ABSTRACT

Title of dissertation: AUTOMATIC FEATURE ENGINEERING FOR DISCOVERING AND EXPLAINING MALICIOUS BEHAVIORS

Ziyun Zhu
Doctor of Philosophy, 2019

Dissertation directed by: Professor Tudor Dumitraș
Department of Electrical and Computer Engineering

A key task of cybersecurity is to discover and explain malicious behaviors of malware. The understanding of malicious behaviors helps us further develop good features and apply machine learning techniques to detect various attacks. The effectiveness of machine learning 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 malicious behaviors, the feature engineering process likely draws from a fraction of the relevant knowledge. Therefore, it is necessary and important to design an automated system to engineer features for discovering malicious behaviors and detecting attacks.

First, we describe a knowledge-based feature engineering technique for malware detection. It mines documents written in natural language (e.g. scientific literature), and represents and queries the knowledge about malware in a way that mirrors the human feature engineering process. We implement the idea 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 comparable performance to 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.

Second, we propose a data-driven feature engineering technique called ReasonSmith, which explains machine learning models by ranking features based on their global importance. Instead of interpreting how neural networks make decisions for one specific sample, ReasonSmith captures general importance in terms of the whole data set. In addition, ReasonSmith allows us to efficiently identify data biases and artifacts, by comparing feature rankings over time. We further summarize the common data biases and artifacts for malware detection problems at the level of API calls.

Third, we study malware detection from a global view, and explore automatic feature engineering problem in analyzing campaigns that include a series of actions. We implement a system ChainSmith to bridge large-scale field measurement and manual campaign report by extracting and categorizing IOCs (indicators of compromise) from security blogs. The semantic roles of IOCs allow us to link qualitative data (e.g. security blogs) to quantitative measurements, which brings new insights to malware campaigns. In particular, we study the effectiveness of different persuasion techniques used on enticing user to download the payloads. We find that

the campaign usually starts from social engineering and “missing codec” ruse is a common persuasion technique that generates the most suspicious downloads each day.

AUTOMATIC FEATURE ENGINEERING FOR DISCOVERING AND EXPLAINING MALICIOUS BEHAVIORS

by

Ziyun Zhu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Tudor Dumitraș, Chair
Professor Michel Cukier
Professor Michelle Mazurek
Professor Joseph JaJa
Professor Charalampos Papamanthou

ACKNOWLEDGMENT

I have been a PhD student at the University of Maryland since 2013. During these years, I meet different people and I believe it is impossible for me to finally get my PhD without them.

I would like to thank my advisor Tudor Dumitraș first. I started working with him in my second year. He always supports and encourages me to work on the project. When the result did not look promising, we brainstormed about the problems and solutions to find a correct direction. When I started writing papers, he taught me how to state the problem and phrase it. In addition, he also influences me a lot in my life. He is so positive and optimistic to life and is friendly to everyone. I spent wonderful and unforgettable five years in the group. Without him, I would never be determined to pursue my PhD.

Besides my advisor, I am grateful to other committee members, Joseph JaJa, Michelle Mazurek, Michel Cukier and Charalampos Papamanthou for their valuable comments and time. Dr. Mazurek and Dr. JaJa are on my committee for my proposal as well, and provides valuable feedback for the future directions.

I would like to thank my collaborators, Armin Sarabi, Chaowei Xiao, Dr. Mingyan Liu, Elissa M Redmiles, Sean Kross, Dhruv Kuchhal. Another project I did during my PhD study is measuring software patching. From the collaboration, I learned how to model software patching behavior using rigorous mathematic theory and how to conduct excellent user study. Besides, from the collaboration with Omer Yampel in ReasonSmith project, I learned what the attacks and defenses are in industrial settings.

I would also like to thank other students in our lab, Dr. BumJun Kwon, Octavian Suciu, Sanghyun Hong, Doowon Kim, Virinchi Srinivas, Yigitcan Kaya, and Erin Avllazagaj. It is a convention in our group to have a coffee break every morning, where we discuss and brainstorm new ideas for our research. I was inspired during the conversation with them, and learned a lot from their cultures as well.

My PhD journey would not have been possible without the support of my parents. I know it is difficult for them to let their only child go the opposite side of the earth. Thank you for encouraging me to pursue my dream.

Finally, I would like to thank my wife, Weiru. I was so lucky to meet her when I started my third year of my PhD. Because of her companion and support, I can overcome all obstacles and never feel depressed. We got married in January this year, and I think this dissertation will be the best gift for us.

Zhu, Ziyun

April, 2019
Maryland, USA

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

In recent years, machine learning is widely used in detecting malware because it learns common patterns from known malware automatically. The key of machine learning is feature engineering, which is a process of using domain knowledge to create features. The feature set allows researchers to represent samples in a machine-readable way, which facilitates the detection and analysis by using machine learning techniques. For example, the earliest Android malware families exhibited simple malicious behaviors [1] and could often be identified based on the observation that they requested the permissions essential to their operation [2]. To engineer such features, researchers reason about the properties that attacks are likely to have in common. This amounts to *generating hypotheses* about attack behavior. While such hypotheses can be tested using statistical techniques, they must be initially formulated by human researchers. The best sources to collect the feature hypotheses is scientific literature. Different from security blogs and industry reports, scientific literature is carefully reviewed by researchers and the conclusions are more representative. However, due to security arms race, malware has increasingly adopted more evasive techniques, and in response the security community has proposed a variety

of new features to detect these behaviors. For example, Google Scholar estimates that 32,300 papers have been published on Android malware and over 969,000 on intrusion detection.Moreover, the volume of scientific publications is growing at an exponential rate [3], as shown in Figure 1.1.

In addition, 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 [4], data scientists from New York University invested 30 person-months in identifying and incorporating informative features [5]. Because good machine learning models require a substantial manual effort, labor market estimates project a deficit of 190,000 data scientists by 2018 [6]. In the context of Android malware detection, the Drebin [7] 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.* So in this dissertation, we first ask the question (#1): **Can we automatically generate feature hypotheses from scientific literature?**

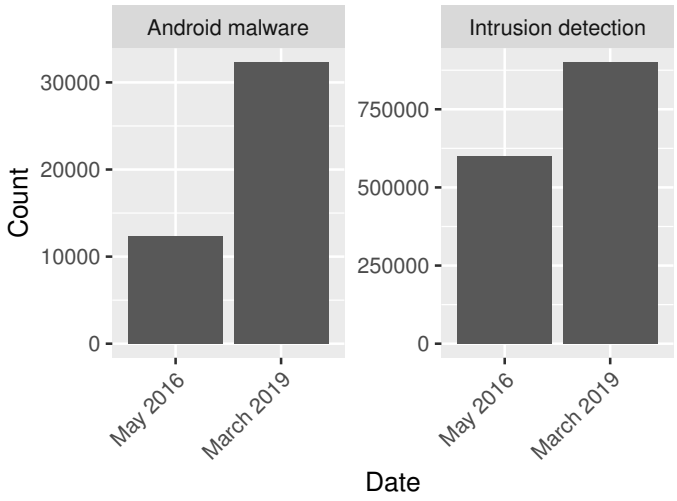


Figure 1.1: Paper estimation from Google Scholar.

Since researchers cannot discuss every aspects of security, the feature hypotheses from scientific literature may not be sufficient. An alternative solution to engineer features is directly using machine learning to learn useful features, because machine learning algorithm is able to test arbitrary feature hypotheses from data. However, due to the black-box nature of machine learning, we cannot examine which features are considered important for the model, and consequently, the model is likely to learn data biases and artifacts. For example, the synthetic data utilized in the DARPA IDS evaluation [8] was criticized for the lack of information on the validation of test data [9]—such as measures of similarity with the traffic traces or a rationale for concluding that similar behaviors should be expected when exposing the systems-under-test to real world data. Mahoney et al. shows that the intrusion can be detected simply from TTL field, because attack traffic and background traffic are generated from different physical machines and only a few values of TTL present in the data [10, 11]. Biases and artifacts from data sampling and generation

process would be reflected in the quality of the machine-learning techniques trained with this data. Unfortunately, in security it is generally difficult to obtain a clean training set for ML-based detectors.

In addition, in order to achieve good performance, deep neural network is used in detecting malware, e.g. multi-layer perceptron [12,13], CNN [14,15], RNN [16,17], etc.. The complexity and flexibility of neural network architecture makes the model powerful to fit the data and captures the common pattern of the attacks, but at the same time, makes the explanation much more difficult than traditional machine learning systems like logistic regression and decision tree. To address this problem, people propose different systems to explain machine learning models [18–20]. However, most of the methods are only able to explain single instances, and therefore it is still unknown if the explained sample and its explanation is *representative*. Given the black-box nature of machine learning and the presence of data artifacts and biases, we ask the question (#2): **Can we identify important features and data artifacts from neural network models?**

The features of malware detection are usually the observations from static or dynamic analysis, and the behaviors are monitored from a single machine. However, from a global view, malware relies on a chain of actions to be delivered to victims, which is called malware *campaigns*. It is difficult to systematically collect measurement that captures a complete chain of campaign actions, because campaigns involve the interactions between different hosts and are only active in a certain time. Therefore, researchers come up with the idea of threat intelligence to share the technical details of breaches and attacks. It makes everybody safer

by making it harder for attackers to reuse attack methods and artifacts, thus increasing their *work factor* [21]. Threat intelligence is a billion-dollar industry [22] and has introduced standards [23–25], and information-sharing platforms [26–29]. These standards define threats using *indicators of compromise* (IOCs). For example, to represent recent measurements of *malware delivery networks* [30–33] using these standards, we can identify IOCs such as file hashes of malware droppers and of their payloads, URLs and IP addresses of command-and-control (C&C) servers, or names of malware families and exploit kits. However, individual IOCs do not allow us to distinguish between components used for these various stages. Understanding the sequential actions of malicious campaigns requires careful manual analysis, and the results of this analysis are seldom encoded in a machine-readable format. Unfortunately, these results are often published, either in practitioner-oriented journals such as Virus Bulletin, or on the blogs of analysts or security companies, such as Webroot or Trend Micro. On the other hand, field-measurements capture the global trend of different attacks, where we can estimate the attack influence and victim volumes in the real world. By linking qualitative data like security blogs to field-measurements, we are able to learn both detailed behaviors from campaign operators and the global influence from the campaign. Therefore, in this dissertation, we ask the question (#3): **Can we identify and categorize campaign indicators (i.e. IOCs) from security reports?**

1.2 Challenges and contributions

1.2.1 Can we automatically generate feature hypotheses from scientific literature?

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 [1], 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 [34]). 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.

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 [35]. 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 [7, 19]; however, classifiers trained for malware detection typically use a large number of low level features [7], 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* [36]. 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".

Challenges. Natural language often contains ambiguities that cannot be resolved without a deep understanding of the subject under discussion. For example, the phrase "sends SMS message "798657" to multiple premium-rate numbers in Russia" [1] 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()`" [7] 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") [37]. To this end, specialized ontologies have been developed in other scientific domains, such as medicine [38]. Unfortunately, *security ontologies are in an incipient stage.* CAPEC [39], MAEC [40] and OpenIOC [23] 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 [41–43]. 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.*

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,[1] 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.

Contributions. In summary, we make the following contributions:

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

- 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`.

1.2.2 Can we identify important features and data artifacts from neural network models?

Machine learning especially deep learning is capable to learn complicated features automatically from data, which seems overcome the difficulties from manual feature engineering. However, due to the black-box nature of deep learning models, we have no clues about what features the system has learned. As a result, we cannot tell which features are important with respect to the training data and cannot distinguish if the model generalize security knowledge or merely memorize the training instances.

For malware detection, biases and artifacts are common in the data set, and can be originated from data generation process. For example, executing malware in sandbox environment is widely used to collect data from dynamic analysis. However, the virtual machine creates unique environment (e.g. network settings) that potentially changes the malware behaviors. The malware attempts to scan the network range can connect to internal gateway, which will never happen in the real world. The standard machine learning validation methods like cross-validation cannot solve this problem, because the artifacts will always exist in both training and testing sets. As a result, the neural networks can easily memorize the data artifacts, but such knowledge cannot be generalized to the area outside this environment. Therefore, it is important to design a tool that can pinpoint the place where data artifacts are likely to happen.

Research has been done to explain how the machine learning is making de-

cisions, and approximate the decision in the form of feature weight [18, 19], which helps human to verify if the decision is correct. However, such explanation can only interpret the model locally, and it is still unclear what the model uses features globally. Moreover, the problem cannot be solved by explaining more samples, since it is difficult state that the selected samples are representative to the whole data set. Neural networks are usually trained on millions of samples, however humans are unlikely to verify the explanation of even a small proportion of samples.

An alternative solution is to generate explanation at the feature level, which is widely used in traditional machine learning models. For example, logistic regression can be used in classification problems. The model parameter directly indicates the feature influence, which allows people to validate and adjust the model without retrieving any samples from training set. Therefore a feature-oriented solution is more efficient in examining the global model behaviors. Fortunately, input features for malware detection usually have their own semantic meaning especially for the observation at the level of API calls.

Challenges. The first challenge to reasoning deep neural network models based on input features is finding a metric that captures the general feature importance. In logistic regression model, since there is no hidden layer and no non-linear activation functions except for the output layer, system weight directly reflects the general contribution from each features. However, deep neural networks have multiple hidden layers, non-linear activation functions and different network architectures. These characteristics makes the model powerful to mine and memorize patterns from train-

ing data, but on the other hand, makes the relationship between input features and the output complicated. Therefore, it is almost impossible to derive a formula that explicitly calculates feature influence from model weights.

To overcome this, we model the global feature influence as a random variable that is dependent on the input feature, and the input feature is another random variable. If the feature influence is small with high probability, then we can conclude that the model does not consider this feature important. Therefore we can identify a feature set that are not used by the model.

The second challenge is how to define feature influence and how to estimate the distribution. To overcome this challenge, we define feature influence using the gradient of output with respect to the input. The gradient indicates the directions of input if we need to change output, and the magnitude of it reflects the feature importance locally. Similar idea has been applied in both model visualization [20] and adversarial machine learning [44, 45]. Instead of directly estimating the distribution, we characterizes the random variable using mean and covariance, which can be efficiently estimated empirically using maximum likelihood estimation.

Goals. The *first goal* is to design a system that is able to measure the influence of all features for a deep neural network model. The feature influence reflects the usage of each feature globally, such that by removing features with low influence, the negative affect should be minimum. In addition, the reasoning system should be compatible with different network architectures and cannot affect or change the model to be explained. Our *second goal* is to apply the proposed model to malware

detection problems and identify biases and artifacts that learned by the malware detector.

Contributions. In summary, we make following contributions.

- We propose a method called ReasonSmith to rank features based on their global importance. Compared to the traditional metrics like mutual information, ReasonSmith assigns better importance score to features for machine learning model. By removing less importance features, the machine learning model is still able to achieve high true positive rate with a low false positive rate.
- We propose a method to generate hypotheses on data biases and artifacts using ReasonSmith results and data sets in different time.
- We conduct an empirical study on both Windows and Android data sets, and find new features that are absent from prior knowledge and identify common data artifacts.

1.2.3 Can we identify and categorize campaign indicators from security reports?

A *malware delivery campaign* [30, 33, 46–48] involves multiple steps: baiting the user to perform a risky action, such as opening an email attachment or visiting an unknown site; exploiting an unpatched vulnerability on the user's computer; and installing a *dropper* [30], which then downloads additional malware. If one

step fails, the malware delivery fails; for example, an exploit kit [49] is useless unless the attacker can lure users to the landing page. The technical challenges for implementing each of these steps make it difficult for an individual actor to execute a campaign from end to end. Instead, malware creators rely on a thriving underground economy [31, 32, 49, 50], where specialized services are provided for a fee and one can quickly set up a campaign by summoning the hacking expertise of third parties.

In this ecosystem, we call the actors that provide malware-delivery services *suppliers* and the actors that utilize the service *customers*. Understanding the business relationships among suppliers and their customers can uncover important dependencies among these actors and may guide effective interventions [51]. For example, we distinguish between *tier 1 suppliers*, which are directly involved in user baiting and exploitation, and other suppliers that act as middlemen. Because only tier 1 suppliers bring victims to the delivery network, the effectiveness of their techniques is critical for the entire ecosystem.

However, neither threat intelligence nor measurement studies can provide a complete picture of malware delivery campaigns. Threat intelligence reports analyze the strategies employed in a campaign, but do not assess the effectiveness of these strategies in a systematic manner. For example, the reports may discuss a tier-1 supplier's baiting methods (e.g. spam, malvertising, compromised sites) but do not quantify the volume of malware downloads that these methods produce in the wild, as this would require comprehensive field data. Field measurements can record download events systematically, and provide various IOCs, but they may not be

able to indicate the role of them. For example, the measurement data does not usually indicate whether a dropper corresponds to a tier 1 supplier, nor does it indicate how the dropper was installed on the host. Moreover, measurement studies often focus on a single phase of the campaign—e.g., baiting [52,53], exploitation [54, 55], or installation [30, 33]—and do not shed light on the strategies of long-lasting campaigns. These insights require the ability to connect the threat intelligence with the field measurements.

Challenges. The key challenges to automating the insight generation process are the semantics of security threats in a machine-readable format, extracting them from the available threat intelligence, and developing analytics that combine the threat intelligence with measurement data. Existing standards for sharing threat intelligence like OpenIOC [23] and STIX [25] are incomplete and do not capture important concepts, such as the persuasion strategy employed by attackers to convince users into running malware or to lure them to an exploit kit. Worse, the IOC feeds currently available omit critical information, even when this information can be represented in the current standards. For example, STIX specifies the `kill_chain_phase` IOC attribute, which indicates the role of the indicator in the campaign; however, current feeds omit this attribute. Instead, this information is usually provided in the threat intelligence reports in natural language. However, prior work on mining security articles [56, 57] does not extract the roles of IOCs in campaigns, and does not correlate the information extracted from natural language with field measurement data.

To overcome these challenges, we aim to extract the threat and campaign semantics from intelligence reports written in colloquial English. In doing so, we further identify three technical challenges that are specific to security discourses. First, natural language processing (NLP) techniques usually rely on the context of complete sentences. However in security articles IOCs are often included in bulleted lists and tables, while the relevant relations are discussed in the text. Second, some indicators are presented in an obfuscated form. For example, to prevent the misclicking of a malicious site, malicious links are transformed to use "hxxp" instead of "http", and "[dot]" or (dot)" instead of ".", and authors use different names and delimiters for malware families. Third, the security arms race gives rise to a growing number of technical terms [56], while language models cannot usually handle previously unseen words.

Goals. Our *first goal* is to build a generic system that mines descriptions of security threats written in natural language, and extracts threat intelligence in a format that enables correlations and complements measurement data. Specifically, we aim to extract both IOCs *and* their roles in an attack, which allows us to augment the measurement data with the semantics of security threats observed. To this end, we develop new NLP techniques that address technical challenges specific to the security domain.

Our *second goal* is to apply our system to the concrete problem of understanding malware delivery campaigns and to gain new insights into this security threat. To this end, we propose a model of malware delivery campaigns (detailed in Sec-

tion 5.1.1), and we build a Web application for manually labeling articles according to this model. We also correlate the threat intelligence with a comprehensive data set of download events [30], observed on 5M hosts. With this combined data set, we ask the following research questions:

1. What is the relative effectiveness of various persuasion techniques in generating downloads on real-world hosts?

2. What roles do various attack groups play in the malware delivery ecosystem, and what are the business relationships among them?

3. How long do malware delivery campaigns remain active?

4. How long do their support infrastructures remain active?

In addition to our findings and their actionable implications, we expect that the process of answering these research questions will provide important lessons about the utility of threat intelligence—both for the tasks it was originally meant to address and for the novel application proposed in this paper. This is our *third* and final *goal*.

Contributions. In summary, we make the following contributions:

- We design ChainSmith, an IOC extraction system that collects indicators from security articles and classifies them into different campaign stages.

- We evaluate the effect of different persuasion techniques on the subsequent payload delivery in the wild, by connecting campaigns from blog posts and the real-world telemetry data.

- We report new findings about the underground business relationship and the characteristics of campaign infrastructures. This allows us to assess the utility of threat intelligence.

- We set up a Web application (`http://ioc-chainsmith.org`) to release the latest data from ChainSmith to further stimulate the research on threat intelligence.

1.3 Assumptions and limitations

Behaviors with complex operations. Since the features discussed in scientific literature are likely to be semantically meaningful, we focus on concrete features, which do not impose additional manual effort for the data collection. We are not able to extract behaviors that encode more complex operations, such as specific conditions or behavior sequences [58]. 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.

White-box assumption. For data-driven feature engineering approach, we must have complete access to the neural network models in order to calculate gradient for importance estimation. The primary usage of the reasoning tool is finding data

biases and artifacts during training. Consequently, we expect people who train the model to use this tool for examining and debugging the model, rather than people from third-party who do not have permissions to access internal structure of the model. In addition, since the proposed system identifies the key knowledge extracted from the model, which might be sensitive and crucial, the white-box assumption helps to protect data privacy and business secretes. 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. Although ReasonSmith system requires access to the original model, it is compatible to any architecture of neural networks and does not affect its training process. In this dissertation, we only test the case for multi-layer perceptron, but it can be applied to other models, which is discussed in Chapter 6.

Gaussian distributed interpretation. To model the global interpretation for neural network models, We use Gaussian distribution to estimate the local interpretation for a single instance. However, the Gaussian distribution is a strong assumption. The benefit of this assumption is computational complexity, and we are able to obtain a less precise but faster evaluation for the model and the data. Additionally, we discuss a more precise estimation method without the Gaussian assumption in Chapter 6.

Human examination for data artifacts. Since machine learning model cannot learn the semantic meaning of features, model cannot tell if the features come from

data artifacts. Distinguishing if a feature reflects the core malware behavior or the data artifact requires human intervention and the process is based on the understanding of malware behaviors. We aim to provide a method to efficiently prioritize the features for manual examination, but we cannot completely replace this step. Additionally, we do not aim to enumerate all biases and artifacts from data. Our system is able to identify the most important features learned by the model, which provides an opportunity to verify if there are any biases and artifacts learned. However, even though biases and artifacts are common in training data, model may not consider them important.

Noise from qualitative data. In this dissertation, we use scientific literature and industry reports to generate feature hypotheses and use security blogs to extract concrete indicators for malware campaigns. However, we are unable to assess the quality of the information discussed in these articles. Although it is likely that these sources contains fake information, we use the information to generate hypotheses and further test the hypotheses using measurement data. For example, the features engineered from scientific literature are further tested on malware data set using machine learning algorithm, and the malware campaign indicators from security blogs can also be found from measurement data. Therefore, quantitative sources provide external validation for the information extracted from qualitative sources.

Performance of malware detector. We aim to engineer informative features for detecting malware in general from both qualitative sources (i.e. scientific literature and security blogs) and quantitative sources (i.e. malware behavior measurements),

so we do not aim to outperform existing malware detection systems on a specific data set in terms of precision and recall. In addition, since it is unlikely to obtain a precise time stamp for all the information collected from qualitative sources, it is difficult to compare two feature set (i.e. automatic feature set from FeatureSmith and manual engineered set) with exactly the same time span. However, our system is able to engineer features automatically and can efficiently update the security knowledge by adding new articles and get rid of the intensive manual effort to engineer features.

1.4 Overview of natural language processing

In this chapter, we briefly introduce the state-of-the-art natural language processing techniques, which will be used in the automatic feature engineering systems.

1.4.1 Syntactic parsing

Syntactic parsing performs tokenization, part-of-speech tagging and dependency parsing, which processes the raw string to a tree structure that indicates the word dependency. This stage is equivalent to the lexical analysis and syntactic analysis in a compiler. Dependency parsing represents sentences as a directed graph to express the relationship between words. The parser identifies the grammatical relationship between words and labels each relationship with a type. For example, Figure 1.2 shows one example of dependency tree, where the arc label is the dependency type [59]. From the typed dependencies, we know that the MD5 "`4462c5b3556c5cab5d90955b3faa19a8`" is the object of "drop", while the subject

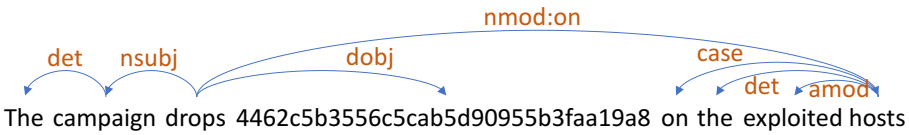


Figure 1.2: Example of dependency parsing.

of "drop" is "campaign". Therefore, we learn that the given hash depends on "drop" instead of "on", even though they are equally close to the hash. Syntactic parsing is widely used in NLP applications and the state-of-the-art system has a reasonable experimental performance.

Most of the NLP tools perform syntactic parsing. In this work, we use Python NLTK [60] for sentence and word tokenization and Stanford CoreNLP [61] for dependency parsing.

1.4.2 Semantic parsing

Since syntactic parsing cannot tell the semantic similarity among words, to make the system more generic, we have to understand the meaning of words. The state-of-the-art technique for semantic parsing is word embedding, which represents words, with close semantic meaning, in a close position in the vector space. The most popular embedding method is word2vec [62, 63], which trains word vectors by maximizing the probability of a word given its adjacent words. For example, in Figure 1.2, "campaign" and the MD5 are considered as the *context* of "drop", and the system should maximize the probability of "drop" given its context words. Instead of using adjacent words as the context, dependency-based word embedding utilizes word dependency to train word vector [64]. For example, for the MD5 in

Figure 1.2, the context is `drop:dobj` rather than {"drop", "on"}. The benefit of dependency-based `word2vec` is that it learns functional similarity rather than topical similarity. In addition, the embedding of dependency can be trained using related words, which indicates the common context pattern of a word.

1.4.3 Named entity recognition

Named entity recognition (NER) locates and classifies named entities into pre-defined categories. There is no general technique that is applicable to all NER problems, because the entity categories or ontology is usually task-dependent. Most of the NLP research focuses on identifying the names of persons, locations and organizations. While in security, the categories may include URLs, IP addresses, file names, etc.. Although there is no standard method for NER problems, the solution usually includes: fixed dictionary, regular expression or machine learning based classifier. Since most security entities (e.g. URLs, and IP addresses) tend to follow fixed patterns, regular expression is necessary to select entity candidates and remove irrelevant words. Table 1.1 lists the rules and additional constraints of recognizing the most common entity types in security. We should also take into account certain writing practices common to security articles that do not follow general patterns. For example, to prevent the mis-clicking of malicious site, some bloggers use "hxxp" instead of "http", and "[dot]" or (dot)" instead of ".". Additionally, authors might use different delimiters for malware family names, e.g. replacing underscores with semicolons, so we accept the delimiter to be any one of underscore, slash, period,

Table 1.1: Rules of named entity recognition.

Type	**Rules**
URL	Identified top level domain must be found [65].
IPv4	Contains 4 digits (<256) and the address is not reserved [66].
hash	A hexadecimal string of length 32, 40 or 64.
family	Starts with malware types, and contains common delimiter [67].
EK	Either in defined dictionary [68] or ends with EK or exploit kit.
vuln.	CVE-[0-9]{4}-[0-9]{4,5}

colon, and semicolon.

Because security narratives often include multi-word expressions, we cannot simply analyze individual words. For example, *Black Hole* is the name of an exploit kit; in this context, the words *Black* and *Hole* are meaningless when considered separately. Mikolov et al. propose a simple unsupervised approach to identify multi-word expressions [63]. The intuition behind this method is that the joint probability of words is much higher than the product of the probability of individual words for multi-word expression. In the example of "Black Hole", since the word "Black" is likely to appear together with "Hole", then we identify "Black Hole" as a multi-word expression.

1.5 Structure

The paper is organized as follows. In Chapter 2, we review the related work. In Chapter 3, we introduce a knowledge-based feature engineering system that mines features for malware detection from natural-language sources. In Chapter 4, we introduce a data-driven feature engineering approach that ranks features based on their global importance. In Chapter 5, we study automatic feature engineering problem for malware delivery campaign.

Chapter 2: Related Work

2.1 Knowledge-based feature engineering

Research on mining scientific literature dates back to Swanson [69], 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. [70] 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 [41] and latent semantic indexing (LSI) [42], 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 [43]. 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 [36] 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 [71]. 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.

Few references in security utilize natural language processing in system design. Neuhaus et al. analyze the trend of vulnerability by applying LDA to vulnerability description [72]. Pandita et al. identify Android permissions that are implicitly stated in the app description by using a dependency parser and first order logic [73]. Zhu et al. propose a framework to represent the knowledge available in security literature and generate features for detecting malware [56]. Liao and Yuan et al. propose a system to automatically extract OpenIOC items from blog posts [57]. Sun et al. design a system to generate human-friendly report for the results from Cuckoo sandbox [74]. Panwar designs a framework to generate IOCs in STIX format from Cuckoo sandbox results [75]. Zhu et al. propose a system to identify and categorize IOCs from blog posts, and show a use case of combining qualitative data to quantitative measurements [76]. A concurrent work by Husari et al. convert the Symantec malware report to STIX format by utilizing a pre-defined ontology [77].

In terms of the NLP techniques, we use word embedding instead of manually defined rules to learn the semantics of sentences, which is more general and applicable to a broader area. In addition, most of the work only focus on how to apply NLP to security but do not show security implications behind it.

2.2 Data-driven feature engineering

Since it is difficult to engineer a good feature set for malware detection, people tend to rely on the model to engineer features automatically from the data. However, due to the black-box nature of deep neural networks, researchers have limited understanding about how the model is making decisions even if it achieves perfect performance in testing. The absence of explanability makes the model vulnerable to potential evasion and poisoning attacks. Therefore, a lot of work have been done in recent years to interpret how models make decisions. The difficulty of generating explanation from model depends on the model complexity. For the traditional machine learning models, like logistic regression, SVM, decision tree, the feature importance can be derived either through rigorous statistical inference or from intuition. For example, whether the parameter from regression model is non-zero can be tested using t-test [78], which tells us if corresponding features are not significant.

In recent years, deep neural networks are widely used in malware detection, because deep neural network is more flexible and can better fit the data. However, the complexity of deep neural networks, on the other hand, makes it difficult to interpret the detection results. Most of work on DNN explanation focus on iden-

tifying the most important features used by model for a specific sample. Ribeiro et al. propose LIME to explain the model locally by training a separate explanation regression model [19]. A new data set is generated around the sample to be explained using the trained model, which is then used to train a logistic regression model. The feature importance can be derived directly from the weights of logistic regression model. Guo et al. extend the idea of LIME and propose a system LEMNA specific to security applications [18]. They add fused lasso to handle data dependency from binary reverse engineering and mixed regression model to handle non-linearity. Both solutions assume that the model is a black-box and we can only infer the model from input-output relationships.

Without black-box solution, others try to obtain the explanation from gradient, which is also the basic of ReasonSmith technique. The key idea is that the gradient of the output with respect to the input indicates the direction of input in order to change output, which is an indicator of feature importance by itself. Simonyan et al. first proposed the gradient solution to explain CNN image classifiers [20]. They define the saliency map as the absolute value of output gradient. Gan et al. proposed a spatial-temporal saliency map to event detection in video [79]. The gradient-guided techniques are widely adopted in adversarial machine learning to efficiently create evasion samples [44, 45]. Instead of using gradient, Zhou et al. proposed an class-specific saliency map that is derived from global average pooling to explain object detectors [80].

Most of model interpretation work focuses on explaining a single instance. However, explaining a few samples does not provide a strong evidence of the relia-

bility of the model, because the sample may not be representative. In general, it is hard to evaluate the general misbehaviors from the model. People have developed several techniques to evaluate if the model generalizes the knowledge from data or simply memorize it. Zhang et al. randomly change the labels and test the performance drop of the model to see if the model actually memorizes the fake data. Morcos et al. evaluate the robustness of model by testing the model reliance on single directions [81]. They conclude that if the model is less reliance on single directions, then the model is more likely to generalize knowledge from data. However, both techniques can only be used to compare models and fail to develop a criterion to determine if a model only memorizes the data. Pendlebury et al. proposed a result-oriented strategy and developed some metrics to evaluate model and data bias [82]. The key idea is that the performance of the model that is built on artifacts will degrade in the future. However, this strategy requires the data in a long time span. Our technique is feature-oriented, which models the feature influence as a random variable. Different from image classifier where the individual features (pixels) are not meaningful, malware detector usually takes observations from static and dynamic analysis as features, and each features have their own semantic meanings.

2.3 Malware detection

Android malware detection has been studied several years, Zhou et al. conducted the first systematic analysis of Android malware behaviors, from the initial infection to the malicious functionality [1]. As these behaviors often require specific

Android permissions, Felt et al. [83] and then Au et al. [84] 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 [2,85] to machine learning techniques [7,86,87]. 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 [2]. Zhang et al. construct API dependency graphs for each app, and identify the malware by detecting anomalies on these graphs [85].

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 [86]. Aafer et al. used k-nearest neighbors and extracted Android API calls as features [87]. 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 [7]. In [12], Grosse et al. use multi-layer perceptron to detect Android malware, and apply it to the same data set as Drebin to study adversarial examples. In [16], Xu et al. use both multi-layer perceptron and long short-term memory (LSTM) to detect Android malware, where the former network is used for fast filtering and the latter network is to generate more accurate result.

Different from Android system where API calls and permissions are semanti-

cally meaningful by themselves, Windows API methods are only meaningful when considering the arguments. For example, system information is stored in registry keys, and any attempts to edit the system configuration can be reflected from registry keys and values. However, the API method to edit registry key (i.e. `RegSetValueExA` or `RegSetValueExW`) only indicates the behavior, and cannot tell if the program tries to edit sensitive configurations. To solve the missing semantic problem, analysts manually create heuristics to label malware based on API calls and their corresponding arguments. For example, Cuckoo sandbox is an open-source tool to execute and analyze malware [88]. Cuckoo has a signature class that is manually created to label malware. The signatures can be either general to all malware (for example `...\Windows\CurrentVersion\Run\...` represents autostart), or specific to one malware family (e.g. `wmpsl64.exe` is the payload name of Bublik trojan).

Signatures label and detect malware in a deterministic way, and its coverage is generally smaller compared to machine learning systems. For Windows malware detection problem, features are usually the event sequence from dynamic analysis, where the event is the combination of API call and its arguments. Schultz et al. use loaded DLLs, strings and byte sequences as features to detect Windows malware, and compare the performance of different machine learning models including Ripper (rule-based model) and Naive Bayesian [89]. To capture temporal information from execution traces, Kolter et al. use n-gram instead of single API calls as features, and further compared different detection models including Naive Bayesian, decision tree, support vector machine (SVM), and boosted decision trees. [90]. Association rules are also used in detecting malware. In [91], Ye et al. use API sequence as

feature and Objective-Oriented Association mining as the model. In [92], Ravi et al. use 4-gram from Windows API call sequence as features. In addition, researchers develop different tools to learn the meaning of arguments. Pascanu et al. propose a 2-stage architecture, where the former network is used to extract feature from arguments (e.g. file names) and the latter network is used for detection [93]. Authors further evaluate different architecture and conclude that Echo state networks (ESNs) and logistic regression are the optimal architecture for extracting features from argument and final detection decision respectively For the same problem, Athiwaratkun evaluate other architecture options including character-level CNN, long short-term memory (LSTM) and gated recurrent unit (GRU) [17]. Results show that the LSTM with temporal max pooling and logistic regression achieves optimal performance. Similarly, the system from Tobiyama et al. consists of two stages, but they use RNN to extract features and CNN for detection. Instead of using features from dynamic analysis, Saxe et al. propose using features from PE meta for malware detection [94].

2.4 Malware campaign and threat intelligence

Most prior measurement studies focused on only one stage of malware delivery campaigns. Prior work has identified social media advertising [95–97], spam email [98, 99], compromised site [53], or SEO poisoning [52] as techniques used for delivering payloads. However, most of the work did not quantify the volume of malware downloads resulting from these techniques. Nelms et al. [97] measure the

influence of different persuasion techniques on malware downloads quantitatively by parsing HTTP response and manually annotating the content. However, this method is not scalable from both manual annotation and network monitoring. In addition, this method fails if the package is encrypted using HTTPS. The advantage of our technique is that there are less restrictions on measurement data, because the campaign information is collected from independent sources.

Eshete et al. propose a system to detect if the URL is the landing page of exploit kit based on the common techniques and behaviors of exploit kit. [54]. Moreover, Eshete et al. design a exploit kit infiltration toolkit to detect the exploited vulnerabilities [55].

The malicious downloading behavior was analyzed by Kwon et al. in [30,33]. In [30], a classifier is designed to detect malware using downloader-payload relationships. In [33], the authors propose a system to detect coordinated behaviors among downloaders on multiple hosts. PUP delivery services were studied in [31,100]. The former identifies the PUP publisher and the delivery service from the certificate and structure of download relationships, while the latter closely investigates a few prevalent PPI services and monitors the subsequent delivery behavior.

Zhang et al. propose a system to detect malicious servers and group them into campaigns using HTTP traces from ISPs [101]. Plohmann et al. comprehensively analyze domain generating algorithms and pre-compute the DGA domains in [102].

IOCs (indicators of compromise) are commonly used to model malware campaigns. The IOCs specified in OpenIOC, STIX and CybOX standards [23–25] define the role of an indicator in a campaign, for instance that an IP address corresponds

to a C&C server. Researchers at Lockheed-Martin corporation proposed a cyber kill chain model to describe the action sequence from attacker to launch malware campaign [103]. STIX includes the cyber kill chain definition, however this is seldom used in existing IOC feeds. For example, Hailataxii [29] is a repository for threat intelligence data formatted according to the STIX standard. Although more than 700,000 indicators are provided by Hailataxii, none of them include the kill chain phase, which can only be inferred manually from natural language description of indicator.

In [57], Liao and Yuan et al. propose a system to automatically collect IOC from blog posts in natural language. They model the problem as graph similarity problem, and identify the IOC item if it has a similar graph structure as the training set. However, the identified IOCs do not preserve their roles in a malicious campaign, which makes it difficult to analyze the characteristics of campaign in different stages and to correlate with field measurements. To address this problem, Zhu et al. and Husari et al. propose two different approaches to further categorize and bring more semantics to IOCs [76, 77].

Chapter 3: Knowledge-based feature engineering

In this chapter, we propose a system that automatically engineers features from scientific literature for malware detection. In Chapter 3.1, we introduce semantic network to represent security knowledge and engineer features from it. Then we show how the semantic network is constructed for Android malware detection in Chapter 3.2. We evaluate the performance of knowledge-based feature engineering system in Chapter 3.3.

3.1 Semantic network

3.1.1 Definition

We model the concepts discussed in natural language 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 between the 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: threats V_t, behaviors V_b, and features V_f. We define two types of edges: (1) links between threats and

behaviors $E_{tb} = \{\{u, v\}, \forall u \in V_t, \forall v \in V_b\}$; and (2) links between behaviors and features $E_{fb} = \{\{u, v\}, \forall u \in V_b, \forall v \in V_f\}$. 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.

3.1.2 Behavior extraction

We extract suspicious behaviors discussed in the security literature in two steps: (1) we identify phrases that may correspond to suspicious behaviors, and (2) we apply filtering and weighting techniques to find the most relevant ones.

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.

Behaviors are constructed from certain typed dependency and part-of-speech as listed on Table 3.1. [1] 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 can apply WordNet [105] to lemmatize words based on their part of speech. From typed dependencies, we decompose a complex sentence into several simple relations.

Table 3.2 shows one example of behavior extraction.

Filtering and weighting.

To determine which behaviors are most relevant to attacks, we assign weights that capture how semantically close these behaviors are to the malicious functionality. We determine the weights in two steps:

1. Word weighting: assign weights for both verbs and noun phrases based on how close they are to the problem.

2. Behavior weighting: assign weights to each behavior based on the weight of subject, verb and object.

[1] We apply both *collapsed* and *ccprocessed* options in Standard typed dependency parser [104]. The former is to simplify the relationship with fewer prepositions and the latter is to propagate the dependency if a conjunction is found.

Table 3.1: 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>

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 should select a topic word that best describes the problem. For example, we can select the word *Android* to engineer features for Android malware detection. We collect all the noun phrases from subject and object and verbs

Table 3.2: 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." [2]
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

in behaviors. Then, we evaluate the importance of each word[2] by computing the *mutual information* of the word and the topic word; we do this for both the verbs and noun phrases from the 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 frequencies from 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 tends to find general words but ignores the less frequent words. To solve this problem, we scale the mutual information by the entropy of the word. The weight of word $S(w)$

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

is calculated using Equation (3.1), where $H(w)$ is the entropy of word w, $I(w; w_T)$ is the mutual information between word w and topic word w_T. This metric captures the *fraction* of uncertainty of word w given the topic word, and the value ranges from 0 to 1.

$$S(w) = \frac{I(w; w_T)}{H(w)} = 1 - \frac{H(w|w_T)}{H(w)} \tag{3.1}$$

In the last step, we assign an initial weight for each behavior $W(b)$ based on the weights of verb and noun phrases The behavior weight $W(b)$ is the product of the verb weight $W(v)$ and the maximum noun phrase weight, as shown in Equation (3.2),

$$W(b) = W(v) * \max_{n \in \{s,o\}} W(n) \tag{3.2}$$

where behavior b consists of subject s, verb v, and object o.

3.1.3 Feature generation

We utilize the semantic network to rank the features and to determine which ones are most relevant for detecting attacks. Let T, B, F be three random variable for threat, behavior and feature respectively. We compute the probability of feature π_F from the probability of malware π_M using Equation (3.3):

$$\pi_F = \pi_T * P_{B|T} * P_{F|B} \tag{3.3}$$

The transition probability $P_{B|T}$ and $P_{F|B}$ is estimated using the edge weight of semantic network E and weight of behaviors W by Equation (3.4):

$$P_{B|T}(b|t) = \frac{E(b,t)W(b)}{\sum_b E(b,t)W(b)}$$
$$P_{F|B}(f|b) = \frac{E(f,b)}{\sum_f E(f,b)} \tag{3.4}$$

The intuition behind this equation is that the most informative features correspond to some malicious behaviors that are shared by multiple attacks, 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 our problem.

3.2 Semantic network construction

We apply our technique to engineer features for Android malware detection. Figure 3.1 shows the pipeline of feature engineering system.

3.2.1 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 permissions, API calls, etc.), for determining which features can be tested experimentally, and malware samples, for validating the feature generation process. Table 3.3 summarizes these data sets. In this chapter, we discuss the data collection process and the preprocessing we apply to each type of data.

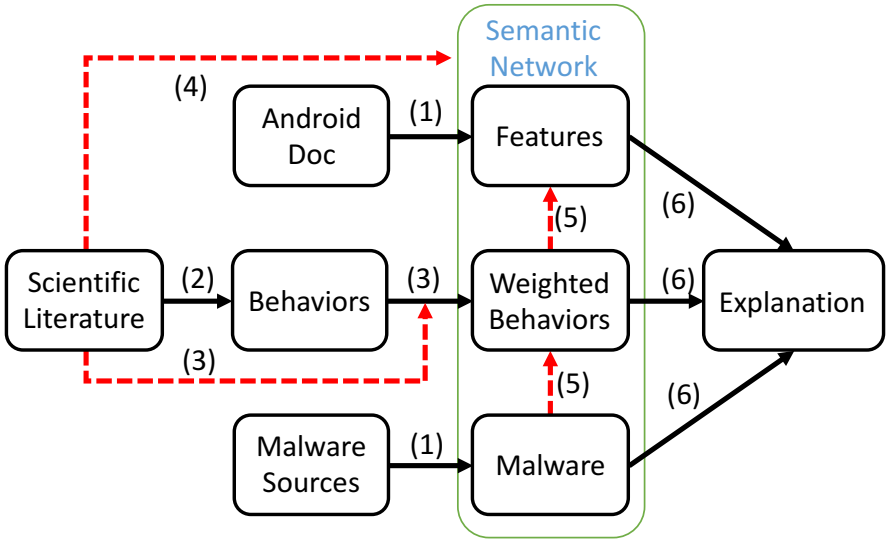


Figure 3.1: General architecture for automatic feature engineering: (1) data collection; (2) behavior extraction from scientific papers; (3) behavior filtering and weighting; (4) semantic network; (5) feature generation; (6) explanation generation. Black lines indicate the data flow and red dashed lines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

3.2.2 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)[3], 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.[4] Most of articles returned from Google Scholar are scientific papers that require peer reviews before publication, which means that we can ensure the quality of the corpus. Other data sources (e.g. analyst blogs) could be informative, but the quality is not guaranteed. Google Scholar results include industry reports as well. We believe that the industry reports are of high quality as well because Google Scholar ranks articles based on citation [106] and these articles have high ranks from it. In total, our corpus includes 1,068 documents. In addition, we also collect the publication time for the articles using the conference year or the year

[3]Including workshop papers.

[4]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.

Table 3.3: 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	

provided by Google Scholar, and successfully collect the time for 916 articles.

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.

3.2.3 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 [107]. 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.

3.2.4 Malware families

We collect the malware family names from both the Drebin dataset [7] and from a list of malware families [108] caught by the Mobile-Sandbox analysis platform [109]. 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 [7], 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 [110], 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` [111] to extract all the Android API

calls. This allows us to expand the feature vectors and test the features omitted by Drebin.

We obtain the expanded feature vectors for 5,552 malware samples and 5,553 benign apps.[5] 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 Chapter 3.3; the feature generation utilizes the malware names and the document corpus.

3.2.5 Network construction

After we extract the key components (i.e. malware, behaviors and features), we construct the semantic network according to the definition in Chapter 3.1. Figure 3.2 shows part of our semantic network.

This step produces 339,651 unique behaviors. Then we choose the word *Android* as the topic word to evaluate the importance of the behaviors. We remove the behaviors that are unlikely relevant to Android, and obtain 82,035 behaviors. In total, we have 47,186 noun phrases and 1,682 verbs. Table 3.4 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.

Table 3.5 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

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

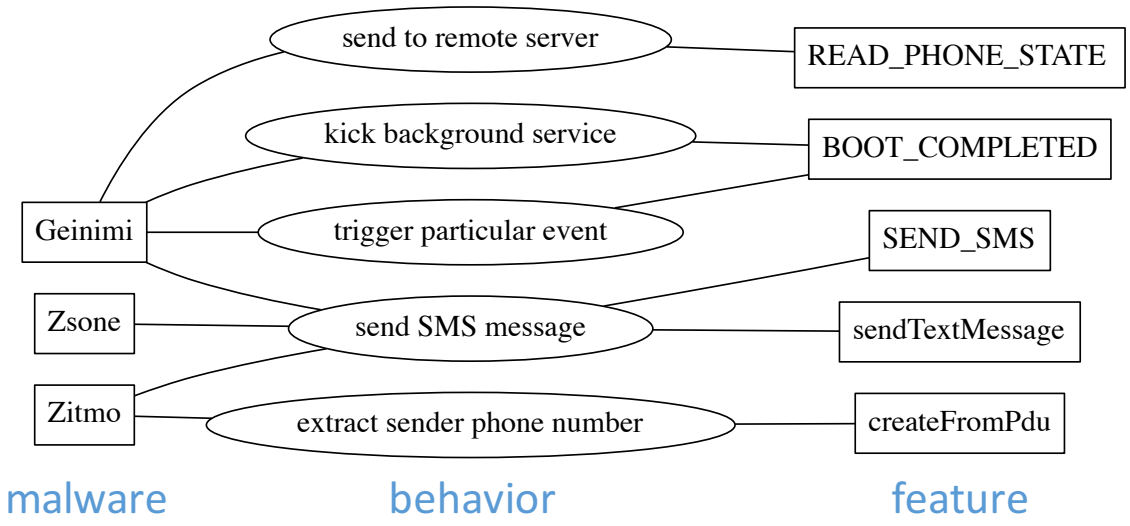


Figure 3.2: Excerpt from our semantic network. The nodes correspond to malware families, malware behaviors, and concrete features. Unlike in an ontology, the categories of malware behavior are not predetermined.

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*".

Table 3.4: Top 5 behaviors related to Android.

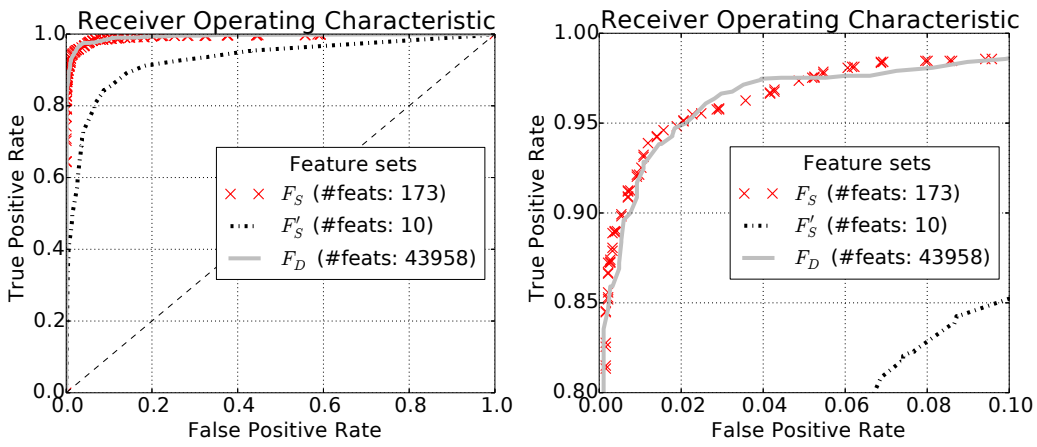
rank	behavior
1	Over-privileged apps overstep permission
2	manufacturer customize smartphone OS
3	malware author download Android's source code
4	download from official Android Market
5	download from app store

Table 3.5: 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

3.3 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 Chapter 3.2. We train random forest classifiers [112] with (i) the features generated by FeatureSmith and (ii) the manually engineered features from Drebin [7]. We compare the performance of these classi-



(a) Original

(b) Zoom-in

Figure 3.3: 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).

fiers in Chapter 3.3.1. In Chapter 3.3.2, 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 Chapter 3.3.3.

3.3.1 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 [112].

Figures 3.3a and 3.3b 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.[6] 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

[6]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 [7].

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 [7]. 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.

3.3.2 Tapping into hidden knowledge

We evaluate the contribution of individual features to the classifier's performance by using the mutual information metric [34]. Intuitively, mutual information quantifies the loss of uncertainty for malware detection when the app has the given feature. Table 3.6 lists the 5 features with the highest mutual information. When present together, these features indicate an app that triggers some activity right

Table 3.6: 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

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 3.6 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 3.4 shows the cumulative mutual information for the top 150 features in the FeatureSmith and TF rankings. Be-

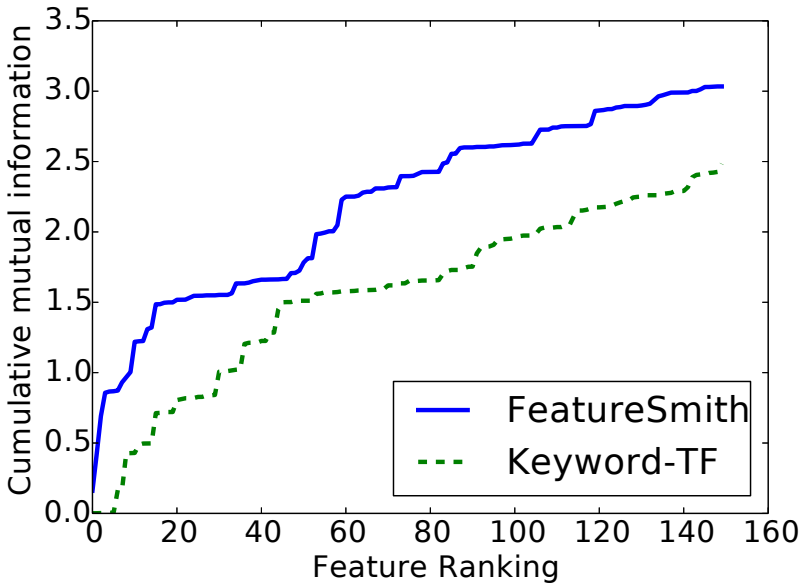


Figure 3.4: Cumulative mutual information of top-ranked features.

cause 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 [113] 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).

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 [85], 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 [114], and `isMusicActive`, which can be used to infer the user's location [115]. 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 [116] and as a method that leaks private data [114]. `getNetworkOperatorName` is another method that potentially leaks private data [117]. 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

Table 3.7: An example of feature explanation.

	`getSimOperatorName`
Behavior	return privacy-sensitive information leak privacy-sensitive return value leak to remote web server ...
Reference	[114]: 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.

features are related to *"steal sensitive information"* behavior, which will never be identified by parsing developer documents only. Table 3.7 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 [35].

3.3.3 Knowledge evolution over time

Our results from the previous chapter 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 chapter, we characterize the growth of FeatureSmith's semantic network, which is a representation of the existing knowledge about Android malware. Intuitively, as we

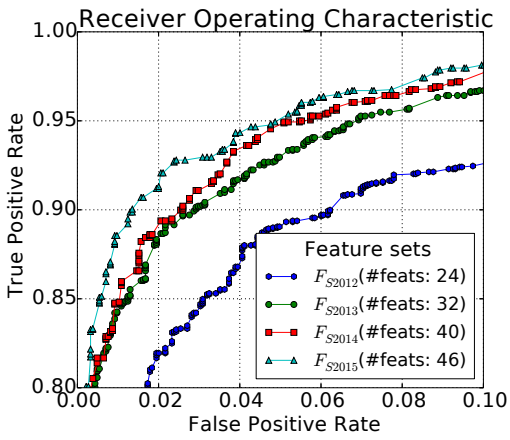


Figure 3.5: ROC curve of malware detection for classifiers with feature sets from different years

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 investigate how this evolution affects our ability to engineer effective features for malware detection. However, because we cannot identify the publish year for some articles, we cannot reliably tell the earliest year that a feature is discovered by FeatureSmith. For those features that we are able to identify the discovered year, we train 4 malware detectors using the same algorithm with the features discovered before 2012, 2013, 2014 and 2015.

Figure 3.5 shows the ROC curve of the classifier trained using features discovered in different years. The figure shows 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 1% false positive rate, the true positive rate increases from 73.1% in 2012 to 89.2% in

2015.[7] 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. In addition, the performance improvement diminishes after 2013. This suggests that the most important and salient features are likely to be discovered first. By using these features, we can capture the core behaviors from malware. The follow-up work is able to tell the new behaviors or missing knowledge from prior work. And the discovered features are complementary to the original feature set and are useful in improving the detection performance.

[7] 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%.

Chapter 4: Data-driven feature engineering

In this chapter, we introduce a data-driven feature engineering method called ReasonSmith, which ranks features based on their global importance. In Chapter 4.1, we introduce the design of ReasonSmith. Then we evaluate ReasonSmith for both Android and Windows malware detection. In Chapter 4.2, we demonstrate the data set and data preprocessing used in the experiment. In Chapter 4.3, we compare ReasonSmith with traditional metric mutual information, and further identify data biases and artifacts.

4.1 Method

4.1.1 Feature importance criterion

To capture feature global importance, we would like to select a subset of features (k out of n) that maximize the performance in testing. Let TPR_F be the true positive rate of a malware detector in testing using feature set F. Equation (4.1) shows the criterion of feature selection based on the global importance, where $|F|$ is the cardinality of feature set F. We focus on the true positive rate when false positive rate is 1%, which is also used in other work to evaluate malware detector

performance [7, 56, 89].

$$\begin{aligned} \max_{F} \quad & TPR_F \\ \text{s.t.} \quad & |F| = k \end{aligned} \tag{4.1}$$

Intuitively, if we only use a subset of features in testing, the malware detector should perform worse than the original system, because the model uses additional features (i.e. the removed features) for training. If the performance of malware detector does not drop after feature removal, then the model does not learn security knowledge from the removed features. However, if the performance drops dramatically, then the removed features should be important to characterize the security knowledge learned by the model.

However, we cannot find an optimal solution for this problem in practice because we need to test the performance of all $\binom{n}{k}$ feature subsets. As an alternative solution, we can estimate the global importance for individual features, and further select the features based on the importance. Therefore the original problem changes to finding a ranking method such that the model can achieve high true positive rate in testing by using the top k features.

4.1.2 ReasonSmith

Different from prior work on explaining instances [18–20], we focus on the overall performance of the features on the model, which extends the idea of generating local interpretation and further derives a global interpretation. We call our ranking method ReasonSmith. It first estimates feature importance for single instances, and

then evaluates the global importance based on the local interpretations.

Local interpretation. Let $\boldsymbol{x}$ be the feature vector and $\boldsymbol{w}$ be the local interpretation for the corresponding feature. A greater absolute value means that the corresponding feature is more important. Feature influence $\boldsymbol{w}$ captures the local influence of input $\boldsymbol{x}$ for the machine learning model $y = f(\boldsymbol{x})$. There are several techniques from prior work to obtain a local interpretation of a specific input. But in our case, we need to collect the feature influence for individual instances in a large scale. So we have a stronger constraint on computational complexity. Some techniques like LIME [19] and LEMNA [18] do not satisfy our requirements, since they need to train a separate model for each instance to be explained. To this end, we choose gradient-based solution [20] to learn the feature importance. For each instance to be explained, we can obtain the result by passing the input to the model only once, and consequently it is possible for us to collect feature influence in a large scale. In this dissertation, we define local interpretation as the gradient of the output with respect to the input (as shown in Equation (4.2)). For malware detection problem, the output is usually the probability that a sample is malware. By definition, if the gradient is close to 0, then the corresponding feature is less important to make this specific decision.

$$\boldsymbol{w} = \frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x}) \tag{4.2}$$

Global interpretation. From a global view, the interpretation is a random variable. We use Gaussian mixture model to estimate the distribution of feature influence $\boldsymbol{w}$. Let us assume that the machine learning model generalizes security

knowledge from data and the knowledge can be summarized with K rules and each local explanation comes from one of its rules.

Let us define feature importance score S as the metric of feature global importance, which is defined as the probability that $\boldsymbol{w}$ is non-zero. For a specific feature j, the probability that its value is small can be written in Equation (4.3), where $\boldsymbol{R}_k$ is the kth Gaussian distribution. For each Gaussian distribution, parameters can be estimated using EM algorithm.

$$S_j = 1 - P(w_j = 0) = 1 - \sum_k P(w_j = 0 \mid j \in \boldsymbol{R}_k) P(j \in \boldsymbol{R}_k) \tag{4.3}$$

In a special case, let us assume that there is only one rule learned by the model. Then we can estimate mean and covariance using Equation (4.4) and (4.5). Because only one primary rule exists in this case, we can save much computation from EM algorithm, which allows us to obtain a less precise but faster result. In this paper, we use this special case to estimate the underlying feature influence.

$$\boldsymbol{\mu_w} = \frac{1}{N} \sum_{i=0}^{N} \boldsymbol{w^i} \tag{4.4}$$

$$\boldsymbol{\Sigma} = \frac{1}{N-1} \sum_{i=0}^{N} (\boldsymbol{w^i} - \boldsymbol{\mu_w})(\boldsymbol{w^i} - \boldsymbol{\mu_w})^T \tag{4.5}$$

In addition, the malware detector takes binary features as input. We set the local interpretation to 0 if the corresponding feature is 0, because the model should not be influenced by absent events. Equation (4.6) is the formula for binary features in this paper, where $\odot$ is element-wise product.

$$\boldsymbol{w^i} = \frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x^i}) \odot \boldsymbol{x^i} \tag{4.6}$$

4.1.3 Data bias analysis

The bias evaluation process consists of three steps:

1. Split data into several groups based on time.
2. Train individual models on different data groups, and apply ReasonSmith to rank all the features by feature importance score $\boldsymbol{S}$.
3. Compare feature rankings in different data groups.

Data split. Prior work shows that the performance of malware detector generally degrades over time due to either concept drift or training biases [82]. It is difficult to evaluate the future performance drop using conventional evaluation method like cross validation. Therefore, we split the data based on time, which makes it possible to identify the changes of data and model over time.

Then we train individual malware detectors using the same model and calculate feature rankings from ReasonSmith for each data set.

Importance over time. We rank features by importance score $\boldsymbol{S}$, and the features with high scores are on the top of the list. We can obtain multiple rankings from different data groups for features. Then we can calculate the average ranking and ranking variance for a specific feature, which can be used to distinguish data artifacts and data biases.

A higher ranking mean suggests that the feature is consistently important across different groups. The features with high ranking mean are robust and useful

Table 4.1: Data set after preprocessing

		time span	usage	# of samples
Android	$\mathbb{S}_1$	07/01/2009 - 06/30/2012	train	21,635
	$\mathbb{S}_2$	07/01/2012 - 08/31/2012	test	7,070
	$\mathbb{S}_3$	09/01/2012 - 09/30/2012	test	18,873
	$\mathbb{S}_4$	10/01/2012 - 01/01/2015	test	4,491
Windows	$\mathbb{S}_5$	07/01/2018 - 07/31/2018	train	71,919
	$\mathbb{S}_6$	08/01/2018 - 08/20/2018	test	34,341

in different period of time, and are more likely to be indicative to the main functionalities of malware. By manually examining these features, analysts can generate hypotheses about the malware behaviors from the training data set. However, these features can also be data artifacts. Different from data biases, data artifacts might result from data generation mechanism and exist for a long period of time. Therefore the features from data artifacts are likely to have a high ranking mean as well.

A high ranking variance indicates that the importance of the corresponding feature changes a lot and the feature is only important in particular data groups. Because data biases result from data sampling process, features from data biases are likely to have high uncertainty values. By manually examining high-uncertainty features, analysts can learn the malware trend over time.

4.2 Data sets

4.2.1 Data collection

We collect malware samples for both Windows and Android systems from VirusTotal [110] and Drebin [7]. Table 4.1 summarizes our data sets. Drebin provides a data set of Android reports from static analysis for 129,013 apps. Drebin extracts 8 type of features: hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls and network addresses.

We collect Windows data set of 114,482 samples from VirusTotal by using version 2 of their API. When a user submits a file under 8 megabytes in size which has never been submitted before to VirusTotal, the sample is dynamically analyzed in a custom Cuckoo Sandbox and the report becomes available via the `/file/behaviour` API endpoint. In order to pull back all Windows samples which contain a sandbox report, we use the `behaviour` search modifier and pair it with common file formats used by malware. To gather our dataset, we pulled hashes that have an analysis report attached to them which contain one of the following types:

1. PE executables: `*.exe`, `*.dll`
2. Python scripts: `*.py`
3. Malicious documents: `*.rtf`, `*.doc(x)`, `*.xls(x)`

The dynamic analysis report is composed of two main parts; the sequence of

windows API calls performed by the submitted sample and any of its descendants, as well as all network traffic observed. Each observed API call contains all parameters passed to that API call. For example, `CreateProcessInternalW` contains the full file path of the launched process in the `lpApplicationName` parameter, and the command line used to launch it will be in the `lpCommandLine` parameter. Network traffic is made up of all HTTP, DNS, and IP traffic. HTTP requests contain all relevant fields, such as HTTP method, user agent, destination host, URI, and request body. DNS traffic contains the hostname requested and the IP it resolved it. IP traffic contains the source and destination IP and ports used, as well as the protocol.

4.2.2 Data processing

For Android data, we apply the same data processing as we did in Chapter 3.2.1. Therefore, we only introduce the feature extraction and data filtering for Windows data set in this chapter.

Feature extraction. Different from Android system where individual API calls and permissions are semantically meaningful, Windows API calls only define the actions, and are only meaningful with the arguments. Therefore, we define the features as a combination of action (i.e. API calls) and target (i.e. arguments). We further define several high-level actions based on API calls, which are listed in Table 4.2.

Target can be one of 6 types, including processes, files, domains, URIs, IP addresses and registry keys. Then we apply the following rules to normalize the

Table 4.2: Mapping between API calls and high-level actions.

Observed API Call	Action
`RegCreateKeyExW`	Created Key
`RegEnumKeyExW`	Enum Key
`RegEnumValueW`	Enum Value
`RegOpenKeyExA`	Open Key
`RegOpenKeyExW`	Open Key
`RegQueryValueExW`	Query Value
`RegSetValueExA`	Set Value
`RegSetValueExW`	Set Value
`CreateFileW`	File Created
`DeleteFileW`	File Deleted
`CopyFileExW`	File Copied
`MoveFileWithProgressW`	File Moved
`ReadFile`	File Read
`WriteFile`	File Written
`ShellExecuteEx`	Shell Command Executed
`CreateProcessInternal`	Process Lanunched
`CreateRemoteThrea`	Process Injected
`LoadLibrary`	Module Loaded
`CreateMutexW`	Mutex Creation

Table 4.3: Relationship between target types and high-level actions.

Target type	Actions
Processes	Launched, Injected
Files	Created, Deleted, Copied, Moved, Read, Written
Registry Key	Created Key, Enum Key, Enum Value, Open Key, Open Key, Query Value, Set Value,Set Value
Modules	Loaded
Shell Command	Executed
Addresses	Connected To, Resolves To
URLs	URI Of, HTTP Request To
Domain	DNS Query For, Domain Of

target string.

1. Normalize Windows file path by using default environment variable. For example, `C:\Program Files` is normalized to `%ProgramFiles%`.

2. Identify GUID, SID, and replace them by a fixed token `<GUID>` or `<SID>`.

3. Replace SHA256 process names by `<SELF>`, because VirusTotal renames submitted executables to the SHA256 hash of their contents.

Then we encode the high-level action and normalized argument string as binary features that are readable by neural networks.. Table 4.3 lists all types of features in Windows data set.

Sample filtering. Because Windows feature set is collected from dynamic analysis instead of static analysis, it is likely that we cannot observe the correct and complete execution traces. There we remove the samples if their total execution time is less than 2 seconds, or `drwtsn32` is called which is an indicators for crashes.

Creation time estimation. We approximate the sample creation time using the first submission time from VirusTotal report, which is an upper bound for the sample creation time. We split the samples into different data sets to by their creation time identify data biases. Table 3.3 gives basic statistics for different data sets.

Label Creation. In addition, the label of samples are determined from anti-virus labels. For Windows application, we label it as malicious if it has more than a 30% detection rate from VirusTotal, which is the same threshold used in prior work [30, 100]. While for Android application, we label it as malicious if there are more than 4 detections from AV vendors, which is also in line with prior work [82, 118]. If there are no AV detections, we label the application as benign. We drop the rest of applications if they do not satisfy either malicious or benign criterion.

4.2.3 Analyst features

We define analyst features as the features that are manually engineered by analysts and are used for detecting malware. For the Android data set, we select 4 Drebin feature categories as the analyst features: permissions, real permissions, restricted API calls and suspicious API calls. Because permissions are important in Android to protect sensitive data and crucial services, they are used as key features

in most of Android malware detectors. Similarly, restricted APIs are the API calls protected by the permissions, and can also be considered as analyst features. In Drebin, the authors manually developed heuristic rules to identify suspicious API calls. In total, we consider 373 features as semantic features (out of 11,731 features). For the other 4 feature types, the analysts simply use all the features that observed in the data set for example, a specific URL or a user-defined activity name. As a result, we do not consider them as analyst features.

Cuckoo sandbox provides over 400 manually crafted signatures to analyze execution traces and label suspicious behaviors. Although these signatures include many human-defined conditions, which are difficult to be directly applied to ML systems, the individual rules used in signatures can also be a good indicator of maliciousness. We extract all the regular expressions in the `Signature` class from Cuckoo repository [88], and then check if our features match any of the regular expressions. As a result, we identify 1271 compilable regular expressions, from which we further label 867 features as semantic features (out of 41,858 features).

4.3 ReasonSmith evaluation

First, we train 2 deep neural networks for Android and Windows malware detection. We use multi-layer perceptron as basic architecture which contains two hidden layers with 1024 and 32 nodes respectively. The detector takes binary features as input and returns the likelihood of maliciousness. We use ReLU as the activation function for all hidden nodes and sigmoid function in the output layer.

Table 4.4: Performance of Windows and Android malware detectors. Note that S_1 and S_5 are used as the training sets.

	testing set	TP(FP=0.01)	TP(FP=0.05)	TP(FP=0.1)
Android	$\mathbb{S}_1$	0.962	1.000	1.000
	$\mathbb{S}_2$	0.896	0.951	0.957
	$\mathbb{S}_3$	0.858	0.878	0.890
	$\mathbb{S}_4$	0.890	0.917	0.926
Windows	$\mathbb{S}_5$	0.996	1.000	1.000
	$\mathbb{S}_6$	0.948	0.956	0.992

Since the feature set contains some features that have low occurrence which might negatively affect the system performance, we remove those uncommon features during training. In this work, the feature occurrence threshold is 50 for both detectors. As a results, the size of input feature is 41,858 for the Windows detector and 11,731 for the Android detector. Table 4.4 shows the performance of the two malware detectors, and both models have high true positives and low false positives in testing. We use these two detectors as the example in this dissertation, and apply ReasonSmith to examine the model and the data.

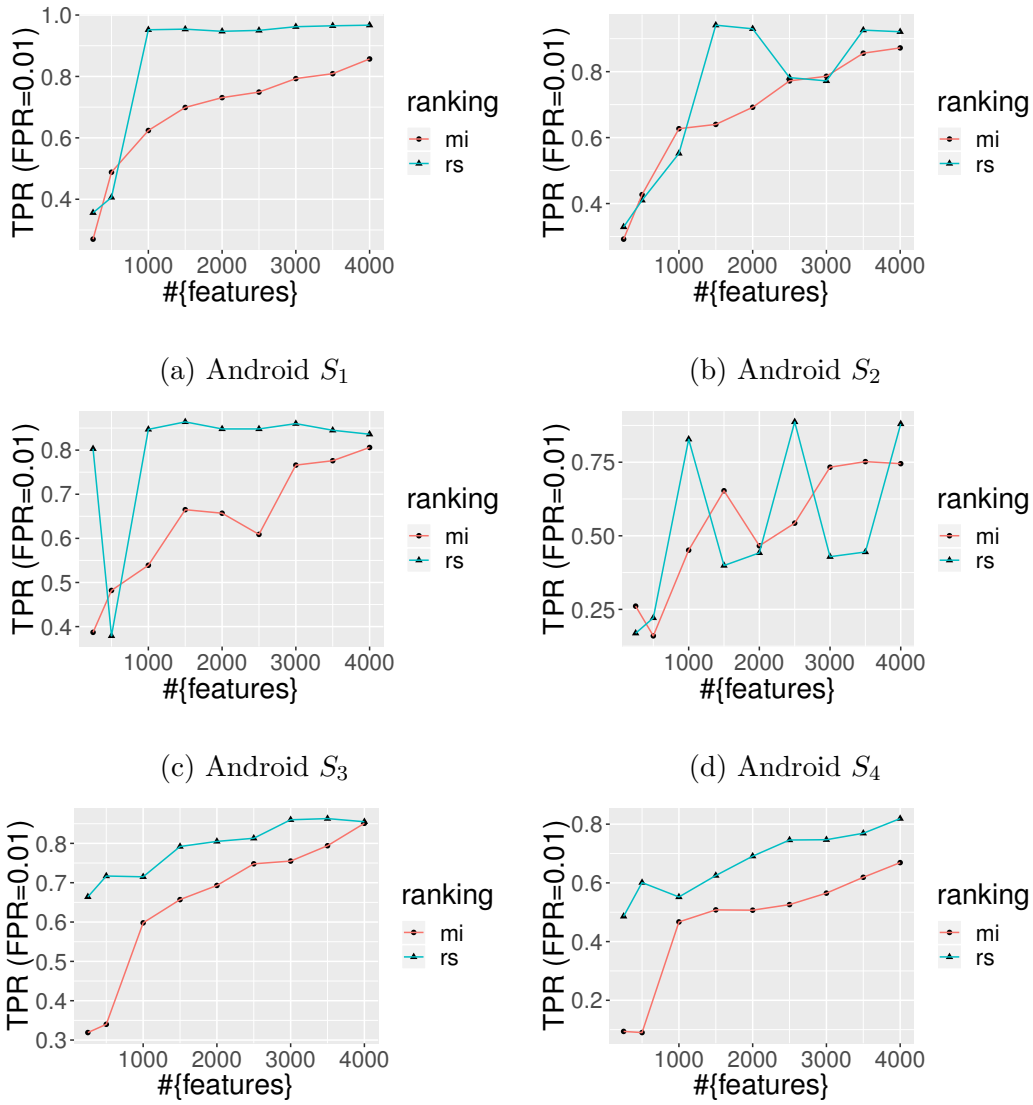
4.3.1 Feature ranking performance

As we discussed in Chapter 4.1.1, we measure the performance loss if we use a subset of features in testing. In general, dropping features causes the model losing

information learned during training, and therefore performance should decrease[1]. We choose mutual information as the baseline metric to rank features, which is usually used to measure feature global importance in tree-based models. For a fair comparison, two feature sets must contain the same number of features. Because machine learning system returns the likelihood of maliciousness, which will have different results using different thresholds. To make the performance comparable, we use true positive rate as basic performance metric when false positive rate reaches 0.01.

Figure 4.1 shows the true positive rate for both training and testing sets when true positive rate is 0.01. Figure 4.1a and 4.1e shows the experiment for training set. If we remove the features with low ranks, then the performance drop is smaller for ReasonSmith ranking than that for mutual information ranking. ReasonSmith outperforms mutual information in all feature sizes, which shows that ReasonSmith captures important features learned by neural networks. It also suggests that some important features by neural networks are *invisible* by only looking at the relationship between output and individual features. In addition, ReasonSmith has a better performance than mutual information based ranking in most of experiments for testing sets. However, because the estimation is based on training sets and it is not a guarantee that the underlying distribution of testing set is the as training set, ReasonSmith ranking has lower false positive rate than mutual information ranking in some cases.

[1]If the model over-fits the data, then dropping features might enhance the detector performance.



(a) Android S_1

(b) Android S_2

(c) Android S_3

(d) Android S_4

(e) Windows S_5

(f) Windows S_6

Figure 4.1: Performance of malware detector when only a subset of features is used in testing. "rs" represents ReasonSmith based ranking, while "mi" represents mutual information based ranking. Note that S_1 and S_5 are used as training set.

4.3.2 Malicious behaviors

To identify the features that are robust over time, we train separate models for each data set using the same configuration, and run ReasonSmith to rank the features. Then we calculate mean and variance of the rankings from different data set.

Features that are consistently important across different training sets are robust against concept drift or sampling biases. However, These features can be either suspicious behaviors or artifacts caused by a particular environment. To verify if DNN model actually learns and generalizes suspicious behaviors, we manually examine the features with high ranking mean.

For Windows data set, top 10 features are listed below.

1. Load `advapi32.dll`
2. Load `user32.dll`
3. Load `kernel32.dll`
4. Execute `(null)`
5. Write `c:\docume~1\alluse~1\applic~1\mozilla\ctvqzym.exe`
6. Write `%path%\tasks\hiqvsnd.job`
7. Open `HKLM\system\setup\software\microsoft\windows \currentversion\explorer\user shell folders`
8. Query value `HKLM\software\microsoft\cryptography`

9. Open `HKLM\software\policies\microsoft\windows\safer\codeidentifiers`
10. Load `user32`

The following features have highest average ranking in Android experiment.

1. Feature `android.hardware.touchscreen`
2. API call `getSystemService`
3. Intent `android.intent.action.MAIN`
4. API call `android.content.Intent.setDataAndType`
5. Intent `android.intent.category.LAUNCHER`
6. Permission `SEND_SMS`
7. Activity `.MainActivity`
8. Permission `READ_PHONE_STATE`
9. API call `org.apache.http.impl.client.DefaultHttpClient`
10. Permission `INTERNET`

Known behaviors. We compare features with highest ranking mean with analyst features to test if machine learning model and human analysts can reach an agreement on important features. Figure 4.2 shows the number of overlapping features between two feature sets. In general, the top features selected by models are different from analyst features. Only 27 features (Android) and 2 features (Windows) are in the top 100 features for the models.

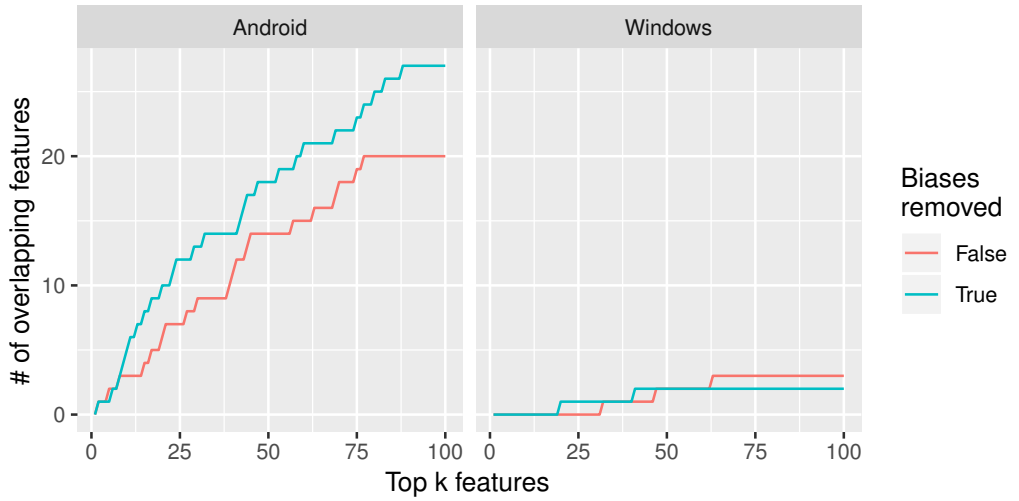


Figure 4.2: The number of overlapping features between data-driven features and knowledge-based features. Biases removed means feature set is selected based on significance across different data groups, and we use the feature ranking directly from training set in the case without bias removal.

In addition, we do the same analysis for the feature ranking without removing data biases. We use the ranking from S_1 and S_5 as examples. We find that it is more likely to include analyst features if we use ranking mean to rank features than simply using ranking from one data set. It suggests that data biases memorized by the machine learning model is less important in a long-term evaluation. By combining the knowledge over time, the model is more likely to generalize the knowledge that is close to human analysts. It also suggests that human analysts take into account if the features are useful in the future when they engineer features.

We find that machine learning system actually learns some similar knowledge as human analysts. For example, the presence of `vboxminirdrdn` (rank 20) can be used detect if the program is running in virtual environment, which is included in

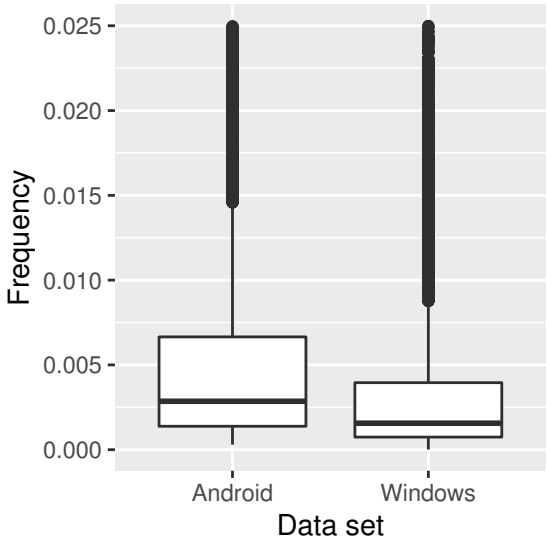


Figure 4.3: Feature frequency. We remove feature outliers with feature frequency greater than 0.025.

Cuckoo signatures.

Compared to Windows experiment, the Android model is more likely to reach an agreement with human analysts. One possible reason is that Android features have higher occurrence in general, and human analysts are good at finding features from common behaviors. However, Windows features, which includes the argument string, have more variations, which makes it difficult for human analysts to identify the common behaviors. Figure 4.3 shows the feature frequency for both Android and Windows.

New behaviors. Machine learning model is also able to learn suspicious behaviors that are overlooked by human analysts. For example, querying value from registry key `HKLM\software\microsoft\cryptography` (rank 8) indicates an intent to get machine GUID. This behavior itself is not malicious, but it is usually the first step

for attackers to fingerprint the victims and keep tracking them. Similar behaviors are also used for Android malware, which uses API call `getDeviceId` to get the machine ID.

Another example of the missing behavior is querying value from registry key `...\explorer\shell folders`, which ranks 17 in our experiment. The registry key stores the default path of user shell folders, for example, "My Documents", "My Photos", etc.. It might correspond to the attempts to access user's data, e.g. steal personal information or encrypt user documents.

4.3.3 Artifacts

Since machine learning model only generalizes security knowledge from data, the most significant and robust features used by the model is likely to result from data artifacts. We find that many top features in our experiment come from data artifacts. Even though the model can achieve high true positive rate with low false positive rate in experimental setting, the features may not be meaningful for human analysts.

In summary, we find 3 types of data artifacts from our experiments, i.e. environment-sensitive features, side-effects from malicious behaviors, and missing components.

Environment-sensitive features.

Environment-sensitive features are the features that directly relate to the malicious behaviors of malware, but the behaviors might be different in different envi-

ronment.

For example, `...\mozilla\ctvqzym.exe` (rank 5) and `%path%\tasks\hiqvsnd.job` (rank 6) are two malicious payloads from `Win32/Kryptik` malware family. Our dataset contains 4589 samples from this family, and every sample show the same pattern of activity: an executable payload named `ctvqzym.exe` is dropped in a folder mimicking mozilla and a schedule task to maintain persistence is created using the `hiqvsnd.job` file. All samples used the exact same names and file paths for their dropped files.

Although these features are important in our dataset to distinguish malicious and benign samples, they may not be extended outside Cuckoo sandbox environment. For example, while the combination of dropping the executable in a mozilla folder and creating the scheduled task in a `.job` file is a behavior performed by all samples in the Kryptik family, the consistent file names is not. We find that running the same samples in *a different sandbox* yield different file names for both the scheduled task file and the dropped executable. We run the sample in two other sandboxes, on the first, the executable is named `byiamvi.exe` and the scheduled task file is named `rwdqoxi.job`, on the second, they are named `ltyfsyc.exe` and `bikzrwg.job`. At a high level, the way the model learned to detect samples from the Kryptik family is through the combination of the above two features, creating the executable and setting up the scheduled task. In reality, the model learned to hone in on two values which are artifacts of the used sandbox environment. In practice, the model will not be able to correctly classify Kryptik samples on any host that is not configured the exact same way the sandbox was.

Another example of environment-sensitive artifact is connecting to `10.0.2.2`, which ranks 18 in our experiment. The IP address is in internal IP ranges, which is used as the gateway for guest OS by default in Windows when it tries to connect to host OS. The event can appear when the malware scans network ranges, or attempts to move laterally to other machines. All the activities from the malware are isolated in sandbox environment, and the network connection is guarded through an internal gateway, which in our case is `10.0.2.2`. However, the network traffic will never go through the internal gateway in practice.

The third example is executing `(null)` commend (rank 4). This feature is the event that an command execution is monitored but Cuckoo fails to obtain the actual command string. This is possibly because malware launches other application via an unusual way which causes some unexpected errors in Cuckoo to obtain the command string. It does not tell the actual command the program trying to run, but makes the behavior stands out by accident.

Side-effects from malicious behaviors. Different from environment-sensitive features, we use *side-effects* to refer to the behaviors that are not from attackers, but are highly correlated with malicious behaviors.

For example, dynamic analysis is likely to include behaviors from operating system. To run start a new process, system might check software restriction policy from registry key `...\safer\codeidentifiers` (rank 9) and search for the command under registry key `HKCR\*\shell` (rank 16). To communicate with remote server, the system might open registry key `...networkprovider\hworder` (rank

19) that stores the network provider order and open pipe `pipe\wkssvc` (rank 21) that maintains network connection with remote server. In addition, the relationship between these correlated behaviors might be difficult to verify if they are applicable in different environments, e.g. different Windows versions.

Missing components. Because malware usually have limited functionalities, malware may not have some common modules that are usually implemented in benign applications. For example, feature `android.hardware.touchscreen` (rank 1) is common and present in the manifest in almost every Android applications that require touch screen, and therefore it is impossible to be considered as important for human analysts. Table 4.5 shows the percentage of applications that do not require touch screen feature. We find that the percentage of malware is high for the apps without touchscreen feature especially when the apps exhibit sensitive behaviors. One possible reason is that there is no need for malware author to implement a UI if the malware is only a service. Another benefit for disabling touchscreen is that the malware is likely to be exposed to more devices from Android market like Google Play where the app list is determined by customers' devices. However, these features cannot tell any actual behaviors of Android apps, event though they are useful to distinguish malicious from benign apps. In addition, attackers can easily create evasive samples by adding those *missing components* in Android manifest, which makes the system vulnerable in practice.

Table 4.5: Percentage of malware under certain conditions.

Condition	Total	No touchscreen
Call `bin/su`	83.5%	100%
Call `getSubscriberId`	62.2%	92.3%
Call `getDeviceId`	38.8%	72.7%
Connect to `maps.google.com`	50.9%	89.7%

4.3.4 Biases and concept drift

Features that are important to only a few training sets indicates the differences of data sets. Such difference possibly results from sampling bias and concept drift. It is likely that during a specific time one particular malware family is prevalent but suddenly becomes inactive at some point. The sampling bias makes system vulnerable by emphasizing the behaviors from a specific malware family. Additionally, the difference in data sets can be the result of concept drift as well. It is well-known in security community that malware is always showing new behaviors in order to evade detections. Top 10 features with highest ranking variance in Android data are listed below.

1. Intent `android.intent.action.BATTERY_CHANGED_ACTION`
2. API call `android.os.Handler.sendMessageDelayed`
3. Service receiver `com.google.update.UpdateService`
4. Activity `com.google.update.Dialog`

5. Service receiver `com.google.update.Receiver`
6. API call `java.util.SortedMap.entrySet`
7. API call `java.util.GregorianCalendar.after`
8. API call `android.widget.TextView.setInputType`
9. Activity `.Activity1`
10. API call `java.io.RandomAccessFile.writeByte`

The top 10 features with highest ranking variance for Windows are listed below.

1. Write `c:\python27\pythonw.exe`
2. Open key `HKCR\jpegfile`
3. Open key `HKCR\http`
4. Open key `HKLM\software\microsoft\windows\currentversion\internet settings\zonemap\domains\dnsnb8.net`
5. Copy `%programfiles%\winrar\rar.exe`
6. Open key `HKCU\software\microsoft\windows\currentversion\internet settings\zonemap\domains\dnsnb8.net`
7. Create key `HKLM\software\gtplus`
8. Open `HKLM\system\currentcontrolset\control\securityproviders\saslprofiles`
9. Create `c:\documents and`

10. Create `c:\documents`

Figure 4.4 shows the top 10 features with highest ranking variance for Android data set. Most of features are dominant in one particular family (either benign or malicious). If one malware family is prevalent in a particular time, then the model learns it as the general knowledge for security. If the malware family becomes less prevalent or changes the behavior, then the important features learned from the past is no longer useful. For example, intent `BATTERY_CHANGED_ACTION` is used only in DroidKungfu family in our data set, and therefore is useful in detecting malware for the model. However, DroidKungfu family becomes less and less prevalent in dataset. 9.2% of malicious apps belong to this family in S_1, but the percentage decreases to 4.7%, 1.9% and 2.1% in S_2, S_3 and S_3 respectively. It suggests that this intent becomes less useful in detecting malware over time. Similarly, activity `com.google.update.Dialog`, service receiver `com.google.update.Receiver` and `com.google.update.UpdateService` become less important since DroidKungfu gradually disappears. It is common in security that malware keeps evolving due to arms race. If the malware is detected and the attack becomes less effective, then the attackers have to change their strategy. Since we cannot distinguish if the malware indeed disappears in the wild, the drifted security knowledge can be a result from sampling bias as well.

We find a similar trend for Windows experiment as well. Figure 4.5 shows the distribution of malware families for the top 10 features with high uncertainty. Open registry key `...\zonemap\domains\dnsnb8.net` is mostly observed from fam-

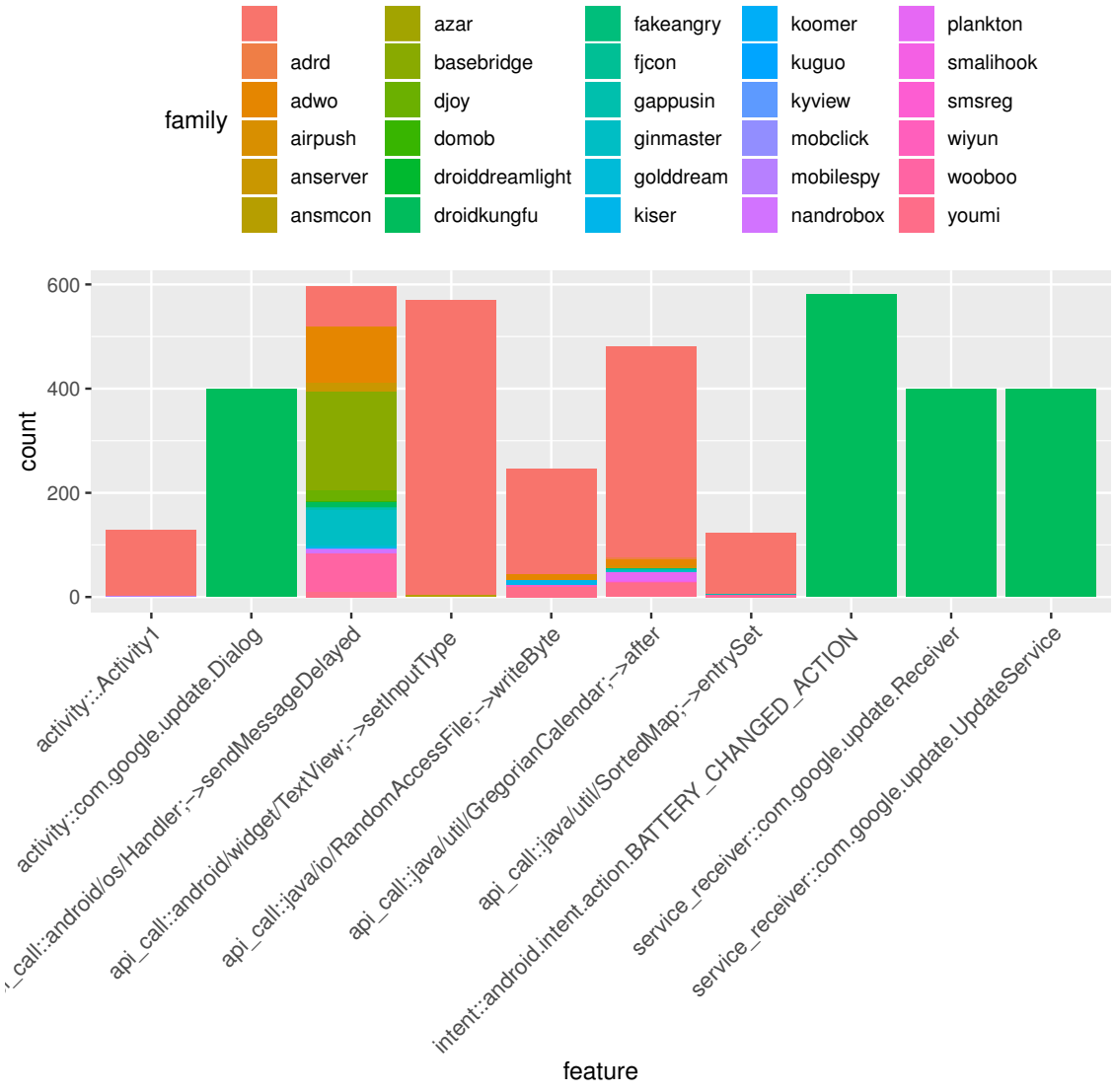


Figure 4.4: Top 10 features with the highest uncertainty (Android). An empty family represents benign samples

ily `wapomi`. The frequency of this malware family is 2.7%, and the frequency of this family decreases to 0.5%. Although the frequency change is not large, the machine learning model is sensitive and is able to capture the changes in the data.

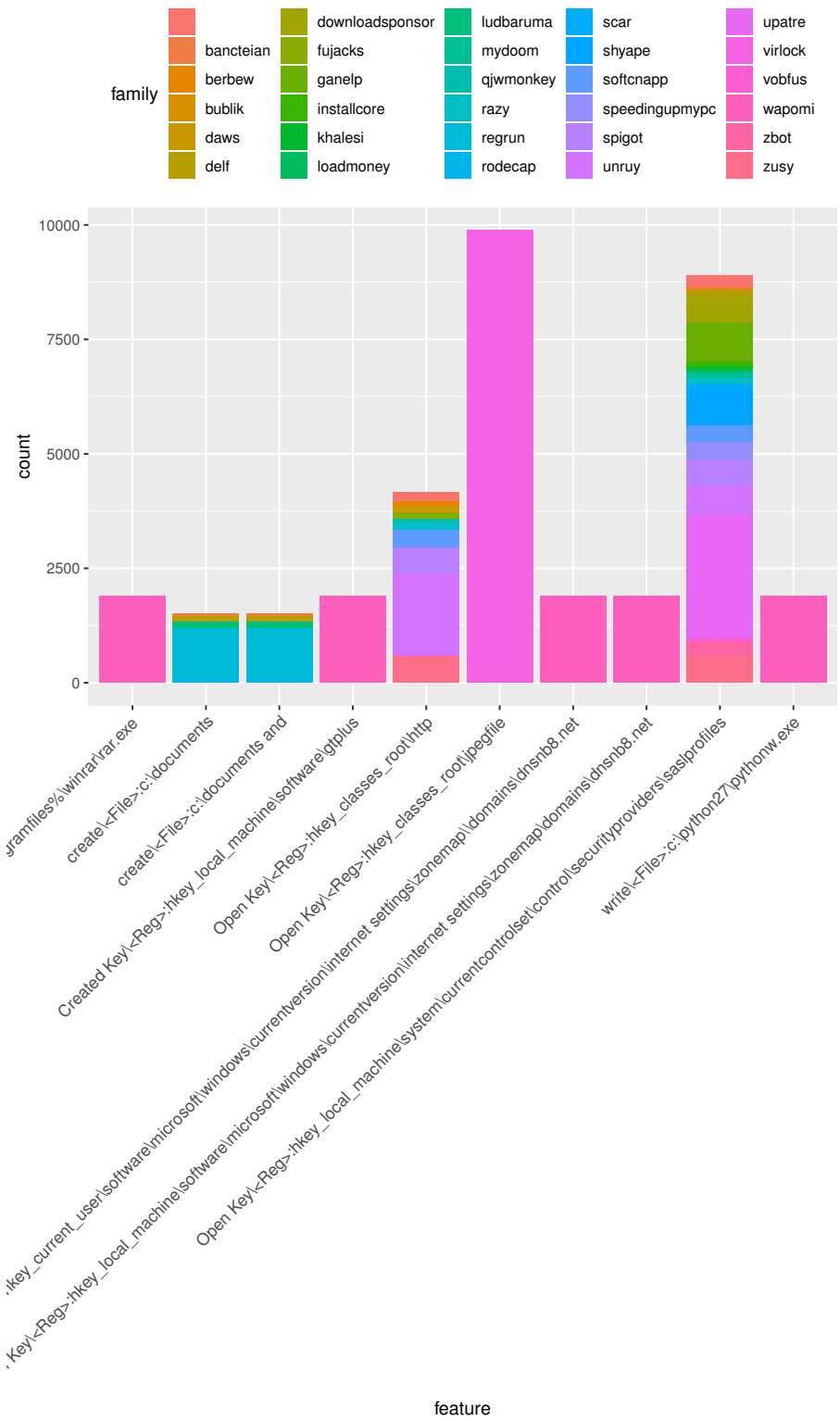


Figure 4.5: Top 10 features with the highest uncertainty (Windows). An empty family represents benign samples.

Chapter 5: Feature engineering for malware campaigns

In this chapter, we explore the feature engineering problem for malware delivery campaigns. In Chapter 5.1, we introduce a chain model for malware delivery campaigns, which considers both existing campaign ontology and common information mentioned in security blogs. In Chapter 5.2, we evaluates the performance of IOC detection and categorization. In Chapter 5.3, we analyze the effect of different baiting techniques on binary delivery.

5.1 Malware delivery model

5.1.1 Definition

The emerging standards for sharing threat intelligence [23–25] are based on the observation that cyber attacks often follow certain high-level patterns. This allows analysts to define generic models for representing and sharing the information.

In particular, the STIX standard [25] has been adopted by an increasing number of security products [119]. STIX defines a comprehensive schema that uses IOCs to describe campaigns [25]. In this schema, a `campaign` consists of a set of `indicators` which specify an attack pattern. An `Indicator` is then defined to be

a set of `observables`, which includes all the resources and infrastructure in the attack pattern. An `Observable`, defined in CybOX (v2.1) [24], contains 88 different types including URI, IP address, file, registry key, etc.. To add semantics for the attack pattern, STIX defines `kill_chain_phase` as an attribute for `indicator`. A *kill chain*[1] breaks down an attack in a sequence of stages. The original definition of the cyber kill chain [103] specified 7 stages: reconnaissance, weaponization, delivery, exploitation, installation, command & control and actions on objective.

Malware delivery campaigns. To represent malware delivery campaigns, we adopt three stages from the STIX data model: *exploitation*, *installation*, and *command & control*. Additionally, we redefine the first stage of these campaigns based on our observation that, while attackers will sometimes gain an initial foothold on a host by delivering malicious files (e.g. as email attachments), in other causes they rely on malvertising to lure users to an exploit kit's landing page or to persuade them to download and install a dropper. We therefore replace the delivery stage with a *baiting* stage, which captures all these strategies.

There are 4 different types of `indicators`, which cover the most common and important stages involved in malware delivery.

1. *Baiting*: This is the first stage of delivery campaign. The most common approaches to draw users to the malware delivery chain are email spam, malvertising and compromised sites. Once the user clicks on a link from a spam

[1] "Kill chain" is a military term describing the stages of an attack that results in the destruction of a target (i.e. a "kill"). Separating the attack stages is helpful for identifying different defensive techniques that could break the chain and disrupt the attack.

message, or visits a compromised site, the browser will be redirected to a malicious site. Before landing on the exploit server, users might be redirected through several relay pages, which are included this stage.

2. *Exploitation*: After user is redirected to the landing page of an exploit kit, the exploit server fingerprints the browser and the installed plugins and tries to identify open vulnerabilities. The server then tries to exploit a vulnerability and deliver a payload.

3. *Installation*: A malicious payload is downloaded on the end host, and the exploit server installs and executes the malware. Without using exploits, attackers can also lure users to install and execute the malware through social engineering.

4. *Command & Control*: The executed malware contacts its command and control server and receives remote commands. The commands may involve downloading the next stage of the malware, dropping unrelated samples (e.g. in the case of a Pay-per-Install infrastructure [31]) or performing other malicious actions (e.g. spam, DDoS).

These stages are not necessary for every payload delivery. For example, campaigns can entice users to download and install the payload by users themselves through social engineering. In this case, the exploit stage would be unnecessary.

Table 5.1 lists all the observables we use for each type of indicators. `Observable` contains 6 types, i.e. *URL*, *IP address*, *file hash*, *malware family*, *exploit kit*, and *vulnerability*, in which CybOX `observable` includes the former 3 types. Figure 5.1

shows one example of how the malware delivery schema can well fit the actual blog posts. Our payload delivery model simplifies the data model in STIX in that (1) we primarily focus on network activity in the campaign rather than the behavior on the victim machine, and (2) many `observables` are unlikely to be discussed in security articles and cannot be used to uniquely determine the campaign. Note that the term IOC (indicator of compromise) is equivalent to the concept of an observable in STIX. In order not to confuse the reader, we use the terms IOC and indicator phase instead of STIX's observable and indicator in rest of this paper.

A *campaign group* corresponds to the actors of a campaign, represented by a set of IOCs. A *supplier* corresponds to an actor that provides delivery services by controlling droppers in the field. A *tier-1 supplier* is a supplier that operates baiting and exploitation domains.

High-risk binaries. We utilize the VirusTotal service [110] to assess the binaries recorded in the measurement data. VirusTotal provides file scan reports for up to 54 anti-virus (AV) products. In line with prior work [33], if a binary receives more than 30% anti-virus detection from VirusTotal, then we considered it a *high-risk binary*. Because the blog posts discuss both malware and potentially unwanted programs (PUPs), we do not aim to distinguish if a specific binary is malware or PUP. Instead, we focus on binaries that present a *high risk* to the end hosts because (1) they are highlighted as security threats in the articles we analyze, and (2) they have a high detection rate among AV products. A *high-risk download* is a download event with a high-risk binary payload.

...
Upon clicking on the links, users are exposed to the client-side exploits served by the latest version of the **Black Hole** Exploit Kit.
...
Sample compromised URLs used in the campaign:
hxxp://www.alacinc.org.nz/impdiscm.html;
...
Client-side exploits serving URLs:
hxxp://netgear-india.net/detects/discover-important_message.php;
Upon loading, these URLs attempt to exploit **CVE-2010-0188** by

dropping a malicious PDF file.
Sample detection rate for the dropped malware:
MD5: **80601551f1c83ee326b3094e468c6b42**
detected by 4 out of 44 antivirus scanners as
UDS:DangerousObject.Multi.Generic.
Upon execution, the sample phones back to
200.169.13.84:8080/AJtw/UCyqrDAA/Ud+asDAA

(a) Blog post from Webroot [120]

```
{
   baiting: {
      server_URL: ["www.alacinc.org.nz"]
   },
   exploitation: {
       server_URL: ["netgear-india.net"],
       exploited_vulnerability: ["CVE-2010-0188"],
       exploit_kit: ["Black Hole"]
   },
   installation: {
      malware_hash: ["80601551f1c83ee326b3094e468c6b42"],
      malware_family: ["UDS:DangerousObject.Multi.Generic"]
   },
   C&C: {
      C&C_server_IP: ["200.169.13.84:8080"]
   }
}
```

(b) Structured malware delivery campaign

Figure 5.1: An example of mapping unstructured blog post to structured schema for malware delivery.

Table 5.1: IOCs and indicator phase in malware delivery model.

Phase	IOCs	IOC Type
baiting	attachment_hash	hash
	attachment_family	malware family
	server_URL	URL
	server_IP	IP
exploitation	exploit_site_URL	URL
	exploit_site_IP	IP
	exploited_vulnerability	vulnerability
	exploit_kit	exploit kit
installation	malware_hash	hash
	malware_family	malware family
C&C	C&C_server_URL	URL
	C&C_server_IP	IP

Generality. Our system for mining security articles requires a model of the malicious activity to extract the relevant semantics. The model defined in this section is specific to malware delivery campaigns; however, our NLP techniques are generic and may be applied to other models. The results in this paper could be extended to other cyber threats by defining a pertinent model, for example by starting from an existing standard such as STIX.

5.1.2 Classification

In this section we describe the design of ChainSmith, a system that extracts and categorizes IOCs from security technical articles in colloquial English, according to the schema defined in Chapter 5.1.1. The key intuition behind ChainSmith is that the context words in adjacent sentences indicate the stage of a campaign, and the context words that directly relate to the IOC determine its level of maliciousness. In addition, we are able to group IOCs from different actors by considering the campaign stages and article post time.

Figure 5.2 shows the architecture of ChainSmith. The system consists of 6 components: *article crawler*, *expression detector*, *syntactic parser*, *semantic parser*, *named entity recognition*, and *IOC classifier*. In this chapter, we first describe the data collection in Chapter 5.1.3 and then introduce the design of the classifier in Chapter 5.1.4.

5.1.3 Data collection

Just like the actors involved in malware delivery tend to specialize on narrow tasks, security analysts also focus on specific phases of the delivery campaigns. The full picture of an end-to-end campaign often emerges only after reading articles from multiple sources. We therefore implemented a generic crawler to collect security articles published online.

We use our crawler to mine 10 sources, listed in Table 5.2. We select these sources in that (1) the articles are likely to include detailed information about the

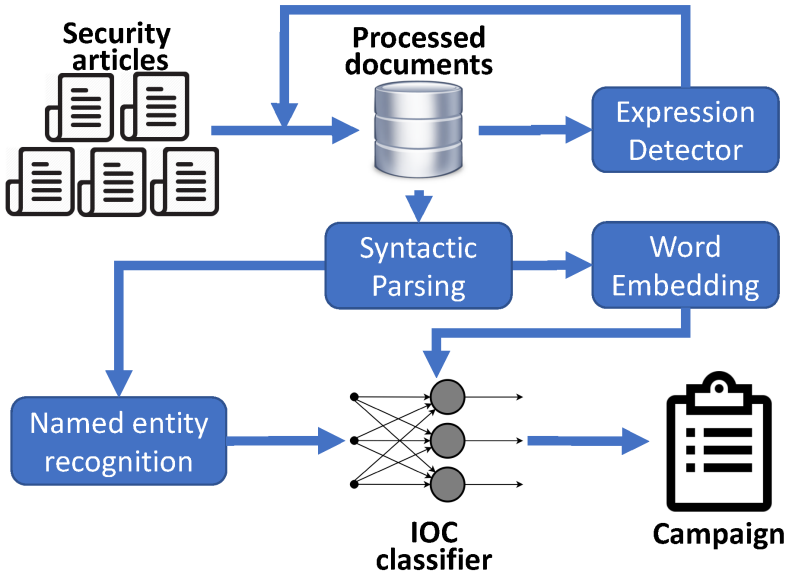


Figure 5.2: Architecture of ChainSmith. The article crawler module is not shown in this figure.

campaign, and (2) the sources are diverse—including news websites covering cyber threats, blogs from anti-virus companies, or the personal blogs of security experts.

5.1.4 Campaign extraction

Since the design of *expression detector*, *syntactic parser*, *semantic parser*, *named entity recognition* follows standard syntactic analysis (Chapter 1.4) or has been used in FeatureSmith, we focus on *IOC classifier* in this chapter.

The primary goal of this step is to identify whether the given word is an IOC as well as the stage of the campaign it belongs to. Basically, we select 2 types of features for this task.

Sentence-level feature. This type of feature captures the topic of the sentence and is useful to detecting the categories of features. First, we identify informative

Table 5.2: Summary of security articles. For some articles, we fail to identify the posting time.

Article source	**Time span**	**Count**
Forcepoint	2010-02-11 – 2017-05-13	227
Hexacorn	2011-10-01 – 2017-05-15	331
Malwarebytes	2012-04-20 – 2017-05-15	1590
Sophos	2000-11-24 – 2017-04-17	1171
Sucuri	2009-09-13 – 2017-05-12	927
TaoSecurity	2003-12-01 – 2017-05-08	2653
Trend Micro	2009-09-13 – 2017-05-15	1382
Virus Bulletin	2005-09-01 – 2016-01-29	601
WeLiveSecurity	2009-05-08 – 2017-05-16	4295
Webroot	2009-03-23 – 2017-05-14	978
Total	2000-11-24 – 2017-05-16	14155

words in the sentence. We use Equation (5.1) to estimate the importance of a word in identifying topics,

$$S(w) = \max_{t \in T} \frac{p(w|t)}{p(w)} \tag{5.1}$$

where T is the set of categories, w is word we evaluated, $p(w)$ is the probability of word w, and $p(w|t)$ is the probability that word w will be used for describing topic t. A higher word score means that w is more likely to be used in a certain campaign phase. For example, *iframe*, *src*, *malvertising* have high scores for the

topic phishing, and *exploit-serving* has a high score for topic of exploitation. We consider *informative words* to be words with high score and high occurrence. For each sentence, we determine the *context* from 3 sources: (1) informative words from the current sentence; (2) informative words from previous sentences, if no informative words are found for the current sentence; and (3) informative words from the previous sentence that mention the same IOC. The motivation behind source 2 is that the topic of an article is usually consistent. For example, if the topic of current sentence is command and control, then it is very likely that the next sentence is also discussing command and control. The last context source comes from the observation that the same entity can be discussed repeatedly in different sentences. We use 3 types of features to identify the sentence topic:

- Average word embedding of the context words.
- Average word embedding of the article title.
- Number of entities of each type.

Word-level feature. This type of feature provides word-specific information and is useful to provide additional validation for classification results. The entities in the same sentence may not be of the same category. Therefore, I propose some word-level features that differentiate between the entities.

- Average dependency embedding for all dependencies connected to the named entity.
- Type specific feature.

The first feature is useful in determining which category the named entity belongs to, since the dependency tells the grammatical structure within a sentence. The type specific feature is useful to tell if an entity is malicious. For example, the mis-click prevention strategy (e.g. "hxxp" and "[dot]") is an indicator of the maliciousness of URLs and IP addresses.

Classifier design. We first train a classifier to check if the topic of a sentence falls in any of these 4 phases using sentence-level features. Because the topic may not be mutually exclusive, we train 4 binary classifiers to identify topic probabilities. We implement the classifier using neural network with 1 hidden layer and 50 hidden nodes. To reduce the false positives, we skip the IOC candidates if no informative words are found.

Next, we use the result of topic classification as the feature for another neural net for IOC classification. The candidate IOCs extracted in the previous steps may not be related to malware delivery campaigns. For example, a file hash mentioned in the blog post may refer to a benign executable, such as a new patch or anti-virus product. To verify that the named entities identified represent IOCs of a campaign, we train a multi-class classifier for each entity type (e.g. URL, IP address) using both topic probabilities and word-level features. Instead of applying a softmax layer to the output layer, we use a logistic function to scale the output probability, such that the sum of the probabilities is not necessarily to be 1. To reduce the false positive rate, we add a special label "malicious" as another output, for the case when the classifier returns more than one label from a single sentence. For these

IOCs, the classifier cannot reliably tell which stages they belong to. Therefore, rather than drop the results, we label them as "malicious".

Campaign group identification. Technical articles are likely to mention both suppliers and customers in the underground economy. From our manual investigation, we find that different from scientific literature, security blogs usually discuss one observation from single campaign rather than investigate the problem in a broader view. For example, blog posts are likely to record the current status of single campaign as shown in Figure 5.1. However, unlike scientific literature, the comparison between different campaigns is usually absent from blog posts. Therefore, we assume that each technical article discusses one supplier and multiple customers. For each article, we group all the IOCs in baiting and exploitation phase as one group and consider the other IOCs as individual groups (one IOC for each group). In addition, as campaigns may change infrastructure and strategies over time to evade detection, we must connect campaigns from different articles in order to study the long-term behavior and campaign evolution. We use a more conservative idea to connect different campaigns with the same IOCs than the method in [57] . If two campaigns contain identical IOCs and appear within a short time frame of another, then we consider that they are in fact the same campaign. For example, if one domain is mentioned by two articles in May 2012 and August 2012, then we group the campaigns from two articles. In this study, we choose the time window to be 6 months, which means that the campaign groups are merged only if they share common IOCs and are discussed within 6 months of one another.

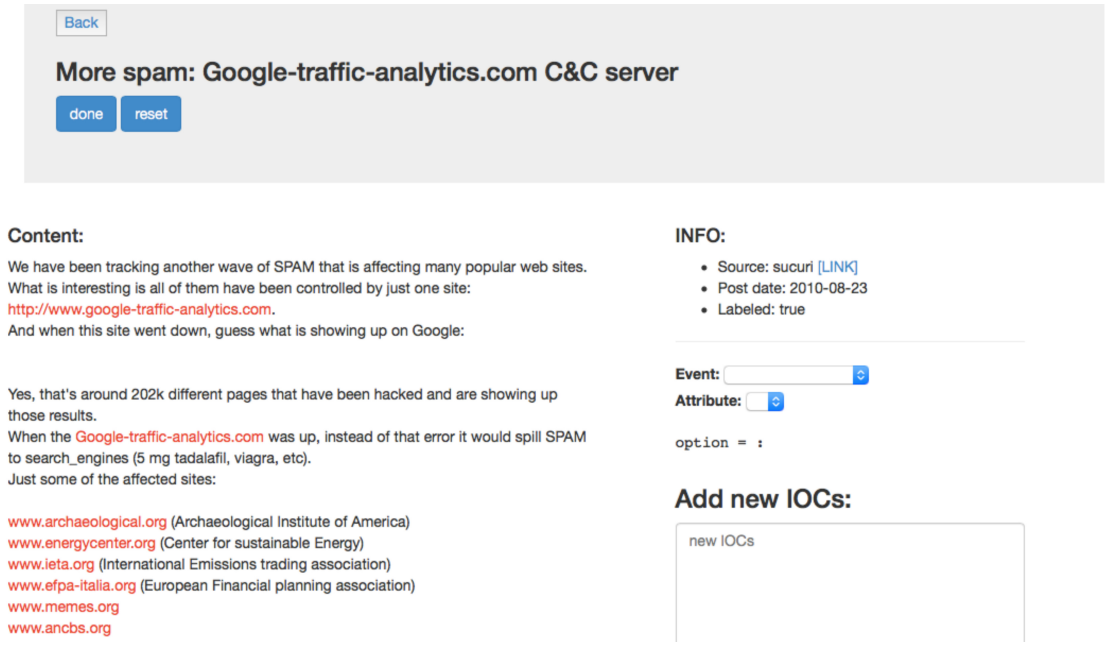


Figure 5.3: Screenshot of annotation website.

5.2 Evaluation results

A key challenge for semantics extraction is the lack of a ground truth. While IOCs are available on several threat intelligence platforms, the malware delivery phase that these IOCs correspond to are usually unavailable. To train and evaluate ChainSmith, we built a web application for manually annotating articles, as shown in Figure 5.3. The annotators can first specify which campaign stage the IOC belongs to and then enter the IOC name in the box. This tool allows us to collect the ground truth for our system. 4 graduate students from our group participated in the annotation exercise. To increase their chances of extracting meaningful information about malware delivery campaigns, we presented only articles that contain at least 20 named entities for labeling. In this way, we annotate 153 articles and 6264 IOCs.

Even with the ground truth collected from manual annotation, off-the-shelf NLP techniques are inadequate for characterizing malware delivery campaigns. We design a baseline system that models the campaign stage classification using traditional topic modeling. We consider each sentence as an individual document, and train a topic model using Latent Dirichlet allocation (LDA) [121]. LDA is an unsupervised algorithm widely used for topic modeling. For example, the Toronto Paper Matching System (TPMS) [122] employs LDA to model the reviewers' research areas and the areas of papers under submission. Additionally, a growing number of conferences utilize TPMS for making automated reviewer assignments. After training in this manner, we identify the topic for each stage using the training set. If a sentence belongs to one malware campaign stage, then we extract all the named entities and label them as the attribute of that stage.

We use 5-fold cross-validation to evaluate the performance of ChainSmith and of our baseline system. Table 5.3 shows the results. ChainSmith achieves 91.9% precision and 97.8% recall, which is 13.7% and 31.1% higher than the baseline model, respectively. Since the campaign stage classification is a multi-class problem, we take the average precision and recall for all classes. Out of the detected IOCs, we are able to classify 86.2% of them into a campaign stage with 78.2% precision and 80.7% recall, which is more than twice as high as the baseline model. [2] ChainSmith outperforms the baseline because it is able to better determine the sentence context

[2]Because ChainSmith is able to detect but fails to categorize part of IOCs, we cannot precisely calculate the recall . For a fair comparison to the baseline model, we calculate the lower bound by multiplying the classification rate (86.2%) to the recall (80.7%).

Table 5.3: Performance comparison of ChainSmith and a baseline NLP system, which utilizes Latent Dirichlet Allocation.

IOC detection		
	Precision	Recall
ChainSmith	91.9%	97.8%
Baseline	78.2%	66.7%
Stage classification		
	Precision_avg	Recall_avg
ChainSmith	78.2%	69.6%– 80.7%
Baseline	37.0%	28.7%

and the informative words.

We run ChainSmith on all 14,155 articles we collected, and discover 24,653 IOCs. Table 5.4 shows the summary for each type of IOCs. In addition, from the extracted IOCs, we further identify 8,902 campaign groups mentioning either suppliers or customers. We use one Sun Fire X2200 M2 server with 8 logical processors and 8GB RAM for this experiment. Extracting IOCs from one security article requires 0.214 seconds on average for each article.

To assess the need for ChainSmith, we collect 568,348 fully qualified domains and 297,218 IPv4 addresses from *Hailataxii* [29], a repository providing threat intelligence feeds in the STIX format [25]. [3] Compared to the IOCs collected from

[3]The data is collected by 2016-12-15.

Table 5.4: Summary of extracted IOCs.

Stage	**Attribute**	**Count**
baiting	`server_URL`	4,874
	`server_IP`	1,001
exploitation	`exploit_site_URL`	3,735
	`exploit_site_IP`	732
	`exploited_vulnerability`	554
	`exploit_kit`	572
installation	`malware_hash`	4,263
	`malware_family`	1,604
C&C	`C&C_server_URL`	2,161
	`C&C_server_IP`	1,271
unknown		3,879
Total		24,653

ChainSmith, only 60 overlapping domains and 26 overlapping IP addresses are found in both data sets, and Jaccard index is $1.0 * 10^{-4}$ and $8.7 * 10^{-5}$ for domain and IP address respectively. This illustrates the fact that most of the indicators discussed in the threat intelligence reports are not made available in a machine-readable format. Moreover, none of the IOCs from Hailataxii specifies the `kill_chain_phase` attribute, which would preclude the analysis we conduct in the rest of this section from being performed on the dataset.

Next, we link the campaign groups extracted from blog posts to Kwon *et al.*'s data set of download events [30]. This data set is based on Symantec's WINE platform, which provides security telemetry from real-world hosts. Each download event in the data set specifies the file hash of the downloader and its payload (the downloaded file), and optionally the URL or IP from which the payload was downloaded. In total, the telemetry data records 50.5M download events from 5M real-world users. Using the download event data, we check if IOCs for each campaign group are used as the downloader or downloading portal. We are able to find the subsequent download events for 59 groups, 37 of which produce high-risk downloads in the measurement data. We label all the suspicious download events based on the campaign IOCs. Consequently, we further discover 224 suspicious downloaders and 3,395 suspicious payloads.

Data release. We plan to release the ChainSmith system in the form of a Web application at `http://ioc-chainsmith.org`. We set up a web crawler for collecting articles each week. Then ChainSmith parses the articles and updates the database

with new results. The website provides the entire data for download as well as a search interface to access the data.

5.3 Security implications

5.3.1 Persuasion techniques

To entice users to download the payloads, attackers are creative in designing persuasion methods. Email spam, compromised websites, and malvertising can be used as the basic approaches to targeting large group of individuals. In addition, the attacker can post deceptive advertisements to persuade users to click on malicious link. By integrating human discovery from blog posts and the real-world download data from WINE, we aim to study different persuasion strategies as well as their impact on subsequent download behaviors.

We identify 59 campaign groups that are both reported by security analysts and WINE. Then we manually label the campaign groups based on the platform where the information is displayed and the trigger that initiated the download, e.g. the message that lures users to click or exploit kit that exploits a machine directly. As a result, we are able to label baiting techniques for 44 campaign groups.[4]

Table 5.5 shows the top campaign groups that drop the most high-risk binaries in WINE. To study the timeline of campaign groups, we further define *active time*

[4]Some blog posts focus more on the payload and only report the fact where the payload is dropped, without describing the download infrastructure. In this case, we are not able to collect the baiting techniques.

Table 5.5: Top 10 campaign groups that drop the most suspicious binaries. The group name is identified from the information provided by the references. Active time corresponds to the time span of suspicious downloads, and discovery time corresponds to the date when the references are posted. hrd: high-risk download, hrb: high-risk binaries.

Rank	Name	# of hrb.	# of hrd.	Active time	Discovery time	Persuasion technique
1	Pinball Corp	874	25,578	05/19/2010 - 10/06/2013	03/29/2011	Advertise missing codec.
2	InstallCore	281	502	03/19/2012 - 02/07/2014	06/24/2013 - 10/18/2013	Advertise missing codec.
3	YieldManager	167	29,756	10/06/2012 - 06/28/2014	12/20/2012 - 07/03/2013	Advertise Flash Player.
4	Somoto installer	137	4,703	04/16/2012 - 06/27/2014	05/15/2013 - 07/26/2013	Advertise missing codec.
5	EzDownloaderpro	29	119	07/06/2012 - 09/30/2013	10/22/2013	Download portal.
6	OpenCandy	18	497	07/29/2013 - 06/15/2014	05/02/2014	Download portal.
7	Clikug	16	450	11/20/2013 - 06/30/2014	12/04/2013- 02/13/2014	Advertise Skype credit generator.
8	Awimba LLC	14	144	05/28/2012 - 05/23/2013	06/19/2013	Flash update notification.
9	BubbleDock	10	58	08/08/2013 - 01/05/2014	11/11/2013	Advertise missing codec.
10	–	6	88	02/04/2013 - 10/27/2013	03/02/2013	Flash update notification.

to be the time span when high-risk downloads are observed in measurement data, and *discovery time* as the time period that the corresponding blogs are posted.

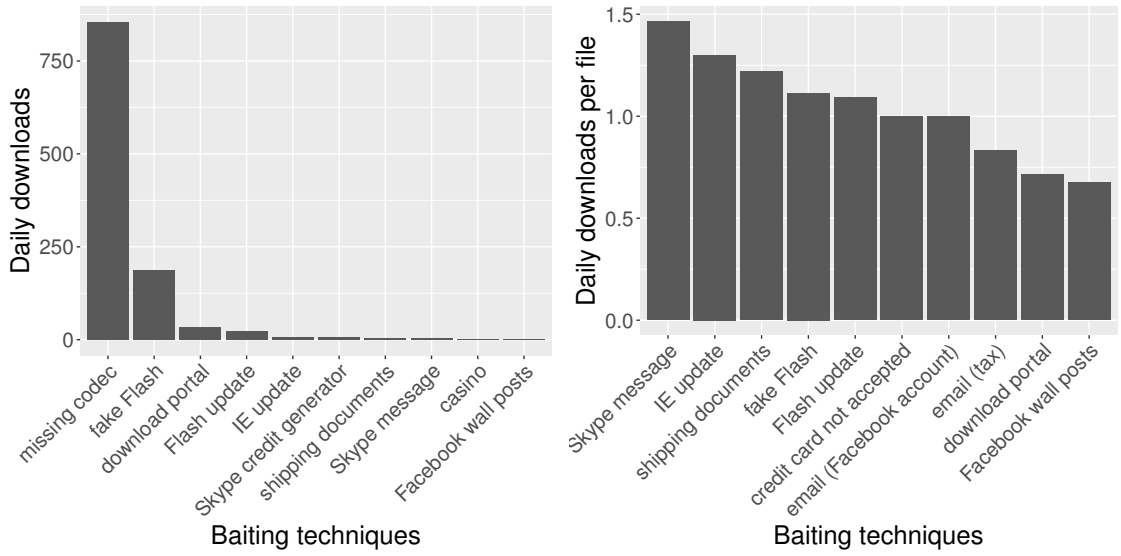
26 campaign groups start from social engineering, which is the most commonly used technique in our data. Unlike drive-by downloads, social engineering does not involve exploitation and intrusion, and malware delivery is initiated by user actions. Table 5.5 shows that all the top campaign groups entice users simply through social engineering, which suggests that social engineering is prevalent and effective.

To dig deeper into the social engineering methods, we group the high-risk binary payloads according to the persuasion technique utilized, and we calculate the daily rate of high-risk downloads produced by each technique. Figure 5.4 shows the daily high-risk downloads for the top 10 baiting techniques. For a fair comparison among the social engineering methods, we remove drive-by downloads that deliver payloads directly using exploits. Figure 5.4a shows the total daily downloads of high-risk binaries. Media player related advertisement involves providing deceptive information that entices users to download something in order to watch an online video. For example, group 1 in Table 5.5 provides a fake video, and asks users to download the missing plug-in in order to play it. Most high-risk download events occur in response to media player advertising. In fact, this ruse can be substantially more persuasive than other techniques: groups 1 and 3 from Table 5.5 generate an order of magnitude more downloads than other groups.

Figure 5.4b shows the daily downloads per file, for each persuasion technique, which reflects the delivery ability from the perspective of customers. The most prolific campaign group sends messages containing the download URL to the contacts

of compromised Skype accounts. The effectiveness of this technique in distributing high-risk files suggests that users are less alert to the message received from their friends and colleagues. However, we do not observe a large volume of downloads for these campaigns, which are limited by the number of compromised accounts. Another effective baiting trick is to present a fake software update notification, and the common target applications are Flash Player and IE. To enhance credibility, the page hosting the supposed plug-in can even mimic the Flash Player installation process; instead, a dropper is usually installed on the user's machine. The least attractive baiting content is anti-virus scanners. These anti-virus scanners are usually labeled as PUP (potentially unwanted program) since they do not behave as they claimed and ask users to buy the license. The low rate of downloads suggests that users are less susceptible to the rogue anti-virus advertisement, which is also supported by prior work [97]. Interestingly, an application that warns about an impending zombie invasion produces a higher download rate than the rogue anti-virus scanners.

We do not include drive-by downloads in our analysis, as the total number of downloads is too small to accurately estimate the influence and effectiveness of the technique. This suggests that attackers likely change the domains used for exploitation and intrusion frequently so that the IOCs collected from blog posts cannot be used to capture the download behaviors. The binary download behaviors from three exploit kits are recorded in WINE. However, the total number of high-risk downloads is less than 10 for each of groups, which suggests that the domains can change rapidly for the exploitation and intrusion campaigns.



(a) Total daily downloads.

(b) Daily downloads per file.

Figure 5.4: Daily downloads of high-risk binaries. Total daily download (a) reflects the frequency that a specific baiting technique is used in high-risk download; while the average of individual file (b) reflects the effectiveness of binary delivery of a single file.

Table 5.5 shows the dates when campaigns from the top tier-1 groups were active. This shows that these campaigns are long lasting, usually exceeding 1 year in duration. Because the download event data-set was collected until June 2014, we cannot track the full length of some campaigns persisting past that date. The campaigns for half of the top tier-1 groups remain active at that date, suggesting that the real duration may be significantly longer. These long durations suggest that we currently lack the technical means to stop the delivery campaigns involved in social engineering. Interestingly, for 7 out of the 10 campaigns, the articles mentioning them are published well within the activity period of the campaigns, suggesting that the campaigns go on after they are discovered by security analysts. This suggests that the baiting techniques remain effective in luring users, and the benefits of switching to another baiting technique do not outweigh the costs.

Implications. Comparing the effectiveness of the persuasion techniques employed by malicious actors suggests areas where public awareness and education are most likely to have an impact on malware delivery. In addition, the campaigns from tier-1 suppliers usually go on for at least 1 year—even after they have been discovered. This reveals the limitations of existing security tools in preventing campaigns that rely upon social engineering. Because most tier-1 suppliers employing social engineering do not exhibiting any exploitation or intrusion intentions, and because the downloaded payload may not expose obvious malicious behavior, it is difficult to detect and determine whether to block such campaigns in the gray area.

5.3.2 Underground business relationships

To study the business relationship between tier-1 suppliers and their customers, we identify the ownership of payloads from the certificate and the anti-virus signatures. If the payload is signed, which is often the case for PUPs, then we consider the publisher in the certificate as the owner of the payload. If the payload is not signed, then we identify the malware family using the AVclass tool [123]. As a result, we identify 289 payload owners. The payload owners maintain a direct relationship with tier-1 suppliers if the payload is dropped from the domains of tier-1 suppliers. Otherwise, we consider the relationship as indirect if the binary is bundled by the reseller without establishing direct relationship with tier-1 suppliers. 87.5% of payload owners maintain a direct relationship with the tier-1 suppliers, which suggests that the majority of downloads are directly associated with the baiting techniques.[5] In addition, one actor might have different ways to deliver its payloads. For example, Somoto better installer has its own advertising network [124], but it is also delivered through OpenCandy in our data set.

Some business relationships might be hidden from the measurement data because the real creator of a binary may not be the parent process. For example, if the application creates a service, then `svchost.exe` might be recorded as the parent of any subsequent behaviors from the service. Therefore, the parent-child

[5]The actual percentage of direct should be higher, because the payload owner and tier-1 supplier can be the same. In this case, the customer of the payload dropper also has a direct relationship with tier-1 suppliers.

relationship might be incomplete from the measurement data. However, blog posts provide another complementary data source to bridge the missing relations because security analysts are able to differentiate and report the real causal relationships. By integrating the results from security articles, we are able to formulate hypotheses about the business relationship between campaign groups. For example, tpstneuknash[dot]com is reported as the C&C server for ZeroAccess, and it delivers TrojanSpy:Win32/Bafi.E in WINE. Therefore, it is likely that the victim of ZeroAccess receives the command to download Win32/Bafi from tpstneuknash[dot]com. Moreover, such business relationship might be previously unknown, since Win32/Bafi is absent from malware families that are dropped by ZeroAccess [125]. Moreover, Symantec security response lists two malware families dropped by ZeroAccess, and the payload malware we discovered is not on the list, which implies that the business relationship might be previously unknown.

Implications. Although not all campaign groups have infrastructure for baiting, most of the groups are the direct customer of tier-1 suppliers. In addition, the tier-1 suppliers can also be the customer of another group. This suggests that the business relationship is prevalent and the actors in the underground market can be highly connected. Moreover, there might be more business relationships hidden from measurement data due to the fact that it is hard to find the real parent of a download event.

5.3.3 Lifecycles of campaigns and infrastructures

As noted in Chapter 5.3.1, most of campaign groups use social engineering for binary delivery. This suggests that most of the domains used in exploitation and intrusion may change frequently, and consequently it is difficult to use IOCs for detecting long-running campaigns. Figure 5.5 shows the distribution of IOC occurrence in technical blogs for file hash, URL and IP address.[6] Owing to malware polymorphism and domain generating algorithms, most domains and malware will not be identical during the same campaign and therefore security analysts are likely to report different IOCs from the same campaign. In addition, IP addresses are more likely to be discussed in different articles, because they are more difficult to be changed than URLs and file hashes. This suggests that attackers are likely to change the domains they used in order to evade detection, and therefore IOCs are usually short-lived and may never be used again.

While their utility for real-time detection is limited, IOCs can be useful in security forensics to identify long-term campaign groups. Domains can be changed frequently, but the attackers are still likely to reuse part of the infrastructure, which makes it possible to connect the dots and observe the long-term evolution of a campaign. Figure 5.6 shows examples of spam campaigns related to the Cridex family collected from 6 different articles. All the spam emails redirect users to the BlackHole exploit kit, and drop one version of Cridex malware. The malware communicates with different command and control servers, and the IP of the C&C server keeps

[6] We use fully qualified domain for counting URL, which ignores the associated path name.

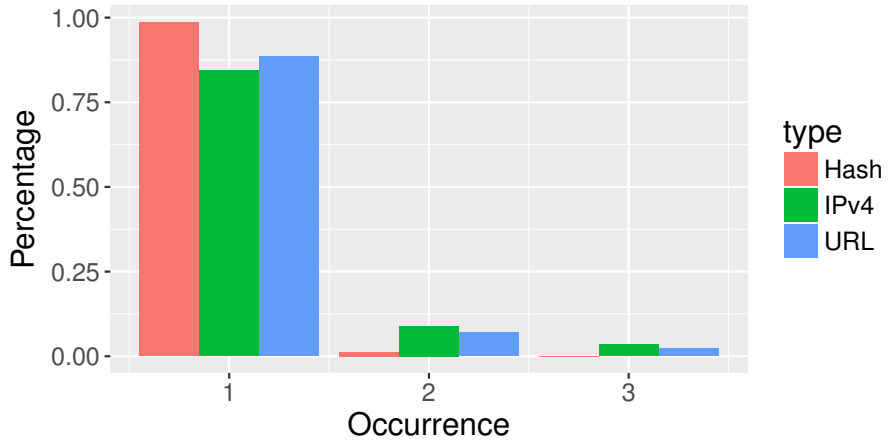


Figure 5.5: Percentage of the IOC occurrence in technical blogs for file hash, URL and IP address. The distribution of IOC occurrence is highly skewed, and therefore we only show the percentage of the occurrence less than 3.

changing over time, which is likely because the server's IP gets blacklisted. But in some cases, the attacker reuses old C&C IPs. For example, as shown in Figure 5.6, attacker reused 210.56.23.100 on November 19 (4 months later), and reused 95.142.167.193 on November 26 (3 months later). One possible explanation for this behavior is that IPs are removed from blacklists after several months so that the attackers may add them back to the campaign infrastructure. Kührer *et al.* studied 15 blacklists from which they estimated the average blacklisting time [126]. Although the average listing time varies for different blacklists, the malicious domain will be delisted after 100 days on average. This suggests that attackers might actively check if the domain and IP are removed from blacklist, and utilize the old delisted domain and IP.

Implications. Our findings provide important lessons for the most effective uses of threat intelligence. The premise of threat intelligence is that sharing the technical

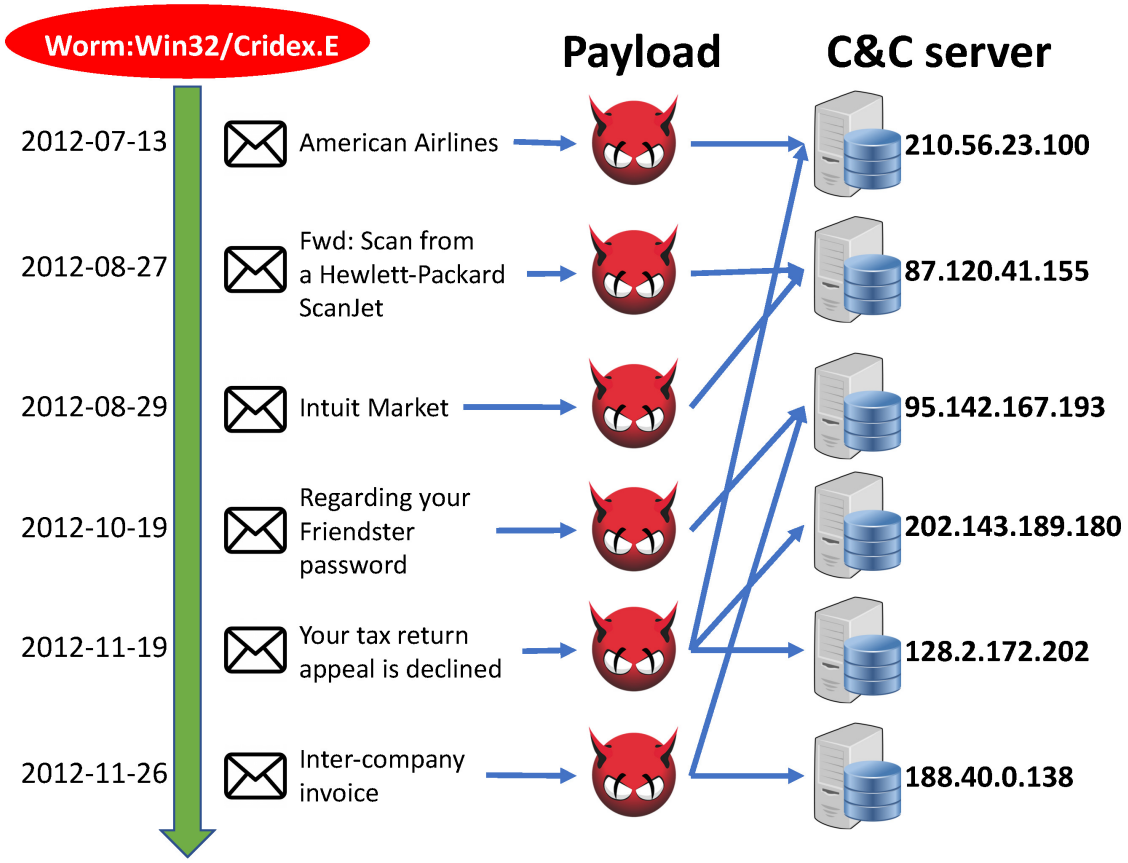


Figure 5.6: Spam campaigns that drop Worm:Win32/Cridex.E.

details of breaches and attacks makes everybody safer because it makes it harder for attackers to reuse attack methods and artifacts, thus increasing their *work factor* [21]. Our results suggest that the detection of IOCs based on URLs and IPs is not an effective mitigation, as the network identities of servers involved in malware delivery campaigns already change frequently. While several domain generation algorithms have been reverse engineered in prior work [102], future IP and domain name changes within one campaign cannot, in general, be predicted based only on the threat intelligence. Even though the URLs and IP addresses of malicious servers are short-lived, campaigns using this infrastructure can be long-lasting and they may

continue after some components are taken down. This suggests that the connections established among IOCs and the qualitative insights about the campaigns are the most useful outcomes of threat intelligence.

Chapter 6: Discussion

6.1 Automatic feature engineering

The fundamental reason why we can extract salient malware features from scientific papers is that researchers tend to show the useful features and ignore those that do not work. Additionally, as the publication process focuses on novelty, papers often show examples that are absent in 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 papers, which facilitates feature mining process.

In some cases, the relationship between malware and features is not stated explicitly. For example, researchers 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, like our system in Chapter 5, which allows 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.

Since knowledge from scientific literature cannot cover every aspects of security, we propose another data-driven feature engineering approach that learns im-

portant features directly from data. The key difference between ReasonSmith and the state-of-the-art technique for model explanation is that we model the feature importance as a random variable, and the analysis is feature-oriented. In security applications like malware detection, features usually have their own semantic meaning. By identifying most important features, researchers can easily verify if the features are indicators for new behaviors or artifacts learned from training data. However, the state-of-the-art techniques are sample-oriented, which focus on interpreting individual features. The explanation only shows the local representation of the system, and it is still unclear how the model works for the whole data set. In addition, the sample-oriented technique is able to tell if a feature is important by giving one example, but not able to tell if a feature is not important.

In this dissertation, we only test the case when local interpretation can be modeled by single Gaussian distribution, which is a strong assumption. To better estimate the distribution, we should evaluate the performance for Gaussian mixture model, where k is greater than 1. Another approach is to use non-parametric estimation, for example, neural representation. The idea is similar to GAN, which models the input distribution by a feed-forward generator [127]. The generator returns a fake local interpretation and can be trained together with a discriminator that tries to distinguish the real local interpretation and the generated one. Then we can further apply Monte Carlo method to generator to estimate the probability that interpretation w is non-zero.

We propose two different methods to engineer feature hypotheses for malware detection from security literature and malware data set, which helps analysts to effi-

ciently identify important features and data artifacts. However, in this dissertation, we did not answer how to select features in practice. Because we show that data artifacts are common in security in Chapter 4, it is difficult to develop some criteria to select features automatically. Whether the features are meaningful to security analysts should be a primary metric to evaluate the feature quality. As a future work, we could study how security analysts think about the important features through a user study, and further develop a framework for feature selection.

6.2 Malware delivery campaign

As social engineering is the prevalent strategy to start the delivery campaign, human factors play an important role in the security arms race. The state-of-the-art detection systems focus more on blocking network intrusion and removing malicious programs, but they usually ignore whether the behaviors are consistent with what is as expected. In many cases, especially for PUP delivery, the message provided in the campaign is inconsistent with the behavior behind the scenes. This provides an opportunity for stopping these campaigns by blocking the misleading advertisement, regardless of whether the downloaded payload is malicious or not. For example, a benign download should be blocked when it is bundled with a fake Flash update notification. In this case, understanding the semantics of advertisement is the key to preventing the deception.

In this dissertation, we compare the IOCs collected from blog posts to the real-world measurement data. We show that threat intelligence is useful in secu-

rity forensics, because it provides semantics to the measurement data and make it possible for security researchers to explain the measurement data. However, measuring the effectiveness of IOCs in detecting malicious campaigns is equally important and has not been studied thoroughly. In fact, that few IOCs are shared in both measurement data and security articles suggests that the infrastructure in the campaign is likely to change frequently. Therefore, how to make the most use of threat intelligence in real-time campaign detection should be an important next step.

Automatically extracting the semantics of security threats from natural language documents is a promising direction for the analysis of long-lasting campaigns. In particular, determining semantics of the *relationships among indicators of compromise* is key to reconstructing chains of actions and distinguish different actors in the campaign. Prior techniques for extracting IOCs from technical documents [57, 128, 129] are unable to reconstruct campaigns automatically (in [57], the authors established some links between a C&C infrastructure and the exploits utilized through manual analysis). In contrast, ChainSmith automatically reconstructs the semantics of entire delivery campaigns. One benefit of semantic relationships is that they allow us to identify different actors in campaigns. To stimulate further research on cyber threat intelligence, we released a Web application (`http://ioc-chainsmith.org`) for providing data extracted automatically from natural language reports in a timely manner.

Chapter 7: Conclusion

In this dissertation, we study automatic feature engineering for malware detection using both qualitative data (e.g. scientific literature and security blogs) and quantitative data (e.g. malware execution traces).

Knowledge-base feature engineering. In Chapter 3, we describe 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`.

Data-driven feature engineering. In Chapter 4, we describe ReasonSmith system that evaluates the global importance of all features for deep neural network based models. The system models feature importance as random variable that can be estimated empirically from a given data set. We evaluate our system using both Windows and Android data sets with more than 100k samples in each. By removing the features that are less important, the DNN-based malware detectors can still have high true positive rate with low false positive rate, which shows that ReasonSmith provides better feature score that captures the global influence. In addition, we further distinguish data biases and artifacts by applying ReasonSmith to data in different time spans. We find that many important features learned by neural networks are artifacts that potentially degrade the performance of malware detector in practice.

Feature engineering for malware campaigns. In Chapter 5, we describe ChainSmith, a system that automatically extracts IOCs from technical articles and industry reports, and classifies them into different campaign stages, i.e. baiting, exploitation, installation and command and control. This provides a semantic layer on top of IOCs that captures the role of indicators in a malicious campaign. ChainSmith achieves 91.9% precision and 97.8% recall in extracting IOCs from natural

language documents and is able to determine the campaign stage for 86.2% of IOCs with 78.2% precision and 80.7% recall. In addition, we identify 8,902 campaign groups from IOCs based on the stages and article post time. This makes it possible to combine threat intelligence with field-gathered data. The data is released at `http://ioc-chainsmith.org`.

We use a data set of download events to measure the effectiveness of different persuasion strategies employed in the baiting stage. We find that most campaigns deliver payloads through social engineering, without exhibiting any intrusion intentions. Media player advertising is most persuasive and generates the largest number of high-risk downloads, but "friends recommendations" and fake update notifications are most effective, as they generate the most daily downloads per file. In addition, we find that most of the customers in the malware delivery ecosystem have direct relationships to the suppliers that operate baiting services. Finally, we give use cases for leveraging IOCs in security forensics, which sheds new light on the best uses of threat intelligence.

Bibliography

[1] Yajin Zhou and Xuxian 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.

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

[3] Peder Olesen Larsen and Markus 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.

[4] Masayo Ota, Huy Vo, Claudio Silva, and Juliana 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.

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

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

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[8] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. *DARPA Information Survivability Conference and Exposition,*, pages 12–26, 2000.

[9] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.

[10] Matthew V Mahoney and Philip K Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. Technical report, 2001.

[11] Matthew V Mahoney and Philip K Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 220–237. Springer, 2003.

[12] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[13] George E Dahl, Jack W Stokes, Li Deng, and Dong 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] Meenu Ganesh, Priyanka Pednekar, Pooja Prabhuswamy, Divyashri Sreedharan Nair, Younghee Park, and Hyeran Jeon. Cnn-based android malware detection. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 60–65. IEEE, 2017.

[15] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582. IEEE, 2016.

[16] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 473–487. IEEE, 2018.

[17] Ben Athiwaratkun and Jack W Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486. IEEE, 2017.

[18] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[19] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[20] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[21] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[22] International Data Corporation. Worldwide threat intelligence security services 2014–2018 forecast: "iterative intelligence" — threat intelligence comes of age. IDC Market Forecast, Mar 2014.

[23] The openioc framework. `http://www.openioc.org/`, 2017.

[24] A structured language for cyber observables. `https://cyboxproject.github.io`, 2017.

[25] STIX: Structured threat information expression. `http://stixproject.github.io`, 2017.

[26] Facebook. ThreatExchange. `https://threatexchange.fb.com/`, 2017.

[27] Cyber security / information assurance (CS/IA) program. `http://dibnet.dod.mil/`, 2017.

[28] ThreatConnect. `https://www.threatconnect.com/`, 2017.

[29] Hail a TAXII: a repository of open source cyber threat intellegence feeds in STIX format. `https://http://hailataxii.com`, 2017.

[30] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraș. The dropper effect: Insights into malware distribution with downloader graph analytics. In *CCS*, 2015.

[31] Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, Elie Bursztein, and Damon McCoy. Investigating commercial pay-per-install and the distribution of unwanted software. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 721–739, Austin, TX, August 2016. USENIX Association.

[32] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring PUP prevalence and PUP distribution through Pay-Per-Install services. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 739–756, Austin, TX, August 2016. USENIX Association.

[33] BumJun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitraș. Catching worms, trojan horses and PUPs: Unsupervised detection of silent delivery campaigns. In *Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, Feb 2017.

[34] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

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

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

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

[38] Domenico M Pisanelli. *Ontologies in medicine*, volume 102. IOS Press, 2004.

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

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

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

[42] Michael D Gordon and Susan Dumais. Using latent semantic indexing for literature based discovery. 1998.

[43] Scott Spangler, Angela D Wilkins, Benjamin J Bachman, Meena Nagarajan, Tajhal Dayaram, Peter Haas, Sam Regenbogen, Curtis R Pickering, Austin Comer, Jeffrey 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.

[44] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[45] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[46] Luca Invernizzi, Sung-Ju Lee, Stanislav Miskovic, Marco Mellia, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, and Giovanni Vigna. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, 2014.

[47] Babak Rahbarinia, Roberto Perdisci, and Manos Antonakakis. Segugio: Efficient behavior-based tracking of malware-control domains in large ISP networks. In *DSN*, 2015.

[48] Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahamad. Webwitness: Investigating, categorizing, and mitigating malware download paths. In *USENIX Security Symposium*, 2015.

[49] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing compromise: the emergence of exploit-as-a-service. In *ACM Conference on Computer and Communications Security*, Raleigh, NC, Oct 2012.

[50] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[51] Kurt Thomas, Danny Huang, David Wang, Elie Bursztein, Chris Grier, Thomas J Holt, Christopher Kruegel, Damon McCoy, Stefan Savage, and Giovanni Vigna. Framing dependencies introduced by underground commoditization. In *WEIS*, 2015.

[52] Xiaojing Liao, Chang Liu, Damon McCoy, Elaine Shi, Shuang Hao, and Raheem Beyah. Characterizing long-tail seo spam on cloud web hosting services. In *Proceedings of the 25th International Conference on World Wide Web*, pages 321–332. International World Wide Web Conferences Steering Committee, 2016.

[53] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhongyu Pei, Hao Yang, Jianjun Chen, Haixin Duan, Kun Du, Eihal Alowaisheq, Sumayah Alrwais, et al. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 707–723. IEEE, 2016.

[54] Birhanu Eshete and VN Venkatakrishnan. Webwinnow: Leveraging exploit kit workflows to detect malicious urls. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 305–312. ACM, 2014.

[55] Birhanu Eshete, Abeer Alhuzali, Maliheh Monshizadeh, Phillip A Porras, Venkat N Venkatakrishnan, and Vinod Yegneswaran. Ekhunter: A counter-offensive toolkit for exploit kit infiltration. In *NDSS*, 2015.

[56] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778. ACM, 2016.

[57] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem 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.

[58] Karim O Elish, Xiaokui Shu, Danfeng Daphne Yao, Barbara G Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Computers and Security*, 49(C):255–273, 2015.

[59] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

[60] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[61] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[62] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[63] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[64] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *ACL (2)*, pages 302–308. Citeseer, 2014.

[65] Top level domains. `http://data.iana.org/TLD/tlds-alpha-by-domain.txt`, 2017.

[66] Reserved ip addresses. `https://en.wikipedia.org/wiki/Reserved_IP_addresses`, 2017.

[67] Malware naming convention. `https://www.microsoft.com/en-us/security/portal/mmpc/shared/malwarenaming.aspx`, 2017.

[68] Overview of exploit kit. `http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html`, 2010.

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

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

[71] Jennifer Chu-Carroll, Eric W Brown, Adam Lally, and J William Murdock. Identifying implicit relationships. *IBM Journal of Research and Development*, 56(3.4):12–1, 2012.

[72] Stephan Neuhaus and Thomas 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.

[73] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: towards automating risk assessment of mobile applications. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 527–542. USENIX Association, 2013.

[74] Bo Sun, Akinori Fujino, and Tatsuya Mori. Poster: Toward automating the generation of malware analysis reports using the sandbox logs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1814–1816. ACM, 2016.

[75] Anupam Panwar. *iGen: Toward Automatic Generation and Analysis of Indicators of Compromise (IOCs) using Convolutional Neural Network*. PhD thesis, Arizona State University, 2017.

[76] Ziyun Zhu and Tudor Dumitras. Chainsmith: Automatically learning the semantics of malicious campaigns by mining threat intelligence reports. In *3rd IEEE European Symposium on Security and Privacy*. IEEE, 2018.

[77] Ghaith Husari, Ehab Al-Shaer, Mohiuddin Ahmed, Bill Chu, and Xi Niu. Ttpdrill: Automatic and accurate extraction of threat actions from unstructured text of cti sources. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 103–115. ACM, 2017.

[78] George Casella and Roger L Berger. *Statistical inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[79] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alex G Hauptmann. Devnet: A deep event network for multimedia event detection and evidence recounting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2568–2577, 2015.

[80] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[81] Ari S Morcos, David GT Barrett, Neil C Rabinowitz, and Matthew Botvinick. On the importance of single directions for generalization. *arXiv preprint arXiv:1803.06959*, 2018.

[82] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. Tesseract: Eliminating experimental bias in malware classification across space and time. *arXiv preprint arXiv:1807.07838*, 2018.

[83] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638. ACM, 2011.

[84] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[85] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo 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.

[86] Hao Peng, Christopher S. Gates, Bhaskar Pratim Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In Ting Yu, George Danezis, and Virgil 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.

[87] Yousra Aafer, Wenliang Du, and Heng 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.

[88] Cuckoo sandbox repository. `https://github.com/cuckoosandbox`.

[89] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[90] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[91] Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. Imds: Intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1043–1047. ACM, 2007.

[92] Chandrasekar Ravi and R Manoharan. Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications*, 43(17):12–16, 2012.

[93] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE, 2015.

[94] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

[95] Xianchao Zhang, Zhaoxing Li, Shaoping Zhu, and Wenxin Liang. Detecting spam and promoting campaigns in twitter. *ACM Transactions on the Web (TWEB)*, 10(1):4, 2016.

[96] Cheng Cao and James Caverlee. Detecting spam urls in social media via behavioral analysis. In *European Conference on Information Retrieval*, pages 703–714. Springer, 2015.

[97] Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahamad. Towards measuring and mitigating social engineering software download attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 773–789, Austin, TX, 2016. USENIX Association.

[98] Enrico Blanzieri and Anton Bryl. A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review*, 29(1):63–92, 2008.

[99] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 469–485, Vancouver, BC, 2017. USENIX Association.

[100] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: Abuse in Authenticode code signing. In *CCS*, 2015.

[101] Jialong Zhang, Sabyasachi Saha, Guofei Gu, Sung-Ju Lee, and Marco Mellia. Systematic mining of associated server herds for malware campaign discovery. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 630–641. IEEE, 2015.

[102] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A comprehensive measurement study of domain generating malware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 263–278, Austin, TX, 2016. USENIX Association.

[103] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1:80, 2011.

[104] Marie-Catherine De Marneffe and Christopher 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.

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

[106] About google scholar. `https://scholar.google.com/scholar/about.html`.

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

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

[109] Michael Spreitzenbarth, Felix C. Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In Sung Y. Shin and José Carlos 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.

[110] Virus total. `www.virustotal.com`, 2017.

[111] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay 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.

[112] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

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

[114] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[115] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng 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.

[116] Harunobu Agematsu, Junya Kani, Kohei Nasaka, Hideaki Kawabata, Takamasa Isohara, Keisuke Takemori, and Masakatsu 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.

[117] Shunya Sakamoto, Kenji Okuda, Ryo Nakatsuka, and Toshihiro 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.

[118] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–141. Springer, 2016.

[119] STIX support survey. OASIS Cyber Threat Intelligence (CTI) TC Wiki. `https://wiki.oasis-open.org/cti/Products`.

[120] Webroot: Your discover card services blockaded themed emails serve client-side exploits and malware. `https://www.webroot.com/blog/2012/11/08/your-discover-card-services-blockaded-themed-emails-serve-client-side-exploits-and-malware/`, 2017.

[121] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[122] Xiang Liu, Torsten Suel, and Nasir Memon. A robust model for paper reviewer assignment. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 25–32. ACM, 2014.

[123] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[124] Somoto baiting technique. `https://www.webroot.com/blog/2013/07/03/deceptive-ads-targeting-german-users-lead-to-the-w32somotobetterinstaller-potentially-unwanted-application-pua/`, 2013.

[125] Symantec security response: Zeroaccess. `https://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99&tabid=2`, 2013.

[126] Marc Kührer, Christian Rossow, and Thorsten Holz. Paint it black: Evaluating the effectiveness of malware blacklists. In *RAID*, 2014.

[127] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[128] Ioc parser. `https://github.com/armbues/ioc_parser`, 2017.

[129] Alienvault otx. `https://otx.alienvault.com`, 2017.