting from the analysis of malicious HTTP traces. The first phase consists in a *coarse-grained clustering* where malware samples are grouped together according to simple statistical similarities; subsequently, a *fine-grained clustering* further splits samples considering structural properties of HTTP queries. In the final step, fine-grained clusters whose centroids are close to each other are merged together. The system is tested on HTTP traces generated from 25,000 applications using single-linkage HAC and the Davies-Bouldin (DB) validity index [32] as cutting criteria. While the underlying idea of a multi-step clustering refinement process is quite interesting, this practically results in the biggest limitation to the scalability of their work. Moreover, Perdisci *et al.* limit behavioral analysis to HTTP-based malware only, which in practice can be easily bypassed by using an encrypted protocol (e.g., HTTPS).

In 2013 Hu et. al [10] present MutantX-S, focusing on malware comparison and triage on a large scale. Their system falls into the static-analysis category, since it relies on features extracted from the malware instructions. MutantX-S can efficiently cluster a large number of samples into families based on program static features, by extracting N-gram features directly from the x86 opcode sequences and exploiting a feature hashing technique to reduce features dimensionality, thus

significantly lowering the memory requirement and computation costs. MutantX-S adopts the same prototype-based algorithm of [31] because of its efficiency and explicit expression of malware features.

In the Android context, ClusTheDroid [33] is the first research to combine behavioral analysis and clustering to specifically target Android malware. The goal is both to develop a tool, and to evaluate clustering alternatives. Finally they focused on *single and complete linkage* HAC, using a feature set composed of 38 numerical quantities extracted from the CopperDroid [34] report, and weighted according to a three-level interpretation of malware behaviors.

Differently from most of the previous works [5], [11], [13], [27], [33] that rely on the HAC algorithm (which is both computationally and storage expensive, respectively $\mathcal{O}(n^2 \log n)$ and $\mathcal{O}(n^2)$ [35]), we use HDBSCAN, that with N data points has an average complexity approximately $\mathcal{O}(N \log N)$ [20], and a space requirement $\mathcal{O}(n)$, making it applicable to large datasets. Furthermore, differently from [31], we devise an iterative clustering approach where HDBSCAN is iteratively applied over the entire dataset, without the needed of alternate any classification step, finally discovering precise families of applications with a shared behavior.

B. EVALUATING CLUSTERING RESULTS

The clustering problem is inherently ill-posed, in the sense that there is no single criterion that measures how well a clustering of the data corresponds to the real world [36]. Cluster validity analysis often involves the use of *subjective criteria of optimality* specific to a particular application. Therefore, no commonly accepted standard of validating the output of a clustering procedure exists [37]. In real-world applications, it is often completely infeasible to manually investigate the results of a clustering, making necessary the definition of automatic measures [33]. Helpful metrics to determine the quality of a clustering process are commonly classified in *internal* and *external* indexes. The former evaluates both cluster *cohesion* and *separation*, which determine how distinct or well-separated a cluster is from others. On the other hand, the latter uses a reference set as a means of quality control for the setup of the clustering algorithm [33].

In the field of malware analysis, clustering validation is further complicated by the intrinsic difficulty of establishing a reliable ground truth. Firstly, malware analysis is challenging and it gets more difficult when anti-analysis, triggering sequences and dynamic code loading techniques are in place. Secondly, not even a manual categorization would provide a reliable partition, since most of the malware could not be unequivocally assigned in categories; not to mention the unrealistically high amount of time it would require.

As a reference set is not available, one possibility is to take advantage of labels assigned to each malware sample by several antivirus scanner. The availability of services that specifically provide these results (e.g., Metadefender[17] or VirusTotal[18]) eases the procedure. However, there is an intrinsic complexity in defining a unique labeling schema, since most of the malware result in being marked as belonging to one malicious category only. As a matter of fact, Bailey et. al. [27] showed that antivirus labeling fails in satisfying three fundamental criteria: consistency among different products, completeness in malware tagging, and conciseness in label semantics. One possible explanation is that signatures used in the malware-matching algorithms mostly evaluate static properties of the binary, in contrast to behavioral properties: the result is that families found using static features might be quite different from ones established using behavioral features. Moreover, different AV products apply different criteria and granularity to rule generation, resulting in inconsistent results. Despite the complexity and intrinsic challenges of the procedure, given the importance of automatically building a malware reference dataset to evaluate clustering results, the problem was directly tackled in different researches, such as VAMO [38] and AVclass [23].

In the literature of malware clustering, several techniques are proposed. Bayer *et al.* [13] and Jang *et al.* [12] use precision and recall to compare the results of their system-level behavioral clustering to a reference dataset, defining a manual mapping between labels assigned by different AVs. However, as the dataset size increase this method becomes hardly sustainable and quite costly. Similarly, ClusTheDroid [33] used a reference set developed through manual analysis [39].

On the other hand, Apel *et al.* [5] choose to take into consideration the amount of ''shared behaviour'' that can be found among different analysis traces within the same cluster of applications. In practice, each system call is modeled as a single character, and the evaluation is computed in linear time finding all substrings in a generalized suffix tree, using the algorithm described in [40]. The main limitation of this technique is related to the choice of the reference dataset, since Apel *et al.* use an artificial dataset starting from three real-world malware traces, then divided into blocks of system calls and randomly permutated.

Differently, Perdisci *et al.* [11] tackle the problem by measuring the cohesion and separation of each cluster, in terms of agreement between labels assigned by cluster and multiple AV scanners. However, since AV labels have been shown to be inconsistent [41], the measures of cluster cohesion and separation only give an indication of the validity of the clustering results.

C. SIGNATURE-BASED DETECTION

Early AV products used the hash value of an application to detect malicious software. However, every modification in the source code, as tiny as one byte, results in a detection evasion. Today's signatures are pattern-matching rules commonly defined on static or dynamic properties of applications under analysis and, even though they are assisted by heuristic

[17] https://metadefender.opswat.com

[18] https://virustotal.com/

and AI-based solutions, still represent the most reliable (i.e., with the lowest false positives) antivirus technology.

1) AUTOMATIC SIGNATURE GENERATION

A number of prior works propose systems to automatically generate different types of network signatures to identify malicious traffic.

Honeycomb [42], Autograph [43], and EarlyBird [44] propose the generation of signatures comprising a single contiguous string (i.e., token). Later on, PAYL [45], Nemean [46], Hamsa [47] and Botzilla [48] introduce more complex methods based on the token subsequence signatures.

Other researches like ProVex [49], AutoRE [50], ShieldGen [51], and [52] also tackle the problem of automatically generating network signatures, although their applicability is specific to the network traffic detection.

In 2005, Newsome et. al. introduces Polygraph [53], a system which exploits the Token-Subsequence algorithm to automatically obtain IDS signatures to match polymorphic worms. Polygraph is tested against three real-world exploits and is able to successfully generate HTTP and DNS signatures with a low false positive rate.

Perdisci *et al.* [11] also tackles the problem of automatically generate network signatures for cluster centroids, with the aim of deploying them into an IDS at the edge of a network in order to detect malicious HTTP traffic. Since malware samples may contact legitimate websites for malicious purposes, instead of pre-filtering HTTP traffic against legitimate websites, authors apply a pruning process by testing the signature set against a large dataset of legitimate traffic, while discarding signatures that generate false positives, although such an approach is as effective as it is the legitimate traffic available.

In the Android context, Faruki *et al.* [54] propose *AndroSimilar*, a statistical signature-based solution that generates variable-length signatures for the application under test and identifies malware on the basis of a similarity percentage with a dataset of known malicious samples.

Another approach is presented in *DroidAnalytics* [55], a signature-based analytic system, which extracts and analyzes applications at opcode level. Firstly, a three-level signature (i.e., methods, classes, application) is generated by combining the API call traces, then the malware is associated to a family according to its malicious content.

While [54] shows robustness against control-flow obfuscation, junk method insertion and string encryption, [55] could fail in the detection of repackaged malware. On the the other hand, both solutions are affected by a high false-positive rate due to the wrong choice of signature patterns available in both malicious and benign applications.

Since the release of YARA [56], a patten-matching language designed to help to identify and classify malware samples, a few automatic tools have been proposed to generate malware signatures which balance the required generality to catch future samples with the need of avoiding false positives detections.

In 2013, Chris Clark develops *YaraGenerator*,[19] a python program which automatically generates YARA rules by sampling a small subset of common strings between malware, while blacklisting goodware ones. Although the tool is designed to work with any type of malicious file, in order to increase the efficacy of the results, specific dataset of goodware strings are available for several file formats (e.g., Windows executable, PDF, email and office document).

Similarly, *yarGen*[20] is a python tool developed by Florian Roth to automatically generate YARA rules by combining the topmost malware strings, while removing those that also appear in goodware files. By using fuzzy regular expressions, each malware string is assigned a score proportionally to the inverse of its frequency, and the ''Gibberish Detector'' allows to select real language over character chains without any meaning. The tool also exploits a naive-bayes-classifier to classify candidate strings, avoiding compression or encryption garbage in favor of more generic strings. Finally, each rule is created by combining the 20 strings with the highest score. The result of the generation process may be a single rule, specific to one sample, or a *super rule*, catching malware variants and groups.

While both YaraGenerator and yarGen have been developed aiming at supporting the rule creation, rather then completely replacing the role of expert analysts, as a major drawback, their efficacy strongly relies on the completeness of the dataset of goodware strings.

Differently from previous works, which mostly rely on the search of an optimal sequence of opcodes or strings, the proposed algorithm generates signatures from a set of attributes extracted from the application analysis, finding an optimal combination to minimize false negatives and guarantee zero false positives in the current set of applications. None of the previous researches can be directly applied to solve such a problem. Moreover, the proposed approach exploits an heuristic measure to find the right balance between rule generality and specificity, using the same criteria that expert analysts adopted while authoring existing rulesets.

VII. LIMITATIONS

A major limiting factor of the described semi-supervised approach is represented by the ability to extract meaningful information from the applications under analysis. Indeed, the accuracy of the analysis directly affects the clustering results and the automatic rule generation process. The Android platform lacks of mature reverse engineering tools compared to the ones used for x86 malware [57]. Since each malware is different, automatically finding the malicious code by means of static analysis is difficult, because it is mixed with benign code; moreover dynamic code loading and reflections make the analysis even harder. Unfortunately, most malware include trigger-based anti-analysis techniques that delay or hide their malicious activities at the first application run

[19] https://github.com/Xen0ph0n/YaraGenerator
[20] https://github.com/Neo23x0/yarGen

or in an emulated environment. For instance, the family of applications known as *DroidKungFu*[21] uses a time bomb of 240 minutes to schedule the execution of its malicious code, indeed a simple dynamic analysis fails to observe interesting behaviors. However, in this research we do not address problems related to application analysis, as we focus on the detection of new samples and the automatic generation of new signatures.

Evasion attacks, such as noise-injection attacks [58] and other similar approaches [59]–[62] may affect the correctness results of the clustering and the signature generation. Those attacks rely on the ability of injecting, in the analysis platform, applications specially crafted to mislead the clustering process and the generation of a good detection model.

In the described system, an attacker could exploit such attacks by injecting specially crafted applications with the final goal of generating a false positive or a false negative detection. However, in both cases we assume that the detection information of already known threats (identified through signatures or by triage) cannot be maliciously tampered, thus new injected families will result in a Type 4, 5, 6 or 7, hence will be subject to manual validation.

If the attacker wants to deliberately generate a false positive, several malicious applications whose statistical properties are similar to a target goodware can be injected. Since a false positive detection mainly generates a disruption to a third party service, causing a reputation fail for the AV solution, the magnitude of the echo is proportional to the diffusion of the target goodware. As a matter of fact, the analyst will be alerted by such a family.

On the other hand, if the goal is to generate a false negative, the attacker could inject several goodware with the same statistical properties of a target unknown malicious app. Such a family could be misclassified as a completely goodware even after the validation process, as the manual analysis focus only on few samples. However, such a situation applies only as far as the malware is a zero-day, and no specif knowledge about that threat is available. The identification of zero-day malware is a challenging and an open-research problem in the security community.

Finally, the proposed system strongly relies on the information provided by the platform to automatically extend the detection to new applications and identify new potential malware families. It is a prerequisite that this information is not tampered by any malicious actor. Although Koodous provides protection mechanism for both YARA rules (rules before becoming active undergo a review process) and the triage process (community members are subject to a reputation check), it is not intent of this research to tackle those issues, leaving their study to future works.

VIII. FUTURE WORKS

The work presented in this paper can be improved and extended in a number of ways. At the time of writing, we are focusing our efforts on the correct management of new samples collected every day. Since the current version of the system does not allow to incrementally add new applications to the existing model, when enough samples are collected, those are treated as a new iteration of the clustering process. As an alternative to the iterative approach, incremental clustering algorithms have been proposed [63]–[65], although still non directly applicable to HDBSCAN. Their study and adoption will be addressed in future works.

[21] Sample MD5: `7f5fd7b139e23bed1de5e134dda3b1ca`

IX. CONCLUSION

In this paper, we introduced a set of semi-supervised techniques with the ultimate goal of assisting human experts in the generation of malware family signatures. As a result, we developed a scalable framework able to dig into massive datasets of Android applications with the main purpose of identifying new malware samples, while reducing false positive detections.

Our study shows that combining the scalability of the automatic techniques with the inherent flexibility of the manual analysis, achieves the best performances. Eventually, the proposed approach introduces two essential automation improvements in a well known and tested AVs standard detection mechanism based on signatures. An iterative clustering algorithm allows for easy identification of hard to find potential threats, reducing the human intervention from the manual analysis of thousands of applications to the validation of a much smaller number of clusters where applications reflect a similar class of behavior. Subsequently an automated procedure, which exploits a heuristic optimization strategy, generates a set of YARA rules to cover newly identified malware with an acceptable generalization capability yet minimizing false positives.

Experimental results on a dataset of 1.5 million distinct Android applications confirm the effectiveness of the proposed system, both in the identification of new malware samples and in the generation of new family signatures in the form of YARA rules.

Finally, the proposed approach has been deployed in January 2018 and, since then, it is in use on Koodous, the mobile antivirus platform developed by Hispasec.

ACKNOWLEDGEMENT

The authors would like to thank Dario Lombardo for his support and insightful comments.

This article is based upon work from COST Action CA15140 'Improving Applicability of Nature-Inspired Optimisation by Joining Theory and Practice (ImAppNIO)' supported by COST (European Cooperation in Science and Technology).

REFERENCES

[1] (Aug. 2010). *AndroidOS.FakePlayer | Symantec*. Accessed: Mar. 24, 2017. [Online]. Available: https://www.symantec.com/security_response/writeup.jsp?docid=2010-081100-1646-99

[2] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection & protection: A survey," *Int. J. Adv. Comput. Sci. Appl*, vol. 7, no. 2, pp. 463–475, 2016.

[3] (Mar. 2017). Accessed: Dec. 11, 2017. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf

[4] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Cambridge, MA, USA: Springer, 2008, pp. 98–115.

[5] M. Apel, C. Bockermann, and M. Meier, "Measuring similarity of malware behavior," in *Proc. IEEE 34th Conf. Local Comput. Netw. (LCN)*, Oct. 2009, pp. 891–898.

[6] J. Oberheide, E. Cooke, and F. Jahanian, "CloudAV: N-version antivirus in the network cloud," in *Proc. USENIX Secur. Symp.*, 2008, pp. 91–106.

[7] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.

[8] *Apktool—A Tool for Reverse Engineering 3rd Party, Closed, Binary Android Apps*. Accessed: Mar. 24, 2017. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[9] J. Freke. (2013). Smali, An Assembler/Disassembler for Android's Dex Format, Google Project Hosting. [Online]. Available: http://code.google.com/p/smali

[10] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable malware clustering based on static features," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 187–198.

[11] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of HTTP-based malware and signature generation using malicious network traces," in *Proc. NSDI*, vol. 10, 2010, pp. 1–14.

[12] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature hashing malware for scalable triage and semantic analysis," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 309–320.

[13] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, vol. 9, 2009, pp. 8–11.

[14] (Nov. 2013). *YARA—The Pattern Matching Swiss Knife for Malware Researchers*. Accessed: Mar. 27, 2017. [Online]. Available: https://virustotal.github.io/yara/

[15] A. Desnos. (2011). *Androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications... and More (Ninja!)*. Accessed: Jun. 10, 2014. [Online]. Available: https://github.com/androguard/androguard

[16] A. Desnos and P. Lantz, "DroidBox: An Android application sandbox for dynamic analysis," Lund Univ., Lund, Sweden, Tech. Rep., 2011.

[17] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. KDD*, vol. 96. 1996, pp. 226–231.

[18] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Proc. Pacific–Asia Conf. Knowl. Discovery Data Mining*. New York, NY, USA: Springer, 2013, pp. 160–172.

[19] R. J. G. B. Campello, D. Moulavi, A. Zimek, and J. Sander, "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Trans. Knowl. Discovery Data*, vol. 10, no. 1, 2015, Art. no. 5.

[20] L. McInnes and J. Healy. (2017). "Accelerated hierarchical density clustering." [Online]. Available: https://arxiv.org/abs/1705.07321

[21] L. McInnes, J. Healy, and S. Astels, "HDBSCAN: Hierarchical density based clustering," *J. Open Source Softw.*, vol. 2, no. 11, pp. 1–2, Mar. 2017, doi: 10.21105%2Fjoss.00205.

[22] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external evaluation measure," in *Proc. EMNLP-CoNLL*, vol. 7, 2007, pp. 410–420.

[23] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A tool for massive malware labeling," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Paris, France: Springer, 2016, pp. 230–253.

[24] (Jul. 2016). *malicialab/AVCLASS: AVClass Malware Labeling Tool*. Accessed: Mar. 27, 2017. [Online]. Available: https://github.com/malicialab/avclass

[25] Y. Li *et al.*, "Experimental study of fuzzy hashing in malware clustering analysis," in *Proc. 8th Workshop Cyber Secur. Exp. Test (CSET)*, vol. 5, no. 1, 2015, p. 52.

[26] T. Lee and J. J. Mody, "Behavioral classification," in *Proc. EICAR Conf.*, 2006, pp. 1–17.

[27] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of Internet malware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Berlin, Germany: Springer, 2007, pp. 178–197.

[28] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Paris, France: Springer, 2008, pp. 108–125.

[29] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," *VLDB*, vol. 99, no. 6, pp. 518–529, 1999.

[30] U. Bayer, C. Kruegel, and E. Kirda, "Anubis: Analyzing unknown binaries," Virus Bull., Geneva, Switzerland, Tech. Rep., 2009.

[31] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, 2011.

[32] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "On clustering validation techniques," *J. Intell. Inf. Syst.*, vol. 17, no. 2, pp. 107–145, Dec. 2001.

[33] D. Korczynski, "ClusTheDroid: Clustering android malware," Royal Holloway, Univ. London, London, U.K., Tech. Rep., 2015.

[34] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. NDSS*, 2015, pp. 1–15.

[35] P.-N. Tan, M. Steinbach, and V. Kumar, "Data mining cluster analysis: basic concepts and algorithms," *Introduction to data mining*, 2013.

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[37] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sep. 1999.

[38] R. Perdisci and M. U, "VAMO: Towards a fully automated malware clustering validity analysis," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 329–338.

[39] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2012, pp. 95–109.

[40] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, Sep. 1995.

[41] A. Mohaisen and O. Alrawi, "AV-Meter: An evaluation of antivirus scans and labels," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Milan, Italy: Springer, 2014, pp. 112–131.

[42] C. Kreibich and J. Crowcroft, "Honeycomb: Creating intrusion detection signatures using honeypots," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 51–56, 2004.

[43] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proc. USENIX Secur. Symp.*, San Diego, CA, USA, vol. 286, 2004, p. 19.

[44] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. OSDI*, vol. 6, 2004, p. 4.

[45] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Berlin, Germany: Springer, 2005, pp. 227–246.

[46] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proc. USENIX Secur. Symp.*, 2005, pp. 97–112.

[47] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proc. IEEE Symp. Secur. Privacy*, May 2006, pp. 15 and 47.

[48] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov, "Botzilla: Detecting the 'phoning home' of malicious software," in *Proc. ACM Symp. Appl. Comput.*, 2010, pp. 1978–1984.

[49] C. Rossow and C. J. Dietrich, "ProVeX: Detecting botnets with encrypted command and control channels," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Berlin, Germany: Springer, 2013, pp. 21–40.

[50] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov, "Spamming botnets: Signatures and characteristics," in *Proc. SIGCOMM*, 2008, vol. 38. no. 4, pp. 171–182.

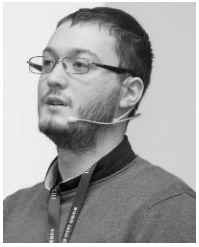
[51] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto, "ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2007, pp. 252–266.

[52] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda, "Automatically generating models for Botnet detection," in *Proc. Eur. Symp. Res. Comput. Secur.* Saint-Malo, France: Springer, 2009, pp. 232–249.

[53] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proc. IEEE Symp. Secur. Privacy*, May 2005, pp. 226–241.

[54] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust statistical feature signature for Android malware detection," in *Proc. 6th Int. Conf. Secur. Inf. Netw.*, 2013, pp. 152–159.
[55] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Jul. 2013, pp. 163–171.
[56] (Jan. 2008). *Virus Bulletin: Rule-Driven Malware Identification and Classification*. Accessed: Mar. 4, 2017. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2008/01/rule-driven-malware-identification-and-classification
[57] N. Kiss, J.-F. Lalande, M. Leslous, and V. V. T. Tong, "Kharon dataset: Android malware under a microscope," in *Proc. Learn. Authoritative Secur. Exp. Results (LASER) Workshop*, 2016, pp. 1–19.
[58] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. IEEE Symp. Secur. Privacy*, May 2006, p. 15–pp.
[59] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting signature learning by training maliciously," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Hamburg, Germany: Springer, 2006, pp. 81–105.
[60] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proc. Netw. Distrib. Syst. Symp.*, 2016, pp. 1–15.
[61] B. Biggio *et al.*, "Poisoning behavioral malware clustering," in *Proc. Workshop Artif. Intell. Secur. Workshop*, 2014, pp. 27–36.
[62] J. Crussell and P. Kegelmeyer, "Attacking DBSCAN for fun and profit," in *Proc. SIAM Int. Conf. Data Mining*, 2015, pp. 235–243.
[63] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental clustering for mining in a data warehousing environment," in *Proc. VLDB*, vol. 98, 1998, pp. 323–333.
[64] N. Goyal, P. Goyal, K. Venkatramaiah, P. C. Deepak, and P. Sanoop, "An efficient density based incremental clustering algorithm in data warehousing environment," in *Proc. Int. Conf. Comput. Eng. Appl. (IPCSIT)*, vol. 2, 2011, pp. 482–486.
[65] A. M. Bakr, N. M. Ghanem, and M. A. Ismail, "Efficient incremental density-based algorithm for clustering large datasets," *Alexandria Eng. J.*, vol. 54, no. 4, pp. 1147–1154, 2015.

ANDREA ATZENI received the M.Sc. and Ph.D. degrees in computer engineering from Politecnico di Torino. He is currently a Senior Research Assistant at the TORSEC Security Group, Politecnico di Torino. In last fifteen years, he contributed to a number of large-scale European research projects under the FP5, FP6 and FP7, and CIP programmes. He addressed, among the others, the definition of security requirements in multi-platform systems, mobile security modelization of user expectation on security and privacy, security specification, risk analysis and threat modeling for complex cross-domain architectures, development of cross-domain usable security, digital and cloud forensics, development and integration of cross-border authentication mechanisms, malware analysis, and modeling.



FERNANDO DÍAZ is currently pursuing the B.Sc. degree in health engineering with the University of Malaga. He is with Hispasec Sistemas as a Security Engineer. He is a Malware Analyst and also a Software Engineer. His daily work focuses on automated malware configuration extractions, distributed analysis environments, and developing software for the Koodous platform. His research includes analysis of new malware families and IoT malware.

ANDREA MARCELLI received the M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2015, where he is currently pursuing the Ph.D. degree in computer and control engineering. He is a member of the CAD Group, Politecnico di Torino. His research interests include malware analysis, semi-supervised modeling, machine learning and optimization problems, with main applications in computer security.

ANTONIO SÁNCHEZ received the M.Sc. degree in computer science from the Universidad de Jan, Spain, in 2013. Since 2012, he has been a Security Engineer at Hispasec Sistemas S.L. He is also a Malware Analyst and also a Research Engineer. His daily work focuses on the improvement of systems for the automatic detection and analysis of malware samples, while his research interests include new techniques for storage, recovering, and correlation of big data.

GIOVANNI SQUILLERO (M'01–SM'14) received the M.S. and Ph.D. degrees in computer science in 1996 and 2001, respectively. He is currently an Associate Professor of computer science at Politecnico di Torino. His research mixes the whole spectrum of bio-inspired metaheuristics with computational intelligence, machine learning, and selected topics in electronic CAD, games, multi-agent systems. Other activities focus on the development of optimization techniques able to achieve acceptable solutions with limited amount of resources, mainly applied to industrial problems. He is a member of the IEEE Computational Intelligence Society Games Technical Committee.

ALBERTO TONDA received the Ph.D. degree in computer science engineering from Politecnico di Torino, Torino, Italy, in 2010. He is a senior Permanent Researcher with the Université Paris-Saclay, INRA, France. His research interests include semi-supervised modeling of complex systems, evolutionary optimization and machine learning, with main applications in food science and industry.