FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature

Ziyun Zhu
University of Maryland,
College Park, MD, USA
zhuziyun@umiacs.umd.edu

Tudor Dumitraș
University of Maryland,
College Park, MD, USA
tdumitra@umiacs.umd.edu

ABSTRACT

Malware detection increasingly relies on machine learning techniques, which utilize multiple features to separate the malware from the benign apps. The effectiveness of these techniques primarily depends on the manual feature engineering process, based on human knowledge and intuition. However, given the adversaries' efforts to evade detection and the growing volume of publications on malware behaviors, the feature engineering process likely draws from a fraction of the relevant knowledge.

We propose an end-to-end approach for automatic feature engineering. We describe techniques for mining documents written in natural language (e.g. scientific papers) and for representing and querying the knowledge about malware in a way that mirrors the human feature engineering process. Specifically, we first identify abstract behaviors that are associated with malware, and then we map these behaviors to concrete features that can be tested experimentally. We implement these ideas in a system called FeatureSmith, which generates a feature set for detecting Android malware. We train a classifier using these features on a large data set of benign and malicious apps. This classifier achieves a 92.5% true positive rate with only 1% false positives, which is comparable to the performance of a state-of-the-art Android malware detector that relies on manually engineered features. In addition, FeatureSmith is able to suggest informative features that are absent from the manually engineered set and to link the features generated to abstract concepts that describe malware behaviors.

1. INTRODUCTION

A key role of the security community is to propose new *features that characterize adversary behaviors*. For example, the earliest Android malware families exhibited simple malicious behaviors [53] and could often be identified based on the observation that they requested the permissions essential to their operation [54]. Subsequently, Android malware has increasingly adopted more evasive techniques, and in response the security community has proposed a variety of new features to detect these behaviors. Drebin [8], a state of the art system for detecting Android malware, takes into account 545,334 features from 8 different classes.

To engineer such features for malware detection, researchers reason about the properties that malware samples are likely to have in common. This amounts to *generating hypotheses* about malware behavior. While such hypotheses can be tested using statistical techniques, they must be initially formulated by human researchers. This cognitive process is guided by a growing body of knowledge about malware and attacks. For example, Google Scholar estimates that 12,400 papers have been published on Android malware and over 600,000 on intrusion detection; moreover, the volume of scientific publications is growing at an exponential rate [24]. In consequence, it is increasingly challenging to generate good hypotheses about malware behavior, and the feature engineering process likely draws from a fraction of the relevant knowledge.

In this paper, we ask the question: *Can we engineer features automatically by analyzing the content of papers published in security conferences?* Our goal is to generate, without human intervention, features for training machine learning classifiers to detect malware and attacks. The key challenge for achieving this is to attach meaning to the words used to describe malware behavior. For example, a human researcher reading the phrase "sends SMS message "798657" to multiple premium-rate numbers in Russia"[1] would probably conclude that this behavior refers to SMS fraud. However, this conclusion is based on the researcher's knowledge of the world, as the phrase does not provide sufficient linguistic clues that the behavior is malicious. Such commonsense reasoning is viewed as a difficult problem in natural language processing [15]. Additional challenges are specific to security research. Papers typically discuss abstract concepts, which do not correspond directly to features that we can extract and analyze experimentally. These concepts may also *not fit any predetermined knowledge classification system*, as the open-ended character of security research and the adversaries' drive to evade detection gives rise to a growing (and perhaps unbounded) number of concepts.

We describe an automatic feature engineering approach that addresses these challenges by mirroring the human process of reasoning about what malware samples have in common. To this end, we build on ideas from cognitive psychology [12] and represent the knowledge reflected in the security literature as a *semantic network*, with nodes that correspond to the *concepts* discussed in the papers and edges that

[1]This quote from [53] describes the behavior of `FakePlayer`, the first Android trojan detected.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978304

connect related concepts. Rather than extracting predetermined categories of knowledge, we propose rules to identify security concepts expressed in natural language. We assess the semantic similarity among these concepts, and we use it to weight the edges in our network. We also map these concepts to concrete features that we can analyze experimentally. Our approach derives from the observation that, when humans describe a concept, they tend to mention closely related concepts at first, and then they discuss increasingly less relevant concepts. The semantic network also allows us to generate explanations for why the automatically engineered features are associated with malware behaviors.

As a proof of concept, we implement these techniques in a system named FeatureSmith, which generates features for separating benign and malicious Android apps. To this end, FeatureSmith mines 1,068 papers published in the security community and constructs a semantic network with three types of nodes: malware families, malware behaviors, and concrete features. These features correspond to Android permissions, intents and API calls and can be extracted directly from the apps using static analysis tools. FeatureSmith ranks features according to how close they are to the malware on the semantic network (like a data scientist would think about the common properties of the malware samples). We compare features generated automatically in this manner with the features engineered for Drebin [8], which required a substantial manual effort (e.g. to list suspicious Android API calls). Machine learning classifiers trained with these two feature sets achieve comparable performances: over 92% true positives for 1% false positives.

Our automatically engineered feature set includes only 195 features, compared to 545,334 in Drebin; nevertheless, some of these informative features are absent from the manually engineered set. For example, the Drebin system cannot identify the `Gappusin` family, which behaves as a downloader [8]. However, with the automatically engineered feature set we can detect this family by observing that it invokes APIs that leak sensitive data, which have been discussed in the context of privacy threats [40]. Because related concepts are often discussed in disjoint sets of papers, identifying all the relevant links would require human researchers to assimilate the entire body of published knowledge.

In summary, we make the following contributions:

- We propose a semantic network model for representing a growing body of knowledge. This model addresses unique challenges for mining the security literature.
- We propose techniques for synthesizing the knowledge contained in thousands of natural language documents to generate concrete features that we can utilize for training machine learning classifiers.
- We describe FeatureSmith, an automatic feature engineering system. Using FeatureSmith, we generate a feature set for detecting Android malware. This set includes informative features that a manual feature engineering process may overlook, and its effectiveness rivals that of a state-of-the-art malware detection system. FeatureSmith also helps us characterize the evolution of knowledge about Android malware.
- We propose a mechanism that uses our semantic network to generate feature explanations, which link the features to concepts that describe malware behaviors.

For reproducibility, we release the automatically engineered feature set and the semantic network used to generate it at http://featuresmith.org.

The rest of this paper is organized as follows. In Section 2, we review the challenges for automatic feature engineering and we state our goals. In Section 3, we describe the design of FeatureSmith. We explore the semantic network and we evaluate the effectiveness of the features generated in Section 4. Finally, we discuss the related work and the applications of automatic feature engineering to other areas in Sections 5 and 6, respectively.

2. THE FEATURE ENGINEERING PROBLEM

Researchers engineer features for malware detection by reasoning about the *properties that malware samples are likely to have in common* (e.g. they engage in SMS fraud) and the concrete features that reflect these behaviors (e.g. the samples request the `SEND_SMS` permission). These features may not single out the malicious apps; for example, the SMS sending code is typically invoked from an `onClick()` method [53], but this method is prevalent across all Android apps. *Feature selection* methods can rank a list of potential features according their effectiveness (e.g. by using mutual information [28]). However, the initial list is the result of a *feature engineering* process, involving human researchers who rely on their intuition and knowledge of the domain.

In consequence, the feature engineering process is crucial to the effectiveness and applicability of machine learning. This process is laborious and requires researchers to assimilate a growing body of knowledge. For example, for a recent effort to model the Manhattan traffic flows and predict the effectiveness of ride sharing [33], data scientists from New York University invested 30 person-months in identifying and incorporating informative features [14]. Because good machine learning models require a substantial manual effort, labor market estimates project a deficit of 190,000 data scientists by 2018 [29]. In the context of Android malware detection, the Drebin [8] feature set consists of 8 types of features; one type encompasses suspicious API calls. To engineer concrete features of this type, Drebin's designers manually identified 315 suspicious API calls from five categories: data access, network communication, SMS messages, external command execution, and obfuscation. For comparison, the Android framework version (API level 19) utilized by the Drebin authors exported over 20,000 APIs. Moreover, this number keeps growing and exceeds 25,000 in the current version (API level 23). This illustrates a key challenge for the feature engineering process: identifying API calls that may be useful to malware authors requires *extensive domain knowledge and manual investigation.*

Additionally, machine learning techniques can be difficult to deploy in operational security systems, as the trained models detect malware samples but do not outline the reasoning behind these inferences. In consequence, there is a *semantic gap* between the model's predictions and their operational interpretation [41]. For example, a machine learning model that successfully separates malicious and benign apps on a testing corpus by relying primarily on the `onClick()` feature would be useless for detecting malware in the real world. Recent work on explaining the outputs of classifiers generally focuses on providing utility measures (e.g. mutual information) for the features used in the model [8, 38]; however, classifiers trained for malware detection typically use a large number of low level features [8], which may not have clear semantic interpretations. To understand what these malware detectors do, and to gain confidence in their

outputs, the human analysts who use them operationally require explanations that link the outputs of the malware detector with concepts that the analysts associate with malware behavior—a cognitive process known as *semantic priming* [12]. Such *explanations should convey the putative malicious behaviors*, rather than the basic functionality described in developer documents. For example, `sendTextMessage` should be relevant to not only "send SMS message" but also "subscribe premium-rate service"; `RECORD_AUDIO` could be related to "record audio" as well as "record phone call".

Goals. Our *first goal* is to design a general approach for discovering valuable features mentioned in natural language documents about malware detection. These features should be concrete named entities, such as Android API calls, permissions and intents,[2] that we can extract directly from a corpus of malware samples using off-the-shelf static analysis tools. Given a feature type, our approach should discover useful feature instances automatically. This automatic feature engineering approach complements the traditional approach, where data scientists manually create the feature sets based on their own domain knowledge. Specifically, while the manual feature engineering process benefits from human creativity and deep personal insights, the strength of our automatic technique is its ability to draw from a larger body of knowledge, which is increasingly difficult for humans to assimilate fully. Our *second goal* is to rank the extracted features according to how closely they are related to malware behavior. Rather than simply extracting all the features mentioned in the natural language documents, we aim to discover the ones that are considered most informative in the literature. Our *third goal* is to provide semantic explanations for the features discovered, by linking them to abstract concepts discussed in the literature in relation to malware behavior. A *meta-goal* is to implement and evaluate a real system for automatic feature engineering based on these ideas; we select the problem of Android malware detection for this proof of concept.

Non-goals. We aim to engineer informative features for detecting malware in general, rather than malware from a specific data set, so we do not aim to outperform existing malware detection systems in terms of precision and recall (feature selection methods would be better suited to this goal [13, 39]). Because we focus on concrete features, which do not impose additional manual effort for the data collection, we do not extract behaviors that encode more complex operations, such as specific conditions or behavior sequences [17]. For example, from the sentence "send SMS without notification," we extract two behaviors—"send SMS" and "send without notification"—rather than a single behavior with a conditional dependence. In addition, owing to limitations in the state of the art techniques for natural language processing, we expect that some of the features we extract will not be useful (e.g., when they result from parsing errors); however, our highest ranked features should be meaningful and informative. Finally, we do not advocate replacing human analysts completely with automated tools. Instead, we discuss techniques for bridging the semantic gap between the outputs of malware classifiers and the operational interpretation of these outputs, in order to allow security researchers and analysts to benefit from the entire body of published research.

[2]Additional examples include blacklisted URLs, Windows registry keys, or fields from the headers of network packets.

2.1 Alternative approaches

If the features can be enumerated exhaustively (e.g. all the Android permissions and API calls), a feature selection method may be applied to identify a smaller feature set that maximizes the classifier's performance [39]. Representation learning [10] automatically discovers useful features (representations) from raw data; for example, a neural network can derive high level features from low-level API calls, for classifying malware [13]. These data-driven alternatives to manual feature engineering identify the best features to model a given ground truth. However, in security it is generally difficult to obtain a clean ground truth for training malware detectors [19, 41]. For example, VirusTotal [5] collects file scanning results from multiple anti-virus products which, however, seldom reach consensus. We found that some benign apps from the Drebin ground truth are labeled as malicious by some VirusTotal products (Section 4.1). Additionally, anecdotal evidence suggests that adversaries may intentionally poison raw data [21, 48, 50], which results in a biased ground truth. By deriving features from the scientific literature, rather than from the raw data, our approach provides a complementary method for discovering useful features and may help overcome biases in the ground truth.

2.2 Overview of Android malware

Android is a popular operating system for mobile devices such as smartphones. Android provides an API (Application Programming Interface) that allows apps to access system resources (e.g. SMS messages) and functionality (e.g. communicating with other apps). All third-party apps running on the Android platform must invoke these *API calls*. Therefore, the API calls represent informative features for exposing the app behavior. In addition, Android utilizes a *permission* mechanism to protect the user's sensitive information (e.g., phone number, location). For example, apps must request the `SEND_SMS` permission to send text message, and the `ACCESS_FINE_LOCATION` permission to obtain the device's precise location. *Intents* help coordinate different components of an app, for example by making it possible to start an activity or service when a specific event occurs. An example of such an event is `BOOT_COMPLETED`, which allows an app to start right after the system finishes booting. In this paper, we consider an app's permissions, intents, and API calls as potential features for malware detection. Permissions and intents are declared in the app's `Manifest.xml`, while API calls can be extracted with static analysis.

The fine grained permission model [9, 18] and additional security features incorporated in Android make it difficult for apps to behave like traditional desktop malware (e.g. bots, viruses, worms), which can propagate, execute and access sensitive data without requesting the user's permission. In consequence, Android malware exhibits new behaviors—e.g., subscribing to premium-rate service, intercepting SMS messages, repackaging benign apps [53]—that involve specific permissions, intents and API calls. `FakePlayer`, the first Android trojan detected in August 2010, masqueraded as a media player and engaged in SMS fraud [22]. Since then, the volume of Android malware has grown exponentially, and nearly one million malicious apps were discovered in 2014 (17% of all Android apps) [47].

2.3 Challenges for mining security papers

Natural language often contains ambiguities that cannot be resolved without a deep understanding of the subject un-

der discussion. For example, the phrase "sends SMS message "798657" to multiple premium-rate numbers in Russia" [53] implies a malicious behavior to a human reader, but this inference is not based on purely linguistic clues. In another example, the phrase "API calls for accessing sensitive data, such as `getDeviceId()` and `getSubscriberId()`" [8] mentions concrete Android features, but inferring that these features would be useful for malware detection requires understanding that Android malware is often interested in accessing sensitive data. To perform such *commonsense reasoning*, natural language processing (NLP) techniques match the text against an existing *ontology*, which is a collection of categories (e.g. malware samples, SMS messages), instances of these categories (e.g. `FakePlayer` *is a* malware sample) and relations among them (e.g. `FakePlayer` *sends* message "798657") [15]. To this end, specialized ontologies have been developed in other scientific domains, such as medicine [36]. Unfortunately, *security ontologies are in an incipient stage.* CAPEC [3], MAEC [4] and OpenIOC [27] provide detailed languages for exchanging structured information about attacks and malware, but they are not designed for being matched against natural language text.

This reflects a deeper challenge for automatic feature engineering. Ontologies are manually constructed and reflect the known attacks and malware behaviors observed in the real world. In contrast, scientific research is open ended and focuses on novel and theoretical attacks. Moreover, the malware behavior evolves continuously, as adversaries aim to evade existing security mechanisms.

There are also technical challenges for applying existing NLP techniques to the security literature. In other scientific fields, such as biomedical research, papers have structured abstracts, often in the IMRAD format (Introduction, Methods, Results, And Discussion). This has facilitated the use of NLP for mining the biomedical literature [20, 42, 44]. In contrast, the titles and abstracts of security papers are too general to extract useful information for automatic feature engineering. While the paper bodies contain the relevant information, they also include a large amount of abstract concepts and terms that represent noise for the feature engineering system. For example, a ten-page paper may mention a specific malware behavior in only one sentence. In consequence, *extracting concrete features from security papers requires new text mining techniques.*

3. AUTOMATIC FEATURE ENGINEERING

Our automatic feature engineering technique mirrors the human process of reasoning about what malware samples have in common. To this end, we build on ideas from cognitive psychology [12] and represent the knowledge reflected in the security literature as a *semantic network*, with nodes that correspond to the *concepts* discussed in the papers and edges that connect *related* concepts. Rather than utilize a pre-determined set of concepts and relation types (i.e. an ontology),[3] we propose rules to identify interesting concepts (e.g. potential malware behaviors) and we derive edge weights that reflect the *semantic similarity* of two concepts, based on how close the terms are in the text and the frequency of these co-occurrences. This approach derives from the observation that, when humans describe a concept, they tend to mention closely related concepts at first, and then they discuss increasingly less relevant concepts.

[3]Some references use the term *semantic network* as a synonym for *ontology* [15]. The key distinction here is that the categories of malware behavior are not predetermined.

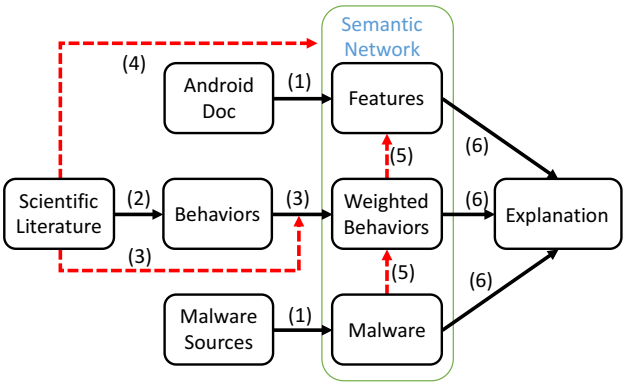


Figure 1: General architecture for automatic feature engineering: (1) data collection (§3.1); (2) behavior extraction from scientific papers (S3.2.1); (3) behavior filtering and weighting (§3.2.2); (4) semantic network construction (§3.3); (5) feature generation (§3.4); (6) explanation generation (§3.5). Black lines indicate the data flow and red dashed lines represent computations.

At a high level, we generate features in two steps. First, we process the scientific literature to extract and organize concepts that are semantically related to the behavior of Android malware. Then we map these concepts to concrete features that we can analyze experimentally. Both these steps are fully automated and require no manual inspection.

Figure 1 illustrates the architecture of FeatureSmith. We first collect named entities for both known malware families (e.g. `DroidKungFu`, `Zsone`, `BaseBridge`) and features (i.e. permissions, intents, and API calls). We also collect scientific papers from a variety of sources. Then, FeatureSmith parses the scientific literature using the Stanford typed dependency parser [16] and processes the dependencies to extract the basic malware behaviors. Next, we mine the papers and construct a semantic network, where the nodes represent the behaviors, the malware families and the concrete features, and the edges indicate which concepts are closely related. We quantify the semantic similarity by assigning edge weights, and we also weight the behavior nodes to focus on the concepts most relevant to Android. Using the semantic network, we calculate a score for each feature and we rank the features based on this score. The score indicates how useful the feature is likely to be for detecting Android malware, according to the current security literature. We utilize the top ranked features, generated in this manner, in a classifier trained to distinguish benign and malicious Android apps. Finally, we generate explanations for why the features selected by this classifier are associated with malware by identifying malware behaviors that are close to these features on the semantic network and by providing links to the papers discussing these behaviors. This technique mirrors the cognitive process of semantic priming [12] and helps human analysts interpret the outputs of our system.

3.1 Data sets

FeatureSmith analyzes three types of data: natural-language documents (e.g. scientific papers), for extracting malware behaviors, lists of named entities related to Android (e.g. development documentation that enumerate permis-

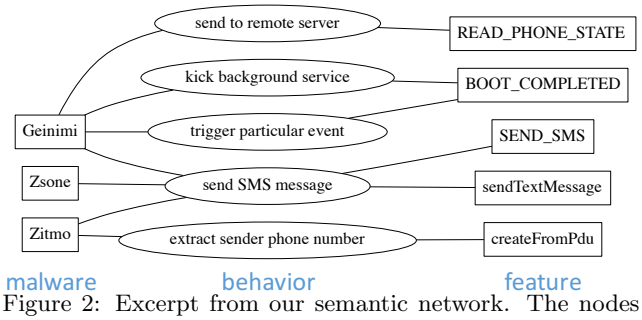


Figure 2: Excerpt from our semantic network. The nodes correspond to malware families, malware behaviors, and concrete features. Unlike in an ontology, the categories of malware behavior are not predetermined.

Table 1: Summary of our data sets.

type	source	number	total
malware	Mobile-Sandbox	210	280
	Drebin	180	
documents	S&P	465	1,068
	Sec	35	
	CSF	327	
	Google	241	
features	permissions	132	11,694
	intents	189	
	API	11,373	

sions, API calls, etc.), for determining which features can be tested experimentally, and malware samples, for validating the feature generation process. Table 1 summarizes these data sets. In this section, we discuss the data collection process and the pre-processing we apply to each type of data.

Documents. Our primary data source consists of scientific papers. We utilize these papers to extract Android malware behaviors and to construct the semantic network. From the electronic proceedings distributed to conference participants, we collect the papers from the IEEE Symposium on Security and Privacy (S&P'08–S&P'15)[4], the Computer Security Foundations Symposium (CSF'00–CSF'14), and USENIX Security (Sec'11). We complement this corpus by searching Google Scholar with the keywords *"Android malware"*, and then we download the PDF files if a download link is provided in the query results. This process may result in duplicate papers, if a returned paper already exists in our corpus. Therefore, we record the hash of all the papers in our corpus, and remove a PDF document if the file hash already exists in the data set.[5] In total, our corpus includes 1,068 documents. Other data sources (e.g. industry reports, analyst blogs) could be informative, but we only collect peer reviewed papers to ensure the quality of the corpus.

We extract the text from the papers in PDF format, for later processing. Extracting clean text from PDF files is a non-trivial task as it is difficult to identify figures, tables, algorithms and section titles embedded in the body content. We develop several heuristics to address this problem. We convert the PDF files to text with the Python `pdfminer` package, which also allows us to record the corresponding font style and size. We consider that the body of the paper is written in the most frequently used font in the document. We extract all the text in this font, as well as single words in a different font but within the body content, which likely represent emphasized words. This excludes the paper titles and the section headings; however, we found that this information is not necessary for automatic feature engineering. Conversely, we also experimented with utilizing only the paper abstracts, which are readily available on publisher web sites, but we found that they are insufficient for our task.

[4]Including workshop papers.

[5]It is possible that the same paper may have multiple hashes, for instance owing to multiple versions of the same paper. We believe such cases are uncommon, and we do not attempt to detect duplicated papers based on content similarity.

Features. The features utilized for Android malware detection must be *representative*, to capture the behavior of various malware families, and *informative*, to distinguish the malware from benign apps. In this paper, we focus on permissions, intents, and API calls as potential features for malware detection. We collect all the permissions, intents and API calls from Android developer documents [1]. Then, we ignore the class name for each feature, because we have found that class names are not mentioned in most papers. However, removing the class name introduces ambiguity in two cases: (1) the feature name coincides with a word or abbreviation that could be frequently mentioned; (2) methods from different classes have the same name. For the first case, we check if the function names can be split into several word components based on the naming rules. For example, we could split `onCreate` into `on` and `Create`, and `SEND_SMS` into `SEND` and `SMS`. Then we remove all the features that cannot be split in this manner, which are more likely to collide with other words and cause ambiguity. For the second case, most of identified informative features are not ambiguous, e.g. `sendTextMessage`. For those ambiguous names, they often have only one meaning in papers. For example, `getDeviceId` could be the method in either `Telephony` or `UsbDevice`, but the method refers to `Telephony.getDeviceId` in almost every paper. In total, we have 132 permissions, 189 intents (including both name and value), 11,373 API calls.

Malware families. We collect the malware family names from both the Drebin dataset [8] and from a list of malware families [2] caught by the Mobile-Sandbox analysis platform [43]. In total, we collect 280 malware names. We utilize these names when mining the papers on Android malware to identify sentences that discuss malicious behaviors. In addition to the concrete family names, we also utilize the term "malware" and its variants for this purpose.

For our experimental evaluation, we utilize malware samples from the Drebin data set [8], shared by the authors. This data set includes 5,560 malware samples, and also provides the feature vectors extracted from the malware and from 123,453 benign applications. While these feature vectors define values for 545,334 features, FeatureSmith can discover additional features, not covered by Drebin. We therefore extract these additional features from the apps.

We first select all malware samples and a random sample of equal size, drawn from the benign apps. As the Drebin data set includes only malware samples, we download the benign apps from VirusTotal [5], by searching for the corresponding file hashes. After collecting the `.apk` files for all the apps, we use `dex2jar` to decompile them to `.jar` files, and use `Soot` [49] to extract all the Android API calls. This allows us to expand the feature vectors and test the features omitted by Drebin.

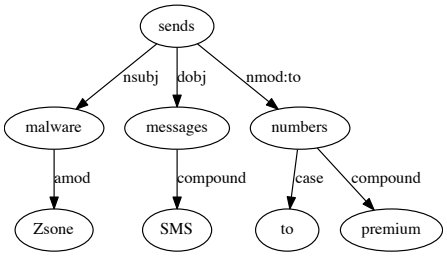


Figure 3: Typed dependency of the sentence *"Zsone malware sends SMS messages to premium numbers"*.

We obtain the expanded feature vectors for 5,552 malware samples and 5,553 benign apps.[6] The collected applications exhibit 43,958 out of 545,334 Drebin features and 133 out of 195 features generated by FeatureSmith. Note that we use the malware samples only for the evaluation in Section 4; the feature generation utilizes the malware names and the document corpus.

3.2 Behavior extraction

We extract malware behaviors discussed in the security literature in two steps: first, we identify phrases that may correspond to malware behaviors, and then we apply filtering and weighting techniques to find the most relevant ones.

3.2.1 Behavior collection

We define a *behavior* as a tuple that consists of subject, verb and object, where either subject or object could be missing. Single words or multi-word expressions are not sufficient to provide a semantic meaning without ambiguity. For example, *number* could refer to *phone number* or *random number* due to a missing modifier, and *data* could refer to *steal data* or *inject data* due to the missing verb. Therefore, we define behavior as a basic primitive in our approach.

We use the Stanford typed dependency parser [16] to decompose the complex sentence and construct behaviors.[7] The parser predicts the grammatical relationships between words and labels each relationship with a type. Figure 3 shows the output from dependency parser for the sentence *"Zsone malware sends SMS messages to premium numbers"*. The head of each relation is called *governor* and the tail is called *dependent*. The parser can also identify the grammatical relation for the words in the clause.

Behaviors are constructed from certain typed dependency and part-of-speech as listed on Table 2. We complete the missing component in behaviors if another behavior with identical verb is found. Furthermore, we extend the subject and object to noun phrases by adding adjective modifiers and identifying multi-word expressions. To reduce the number of word variants, we apply WordNet [31] to lemmatize words based on their part of speech. Table 3 shows one example of behavior extraction. From typed dependencies, we decompose a complex sentence into several simple relations.

3.2.2 Filtering and weighting

The previous step produces 339,651 unique behaviors. To determine which behaviors are most relevant to Android malware, we assign weights that capture how semantically close these behaviors are to the malicious functionality. We determine the weights in three steps:

1. Filtering: select behaviors related to Android applications and remove all the irrelevant behaviors.
2. Word weighting: assign the weights for both verbs and noun phrases based on how semantically close they are to the term *Android*.
3. Behavior weighting: assign the weights to each behavior based on the weight of subject, verb and object.

We do not assign weights to behaviors directly because many behaviors appear only few times in our paper corpus, which might bias our metrics.

In the first step, we select the behaviors in the paper that contains the term *Android*. If the document is about Android, then it must have the word *Android* at least once. Under this assumption, we are able to remove most of behaviors that are unlikely relevant to Android, and obtain 82,035 behaviors.

In the second step, we collect all the noun phrases from subject and object and verbs in filtered behaviors. In total, we have 47,186 noun phrases and 1,682 verbs. Then, we evaluate the importance of each word[8] by computing the *mutual information* of the word and the term *Android*; we do this for both the verbs and noun phrases from the filtered behaviors. Formally, mutual information compares the frequencies of values from the joint distribution of two random variables (whether the two terms appear together in a document) with the product of the frequencies from the two distributions that correspond to the individual terms. Mutual information measures how much knowing one value reduces uncertainty about the other one and is widely utilized in text classification. However, in our case mutual information

Table 2: Rules for matching behaviors. <gov> and <dep> represent the governor word and dependent word in the typed dependency.

Rules	Behavior		
dependency type	subj	verb	obj
dobj		<gov>	<dep>
nsubj	<dep>	<gov>	
nsubjpass		<gov>	<dep>
nmod:agent	<dep>	<gov>	
nmod:to		<gov>_to	<dep>
nmod:with		<gov>_with	<dep>
nmod:from		<gov>_from	<dep>
nmod:over		<gov>_over	<dep>
nmod:through		<gov>_through	<dep>
nmod:via		<gov>_via	<dep>
nmod:for		<gov>_for	<dep>

Table 3: An example of behavior extraction.

text	"For instance, the Zsone malware is designed to send SMS messages to certain premium numbers, which will cause financial loss to the infected users." [54]
behaviors	design for instance
	design Zsone malware
	Zsone malware send SMS message
	Zsone malware send to certain premium number
	certain premium number cause financial loss
	certain premium number cause to infected user

[6]For a few applications, we were unable to either decompile them or extract the method calls.

[7]We apply both *collapsed* and *ccprocessed* options. The former is to simplify the relationship with fewer prepositions and the latter is to propagate the dependency if a conjunction is found.

[8]We use the term "word" for both single words and phrases.

Table 4: Top 5 behaviors related to Android.

rank	behavior
1	Over-privileged apps overstep permission
2	manufacturer customize smartphone OS
3	malware author download Android's source code
4	download from official Android Market
5	download from app store

tends to find general words like *app* but ignores the less frequent words like *screenshot*. To solve this problem, we scale the mutual information by the entropy of the word. The weight of word $S(w)$ is calculated using Equation (1), where $H(w)$ is the entropy of word w, $I(w; Android)$ is the mutual information between word w and word *Android*. This metric captures the *fraction* of uncertainty of word w given the word Android, and the value ranges from 0 to 1.

$$S(w) = \frac{I(w; Android)}{H(w)} = 1 - \frac{H(w|Android)}{H(w)} \quad (1)$$

Although the top ranked words might still be general, we are also able to identify the words with low document frequency but related to Android, e.g. *battery*, *wallpaper*, *camera*.

In the last step, we assign an initial weight for each behavior based on the weights of verb and noun phrases. The behavior weight is the product of the verb weight and the maximum noun phrase weight.[9] Table 4 shows the behaviors with highest weights. Note that this is just the initial weight for how close the behavior related to Android; the ranking of behaviors will change during feature generation.

3.3 Semantic network construction

We model the concepts discussed in the security literature using a semantic network, defined as an undirected graph $G = (V, E)$. The set of vertices V includes the concepts extracted, and the set of edges E captures the pairwise relations among these concepts. Each edge has a weight, which captures the semantic similarity of the two concepts linked.

There are three types of nodes in the semantic network: malware families V_{mal} (Section 3.1), behaviors V_{behav} (Section 3.2), and features V_{feat} (Section 3.1). We define two types of edges: (1) links between malware and behaviors $E_{mb} = \{\{u, v\}, \forall u \in V_{mal}, \forall v \in V_{behav}\}$; and (2) links between behaviors and features $E_{fb} = \{\{u, v\}, \forall u \in V_{behav}, \forall v \in V_{feat}\}$. An edge may not connect two nodes of the same type. Nevertheless, two concepts from the same set may be semantically related; for example, an API call might require certain permissions; and two malware families could have a shared module. We can establish these connections by traversing one or more hops on the semantic network. This approach has the benefit that the path between two concepts preserves the intermediate concepts (the API call and the shared module, in our previous example), which helps the reasoning process.

We create an edge if two nodes appear within N sentences for no less than M times. In our experiments, we set $N = 3$ and $M = 1$. However, using a larger N could on the contrary introduce more noise. M is another parameter to balance the precision and recall. Because we aim to identify novel ideas, rather than common sense, we choose a small M. Each edge is weighted by M. If two nodes appear together frequently, then these two concepts are more likely to be related. Figure 2 shows part of our semantic network.

[9]If the word is not in the dictionary, then the score is 0.

Table 5: Top 5 features.

rank	feature	type
1	`sendTextMessage`	API method
2	`SEND_SMS`	permission
3	`BOOT_COMPLETED`	intent
4	`RECEIVE_SMS`	permission
5	`onStart`	API method

3.4 Feature generation

The concrete features generated correspond to Android permissions, intents, and API calls, and they are identified as described in Section 3.1. We utilize the semantic network to rank the features and to determine which ones are most relevant for detecting Android malware.

Let M, B, F be three random variable for malware, behavior and feature respectively. We compute the probability of feature π_F from the probability of malware π_M using Equation (2):

$$\pi_F = \pi_M * P_{B|M} * P_{F|B} \quad (2)$$

The transition probability $P_{B|M}$ and $P_{F|B}$ is estimated using the edge weight of semantic network E and weight of behaviors W by Equation (3):

$$P_{B|M}(b|m) = \frac{E(b,m)W(b)}{\sum_b E(b,m)W(b)}$$
$$P_{F|B}(f|b) = \frac{E(f,b)}{\sum_f E(f,b)} \quad (3)$$

In our experiments, we assign equal probabilities for all the malware nodes since our goal is to find general features for Android malware detection. The intuition behind this equation is that the most informative features correspond to some malicious behaviors that are shared by multiple malware families, as captured by the edge weights and the number of incoming edges. Additionally, we consider the behavior weights to ensure that we propagate a higher weight to the behaviors that are closely related to Android.

Table 5 shows the top 5 features extracted in this manner. The `sendTextMessage` method and the `SEND_SMS`, `RECEIVE_SMS` permissions correspond to apps that send text messages. The behaviors that contribute to these two features are also related to text messages, e.g. "*send SMS message*", "*subscribe premium-rate service*". Malware often listens for the `BOOT_COMPLETED` event, which indicates that the system finished booting. The corresponding behavior contains *"register for related system-wide event"* and *"kick off background service"*. Papers using static or dynamic analysis often mention `onStart`, as it is usually an entry point for malware behavior. This feature can be reached from multiple behavior nodes, e.g. "*send data to server*" and "*register premium-rate service*", as it may be involved in various malicious activities. Besides, some other features related to user's sensitive information have high rank, for example, `getDeviceId` and `READ_PHONE_STATE`. The corresponding behaviors reveal the malicious actions like "*return IMEI*" and "*return privacy-sensitive information*".

3.5 Automatic explanations

FeatureSmith generates explanations for each informative feature, consisting of the related malware samples, behaviors and literature references. Starting from a feature, we follow the links in the semantic network back to the behaviors that contribute to the feature. We first define the contribution of behavior b to the feature f as the joint probability

$P_{FB}(f, b) = \pi_B P_{F|B}(f|b)$. Intuitively, the contribution is the probability that feature f received from behavior b. We then rank behaviors based on their contribution. Synthesizing coherent explanations in natural language requires NLP techniques that are out of scope for this paper; instead, FeatureSmith simply outputs the behaviors recorded in the semantic network. Next, we return the sentences if the feature and corresponding behaviors occur within a sentence-based window. The original text from papers can help the FeatureSmith users better understand the related behaviors and to reason about the utility of the extracted features.

To increase the relevance of the behaviors returned, we could manually create a stopword list to filter out some behavior that are obviously irrelevant to malware or are too general to provide any information, e.g. "show in Figure", "for example". Stopwords list will affect the propagation in step (5) in Figure 1.

4. EVALUATION RESULTS

We evaluate FeatureSmith by measuring the effectiveness of the automatically generated features. In our experiments, we utilize a corpus of malicious and benign Android apps, collected as described in Section 3.1. We train random forest classifiers [26] with (i) the features generated by FeatureSmith and (ii) the manually engineered features from Drebin [8]. We compare the performance of these classifiers in Section 4.1. In Section 4.2, we drill down into FeatureSmith's ability to discover informative features that may be overlooked during the manual feature engineering process. Finally, we characterize the evolution of our community's knowledge about Android malware in Section 4.3.

4.1 Feature effectiveness

To evaluate the overall effectiveness of automatically engineered features, we train 3 random forest classifiers with the same ground truth but different feature sets:

- F_S: All features from FeatureSmith
- F'_S: Top 10 features from FeatureSmith ($F'_S \subseteq F_S$)
- F_D: Drebin features ($F_S \nsubseteq F_D$)

We randomly select 2/3 of apps for our training set and utilize the rest for the testing set. We choose the random forest algorithm, which trains multiple decision trees on random subsets of features and aggregates their votes for the final prediction, because this technique is less prone to overfitting than other classifiers [26].

Figures 4a and 4b compare the performance of the three classifiers using a receiver operating characteristic (ROC) plot. This plot illustrates the relationship between false positives and true positives rates of these classifiers. The figure suggests that *automatically and manually engineered features are almost equally effective*, as the ROC curves are practically indistinguishable. At 1% false positive rate, the classifiers using F_D and F_S both have 92.5% true positives.[10] F_S contains much fewer features compared to F_D (173 instead of 43,958 and 44 in common), but this dimensionality reduction does not degrade the performance of classifier. The features themselves are not equally informative; if we randomly select 173 features from F_D, the ROC curve is close to the diagonal, which means that the classifier is equivalent to making a random guess. This suggests that FeatureSmith is able to discover representative and informative features from scientific papers. When using only the top 10 features suggested by FeatureSmith (feature set F'_S), our classifier achieves 44.9% true positives for 1% false positives. This is comparable to the performance of three older malware detection techniques, which provide detection rates between 10%–50% at this false-positive rate [8]. This shows that FeatureSmith's ranking mechanism *singles out the most informative features for separating benign and malicious apps.*

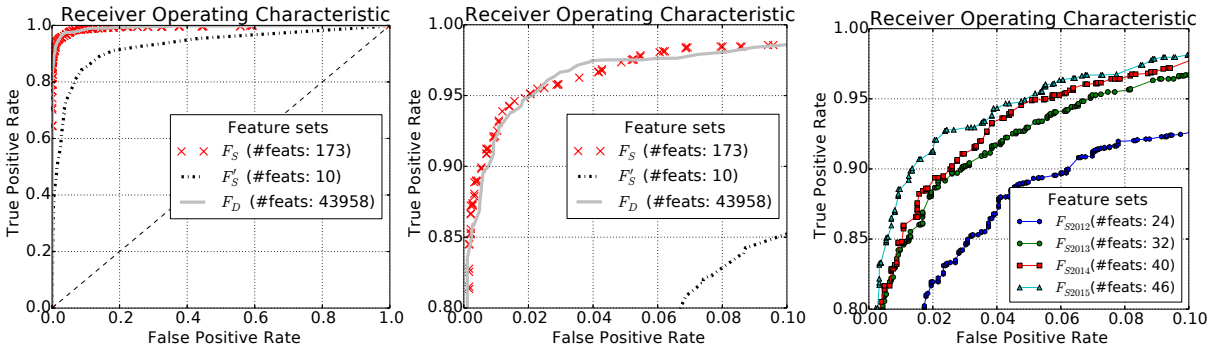
We further examine all the false positive results (18 apps) from the testing set. 8 apps are labeled as malicious by at least one of VirusTotal's anti-virus products, perhaps because they were determined to be malicious after the Drebin paper was published. Although these apps are considered benign in our dataset, they are actually malicious, which suggests that our real false positive rate may be even lower. Other benign apps from our false positive set exhibit behavior similar to malware, including two Chinese security apps, which intercept incoming phone calls and filter spam short messages, one Korean parental supervision app, which tracks a child's location, and a banking app. We could not find any information about the remaining 6 apps.

4.2 Tapping into hidden knowledge

We evaluate the contribution of individual features to the classifier's performance by using the mutual information metric [28]. Intuitively, mutual information quantifies the loss of uncertainty for malware detection when the app has the given feature. Table 6 lists the 5 features with the highest mutual information. When present together, these features indicate an app that triggers some activity right after booting the system, starts a background service, accesses sensitive information and sends SMS messages. FeatureSmith ranks these features in the top-60, and the three best features in the top-11.

To provide a baseline for comparison, we also compute a simpler ranking that, unlike FeatureSmith, does not take into account the semantic similarity between features and malicious behaviors. We extract all the API calls, intents and permissions mentioned in our paper corpus, whether they are related to malware or not, and we rank them by how often they are mentioned. This term frequency (TF) metric is commonly used in text mining for extracting frequent keywords. This ranking does not place the features from Table 6 among the top features. For example, `BOOT_COMPLETED` and `RECEIVE_BOOT_COMPLETED` are not mentioned frequently in papers, and therefore have a low TF rank. Figure 5 shows the cumulative mutual information for the top 150 features in the FeatureSmith and TF rankings. Because it uses a semantic network, FeatureSmith assigns consistently higher ranks for the features more likely to be related to malware, *even if they are not mentioned very frequently.* Additionally, we compute the Kendall rank correlation [23] between FeatureSmith's ranking and the mutual information based ranking, and perform a Z-test to determine if the two ranking systems are correlated. The p-value is $1.9 * 10^{-4}$ (<0.05) which demonstrates that FeatureSmith based ranking is statistically dependent with the mutual information based ranking. We repeat the hypothesis test for the TF based ranking, and we obtain a p-value is 0.14 (>0.05).

[10] We note that our goal is not to reproduce or exceed the performance of the Drebin malware detector—we use random forests while Drebin uses SVM—but to perform a fair comparison of the feature sets. Nevertheless, our classifier using F_D achieves the same performance as reported in the Drebin paper [8].



(a) Complete. (b) Zoom-in. (c) Changes over time.

Figure 4: ROC curve of malware detection for classifiers with different feature sets (including the count of features utilized from each set, as the apps in our corpus exhibit a subset of the manually and automatically engineered features).

Table 6: 5 most informative features.

feature	MI	#usage		ranking	
		malicious	benign	FeatureSmith	Keyword-TF
`BOOT_COMPLETED`	0.27	3,555(64%)	441(8%)	3	151
`SEND_SMS`	0.26	3,227(58%)	302(5%)	2	9
`READ_PHONE_STATE`	0.22	5,011(90%)	2,236(40%)	11	16
`startService`	0.18	3,408(61%)	791(14%)	60	37
`RECEIVE_BOOT_COMPLETED`	0.17	2,672(48%)	373(7%)	54	351

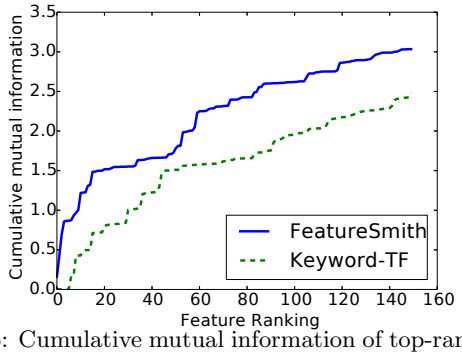


Figure 5: Cumulative mutual information of top-ranked features.

Table 7: An example of feature explanation.

	`getSimOperatorName`
Behavior	return privacy-sensitive information leak privacy-sensitive return value leak to remote web server ...
Reference	[37]: *Examples of such methods are* `getSimOperatorName` *in the TelephonyManager class (returns the service provider name),* `getCountry` *in the Locale class, and* `getSimCountryIso` *in the TelephonyManager class (both return the country code), all of which are correctly classified by SUSI.*

Among the features with a low mutual information, we also find several instances that are related to malware behaviors. For example, FeatureSmith identifies `createFromPdu`, `getOriginatingAddress` and `getMessageBody` from [51], which are used in Zitmo for extracting the message sender phone number of message content. FeatureSmith also identifies `onNmeaReceiced` and `onLocationChanged`, which could potentially leak location data [37], and `isMusicActive`, which can be used to infer the user's location [52]. These features do not help the classifier, as they might not be representative of the malware families from the Drebin malware data set, or the data set might not cover all the malware behavior. Nevertheless, these features provide useful information to researchers interested in malware behavior.

FeatureSmith generates several informative features that are not included in the Drebin feature set. For example, `getSimOperatorName` is mentioned in two papers, as a method that apps often call after requesting the `READ_PHONE_STATE` permission [7] and as a method that leaks private data [37]. `getNetworkOperatorName` is another method that potentially leaks private data [40]. These two API calls are not among the manually engineered Drebin features, but they have a high mutual information for malware detection. 884 malware samples invoke `getSimOperatorName`, compared to 85 benign apps; `getNetworkOperatorName` appears in 1,341 malware samples and in 378 benign apps. This suggests that *automatic feature engineering is able to mine published information that remains hidden to the manual feature engineering process*, as human researchers and analysts are unable to assimilate the entire body of publicly available knowledge.

FeatureSmith can extract informative features effectively, but it can also generate explanations for features. For example, the behaviors associated to `BOOT_COMPLETED` reveal that this feature could be an indicator of starting background service for the malware. Instead of providing just a basic description of the feature, extracted from Android developer documents, the explanation links the feature to malware behaviors reported in the literature. Besides the `BOOT_COMPLETED` example, many features are related to *"steal sensitive information"* behavior, which will never be identified by parsing developer documents only. Table 7 shows an explanation for an API call that leaks personal data. These *explanations refer to abstract concepts that human analysts associate with malware behavior* and provide semantic insights into the operation of the malware detector, which is key for operational deployments of such detectors [41].

4.3 Knowledge evolution over time

Our results from the previous section suggest that manual feature engineering may overlook some informative features, perhaps because it is challenging for researchers and analysts to consider the entire body of published knowledge. In this section, we characterize the growth of FeatureSmith's semantic network, which is a representation of the existing knowledge about Android malware. Intuitively, as we add more documents to the system, we create more behavior nodes, and the underlying structure of the network reflects the semantic similarity among these behaviors. We construct several versions of the semantic network, by considering only the papers published before 2010, 2012, and 2014, respectively. We determine the publication year by extracting the paper's title, as the text using the largest font, and by querying Google Scholar with this title. In total, we are able identify the publication year for 913 papers.

We also investigate how this evolution affects our ability to engineer effective features for malware detection. We utilize FeatureSmith to generate the features from the literature published in different years. Figure 4c shows the ROC curve of the classifier trained using features discovered in different years. The figure show that, as more papers are published over time and knowledge accumulates, FeatureSmith is able to generate more informative features and the performance of the corresponding classifier improves. At a 1% false positive rate, the true positive rate increases from 73.1% in 2012 to 89.2% in 2015.[11] In addition, we use the classifiers in different years to detect the malware samples from different families. We determine the threshold by setting a fixed 1% false positive rate. With a growing knowledge on malware behaviors, the classifier performs better. For example, we are able to detect most of the samples from the `Gappusin` family using the classifier in 2014, while we cannot detect any apps from this family using the classifier in 2012. In 2012, the feature set primarily consists of the permissions and API calls related to some obvious behaviors like SMS fraud. However, in the later years, the publications started covering functions that could leak sensitive information. As a result, we can detect `Gappusin` using the features extracted two years later.

5. RELATED WORK

Literature-based discovery. Research on mining the scientific literature dates back to Swanson [45], who hypothesized that fish oil could be used as a treatment for Raynaud's disease by observing that both had been linked to blood viscosity in disjoint sets of papers. Building on this observation, Swanson *et al.* [46] designed the Arrowsmith system for finding such missing links from biomedical articles. To reduce false positives, the system relies on a long list of stopwords and can only process the paper abstracts. Follow-on work proposed additional techniques, e.g. clustering [44] and latent semantic indexing (LSI) [20], but still focuses on either abstracts or titles. More recently, Spangler *et al.* mine paper abstracts and suggest kinases that are likely to phosphorylate the protein p53, by using all the single words and bigrams as the features but without checking whether all the features are meaningful [42]. In contrast to these approaches, we mine document bodies, we propose rules for extracting multi-word malware behaviors and we link these behaviors to concrete Android features.

Semantic networks are based on cognitive psychology research [12] that observed that concepts that are mentioned together in natural language are more likely to be related, which provides a mechanism for estimating the semantic similarity of two concepts. IBM Watson utilized a semantic network for answering Jeopardy! questions from the "common bonds" and "missing links" categories [11]. Two questions are solved by searching for the entities that are close on the semantic network to the entities provided in the question. Our approach differs from the previous work on semantic networks in two aspects. The nodes in our semantic graph are behaviors (verb phrases instead of single words or noun phrases), as these behaviors are more meaningful for capturing the malicious actions. Another difference is that our semantic network is a tripartite graph, which mirrors the malware-behavior-feature reasoning process and which reduces the computation time.

In security, few references rely on NLP techniques. Neuhaus *et al.* [32] used Latent Dirichlet Allocation to build topic models for the CVE database, and analyze vulnerability trends. Pandita *et al.* [34] proposed a framework for identifying permissions needed from Android app descriptions. Liao et al. [25] mine Indicators of Compromise (IOCs) from industry blogs and reports by matching them to the OpenIOC ontology [27]. Instead, we focus on automated feature engineering for malware detection and we extract open-ended behaviors, rather than concepts from a predetermined ontology.

Android malware. Zhou *et al.* conducted the first systematic analysis of Android malware behaviors, from the initial infection to the malicious functionality [53]. As these behaviors often require specific Android permissions, Felt *et al.* [18] and then Au *et al.* [9] proposed static analysis tools to analyze the Android permission specification.

Subsequently, considerable efforts have been devoted to detecting Android malware, ranging from static and dynamic analysis [51, 54] to machine learning techniques [6, 8, 35]. Approaches based on static or dynamic analysis typically propose heuristics or anomaly detection strategies for identifying malware. Zhou *et al.* first apply permission-based filtering to filter out most of apps that are unlikely to be malicious, and then generate behavioral footprints for from static and dynamic analysis [54]. Zhang *et al.* construct API dependency graphs for each app, and identify the malware by detecting anomalies on these graphs [51].

Machine learning techniques typically model malware detection as a binary classification problem. Peng *et al.* applied a Naive Bayes model to assess how risky apps are given the permissions they request [35]. Aafer *et al.* used k-nearest neighbors and extracted Android API calls as features [6]. Arp *et al.* built the Drebin system, which utilizes features extracted from the manifest file and from the bytecode (including permissions, intents, network addresses, API calls, etc.) and trains an SVM classifier for malware detection [8]. In all these cases, the features are the result of a manual engineering process; we complement these efforts by proposing an automatic feature engineering technique.

6. DISCUSSION

The fundamental reason why we can extracting salient malware features from scientific papers is that researchers tend to show the useful features and ignore the features that

[11] Because we cannot identify the publication years for some documents downloaded from Google Scholar, in this experiment the true positive rate does reach our top rate of 92.5%.

do not work. Additionally, as the publication process focuses on novelty, papers often show examples that are absent in the prior work. This enables us to extract features automatically by mining scientific papers. Moreover, the features that are not related to malware are seldom mentioned in the papers, which facilitates the feature mining process.

In some cases, the relationship between malware and features is not stated explicitly. For example, researchers could illustrate the behavior of malware without mentioning any specific API calls; similarly, when analyzing the Android API, researchers may list the calls that leak personal data without mentioning specific malware families. In these cases, the middle behavior nodes from our semantic network help us link the malware to features. These nodes also allow us to discover more related features from Android developer documents. These documents illustrate the functionality of API calls, which reveals the relationships between behaviors and features. This allows us to fill some gaps left in the research papers on Android malware.

A potential direction for further improving FeatureSmith is to combine the behaviors with the same semantic meaning. One method is to manually create a task-specific ontology, which would require an intensive annotation effort. An alternative solution is to utilize word embeddings, for example word2vec [30], which could allow us to determine whether two behaviors are identical. Another benefit from embedding words or behaviors with the semantic meaning is constructing behavior sequences. For example, we could identify features that represent the initial step in a sequence of actions, such as the `onClick` feature that is usually the entry point of the malicious activity.

FeatureSmith provides a general architecture for extracting informative features from natural language, which could be adapted to other security topics. For example, we could extract the features for iOS or Windows malware by using a different set of concrete malware families and features. However, our feature engineering process works under the assumption that a feature is be a named entity. If the features are associated with some operations, such as "max", "number of", our current implementation cannot identify these features automatically. Besides malware detection using function calls as features, network protocols is another area where we can identify a large amount of named entities. For example, instead of malware we could look at network attacks and instead of API calls we could utilize various fields from protocol packets.

7. CONCLUSIONS

We describe the FeatureSmith system that automatically engineers features for Android malware detection by mining scientific papers. The system's operation mirrors the human feature engineering process and represents the knowledge described using a semantic network, which captures the semantic similarity between abstract malware behaviors and concrete features that can be tested experimentally. FeatureSmith incorporates novel text mining techniques, which address challenges specific to the security literature. We use FeatureSmith to characterize the evolution of our body of knowledge about Android malware, over the course of four years. Compared to a state-of-the-art feature set that was created manually, our automatically engineered features shows no performance loss in detecting real-world Android malware, with 92.5% true positives and 1% false positives. In addition, FeatureSmith can single out informative features that are overlooked in the manual feature engineering process, as human researchers are unable to assimilate the entire body of published knowledge. We also propose a mechanism for utilizing our semantic network to generate feature explanations, which link the features to human-understandable concepts that describe malware behaviors. Our semantic network and the automatically generated features are available at http://featuresmith.org.

Acknowledgments

We thank Hal Daumé and Jeff Foster for their feedback. We also thank the Drebin authors for giving us access to their data set. This research was partially supported by the National Science Foundation (grant 5-244780) and by the Maryland Procurement Office (contract H98230-14-C-0127).

References

[1] Android developer documents. http://developer.android.com/index.html.

[2] Android malware family list. http://forensics.spreitzenbarth.de/android-malware/.

[3] Common attack pattern enumeration and classification (CAPEC). https://capec.mitre.org.

[4] Malware attribute enumeration and characterization MAEC. https://maec.mitre.org/.

[5] Virus total. www.virustotal.com.

[6] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[7] H. Agematsu, J. Kani, K. Nasaka, H. Kawabata, T. Isohara, K. Takemori, and M. Nishigaki. A proposal to realize the provision of secure android applications–adms: An application development and management system. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 677–682. IEEE, 2012.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[9] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[10] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[11] J. Chu-Carroll, E. W. Brown, A. Lally, and J. W. Murdock. Identifying implicit relationships. *IBM Journal of Research and Development*, 56(3.4):12–1, 2012.

[12] A. M. Collins and E. F. Loftus. A spreading-activation theory of semantic processing. *Psychological review*, 82(6):407, 1975.

[13] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[14] DARPA. DARPA goes "Meta" with machine learning for machine learning. http://www.darpa.mil/news-events/2016-06-17, 2016.

[15] E. Davis and G. Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Commun. ACM*, 58(9):92–103, 2015.

[16] M.-C. De Marneffe and C. D. Manning. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8. Association for Computational Linguistics, 2008.

[17] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for android malware detection. *Computers and Security*, 49(C):255–273, 2015.

[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638, 2011.

[19] C. Gates and C. Taylor. Challenging the anomaly detection paradigm: a provocative discussion. In *Proceedings of the 2006 workshop on New security paradigms*, pages 21–29. ACM, 2006.

[20] M. D. Gordon and S. Dumais. Using latent semantic indexing for literature based discovery. 1998.

[21] C. Kanich, N. Chachra, D. McCoy, C. Grier, D. Y. Wang, M. Motoyama, K. Levchenko, S. Savage, and G. M. Voelker. No plan survives contact: Experience with cybercrime measurement. In *CSET*, 2011.

[22] Kaspersky Lab. First SMS Trojan detected for smartphones running Android. http://www.kaspersky.com/about/news/virus/2010/First_SMS_Trojan_detected_for_smartphones_running_Android, Aug 2010.

[23] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[24] P. O. Larsen and M. von Ins. The rate of growth in scientific publication and the decline in coverage provided by science citation index. *Scientometrics*, 84(3):575–603, 2010.

[25] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah. Acing the IOC game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[26] A. Liaw and M. Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[27] MANDIANT. The OpenIOC framework. http://www.openioc.org/.

[28] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[29] McKinsey Global Institute. Game changers: Five opportunities for US growth and renewal, Jul 2013.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[31] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[32] S. Neuhaus and T. Zimmermann. Security trend analysis with CVE topic models. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 111–120. IEEE Computer Society, 2010.

[33] M. Ota, H. Vo, C. Silva, and J. Freire. A scalable approach for data-driven taxi ride-sharing simulation. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 888–897. IEEE, 2015.

[34] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: towards automating risk assessment of mobile applications. In S. T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 527–542. USENIX Association, 2013.

[35] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In T. Yu, G. Danezis, and V. D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 241–252. ACM, 2012.

[36] D. M. Pisanelli. *Ontologies in medicine*, volume 102. IOS Press, 2004.

[37] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[38] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[39] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[40] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi. Droidtrack: Tracking and visualizing information diffusion for preventing information leakage on android. *Journal of Internet Services and Information Security (JISIS)*, 4(2):55–69, 2014.

[41] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 305–316. IEEE Computer Society, 2010.

[42] S. Spangler, A. D. Wilkins, B. J. Bachman, M. Nagarajan, T. Dayaram, P. Haas, S. Regenbogen, C. R. Pickering, A. Comer, J. N. Myers, et al. Automated hypothesis generation based on mining scientific literature. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1877–1886. ACM, 2014.

[43] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In S. Y. Shin and J. C. Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1808–1815. ACM, 2013.

[44] J. Stegmann and G. Grohmann. Hypothesis generation guided by co-word clustering. *Scientometrics*, 56(1):111–135, 2003.

[45] D. R. Swanson. Fish oil, raynaud's syndrome, and undiscovered public knowledge. *Perspectives in biology and medicine*, 30(1):7–18, 1986.

[46] D. R. Swanson and N. R. Smalheiser. An interactive system for finding complementary literatures: a stimulus to scientific discovery. *Artificial intelligence*, 91(2):183–203, 1997.

[47] Symantec Corporation. Symantec Internet security threat report, volume 20, April 2015.

[48] K. Thomas, C. Grier, and V. Paxson. Adapting social spam infrastructure for political censorship. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2012.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[50] D. Y. Wang, S. Savage, and G. M. Voelker. Juice: A longitudinal study of an seo botnet. In *NDSS*, 2013.

[51] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[52] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 915–930. IEEE, 2015.

[53] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109. IEEE Computer Society, 2012.

[54] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.