Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability

Yu Feng
University of Texas at Austin
yufeng@cs.utexas.edu

Osbert Bastani
Stanford University
obastani@cs.stanford.edu

Ruben Martins
University of Texas at Austin
rmartins@cs.utexas.edu

Isil Dillig
University of Texas at Austin
isil@cs.utexas.edu

Saswat Anand
Google, Inc.
saswatanand@google.com

Abstract—This paper proposes a technique for automatically learning semantic malware signatures for Android from very few samples of a malware family. The key idea underlying our technique is to look for a *maximally suspicious common subgraph (MSCS)* that is shared between all known instances of a malware family. An MSCS describes the shared functionality between multiple Android applications in terms of inter-component call relations and their semantic metadata (e.g., data-flow properties). Our approach identifies such maximally suspicious common subgraphs by reducing the problem to *maximum satisfiability*. Once a semantic signature is learned, our approach uses a combination of static analysis and a new *approximate signature matching* algorithm to determine whether an Android application matches the semantic signature characterizing a given malware family.

We have implemented our approach in a tool called ASTROID and show that it has a number of advantages over state-of-the-art malware detection techniques. First, we compare the semantic malware signatures automatically synthesized by ASTROID with manually-written signatures used in previous work and show that the signatures learned by ASTROID perform better in terms of accuracy as well as precision. Second, we compare ASTROID against two state-of-the-art malware detection tools and demonstrate its advantages in terms of interpretability and accuracy. Finally, we demonstrate that ASTROID's approximate signature matching algorithm is resistant to behavioral obfuscation and that it can be used to detect zero-day malware. In particular, we were able to find 22 instances of zero-day malware in Google Play that are not reported as malware by existing tools.

I. INTRODUCTION

Due to the enormous popularity of Android as a mobile platform, the number of applications ("apps") available for Android has skyrocketed, with 1.6 million apps being currently available for download. Unfortunately, the soaring number of Android users has also led to a rapid increase in the number of Android malware, with 4,900 malware samples being introduced *every day* [1]. Correspondingly, this upsurge in Android malware has also led to a flurry of research for automatically detecting malicious applications [2, 3, 4, 5, 6, 7].

Generally speaking, approaches for automated malware detection can be classified as either *signature-based* or *learning-based*. Signature-based techniques look for specific patterns in the application to determine whether the app is malicious, and, if so, which malware family the app belongs to [8, 2, 9, 10]. These patterns can either be syntactic (e.g., sequence of instructions) or semantic (e.g., control- or data-flow properties). Signature-based approaches allow security analysts to quickly identify the malicious component of an application; hence, they are widely-used by several commercial anti-virus (AV) companies. However, one key shortcoming of these techniques is that they require a trained security analyst to *manually* write suitable signatures that can be used to detect each malware family. Unfortunately, this manual effort is typically time-consuming and error-prone.

Learning-based techniques [3, 6, 11, 12, 4, 13, 14, 5] aim to address this limitation by automatically learning a malware classifier from data. These techniques extract various features from the application and use standard machine learning algorithms to learn a classifier that labels apps as either benign or malicious. Compared to signature-based techniques, current learning-based approaches suffer from a number of shortcomings:

- They produce results that are difficult to interpret (for example, they typically cannot be used to determine the malware family), which makes it difficult for a security analyst to discharge false positives.
- They typically require a large number of samples from each malware family, which is problematic for families that have recently emerged or that are rare (58% of malware families in [3] have fewer than 5 samples).

This paper aims to overcome these disadvantages of existing malware detectors by proposing a new technique to *automatically infer malware signatures*. By identifying malware based on inferred signatures, our approach retains all the advantages of signature-based approaches: it can pinpoint the location of the malicious components as well as the

corresponding malware family, and requires very few samples (<5 in our evaluation). Furthermore, our approach can learn *semantic* signatures that are resilient to low-level obfuscation mechanisms and produces very few false alarms. Finally, because our signature matching algorithm uses *approximate* (rather than *exact*) matching, our algorithm is also resilient to high-level obfuscation mechanisms that modify the program's control flow and data-flow properties.

The first key insight underlying our approach is to automatically identify a semantic pattern that (a) occurs in all instances of a given malware family, and (b) is *maximally suspicious* (i.e., maximizes the number of "suspicious" features that are typically *not* found in benign apps). Here, criterion (a) serves to minimize the number of false negatives: If the pattern does not occur in *all* instances of the malware family, then our signatures would not match all malicious apps, thereby resulting in false negatives. In contrast, criterion (b) serves to minimize false positives: By requiring that the identified semantic pattern is *maximally suspicious*, we ensure that our signatures are unlikely to flag benign apps as malicious.

The second key insight underlying our technique is to automatically learn these semantic patterns by finding a *maximally suspicious common subgraph (MSCS)* of the malware instances. An MSCS describes the shared functionality between multiple Android applications in terms of inter-component call relations and their semantic metadata (e.g., data-flow properties, intent filters, API calls etc.). Our approach automatically finds an MSCS by reducing the problem to *maximum satisfiability (MaxSAT)*. Intuitively, the MaxSAT problem aims to maximize the amount of suspicious metadata while ensuring that this functionality is shared between all instances of the malware family. The solution to the MaxSAT problem can then be directly translated into a semantic signature that characterizes a specific malware family.

Our third and final insight is to utilize the proposed signature inference algorithm for approximate matching. Specifically, given a database of malware signatures and a new application $\mathcal{A}$, we must decide whether $\mathcal{A}$ matches any of these signatures. Rather than performing *exact* signature matching as done in previous work [2], we employ a novel *approximate signature matching* algorithm. The key idea is to generate a new signature $\mathcal{S}$ *assuming* that $\mathcal{A}$ is an instance of family $\mathcal{F}$. We then decide if $\mathcal{A}$ is actually an instance of $\mathcal{F}$ based on the similarity score between $\mathcal{S}$ and $\mathcal{F}$'s existing signature. The main advantage of this approach is that it makes our technique even more resilient to obfuscations, including those that change program behavior.

We have implemented the proposed technique in a tool called ASTROID[1] and evaluate it in a number of ways. Our first experiment shows that the signatures automatically inferred by ASTROID are competitive with (in fact, better than) manually-written signatures used for evaluating previous work [2]. We also compare ASTROID with two state-of-the-art Android malware detectors, namely DREBIN [3] and MASSVET [15], and show that ASTROID compares favorably with these tools in terms of accuracy, false positives, and interpretability. Third, we demonstrate that ASTROID's approximate signature matching algorithm is both resilient to various

behavioral obfuscations and is also useful for detecting zero-day malware. Specifically, in a corpus of apps collected from Google Play, ASTROID identified 103 apps from more than 10 malware families for which we did not previously have signatures. Furthermore, among these 103 apps, 22 of them were previously unknown and are not reported as malware by existing tools.

This paper makes the following key contributions:

- We propose a novel technique for inferring malware signatures from few samples of a malware family.
- We formulate signature inference as the problem of finding a *maximally suspicious common subgraph (MSCS)* of a set of Android applications and show how to reduce MSCS detection to *maximum satisfiability*.
- We propose a novel *approximate signature matching* algorithm that leverages automated signature synthesis.
- We implement our approach in a tool called ASTROID and evaluate it against manually written malware signatures, other state-of-the-art malware detectors, and behavioral obfuscation strategies.

II. BACKGROUND

In this section, we provide some background that is necessary for understanding the rest of this paper.

A. Android Basics

An Android application consists of four kinds of components, namely *Activity*, *Service*, *BroadcastReceiver*, and *ContentProvider*. Every screen of an app corresponds to an Activity. Services run in the background without a user interface, ContentProviders store data, and BroadcastReceivers react asynchronously to messages from other applications.

In Android, different components communicate with each other through *Intents*, which are effectively messages that describe operations to be performed. Components can receive intents from other components or from the Android system; thus intents form the basis of *inter-component communication (ICC)* in Android. Intent objects can have several attributes, three of which we discuss below:

- The optional **target** attribute explicitly specifies the specific component invoked by this Intent.
- The **action** attribute specifies the type of action that the receiver of the Intent should perform.
- The **data type** attribute specifies the type of data that the receiver of the intent is supposed to operate on.

An Intent whose target attribute is specified is called an *explicit Intent*. In contrast, an *implicit Intent* does not have its target attribute specified, so the Android system decides the targets of an implicit intent I at run time by comparing I's action and data attributes against the *intent filters* of other components. The intent filter of a component C specifies the action that C is able to perform and the data it can operate on. Such intent filters are declared in the app's *manifest*, which is an XML file containing information about the app.

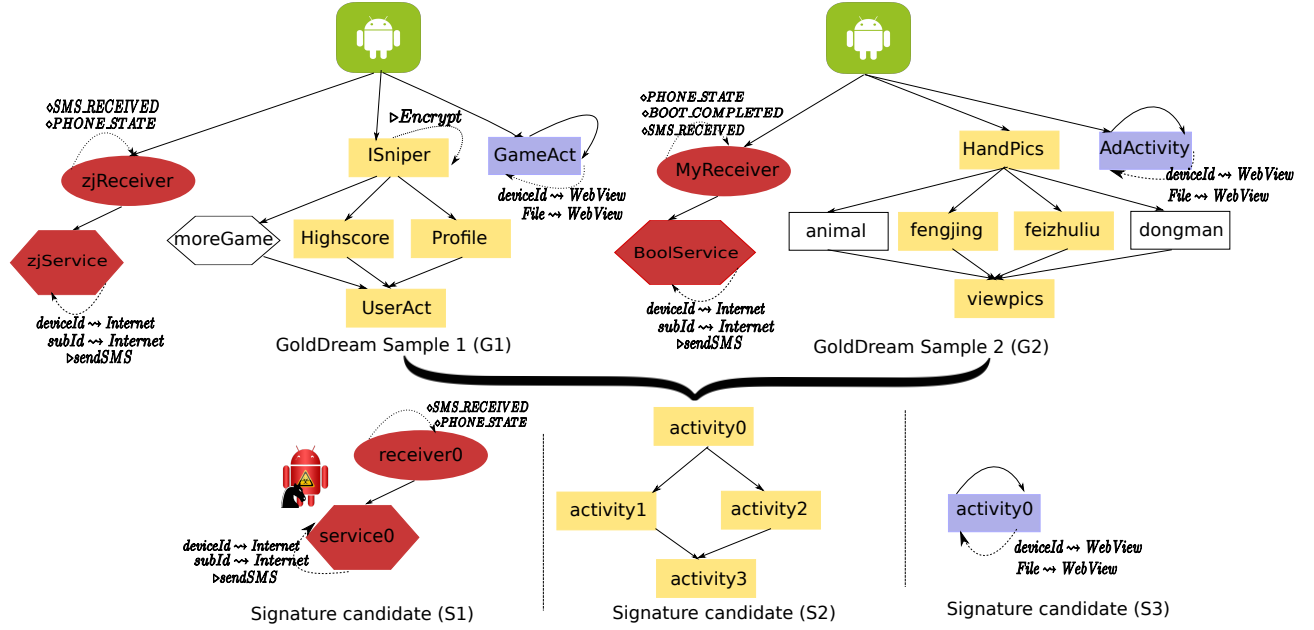[1]ASTROID stands for *Automatic SignaTure findeR for andrOID*

Fig. 1: Motivating example to illustrate our approach

B. Inter-component Call Graphs

Our technique uses the *inter-component call graph (ICCG)* representation introduced in previous work [2]. The ICCG for an Android app summarizes inter-component communication within the app as well as any relevant metadata (e.g., dataflow or API calls). More formally, an ICCG for an application A is a graph (V, X, Y) where:

- V is a set of vertices, where each $v \in V$ is a component of A. The *type* of a vertex v, written $T(v)$, is the type of the component that v represents. Hence, we have:

$$T(v) \in \mathcal{T} = \{activity, service, receiver, provider\}$$

- X is a set of edges representing inter-component call relations. Specifically, $(v, v') \in X$ indicates that component v may invoke component v' either through an explicit or implicit intent.
- Y is a set of labeled edges representing metadata. In particular, $(v, v', d) \in Y$ indicates that components v and v' are related by metadata d.

Metadata edges Y in the ICCG indicate potentially suspicious behaviors of the app. We explain three kinds of metadata that will be used in this paper:

- ***Data flow information***, denoted as *src* $\rightsquigarrow$ *sink*. In particular, $(v, v', s \rightsquigarrow s') \in Y$ indicates that the source s originating from component v flows to the sink s' in component v'. Sources represent confidential information (e.g., IMEI number), and sinks represent externally visible channels (e.g., *Internet*, *SMS*).
- ***Suspicious API calls***, written $\triangleright API$. Specifically, an edge $(v, v, \triangleright f) \in Y$ indicates component v calls Android API method f (e.g., exec).
- ***Suspicious actions***, denoted as $\diamond Act$, represent actions that can be performed by a component. In particular,

$(v, v, \diamond Act) \in Y$ indicates that component Y can perform action *Act*.
- ***Intent filters***, denoted as $\star o$, represent the data type that a given component can operate on. In particular, $(v, v, \star o) \in Y$ indicates that component v can operate on data of type o (passed using intents).

For a given set of apps, the universe of labels from which metadata can be drawn is fixed. In the rest of the paper, we use $\mathcal{D}$ to indicate the domain of edge labels d in Y.

Example 1: Consider the ICCG labeled as *Gold Dream Sample 1* in Figure 1. Here, solid edges represent X, and dashed edges with metadata represent Y. Nodes drawn as rectangles are Activities, while ellipses indicate BroadcastReceivers and hexagons denote Services. Since there is a solid edge from zjReceiver to zjService, broadcast receiver zjReceiver may call zjService. Furthermore, since there is a dashed edge from zjService to itself labeled as deviceId $\rightsquigarrow$ *Internet*, source *deviceId* flows to the Internet within component zjService.

III. OVERVIEW

Suppose that Alice, a security auditor at an anti-virus company, recently learned about a new malware family called GoldDream [16]. Alice would like to update their anti-virus tool to detect GoldDream instances. Alice wants to use a learning-based tool to save work, but then she must manually examine each app flagged by the tool due to high false positive rates (which is time-consuming because the explanations produced by these tools typically do not pinpoint malicious functionalities). Furthermore, the malware family is new, so Alice only has two samples, and current learning-based approaches require many training samples to achieve high accuracy. Defeated, Alice manually writes signatures to detect GoldDream instances by reverse engineering common

3

malicious behaviors from the byte code of the two GoldDream samples, and working to ensure that no benign apps are flagged by her signatures.

ASTROID can greatly benefit Alice by automatically inferring a signature characterizing the GoldDream family from as few as two samples. In the top half of Figure 1, we show the ICCGs of Alice's two GoldDream samples. Observe that these two samples have different component names and perform very different functionalities—the ICCG on the left belongs to a game, while the one on the right belongs to an app for browsing pictures.

To detect the malice shared by these two samples, AS-TROID searches for connected subgraphs of the two ICCGs that are isomorphic to each other. In this case, there are multiple such subgraphs; three are shown in the bottom half of Figure 1. For instance, the red graph labeled *Signature candidate (S1)* is isomorphic to the red subgraphs of $G1$ and $G2$. Similarly, the yellow graph labeled *Signature candidate (S2)* is isomorphic to the yellow subgraphs of $G1$ and $G2$. The key insight in deciding between these candidates is to find the *maximally suspicious common subgraph (MSCS)*. Intuitively, an MSCS is the signature candidate that maximizes the number of metadata edges, where each edge is weighted by its suspiciousness.

Continuing our example, *S2* does not exhibit any "suspicious" behaviors such as suspicious information flows from confidential sources to public sinks. On the other hand, *S1* contains multiple suspicious behaviors, such as calling the *sendSMS* API and leaking confidential data. Based on these suspicious features, ASTROID decides that among the three candidates *S1-S3*, the candidate *S1* most likely encodes the malicious behavior characterizing malware in the GoldDream family.

Internally, ASTROID uses a MaxSAT solver to find MSCSs. Since each suspicious behavior is encoded as a *soft constraint*, an optimal satisfying assignment to the MaxSAT problem corresponds to a maximally suspicious common subgraph of the malware samples. Once ASTROID infers an MSCS of the malware samples, it automatically converts this MSCS into a signature. Hence, ASTROID allows a security auditor like Alice to *automatically* detect future instances of the GoldDream family without having to manually write malware signatures.

IV. SEMANTIC ANDROID MALWARE SIGNATURES

We now formally define our *malware signatures* and state what it means for an app to *match* a signature. Intuitively, a signature for a family $\mathcal{F}$ is an ICCG (V_0, X_0, Y_0) that captures semantic properties common to all malware in $\mathcal{F}$. Ideally, $G_0 = (V_0, X_0, Y_0)$ would satisfy the following:

- G_0 occurs as a subgraph (defined below) of the ICCG $G_S = (V_S, X_S, Y_S)$ of every malware sample $S \in \mathcal{F}$.
- G_0 does *not* occur as subgraph of the ICCG $G_S = (V_S, X_S, Y_S)$ of any sample $S \notin \mathcal{F}$.

By "occurs as a subgraph", we mean there exists an embedding $F_S : V_0 \rightarrow V_S$ such that the following properties hold:

- **One-to-one.** For every $v, v' \in V_0$ where $v \neq v'$, F_S cannot map both v and v' to the same vertex, i.e., $F_S(v) \neq F_S(v')$.

- **Type preserving.** For every $v \in V_0$, F_S must map v to a vertex of the same type, i.e., $T(v) = T(F_S(v))$
- **Edge preserving.** For every $v, v' \in V_0$, F_S must map an edge $(v, v') \in X_0$ to an edge in X_S:

$$(v, v') \in X_0 \Rightarrow (F_S(v), F_S(v')) \in X_S.$$

- **Metadata preserving.** For every $v, v' \in V_0$, F_S must map metadata $(v, v', d) \in Y_0$ to metadata in Y_S:

$$(v, v', d) \in Y_0 \Rightarrow (F_S(v), F_S(v'), d) \in Y_S.$$

Example 2: Consider the candidate signature $S1$ with ICCG (V_0, X_0, Y_0) and the GoldDream sample $G1$ with ICCG (V_1, X_1, Y_1) from Figure 1. Now, let us consider the following candidate embeddings $F_1^{(a)}$ and $F_1^{(b)}$ from V_0 to V_1:

$$\begin{cases} F_1^{(a)}(\texttt{receiver0}) = \texttt{zjReceiver} \\ F_1^{(a)}(\texttt{service0}) = \texttt{zjService} \end{cases}$$

$$\begin{cases} F_1^{(b)}(\texttt{receiver0}) = \texttt{zjReceiver} \\ F_1^{(b)}(\texttt{service0}) = \texttt{GameAct}. \end{cases}$$

Here, candidate $F_1^{(b)}$ is *not* a valid embedding because the types of `service0` and `GameAct` are not compatible, since `service0` is a service whereas `GameAct` is an activity. Thus, $F_1^{(b)}$ does not preserve types. Furthermore, $F_1^{(b)}$ is also invalid for another reason: There is an edge `receiver0` → `service0` $\in X_0$ in $S1$, but there is no corresponding edge `zjReceiver` → `GameAct` in $G1$. Therefore, $F_1^{(b)}$ also does not preserve edges. On the other hand, $F_1^{(a)}$ satisfies all the properties above and is a valid embedding.

Given signature $G_0 = (V_0, X_0, Y_0)$ and app S with ICCG $G_S = (V_S, X_S, Y_S)$, we say that G_0 *exactly matches* (or simply *matches*) S if G_0 occurs as a subgraph of G_S. In other words, given a signature (V_0, X_0, Y_0) and a sample S with ICCG (V_S, X_S, Y_S), we can check whether (V_0, X_0, Y_0) matches S. If so, we have determined that $S \in \mathcal{F}$; otherwise, $S \notin \mathcal{F}$.

In general, our signature may not exactly capture the semantic properties of the malware family $\mathcal{F}$. As a consequence, there may be samples $S \in \mathcal{F}$ such that (V_0, X_0, Y_0) does not match S (called *false negatives*), or samples $S \notin \mathcal{F}$ such that (V_0, X_0, Y_0) matches S (called *false positives*). In practice, a signature should minimize both the false positive rate and the false negative rate, even though there is a tradeoff between optimizing these two values in practice. When detecting malware, we use *approximate matching*, which enables our tool to detect partial matches; tuning the cutoff for what constitutes an approximate match allows us to balance the tradeoff as desired. See Section VII for details.

V. SIGNATURE INFERENCE PROBLEM

Our goal is to *infer* a signature from few samples of a malware family. Suppose we are given n malware samples from a single family $\mathcal{F}$. Naïvely, we can search for any signature that matches each of the n malware samples; then, the resulting signature would intuitively have a low false negative rate. However, even the empty signature fits this criterion (since it can be embedded in any sample), but the empty signature has a high false positive rate.

Instead, we want to maximize the amount of semantic information contained in the signature. More precisely, we seek to find a signature (V_0, X_0, Y_0) that is:

- **A common subgraph:** (V_0, X_0, Y_0) should match each given sample $i \in \{1, ..., n\}$. Intuitively, the common subgraph requirement seeks to minimize false negatives. If the inferred signature S was not a common subgraph of all the samples, then S would not match some samples of $\mathcal{F}$, meaning that we have false negatives.

- **Maximally suspicious:** (V_0, X_0, Y_0) has maximal suspiciousness $|X_0| + \sum_{y \in Y_0} w_y$, where weight w_y indicates indicates the relative importance of metadata edge y. Intuitively, *maximal suspiciousness* seeks to minimize false positives: The less frequently a metadata edge y appears in benign apps (i.e., the more "suspicious" y is), the higher the weight associated with it. Hence, maximizing the suspiciousness score $\sum_{y \in Y_0} w_y$ makes it less likely that a benign app will match the inferred signature. We describe how the weights w_y are chosen in Section VI-C.

In summary, given a set of samples $S_\mathcal{F}$ of malware family $\mathcal{F}$, we refer to the problem of computing a signature (V_0, X_0, Y_0) that is both maximally suspicious and a common subgraph of all $G \in S_\mathcal{F}$ as *signature synthesis*.

VI. SIGNATURE SYNTHESIS AS MAXSAT

As mentioned earlier, our approach reduces the signature synthesis problem to *MaxSAT* [17]. Given a boolean formula in conjunctive normal form (CNF), the MaxSAT problem is to find satisfying assignments to the variables in the formula that maximizes the number of clauses that evaluate to true. For example, given the unsatisfiable formula

$$(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1),$$

the assignment $\{x_0 \mapsto 0, x_1 \mapsto 0\}$ achieves the maximum of three satisfied clauses.

In addition, we can specify that certain clauses *must* be satisfied; these clauses are referred to as *hard constraints* and the remaining clauses are referred to as *soft constraints*. We encode the common subgraph requirement of the signature synthesis problem using hard constraints. In contrast, since the maximally suspicious requirement corresponds to optimizing an objective function, we encode this requirement using soft constraints that should be maximally satisfied.

A. Variables in Encoding

Before we describe how to reduce the signature synthesis problem to MaxSAT, we introduce some terminology and describe the propositional variables used by our encoding.

First, we denote the ICCG of a given sample $i \in \{1, ..., n\}$ of malware family $\mathcal{F}$ as (V_i, X_i, Y_i). Recall that the common subgraph criterion requires that the signature (V_0, X_0, Y_0) matches each i (i.e., we can find isomorphic embeddings $F_i : V_0 \to V_i$). The first difficulty in encoding the common subgraph requirement is that the number of vertices $|V_0|$ in the signature is unknown. However, for each type $r \in \mathcal{T}$, we know that V_0 cannot contain more vertices of type r than any

sample (since it must be embedded in each sample). Hence, we can give an upper bound on the number of vertices of type r in V_0:

$$|\{v \in V_0 \mid T(v) = r\}| \leq m_r.$$

Here, m_r denotes the minimum number of vertices of type r in any sample. This observation immediately gives us an upper bound on the total number of vertices:

$$|V_0| \leq m = \sum_{r \in \mathcal{T}} m_r.$$

Our approach is to fix a large set of vertices V, and then think of V_0 as an unknown subset of V in our encoding. Since there are at most m vertices in the signature, it suffices to take $V = \{v_1, ..., v_m\}$. Furthermore, for each type r, we assign type r to exactly m_r of the vertices in V. Now, we can think of each embedding F_i as a *partial* function $F_i : V \to V_i$, where we require that the domain of F_i is the same for each sample i. Then, V_0 is exactly the *common domain* of the F_i's, and we have $X_0 \subseteq V_0 \times V_0$ as well as $Y_0 \subseteq V_0 \times V_0 \times \mathcal{D}$.

Example 3: Consider the two GoldDream samples shown in Figure 1. The first sample has only 6 activities, 1 service, and 1 receiver, so $m_{\text{activity}} = 6$, $m_{\text{service}} = m_{\text{receiver}} = 1$, and

$$m = m_{\text{activity}} + m_{\text{service}} + m_{\text{receiver}} = 8.$$

Therefore, we take $V = \{v_1, ..., v_8\}$, with 6 activities $v_1, ..., v_6$, 1 service v_7, and 1 receiver v_8. Note that the vertices V_0 used in the candidate signature must be a subset of V, and X_0 must be a subset of $V_0 \times V_0$.

In our description of the variables in our encoding, we use indices $i, j \in \{1, ..., n\}$ to enumerate over the samples and $k, h \in \{0, 1, ..., n\}$ to enumerate over both the samples and the signature. Also, we use the indicator function:

$$\mathbb{I}[\mathcal{C}] = \begin{cases} 1 & \text{if } \mathcal{C} \text{ holds} \\ 0 & \text{otherwise} \end{cases}.$$

For readability, we use the notation $x(a, b, ...)$ to denote a boolean variable $x_{a,b,...}$ indexed over $a \in A$, $b \in B$, and so forth. Our constraint system is defined using the following constants and free variables:

- **Domain indicators.** For every $v \in V$, a boolean variable $d(v)$ indicates whether v is in the domain of the embeddings F_i. These are all free variables.
- **Embedding indicators.** For each $v \in V$,

$$f_i(v, u) = \mathbb{I}[F_i(v) = u].$$

In other words, $f_i(v, u)$ indicates whether the embedding F_i maps v to u. These are all free variables.

- **Type indicators.** For each $v \in V_k$ (or V if $k = 0$) and $r \in \mathcal{T}$, $t_k(v, r)$ indicates whether the type of v is r:

$$t_k(v, r) = \mathbb{I}[T(v) = r].$$

Since the types of all components are known, each $t_k(v, r)$ is a boolean constant.

- **Control-flow indicators.** For each $v, u \in V_k$ (or V if $k = 0$), $x_k(v, u)$ indicates whether there is an edge (v, u) in X_k:

$$x_k(v, u) = \mathbb{I}[(v, u) \in X_k].$$

For $k = 0$, these are free variables; the remainder are constants (since the control-flow edges X_i are known).

- **Metadata indicators:** For every $v, v' \in V_k$ (or V if $k = 0$) and $d \in \mathcal{D}$, $y_k(v, v', d)$ indicates whether there is a metadata $(v, v', d) \in Y_k$:

$$y_k(v, v', d) = \mathbb{I}[(v, v', d) \in Y_k].$$

For $k = 0$, these are free variables; the remainder are constants (since the metadata edges Y_i are known).

Each assignment to the free variables corresponds to a candidate signature together with embeddings F_i into the ICCG of each given sample i.

Example 4: Consider the ICCGs of samples from the GoldDream malware family shown in Figure 1. Recall that, given these samples, our algorithm uses $V = \{v_1, ..., v_8\}$. We first describe the constants constructed by our algorithm, and then the free variables.

The constants include type indicators both for the samples and for the signature, as well as control-flow and metadata indicators for the samples. For the signature, the non-zero type indicators are

$$t_0(v_\ell, \texttt{activity}) = t_0(v_7, \texttt{service})$$
$$= t_0(v_8, \texttt{receiver}) = 1,$$

for $1 \le \ell \le 6$, and the remaining type indicators are zero. For the vertex $\texttt{zjReceiver}$ in first sample G1, the type indicators are:

$$t_1(\texttt{zjReceiver}, \texttt{receiver}) = 1$$

$$t_1(\texttt{zjReceiver}, \texttt{service})$$
$$= t_1(\texttt{zjReceiver}, \texttt{activity})$$
$$= 0$$

For the outgoing edges from $\texttt{zjReceiver}$, the control-flow indicators are

$$x_1(\texttt{zjReceiver}, \texttt{zjService})$$
$$= 1$$

and

$$x_1(\texttt{zjReceiver}, \texttt{zjReceiver})$$
$$= x_1(\texttt{zjReceiver}, \texttt{ISniper})$$
$$= x_1(\texttt{zjReceiver}, \texttt{moreGame})$$
$$= x_1(\texttt{zjReceiver}, \texttt{HighScore})$$
$$= x_1(\texttt{zjReceiver}, \texttt{Profile})$$
$$= x_1(\texttt{zjReceiver}, \texttt{UserAct})$$
$$= x_1(\texttt{zjReceiver}, \texttt{GameAct})$$
$$= 0$$

Assignments to the free variables correspond to candidate signatures along embeddings F_1 and F_2 into the samples G1 and G2 respectively. For example, consider the candidate signature S1, along with the candidate embedding $F_1^{(a)}$ into G1 described in Example 2, and the candidate embedding

$$\begin{cases} F_2^{(a)}(\texttt{receiver0}) = \texttt{MyReceiver} \\ F_2^{(a)}(\texttt{service0}) = \texttt{BoolService} \end{cases}$$

into G2. Then, the domain is represented as $V_0 = \{\texttt{service0}, \texttt{receiver0}\}$, so the assignments to the domain indicators are

$$d(v_\ell) \mapsto 0 \qquad (\text{for } 1 \le \ell \le 6)$$
$$d(v_7) \mapsto 1, \qquad d(v_8) \mapsto 1.$$

The assignments to the embedding indicators for $F_1^{(a)}$ are

$$f_1(v_7, \texttt{zjReceiver}) \mapsto 1, \quad f_1(v_8, \texttt{zjService}) \mapsto 1$$
$$f_1(v_7, w) \mapsto 0 \quad (\text{for } w \ne \texttt{zjReceiver})$$
$$f_1(v_8, w) \mapsto 0 \quad (\text{for } w \ne \texttt{zjService})$$
$$f_1(v_\ell, w) \mapsto 0 \quad (\text{for } 1 \le \ell \le 6 \text{ and } w \in V_1)$$

The assignments to the control-flow indicators are

$$x_0(v_7, v_8) \mapsto 1, \quad x_0(v_\ell, v_{\ell'}) \mapsto 0 \quad (\text{for } (\ell, \ell') \ne (7, 8)).$$

If instead the candidate embedding of the signature S1 into the sample G1 were $F_1^{(b)}$ (also described in Example 2), then the assignments to the embedding indicators would become

$$f_1(v_7, \texttt{zjReceiver}) \mapsto 1, \qquad f_1(v_8, \texttt{GameAct}) \mapsto 1$$
$$f_1(v_7, w) \mapsto 0 \quad (\text{for } w \ne \texttt{zjReceiver})$$
$$f_1(v_8, w) \mapsto 0 \quad (\text{for } w \ne \texttt{GameAct})$$
$$f_1(v_\ell, w) \mapsto 0 \quad (\text{for } 1 \le \ell \le 6 \text{ and } w \in V_1),$$

and assignments to the remaining free variables would be unchanged.

B. Encoding of Common Subgraph

We now describe how to encode the requirement that (V_0, X_0, Y_0) should be a common subgraph of all the samples. Recall from earlier that the common subgraph requirement corresponds to our hard constraints.

- **Constant domain.** For every $v \in V$, the domain of F_i is the same for each i:

$$\bigwedge_i \left\{ d(v) = \bigvee_{w \in V_i} f_i(v, w) \right\}$$

  Here, the conjunction over i states that $d(v)$ should be assigned to true if v is in the domain of the embedding F_i. Therefore, this constraint ensures that each v is either in the domain of F_i for every i, or not in the domain of F_i for any i.

- **Function property.** For every $v \in V$ and $w, w' \in V_i$ where $w \ne w'$, F_i cannot map v to both w and w'.

$$(\neg f_i(v, w)) \vee (\neg f_i(v, w')).$$

- **One-to-one.** For every $v, v' \in V$ where $v \ne v'$ and $w \in V_i$:

$$(\neg f_i(v, w)) \vee (\neg f_i(v', w)).$$

- **Type preserving.** For every $v \in V$, $w \in V_i$, and $r \in \mathcal{T}$, we have:

$$f_i(v, w) \Rightarrow (t_0(v, r) = t_i(w, r)).$$

- **Control-flow preserving.** For every $v, v' \in V$ and $w, w' \in V_i$:

$$f_i(v, w) \wedge f_i(v', w') \wedge x_0(v, v') \Rightarrow x_i(w, w').$$

- **Metadata preserving.** For every $v, v' \in V$ and $d \in \mathcal{D}$, we have:

$$f_i(v, w) \wedge f_i(v, w') \wedge y_0(v, v', d) \Rightarrow y_i(w, w', d).$$

- **No spurious control-flow.** For every $v, v' \in V$, the edges X_0 are a subset of $V_0 \times V_0$:

$$x_0(v, v') \Rightarrow d(v), \quad x_0(v, v') \Rightarrow d(v')$$

- **No spurious metadata.** For every $v, v' \in V$ and $d \in \mathcal{D}$, the metadata Y_0 are a subset of $V_0 \times V_0 \times \mathcal{D}$:

$$y_0(v, v', d) \Rightarrow d(v), \quad y_0(v, v', d) \Rightarrow d(v')$$

Example 5: Consider the ICCGs for the samples G1 and G2 from the GoldDream family shown in Figure 1, along with the candidate signature S1. In Example 4 we described the constants in our MaxSAT encoding of the signature synthesis problem, as well as the assignments to free variables corresponding to the candidate signature S1 together with candidate embeddings $F_2^{(a)}$ and either $F_1^{(a)}$ or $F_1^{(b)}$.

Recall that $F_1^{(a)}$ satisfies the properties described in Section IV (as does $F_2^{(a)}$), whereas $F_1^{(b)}$ is neither type preserving nor edge preserving. In particular, the candidate embedding $F_1^{(b)}$ has corresponding free variable assignments

$$x_0(v_7, v_8) \mapsto 1, \ f_1(v_7, \texttt{zjReceiver}) \mapsto 1,$$
$$f_1(v_8, \texttt{GameAct}) \mapsto 1.$$

These assignments violate the type preservation constraint

$$f_1(v_8, \texttt{GameAct})$$
$$\Rightarrow (t_0(v_8, \texttt{activity}) = t_1(\texttt{GameAct}, \texttt{activity}))$$

because one of these subterms $t_0(v_8, \texttt{activity}) = 0$ but $t_1(\texttt{GameAct}, \texttt{activity}) = 1$, so $F_1^{(b)}$ does not satisfy the type preservation constraints. In addition, these assignments violate the control-flow preservation constraint

$$f_1(v_7, \texttt{zjReceiver}) \wedge f_1(v_8, \texttt{GameAct}) \wedge x_0(v_7, v_8)$$
$$\Rightarrow x_1(\texttt{zjReceiver}, \texttt{GameAct})$$

since the constant $x_1(\texttt{zjReceiver}, \texttt{GameAct}) = 0$, so $F_1^{(b)}$ is not control-flow preserving.

On the other hand, it is not difficult to verify that the candidate signature S1 together with the candidate embeddings $F_1^{(a)}$ and $F_2^{(a)}$ correspond to an assignment that satisfies the constraints described above.

C. Encoding of Maximally Suspicious

Now, we describe how to encode the maximally suspicious requirement, which is that the synthesized signature (V_0, X_0, Y_0) has a maximal suspiciousness $|X_0| + \sum_{y \in Y_0} w_y$, where the weight w_y indicates the relative importance of metadata edge y (as described in Section V). In particular, we maximize the objective

$$\mathcal{O} = \sum_{v, v' \in V} x_0(v, v') + \sum_{v, v' \in V} \sum_{d \in \mathcal{D}} w_{(v, v', d)} y_0(v, v', d).$$

To choose the weights w_y, we use the frequency of the metadata in benign samples to assign weights to different kinds of metadata. In particular, we define w_y to be:

$$w_y = \frac{\#\{\text{benign apps}\} + 1}{\#\{\text{benign apps containing } y\} + 1}$$

In other words, w_y is the inverse frequency with which metadata edge y occurs in benign samples, computed on a large corpus of benign apps (we add one to avoid division by zero). The intuition is that metadata edges in the malware samples that rarely occur in benign apps are more likely to correspond to malicious behaviors.

However, some kinds of behaviors in Y_0 are strictly more dangerous than others. In particular, according to previous literature [3, 7], suspicious intent filters are far more likely to indicate malicious behavior than API calls, which are in turn much more suspect than data-flows:

$$\text{intent filters} \ > \ \text{API calls} \ > \ \text{data-flows}.$$

Rather than simply optimizing the weighted sum, we first prefer signatures that have the highest weighted sum restricted to intent filters, regardless of other metadata. Ties are broken by the weighted sum restricted to suspicious API calls, and further ties are broken by the weighted sum restricted to data-flows. The number of edges in X_0 is considered last, since it is already indirectly optimized by the other objectives. To incorporate this strict ordering on different kinds of metadata, we group the sums in the objective according to the different kinds of metadata and then encode the objective as a lexicographical optimization problem in the MaxSAT solver [18].

VII. Approximate Signature Matching

In addition to making semantic malware detection fully automatic, another benefit of our signature inference algorithm is that it enables *approximate signature matching*. Suppose we are given an app $\mathcal{A}$, and we want to determine if $\mathcal{A}$ is an instance of malware family $\mathcal{F}$. Even if $\mathcal{A}$ does not exactly match $\mathcal{F}$'s signature, we might want to determine if $\mathcal{A}$ exhibits a high degree of similarity with other instances of $\mathcal{F}$. This problem, which we refer to as *approximate matching*, is useful both for detecting zero-day malware and also for mitigating behavioral obfuscation. This section explains how we leverage signature inference for approximate signature matching.

To perform approximate matching between an app $\mathcal{A}$ and a malware family $\mathcal{F}$ with signature $\mathcal{S}_\mathcal{F}$, we first assume that $\mathcal{A}$ and $\mathcal{S}_\mathcal{F}$ belong to the same malware family. We then compute a new signature $\mathcal{S}$ that captures the common behavior of $\mathcal{A}$ and all instances of $\mathcal{F}$. If $\mathcal{S}_\mathcal{F}$ and $\mathcal{S}$ are "similar" (see below), there is a high probability that $\mathcal{A}$ is an obfuscated instance of malware family $\mathcal{F}$.

To understand how we measure similarity, note that $\mathcal{S}$ is always a subgraph of $\mathcal{S}_\mathcal{F}$; hence, we can measure similarity in terms of number of nodes and edges that are removed from $\mathcal{S}_\mathcal{F}$ to form $\mathcal{S}$. Specifically, given signature $\mathcal{S}$, let $f(\mathcal{S})$ be a function that measures the size of S as a weighted sum of the number of nodes and edges in $\mathcal{S}$. Then, given app $\mathcal{A}$ and family $\mathcal{F}$ with signature $\mathcal{S}_\mathcal{F}$, we define the similarity metric as follows:

$$\delta(\mathcal{A}, \mathcal{F}) = \frac{f(\textsc{InferSignature}(\mathcal{A}, \mathcal{S}_\mathcal{F}))}{f(\mathcal{S}_\mathcal{F})}$$

Hence, if $\delta(\mathcal{A}, \mathcal{F})$ is sufficiently high, then $\mathcal{A}$ is more likely to be an instance of family $\mathcal{F}$. As we show in Section IX, approximate matching using this similarity metric between signatures makes ASTROID more resilient to behavioral obfuscation.

Zero-day malware detection. We can also use approximate signature matching to detect zero-day malware. Suppose we have a database of signatures for existing malware families $\mathcal{F}_1, \ldots, \mathcal{F}_n$, and suppose that an app $\mathcal{A}$ does not match any of them. Now, to determine whether $\mathcal{A}$ belongs to a new (unknown) malware family, we compute $\delta(\mathcal{A}, \mathcal{F}_i)$ for each $1 \leq i \leq n$ and report $\mathcal{A}$ as malware if $\delta(\mathcal{A}, \mathcal{F}_i)$ exceeds a certain cutoff value for *some* malware family $\mathcal{F}_i$. We explain how we pick this cutoff value in Section VIII.

VIII. IMPLEMENTATION

We have implemented the proposed technique in a tool called ASTROID for inferring semantic malware signatures for Android. Our implementation consists of about 7,000 lines of Java code and uses the OPEN-WBO MaxSAT solver [19]. Our implementation builds on top of APPOSCOPY [2] for statically constructing ICCGs and taint flows of Android apps. Specifically, APPOSCOPY implements a field- and context-sensitive pointer analysis and constructs a precise callgraph by simultaneously refining the targets of virtual method calls and points-to sets. For context-sensitivity, APPOSCOPY uses the hybrid approach proposed in [20]. Specifically, it uses call-site sensitivity for static method calls and object-sensitivity for virtual method calls. To scale in large apps, APPOSCOPY also leverages the EXPLORER [21] tool to construct ICCGs in a demand-driven manner. APPOSCOPY's average static analysis time for constructing an ICCG is 87 seconds for an app from the Android Genome benchmarks and 126 seconds for an app from Google Play, including analysis time for all third-party libraries. Unlike DroidSafe [22] which analyzes the source code of Android framework directly, APPOSCOPY uses about 1,210 manually-written models for classes that are relevant to its underlying taint analysis.

Once a signature for a given malware family is generated, ASTROID can perform both exact matching (described in Section IV) as well as approximate matching (described in Section VII). Our implementation of the approximate matching algorithm differs slightly from the description in Section VII; in particular, we use the transitive closure of the control-flow edges in the ICCG (computed in a preprocessing step) rather than the original ICCG control-flow edges; this modification increases the resilience of ASTROID against obfuscations that introduce dummy components.

We use approximate matching both to detect obfuscated apps and to detect zero-day malware. Recall from Section VII that our approximate matching algorithm uses a cutoff value for how high the similarity metric must to count as a match. We choose a cutoff of 0.5 for zero-day malware, and a stricter cutoff of 0.8 for obfuscated malware. In other words, an app with a similarity score = 1.0 is flagged as an unobfuscated instance of a known malware family, an app with similarity score > 0.8 is flagged as an obfuscated instance of a known
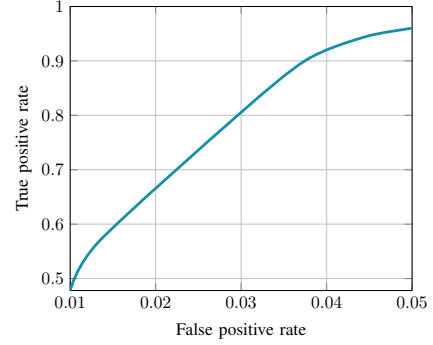

Fig. 2: Effect of varying cutoff values on false positives and false negatives for zero-day malware detection

malware family, and an app with similarity score > 0.5 is flagged as a possible zero-day malware.

We describe our methodology for choosing the similarity metric cutoff for zero-day malware detection; the cutoff for obfuscated malware is chosen similarly. We use the Android Malware Genome Project as a training set. For each family $\mathcal{F}$ in the Android Malware Genome Project, we omit the signature for $\mathcal{F}$ from ASTROID's database of signatures (so samples in $\mathcal{F}$ appear as zero-day malware to ASTROID). Then, we use the remaining signatures in the database to try and detect samples in $\mathcal{F}$ using our zero-day malware detection algorithm in Section VII, using each cutoff value 0.6382, 0.5834, 0.5832, 0.4927, and 0.4505 (selected by manually binning the data).

For each cutoff value and each $\mathcal{F}$, we compute the true positive rate (i.e., the fraction of samples in $\mathcal{F}$ that ASTROID detects); then, for each cutoff value, we take the weighted average over families $\mathcal{F}$ to obtain an overall true positive rate for that cutoff value. Finally, we select the largest possible cutoff that still achieves a 90% true positive rate. The selected cutoff is 0.4927, which we round to 0.5. Figure 2 shows the ROC curve obtained using the various cutoff values, with the y-axis showing the computed true positive rate. The x-axis shows the corresponding false positive rate, which is computed on an independent test set of benign apps.

IX. EVALUATION

The goal of our evaluation is to answer the following questions:

Q1. How do the signatures synthesized by ASTROID compare with manually-written signatures used for evaluating AP-POSCOPY in terms of precision and accuracy?
Q2. How do the quality of learned signatures improve as we increase the number of samples?
Q3. How does ASTROID compare against other state-of-the-art malware detectors?
Q4. How effective is ASTROID at detecting zero-day malware?
Q5. How resistant is ASTROID to behavioral obfuscation?

In what follows, we describe a series of five experiments designed to answer these questions. All experiments are conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 32G of memory running on Ubuntu 14.04.

Fig. 3: Accuracy of APPOSCOPY with a manual and an automated signature synthesized by ASTROID

Id	Malware Family	#Samples
A	DroidKungFu	444
B	AnserverBot	184
C	BaseBridge	121
D	Geinimi	68
E	DroidDreamLight	46
F	GoldDream	46
G	Pjapps	43
H	ADRD	22
I	jSMSHider	16
J	DroidDream	14
K	Bgserv	9
L	BeanBot	8
M	GingerMaster	4

A. Astroid vs. Manual Signatures

In our first experiment, we compare the signatures synthesized by ASTROID with manually written signatures used for evaluating APPOSCOPY [2]. Since ASTROID generates semantic malware signatures in the specification language used in APPOSCOPY, we use APPOSCOPY's (exact) signature matching algorithm to decide if an app matches a signature. According a co-author of [2], it took *several weeks* to manually construct signatures for all malware families in their dataset, even with full knowledge of the malware family of each app and the nature of the malice.

Accuracy on known malware. Since the authors of [2] have manually written semantic signatures for all malware families from the Android Malware Genome Project [23], we use ASTROID to synthesize signatures for malware families from this dataset. The table in Figure 3 shows the malware family names and the number of samples for each family. For each family $\mathcal{F}$, we randomly select 5 samples from $\mathcal{F}$ and use ASTROID to infer a signature for $\mathcal{F}$ from these samples. Since different sets of samples may produce different signatures, we run ASTROID with 11 different sets of randomly chosen samples, and report our results for the signature that achieves the median accuracy out of these 11 runs.

The plot in Figure 3 compares the accuracy ASTROID against APPOSCOPY. Here, accuracy is the number of correctly classified samples divided by the total number of samples. For most malware families, ASTROID yields similar or better accuracy than APPOSCOPY, and the overall accuracy of ASTROID (93.8%) is higher than that of APPOSCOPY (90%). Furthermore, for two malware families (specifically, Pjapps and BaseBridge), ASTROID generates significantly better signatures than manually written ones. In summary, these results show that ASTROID can achieve *significantly fewer false negatives* compared to APPOSCOPY.

False positives on known malware. ASTROID reports zero false positives on the Android Malware Genome Project samples. In contrast, APPOSCOPY reports two such false positives (specifically, it misclassifies two instances of other malware families as belonging to Geinimi). Hence, the signatures synthesized by ASTROID outperform manually written ones, both in terms of accuracy as well as precision.

False positives on benign apps. To determine whether ASTROID produces false positives on benign apps, we analyze a corpus of 10,495 apps downloaded from Google Play during 2013-14. According to [7], over 99.9% of these apps are known to be benign. ASTROID reports a total of 8 malicious apps (specifically, 2 instances each of DroidDream and DroidDreamLight, 1 instance of Pjapps, and 3 instances of DroidKungFu). We uploaded these 8 apps to VirusTotal [24], which is a service that provides aggregated reports from multiple anti-virus tools. VirusTotal agreed with ASTROID on each of these 8 apps.[2] In other words, ASTROID reported *zero false positives* of either kind.

B. Varying the Number of Samples

We evaluate the performance and accuracy of ASTROID with respect to the number of samples. In Figure 4, the blue line plots signature inference time (in seconds) against the number of samples. ASTROID is quite fast, taking less than two seconds on average to infer a signature for a malware family. Since manually writing malware signatures typically requires *several hours* of human effort, we believe these statistics show that ASTROID is quite practical.

The bars in Figure 4 show how the accuracy varies with respect to the number of samples. ASTROID achieves 90.8% accuracy using *only two samples*. When we increase the number of samples to 5, ASTROID achieves 93.8% accuracy. More samples do not seem to benefit, likely due to imprecision in the static analysis for ICCG constructions. These results

---

[2]To account for the inclusion of some lower-quality AV tools in VirusTotal results, we only consider VirusTotal to report an app as malware if the majority (i.e., more than half) of AV tools classify the app to be malicious.
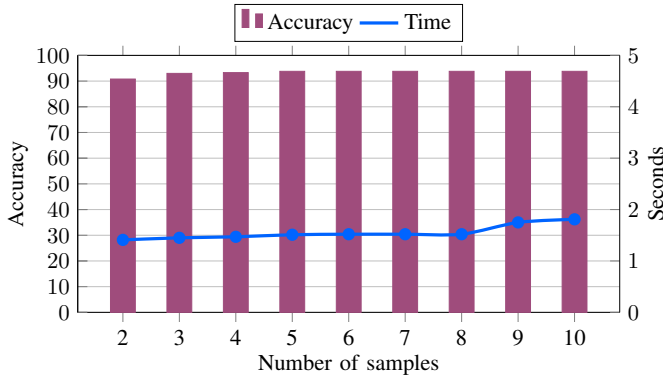
Fig. 4: Accuracy and time to synthesize the signature with different number of samples

Fig. 5: Comparison between DREBIN with FP0.01% and FP1%, MASSVET and ASTROID

show that ASTROID can infer very good signatures from a very small number of malware samples.

C. Comparison Against Existing Tools

We compare ASTROID to two state-of-the-art malware detection tools, namely DREBIN [3] (a learning-based malware detector) and MASSVET [15] (which detects repackaged malware). We do not explicitly compare ASTROID against anti-virus tools (e.g., VirusTotal) because ASTROID is an extension of APPOSCOPY, which has already been shown to outperform leading anti-virus tools in the presence of obfuscated malware.

Comparison to Drebin. We first compare ASTROID against DREBIN [3], a state-of-the-art malware detector based on machine learning. Since our goal is to infer malware signatures using few samples, we focus on the setting where only five samples are available for the target family. We believe this setting is very important because many malware families have very few known samples. For example, in the Drebin dataset, only 42% of families have more than 5 samples.

DREBIN takes as input a set of apps and extracts features using static analysis. It then trains an SVM on these feature vectors to classify apps as malicious or benign. Unlike ASTROID, which classifies each app into a specific malware family, DREBIN only determines whether an app is malicious or benign. Since the implementation of DREBIN is unavailable, we used their publicly available feature vectors and train SVMs based on the methodology described in [3] using the libsvm library [25]. For each malware family $\mathcal{F}$, we start with the full DREBIN training set, but remove $\mathcal{F}$ except for 5 randomly chosen samples from $\mathcal{F}$ (to achieve our small-sample setting). In summary, we train DREBIN on a large number of benign apps, *all* samples from other malware families, and 5 samples from the target family. As with ASTROID, we train the SVM with 11 different sets of randomly chosen samples and report the median accuracy.

In order to perform an apples-to-apples comparison, we use the 1,005 samples in the Android Malware Genome Project
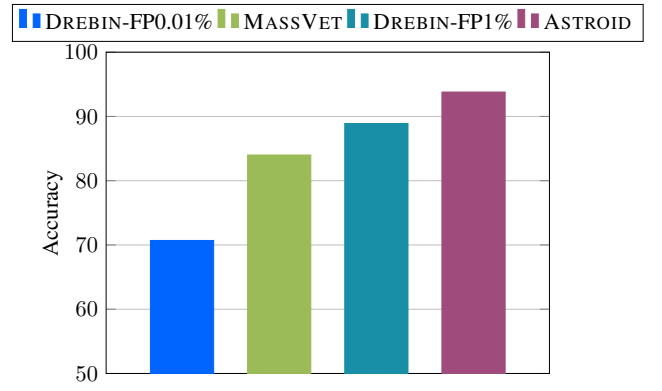
for which DREBIN feature vectors are available.[3] We use two cutoffs to evaluate the accuracy on these apps. The first achieves a false positive rate of 1%, which is used in the original evaluation [3]. However, 1% false positives still amounts to more than 10,000 false positives on Google Play, which contains more than a million apps. For a closer comparison to ASTROID (which reports zero false positives on Google Play apps), we choose a second cutoff that achieves a false positive rate of 0.01%.

As shown in Figure 5, ASTROID outperforms DREBIN in our small-sample setting: While ASTROID achieves 93.8% accuracy, DREBIN yields 70.7% and 88.9% accuracy using the 0.01% and 1% cutoff values respectively. We believe these results indicate that ASTROID compares favorable with existing learning-based detectors in the small-sample setting.

Comparison to MassVet. We also compare ASTROID against MASSVET [15], a state-of-the-art tool for detecting repackaged malware—i.e., malware produced by adding malicious components to an existing benign app. MASSVET maintains a very large database of existing apps and checks whether a given app is a repackaged version of an existing one in the database. The authors of MASSVET provide a public web service [26], which we use to evaluate MASSVET.

As shown in Figure 5, the overall accuracy of MASSVET on the Android Malware Genome Project is 84.0%, which is significantly lower than the 93.8% accuracy of ASTROID. These false negatives occur because many malware samples are not repackaged versions of existing benign apps, and even repackaged malware may go undetected if the original app is missing from their database.

We also evaluate the false positive rate of MASSVET on a corpus of 503 benign apps from Google Play. [4] MASSVET reports that 176 of these apps are malware, but all except one

---

[3]In this experiment, we use the Android Malware Genome Project dataset as opposed to the DREBIN dataset because some of the malware families in the DREBIN dataset are mislabeled, which is problematic for multi-label classification (as done by our technique). For example, some of the family labels in the Drebin dataset do not match the labels in the Malware Genome project.

[4]We were unable to apply MASSVET to all 10,495 apps from Google Play due to limitations of the web service.

of these apps are classified as benign by VirusTotal. Since several benign applications repackage existing apps by adding ad libraries, MASSVET seems to mistakenly classify them as malware. In summary, this experiment demonstrates that ASTROID achieves both higher accuracy as well as a lower false-positive rate compared to MASSVET.

Interpretability. In addition to comparing favorably with MASSVET and DREBIN in terms of accuracy and false positives, we believe that ASTROID produces better explanations of malice compared to existing tools. In the Appendix, we compare the semantic signatures produced by ASTROID with the evidence of malice produced by DREBIN and MASSVET. Since ASTROID can pinpoint the malicious components and their suspicious metadata (e.g., sensitive information leaked by the component), we believe that the signatures generated by ASTROID are more interpretable (and therefore more helpful) to security analysts.

D. Detection of Zero-day Malware

To evaluate whether ASTROID can effectively detect zero-day malware, we conduct experiments on two different datasets and evaluate ASTROID's accuracy and false positive rate.

Malware from Symantec and McAfee. In our first experiment, we use 160 malware samples obtained from Symantec and McAfee, two leading anti-virus companies. Even though these applications are known to be malicious, none of them belong to the malware families from the Android Malware Genome Project dataset. Since ASTROID's database only contains signatures for the families shown in Figure 3, all of these 160 applications are zero-day malware with respect to ASTROID's signature database. Using ASTROID's approximate signature matching algorithm (with the cutoff of 0.5, as described in Section VIII), ASTROID correctly identifies 147 of these apps as malware; hence, ASTROID's accuracy for zero-day malware detection on this dataset is 92%. In contrast, MASSVET's accuracy on this dataset is 81%. We were not able to compare against DREBIN on this dataset because we do not have their feature vectors available.

Apps from Google Play. For our second experiment, we analyze 10,495 Google Play apps using ASTROID's approximate signature matching algorithm. As before, ASTROID's signature database only contains the malware families from Figure 3. Among these apps, ASTROID reports 395 of them (i.e., 3.8%) as being malicious. Of these 395 apps, 8 exactly match one of the signatures in ASTROID's database; as discussed in Section IX-A, these 8 apps are all malicious.

To investigate which of the remaining 387 apps are known malware, we use VirusTotal to analyze each of these apps. Among these 387 samples, 21 of them are reported as malicious by the majority (i.e., more than half) of the anti-virus tools, and 81 of them are reported as malicious by at least one anti-virus tool. Of the remaining 306 apps reported by ASTROID, we randomly selected 40 apps and manually inspected them. Our manual inspection shows that 22 of these 40 apps are actually malicious since they contain highly suspicious behaviors:

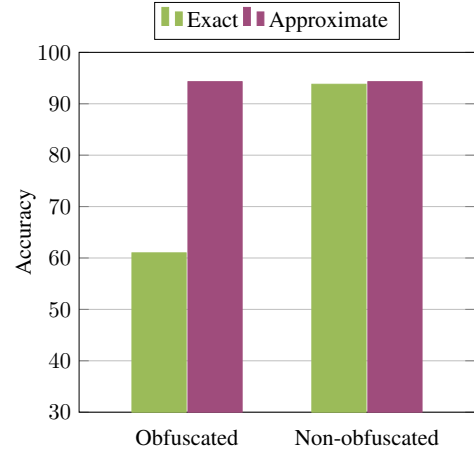- 4 apps appear to be SMS Trojans because they automatically block all incoming SMS events when they receive

Fig. 6: Evaluation of different matching techniques

an SMS at a specific time or the contents of the SMS match a certain pattern.
- 1 app silently records audio in a background process and aborts all incoming SMS events.
- 1 app automatically locks the screen and prevents the user from unlocking it.
- 11 apps dynamically install apk files from untrusted sources and leak the user's device id, IMSI number, and other confidential phone information to untrusted remote servers.
- 1 app silently takes pictures (i.e., without the user triggering it) and saves the picture to an encrypted file.
- 2 apps contain Android.Cauly library, which has recently been classified as PUA ("Potentially Unwanted App") by Symantec.
- 2 apps perform highly suspicious actions without the user triggering them. For instance, they send email or SMS messages to a fixed address or phone number.

Hence, our manual inspection shows that ASTROID can detect malicious apps that are not identified by existing anti-malware tools. Furthermore, based on our manual inspection, we estimate that of the 306 apps not identified as malicious by existing tools, 55% are in fact malicious. By this estimate, ASTROID's false positive rate is approximately 1.3% for zero-day malware detection (i.e., using approximate matching with a cutoff of 0.5).

Finally, we remark that the partial match (i.e., the subgraph INFERSIGNATURE($\mathcal{A}, \mathcal{S}_\mathcal{F}$) of the ICCG of $\mathcal{A}$) computed by our approximate matching algorithm was indispensable for finding the malicious behaviors in the 40 randomly selected apps. In every case, some part of malicious code was contained in the partial match, allowing us to quickly identify the malicious behavior. In particular, examining all 40 apps took a single analyst only a few hours. As discussed earlier, the interpretability of the inferred signatures is an important feature of ASTROID.

E. Detection of Obfuscated Apps

To evaluate whether ASTROID is resilient to obfuscations, we perform a combination of low-level (syntactic) and high-level (semantic) obfuscations. First, we obfuscate existing

malware using the ProGuard tool [27] to rename method/class names, encrypt strings, and modify the program's control flow. Second, we also perform obfuscations at the ICCG level, such as inserting dummy components and removing taint flows.

Figure 6 shows the accuracy of signature matching using exact vs. approximate matching for 1,025 malware samples from the Android Malware Genome Project and their corresponding obfuscated versions. When using the exact matching algorithm of APPOSCOPY, ASTROID can detect 93.8% of the non-obfuscated malware samples but only 61% of the obfuscated malware samples. In contrast, when we use the approximate matching algorithm (with the cutoff of 0.8, as described in Section VIII), ASTROID is able to detect 94.3% of malware samples for both obfuscated and non-obfuscated versions. Hence, this experiment demonstrates that ASTROID's approximate signature matching algorithm significantly increases the resilience of ASTROID to behavioral obfuscations. In contrast, other anti-virus tools (e.g., Symantec, McAfee, Kaspersky etc.) have been shown to be much less resilient to even a subset of the obfuscations that we employ in this experiment [2].

False positives. Both exact and approximate matching report zero false positives on malware samples. As before, with exact matching, ASTROID reports that 8 apps are malicious from the corpus of 10,495 Google Play apps, but all of these apps are classified as malware by VirusTotal. When using approximate matching, ASTROID reports a total of 13 apps as malicious, 9 of which are classified as malware by VirusTotal. Hence, the false positive rate of ASTROID remains very low (<0.04%) even with approximate matching.

X. LIMITATIONS

Like any other malware detection tool, ASTROID has a number of limitations:

First, the quality of the signatures inferred by ASTROID depends on the precision of the underlying static analysis used to construct the ICCG of the samples. In particular, sources of imprecision (or unsoundness) in the analysis can degrade the quality of the signatures inferred by ASTROID. However, our experiments indicate that ASTROID can synthesize high-quality signatures despite possible imprecision in the static analysis.

Second, ASTROID's signature matching algorithm (both the exact and approximate variants) are also affected by the quality of the underlying static analysis. Since signature matching requires computing the ICCG of the application under analysis, any source of unsoundness in the analysis may translate into false negatives in the context of malware detection. For example, if an app dynamically loads a malicious payload, then ASTROID may fail to flag it as malware. However, such attempts to escape detection can be identified as suspicious, thereby requiring further scrutiny. On the other hand, sources of imprecision due to the underlying static analysis may also translate into false positives. For instance, given a benign app, if the analysis generates a lot of spurious taint flows and inter-component call edges, ASTROID may mistakenly mark it as malware. However, our evaluation show that the underlying static analysis achieves a high precision without sacrificing scalability.

Third, ASTROID requires an analyst to provide at least two representative samples of a given malware family, so there is still a minimal amount of human effort involved in using ASTROID. However, we believe this effort is miniscule compared to the laborious task of writing malware signatures manually.

Finally, ASTROID's accuracy in detecting malware is dependent on the initial database of malware signatures. The larger the database, the higher the accuracy in detecting existing and zero-day malware. However, our experiments show that ASTROID can effectively detect new malware families even though we added only thirteen malware signatures to its database. Furthermore, we believe that it is easier to maintain a database of malware signatures compared to the task of maintaining a much larger database of individual malware samples.

XI. RELATED WORK

Android malware detection and classification have been extensively studied in recent years. In this section we briefly discuss prior closely-related work.

Machine learning approaches. Most prior malware classification techniques are based on machine learning [3, 6, 11, 12, 4, 13, 14, 5]. The key idea underlying all ML-based approaches is to extract a feature vector summarizing the application's behavior and then train a classifier to predict whether an app is malicious or benign.

Many ML-based approaches generate their feature vectors from a graph representation of the program. For instance, SMIT [11] and Gascon et al. [12] model programs using callgraphs and use clustering algorithms (e.g., KNN) to group similar applications.

Similar to SMIT, both MASSVET [15] and DROIDSIFT [4] use a graph abstraction to represent Android apps. Specifically, MASSVET [15] computes graph similarity between a given app $\mathcal{A}$ and the database of existing apps to determine if $\mathcal{A}$ is the repackaged version of an existing app. As we show in our evaluation, ASTROID can achieve better precision and fewer false positives compared to MASSVET. Similarly, DROIDSIFT abstracts programs using an *API dependency graph* and employs graph similarity metrics to cluster applications into different malware families. Similar to ASTROID, DROIDSIFT performs multi-label rather than binary classification and employs a semantics-based approach to resist obfuscation. However, unlike DROIDSIFT, ASTROID can infer signatures from very few samples and does not need a large training set. We tried to compare ASTROID against DROIDSIFT but we were not able to reproduce their results using DROIDSIFT's web service [28]. While we have contacted the authors, the issues with DROIDSIFT's web service have not been resolved by the time of this submission.

Another related tool is HOLMES [6], which detects Windows malware by constructing the program's *behavior graph*. Such behavior graphs are constructed dynamically by analyzing data dependencies in program traces. Given a behavior graph, HOLMES then uses *discriminative subgraph mining* to extract features that can be used to distinguish malicious applications from benign ones. DROIDMINER [14] and Bose

et.al [5] also use a variant of behavior graphs to abstract malware as a set of malicious components and use machine learning for classification. In contrast to these these techniques which require a large number of samples (e.g., HOLMES uses 492 samples in their evaluation), ASTROID requires as few as two samples to automatically generate malware signatures. Second, our constraint-based approach is guaranteed to generate the optimal signature (in terms of maximizing suspicious behaviors). Finally, we believe that the semantic signatures synthesized by ASTROID are easier for security analysts to interpret.

Another state-of-the-art malware detector for Android is DREBIN [3], which combines syntactic and semantic features into a joint vector space. The syntactic features are obtained from the application's manifest file, while the semantic features are obtained through static analysis. Some of the features used by DREBIN are similar to ASTROID; for instance, DREBIN also extracts data flow information as well as suspicious API calls. As we demonstrate experimentally, ASTROID can achieve the same or better precision as DREBIN using much fewer samples.

Signature-based approaches. As mentioned earlier, signature-based approaches look for explicit patterns to identify instances of a malware family [8, 2, 9, 10, 29, 30, 31]. These signatures can be either syntactic or semantic, or a combination of both. Our approach extends the applicability of signature-based detectors by automatically synthesizing signatures from a handful of malware samples.

Among signature-based detectors, APPOSCOPY [2] uses semantic signatures to identify Android malware. Specifically, it performs a combination of static taint analysis and inter-component control flow analysis to determine whether a signature matches an Android application. As mentioned earlier, ASTROID is integrated as a plug-in to APPOSCOPY and can generate signatures in APPOSCOPY's malware specification language.

KIRIN [8], which is another signature-based tool, leverages Android permissions to detect malware. Specifically, KIRIN uses permission patterns to perform binary classification of apps as benign vs. malicious. While FACT [29] obtains a signature by computing the unweighted maximal common subgraph extracted from dynamic analysis, ASTROID computes a weighted *maximally suspicious common subgraph* using static analysis.

Zero-day malware detection. Zero-day malware detectors [32, 7, 15] can detect malicious applications that belong to new malware families. For instance, RISKRANKER [32] ranks Android applications as high-, medium-, or low-risk depending on the presence of suspicious features, such as certain kinds of function calls. As another example, DROIDRANGER [7] uncovers zero-day malware by performing heuristic-based filtering to identify suspicious behaviors. Some ML-based approaches (e.g., [3], [5]) can, in principle, also uncover zero-day malware, even though their detection rate is much higher for instances of known malware families. Even though the primary goal of ASTROID is not zero-day malware detection, our experiments in Section IX-D show that ASTROID can nonetheless be successfully used to detect instances of unknown malware families.

Information flow analysis for Android. Several tools, including ASTROID, use information flow as a component of malware signatures or feature vectors. While information flow does not directly predict malware, ASTROID can benefit from recent advances in information flow analysis to improve the quality of its signatures. Some examples of Android information flow analysis tools include FLOWDROID [33], APPINTENT [34], APPAUDIT [35], TAINTDROID [36], DROIDSCOPE [37], CHEX [38], EPICC [39], HI-CFG [40] and APPCONTEXT [41].

XII. CONCLUSION

We have presented a new technique for automatically inferring interpretable semantic malware signatures from a small number of malware samples. Our technique significantly improves the usability of signature-based malware detectors by eliminating the human effort required for writing malware signatures. Furthermore, we show that ASTROID s signature inference algorithm enables approximate signature matching, which is useful both for zero-day malware detection and for making our technique more resilient to behavioral obfuscation.

We implemented our technique in a tool called ASTROID, which we evaluated both on malicious apps in the Android Genome Malware Project as well as on benign apps from Google Play. Our experiments show that (i) the signatures automatically synthesized by ASTROID are better than manually-written signatures in terms of accuracy and false positives, and (ii) the proposed approximate signature matching algorithm allows detecting zero-day and behaviorally- obfuscated malware with a very low false positive rate. Our tool is publicly available [42] and can be easily used by security analysts to synthesize malware signatures from very few samples.

REFERENCES

[1] Mobile malware report. https://public.gdatasoftware.com/Presse/ Publikationen/Malware_Reports/G_DATA_MobileMWR_Q1_ 2015_US.pdf, 2015.

[2] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, pages 576–587. ACM, 2014.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*. The Internet Society, 2014.

[4] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *CSS*, pages 1105–1116. ACM.

[5] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *MobiSys*, pages 225–238. ACM, 2008.

[6] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *Malware*, pages 41–50. IEEE Computer Society, 2010.

[7] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*. The Internet Society, 2012.

[8] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. On lightweight mobile phone application certification. In *CSS*, pages 235–245. ACM, 2009.

[9] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. Automatic generation of string signatures for malware detection. In *RAID*, pages 101–120. Springer, 2009.

[10] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *S&P*, pages 32–46. IEEE Computer Society, 2005.

[11] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *CSS*, pages 611–620. ACM, 2009.

[12] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *AISEC*, pages 45–54. ACM, 2013.

[13] Hao Peng, Christopher S. Gates, Bhaskar Pratim Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *CSS*, pages 241–252. ACM, 2012.

[14] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip A. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *ESORICS*, pages 163–182. Springer, 2014.

[15] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *USENIX Security*, pages 659–674. USENIX Association, 2015.

[16] Xuxian Jiang. Security alert: New Android malware – GoldDream– found in alternative app markets. http://www.csc.ncsu.edu/faculty/jiang/GoldDream/, 2011.

[17] Chu Min Li and Felip Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, volume 185, pages 613–631. IOS Press, 2009.

[18] Joao Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[19] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A Modular MaxSAT Solver. In *SAT*, pages 438–445. Springer, 2014.

[20] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434. ACM, 2013.

[21] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *OOPSLA*, pages 520–534. ACM, 2015.

[22] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*. The Internet Society, 2015.

[23] Android malware genome project. http://www.malgenomeproject.org/, 2012.

[24] VirusTotal. https://www.virustotal.com/en/, 2016.

[25] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2, 2011.

[26] MassVet. https://bdsec.soic.indiana.edu:8080/, 2016.

[27] Proguard. https://www.guardsquare.com/en/proguard, 2016.

[28] DroidSift. https://haven.syr.edu:3000/, 2016.

[29] Young Hee Park, Douglas S. Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *CSIIRW*, pages 45–48. ACM, 2010.

[30] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *FSE*, pages 5–14. ACM, 2007.

[31] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Josef Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring Knowledge at Scale. In *SNAPL*, pages 63–78. LIPICS, 2015.

[32] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys*, pages 281–294. ACM, 2012.

[33] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269. ACM, 2014.

[34] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and Xiaoyang Sean Wang. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *CSS*, pages 1043–1054. ACM, 2013.

[35] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *S&P*, pages 899–914. IEEE Computer Society, 2015.

[36] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407. USENIX Association, 2010.

[37] Lok-Kwong Yan and Heng Yin. Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, pages 569–584. USENIX Association, 2012.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *CSS*, pages 229–240. ACM, 2012.

[39] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security*, pages 543–558. USENIX Association, 2013.

[40] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. HI-CFG: construction by binary analysis and application to attack polymorphism. In *ESORICS*, pages 164–181. Springer, 2013.

[41] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *ICSE*, pages 303–313. IEEE Computer Society, 2015.

[42] Astroid. https://utopia-group.github.io/astroid/, 2016.

[43] Smart phone malware: The six worst offenders. http://tinyurl.com/huaor8y, 2011.

[44] Fake android apps. http://us.norton.com/fake-android-apps/article, 2016.

APPENDIX

APPENDIX A: INTERPRETABILITY OF EXPLANATIONS

In this section, we describe the explanations for the Gold-Dream malware family generated by ASTROID, DREBIN and MASSVET, which are shown in Figure 7. In particular, we

$$V = \{r : \text{receiver}, s : \text{service}\}$$
$$X = \{(\text{SYSTEM}, r), (r, s)\}$$

$$Y = \left\{\begin{array}{l}
(\text{SYSTEM}, r, \text{IntentFilter}(\texttt{BOOT\_COMPLETED})), \\
(\text{SYSTEM}, r, \text{IntentFilter}(\texttt{SMS\_RECEIVED})), \\
(\text{SYSTEM}, r, \text{IntentFilter}(\texttt{PHONE\_STATE})), \\
(\text{SYSTEM}, r, \text{IntentFilter}(\texttt{NEW\_OUTGOING\_CALL})), \\
(s, s, \text{SuspiciousAPI}(\texttt{sendTextMessage})), \\
(s, s, \text{TaintFlow}(\text{DeviceID}, \text{Internet})), \\
(s, s, \text{TaintFlow}(\text{SubscriberID}, \text{Internet})), \\
(s, s, \text{TaintFlow}(\text{File}, \text{Internet})), \\
(s, s, \text{TaintFlow}(\text{SimSerial}, \text{Internet}))
\end{array}\right.$$

SuspiciousAPI(`sendSMS`) (1.07)
NetworkAddress(`lebar.gicp.net`) (0.93)
Permission(`DELETE_PACKAGES`) (0.58)
IntentFilter(`SMS_RECEIVED`) (0.56)
SuspiciousAPI(`getSubscriberID`) (0.53)

onReceive → `createFromPdu`
onReceive → `getOriginatingAddress`
onReceive → `getDisplayMessageBody`

| ASTROID | DREBIN | MASSVET |

Fig. 7: Explanations produced by each tool for the GoldDream family.

discuss how a security analyst might use these explanations to pinpoint and understand the malicious behaviors of Gold-Dream malware.

The left-hand side of Figure 7 shows the signature synthesized by ASTROID for the GoldDream malware family. This signature conveys a wealth of information about the malware family to the auditor:

Components. Simply by looking at the vertices V in the signature, the auditor sees that the malware consists of two components: a receiver r and a service s. Furthermore, the inter-component call relations X convey that r is called by the Android framework (denoted by the vertex SYSTEM), and s is subsequently called by r.

Triggers. The metadata Y encodes intent filters registered by each receiver. For the GoldDream family, the receiver r can be triggered by a variety of common system events including BOOT_COMPLETED (triggered when the app starts) and SMS_RECEIVED (triggered when an SMS message is received).

Malice. The metadata Y also encodes malicious behaviors associated with the relevant components. For the GoldDream family, this includes suspicious API calls (e.g., the service s calls sendTextMessage) and information leaks (e.g., the service s leaks the device ID to the Internet).

In summary, the signature inferred by ASTROID encodes that members of the GoldDream family contain a receiver triggered by common system events, and this receiver calls a service that leaks sensitive information to the Internet.

Unlike ASTROID, DREBIN and MASSVET do not characterize the malice corresponding to a particular malware family. Instead, they produce explanations for why a specific app might be malicious.

Comparison to Drebin. The explanation produced by DREBIN consists of a list of the features most indicative of malicious behavior together with weights indicating their relative significance. Figure 7 (middle) shows this list of top features and corresponding weights for the GoldDream family (obtained by averaging over all members of the family). The most significant feature is the call to the suspicious API sendSMS. Only 6.6% of benign apps in the Drebin dataset call sendSMS, but 92.8% of GoldDream malware make this call. Therefore, calling sendSMS is a good statistical signal that an app is malicious, but it does not give conclusive evidence of malice. Unlike ASTROID, DREBIN fails to pinpoint any malicious components, intent filters, and information leaks, let alone the complex relationships between these entities.

The presence of the network address lebar.gicp.net might be a more conclusive signal of malice, but if this feature were used to filter apps on Google play, then malware developers would quickly learn to obfuscate it. In contrast, ASTROID solely relies on semantic features of apps that are significantly harder to obfuscate.

Comparison to MassVet. The explanation produced by MASSVET consists of a set of method calls added as part of the repackaging process. These method calls typically do not pinpoint the malicious functionality in the app. For example, the explanation produced by MASSVET for an instance of the GoldDream family is shown in Figure 7 (right). It consists of a list of calls to APIs that MASSVET considers suspicious. While these API calls play a role in the malicious functionality of this app, they are also commonly used by benign apps to process SMS messages, and do not capture the overall malice present in this app.

In addition, we believe the results produced by ASTROID confer a number of other benefits:

Malware family. Unlike DREBIN and MASSVET, which can only identify whether an app is malicious or benign, ASTROID determines which malware family the app belongs to. Since some malware families are more malicious than others, the ability to categorize different apps into malware families provides finer-grained information about the threat level compared binary classification as malicious vs. benign. For instance, some malware families merely affect user experience [43] whereas others introduce financial risks by stealing user's personal account or credit card information [44].

Disinfection. Since ASTROID pinpoints the malicious components in the malicious app, it can be used to "disinfect" the app by removing these malicious components.