EnMobile: Entity-based Characterization and Analysis of Mobile Malware

Wei Yang*
University of Illinois at Urbana-Champaign
weiyang3@illinois.edu

Mukul R. Prasad
Fujitsu Laboratories of America
Sunnyvale, California
mukul@us.fujitsu.com

Tao Xie†
University of Illinois at Urbana-Champaign
taoxie@illinois.edu

ABSTRACT

Modern mobile malware tend to conduct their malicious exploits through sophisticated patterns of interactions that involve multiple entities, *e.g.,* the mobile platform, human users, and network locations. Such malware often evade the detection by existing approaches due to their limited expressiveness and accuracy in characterizing and detecting these malware. To address these issues, in this paper, we recognize entities in the *environment* of an app, the app's interactions with such entities, and the provenance of these interactions, *i.e.,* the intent and ownership of each interaction, as the key to comprehensively characterizing modern mobile apps, and mobile malware in particular. With this insight, we propose a novel approach named EnMobile including a new entity-based characterization of mobile-app behaviors, and corresponding static analyses, to accurately characterize an app's interactions with entities. We implement EnMobile and provide a practical application of EnMobile in a signature-based scheme for detecting mobile malware. We evaluate EnMobile on a set of 6614 apps consisting of malware from Genome and Drebin along with benign apps from Google Play. Our results show that EnMobile detects malware with substantially higher precision and recall than four state-of-the-art approaches, namely Apposcopy, Drebin, MUDFLOW, and AppContext.

CCS CONCEPTS

• **Program reasoning** → **Program analysis**; • **Systems security** → *Mobile security*;

ACM Reference Format:
Wei Yang, Mukul R. Prasad, and Tao Xie. 2018. EnMobile: Entity-based Characterization and Analysis of Mobile Malware . In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180223

*This author was an intern at Fujitsu Labs. of America for part of this work.
†This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1513939, CNS1564274.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180223

1 INTRODUCTION

The explosive growth in mobile devices and mobile applications, or 'apps', has spurred the growth of mobile malware. A case in point is the number of unique Android malware samples, which increased from 10.7 million samples in 2015 to more than 19.2 million in December 2016 [3]. Malware detection based on behaviors is a prominent class of malware-detection approaches where characteristics of known malware samples are used as a basis of identifying new malware [4, 7, 21, 32, 39, 41–43, 52]. These approaches typically work in two phases: malicious-behavior characterization and malicious-behavior detection. The behavior-characterization phase creates a specification of the malicious behaviors, in the form of manually specified malware signatures or as automatically mined malware models, in a suitable representation. In the behavior-detection phase, techniques based on static or dynamic program analysis are used to analyze a given mobile app for possible matches against the specified signatures or mined models.

Limitations of existing approaches. Existing malware detection approaches [6, 8, 9, 11, 21, 21, 22, 25, 29, 45, 50, 51] suffer from the overfitting problem – these approaches are tailored to detect *only* the malware samples used for deriving the signatures or models. The two main reasons are the *limited expressiveness* and *limited accuracy* of these approaches.

Limited expressiveness. The existing approaches primarily use information-leaking dataflows *within* the app as the basis of a malicious behavior. However, the existing approaches typically fail to capture malicious behaviors initiated and controlled by malicious servers, such as initiating spams or launching denial-of-service attacks. In these exploits, the *intent* is not information leakage. Further, a key indicator of the malicious intent in these cases is the *control* of the malicious interactions by an external server. Without a proper characterization for these key aspects, the existing approaches turn to easily mutable features (shared across malware samples of the same family) such as network addresses or other string constants to detect these behaviors, allowing malware developers to easily change these features to evade detection.

Limited accuracy. The existing approaches fail to recognize and express *end-to-end interactions* between a malware sample and entities in its environment. Instead they characterize segments of these interactions using *implementation-specific* structures (*e.g.,* API methods, objects) as end-points. For example, consider a sample of the `GingerMaster` malware with the following four-phase malicious behavior: (1) the app retrieves and preprocesses a phone number from the telephony manager (entity *A*); (2) the app writes the preprocessed phone number into a temporary file (entity *B*); (3) the app reads the preprocessed phone number from the same

temporary file (entity *B*); (4) the app sends the preprocessed phone number to a (malicious) server (entity *C*). A typical malware signature based on the existing approaches would characterize this behavior as four interactions. Further, the existing approaches see the app interacting with *some* Java `File` object in Phases 2 and 3, without recognizing that such `File` object is the *same* file written to and later read from.

Therefore, the existing approaches can produce false positives by matching the preceding behavior with that of a benign app where the preprocessed phone number is saved to a file, and non-sensitive information read from a *different* file is sent to a server. In addition, the `GingerMaster` malware family has seven variations (*i.e.,* different implementations) during the period of 2011–2013 [46]. These variations can either skip Phases 2 and 3 by directly sending the preprocessed phone number to the malicious server or replace the temporary file entity in Phases 2 and 3 to be a temporary database. Thus, the existing approaches can also produce false negatives by making their malware signature very specific to Phases 1-4.

To address these limitations of the existing approaches, in this paper, we present a novel approach, EnMobile. The EnMobile approach is motivated by the finding [8, 46, 52] that more than 90% of current mobile malware have a command-and-control (C&C) architecture, where the malware receive and respond to commands from an external actor, *e.g.,* a remote server. Thus, EnMobile is built to directly characterize the underlying C&C structure of the malware. In particular, EnMobile improves the existing malware-behavior characterization in three main aspects.

Entity-based characterization. Rather than using implementation-specific structures (*e.g.,* API methods, objects) or easily mutable features in an app, EnMobile expresses the app's behaviors in terms of interactions among *entities*. Entities are a host of actors on the mobile platform including mobile system components (*e.g.,* the telephony manager, SMS manager, contacts provider), local on-device resources (*e.g.,* files, databases), other mobile apps and libraries, human users, and network locations, *etc.* EnMobile recognizes entities through their identities. For example, files with different names are different entities.

Furthermore, introducing the concept of entity allows security analysts to express an entity interaction in an *end-to-end* fashion, making it much more independent of specific realizations of that interaction (*e.g.,* specifics of Phases 2 and 3 in `GingerMaster`), and hence more robust. For example, for the `GingerMaster` malware family, security analysts can simply specify the end-to-end information flow from the telephony manager (entity *A*) to the malicious server (entity *C*), without enumerating all possible intermediate-point entities (*e.g.,* files, databases).

Flow-provenance predicates. Going beyond using information-leaking dataflows within the app, EnMobile enriches interactions among entities with *provenance* information. Provenance in our context refers to *who* controls the flow, and why, *i.e.,* the specific *intended purpose* of the flow [18, 23, 38, 50]. For example, the existing approaches may produce an information flow (`file` $\rightsquigarrow$ `sendTextMessage`). Such flow can match both the behavior of sending contents of a file out through SMS (a benign action of sending predefined messages) and the behavior of specifying which phone numbers that the SMS should be sent to (a malicious action of sending premium messages). To distinguish such scenarios, we propose a set of data flow predicates (Section 4.1.1) to reflect the intended purpose of an information flow (*e.g.,* passing configuration parameters vs. purely transmitting information to another entity), and a set of control flow predicates (Section 4.1.2) to specify the ownership of information flows, *i.e.,* the entity that initiates or controls the information flow.

Identification of entities and entity references. In order to characterize an app's behaviors directly in terms of its interactions with entities of the app, one challenge is to extract the correspondence between an in-program object (termed as an *entity reference*) and the entity with which the object may interact in a given execution context (*e.g.,* calling context). To infer the entities that each Java object can point to, we develop an identity-propagation algorithm that conducts a flow- and context-sensitive analysis. It extends and adapts modern taint analysis algorithms [9]. Our algorithm needs to correctly resolve two core scenarios. First, multiple objects could point to the same entity. Second, a given program object can interact with different entities under different execution contexts.

As discussed earlier, malware typically perform malicious behaviors segmented into multiple phases (*e.g.,* downloading, preprocessing), storing intermediate computation results in temporary files or databases. This segmentation gives rise to multiple *segments* of information flow, punctuated with interactions with entities (*e.g.,* files or databases). These segments would need to be "stitched together" in order to be properly matched against a signature specified to characterize the end-to-end interaction. To address the challenge, we propose a *flow-sensitive* stitching algorithm to ensure that the connected information flows are feasible in the actual execution.

This paper makes the following main contributions:

- **Characterization.** We identify malware interaction patterns with entities and provenance information of the interactions as a corner stone of comprehensively characterizing mobile malware. We also propose a novel signature-specification language, based on this characterization, that enables security analysts to create robust, abstract specifications.
- **Detection.** We design static analyses to derive the entity-based characterization by analyzing bytecode of a given app, including identifying entities and entity references, extracting provenance information for flows, and matching against signatures in the face of segmented flows.
- **Implementation and Evaluation.** We present a practical implementation of our approach and evaluate its effectiveness, on a set of 6614 apps consisting of malware from Genome [52] and Drebin [8] along with benign apps from Google Play. Our results show that EnMobile achieves substantially higher precision and recall than four state-of-the-art approaches, namely Apposcopy [21], Drebin [8], MudFLow [11], and AppContext [50].

2 RELATED WORK

Existing malware detection approaches characterize malware behaviors by features that commonly exist in malware but not in benign apps. These approaches include mining (MUDFLOW [11]), clustering (CHABADA [25]), classification (AppContext [50]), graph matching (Astroid [22], Apposcopy [21]), and natural language processing (AsDroid [26], WHYPER [35]) *etc.* The practice of copy-and-paste is pervasive in the malware industry, resulting in many code

clones in malware samples [16]. Because the same code snippet has appeared in many malware samples (but not in benign apps), these existing approaches may regard non-essential features (*e.g.,* component type, file name, unrelated information flows) in code clones as discriminative features. EnMobile provides security analysts a way to directly characterize malware behaviors through the high-level interactions among entities rather than relying on implementation-specific characteristics of the malware.

EnMobile also falls into the general category of techniques based on information flow analysis. Much work has been proposed to enhance static analysis of mobile apps [12, 13, 15, 24, 28–31, 33, 34, 38, 47–49]. Information flow analysis tracks whether privacy-sensitive data (*i.e.,* sources) flows to outgoing channels or sensitive outlets (*i.e.,* sinks). EnMobile complements existing analysis by adding entity-based characterization to the information flow. AAPL [30] uses enhanced data flow analysis techniques to increase the number of data flows that can be detected by information flow analysis and then uses the peer-voting mechanism to lower the false positive rate to report illegitimate information leakages. AAPL fails to handle obfuscation techniques, such as string encryption, since it employs constant propagation analysis. Further, it incurs high false positive rate, by matching all sources with all potential sinks. EnMobile resolves these two limitations by precisely computing the identity of an entity. SPARTA [20] and FlowDroid [9] are two general information flow analysis frameworks. SPARTA enables the flow-policy checking by providing an integrity type system to annotate source code with information-flow type qualifiers. FlowDroid is a static taint analysis tool for Android apps based on Soot [27] and Heros [14]. EnMobile complements SPARTA and FlowDroid by analyzing all types of data flows to detect malicious behaviors that are not information leakage, *e.g.,* bot-driven C&C behaviors.

3 A MOTIVATING EXAMPLE

We illustrate our approach using a malware example `TrickMe`, shown in Figure 1. `TrickMe` is derived from several real pieces of malware. Its C&C structure comes from Geinimi [2], its downloading behaviors mimic Answerbot [1], and its information leakage behaviors follow BeanBot [5]. TrickMe has three malicious behaviors: sending the user's SMS to the C&C server through Internet, performing click fraud based on coordinates provided by the C&C server, and downloading and renaming malicious payloads. All three behaviors reside in an Activity component `MainActivity`, `TrickMe`'s MAIN Activity component, which is invoked when the malware is launched.

`TrickMe` receives commands and prepares necessary information for future malicious behaviors in the `onCreate` method of `MainActivity` (Figure 1a). It first opens a network connection to a malicious server (Lines 3 - 5), and reads a message from the server to file `server.xml` (Line 12). It then parses the `server.xml` into the four files `commandFile`, `coordinateFile`, `downloadFile`, and `fileNameFile` (Line 13). Finally, it reads a list of SMSs into file `infoFile` (Line 14).

In the `onStop` method of `MainActivity` (Figure 1b), `TrickMe` launches one of three different malicious behaviors based on different commands received earlier from the server. On command `sendInfo` it sends the content of `infoFile` to the server, on command `click` it computes and clicks on the X-Y coordinates computed based on the numbers in `coordinateFile` to incur click fraud [17], and on command `install` it downloads malicious payloads from URL addresses

```
1  public void onCreate(Bundle b) {
2    ...
3    String v0 = "http://www.malicious.com";
4    URL url = new java.net.URL(v0); //<url, CON_1>
5    HttpURLConnection n = url.openConnection(); //<n, CON_1>
6    f_s = new File("server.xml"); //<f_s, FILE_1>
7    f_c = new File("commandFile");//<f_c, FILE_2>
8    f_info = new File("infoFile");
9    f_n = new File("coordinateFile"); //<f_n, FILE_3>
10   f_d = new File("downloadFile");
11   f_f = new File("fileNameFile");
12   read(f_s, n); // Reading message from n to f_s
13   parse(f_s, f_c, f_n, f_d, f_f); //Parsing f_s into four files
14   readSMS(f_info); ... }
```

(a) `onCreate` **method of** `MainActivity`

```
1  public void onStop() {
2    ...
3    for(String command: readLine(f_c)){
4      if(command.equals("click")){
5        float [] axis = getAxis(readLine(f_n));
6        MotionEvent down = MotionEvent.obtain(...,0, axis[0],axis[1],0);
7        MotionEvent up = MotionEvent.obtain(...,1, axis[0],axis[1],0);
8        ...
9        Activity adActivity = ...; //<s,AdsPlatform>
10       Webview adView = adActivity.findViewbyID (...);
11       adView.dispatchTouchEvent(down);
12       adView.dispatchTouchEvent(up);
13     }
14     if(command.equals("sendInfo")){
15       sendFile("http://www.malicious.com",f_info);
16     }
17     if(command.equals("install")){
18       String [] filename = readLine(f_f);
19       int i = 0;
20       for(String url: readLine(f_d)){
21         File f_i = new RandomAccessFile(filename[i++],"rw");
22         read(f_i, new java.net.URL(url).openConnection());
23     }}}...}
```

(b) `onStop` **method of** `MainActivity`

```
1  public String [] readLine(File file){ //<file, FILE_2>, <file, FILE_3>
2    FileReader r = new FileReader(file); //<r, FILE_2>, <r, FILE_3>
3    BufferedReader br = new BufferedReader(r);//<br, FILE_2>, <br, FILE_3>
4    String line = br.readLine(); ...
5    return line; }
```

(c) `readLine` **method of** `MainActivity`

Figure 1: Motivating Example: `TrickMe` **malware**

listed in `downloadFile`, and names the downloaded files according to the list of names in `fileNameFile`. The downloaded payloads are used by `TrickMe` to launch other malicious behaviors.

Comparison of signatures in Apposcopy and EnMobile. In a signature-based scheme for malware detection, such as Apposcopy [21], security analysts can specify the control-flow and data-flow properties shown in Figure 2 as the signature for the `TrickMe` malware.

```
activity(a), icc(SYSTEM, a, MAIN, _),
flow(a, URLConnection, a, file),
flow(a, BufferedReader, a, URLConnection),
flow(a, BufferedReader, a, file),
flow(a, SMS, a, file)
```

Figure 2: Characterization of `TrickMe` in Apposcopy [21]

The `activity(a)` predicate declares an activity component `a`. The `icc(SYSTEM, a, MAIN, _)` predicate states an inter-component communication from the system to the activity `a`, and the content of the communication is a "MAIN" intent message. The `flow(a, SMS, a, file)` predicate represents an information flow from source `SMS` in component `a` to sink `file` in component `a` (Figure 1a, Line 14).

Although the specified predicates do represent valid information flows and triggering events in `TrickMe`, these predicates are insufficient for representing the unique characteristics of the malware, and therefore may be unable to differentiate malware from a benign app. For example, a benign SMS manager app can sync the app's configuration with a server (`BufferedReader` $\rightsquigarrow$ `URLConnection`), (`URLConnection` $\rightsquigarrow$ `file`), and back up SMSs (`SMS` $\rightsquigarrow$ `file`) where $\rightsquigarrow$ indicates an information flow. Such an app also possesses the same control-flow and data-flow properties as `TrickMe`, as per Apposcopy's characterization, and would therefore be indistinguishable from `TrickMe`.

Figure 3 presents the signature of `TrickMe` specified by security analysts in EnMobile. EnMobile enables an accurate characterization of `TrickMe`, in three respects. First, EnMobile allows designating the entities that certain behaviors may be attributed to, thereby precisely characterizing the purpose of the behavior. For example, a unique behavior of `TrickMe` is its use of the "command" read from "commandFile" sent from the C&C server, to direct the launching of different malicious behaviors. Identifying and implicating the entity, the remote C&C server, rather than the local file "commandFile" (as the existing approaches would do), is key to recognizing the true nature of this behavior.

Second, EnMobile allows stitching segments of the end-to-end flow behavior exhibited by `TrickMe`. For example, the signature in Apposcopy includes only the benign-looking flows (`SMS` $\rightsquigarrow$ `file`) and (`BufferedReader` $\rightsquigarrow$ `URLConnection`), while EnMobile infers these flows as segments of a larger, and potentially malicious flow (*i.e.,* `Transmit*(s, n_2)` of `UploadMessage` in Figure 3 indicating the behavior of sending SMSs to a web server).

Third, EnMobile detects malicious behaviors beyond information leakage. EnMobile captures a number of non-leakage behaviors in `TrickMe`, such as click fraud, downloading and renaming files. Capturing such behaviors requires a nuanced characterization of information flows. As shown in our results (Section 6.3), characterization can significantly impact the accuracy of malware detection.

4 ENTITY-BASED CHARACTERIZATION

Broadly, EnMobile aims to characterize an app in terms of its *relevant* interactions with entities. To this end, it tracks and precisely characterizes information flows associated with the security-sensitive behaviors of an app. We next illustrate some preliminaries before presenting characterization in EnMobile.

Security-sensitive Behavior. A security-sensitive behavior is an invocation of a security-sensitive method. We consider two types of methods as security sensitive: permission-protected methods and

```
TrickMe(a) :- Download(a), SendMessage(a), UploadMessage(a).

ClickAds(a):- Connection(n), AdsPlatform(s), SysUIEvent(e),
Config*(n, s, TOUCH), Control*(n, s, TOUCH), Trigger(e, s, TOUCH).

Download(a) :- Connection(n), SysUIEvent(e), Connection(n_i),
Initiate*(n, n_i), File(f_i), Initiate*(n, f_i), Transmit(n_i, f_i),
Trigger(e, f_i, WRITE), Control*(n, f_i, WRITE).

UploadMessage(a) :- Connection(n), SysUIEvent(e), Connection(n_2),
SmsInbox(s), Transmit*(s, n_2), Control*(n, n_2, WRITE),
Trigger(e, n_2, WRITE).
```

Figure 3: Characterization of `TrickMe` in EnMobile

other source/sink methods that read/write information. Permission-protected methods are API methods that require permissions to access security-sensitive resources and data. We use the list of permission-protected methods specified in PScout [10], and the list of source/sink methods specified in SuSi [36]. Further, we follow PScout [10] to label each security-sensitive method call with one of a small set of abstract actions (*e.g.,* `SEND, RECEIVE, READ, WRITE`), based on its overall behavior. These action labels are used in our characterization (Section 4.1).

Entity. An entity is an external resource that an app interacts with during its execution. Entities may include network locations (*e.g.,* URLs or phone numbers) external to the device running the app, such as the URL of a C&C server. They may also include on-device intermediate storage sites (*e.g.,* files, databases) or specific Android system resources (*e.g.,* the SMS Manager), with which the app may interact during execution. Entities form the sources (providing information) or targets/sinks (consuming information) of information flows to/from the app. An entity is defined by a tuple: < *entity type*, *entity identifier* >.

Entity type. The type identifies the category of entity, such as a file or a network location (`File`, `UrlConnect`), as well as the type of communication channel of the app with the entity, such as an SMS communication with a phone number (`SmsTarget`) and a phone call to a number (`PhoneTarget`).

Entity identifier. The identifier is the name or address of the entity, such as a file name, a URL, or a phone number, which together with the entity type can be used to uniquely identify the entity. In EnMobile, entity identifier values are stored and propagated in the program via (primitive-type or string-type) constants or symbolic expressions (involving variables provided by the external input, *e.g.,* network message, user input).

Entity reference. An entity reference is an in-program object or variable that serves as a proxy of the entity within the app and through which the app communicates with the entity. For example, variable `f_s`, in Figure 1a, Line 6, is a reference of a `File` entity with identifier "server.xml". An entity may have multiple references. Conversely, a single object, such as the Android SMS Manager, may instantiate different entities (*e.g.,* SMSs to different phone numbers) at different points during the app's execution.

4.1 Language Specification

We propose a language to characterize an app based on its interactions with entities. One use of such characterization is to write signatures for recognizing malware. Figure 3 shows a signature characterizing the `TrickMe` example.

Table 1: App-behavior description language

Type	Syntax	Definition
Event Predicate	`SysEvent(v)`, `UiEvent(v)`, `SysUiEvent(v)`	`v`: event of appropriate type (one of three event types: System event, UI event, or System UI event)
Entity Predicate	`Entity(e)`, `File(e)`, `UrlConnect(e)`, `SmsTarget(e)`, `SmsInbox(e)`	`e`: entity of appropriate type (partial list of potential entity types)
Data-flow Predicate	`Transmit(`e_{source}`,` e_{target}`)`	e_{source}: source entity, e_{target}: target entity
	`Transmit*(`e_{source}`,` e_{target}`)`	`Transmit*(`e_s`,` e_t`) :- Entity(`e_i`), Transmit*(`e_s`,` e_i`), Transmit(`e_i`,` e_t`)`
	`Config(`e_{source}`,` e_{target}`,` a`)`	e_{source}: source entity, e_{target}: target entity, a: target entity's action
	`Config*(`e_{source}`,` e_{target}`,` a`)`	`Config*(`e_s`,` e_t`,` a`) :- Entity(`e_i`), Transmit*(`e_s`,` e_i`), Config(`e_i`,` e_t`,` a`)`
	`Initiate(`e_{source}`,` e_{target}`)`	e_{source}: source entity, e_{target}: target entity
	`Initiate*(`e_{source}`,` e_{target}`)`	`Initiate*(`e_s`,` e_t`) :- Entity(`e_i`), Transmit*(`e_s`,` e_i`), Initiate(`e_i`,` e_t`)`
Control-flow Predicate	`Trigger(`$v_{trigger}$`,` e_{target}`,` a`)`	$v_{trigger}$: triggering event, e_{target}: implicated entity, a: security action
	`Control(`$e_{control}$`,` e_{target}`,` a`)`	$e_{control}$: controlling entity, e_{target}: controlled entity, a: security action
	`Control*(`$e_{control}$`,` e_{target}`,` a`)`	`Control*(`e_c`,` e_t`,` a`) :- Entity(`e_i`), Transmit*(`e_c`,` e_i`), Control(`e_i`,` e_t`,` a`)`

The characterization is a set of Datalog rules. Each rule, of the form: `head :- predicate1, predicate2, ...` , is a horn clause, defining a predicate `head` as the conjunction (logical AND) of one or more other predicates (*e.g.,* `predicate1`). A predicate is a relation name with variables or constants as arguments.

Table 1 provides an informal specification of our proposed language. It consists of four kinds of predicates, namely *event*, *entity*, *data-flow*, and *control-flow* predicates, as described next.

Event predicate. Event predicates declare relevant events, as one of three types: *System*, *UI*, or *System UI* events. A system event is one initiated by the system-state changes, *e.g.,* receiving an SMS, a UI event is triggered by interactions on an app's graphical user interface, and a system UI event is triggered by the interactions on the device interfaces, such as pressing an app's icon on the system's screen to launch the app. This categorization follows previous work on Android testing [50].

Entity predicate. Entity predicates declare specific entities, each of a specific type, with which the app interacts during its execution. For example, in Figure 3, `File(f)` denotes a file entity `f`. Table 1 lists a few examples of currently recognized types.

4.1.1 Data-flow predicates. We make the observation that the *intent* of an information flow can be determined based on a specific parameter of the sink method that it flows into. The reason is that each parameter of a (sink) method plays a specific role in executing its behavior. Thus, our characterization categorizes each parameter of a sink method into one of three types: (1) *transmit* parameters, which receive data to be written to a target entity, (2) *config* parameters, which are used to configure security-sensitive behaviors, and (3) *initiate* parameters, which carry identifiers, *e.g.,* the file name, to initialize a target entity. Based on this characterization, information flows can also be categorized as Transmit, Config, or Initiate, and represented using the corresponding predicates as explained below. To implement this characterization, we pre-compile lists of transmit, config, initiate parameters and their corresponding methods for common entities in the Android SDK, as a one-time effort for Android.

Predicate *Transmit* (*Transmit).** The *Transmit* predicate encodes data transmission from a source entity e_{source} to a target entity e_{target}, where the app reads information from e_{source} and writes it to e_{target}. An information flow satisfies a *Transmit* predicate if it flows into a designated transmit parameter of a sink method. For example, in the *TrickMe* characterization (Figure 3), predicate `Transmit(n_i, f_i)` encodes the behavior of downloading payloads from given URLs (`n_i`) to files (`f_i`).

We also define the predicate `Transmit*(`e_{source}`,` e_{target}`)`, to represent information transitively flowing from e_{source} to e_{target} through a sequence of *Transmit* flows. For example, the `Transmit*(s, n_2)` (Figure 3) encodes the behavior of reading an SMS from SMSInbox `s`, storing it into file "f_info" (Line 14, Figure 1a) and subsequently forwarding it to a given URL `n_2` (Line 15, Figure 1b).

Predicate *Config* (*Config).** This predicate encodes information flows from a source entity e_{source} to target entity e_{target} initiated exclusively for configuring the behavior of a security-sensitive action a performed by e_{target}. Similar to *Transmit*, the definition is extended to define predicates *Config**, as per Table 1. For example, in the `TrickMe` malware, the number saved in the "coordinateFile" is used to configure the behavior of *dispatchTouchEvent* (Lines 4-7 in Figure 1b), as the content in the "coordinateFile" is from network connection *n*, the configuration relationship is represented by `Config*(n, s, TOUCH)`.

Predicate *Initiate* (*Initiate).** This predicate represents a behavior where the entity identifier (*e.g.,* a file name) of a target entity e_{target} is read from a source entity e_{source} and flows into an initialization statement used to instantiate e_{target}. *Initiate* can be extended to predicate *Initiate**, as defined in Table 1. In the *TrickMe* signature (Figure 3), predicate `Initiate*(n, f_i)` represents the behavior that file `f_i` is instantiated using its identifier read from file "filenameFile" (Lines 18-22, Figure 1b), which itself is downloaded from network connection `n` (Lines 11-14, Figure 1a).

4.1.2 Control-flow predicates. These predicates capture the "*who*" of security-sensitive behaviors, *i.e.,* which entity or event controls them, a key determinant of the maliciousness of behaviors.

Predicate *Trigger*. The *Trigger* predicate asserts that a given security-sensitive behavior is triggered by a certain event. Specifically, predicate `Trigger(`$v_{trigger}$`,` e_{target}`,` A`)` is true if event $v_{trigger}$ triggers the execution path to a method call performing an action A (*e.g.,* upload information), where e_{target} (*e.g.,* URL connection) is the target entity whose reference in the program makes the security-sensitive method call. For example, the `onStop` method of `TrickMe` (Figure 1b), which can be *triggered* by a System UI event such as pressing the HOME button, contains three specific behaviors. The three `Trigger(e,*,*)` predicates in Figure 3 capture this triggering relationship.

Predicate *Control*. The *Control* predicate asserts that a security-sensitive action a, performed by a reference of entity e_{target}, is control-dependent on another entity $e_{control}$. Specifically, predicate `Control(`$e_{control}$`,` e_{target}`,` a`)` is true if and only if there exists an information flow from a reference of $e_{control}$ to a conditional statement guarding the execution of a security-sensitive method call, performing action a. *Control* can also be extended to predicate *Control**, as defined in Table 1. In the `TrickMe` example, the command sent from URLConnection `n` *controls* the three malicious behaviors. The predicates `Control*(n, *, *)` in Figure 3 encode this control relationship.

5 ENTITY-BASED STATIC ANALYSIS

In this section, we present how EnMobile matches an Android program against the given malware signatures specified with the entity-based characterization. At the meta-level, such process takes four steps: (1) identify entities (*i.e.,* entity type and entity identifier); (2) map entities to program objects (*i.e.,* entity references); (3) extract entity-based flow facts through analysis on entity references and augment the extracted flow facts with provenance information; (4) match the flow facts against the malware signatures.

5.1 Identifying Entities and Entity References

For the purpose of analysis, EnMobile categorizes entity references into two types: initial entity reference and alias entity reference. Normally identifying an entity reference depends on a parameter (*e.g.,* file name) in the statement that initializes the reference. We name such a parameter as an *indicative parameter*. If an indicative parameter is a constant or external input (*e.g.,* user input, network message), we term the entity reference initialized by the parameter as *initial entity reference*. If an indicative parameter is a variable that in turn points to the initial entity reference (*e.g.,* variable `file` and `r` in Lines 2-3, Figure 1c), we term the entity reference initialized by the parameter as *alias entity reference*.

For brevity, we use only entities related to the SMS-sending behavior in `TrickMe` as examples to illustrate the techniques in the rest of this section. For each entity reference involved in sending SMS, we use red comments in Figure 1 to show the pair of the entity reference (*i.e.,* variable) and the entity that the reference points to. For example, in `TrickMe`, `url`, `f_s`, `f_c`, `f_info`, `f_n`, `f_d`, `f_f` in Figure 1a are initial entity references, while `n` in Figure 1a and `r`, `br` in Figure 1c are alias entity references.

Identifying entities. EnMobile identifies entities via initial entity references. In particular, EnMobile identifies the entity type through the Java types of initial entity references. For example, in Figure 1a, `f_s` has java Type `File` indicating the entity type as file.

EnMobile extracts the entity identifier through the indicative parameter of the initial entity reference. An indicative parameter can either be a constant or external input. For a constant identifier, EnMobile records the constant value as the identity of the entity. For an external-input identifier, EnMobile computes a symbolic expression as the identity of the entity. The symbolic expression is computed by a combination of sources of the variable (constant or user input) and the propagation paths from the sources to the variable. The reason why we choose to compute the symbolic expression instead of using constant propagation analysis to infer the actual value of the identifier is to deal with the situations where the identifier value goes through an encryption scheme.

Mapping entities to entity references. Initial entity references are naturally mapped to entities after identifying the entities. Mapping alias entity references to corresponding entities is still challenging for two main reasons. First, multiple references could point to the same entity. In Figure 1c, `r` and `br` point to the file entity referred to by `file`. The identity of the entity can be *propagated* from one reference to another as one reference is used to initialize another object. Second, an entity reference may point to different entities under different execution contexts. In Figure 1c, `r` and `br` can point to "commandFile", "coordinateFile", or "downloadFile" in different executions.

We develop an *identity propagation* algorithm to compute the entities that each alias entity reference points to. For a given initial entity reference, the identity propagation computes a set of entity references that point to the same entity as the initial entity reference; we refer to this set as *reference set*. The idea of identity propagation extends the idea of the taint propagation. The identity taints are generated at each initial entity reference. Any entity reference being tainted points to the same entity as the initial entity reference of the taint.

Table 2 informally presents the flow functions used in the identity propagation algorithm. A flow function of a statement maps the set of dataflow facts **in** that hold before the statement to the set of dataflow facts **out** that hold after the statement. Here a dataflow fact is the reference set of identities. In identity propagation, the flow function maps I_{in} (*i.e.,* reference set before the statement) to I_{out} (*i.e.,* reference set after the statement). After the reference set has been calculated for each identity, EnMobile iterates through all identities and merges the reference sets if two identities are identical (*i.e.,* two identities with the same identifier value and same type).

In the `TrickMe` example, `n` in Figure 1a and `r`, `br` in Figure 1c are alias entity references. The identity taint `CON_1` is generated from `url` and propagated to `n` by applying ⓘ [1]. For `r` and `br`, the identity `File_2` first propagates from Line 3 in Figure 1b to variable `file` on Line 1 in Figure 1c by applying ⓘⓥ. Then the identity further propagates to `r` and `br` by applying ⓘ. Note that identity `File_3` also propagates (Line 5 in Figure 1b) to `file`, `r`, and `br`. However, because our analysis is context-sensitive, the later information-flow analysis is able to tell that the variable `command` on Line 3 in Figure 1b

[1] `URL` is used to initialize a new `HttpURLConnection` object in the implementation of `HttpURLConnectionImpl`.

Table 2: Identity propagation logic

Statement Type	Format	Flow Functions	Propagation Description
Entity Initialization	$x = new(a_{init}, a_0, ..., a_n), n \in \mathcal{N}$	① $I_{out} \stackrel{s}{=} I_{in} \cup x, a_{init} \in I_{in}$	Indicative parameter → Left-hand side (LHS)
Assign	$x = y$	⑪ $I_{out} \stackrel{s}{=} I_{in} \cup x , y \in I_{in}$	Right-hand side (RHS) → LHS
Identity Setter	$x.set(y)$	⑪ $I_{out} \stackrel{s}{=} I_{in} \cup x , y \in I_{in}$	Tainted parameter → Caller object (e.g., y→x)
Call	$c.m(a_0, ..., a_n), n \in \mathcal{N}$	⑳ $I_{out} \stackrel{s}{=} I_{in} \cup \{a'_i\}, \forall a_i \in I_{in}$	Caller parameter → Callee (Context switching)
Return	return $y; x = c.m(a_0, ..., a_n), n \in \mathcal{N}$	⑤ $I_{out} \stackrel{s}{=} I_{in} \cup x , y \in I_{in}$	Returned object → LHS

is tainted by the information from `File_2`, and variable `axis[]` on Line 5 in Figure 1b is tainted by the information from `File_3`.

We perform two customizations in our information flow analysis. First, the sources of our identity propagation are based on a certain type of variables (*i.e.,* certain primitive type or string type of variables in initialization methods) instead of certain methods (*i.e.,* source methods). So in addition to method matching, identity generation requires an additional checking on method parameters. Second, the identity propagation is field-insensitive through certain methods (*e.g.,* initialization methods, setter methods). For example, in an identity setter method, an identity taint propagates from the method parameter directly to the receiver object rather than to the class field that is assigned by the taints in the setter method. To address such difference, we feed predefined knowledge (*e.g.,* initialization methods and parameters of entities) to help EnMobile perform identity propagation according to the high-level semantics.

5.2 Augmenting with Provenance Information

We omit the description of extracting flow facts through entity references given that this process is a standard information-flow analysis. In this section, we illustrate how we augment the extracted flow facts with provenance information in two steps.

Classifying the type of information flows. In this step, we classify the type of flows based on the three types of data flows defined in Section 4. To differentiate the type of data flows, EnMobile needs to track which parameter of the sink method is tainted in the computation. EnMobile first performs traditional information flow analysis and takes the computed flows and sink variables as input, and checks them with the predefined method signatures to determine whether the information flow is *transmit*, *config*, or *initiate*. EnMobile takes lists of method signatures that contain the information of transmit, config, and initiate parameters in the methods. For each information flow, EnMobile derives the flow type based on the sink variable that the information flows into. For example, in the *SmsManager.sendTextMessage* method, the first parameter (destinationAddress) indicates that the flow is a config flow, and the third parameter (text) indicates that the flow is a transmit flow.

Computing control-flow predicates. To identify events $v_{trigger}$ that can trigger Target Entity e_{target}, we first locate the security-sensitive method called by references of e_{target}. Each security-sensitive method corresponds to an action of the entity (*e.g.,* SEND for *sendTextMessage*). Then, we analyze the call path's entrypoints that lead to the method calls. The entrypoints are the top nodes in the call graph of the app. EnMobile follows the inter-component communications to link the e_{target}'s method call to the entrypoints, and the events $v_{trigger}$ can be further inferred from the entrypoints.

To compute control dependencies among entities, EnMobile tracks information flows from entities to conditional statements through inter-procedure control-flow graphs. The value of a conditional statement decides which program branch to take in runtime executions, and thus decides invocations of methods on one of the program branches. For a given method M invoked by an entity e_{target} (M corresponds to Action A), EnMobile computes the information flows from all entities to conditional statements (that control the invocations of M). If an information flow from an entity $e_{control}$ to the conditional statements exists, then $e_{control}$ controls the Action A of e_{target} (*i.e.,* `Control` ($e_{control}$, e_{target}, `A`) holds).

5.3 Matching Against Signatures

To perform malware detection, EnMobile compares the set of flow facts $\mathcal{P}(M)$ extracted from an app M, using the aforementioned analysis, against a pre-compiled library of signatures of known malware. Specifically, for a malware signature $\mathcal{S}$ (as a set of predicates) from the malware library, the comparison checks whether the predicates in $\mathcal{S}$ are a subset of the flow facts (also as a set of predicates) in $\mathcal{P}(M)$, modulo renaming of variables. In the process of signature matching, EnMobile enumerates all feasible combinations of the segmented flows to match the end-to-end characterizations in the signatures. EnMobile determines the feasibility of combinations of segmented flows by incorporating the flow-sensitive information (*i.e.,* taking into account the order of the statements) in the extracted flow facts. Please refer to our project website [19] for more details.

6 EVALUATION

We evaluate EnMobile in characterizing malware behaviors, by investigating the following research questions:

RQ1: How effective is EnMobile in characterizing malicious behaviors in existing malware?

RQ2: How do entity-identity analysis and richer data flow predicates in entity-based characterization contribute to the effectiveness of malicious-behavior identification?

RQ3: What is the effectiveness of EnMobile compared to other state-of-the-art approaches of malware detection?

6.1 Evaluation Setup

Evaluation Subjects. Our subject set consists of a malware dataset and a benign app dataset. Our malware dataset starts with all malware from the Genome [52] and Drebin [8] malware datasets, which are commonly used in malware detection research [11, 21, 25, 26, 29, 50]. The Malware Genome dataset comprises 1,260 malware samples organized into 49 malware families. The Drebin dataset

Table 3: Categorization of Malware by EnMobile

Malware Family	#T	EnMob(%)		Base1 (%)		Base2 (%)		Appo(%)	
		FN	FP	FN	FP	FN	FP	FN	FP
ADRD	91	0.0	0.0	2.3	0.0	0.0	0.0	36.4	0.0
AnserverBot	184	0.6	0.1	2.2	0.0	0.6	0.4	0.0	0.0
BaseBridege	331	10.4	0.2	33.4	0.2	10.4	0.5	50.0	0.1
Boxer	27	0.0	0.2	7.4	0.2	0.0	0.2	25.9	0.4
DroidDream	97	0.0	0.1	4.4	0.1	0.0	0.2	3.1	8.5
DroidDreamL	46	1.1	0.0	1.1	0.0	1.1	0.0	0.0	0.0
DroidKungFu	668	1.8	0.0	5.1	0.0	1.6	0.7	8.1	0.0
ExploitLotoor	70	10.4	0.1	17.9	0.1	10.4	0.4	85.0	1.9
FakeDoc	132	2.3	0.1	11.0	0.1	2.4	0.2	6.3	0.3
FakeInstaller	925	1.8	0.0	5.8	0.0	1.8	0.1	68.4	0.1
FakeRun	61	0.0	0.1	4.9	0.1	0.0	0.3	11.1	0.0
Gappusin	58	8.6	0.0	12.0	0.0	8.6	0.1	58.6	0.0
Geinimi	94	0.0	0.2	4.4	0.2	0.0	0.9	0.0	0.0
GingerMaster	342	3.8	0.1	7.1	0.1	3.8	0.2	70.4	0.0
GoldDream	70	0.0	0.0	1.6	0.0	0.0	0.0	3.2	0.0
Hamob	28	4.5	0.0	18.1	0.0	4.5	0.0	3.7	5.7
Iconosys	152	0.6	0.0	2.6	0.0	0.6	0.2	69.7	0.0
Imlog	43	0.0	0.0	0.0	0.0	0.0	0.0	76.7	29.1
Jifake	29	0.0	0.0	3.4	0.0	0.0	0.0	50.0	0.5
KMin	148	3.7	0.1	5.9	0.1	3.7	0.1	34.7	19.1
MobileTx	69	0.0	0.1	8.8	0.1	0.0	0.6	31.9	0.0
Opfake	613	0.8	0.0	5.6	0.0	0.8	0.1	33.7	19.2
Pjapps	58	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0
Plankton	625	2.1	0.0	3.7	0.0	2.1	0.2	32.1	3.0
SendPay	59	6.9	0.0	8.6	0.0	6.9	0.0	22.4	7.9
SMSreg	41	0.0	0.0	9.8	0.0	0.0	0.3	16.2	44.6
YZHC	37	0.0	0.0	3.6	0.0	0.0	0.1	0.0	5.2
Average	188.8	2.2	0.1	7.1	0.1	2.2	0.2	29.5	5.4

#T = Total number of apps, EnMob = EnMobile, Appo = Apposcopy
FN = False negative rate, Base1 = EnMobile without entity identity
FP = False positive rate, Base2 = EnMobile without data-flow types

comprises 5,560 malware samples organized into 178 malware families. We remove families containing fewer than 20 malware samples as well as malware samples duplicated across Genome and Drebin, yielding 27 families with 5,098 malware samples in total. To collect benign apps, we downloaded a total of 2,700 apps (100 randomly selected apps for each of the 27 categories) from Google Play, as of December 2016. We implement EnMobile using several third-party static analysis frameworks, including Soot [40], FlowDroid [9], and AppContext [50]. To isolate and remove the effects of potential limitations of these frameworks on our conclusions, we further prerun EnMobile on the complete subject set and filter out any apps that cause any of the third-party frameworks to throw exceptions or time out. This step gives us a final analyzable dataset of 4897 malware samples and 1717 benign apps.

Given the large number of unanalyzable benign apps (999), we reassess the distribution of our final benign app dataset. We find that it retains 52 to 77 apps in each Google Play category (originally 100); the size range (42KB to 51,192KB) of a final app's file and the size range (16KB to 8,829KB) of a final app's dex file (bytecode without resource files) remain the same compared to the original dataset. This distribution suggests that our benign app dataset remains broadly representative of real benign apps even after removing unanalyzable apps. All runs of EnMobile have been performed on a desktop with 4 Intel Xeon 3.2 GHz E3-1225 processors and 16 GB of memory with a timeout of 20 minutes per app (the same default timeout set by Apposcopy [21]).

Malware Signature Library. For the purpose of the evaluation, we develop a library of malware signatures, one per family, for each of the 27 malware families (Table 3) in our dataset. Each malware signature characterizes the malware samples of a given family, using the signature language proposed in Section 4.1. The signature is constructed by examining 10 randomly selected malware samples from the family and 100 randomly selected benign apps. Specifically, we first collect the security-sensitive behaviors (*i.e.,* data-flow and control-flow facts) of each of the apps. From these behaviors, the behaviors common to all malware samples, but absent from the benign apps, are identified and expressed in the signature language. The signature-creation procedure entails fewer than six man-hours of effort per new malware signature, as a one-time effort for each malware family.

6.2 RQ1: Entity-Based Characterization

To evaluate the effectiveness of EnMobile's entity-based characterization on our malware dataset, we run EnMobile on all malware samples and benign apps, except the ones used to develop the malware signatures. We evaluate the runs in terms of two metrics.

For the first evaluation metric, we record which malware family signatures (if any) each app matches[2]. Ideally, each malware sample should match its family's signature and no other signatures. Note that this classification problem is qualitatively harder than simply classifying a given app as malware or benign, and is the true test of the accuracy of a signature-based approach, such as EnMobile. Column "EnMobile" in Table 3 shows the results of this evaluation metric. Here, for a given malware family, false negative rate (FN) refers to the percentage of malware samples (out of all samples in the malware family) that are *not* matched by EnMobile to that family's signature. Conversely, false positive (FP) refers to the percentage of apps (out of all benign apps and malware of *other* families) that are *incorrectly* matched by EnMobile to this family's signature. As shown in Table 3, EnMobile can effectively classify malware instances into their appropriate families with on average 2.2% false negatives and around 0.1% false positives. For most malware families, EnMobile has under 5% false negatives and 0.1% false positives.

The second evaluation metric assesses the effectiveness of EnMobile and other approaches, in broadly differentiating malware from benign apps, *i.e.,* classifying malware as malware (vs. benign) and benign apps as such (vs. malware). Such metric is in contrast to the family-based classification in Table 3. As shown in columns "EnMobile" in Table 4, here too EnMobile performs quite well, correctly classifying over 97% of the malware and 99% of the benign

[2]While theoretically possible, no apps ended up matching multiple signatures in our current evaluation.

Table 4: Differentiating Malware and Benign Apps

Apps	#Total Apps.	#Analyzable by EnMobile	#Analyzable by Apposcopy	EnMobile		Base1		Base2		Appo	
				Malicious	Benign	Malicious	Benign	Malicious	Benign	Malicious	Benign
Benign	2716	1717	1592	1.0%	99.0%	0.9%	99.1%	4.8%	95.2%	59.1%	40.9%
Malware	5098	4897	4062	97.2%	2.8%	92.2%	7.8%	97.3%	2.7%	67.6%	32.4%

apps. Further, on manually inspecting the 1% (*i.e.,* 17 out 1717) benign apps being classified as malware, we find that 8 apps actually possess malicious or highly suspicious behaviors. An example app is `com.genericsnippet.funnyecards`, which has now been removed from Google Play. This example demonstrates EnMobile's capability to detect unknown suspicious, even potentially malicious, apps. The other 9 benign apps misclassified by EnMobile contain some interesting suspicious-looking behaviors; for example, an app turns a deprecated smartphone into a baby camera, sending SMS to parents whenever the phone signal changes. In future work, we plan to use app descriptions to check whether such suspicious behaviors are in fact desirable.

6.3 RQ2: Entity Identities and Flow Predicates

Two of the key contributions of EnMobile are (1) its entity-based characterization, built on top of entity-identity analysis (Section 5.1), and (2) the rich set of data-flow predicates (Section 4.1) to identify malicious intents. In this research question, we assess the effectiveness of these specific features by comparing EnMobile against two baseline versions: EnMobile without entity-identity analysis (Base1), and EnMobile without rich data-flow types (Base2). Note that the core information flow analysis in *both* Base1 and Base2, and indeed in EnMobile itself, is at least as precise as the type and/or API-based information flow analysis in previous work [9, 11, 21, 29, 47], albeit implemented in our own framework.

EnMobile without entity identities (*Base1*). To realize Base1, we turn off the entity-identity analysis in EnMobile. Specifically, the analysis retains the type of the entities, but ignores the identities of the entities in the flows. Note that we still need to perform identity propagation to some extent to infer the entity type for some entity references. Indeed, without entity identities, stitching segmented information flows cannot be performed either.

For fair comparison, we also modify EnMobile's malware signature library to make it suitable for Base1. Specifically, in each of the signatures, we remove entity identities but retain entity types. Further, we study malware reports from major anti-virus vendors as well as flows extracted by EnMobile to identify (segmented) information flows common to a majority of the samples in a malware family. We replace the original data-flow predicates, representing a connected flow in the signature, with a set of predicates representing each of the segmented flows. When no flows match a majority of the malware samples, we use flows with the best (highest) match.

Table 3 (evaluating characterization of malware by family) and Table 4 (evaluating basic malware detection of malware vs. benign) show a comparison of EnMobile (column EnMobile in both tables) to Base1 (column Base1). The results show that Base1, *i.e.,* EnMobile without entity identities, produces more false negatives for most malware families (7.1% on average vs. 2.2% for EnMobile) as well as in overall malware detection (7.8% vs. 2.8% in Table 4).

One main reason is that different samples in a malware family typically have different implementations of the same end-to-end flow, through varied sets of segmented flows. Without the benefit of the entity-identity analysis, and the flow stitching that it enables, no single signature can characterize all samples of a malware family, *even* with the preceding custom retrofitting of the signature library for Base1. These results demonstrate the benefit of our entity-identity analysis for accurate malware characterization.

EnMobile without types of data flows (*Base2*). To realize Base2, we simply represent the three types of data flows as a single basic information flow, in both the signatures and in the extracted flow facts for each app. We then perform signature matching based on the extracted flow facts and signatures.

As shown in Tables 3 and 4, Base2 produces more false positives for some malware families and incorrectly marks more benign apps as malware than EnMobile (4.8% vs. just 1% in Table 4). The reason is that the signatures lacking our provenance information incur wrong matching of data flows. For example, the analysis may match a Transmit flow (*e.g.,* a flow sending an SMS) with a possible Config flow (*e.g.,* a flow specifying the SMS recipient's number).

6.4 RQ3: Comparison with Related Approaches

We compare EnMobile with four related state-of-the-art approaches: one signature-based approach (Apposcopy [21]) and three learning-based approaches (MUDFLOW [11], AppContext [50], and Drebin [8]).

Comparison with a signature-based approach (Apposcopy). Apposcopy leverages a list of manually-specified signatures (*e.g.,* Figure 2) to match malware samples. Because Apposcopy provides signatures for only several malware families in our dataset, we use the following methodology to generate the best possible Apposcopy signatures uniformly for *all* malware families. We generate Apposcopy signatures for each family by two means: (i) we follow the same procedure as in creating EnMobile signatures to manually create an Apposcopy signature based on 10 malware samples and 100 benign samples; (ii) we run Astroid [22], an automatic signature generator for Apposcopy, 10 times for each family. Each time Astroid randomly selects five samples from the malware family and produces a signature. We pick the best signature, in terms of the least total number of FP and FN, from among the preceding 11 signatures, to report the results. All runs of Apposcopy are on the same machine as EnMobile with the same timeout threshold per app, *i.e.,* 20 minutes.

The last two columns ("Appo") of Tables 3 and 4 report the results of Apposcopy in detecting malware. As shown in Table 3, Apposcopy performs much worse than EnMobile for most of the malware families, especially the malware families whose most malware samples are from the Drebin malware database. This degradation in performance is likely due to the evolution of malware. For example, in the `Kmin` malware family, the functionality of a receiver

`com.km.HoldMessage` in some malware samples is replaced by a service `com.km.charge.CycleServic` in some other malware samples. This kind of evolution changes the type of the Android component hosting the malicious behavior. Such changes can easily evade Apposcopy's detection because Apposcopy's signatures heavily rely on the internal component structure (including a component's type) to characterize malware. However, EnMobile does not suffer from the same issue because such structural changes do not affect the end-to-end communications among entities.

Comparison with learning-based approaches. We also compare EnMobile with three state-of-the-art learning-based detection approaches: AppContext [50], Drebin [8], and MUDFLOW [11].

Both AppContext and Drebin require a large number of malware samples as training data to train a machine learning model. However, many malware families have very few known samples – only 42% of malware families have more than 5 samples [22]. So, in addition to evaluating AppContext and Drebin following traditional ten-fold cross-validations (**O.** in Table 5), we also evaluate their effectiveness on a smaller training set (**S.** in Table 5) by following the evaluation methodology used in Astroid [22]. As per this methodology, instead of training malware from all families as a whole, we perform the training and testing family by family. For each malware family, the training set consists 10 randomly selected samples from the family, all samples from other malware families, and a similar number of benign apps as in the original training set. The testing set consists of the rest of samples from the malware family and the rest of benign apps. We report the average results of all families in Table 5.

MUDFLOW detects malware by identifying abnormal information flows for each category of sensitive sources. To produce the input that MUDFLOW accepts, we use FlowDroid [9] to extract information flows from all of our subjects. To compute the final result, we feed to MUDFLOW the extracted information flows, with sources and sinks of these information flows categorized using SuSi [36], as well as the permission list of each app.

Table 5 shows the effectiveness of the existing approaches and EnMobile. As shown in the table, EnMobile outperforms all the existing approaches. Note that although AppContext and Drebin reach similar effectiveness as EnMobile when training with the original dataset (*i.e.,* 90% training data and 10% testing data), their effectiveness degrades a lot when using a smaller number of training samples. This result is especially impressive for EnMobile, considering that the difference between the smaller and original training datasets comes from much reduced malware samples in a single malware family. This result is indicative of the overfitting nature of these learning-based approaches. It suggests that EnMobile can be a great substitute for learning-based approaches, in malware detection, when security analysts have access to only a small number of malware samples. EnMobile also outperforms MUDFLOW, exhibiting much higher recall. The advantage of EnMobile over MUDFLOW lies in detecting those malware samples that have C&C behaviors or behaviors of dynamic code loading (*e.g.,* BaseBridge). Because of the dynamic nature of such behaviors (*i.e.,* the loaded code is unknown before the execution), traditional information-flow analysis often fails to detect these behaviors. EnMobile's entity-based characterization allows it to accurately identify the controlling entity of the downloading behavior and the C&C nature of the malware. Thus, EnMobile can outperform the existing approaches by accurately identifying these malware samples without requiring detailed knowledge of the dynamically loaded code.

Table 5: Identification of malware by variations of AppContext, MUDFLOW, Drebin, and Apposcopy

	AppContext		Drebin		MUDFLOW	Apposcopy	EnMobile
	O.	S.	O.	S.			
P.(%)	95.42	76.52	98.47	92.80	97.61	74.48	99.64
R.(%)	97.65	95.15	97.48	89.21	53.46	67.60	97.24

P. = Precision, R. = Recall, O.= Result with Original Training Samples
S.= Result with Smaller Number of Training Samples

7 DISCUSSION

Limitations. Intentional obfuscations of entity identities may sabotage our analysis. For example, creating an alias entity by using symbolic links (*e.g.,* Ink, Shortcut), or using different copies of the same encryption scheme to encrypt entity identities. In these cases, the malware may evade detection of EnMobile. However, since these camouflage attempts have clear patterns and are likely to be suspicious, other techniques such as dynamic analysis [37] can be used to complement EnMobile. Attackers can also hide malicious behaviors matched by our signatures into dynamic loaded code to evade EnMobile's detection. However, security analysts can leverage EnMobile to further characterize the behaviors of dynamic code loading to detect the evolved malware. In our evaluation, signatures characterizing dynamic code loading can successfully match malware of corresponding families (*e.g.,* basebridge).

Threats to Validity. The tuning of malware signatures could affect the results of the evaluation. To prevent EnMobile's signatures from being overfitting for our subjects, when constructing the malware signatures, we strictly constrain ourselves in analyzing no more than 10 malware samples per family. Also, EnMobile is based on behavioral signatures rather than syntactic structures used in much of previous work [8, 21, 44], and doing so further mitigates against overfitting.

8 CONCLUSION

In this paper, we have presented EnMobile, a novel approach for accurately characterizing mobile apps' interactions with entities. We have demonstrated a practical application of EnMobile for detecting malware. Our evaluation results suggest the effectiveness of EnMobile in characterizing differential characteristics of malware and benign apps, and robustness of EnMobile's specification-driven signatures (*i.e.,* based on intrinsic definitions of malware) over implementation-driven ones (*i.e.,* based on features of low-level program structures). We envision a number of applications of EnMobile: with increasing uses of IoT apps, EnMobile can be extended for characterizing broader interactions between the physical world and apps; for human-assisted app auditing, entity-based characterization can enhance security analysts' understanding of app behaviors.

REFERENCES

[1] Android.Answerbot. https://www.symantec.com/security_response/writeup.jsp?docid=2011-100711-2129-99.
[2] Android.Geinimi. https://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99.
[3] In review: 2016's mobile threat landscape brings diversity, scale, and scope. http://blog.trendmicro.com/trendlabs-security-intelligence/2016-mobile-threat-landscape/.
[4] Microsoft malware protection center. http://www.microsoft.com/security/portal/threat/Threats.aspx.
[5] Security alert: New BeanBot SMS Trojan discovered. http://www.cs.ncsu.edu/faculty/jiang/BeanBot/.
[6] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Proc. SecureComm*, pages 86–103, 2013.
[7] Antiy Labs. http://www.antiy.net/.
[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proc. NDSS*, 2014.
[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. PLDI*, pages 259–269, 2014.
[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proc. CCS*, pages 217–228, 2012.
[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining Apps for abnormal usage of sensitive data. In *Proc. ICSE*, pages 426–436, 2015.
[12] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Proc. ASE*, pages 669–679, 2015.
[13] O. Bastani, S. Anand, and A. Aiken. Interactively verifying absence of explicit information flows in Android apps. In *Proc. OOPSLA*, pages 299–315, 2015.
[14] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proc. SOAP*, pages 3–8, 2012.
[15] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Proc. NDSS*, 2015.
[16] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale. In *Proc. USENIX Security*, pages 659–674, 2015.
[17] J. Crussell, R. Stevens, and H. Chen. MAdFraud: Investigating ad fraud in Android applications. In *Proc. Mobisys*, pages 123–134, 2014.
[18] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, pages 255–273, 2015.
[19] EnMobile. https://sites.google.com/site/entitymobile/.
[20] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proc. CCS*, pages 1092–1104, 2014.
[21] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proc. FSE*, pages 576–587, 2014.
[22] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *Proc. NDSS*, 2017.
[23] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards detecting logic bombs in Android applications. In *Proc. Security and Privacy (SP)*, pages 377–396, 2016.
[24] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proc. NDSS*, 2015.
[25] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proc. ICSE*, pages 1025–1035, 2014.
[26] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proc. ICSE*, pages 1036–1046, 2014.
[27] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Proc. CETUS*, 2011.
[28] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond. String analysis for Java and Android applications. In *Proc. FSE*, pages 661–672, 2015.
[29] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTa: Detecting inter-component privacy leaks in Android apps. In *Proc. ICSE*, pages 280–291, 2015.
[30] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proc. NDSS*, 2015.
[31] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. In *Proc. CCS*, pages 229–240, 2012.
[32] Virus Information - McAfee. http://home.mcafee.com/virusinfo/.
[33] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *Proc. ICSE*, pages 77–88, 2015.
[34] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proc. USENIX Security*, pages 543–558, 2013.
[35] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: towards automating risk assessment of mobile applications. In *Proc. USENIX Security*, pages 527–542, 2013.
[36] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. NDSS*, 2014.
[37] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proc. NDSS*, 2016.
[38] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard. Covert communication in mobile applications. In *Proc. ASE*, pages 647–657, 2015.
[39] D. L. Shinder. The pros and cons of behavioral based, signature based and whitelist based security, 2015. http://www.windowsecurity.com/articles-tutorials/misc_network_security/Pros-Cons-Behavioral-Signature-Whitelist-Security.html.
[40] Soot: A framework for analyzing and transforming Java and Android applications. http://sable.github.io/soot/.
[41] Threat Analysis - Sophos. https://www.sophos.com/en-us/threat-center/threat-analyses.aspx.
[42] Security Threat - Symantec. https://www.symantec.com/security_response/.
[43] Threat Encyclopedia - Trend Micro. http://www.trendmicro.com/vinfo/us/threat-encyclopedia/.
[44] VirusTotal - free online virus, malware and URL scanner. https://www.virustotal.com/.
[45] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *BlackHat DC Conference*, March 2007.
[46] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current Android malware. In *Proc. DIMVA*, pages 252–276, 2017.
[47] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proc. CCS*, pages 1329–1341, 2014.
[48] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *Proc. ICSE*, pages 89–99, 2015.
[49] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *Proc. ASE*, pages 658–668, 2015.
[50] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. ICSE*, pages 303–313, 2015.
[51] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proc. CCS*, pages 1105–1116, 2014.
[52] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. Security and Privacy (SP)*, pages 95–109, 2012.