EFFICIENT AND EXPLAINABLE DETECTION OF MOBILE MALWARE WITH MACHINE LEARNING

VON DER
CARL-FRIEDRICH-GAUSS-FAKULTÄT
DER TECHNISCHEN UNIVERSITÄT CAROLO-WILHELMINA
ZU BRAUNSCHWEIG

ZUR ERLANGUNG DES GRADES EINES
DOKTORINGENIEURS (DR.-ING.)

- GENEHMIGTE -

DISSERTATION

VON
DANIEL CHRISTOPHER ARP
GEBOREN AM 19.03.1986
IN BERLIN

Eingereicht am:	19.03.2019
Disputation am:	13.06.2019
1. Referent:	Prof. Dr. Konrad Rieck
2. Referent:	Prof. Dr. Lorenzo Cavallaro

2019

promotionsausschuss:

1. Referent:	Prof. Dr. Konrad Rieck
	Technische Universität Braunschweig
2. Referent:	Prof. Dr. Lorenzo Cavallaro
	King's College London
Vorsitz:	Prof. Dr. Martin Johns
	Technische Universität Braunschweig

Braunschweig, Juni 2019

ABSTRACT

In recent years, mobile devices shipped with Google's Android operating system have become ubiquitous. Due to their popularity and the high concentration of sensitive user data on these devices, however, they have also become a profitable target of malware authors. As a result, thousands of new malware instances targeting Android are found almost every day. Unfortunately, common signature-based methods often fail to detect these applications, as these methods cannot keep pace with the rapid development of new malware. Consequently, there is an urgent need for new malware detection methods to tackle this growing threat.

In this thesis, we address the problem by combining concepts of static analysis and machine learning, such that mobile malware can be detected directly on the mobile device with low run-time overhead. To this end, we first discuss our analysis results of a sophisticated malware that uses an ultrasonic side channel to spy on unwitting smartphone users. Based on the insights we gain throughout this thesis, we gradually develop a method that allows detecting Android malware in general. The resulting method performs a broad static analysis, gathering a large number of features associated with an application. These features are embedded in a joint vector space, where typical patterns indicative of malware can be automatically identified and used for explaining the decisions of our method. In addition to an evaluation of its overall detection and run-time performance, we also examine the interpretability of the underlying detection model and strengthen the classifier against realistic evasion attacks.

In a large set of experiments, we show that the method clearly outperforms several related approaches, including popular anti-virus scanners. In most experiments, our approach detects more than 90% of all malicious samples in the dataset at a low false positive rate of only 1%. Furthermore, even on older devices, it offers a good run-time performance, and can output a decision along with a proper explanation within a few seconds, despite the use of machine learning techniques directly on the mobile device.

Overall, we find that the application of machine learning techniques is a promising research direction to improve the security of mobile devices. While these techniques alone cannot defeat the threat of mobile malware, they at least raise the bar for malicious actors significantly, especially if combined with existing techniques.

ZUSAMMENFASSUNG

Die Verbreitung von Smartphones, insbesondere mit dem Android-Betriebssystem, hat in den vergangenen Jahren stark zugenommen. Aufgrund ihrer hohen Popularität haben sich diese Geräte jedoch zugleich auch zu einem lukrativen Ziel für Entwickler von Schadsoftware entwickelt, weshalb mittlerweile täglich neue Schadprogramme für Android gefunden werden.

Obwohl bereits verschiedene Lösungen existieren, die Schadprogramme auch auf mobilen Endgeräten identifizieren sollen, bieten diese in der Praxis häufig keinen ausreichenden Schutz. Dies liegt vor allem daran, dass diese Verfahren zumeist signaturbasiert arbeiten und somit schädliche Programme erst zuverlässig identifizieren können, sobald entsprechende Signaturen für deren Erkennung vorhanden sind. Durch die rasant steigende Zahl von Schadprogrammen für Android wird es allerdings auch für Antiviren-Hersteller immer schwieriger, die zur Erkennung notwendigen Signaturen rechtzeitig bereitzustellen. Daher ist die Entwicklung von neuen Verfahren nötig, um der wachsenden Bedrohung durch mobile Schadsoftware besser begegnen zu können.

In dieser Dissertation wird ein Verfahren vorgestellt und eingehend untersucht, das Techniken der statischen Code-Analyse mit Methoden des maschinellen Lernens kombiniert, um so eine zuverlässige Erkennung von mobiler Schadsoftware direkt auf dem Mobilgerät zu ermöglichen. Als Ausgangspunkt für die Entwicklung des Verfahrens dienen hierbei die Erkenntnisse aus einer Studie über eine neuartige Variante von Schadprogrammen, die einen Ultraschall-Seitenkanal nutzen, um Smartphone-Benutzer heimlich auszuspionieren. Basierend auf den Ergebnissen einer ausführlichen Analyse dieser Schadprogramme wird anschließend schrittweise ein Verfahren zur Erkennung von Schadsoftware entwickelt, das automatisch Erkennungsmuster für beliebige Varianten mobiler Schadsoftware herleiten kann. Die Methode analysiert hierfür mobile Anwendungen zunächst statisch und extrahiert dabei spezielle Merkmale, die eine Abbildung einer Applikation in einen hochdimensionalen Vektorraum ermöglichen. In diesem Vektorraum sind schließlich maschinelle Lernmethoden in der Lage, automatisch Muster zur Erkennung von Schadprogrammen zu finden. Die gefundenen Muster können dabei nicht nur zur Erkennung, sondern darüber hinaus auch zur Erklärung einer getroffenenen Entscheidung dienen.

Im Rahmen einer ausführlichen Evaluation wird nicht nur die Erkennungsleistung und die Laufzeit der vorgestellten Methode untersucht, sondern darüber hinaus das gelernte Erkennungsmodell im

Detail analysiert. Hierbei wird auch die Robustheit des Modells gegenüber gezielten Angriffe untersucht und verbessert. In einer Reihe von Experimenten kann gezeigt werden, dass mit dem vorgeschlagenen Verfahren bessere Ergebnisse erzielt werden können als mit vergleichbaren Methoden, sogar einschließlich einiger populärer Antivirenprogramme. In den meisten Experimenten kann die Methode Schadprogramme zuverlässig erkennen und erreicht Erkennungsraten von über 90% bei einer geringen Falsch-Positiv-Rate von 1%. Des Weiteren kann bei einer Auswertung mit verschiedenen Mobilgeräten gezeigt werden, dass der Ansatz meist nur wenige Sekunden benötigt, um für eine Applikation eine Entscheidung mitsamt einer passenden Erklärung zu liefern.

Zusammenfassend untermauern die Ergebnisse dieser Arbeit, dass die Verwendung maschineller Lernverfahren einen vielversprechenden Ansatz darstellt, um die Sicherheit mobiler Geräte zu verbessern. Während diese Techniken allein zwar auch die Bedrohung durch mobile Schadanwendungen nicht vollends beseitigen können, sind sie dennoch in der Lage, die erfolgreiche Infektion von Mobilgeräten deutlich zu erschweren.

PUBLICATIONS

This thesis contains ideas and results that have been published by the author in the following peer-reviewed papers and articles:

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2014.

[2] Daniel Arp, Erwin Quiring, Christian Wressnegger, and Konrad Rieck. "Privacy Threats through Ultrasonic Side Channels on Mobile Devices." In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017.

[3] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. "Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection." In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2019).

Besides, the thesis also contains several previously unpublished results. Furthermore, the author of this thesis contributed to the following publications, which also discuss some of the concepts presented in this thesis:

[1] Daniel Arp, Fabian Yamaguchi, and Konrad Rieck. "Torben: A Practical Side-Channel Attack for Deanonymizing Tor Communication." In: *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2015.

[2] Daniel Arp, Erwin Quiring, Tammo Krueger, Stanimir Dragiev, and Konrad Rieck. "Privacy-Enhanced Fraud Detection with Bloom filters." In: *Proc. of Int. Conference on Security and Privacy in Communication Networks (SECURECOMM)*. 2018.

[3] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Structural detection of android malware using embedded call graphs." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2013.

[4] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols." In: *Proc. of Int. Conference on Security and Privacy in Communication Networks (SECURECOMM)*. 2015.

[5] Hugo Gascon, Bernd Grobauer, Thomas Schreck, Lukas Rist, Daniel Arp, and Konrad Rieck. "Mining Attributed Graphs for Threat Intelligence." In: *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*. 2017.

[6] Henning Perl, Daniel Arp, Sergej Dechand, Fabian Yamaguchi, Sascha Fahl, Yasemin Acar, Konrad Rieck, and Matthew Smith. "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[7] Erwin Quiring, Daniel Arp, and Konrad Rieck. "Forgotten Siblings: Unifying Attacks on Machine Learning and Digital Watermarking." In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018.

[8] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. "Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques." In: *International Journal of Information Security (INT J INF SECUR)* 14.2 (2015).

[9] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. "A close look on *n*-grams in intrusion detection: anomaly detection vs. classification." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2013.

[10] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Comprehensive Analysis and Detection of Flash-Based Malware." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[11] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2014.

ACKNOWLEDGEMENTS

First and foremost, I wish to thank my supervisor Prof. Dr. Konrad Rieck for giving me the opportunity to work on my research in a very open and productive environment. Throughout the whole time as a PhD student, he inspired me with tons of ideas and supported me with his positive attitude — especially in challenging periods of my research.

Also, I would like to thank the other members of my PhD committee. In particular, Prof. Dr. Lorenzo Cavallaro for refereeing this thesis and Prof. Dr. Martin Johns for chairing the defense committee. I'm grateful that you have taken the time for doing this, despite your busy schedules. I cannot thank you enough for that!

I want to express my gratitude to all the wonderful people that I met during my PhD years, though I cannot list all of them here explicitly. I'm indebted for all the experiences I could gather during this time, and will always look back to it with fond memories. Special thanks go to Dr. Fabian Yamaguchi, who encouraged me to start a PhD, which turned out to be one of the best decisions I have ever made. Moreover, I would like to thank my long-time office mate Erwin Quiring for many inspiring discussions on security-related topics and, most importantly, making plans on how to save the world. ;) I wish to thank Dr. Christian Wressnegger, who always had an open ear and supported me with his vast knowledge on malware detection. Warm thanks also go to John Boswell for proof-reading large parts of this thesis.

Almost needless to say, I thank my parents, who have always been there for me and supported me in every possible way they could.

Finally, I want to thank my girlfriend Sina, as this thesis would definitely not have been possible without her encouragement and patience over the last few years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API	Application Programming Interface
APK	Android Package Kit
AUC	Area Under (ROC) Curve
CPU	Central Processing Unit
DTMF	Dual-Tone Multi-Frequency Signaling
FFT	Fast Fourier Transform
FPR	False Positive Rate
FSK	Frequency Shift Keying
GPS	Global Positioning System
GUI	Graphical User Interface
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
kNN	k-Nearest-Neighbor
LDA	Latent Dirichlet Allocation
OHA	Open Handset Alliance
OS	Operating System
PUP	Potentially Unwanted Program
ROC	Receiver Operating Characteristic
SDK	Software Development Kit
SMS	Short Message Service
SNR	Signal-to-Noise Ratio
SVM	Support Vector Machine
TAN	Transaction Authentication Number
TPR	True Positive Rate
URL	Uniform Resource Locator

1 INTRODUCTION

1.1 MOTIVATION

Within the last decade, the popularity of smartphones has grown significantly, such that billions of people nowadays own at least one of these devices [227]. Although there are different reasons for their popularity, two essential factors are their ease of use and the high flexibility these devices provide to their users. In particular, the functionality of smartphones is not limited to telephony, but can easily be extended through small applications, also known as *apps*. These applications offer a broad variety of different useful features to the user, ranging from email and online banking functionality to computationally demanding video games. A user can easily extend her device with a specific functionality within minutes by installing an application that provides the required feature.

Smartphones are widespread nowadays.

A considerable part of the success and prevalence of smartphones can be attributed to Google's Android Operating System (OS). Within a relatively short period of time, Android has become by far the most popular OS for smartphones, and runs on more than 85% of all mobile devices at the time of this writing [90]. As the majority of its code base is open source, not only Google itself, but also other companies release hardware with customized versions of the OS. This openness led to a large variety of devices in different price categories, making Android smartphones affordable for most people. Besides, Android also gives its users many possibilities to customize their devices, which resulted in a large and vivid development community around the operating system.

Android is the most popular OS for mobile devices...

But the growing popularity of Android smartphones also has a serious downside, since they have also started to attract the attention of malware authors and dubious advertising companies in recent years. Having mostly commercial interests, these actors try to distribute malicious applications among credulous smartphone users [93, 94, 223]. As a result, anti-virus vendors observe a significant increase in the number of malware for Android in the past few years [93]. For example, Figure 1 depicts the number of Android malware detected by the anti-virus vendor *GData* between 2012 and 2018 [133]. While the vendor already detected around 200,000 malicious applications in 2012, the number rapidly increased up to more than 3 million samples in 2016. Moreover, the anti-virus company even expects a total of roughly 4 million malware samples in 2018.

...and commonly targeted by malware authors.

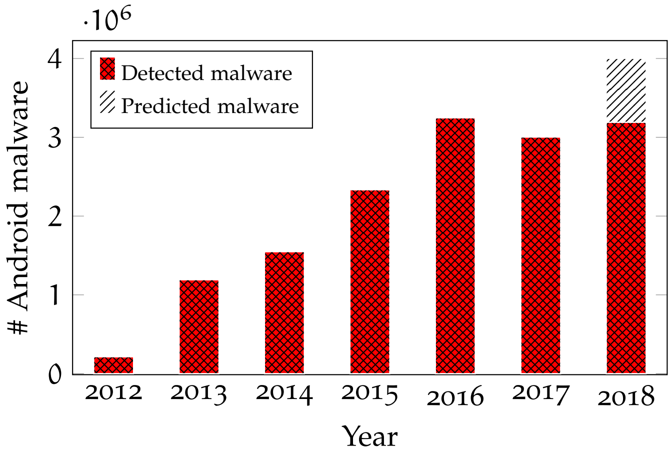


Figure 1: Based on the number of samples obtained until the third quarter of 2018, GData estimates that the number of malicious samples for Android will rise above 4,000,000 [133].

Android is targeted for various reasons...

The reasons why adversaries mainly target Android are manifold. First of all, Android runs on most mobile devices. Consequently, malware authors significantly increase their chances of infecting a large number of devices, if they develop their malware for this particular operating system. Secondly, in contrast to other platforms, Android allows its users to install applications not only from Google's official app store, *GooglePlay*, but also from other sources, such as third-party markets or websites. While this offers high flexibility and is therefore highly appreciated among Android's user base, it also increases the risk of accidentally installing malicious applications, as these alternative sources often do not provide security mechanisms comparable to those of GooglePlay. Thirdly, a large number of mobile devices still run deprecated versions of Android that exhibit serious security vulnerabilities. These vulnerabilities can, in turn, be exploited by malicious applications to escalate their privileges or misuse flaws in the security design of older versions of Android [93].

...and by different types of malware.

In addition to its increasing number, Android malware exhibits a broad spectrum of malicious behaviors. This includes, for instance, malware that sends SMS to expensive premium services owned by the malware authors, or several variants that extort money from the user [131, 202, 223]. The malware authors exploit the fact that modern mobile devices often concentrate large amounts of personal data, ranging from location data to sensitive login credentials [11, 72]. Also, these devices hold various sensors, like microphones and cameras, which can be misused by adversaries to spy on unwitting users without their knowledge or consent [e.g., 138, 142].

To protect users from the growing threat posed by mobile malware, different malware detection solutions exist. The most common one is most likely the installation of an anti-virus scanner on the device. Even though these tools can efficiently detect malware with low run-time overhead, they lack reliable detection in many cases. This is because these scanners often rely upon manually crafted detection patterns, also called *signatures*, which are unavailable for new malware [145, 175, 208]. Thus, anti-virus scanners frequently have problems detecting new malware samples, due to the sheer number of malicious samples arising almost every day. In addition, even small changes to a malicious application can be sufficient to circumvent the signatures of many of these scanners [39].

While various solutions to this problem exist...

A large body of research therefore studies new methods to improve the detection of Android malware. These methods can be roughly categorized into approaches that use static and dynamic analysis. For example, TaintDroid [72], DroidRanger [224], and DroidScope [217] are methods that can monitor the behavior of applications at run-time. Although very effective at identifying malicious activity, run-time monitoring suffers from a significant overhead and cannot be directly applied on mobile devices. By contrast, static analysis methods, such as Kirin [71], Stowaway [77], RiskRanker [97], usually induce only a short run-time burst. While these approaches are efficient and scalable, they also mainly build on manually crafted detection patterns, thus having similar drawbacks as traditional anti-virus solutions. In addition, current methods usually do not provide explanations for their decisions to the user. As these approaches, however, often suffer from false positives, it is essential that users have the possibility to understand the decisions made by these systems. This, in turn, allows them to decide whether or not to still trust an application, despite the assessment of the detector. Moreover, if the decisions of such a detection system can be reconstructed, it can also help researchers to improve the accuracy of the system further.

...they all suffer from miscellaneous problems.

In this thesis, we present novel insights on the Android malware landscape and propose new methods to tackle this growing threat effectively. In particular, we first discuss our analysis results on a sophisticated type of malware that spies on unwitting smartphone users. Based on our findings, we develop and gradually improve a method that allows the detection of Android malware in general. To this end, we leverage techniques from the field of machine learning, as these have already been successfully applied to similar application fields [e.g., 135, 169, 194]. In our case, we use them to build an efficient and effective detection method, which can derive proper detection patterns for Android malware automatically. Besides, our method runs directly on the mobile device, and provides suitable explanations for its decisions to the user. We outline the contributions of this thesis in the following section.

1.2 CONTRIBUTIONS

The findings we discuss throughout this thesis can be summarized into three main contributions:

1. As a motivating example, we present the results of a study, we conducted on the use of an ultrasonic side channel by mobile apps. Throughout this study, we have identified a large number of malicious apps that use this side channel to spy on unwitting smartphone users. In particular, while analyzing several commercial products that use this technique for different goals, we find that one of these companies has used it for illicit purposes. Our findings do not only show how the techniques utilized by mobile malware become more sophisticated over the years, but also how easily new technologies can be misused by malware authors or dubious advertising companies. Overall, we have been able to detect 234 samples containing the functionality, which enables them to listen in the background for ultrasonic signals without the knowledge of smartphone users. In response to our findings, Google has removed applications from the official store that endanger users' privacy by containing this functionality.

2. As our second contribution, we gradually develop a suitable method for Android malware detection. Specifically, we start by proposing a method to identify applications containing the previously mentioned ultrasonic tracking functionality. Afterward, we present a more generic approach that allows detecting Android malware in general. We call this method DREBIN and it enables us to derive the necessary signatures automatically. Throughout an extensive evaluation of the method, we show that DREBIN outperforms related approaches by a large margin, including several popular anti-virus scanners. Apart from its good detection capabilities, DREBIN also exhibits an excellent run-time performance and can therefore run directly on the mobile device. Furthermore, it outputs an explanation for its assessments of analyzed mobile applications. This property does not only help users understand the decisions of the detection system, but can also help to improve it further.

3. As the third contribution, we provide a detailed analysis of the underlying detection model. More precisely, we examine its interpretability, generalization capabilities, and robustness in realistic attack scenarios. To this end, we first compare its explanations for popular malware families with common knowledge about the behavior of these families. In a second step, we discuss possible attacks against the detection model and show how

the method can be improved, such that it provides more robustness against these kinds of attacks.

1.3 STRUCTURE OF THIS THESIS

The thesis is structured as follows. In Chapter 2, we provide the reader with basic background knowledge on the Android operating system, the malware landscape on Android, and the benefits and drawbacks of common approaches used to detect these malicious applications. Moreover, we introduce basic concepts of machine learning theory that are required to understand the explanations provided throughout this thesis. In Chapter 3, we present our findings on several applications that receive and transmit information using an ultrasonic side channel. We hereby focus on a specific malware that uses this technique to spy on smartphone users. In this context, we also propose a first method that allows detecting members of this malware family within a large number of applications. The results in this chapter have initially been presented at the *European Symposium on Security and Privacy (EuroS&P)* in Paris [9].

In Chapter 4, we propose a learning-based method for Android malware detection and provide a broad evaluation of its detection capabilities in Chapter 5. The interpretability and robustness of the underlying detection model is examined in Chapter 6, including realistic attacks against the detection method. We show how the learning algorithm can be improved to fend off these attacks. All of these chapters are mainly based on a paper that has been published at the *Network and Distributed System Security Symposium (NDSS)* [8], along with some previously unpublished results to account for recent developments in this field of research. Besides, we discuss an improved version of the learning method in Chapter 6. This improved version has been developed throughout a joint research project with the PRALabs of the University of Cagliari. The corresponding article is going to be published in the *IEEE Transactions on Dependable and Secure Computing (TDSC)* [57].

Chapter 7, finally, summarizes the obtained results and outlines future research directions.

2 BACKGROUND

In this chapter, we provide the reader with the background knowledge necessary to follow the explanations and descriptions in this thesis. In particular, the chapter is divided into two different parts, where the first discusses the Android operating system and the second part gives an introduction into the field of machine learning.

2.1 ANDROID

The mobile Operating System (OS) Android is developed by a consortium of different companies, the Open Handset Alliance (OHA), under the leadership of Google. From its first commercial release in 2008, Android has become the most popular mobile operating system within a couple of years and runs on roughly 85% of all mobile devices at the time of this writing [90].

Mainly due to its openness...

A main reason for its proliferation is the commitment of Google and the other members of the OHA to openness. As a result, Android is mainly build upon open source components and, for instance, based on a modified version of the Linux kernel [65, 70]. Moreover, Android can even be adapted for special requirements and therefore runs on a large variety of devices.

In contrast to Apple's operating system *iOS*, Android users cannot only install applications (also called *apps*) from Google's official store (*Google Play*) but also from alternative markets or websites. Moreover, Google encourages third-party developers to make applications for Android, for instance, by providing them with the necessary developments kits and API documentation. Consequently, there exists a large community developing applications for Android. Its high popularity and prevalence, however, also makes Android a worthwhile target of malware authors.

...Android has become the most popular mobile operating system.

In the following, we discuss basic concepts of Android which we refer to in this thesis.

2.1.1 *Applications*

Android applications are mostly written in Java and compiled into *Dalvik Bytecode* (or *dex code*). The dex code of an application is similar to Java bytecode but specially optimized to run efficiently on mobile devices [see, 65, 70, 73]. Besides, Android allows developers to implement parts of their applications in native code, using programming languages like C and C++. This feature is mostly intended for computationally intensive applications, such as video games or multimedia applications. Unfortunately, it can be misused by malware authors to hide their malicious code outside the Dalvik bytecode, as we will also discuss in this section.

An app's manifest file contains crucial information,...

The application code and other Android-specific files are packed together in an Android Package Kit (APK) file, which can be installed on the device. An APK file is a zip-like archive that must include certain files and directories. Most importantly, each APK file has to contain the *classes.dex* file (i.e., the Dalvik bytecode) and the *AndroidManifest.xml*. The manifest file of an application holds information mainly required during the installation process, including the package name of the app and the *components* it consists of. This information is used to register the application and its components with the system at install time [70].

An Android application usually consists of several loosely coupled components, which, in turn, belong to one of four different component types:

- *Activities.* These components are individual screens with a user interface and therefore used by all Android applications that provide a graphical user interface (GUI).
- *Services.* This component type runs in the background without a user interface. It is usually employed for long-running tasks, such as downloading files or playing music. Unfortunately, services are commonly misused by malware to perform malfeasant actions on the device without the user's knowledge.

...such as its components...

- *Content Providers.* This type of component provides an interface to application data and is mainly used to share data between applications. For instance, the user's contacts can be accessed through a content provider by an app having the respective permission.
- *Broadcast Receivers.* These components allow an application to react to system-wide events, e.g., when the screen has been turned off or the battery is low on charge. Like in the case of service components, malware uses this functionality to silently start its malicious components.

...,intent filters,...

INTENTS Android application components can communicate with each other using certain *Intent* objects. These are message objects that can be passed between components to exchange data or trigger certain tasks. For instance, they are commonly used to start activities and service components. Moreover, intents also allow the communication between different applications, which are typically isolated from each other due to Android's security policy.

There exist two types of intents, i.e., *explicit intents* and *implicit intents*. *Explicit intents* are sent to a specific application or component, while *implicit intents* are forwarded to all applications that have registered the respective *intent filter* in their manifest file. For instance, the Android system always broadcasts an implicit intent message `BOOT_COMPLETED` to inform about the successful completion of the boot process. A common malicious pattern is to register an intent filter to listen for this particular intent message. Upon receiving the intent, the malware starts its malicious service in the background.

PERMISSIONS Android uses a permission system to restrict access to a set of security-critical API functions and resources. An application that, for instance, wants to send SMS, has to declare in its manifest file that it requires the corresponding `SEND_SMS` permission. The user then needs to grant the requested permissions to this application explicitly.

Up to Android 6, a list of all permissions an application requests was presented to the user prior to its installation. The user had to grant all permissions to the application or cancel the installation process completely. It was not possible for the user to control the access in a more fine-grained way. Unfortunately, users therefore often installed overprivileged applications on their devices, including apps with malicious functionality [4, 77].

...and requested permissions.

Since Android 6, however, the permissions are approved by the user at run-time. The problem of overprivileged applications remains, since the Android platform is highly *fragmented* and a large number of devices still run with old Android versions. Moreover, even when permissions are requested at run-time, it is still not always clear to the user why some permission are requested by an app [204].

2.1.2 *Fragmentation*

Manufacturers offer devices in all price categories, ranging from cheap devices to expensive high-end smartphones. As previously mentioned, this is an essential property of Android and a main reason why it is widely distributed amongst mobile device users. Since Android is mostly open source, it is adapted and modified by manufacturers to their particular needs before being shipped with their devices. Thus, there exist a broad variety of Android devices in the wild [4].

A large number of Android devices...

Unfortunately, this comes at a high cost, as manufacturers do not always provide recent software updates for their modified Android versions. Mobile devices therefore often exhibit serious security vulnerabilities, which remain unpatched, posing a high risk to the private data of smartphone users. As a reaction to this problem, Google introduced the *security patch level* and supplies monthly patches since

...run deprecated versions of Android...

Android 6. Still, only a few manufacturers provide these patches to their customers in time [124].

For example, Google reported that 0.3% of all Android devices still run with Android 2.3 in September 2018 [125]. At first glance, this seems to be a negligible fraction of devices. However, note that more than 2.5 billion devices run Android as their operating system [5]. This means there are more than 7.5 million devices still running a highly deprecated version of Android.

...that exhibit vulnerabilities exploitable by malware.

For malware researchers, the fragmentation of Android is a relevant problem, as malicious applications can still exploit vulnerabilities that have already been closed in more recent Android versions. In consequence, the malware will still be successful, since many Android devices are susceptible to these vulnerabilities.

2.1.3 *Malware*

As already discussed in the introduction of this thesis, the number of mobile malware has grown significantly within the past decade. While a small fraction of these applications have also been found in the official Google Play Store in the past, most of this malware spreads through other sources, such as alternative markets or websites [13].

To trick users into installing these malicious applications, malware authors follow different strategies [223]. A widespread method is to *repackage* legitimate applications with malicious functionality and upload them to alternative markets, which are particularly popular in countries like Russia or China. Furthermore, malware authors often use social engineering techniques, for instance, to get users to visit compromised websites. The malware is then downloaded from these websites—in some cases even without the user's knowledge.

Malware infects devices through different attack vectors.

Once installed on the device, malware pursues different goals. While some malicious applications try to escalate their privileges by exploiting vulnerabilities in the operating system, others silently wait for commands from external *Command and Control (C&C) Servers*. Other types silently steal and transmit sensitive information, such as banking credentials, to the malware authors. In most cases, malware authors pursue a financial interest, i.e., they try to steal money from the user of an infected device. We discuss some common types of malware in the following [202, 223]. Note that also combinations of these behavioral patterns can often be found in malware.

A large variety of malware exists, including malware variants that...

- *Premium-SMS malware.* A common scam targeting mobile device users is the subscription to expensive premium services owned by the malware authors. In the case of Premium-SMS malware, the malicious applications send, for instance, SMS to fee-based services without the knowledge of the users. Zhou et al. [223]

...send SMS to premium services,...

found that a large fraction of the malware analyzed by them belong to this kind of malware.

- *Ransomware.* This type of malware locks the user's device and then demands a ransom from the user to unlock the device again. To lock the device, Android ransomware usually uses screen overlays, which are rendered on the very top of the screen, thus effectively hindering the user from accessing the compromised device. Similarly, other variants of this malware encrypt the data on the device. After successfully locking the device, the fraudsters blackmail users to pay several hundred dollars to regain access to their device. Interestingly, the number of samples belonging to this type of malware has significantly grown in the past few years, according to Wei et al. [202].

...blackmail mobile device users,...

- *Banking trojans.* The members of this type try to steal the user's bank account credentials. For example, the malware family *Zitmo* aims to circumvent the two-factor authentication mechanism used by most banks to protect a user's bank accounts [76]. Due to this security mechanism, it is not sufficient for the fraudsters to solely compromise the victim's computer, since they still require the mobile transaction authentication numbers (TANs). Therefore, the fraudsters trick users into installing malware on their mobile devices. The banking trojan then intercepts the mobile TAN and forwards them to the malware authors.

...steal sensitive data from the device,...

- *Adware.* This type of mobile applications cannot necessarily be categorized as malware, even though there is an intersection between both application sets. In particular, advertising libraries are a prevalent way for developers of mobile applications to earn money. In some cases, however, the advertising is very aggressive and can thus significantly impact the user experience. For instance, Google banned several applications from their market due to dubious advertising practices [116]. Nonetheless, since also a large fraction of legitimate applications are using advertising libraries, any boundary between legitimate and illicit usage of these libraries is somewhat arbitrary. Applications belonging to this category are therefore also oftentimes considered as *grayware*.

...or use aggressive advertising.

Regardless of the malware type, the applications usually try to remain undetected by users and anti-virus scanners.

To avoid being detected by users, malware often actively hides suspicious elements, such as application icons, from them. In many cases, the malicious functionality runs as a background service on the device and is therefore not easy to detect. Furthermore, malicious services are commonly triggered by system events like, for instance, in-

Malware uses various techniques...

tent messages that are only sent if the screen is turned off. This makes their detection even more difficult.

Moreover, several obfuscation techniques exist that allow malicious applications to impede their detection by anti-virus scanners. While the previously mentioned *repackaging method* can already be sufficient to slip through the recognition system of many anti-virus scanners, more advanced obfuscation techniques download or decrypt the malicious payload at run-time and are therefore extremely hard to detect [see 160].

...to slip through current detection systems.

In the following sections, we give more details on the limitations of current detection systems and discuss techniques that can alleviate the drawbacks of these systems. Furthermore, we provide more examples of actual malware and details on their inner working throughout this thesis. For instance, in Chapter 3 we discuss in detail a malware that uses an ultrasonic side channel to steal sensitive data from the mobile device.

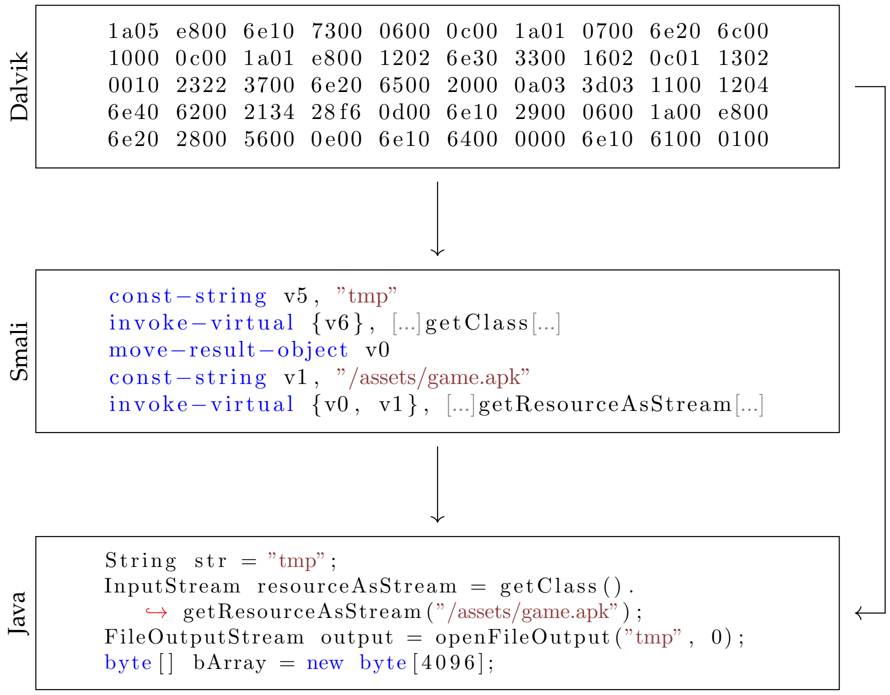
2.1.4 *Anti-Virus Scanners*

AV scanners often rely on signatures...

Anti-virus scanners are still the most common way to detect malicious applications on computers and mobile devices. Unfortunately, the detection capability of these scanners is often limited, as their detection mechanism relies on *detection signatures* in many cases [see 16, 145, 189, 208]. These signatures are known detection patterns, like unique byte sequences in malicious samples, which are stored in the scanner's database. If a signature in the database matches with (parts of) an application, the application gets *flagged* as malicious by the anti-virus scanner.

...which can be easily circumvented.

While the signature-based approach is a simple yet often effective method for the detection of malware, it exhibits several drawbacks. Most importantly, it fails as soon as there exists no proper signature for a malware in the database. Therefore, the signature database needs to be continuously updated to allow for reliable detection. Due to the growing number of malicious software for Android (see Chapter 1), however, it becomes increasingly difficult for anti-virus vendors to provide signatures for new malware in time. This, in turn, leads to time windows in which devices remain vulnerable [53]. Besides, malware authors usually only need to slightly modify their malicious samples to bypass anti-virus products, for example, by repackaging their malicious code into different legitimate applications [39, 223]. The reason for this is that these signatures are often crafted such that they ideally produce no false positives. Consequently, they are not generic enough to compensate even for small modifications to the malware. Thus, simple obfuscation techniques can already be sufficient to circumvent these detection systems.



```
1a05 e800 6e10 7300 0600 0c00 1a01 0700 6e20 6c00
1000 0c00 1a01 e800 1202 6e30 3300 1602 0c01 1302
0010 2322 3700 6e20 6500 2000 0a03 3d03 1100 1204
6e40 6200 2134 28f6 0d00 6e10 2900 0600 1a00 e800
6e20 2800 5600 0e00 6e10 6400 0000 6e10 6100 0100
```

```
const-string v5, "tmp"
invoke-virtual {v6}, [...]getClass[...]
move-result-object v0
const-string v1, "/assets/game.apk"
invoke-virtual {v0, v1}, [...]getResourceAsStream[...]
```

```
String str = "tmp";
InputStream resourceAsStream = getClass().
    ↪ getResourceAsStream("/assets/game.apk");
FileOutputStream output = openFileOutput("tmp", 0);
byte[] bArray = new byte[4096];
```

Figure 2: The Dalvik bytecode (*dex code*) of an application can be disassembled to *smali* code or decompiled to Java code.

2.1.5 *Application Analysis*

To overcome the limitations of the prevalent signature-based approach, more advanced methods have been proposed to fend off malware more effectively. While these approaches may significantly differ from each other, they all rely on *static* and *dynamic* analysis techniques to gather information from an application [23]. The extracted information is, in turn, used to decide whether the application exhibits malicious characteristics or not. In the following, we provide a brief description of both analysis techniques and discuss their advantages and disadvantages.

STATIC ANALYSIS When performing a static analysis of an application, information is extracted from the application without executing it, for instance, by analyzing its source code. In the case of Android application analysis, the source code of an application is in most cases unavailable. Therefore, the *Android manifest* and the *dex code* are considered as a starting point for the analysis, as these usually contain the main functionality of an application.

In static analysis, applications are analyzed without executing them.

Before the analysis, both files need to be translated into an interpretable format that can be easily processed later on. While the translation of the manifest file is straightforward, it is more involved for the dex code, i.e., the compiled source code of the application. In particular, the decompilation of Dalvik bytecode is often prone to errors, and sometimes even consciously prevented by application developers to hinder theft of intellectual property by competitors. To impede the static analysis of their applications, app developers and malware authors apply obfuscation strategies that lead decompilation tools to fail or even crash [45, 73].

These techniques are often successful in detecting malware...

Instead of using the decompiled Java code of an application, a common approach is therefore to disassemble the bytecode into an intermediate representation, such as *smali* [87], which is easier to handle than the original bytecode. Figure 2 depicts an example of different representations of the same code snippet. Note that, while it is in principle possible to decompile native libraries in a similar way, it is far more complex, as the code lacks crucial information, such as type information and variable names.

...but strong obfuscation can hinder their success.

Overall, static analysis poses an efficient way to identify characteristic functionalities of an application. However, it also has inherent limitations. Most importantly, it does not allow analyzing code that has been encrypted or modified with advanced obfuscation techniques. For example, the execution of code that is downloaded by a malicious application at run-time is almost impossible to detect solely with techniques of static analysis [136, 145, 160]. In presence of these advanced obfuscation techniques, further analysis techniques need to be applied in many cases.

Dynamic analysis can help tackling obfuscation,...

DYNAMIC ANALYSIS To compensate for the weaknesses of static analysis, another strain of methods relies on information that is gathered during the actual execution of an application. In particular, applications are executed in controlled environments, and the actions performed during their execution are documented and analyzed. Using this approach, it is possible to observe malicious behavior which would usually remain hidden when solely relying on static analysis techniques [205]. For instance, dynamic analysis can allow extracting URLs from the network traffic of a malicious application, even though they are stored encrypted in the source code of the malware. Similarly, it might be possible to identify malicious payload that is downloaded at run-time by malicious software.

...but also exhibits inherent limitations.

Despite its advantages, also dynamic analysis techniques suffer from several drawbacks. First of all, these approaches are in general computationally more demanding than static analysis methods. Second, most approaches do not execute the applications directly on the devices' hardware but instead in an emulated environment—also known as *sandboxes*. These sandboxes mostly exhibit various artifacts that allow malicious applications to detect their execution in such an environment and, in consequence, avoid running their malicious code [see 85, 159]. While there also have been proposed approaches that run directly on the hardware [e.g., 117, 147, 210, 214], these lack the flexibility of an emulated environment and also often exhibit artifacts that allow their detection [144].

While the first two limitations of dynamic analysis systems can at least be improved, the third limitation cannot be fixed. In particular, it is impossible to ensure that a dynamic analysis approach triggers all possible actions of an arbitrary program [56]. Thus, also dynamic analysis techniques only allow proving the presence of malicious functionality in an application, but not its absence.

To maximize the probability of a successful detection of malware, static and dynamic analysis techniques should be combined. However, the methods presented in this thesis solely rely on static analysis, since they are designed to run on limited hardware resources. With the growing computational power of mobile devices, however, it should also be possible to extend them with techniques of dynamic analysis in the future.

2.2 MACHINE LEARNING

In recent years, machine learning has become an essential part in various application fields, ranging from recommendation systems in online shops [95, 149, 174] to security-related applications like spam and fraud detection [10, 100]. The popularity of these techniques results from their ability to automatically infer general patterns and dependencies from large amounts of data, thus enabling computer systems to reliably solve different tasks without requiring to be explicitly programmed.

As a result, a large number of machine learning techniques have been proposed that tackle various problems. These methods are commonly divided into three different categories:

While there exist three different types of learning problems,...

- *Supervised Learning.* In these kinds of problems, some data points are given along with their corresponding labels. For instance, in the case of Android malware detection, we have a data set of Android apps given and for each application the information whether it is considered malicious or benign. The learning algorithm is then supposed to derive a model that can predict the correct labels for unknown instances.
- *Unsupervised Learning.* In the unsupervised setting, also some training examples are given but in this scenario without any label information. Instead, the learning algorithm tries to find generic patterns or structures in the underlying data. An example of such a problem is *topic modeling* [28], where a learning algorithm tries to automatically identify relevant topics within a set of unlabeled documents. Finally, it assigns each document to those topics which are probably discussed in it.
- *Reinforcement Learning.* The third strain is reinforcement learning. In this setting, the data does also not contain explicit labels for the data. Instead, it gets feedback on how good or bad a particular action of the machine learning model was. Using the retrieved feedback, the model is improved stepwise in a trial-and-error approach. These algorithms are, for example, used to let computers automatically infer the mechanisms behind games like chess or backgammon.

...we solely focus on Android malware detection...

In this section, we introduce several basic concepts of machine learning that are essential to follow the explanations provided in this thesis. We thereby specifically focus on the application of machine learning for Android malware detection, i.e., a *supervised learning problem* where the detection system needs to distinguish between two different classes. While a detailed introduction into the field of machine learning is out of scope, interested readers are referred to a large body of literature that exists on this topic [e.g., 3, 27, 67].

We start with a formal description of the learning problem considered in this thesis. Afterward, we discuss the SVM algorithm in more detail, as it poses the basis for our proposed detection method. Finally, we give details on several metrics that allow examining the performance of detection systems.

2.2.1 *The Learning Problem*

Let us consider some input $\mathbf{x} \in \mathcal{X}$ that describes the information we have about an Android application. For instance, this information might include the API functions the application uses, the network addresses it tries to communicate with, or some available metadata, like its ratings in the market it has been downloaded from. We assume that there exists an optimal—but unknown—prediction function $f : \mathcal{X} \to \mathcal{Y}$ that allows predicting the correct label for each application of the input space $\mathcal{X}$, i.e., it correctly assigns the true value of the two possible outcomes ($\mathcal{Y} = \{\textit{benign}, \textit{malicious}\}$ or $\mathcal{Y} = \{-1, +1\}$) to each application.

...which is a supervised learning problem.

Unfortunately, we neither have any information on what this prediction function f looks like exactly nor do we know the actual distribution of malicious and benign applications in $\mathcal{X}$. Looking for that particular function f therefore resembles the search for a needle in a haystack. Luckily, there exist a large number of machine learning algorithms that allow us to find appropriate solutions for this task despite our lack of knowledge. Using these techniques, we strive to find a proper approximation $\hat{f} : \mathcal{X} \to \mathcal{Y}$ of the actual prediction function f solely based on a dataset $\mathcal{D}$. In the following, we assume that this dataset contains N samples $\mathbf{x}_i$ along with their corresponding true labels y_i, i.e., $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$.

We are looking for a prediction function...

To find a good approximation $\hat{f}$ based on the given data, we apply a *learning algorithm* that determines the most suitable function from a set of possible candidate functions $\mathcal{H}$, where $\mathcal{H}$ is often referred to as the *hypothesis set* [3]. For example, $\mathcal{H}$ could be the set of all possible linear functions from which the algorithm selects the classifier that yields the lowest number of misclassifications on $\mathcal{D}$.

To compare the different candidates with each other, the learning algorithm uses an *error function* $E(\hat{f})$ to assess the costs of its decision when preferring one candidate over another. At first glance, it might seem to be sufficient to select the hypothesis as our final classification model that yields the lowest number of misclassifications on the available data $\mathcal{D}$. However, this will often result in a classifier that performs poorly on previously unseen data. Instead, further considerations are necessary to find a proper error function that allows the learning algorithm to select a classification model with good generalization properties.

...that enables us to distinguish between malicious and benign data.

2.2.2 *Generalization and Regularization*

Finding a classification model that performs well in general in the previously discussed setting is difficult, since we do not know the actual underlying distribution of applications. Instead, we only know a small fraction of N samples, which the learning algorithm uses to derive some estimates about the actual distribution. Depending on the expressiveness of the given data $\mathcal{D}$, the resulting classification model will perform well or poorly on unseen data. Unfortunately, the given data is often noisy, thus making it difficult for the learning algorithm to select a proper classification model.

When using an error function $E_{\text{in}}(\hat{f})$ that only measures the number of misclassifications on this data, we will mostly end up with a classifier that is affected by noise. The error function $E_{\text{in}}(\hat{f})$ is the *empirical error* or also often called the *in-sample error*. Instead of solely using $E_{in}(\hat{f})$ to measure the performance of different classifiers, we therefore need a possibility to estimate the *expected error* (or *out-of-sample error*) $E_{out}(\hat{f})$. This would, in turn, allow us to assess the general detection performance of all possible classification models in our hypothesis set $\mathcal{H}$.

Solely minimizing the error on the available data...

REGULARIZATION Fortunately, there exists the principle of *Structural Risk Minimization*, which was proposed by Vapnik and Chervonenkis in 1974 [196]. The concept allows bounding the expected error of a classification model. Consequently, it becomes feasible to select a classification model for which we can derive certain guarantees on its detection performance on unseen data. In particular, Vapnik and Chervonenkis show that it is possible to find an upper bound for the (unknown) expected error $E_{\text{out}}(\hat{f})$ by introducing an additional *regularization term* [3]:

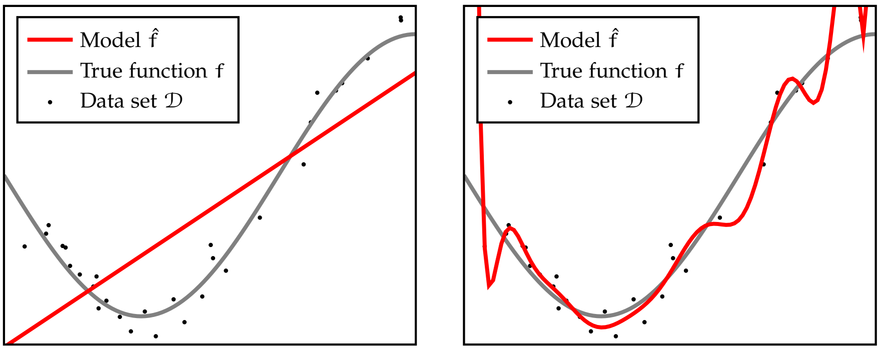
$$E_{\text{out}}(\hat{f}) \leqslant E_{\text{in}}(\hat{f}) + \Omega(\mathcal{H}), \tag{1}$$

...leads to overfitted classification models.

where the regularization term Ω depends on the considered hypothesis set $\mathcal{H}$. While it is out of the scope of this thesis to discuss the underlying mathematical considerations behind this expression in further detail, we want to at least motivate how regularization helps us solving our initial problem. Interested readers are referred to the large body of literature that exists on this topic and in which the underlying concepts are discussed in detail [e.g., 3, 196].

To tackle this problem...

Simply put, the regularization term penalizes the selection of more complex classification models, since these have a higher probability to fit towards noise. In contrast, the empirical error term ensures that the resulting classification model is not too simple (i.e., too far away from the actual model f), as it would otherwise not yield a satisfactory detection performance on the given dataset $\mathcal{D}$. If a classifier fits to the noise in the training data without generalizing the underlying



(a) Underfitting

(b) Overfitting

Figure 3: Underfitting and overfitting

concept, we refer to it as an *overfitted* model. Conversely, if a classifier does not generalize the underlying concept of the data, since the model lacks complexity, we refer to it as an *underfitted* model.

Figure 3 depicts examples of *underfitting* and *overfitting* for a regression problem. In the first case, a linear model (polynomial with degree 1) is used to describe the underlying data. Unfortunately, it does not fit to the data very well, as the distribution of the data points is not linear. Using a polynomial of degree 15, however, also leads to unsatisfying results, since it fits noisy points within the dataset. By adding a regularization term to the optimization problem that the learning algorithm solves, it is possible to find a solution that is close to the true function f.

...we apply the concept of regularization...

2.2.3 *Training and Testing*

Although the introduction of a regularization term is essential for finding a good classification model, the boundary defined in Equation 1 is in practice still too loose to get a good estimate of the expected error E_{out}.

As a remedy, it is common practice to split the available dataset $\mathcal{D}$ into a separate *training dataset* $\mathcal{D}_{\text{train}}$ and a *test dataset* $\mathcal{D}_{\text{test}}$. The classification model is then solely trained on the training dataset and its performance is evaluated on the remaining test data to get an estimate of the expected error E_{out}. During the learning phase, different classification models need to be compared with each other in order to eventually select the best model. Comparing the models based on their performance on the test dataset, however, would again lead to overfitting. The reason is that a classifier might perform well on the test data by coincidence and is then preferred over another model with better generalization performance.

...and carefully split the available data...

To avoid this, we further split the training data into a (smaller) *training dataset* $\mathcal{D}^{o}_{\text{train}}$ and a *validation dataset* $\mathcal{D}_{\text{val}}$. We train different models on $\mathcal{D}^{o}_{\text{train}}$ and evaluate their detection performance on $\mathcal{D}_{\text{val}}$. Finally, we can pick the best model based on the results obtained on $\mathcal{D}_{\text{val}}$, retrain it again on the complete training data $\mathcal{D}_{\text{train}}$, and evaluate its performance on $\mathcal{D}_{\text{test}}$.

...into different datasets for training and testing.

While this approach significantly lowers the chance of overfitting the model, it introduces another problem. Since less data is available for training, the overall detection capabilities of the resulting classifier are often negatively affected. Consequently, a less suitable model might be picked by the learning algorithm due to the lack of data. Fortunately, there exist several techniques that help avoid overfitting, while still allowing us to consider the complete training data $\mathcal{D}_{\text{train}}$. One commonly used technique for this purpose is k-fold cross-validation.

K-FOLD CROSS VALIDATION This approach selects the best model by first splitting the training data $\mathcal{D}_{\text{train}}$ into K equally-sized chunks. The learning algorithm then trains different models on $K - 1$ chunks and evaluates their performance on the hold-back chunk. This process is repeated K times, where each time another chunk is picked as the validation set. Finally, the model that yields the best average performance for all runs is selected and retrained on the complete data $\mathcal{D}_{\text{train}}$. Commonly used values for K are 5 and 10, respectively.

2.2.4 *From Applications to Vectors*

Machine learning algorithms cannot handle Android apps directly.

After having obtained an overview of the principles behind machine learning, let us have a short outlook at how these techniques can help us to address the problem of Android malware detection in practice. At first glance, machine learning techniques do not seem to be particularly useful for this purpose, as they usually operate on mathematical vector spaces. Therefore, they are not able to handle Android applications directly.

To solve this problem, we first extract *features* from an Android application that later allow its description in a vector space. For instance, the number of requested permissions or the occurrence of certain API calls may be used as such features. Note that we provide details on the exact features we use for the detection of malicious Android applications in Chapter 4.

Therefore, we use specific functions...

Following the feature extraction step, we can finally make use of a particular group of functions that enable us to derive vector representations of Android applications based on the extracted features. Again, we provide more concrete examples of some mapping functions later in this thesis. For now, it is sufficient to know that these functions exist and that they pose a useful tool to make machine learning techniques applicable to Android malware detection.

Formally, the mapping process can be defined as follows. Let $\mathcal{Z}$ be the set of all Android applications and $z \in \mathcal{Z}$ a specific Android application belonging to that set. A feature map $\varphi : \mathcal{Z} \to \mathcal{X}$ then maps the application z to $\mathcal{X} \subseteq \mathbb{R}^d$ where it is represented by a d-dimensional vector $\boldsymbol{x}$:

$$\boldsymbol{x} = \varphi(z) = (\varphi_1(z), \dots, \varphi_d(z)) \quad \text{with} \quad d \in \mathbb{N}^+. \tag{2}$$

Oftentimes, each feature is associated with a particular dimension $\varphi_i(z)$ in the vector space. The dimensionality d of that vector space can even be infinite. While we do not have to deal with such large vector spaces throughout this thesis, it should be mentioned that there indeed exist machine learning techniques that tackle learning problems even in such an *infinite dimensional space* [48, 176].

...that enable us to map Android apps into a vector space.

2.2.5 *Support Vector Machines*

The Support Vector Machine (SVM) builds a class of supervised learning methods, which are widely applied in many different fields, such as pattern recognition or malware detection. Initially proposed by Vapnik in 1979 [195] as a method for solving linear problems, it became particularly popular in 1992 when Boser et al. [32] generalized its mathematical formulation such that it could also be applied to solve non-linear problems. The reason for its popularity originates from the fact that the SVM provides a simple mathematical formulation combined with good generalization guarantees, thus effectively minimizing the chance of overfitting.

The SVM allows solving linear and non-linear problems.

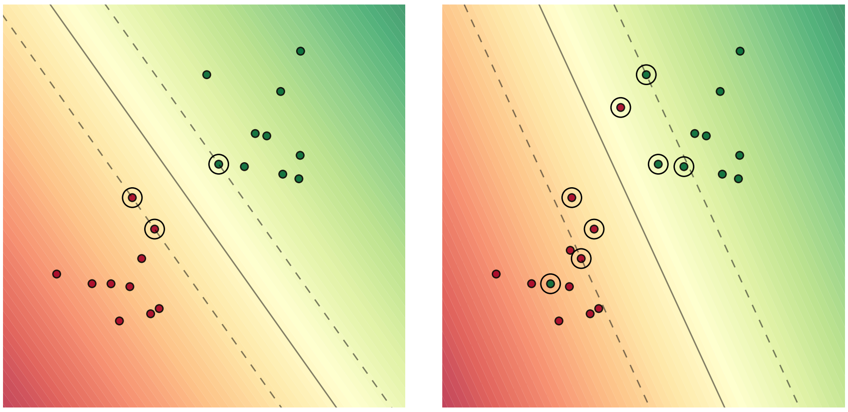
In the following, we discuss the details of the SVM formulation that are necessary to follow the descriptions provided throughout this thesis.

DATASET In the following, we consider a set of training data $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$ that consists of samples in a d-dimensional vector space $\mathbb{R}^d$. Each sample belongs to one of two different classes with labels $y \in \{-1, +1\}$.

HARD-MARGIN SVM Ideally, the two classes are distributed in the vector space such that they can be easily separated through a simple linear function $f : \mathcal{X} \to \mathbb{R}$:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b, \tag{3}$$

where $\boldsymbol{w} \in \mathbb{R}^d$ denotes the vector of *feature weights*, and $b \in \mathbb{R}$ the so-called *bias*. Figure 4a depicts an example for such a distribution in $\mathbb{R}^2$. However, although we already consider a restricted set of linear functions, namely those that separate both classes, there still exists an indefinite number of suitable candidates to pick from, i.e., the cardinality of the hypothesis set $\mathcal{H}$ is infinite. Depending on the selected



(a) Hard-margin SVM

(b) Soft-margin SVM

Figure 4: The *soft-margin* SVM can be applied if the two classes are not linearly separable.

The SVM solves an optimization problem...

function, the classifier might perform poorly on the (still unknown) test data. Fortunately, there is only one particular function that exhibits the highest robustness towards noise, thus effectively minimizing the risk of overfitting—the one that separates both classes with maximum margin. The SVM algorithm allows us to find this particular function.

To this end, the SVM algorithm solves an optimization problem that selects the parameters $\boldsymbol{w}$ and b of the classifier function such that the margin between both sets of the training data is maximized. Mathematically, this can be described as follows:

$$\min_{\boldsymbol{w},b} \quad \tfrac{1}{2}\boldsymbol{w}^\top\boldsymbol{w}\,, \tag{4}$$

$$\text{s. t.} \quad y_i(\boldsymbol{w}^\top\boldsymbol{x}_i + b) \geqslant 1\,,\ i = 1,\ldots,n\,. \tag{5}$$

...which allows finding the hyperplane...

Note that minimizing the term $\frac{1}{2}\boldsymbol{w}^\top\boldsymbol{w}$ corresponds to maximizing the margin [3]. In Figure 4 the margin is visualized by dashed lines. The vectors lying on the margin are the so-called *support vectors* and uniquely define the hyperplane. Consequently, the algorithm selects only a small number of important data points to derive a classification model, instead of memorizing the complete training data. After having successfully determined the optimal weight vector $\boldsymbol{w}_*$ and bias b_*, the SVM can finally classify unknown applications by applying the decision function $h : \mathcal{X} \to \{-1, 1\}$:

$$h(\boldsymbol{x}) = \operatorname{sign}(\boldsymbol{w}_*^\top\boldsymbol{x} + b_*). \tag{6}$$

...that separates both classes with maximum margin.

However, throughout this thesis, we do not consider the bias term during training (i.e., $b = 0$) but instead set it later manually when calibrating the classifier (see Chapter 4). Therefore, the optimization problem is simplified such that the SVM only needs to determine the weight vector $\boldsymbol{w}$ [32]. Similarly, it is possible to extend the weight vector $\mathbf{w}$ and each instance $\mathbf{x}_i$ with an additional dimension, i.e., $\mathbf{w}^T \leftarrow [\mathbf{w}^T, b]$ and $\mathbf{x}_i^T \leftarrow [\mathbf{x}_i^T, 1]$ and omit optimizing the bias term explicitly [103].

SOFT-MARGIN SVM The previously discussed formulation of the SVM has a major restriction, as it only allows solving problems where the classes are perfectly linear separable. In practice, however, this underlying assumption is rather unlikely to hold. Therefore, the so-called *soft-margin SVM* [48] introduces additional *slack variables* $\xi_i \geqslant 0$, which allow the SVM to misclassify a certain number of samples:

By introducing slack variables...

$$\min_{\mathbf{w},b} \quad \tfrac{1}{2}\mathbf{w}^\top\mathbf{w} + C \textstyle\sum_{i=1}^{N} \xi_i \,, \tag{7}$$

$$\text{s. t.} \quad y_i(\mathbf{w}^\top\mathbf{x}_i + b) \geqslant 1 - \xi_i \,,\; i = 1, \ldots, n \,. \tag{8}$$

In this case, the *cost parameter* $C > 0$ weights the penalty for misclassifications. Figure 4b shows an example for a soft-margin SVM. By allowing the misclassification of a small number of noisy data points, it still enables us to find a line that separates both classes. Using the initial formulation of the SVM, this would not have been possible. Note, however, that not only the points lying directly on the margin are now considered as support vectors. Additionally, all misclassified data points and points lying inside the margin are also used to describe the classification model. Consequently, the number of support vectors can also indicate how certain the decisions of the trained model will be for unseen data. In Chapter 6, we will discuss this property of the SVM in more detail.

...we can compensate some noise in the available data...

There exist various formulations of the SVM algorithm that account for the slack with different loss functions. Two commonly used loss functions are, for instance, the *hinge loss* ($q = 1$) and the *square loss* ($q = 2$) function:

$$\min_{\mathbf{w},b} \quad \tfrac{1}{2} \underbrace{\mathbf{w}^\top\mathbf{w}}_{\textit{regularizer}} + C \textstyle\sum_{i=1}^{n} \underbrace{\max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))^q}_{\textit{loss function}}, \tag{9}$$

This unconstrained formulation of the optimization problem consists of two terms. The first term is a *regularization term*, which measures the complexity of the classification model. The second term depends on the training data $\mathcal{D}$ and penalizes the empirical error. Note that it is also possible to use a different regularization term. We will also examine the effects of different regularizers in Chapter 6.

...and still use a linear model.

2.2.6 *Evaluation Metrics*

Throughout this thesis, we assess the performance of Android malware detection methods using various evaluation metrics. Each of these metrics provides information on certain aspects of the performance. In the following, we briefly discuss some evaluation metrics used to measure the performance of machine learning algorithms.

RECEIVER OPERATING CHARACTERISTIC Plotting a Receiver Operating Characteristic (ROC) curve is a very common method to assess the performance of binary classifiers [75]. In our case, the two considered classes contain the malicious and legitimate applications, respectively. However, instead of making a strict binary decision directly, many classifiers apply a threshold to the output of an internal scoring function. For instance, the decision function of an SVM applies a threshold $(-b)$ to the dot product $\boldsymbol{w}^{\top}\boldsymbol{x}$ to determine the class of an application.

ROC curves are commonly used to compare binary classifiers.

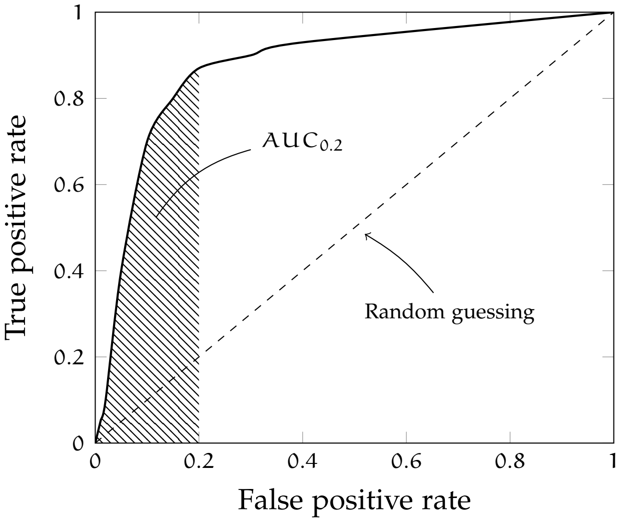


Figure 5: Example of a ROC curve.

Depending on the selected threshold, the number of correctly identified malicious samples (i.e., *true positives*) and the number of false alarms (i.e., *false positives*) may vary. A ROC curve plots the *true positive rate* (TPR) against the *false positive rate* (FPR) for various decision thresholds of the classifier. Figure 5 illustrates an example of such a ROC curve. In our case, the TPR and FPR are defined as:

$$\mathsf{TPR} = \frac{\text{\# true positives}}{\text{\# malware samples}}, \tag{10}$$

$$\mathsf{FPR} = \frac{\text{\# false positives}}{\text{\# benign samples}}. \tag{11}$$

In order to compare the performance of different classifiers with each other, a conventional method is to consider the Area Under (ROC) Curve (AUC) [33]. Using the AUC allows comparing two classifiers solely based on a single scalar value. Unfortunately, the expressiveness of this metric is often limited, especially if the two considered classes are unbalanced—like in the case of Android malware detection. To clarify this, let us consider an example where two classifiers yield the same AUC of 0.9. However, while the first one provides a constant TPR of 0.9, the second classifier detects no malicious samples at all, until reaching an FPR of 0.1, i.e., 10% false positives. Afterward, its TPR increases to 1.0 instantly. In practice, we would prefer the first classifier over the second one, as it enables us to detect 90% of all malicious samples without any false positives. When solely considering the AUC values, however, it does not seem to make any difference, which classifier we select.

The bounded AUC ensures that we select a classifier...

As a solution to this problem, it is possible to use the bounded AUC for performance measurements instead. This metric considers the AUC up to a predefined false positive rate (see Figure 5). Note that a normalized bounded AUC of c (with $0 \leqslant c \leqslant 1$) guarantees that the classifier reaches at least a true positive rate of c at the considered FPR boundary [167]. In this thesis, we refer to an AUC bounded at an FPR of 0.01 as $\mathrm{AUC}_{0.01}$. Moreover, we refer to the detection rate of a classifier calibrated to a FPR of 1% as $\mathrm{TPR}_{0.01}$. Note that the actual FPR of such a classifier might even be lower than 1% on the test data, but never exceeds it throughout all experiments presented in this thesis.

...with a low false positive rate.

PRECISION AND RECALL Instead of using the bounded AUC, the *precision* and *recall* of a classifier are also common metrics to assess its performance on unbalanced datasets. In the context of malware detection, these two metrics can be described as follows:

$$\mathrm{precision} = \frac{\text{\# true positives}}{\text{\# flagged samples}} \tag{12}$$

$$\mathrm{recall} = \mathrm{TPR} = \frac{\text{\# true positives}}{\text{\# malware samples}} \tag{13}$$

The precision of a classifier corresponds to the probability that an application indeed exposes malicious behavior when being flagged as malware by the classifier. The recall, however, simply corresponds to the true positive rate.

Alternatively, one can use precision and recall...

A disadvantage of these two metrics is that they should always be considered together in order to allow for a meaningful interpretation of the obtained results. Nonetheless, it highly depends on the particular problem whether a higher recall or a higher precision is more desirable. To account for this, classification systems are commonly compared with each other using the F_β-measure:

$$F_\beta = (1+\beta^2) \cdot \frac{\mathrm{recall} \cdot \mathrm{precision}}{\beta^2 \cdot \mathrm{precision} + \mathrm{recall}}, \tag{14}$$

where β is a non-negative real valued number that puts more emphasize on precision or on recall, depending on the requirements of the classification model. If $\beta = 1$, precision and recall are weighted equally, corresponding to their harmonic mean. Throughout this thesis, we often provide the precision, recall, and F_1-score in addition to other metrics.

...to measure the performance of a classifier.

2.3 CHAPTER SUMMARY

In this chapter, we have provided the reader with the necessary background knowledge to follow the explanations and descriptions in the upcoming chapters. In particular, we have first briefly explored the internals of the Android OS and its ecosystem. Afterward, we have also discussed the increasing threat of malware targeting the operating system as well as the problems which current solutions have to detect these applications.

In the second part of this chapter, we have discovered the basic concepts of machine learning and also have provided an introduction to the principles of Support Vector Machines. Finally, we have discussed various metrics that are commonly used to evaluate the performance of learning-based detection systems.

3 ULTRASOUND-BASED TRACKING MALWARE

Equipped with the necessary background knowledge from the previous chapter, we can begin our journey towards an efficient learning-based algorithm for Android malware detection. We start our exploration by presenting an empirical study, which examines the prevalence of a sophisticated kind of spyware. In particular, the malware uses an ultrasonic side channel to track unwitting mobile users. We provide a detailed description of the underlying technology in this chapter. While the technology itself is not necessarily harmful and can even be useful for legitimate purposes, we find that it has already been misused by malware to spy on mobile device users.

Based on this actual case of ultrasound-based tracking malware, we are able to give the reader practical insights into the manual analysis of Android applications. That is, how it allows identifying malicious functionalities relevant for the crafting of proper detection methods. Using the characteristics derived through manual inspection, we can build a first learning-based method that allows scanning for characteristic code regions in applications. In addition to that, we also examine the proliferation of ultrasonic beacon technology in general. In summary, we discuss the following aspects in this chapter:

1. We reverse engineer the inner workings of three commercial tracking technologies using ultrasonic beacons. Moreover, we provide detailed insights on how the malicious and legitimate usage of this technology differ from one another.

2. We conduct an empirical study to show where ultrasonic audio beacons are currently used. To this end, we implement two detection methods, which allow us to efficiently scan mobile applications and audio data for indications of ultrasonic side channels.

3. Finally, we empirically evaluate the reliability of the ultrasonic technique under different conditions and present limitations that help determine how and which defenses should provide proper protection.

Before discussing the threats to privacy induced by ultrasonic side channels, we first give a short overview on tracking technologies, particularly focussing on their ultrasonic variants. In this context, we also briefly describe three commercial solutions we investigate throughout our study. Note that not all of these solutions necessarily apply the technology for illicit purposes, but were rather picked for our study since they are known to use the ultrasonic beacon technology.

3.1 MOBILE DEVICE TRACKING

Many companies are using tracking technologies...

Nowadays, a large number of companies and websites uses various tracking technologies to fingerprint desktop and mobile devices. As an example, websites often identify their visitors by collecting information about the devices visiting them. For instance, the used screen resolution and the fonts installed on a system already pose relevant features for this purpose [68]. Using the acquired information, the website operators can, in turn, derive unique fingerprints that often allow the distinct identification of individual users.

...to provide users with targeted advertisements.

While such tracking may help in identifying fraud, for example logins from unknown devices, it is often used for targeted advertising that can impact the privacy of users. Moreover, devices are no longer only fingerprinted and monitored as users surf the web, but also when they open applications on smartphones and other mobile devices [e.g., 105, 123, 152]. Consequently, it becomes possible to track the location of users and their activity even across different devices and applications. Various advertising platforms already provide corresponding services to their customers, including Google's Universal Analytics and Facebook's Conversion Pixel.

A new tracking technology based on ultrasounds...

Recently, several companies have started to explore new ways to track user habits and activities with ultrasonic beacons. In particular, they embed these beacons in the ultrasonic frequency range between 18 and 20 kHz of audio content and detect them with regular mobile applications using the device's microphone. This side channel offers various possibilities for tracking: The mobile application *Shopkick*, for instance, provides rewards to users if they walk into stores that collaborate with the Shopkick company. In contrast to GPS, loudspeakers at the entrance emit an audio beacon that lets Shopkick precisely determine whether the user walked into a store. Furthermore, mobile applications like *Lisnr* and *Signal360* present location-specific content on mobile devices, such as vouchers for festivals and sports events via ultrasonic beacons.

...raises serious privacy concerns.

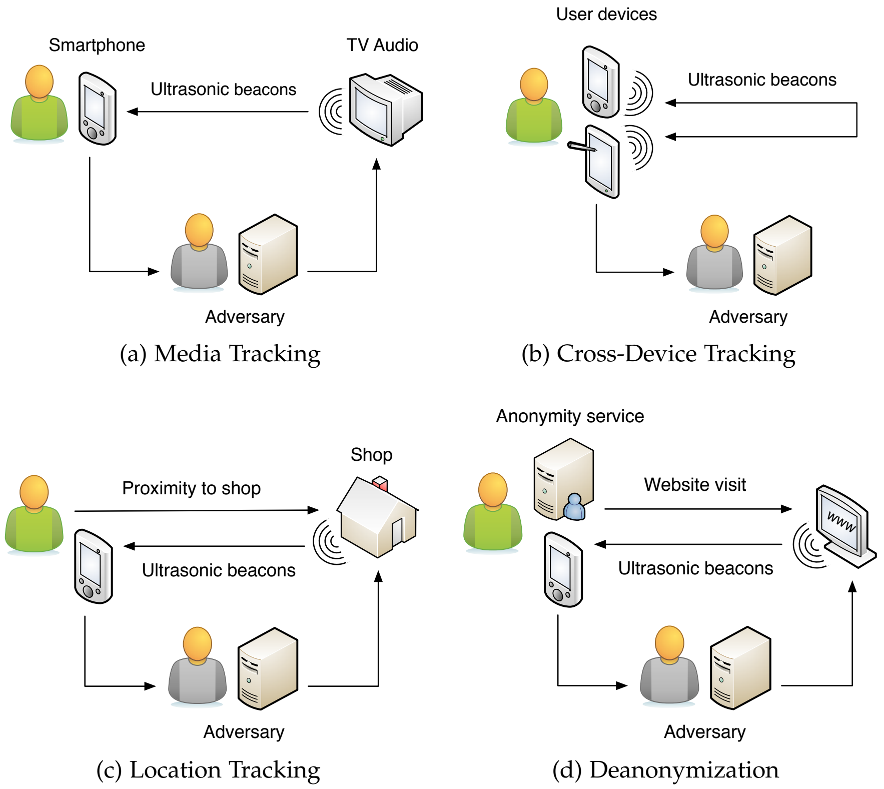
In a particularly alarming case, the developers of *SilverPush* filed a patent, which even raised attention in the media [198] due to its threat to privacy: The patent proposes to mark TV commercials using ultrasonic beacons, thus allowing them to precisely track a user's viewing habits. In contrast to other tracking products, however, the number and names of mobile applications carrying this functionality are unknown. Therefore, the user does not notice that her viewing habits are monitored and linked to her mobile devices.

In the following, we systematically investigate the technical implementation, prevalence, and privacy implications induced by ultrasonic user tracking. In particular, we gain detailed insights into the current state of the art by examining the communication protocols and signal processing of three commercial solutions: Shopkick, Lisnr and SilverPush. By doing this, we are able to develop a proper detection mechanism for mobile applications as well as methods for detecting ultrasonic beacons in audio. These detection methods let us obtain an overview of the current prevalence of ultrasonic tracking used in practice. We start with a discussion on the threats to privacy induced by ultrasonic side channels.

3.2 PRIVACY THREATS

Ultrasonic beacons can be used to...

Ultrasonic side channels on mobile devices can be a threat to the privacy of a user, as they enable unnoticeably tracking locations, behavior and devices. For example, an adversary can spy on the TV viewing habits of a user, locate its position if in range of an ultrasonic signal, or even weaken anonymization techniques. The user just needs to install a regular mobile application that is listening to ultrasonic signals through the microphone in the background. Figure 6 summarizes the resulting privacy threats:



(a) Media Tracking

(b) Cross-Device Tracking

(c) Location Tracking

(d) Deanonymization

Figure 6: Examples of different privacy threats introduced by ultrasonic side channels. (a) Ultrasonic beacons are embedded in TV audio to track the viewing habits of a user; (b) ultrasonic beacons are used to track a user across multiple devices; (c) the user's location is precisely tracked inside a store using ultrasonic signals; (d) visitors of a website are de-anonymized through ultrasonic beacons sent by the website.

MEDIA TRACKING An adversary marks digital media in TV, radio, or the web with ultrasonic beacons and tracks their perception with the user's mobile device. The audio signal may carry arbitrary information, such as a content identifier, the current time, or broadcast location. As a result, it becomes possible to link the media consuming habits to an individual's identity through her mobile device. Where traditional broadcasting via terrestrial, satellite, or cable signals previously provided anonymity to a recipient, her local media selection becomes observable now. In consequence, an adversary can precisely link the watching of even sensitive content, such as adult movies or political documentations, to a single individual — even at varying locations. Advertisers can deduce what and how long an individual is watching and obtain a detailed user profile to deliver highly customized advertisements.

...spy on users' viewing habits,...

CROSS-DEVICE TRACKING Ultrasonic signals also enable an adversary to derive what mobile devices belong to the same individual. When receiving the same signal repeatedly, devices are usually close to each other and probably belong to the same individual. Consequently, an advertiser can track the user's behavior and purchase habits across her devices. By combining different information sources, the advertiser can show more tailored advertisements. Similarly, an adversary can link together private and business devices of a user, if they receive the same ultrasonic signal, thereby providing a potential infection vector for targeted attacks.

...link their devices,...

LOCATION TRACKING An ultrasonic signal also enables an adversary to track the user's movement indoor without requiring GPS. A location, for example a drug store, emits an ultrasonic signal with a location identifier. This information reveals where and when an individual usually stays. Furthermore, the adversary can learn when people are meeting or are in close proximity to each other.

...track their location,...

DE-ANONYMIZATION The side channel through ultrasonic codes makes the de-pseudonymization of Bitcoin and de-anonymization of Tor users possible. As an example, a malicious web service can disclose the relation between a Bitcoin address and a user's real-world identity. Whenever the service shows a uniquely generated address to which the user has to pay, it also transmits an ultrasonic signal to the payer's mobile device. This, in turn, enables the service to link the user's Bitcoin address to her mobile device. A similar attack strategy against Tor users has recently been demonstrated by Mavroudis et al. [139].

...or even de-anonymize them in the internet.

In summary, an adversary is able to obtain a detailed, comprehensive user profile by creating an ultrasonic side channel between the mobile device and an audio sender. Our case study on three commercial ultrasonic tracking technologies reveals that the outlined tracking mechanisms are not just a theoretical threat, but have already been actively deployed.

3.3 technical background

Before presenting the current state of the art on ultrasonic side channels, we briefly introduce the basics of acoustic communication and corresponding information encoding. A reader familiar with these topics can directly proceed to our methodology on detecting ultrasonic implementations in Section 3.4.

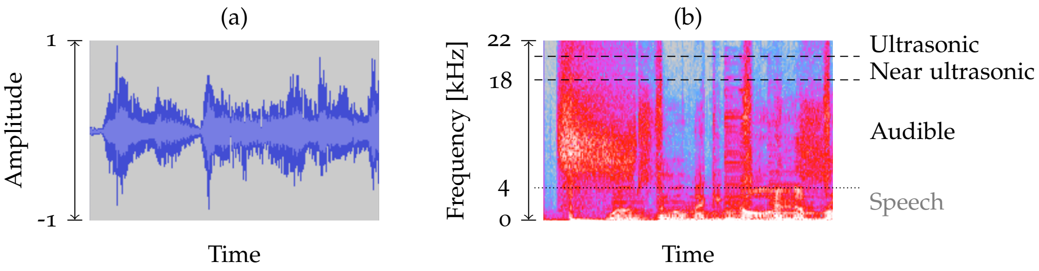


Figure 7: (a) Audio wave of a music track, (b) spectrogram of the frequencies contained in the music track.

3.3.1 *Audible and Inaudible Sound*

Sound can be formally described as a sum of waves with different frequency. While natural sound is usually composed of a wide spectrum of these frequencies, humans are only able to perceive a particular range, where frequencies outside of this range remain inaudible. For designing an inaudible side channel it is thus essential to first pick an appropriate frequency band for transmission:

- *Infrasound ($\leqslant$ 20 Hz):* Frequencies below 20 Hz can generally not be perceived by the human ear. Due to the long wave length, however, infrasound is difficult to create with small devices and moreover less efficient in transmission.

In general, the human ear can perceive frequencies up to 16 kHz reliably.

- *Audible sound (20 Hz–20 kHz):* In general, humans are able to perceive frequencies *consciously* between 20 Hz and 20 kHz. This upper bound decreases with age [101], such that humans of 30 years and older often cannot recognize sound above 18 kHz.
- *Ultrasound ($\geqslant$ 20 kHz):* Frequencies above 20 kHz can also not be perceived by humans. Moreover, the small wave length enables

creating ultrasound from small devices and also provides the ground for a quick transmission.

The frequency range between 18 and 20 kHz can thus be used to transmit information...

As a consequence, ultrasound theoretically is a perfect match for designing an inaudible yet effective side channel between devices. However, most loudspeakers and microphones deployed in commodity hardware are not designed to transmit inaudible sound. Instead, these devices exactly aim at the audible range of frequencies between 20 Hz and 20 kHz [102]. This problem is alleviated by the decreasing hearing performance of humans, leaving a near-ultrasonic frequency range of 18 kHz to 20 kHz for transmission, which is only perceived by very young or sensitive humans.

Consequently, commodity and thus existing audio hardware can be leveraged for establishing a side channel. No additional hardware or technology is needed. An alternative to sound, for example the iBeacon solution, requires a dedicated sender device that emits the Bluetooth signal. Moreover, the receiving device needs to support the Bluetooth Low Energy standard.

...without being recognized by most human beings.

To visually present sound in this paper, we make use of the plots shown in Figure 7, where (a) depicts the amplitude and (b) the spectrogram over time for an exemplary sound. In the latter case, the individual frequencies of the sound are plotted over the y-axis and their power is indicated by brightness. The sound corresponds to a music track and it is visible that also inaudible frequencies above 18 kHz are part of the recording.

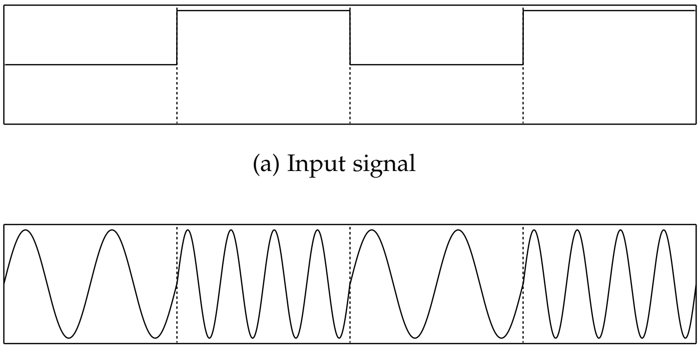
3.3.2 *Encoding of Information*

So far, we have identified the frequency band from 18 kHz to 20 kHz as a promising channel for designing inaudible communication. It thus remains to investigate *how* information can be encoded on this channel. Fortunately, acoustic and electromagnetic waves share several similarities and many basic concepts developed in telecommunication can also be applied for acoustic communication, such as different variants of signal modulations.

M-FSK allows encoding M different symbols...

However, when transmitting information using inaudible sound, we need to make sure that no frequencies outside the selected band occur. This requirement renders the concept of *Frequency Shift Keying (FSK)* attractive for this purposes since other concepts like *Phase Shift Keying (PSK)* potentially introduce discontinuities in the signal. These discontinuities may lead to high instantaneous frequencies and result in perceptible clicks.

In FSK each bit or symbol is represented by a separate frequency within the specified frequency band. An example is depicted in Figure 8 where a simple bit sequence is transmitted using two different frequencies. Obviously, it is possible to generalize this binary FSK and encode M symbols with M separate frequencies. This generalization



(a) Input signal

(b) Frequency Shift Keying (FSK)

Figure 8: Information encoding using FSK modulation.

is known as M-FSK and a variant of it is used by SilverPush and also by Lisnr.

Even though these implement a vanilla M-FSK, the changing frequencies within the given band can also introduce minor discontinuities and thus audible clicks [102]. This effect can be prevented using techniques like continuous-phase frequency shift keying or at least mitigated when lowering the amplitude at frequency transitions as proposed by Deshotel [60].

...using M separate frequencies.

3.3.3 *Sending and Receiving*

Equipped with a frequency band and a simple encoding scheme, an attacker only needs to construct a corresponding sender and a receiver. In the case of media- and cross-device tracking, implementing a sender is rather straightforward, as the attacker just needs to embed the prepared frequency signal into the audio stream broadcast via TV, radio or a web stream. Designing a receiver is a little bit more involved, as the corresponding device needs continuously monitor the sound using a built-in microphone.

Ultrasonic beacons can be injected into various media...

Without loss of generality, we focus on a receiver implemented for the Android platform, as the same concepts also apply to other mobile platforms. The Android platform provides a dedicated class called `AudioRecord` for recording audio data from the microphone without compression. Note that compression algorithms can foil the plan of an ultrasonic side channel, as they may cut off inaudible sounds from the recording. While this class is easy to access, an app still requires the `RECORD_AUDIO` permission for recording audio. Thus, the user also needs to explicitly grant this permission to the app. Unfortunately, users tend to blindly grant permissions to Android applications, if they are interested in their functionality. As a consequence, the permission-based security mechanism of Android does not really stop an application from listening for inaudible beacons.

...including music, television, and even web streams.

The beacons can be silently received by mobile devices.

Furthermore, a continuous stealthy recording can be easily implemented on Android using the concept of services that work in the background so that a user can even switch to another application. In consequence, a covert transmission of an ultrasonic signal can take place at any time, since it may not be clear when a viewer, for example, will watch a TV program that contains the embedded audio beacon. To revive a service after a shutdown of the device, techniques known from Android malware can be employed, such as triggering the service on events like boot-up or finished phone calls.

3.4 METHODOLOGY

To conduct our study, we build two different tools.

With these basics of communication in mind, we are ready to develop two tools for detecting indicators for ultrasonic side channels: While the first one identifies the receiving implementation in an Android application, the other spots the corresponding ultrasound beacons in an audio signal. In the following, we discuss both tools in more detail.

3.4.1 *Detecting Mobile Applications*

To study the prevalence of mobile applications using inaudible sound to track user behavior, we require a detection tool capable of efficiently scanning a large amount of Android applications for corresponding implementations.

The first one detects apps which contain code regions...

Automatically identifying algorithms in program code, however, is a challenging task that requires to abstract from concrete implementations. In the general case determining whether an algorithm is present in a program is undecidable [166]. As a remedy, we thus use a lightweight detection method, which is capable of performing a fuzzy matching of interesting code fragments on a large set of applications.

The design of our method is inspired by a detection technique developed in the context of network intrusion detection [201, 207]. Figure 9 depicts the detection method used. In the first step, we manually select methods from the available sample applications that are known to be crucial for their functionality. This, for instance, includes the Goertzel algorithm and the CRC checksum calculation present in samples of SilverPush and Shopkick, respectively.

...characteristic for ultrasound-based communication.

Throughout the training step, our method identifies the code regions containing these methods and extracts all n-grams with $n = 2$ from the corresponding byte sequences. To generalize different implementations, it keeps only *shared n-grams*, that is, byte sequences of length n that are present in all methods of the same functionality. These shared n-grams are stored in a Bloom filter [30], a classic data structure that allows to compactly describe a set of objects. As a result of this learning phase, our method provides a set of Bloom fil-

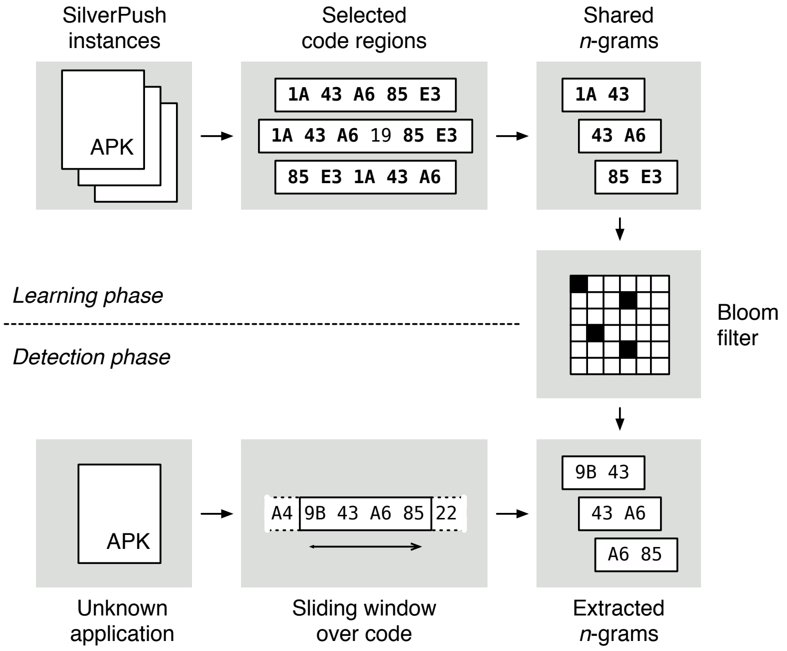


Figure 9: Schematic depiction of the detection method for mobile applications that employ inaudible sound.

ters, where each filter represents one characteristic method indicative of inaudible tracking.

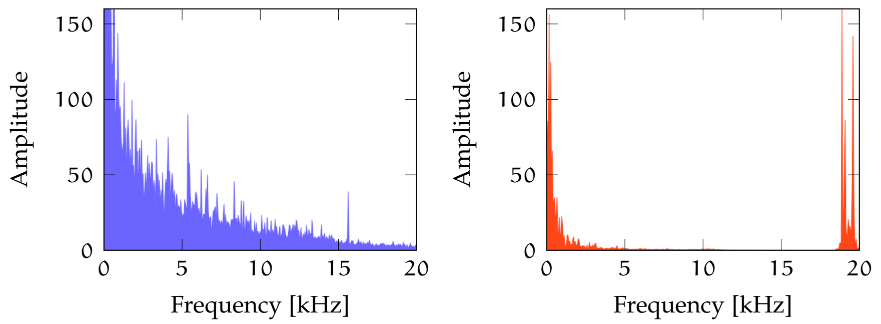
Scanning an unknown Android application for occurrences of the learned patterns is conducted similarly: Our method first identifies all Dalvik code regions in the application and then extracts n-grams by moving a sliding window of 100 bytes over the code. The extracted n-grams under the window are compared against the different Bloom filters and a match occurs if a pre-defined amount of the n-grams is also present in one of the Bloom filters. Ultimately, an application is flagged as being suspicious, if at least one characteristic method is found in the code regions. Note that this approach can be applied to spot arbitrary code of interest.

3.4.2 *Detecting Ultrasonic Beacons*

The second tool scans for unusual signals...

As ultrasonic beacons may vary between different techniques, we also need a broad detection approach to spot previously unknown beacons. Furthermore, the approach must be able to analyze large amounts of data efficiently and we need to ensure that the algorithm produces no or at least only few false positives which can be manually verified later.

Based on our insights from exploring current commercial tracking technologies (see Section 3.5), we assume that the energy of the beacons in the frequency band between 18 kHz and 20 kHz is higher than in other common signals. Thus, an anomaly detection in the



(a) Music library

(b) Lisnr sample

Figure 10: Plot (a) shows the frequency distribution of more than 1,500 songs whereas Figure (b) depicts the frequency distribution of an audio sample containing a Lisnr audio beacon.

...in the ultrasonic frequency band.

considered frequency band seems a promising candidate to identify arbitrary ultrasonic beacons. To this end, we require a meaningful model of the energy distribution for each signal class of interest. This includes audio files, TV streams and environmental sounds in a common shopping mall.

3.4.3 *Discussion*

The described tools allow the scanning of large amounts of data for evidence of ultrasonic beacon technology. In order to be applicable for this purpose, both tools are highly optimized to produce only very few false positives. Moreover, they can scan the data within a reasonably short amount of time. Note, however, that the efficiency of these tools is achieved at the cost of a loss in generality. This holds, in particular, for the application scanner as it needs to identify characteristic methods for each SDK individually, which can be used for training a precise detection model. Nonetheless, despite this drawback, the method fits perfectly the requirements we have for this study, as we will see in the next section.

3.5 EMPIRICAL STUDY

We proceed with an investigation of commercial ultrasonic tracking technologies, namely SilverPush, Lisnr, and Shopkick. These three applications use ultrasound to send messages to the mobile device, but with different use cases: SilverPush targets media and cross-device tracking, while Lisnr and Shopkick perform location tracking (cf. Section 3.2). In the following, we especially focus on the inner workings of SilverPush and Lisnr and additionally discuss Shopkick where it differs to Lisnr or SilverPush.

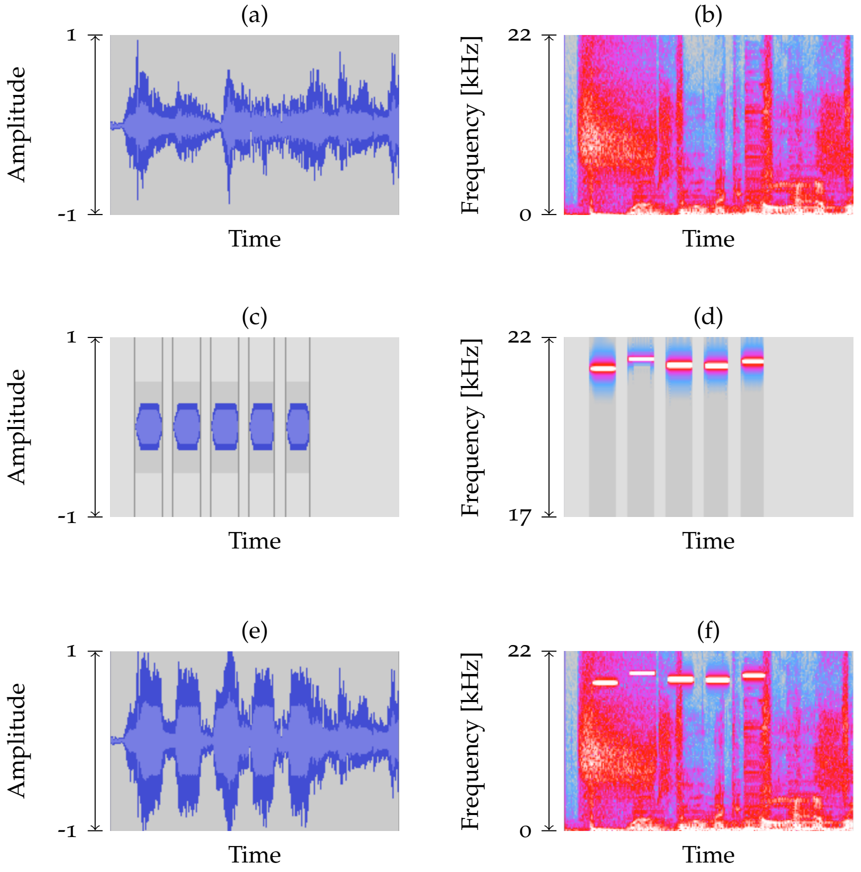


Figure 11: Example of transmission of ultrasonic beacons. The upper three panels depict the audio wave of an audio signal, while the lower three panels show the corresponding Spectrogram. (a)-(b) show a music track, (c)-(d) a ultrasonic beacon, (e)-(f) shows the result after embedding the beacon into the original track.

To gain insight into their functionality, we make use of the reverse-engineering tools *Radare2* and *Androguard*. In particular, we apply Radare2 to extract the Dalvik bytecode of Android applications and employ Androguard to decompile Java code from the applications. We switch to Radare2 when an application uses native code through the Android Native Development Kit (NDK) or Androguard does not resolve a method's control flow correctly. As no obfuscation has been used in the SilverPush, Lisnr, and Shopkick samples, this semi-automatic analysis proceeds rather quickly and we gain detailed insights on their communication protocols and signal processing.

3.5.1 *Case Study SilverPush*

We start our investigation of the SilverPush implementation with the GitHub repository of Kevin Finisterre [81], who collected initial information about SilverPush after the media coverage in November 2015.

The repository contains 21 Android applications that we examined for the functionality to retrieve ultrasonic beacons.

communication protocol SilverPush uses the near-ultrasonic frequency range to transmit audio beacons, as Section 3.3.1 generally motivates. These beacons consist of five letters from the English alphabet where each letter is encoded using a separate frequency in the range between 18 kHz and 20 kHz. The encoding scheme thus corresponds to an M-FSK with M being the number of letters in the used alphabet.

SilverPush encodes all letters of the English alphabet...

As the acoustic transmission can be subject to noise or other high-frequency sounds, the implementation contains two simple mechanisms for error detection: (1) no letter must appear twice in a transmitted beacon and (2) the letter 'A' must be present in every beacon. Obviously, these mechanisms limit the set of available beacons for transmission, but in combination realize a naive but effective error detection. The audio snippet in Figure 11 (c) and (d) contains a valid audio beacon of SilverPush, where Figure 11 (e) and (f) depict the same beacon exemplarily embedded into an audio signal.

...using a separate frequency for each of these letters.

Listing 1: Decompiled Goertzel algorithm.

```
public double getMagnitude()
{
  a = new double[2];
  b = 0;
  while (b < this.n) {
    this.processSample(this.data[b]);
    b = (b + 1);
  }
  this.getRealImag(a);
  c = this[0];
  d = this[1];
  e = Math.sqrt(((c * c) + (d * d)));
  this.resetGoertzel();
  return e;
}
```

signal processing The SilverPush implementation records audio from an available microphone at a sampling rate of 44.1 kHz and directly analyses the recorded data in blocks of 4,096 audio samples. Due to the use of a sampling frequency of 44.1 kHz, the implementation is capable of detecting beacons up to 22 kHz—provided that the available loudspeakers and microphones support such a high frequency. The developers seem to have been aware of this problem and thus limited the FSK encoding of letters to 20 kHz.

Instead of analyzing the full frequency spectrum...

To decode the beacons from the raw audio data, the implementation makes use of the so called *Goertzel algorithm*, a classic signal processing algorithm that is widely used in telecommunication systems, for example, for identifying DTMF tones in software. The algorithm's advantage compared to the more common *Fast Fourier Transform (FFT)* is its ability to detect a single target frequency precisely with little computational effort. On the contrary, the Fourier transform provides access to several frequencies at once and thus is a more robust tool for spotting a signal. It is worth noting that we found one seemingly older Android application of SilverPush during our empirical evaluation that uses a Fourier transform. However, it seems that all current instances use the Goertzel algorithm.

...it scans only for certain frequencies.

Listing 1 shows the decompiled and characteristic Goertzel algorithm as found in the implementation of SilverPush. The algorithm runs over all 4,096 audio samples, calculates the real and imaginary part of a specified target frequency in lines 5–9, and finally returns the magnitude obtained from line 12.

DATA COLLECTION After collecting a valid beacon, the implementation then sends the resolved audio beacon to a server in unencrypted form, together with device information that are usable to identify the device, such as the IMEI, the Android ID, the device model, and even the phone number. While this transmission of personal data is already a privacy invasion, the fact that it is triggered from the audio of a TV transmission makes this a more than questionable approach.

The malware collects lots of sensitive information...

Listing 2: Decompiled emulator detection.

```
if (Build.BRAND.contains("generic")
     || Build.DEVICE.contains("generic")
     || Build.PRODUCT.contains("sdk")
     || Build.HARDWARE.contains("goldfish")
     || Build.MANUFACTURER.contains("Genymotion")
     || Build.PRODUCT.contains("vbox86p")
     || Build.DEVICE.contains("vbox86p")
     || Build.HARDWARE.contains("vbox86")) {
   SP_MiscUtil.a(context, "audio_tracker", false);
   ...
   Log.w(..., "Working on an Emulator! SilverPush SDK Disabled");
   return false;
}
```

EMULATOR DETECTION In addition to transmitting sensitive data through an ultrasonic side channel, the SilverPush SDK also shares another characteristic with common malware. In particular, it checks whether the application is running in an emulator before starting its background service. In the case of a successful detection, it prevents the service from being started. Listing 2 depicts the decompiled code snippet of this functionality, which can be found in the class *com.silverpush.sdk.android.SilverPush*[1]. As the snippet shows, SilverPush checks the build information in order to verify whether it is

...and cannot be detected when executed in an emulator.

[1] 919ad85f95d3562c09fc0d589d5521916720678ee7db877d56f4b8f91bb8c20b

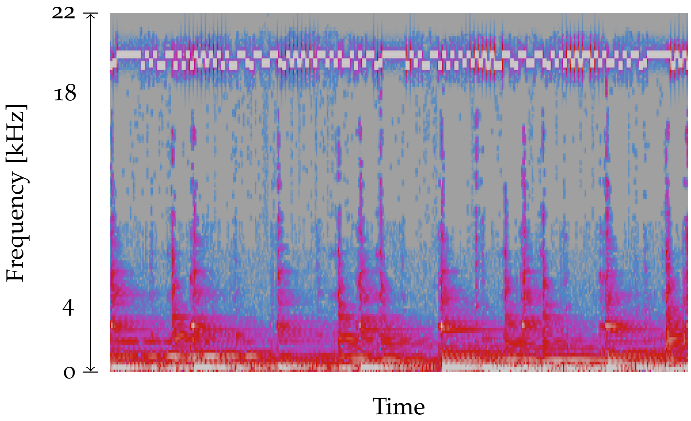


Figure 12: Spectrogram of a disclosed Lisnr audio beacon. An FSK scheme encodes a repeating bit sequence in the near-ultrasonic frequency range between 18.5 and 19.5 kHz.

being executed in an emulator — a procedure that can often be found in malicious samples [36].

3.5.2 *Case Study Lisnr*

We continue our investigation with Lisnr that realizes an ultrasonic side channel to display location-specific content on the mobile device. For example, during a festival, participants can receive notifications, such as welcome messages or vouchers when they are close to a specific location.

Lisnr's technique resembles that of SilverPush...

communication protocol Figure 12 shows a disclosed Lisnr audio beacon in the near-ultrasonic frequency range that we spotted in a music song. The switching frequencies reveal an M-FSK encoding scheme ($M = 3$) between 18.5 and 19.5 kHz. Moreover, the beacon is continuously repeated, as the unique frequency block order in the figure also emphasizes.

signal processing Lisnr records audio with 44.1 kHz and generally analyzes the data in blocks of 4,410 samples. In contrast to SilverPush, its audio analysis is implemented in native code using the Android NDK. In this way, the computationally demanding analysis runs directly on the smartphone's CPU without an intermediate virtual machine. We find that the native code in Lisnr implements both, the Goertzel algorithm and an FFT, for decoding ultrasonic signals. After detecting a code, Lisnr shows location-specific content to the mobile device user.

Similarly, Shopkick implements an FFT in native code for detecting audio beacons in collaborating shops. If a customer wants to earn a reward, she needs to start the audio analysis manually and the applications then performs an analysis of the full frequency spectrum, which is computationally more demanding than the Goertzel algorithm. Thus, it runs for a few seconds only in order to avoid that the battery drains too quickly.

DATA COLLECTION In contrast to SilverPush, we find no indication that the Lisnr SDK collects lots of sensitive information from the device. Instead, Lisnr seems to mainly use the *Android Advertising ID* (AAID) in order to identify devices. The Advertising ID is a resettable ID, which has been introduced by Google to allow personalized advertising, while limiting the impact on users' privacy. Unlike Lisnr, Shopkick tries to gather lots of sensitive data, including the email address and phone number of the user. In contrast to SilverPush, however, this information must be actively provided to Shopkick by the user. Still, it is questionable if Shopkick actually requires all this information just to properly provide users with its services.

...but it is less privacy-invasive. Most importantly,...

3.5.3 *Discussion*

In summary, SilverPush and Lisnr share essential similarities in their communication protocols and signal processing. Both, for example, use an FSK near the ultrasonic range and employ the Goertzel algorithm in the background. However, SilverPush does not inform the user about the tracking whereas the user is aware of Lisnr's and Shopkick's audio analysis. All these technologies show that the step between a legitimate use and spying is rather small. The privacy threat posed by ultrasonic beacons hinges on the notification of the user, who solely depends on this information: First, she cannot hear the audio beacons when, for example, watching TV. Second, she may not know that her mobile device is listening in the background, since there is no visible indication that an application contains this form of device tracking.

...users are informed about its presence in an application.

3.6 EVALUATION

With our two tools to spot ultrasonic implementations from Section 3.4 and our insights into the current state of the art in ultrasonic tracking from previous section, we are ready to conduct an empirical evaluation and assess the impact of this privacy threat in practice. We especially perform the following three groups of experiments:

1. *Controlled experiment*. We first examine the technical reliability and evaluate limitations of ultrasonic side channels under realistic conditions with human subjects and mobile devices (Section 3.6.1).

2. *Audio beacons in the wild.* To uncover the presence of ultrasonic beacons, we scan different locations, TV channels, and websites for indications of ultrasonic side channels (Section 3.6.2).
3. *Applications in the wild.* We finally investigate the presence of ultrasonic implementations by analyzing over 1,3 Million Android applications collected in December 2015 (Section 3.6.3).

3.6.1 *Controlled Experiment*

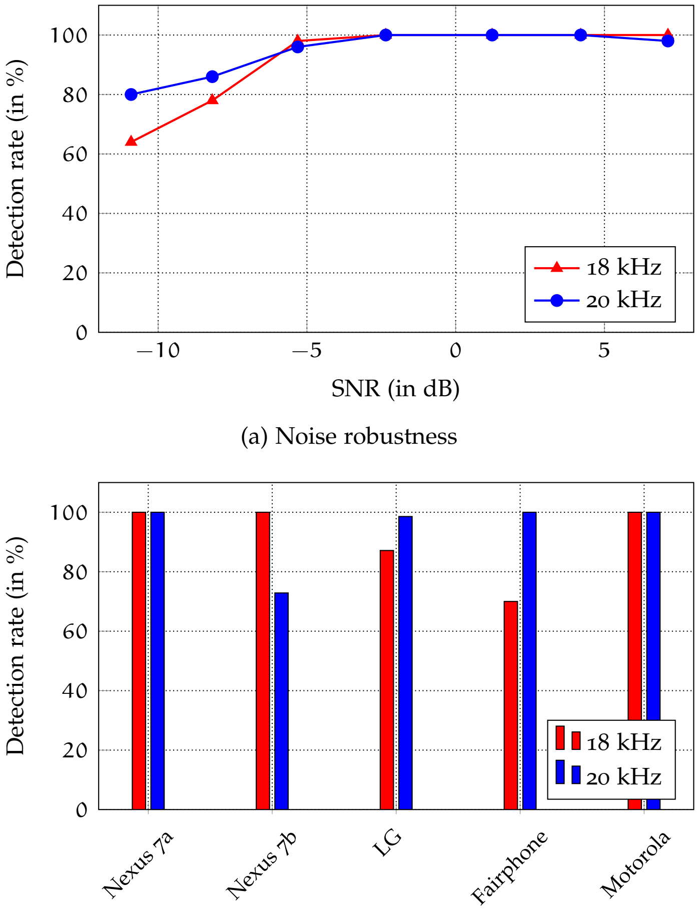
Although the companies behind SilverPush, Lisnr, and Shopkick market their technique as an effective approach for their respective tracking scenario, we have been skeptical about the reliability of the underlying side channel in practice. In particular, it is questionable to which extent the built-in microphones of common devices are capable to reliably perceive high frequencies in presence of environmental noise, since they are mainly intended to work within the voice band. Moreover, the audio beacons might still get detected by some people due to the varying frequency sensitivity of the human ear. Consequently, we first conduct a proof-of-concept experiment consisting of two different scenarios: In the first scenario we explore hardware limitations of common devices, while in the second scenario, we answer the question whether ultrasonic beacons are undetectable by the human ear.

EXPERIMENTAL SETUP We create ultrasonic beacons that cover different frequencies, lengths and sound levels. In particular, we choose frequencies between 18 and 20 kHz and vary the signal length between 0.3 and 1 seconds, and the sound level between 0 and 18 dB. The resulting audio beacons are then embedded in different video files that cover realistic conditions such as speech, music or silence. The files are played through standard TV loudspeakers at a common loudness level of 60 dBA. In both scenarios, the TV plays the test sequences while users or devices listen to it in a fixed distance of about two meters.

Despite their inherent hardware limitations,...

DEVICE EXPERIMENT In the first scenario, we are interested in determining whether and how effective mobile devices can spot the embedded beacons. To this end, we consider five Android devices, namely two *Asus Nexus 7*, an *LG-P880*, a *Motorola Moto G 2*, and a *Fairphone 1*, which each run a frequency analysis to spot anomalies in the ultrasonic range. The devices are exposed to the prepared video files, containing embedded beacons of varying frequency ranges and sound levels, such that a detection rate can be measured over multiple experimental runs.

The results of this experiment are presented in Figure 13a, where the average detection performance of all devices on 10 repetitions



(a) Noise robustness

(b) Device performance

Figure 13: Results for the device experiment. Figure (a) presents the detection performance vs. signal-to-noise ratio for different frequency bands and (b) the detection performance for different frequency bands and mobile devices.

is plotted against different signal-to-noise ratios (SNR). The SNR describes the sound level of the audio beacon compared to the sound level of the commercial, that is, the SNR increases when amplifying the audio beacon. In particular, an increase of the SNR by 6 dB corresponds to an amplification of the audio beacon by a factor of 2.

...mobile devices usually detect ultrasonic beacons reliably.

We observe that the devices are able to reliably detect the audio beacons even at very low SNRs. Starting from an SNR of -5 dB almost all beacons are correctly identified on both frequency bands. However, we notice a variance in the success rate among the different devices. Figure 13b presents the detection performance for each of the devices and frequency bands. While some devices, such as the Fairphone, have problems in detecting audio beacons close to the audible frequency range, the reverse holds true for one of the Nexus 7 tablets, which does not accurately detect audio beacons at 20 kHz.

As in the case of the two Nexus 7 devices, it is likely that frequency response patterns of the built-in microphones vary depending on the particular model and device, thus having an influence on the detection performance. Moreover, since our audio analysis runs as a background process, the performance may also depend on the current load on the device and timing of running processes. Nonetheless, all devices attain a detection rate of at least 60%, which is sufficient to spot audio beacons if multiple repetitions are embedded in sound.

In contrast to most mobile devices,...

USER EXPERIMENT In the second scenario, we ask 20 human subjects between the age of 20 and 54 to watch in total 10 minutes of videos. Some contain audio beacons at a frequency of 18 kHz in order to cover the lower end of the near-ultrasonic range. The beacons are embedded at various spots with different loudness levels ranging from 0 to 18 dB and the participants are asked to note down when they perceive a beacon in the audio.

...humans often have problems perceiving ultrasonic beacons.

None of the human subjects is able to spot the embedded beacons reliably even at the highest loudness level, although the frequency of 18 kHz lies within the age-dependent audible range and the participants are aware of the presence of audio beacons in the video clips. Two participants at the age of 23 and 27 are able to spot 17 and 6 beacons, respectively, from a total of 26 embedded beacons. Moreover, six participants state that they have perceived some anomalies in the signal. However, only few of these are indeed audio beacons. On the contrary, all participants are able to identify the beacons at the highest sound level without background sound. The reason for this discrepancy is that the human ear masks the tone in the presence of nearby frequencies and sounds. This effect is well-known and exploited in audio compression formats like MP3 and AAC which apply psychoacoustic models to lower the used transmission rate.

In summary, although our participants are aware of the audio beacons, they had considerable problems to identify the audio beacons reliably. The beacons are mainly perceived as an usual anomaly in sound. Hence, if not aware, a user might not even notice the ultrasonic signals. At the same time, different mobile devices already successfully tracked the signal at a SNR of -5 dB. In the end, our experiment confirms the technical feasibility to transmit ultrasonic beacons to a mobile device covertly, but also spots the limitations of this side channel.

3.6.2 *Audio Beacons in the Wild*

The previous experiment demonstrates that ultrasonic side channels are technically well realizable. In the next step, we explore whether this new form of tracking is already employed in practice.

Country	# TV channels	Size
United States	7	25h
Germany	5	24h
Spain	6	23h
Austria	3	21h
United Kingdom	2	16h
Philippines	5	16h
India	10	15h

Table 1: Dataset from TV streaming analysis.

Regarding Lisnr, we can spot audio beacons in recordings from the web that corresponds to events where Lisnr also participated. It shows that this technology is actively deployed, but rather at specific events, yet. We thus also investigate Shopkick that appears to be more widespread. To this end, we record audio in 35 stores in two European cities and detect an ultrasonic signal from Shopkick at four stores. Although we acknowledge that the user starts the Shopkick application intentionally, our findings underline the active distribution of ultrasonic tracking in the daily life.

We are able to spot ultrasonic signals in European stores...

The last question is whether TV streams contain ultrasonic beacons, especially from SilverPush. In fact, we have no information when, how, and where these beacons are transmitted in TV. Our search is thus close to finding a needle in a haystack. Consequently, we conduct a broad search across different countries and TV channels, rather than focusing on a specific scenario.

...and various media...

In particular, we record TV streams retrieved over the Internet from 7 different countries, where we focus on channels presenting a lot of commercials. Table 1 summarizes the number of TV channels and the total duration of analyzed audio signals per country. We need to note that the quality of the transmitted audio streams differs considerably between the recorded channels. While we are generally able to retrieve audio with a sufficient sampling rate between 40 and 48 kHz, the channels make use of different compression settings that potentially filter out inaudible high frequencies (see Section 3.7).

We analyze the recorded data, comprising almost 6 days of audio, with our standalone detection tool presented in Section 3.4.2. Although our tool is capable of detecting ultrasonic beacons at arbitrary frequencies between 18 and 20 kHz, we do not find any indications of such beacons in the recorded data, leaving us with a negative result. On the one hand, it seems that ultrasonic device tracking is not used in the considered TV channels; on the other hand, we cannot rule out that the beacons have been initially present but later removed due to compression for Internet streaming. In addition, we also visited the global, Indian, and Philippine Top 500 Alexa websites and recorded

...but not in television streams or websites.

their audio output to spot ultrasound. Similar to TV streams, we do not find any indications of ultrasonic beacons.

3.6.3 *Applications in the Wild*

Our Lisnr and Shopkick findings emphasize their active deployment, but we cannot quantify their distribution on the receiving side yet. In consequence, we would like to determine the distributions of Lisnr, Shopkick, and SilverPush. To this end, we focus on the landscape of Android applications and apply the static detection method presented in Section 3.4.1 to search for Lisnr, Shopkick, and SilverPush implementations in the wild.

When scanning applications for characteristic functions,...

In particular, we retrieve all Android applications submitted to the VirusTotal service in the end of December 2015. In total, we obtain a dataset of 1,320,822 applications, covering numerous benign as well as malicious samples and a total volume of over 8 Terabytes. We then apply our detection tool to scan for applications that contain code fragments similar to our initial 21 SilverPush samples as well as 4 Lisnr samples we identified during the research. Finally, we scan for similar code fragments found in different versions of the official Shopkick application.

Within the 1,320,822 Android applications, our scan yields 2 and 1 samples with functionalities of Lisnr and Shopkick, respectively. These samples are either applications that have been released by these companies themselves or by other companies officially collaborating with Shopkick or Lisnr. The user is thus aware of the deployed technology and needs to start the audio analysis manually.

On the other hand, our scan returns 39 unique SilverPush matches within our Android application dataset. We manually verify that each of these matches is indeed an instance of the SilverPush implementation embedded into applications from India and the Philippines. Table 2 lists five representative applications from our dataset along with their developer and number of downloads as reported by the Google Play Store.

...we find 234 apps potentially listening in the background.

The download numbers are considerable: Two applications have between 1 and 5 Million downloads, while the other three have about 50,000 to 500,000 downloads. It becomes evident that SilverPush has already been deployed in real-world applications. While in April 2015 only six instances have been known, our experiment unveils another 39 installations. Moreover, with the help of VirusTotal we have been able to identify further instances, reaching a total of 234 samples in January 2017. These additional samples have been identified by searching for virus labels containing the term “SilverPush” and then eliminating false positives using our detection tools. Based on this strategy we obtain 244 applications, where 10 samples are false pos-

Application Name	Version	Downloads
100000+ SMS Messages	2.4	1,000,000 – 5,000,000
McDo Philippines	1.4.27	100,000 – 500,000
Krispy Kreme Philippines	1.9	100,000 – 500,000
Pinoy Henyo	4.0	1,000,000 – 5,000,000
Civil Service Reviewer Free	1.1	50,000 – 100,000

Table 2: Third-party applications with SilverPush functionality.

itives that do not contain actual functionality but just strings related to the SilverPush implementation.

Our analysis provides us with two important insights regarding SilverPush: First, the number of mobile applications containing the SDK has grown during our study. Second, the applications have not only been downloaded a few hundred times, but some of them have possibly been installed by thousands of people. Even if the audio beacons are not embedded in actual TV commercials, our findings indicate that the SilverPush SDK has been deployed in a large number of applications between 2015 and 2017.

3.7 DISCUSSION

During the analysis and evaluation of the ultrasonic tracking technologies, we have gained insights into their capabilities but also spotted some limitations. In this section we therefore discuss requirements that have to be satisfied in order to allow the tracking to work properly. Furthermore, we discuss countermeasures to alleviate this new privacy threat.

3.7.1 *Limits and Challenges*

Although we are able to verify the feasibility of ultrasonic side channel communication under realistic conditions throughout our empirical study, we have experienced several issues which may impede a successful communication. In particular, there exist a bunch of challenges on the sender and the receiver side, which have to be considered in order to allow an inaudible communication between the devices.

BANDWIDTH RESTRICTIONS When analyzing the frequency spectra of the TV channels recorded for our analysis, we find that several of them are cut off at a frequency higher than 18 kHz and can thus not contain any audio beacons. Figure 14 depicts, for instance, a typical TV signal received via DVB-T. The spectrum of the signal

Common compression algorithms...

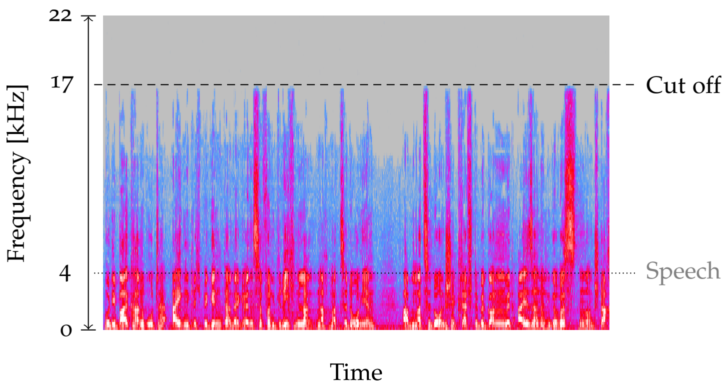


Figure 14: Spectrogram of DVB-T recording. Note that audio frequencies above 17 kHz are cut off.

clearly shows the absence of any frequencies above 17 kHz. In principle, common video broadcasting standards like ASCT, DVB and ISDB allow sampling rates of less than 40 kHz, which would remove the desired frequency band. However, since the sampling frequency has been high enough in the recorded data, the low-pass filtering of the signal most probably results from the compression applied to the audio signal.

...remove high frequencies from audio signals.

Several audio compression algorithms are capable of removing frequencies that are inaudible, such as MP3 and AAC. As both formats use a psychoacoustical model and offer various options, it thus cannot contain any audio beacons. Figure 15 gives a tendency for MP3 and AAC by using a fixed bitrate as indicator of quality. In particular, we compressed a stereo music track with embedded high-frequency tones with `ffmpeg`'s built-in MP3 and AAC encoder and tried to detect these tones after compression. As an example, for MP3 a bitrate of 320 kb/s allows frequencies up to 20 kHz, while a common bitrate of 128 kb/s removes ultrasonic frequencies entirely from the signal.

Consequently, it becomes unlikely to find ultrasonic beacons...

Furthermore, we have also uploaded videos with embedded audio beacons to YouTube to test whether high-frequency tones are preserved. YouTube always encodes an uploaded video to ensure that it can be played with different devices in different quality levels. In our tests, the highest quality of a stereo signal reaches up to 18.5 kHz, while a mono signal conveys audio beacons in the full frequency spectrum between 18 and 20 kHz. As a consequence, ultrasonic side channels are currently only possible if a mono recording is uploaded to the YouTube platform.

Finally, a legitimate question arises why an adversary does not simply use the audible frequency range. The device could perform sound or speech recognition to identify the TV viewing habits or the location. The music recognition service Shazam already provides additional information about a brand or product based on the identified sound [115]. There are, however, two problems. First, Shazam's recognition algorithm requires a full frequency analysis through a Fourier Transform [199]. This analysis is computationally more demanding than the beacon detection through the simple Goertzel algorithm (see Section 3.5). In consequence, a persistent background monitoring drains the battery of mobile devices more quickly. Second, an audio beacon can carry additional information about the location or the played media that, in turn, facilitates tracking (see Section 3.2).

...in compressed media, such as YouTube videos.

SOFTWARE RESTRICTIONS Another restriction arises from the new permission model introduced by Android 6 [63]. In contrast to previous Android versions, the new system differentiates between normal and dangerous permissions and the user has to grant dangerous permissions at run-time. The set of dangerous permissions also comprises permissions like `RECORD_AUDIO` and `READ_PHONE_STATE`, which are crucial for SilverPush's functionality. It should thus raise the users' doubts when an application, for example, unexpectedly wants to record audio.

Android 6 (and above) provides further protection...

Although this new permission model increases security theoretically, there are three practical problems: First, when an application targets an SDK smaller than 23, the old permission model is used again where the user is only asked at installation time. Second, famous applications can carry the ultrasonic tracking functionality. It is unclear if a user questions the necessity of a dangerous permission in this case. Therefore, the new permission model might alleviate the risk, but can unfortunately not entirely prevent unauthorized tracking. Thirdly, due to the fragmentation of Android, a large number of devices still run Android versions below 6.0 and are thus not protected (see Chapter 2). That is, even though Android 6 has already been released in 2015, roughly 30% of all Android devices still run an Android version lower than 6.0 in October 2018 [125].

...but still does not run on all devices.

HARDWARE LIMITATIONS Finally, we notice that various limitations are introduced by the built-in microphones and speakers in common hardware. As we have already discussed in Section 3.6.1, the detection performance differs between several devices. Moreover, although the SilverPush patent, for instance, also considers inaudible frequencies in the infrasound range below 20 Hz, it is unlikely that such frequencies can actually be used, since they require specialized hardware to send and receive sound. Consequently, only the considered range of 18 to 20 kHz is a realistic and technically feasible range for transmission of inaudible beacons.

3.7.2 *Countermeasures*

Based on the different challenges for transmission, we identify defenses to limit the tracking via ultrasonic beacons. Obviously, a sim-

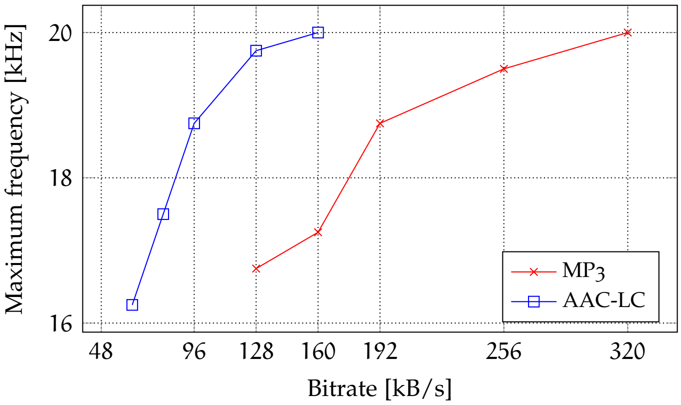


Figure 15: Frequency bandwidth of MP3 and AAC.

ple yet effective defense strategy is to filter out frequencies above 18 kHz in the transmitted audio signal, e.g., in the radio or TV device. However, manipulating either the hardware or software of these devices is not feasible for regular users. Moreover, the emitting sender is not always within the user's control, for example during location tracking.

Therefore, practical countermeasures should address the mobile device. If the device is not secretly listening, a transmitted audio beacon is harmless. Hence, we consider the following countermeasures for the Android operating system:

DETECTION OF IMPLEMENTATIONS An option is to scan for applications with known ultrasonic side channels functionality. Our detection tool presented in Section 3.4.1 might provide a good start for the development of a corresponding defense. Similarly to a virus scanner, such a detection can be applied locally on the device as well as directly on a market place. As our approach builds on static code analysis, however, detecting the corresponding functionality can be hindered by obfuscating the respective implementations. Moreover, the detection tool requires manual effort to identify the characteristic functions for each library separately.

NOTIFICATION Just as for Bluetooth or Wifi, a more fine-grained control of the audio recording is likely the best strategy for limiting the impact of ultrasonic side channels. A combination of user notifications and a status in the pull down menu can inform the user when a recording is taking place and lets her detect unwanted activities.

3.7.3 *Limitations*

Our study deals with a real-world threat and underlying technical problems. It is thus naturally subject to certain limitations, which we briefly discuss in the following.

First of all, our study could not reveal any indications of ultrasonic sounds in TV streams. However, whether this finding is to be interpreted as a negative or positive result is unclear. While we designed our study with great care and as broad as possible, it is not unlikely that we simply missed audio beacons due to monitoring TV channels at the wrong time or place. Moreover, the beacons could have been obfuscated using code spread spectrum techniques. In this case, our detection method from Section 3.4.2 would have missed these signals. However, we could not find any indications throughout our analysis that SilverPush uses this kind of technique. In addition, the detection of Lisnr or Shopkick beacons makes it rather unlikely that we missed beacons in TV streams due to the high similarity of SilverPush to Lisnr or Shopkick.

Second, although our detection tool provides an efficient way to identify the functionality of SilverPush, Lisnr, and Shopkick, it relies on the knowledge of currently used code. Changing the code basis would possibly prevent a detection, but it seems unlikely that the developers permanently adapted their code to avoid detection in this case. However, since it is not uncommon for malware authors to change their code base regularly, the detection tool is not suitable for malware detection in general. In the next chapter, we therefore propose another learning-based method which is more generic, thus allowing the detection of different variants of malware.

3.7.4 *Conclusion and Outlook*

Since we could not find indications for ultrasonic beacons in TV, the question arises whether and how this technology is still used. Interestingly, the developers of SilverPush contacted us in May 2017, shortly after our findings had been published. According to them, the deployment of the SilverPush library has been stopped in the end of 2015, and it has only been used by around 10 applications in total.

SilverPush probably stopped ultrasound tracking...

However, the timestamp analysis of our collected data shows that there are also applications from 2016 containing the SDK. Moreover, we still detected an additional application in May 2017 that contains the SDK and has been released in March 2017. Figure 16 depicts the distribution of the collected Android applications over time. Note that the plot does not show all collected samples but only those 179 applications for which we could extract the respective timestamps.

Overall, there are two observations which support the statement of the SilverPush developers regarding the end of the malware's active

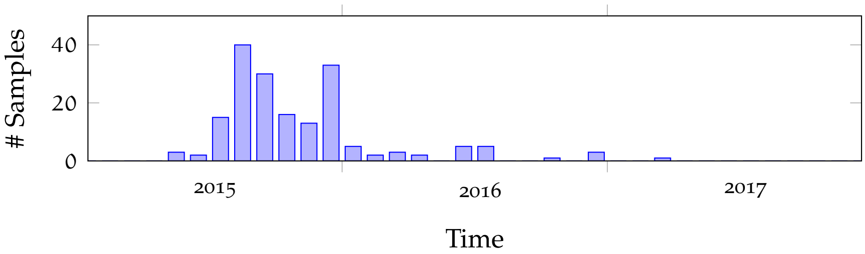


Figure 16: Time distribution of Silverpush samples.

deployment. First, the majority of applications has been created in 2015. Second, the number of new applications has dropped in January 2016. Still, it remains unclear why we have been able to identify further applications containing the SilverPush SDK and, moreover, why the number differs significantly from the one reported by the developers. Possibly, the SDK has been repackaged without the knowledge of the SilverPush developers. Unfortunately, the developers did not provide us with a proper answer to this question.

...and its apps have been removed from Google Play.

Google contacted us roughly at the same time as the SilverPush developers. Using our findings, Google was able to remove all remaining applications from their GooglePlay store. Moreover, they could even identify further samples containing the SilverPush SDK in the store. Overall, it should therefore be unlikely that SilverPush still poses a threat to users' privacy—at least when installing applications from GooglePlay. However, as we have broadly discussed throughout this chapter, the underlying technology works reliably and can thus still be misused by malware authors.

Unfortunately, new technologies raise similar privacy concerns.

Furthermore, other technologies can be used to track users with their mobile devices. For instance, audio fingerprints, as used by Shazam [199], were initially considered too computationally expensive to be computed on mobile devices in the background (see Section 3.7.1). However, the technology becomes more attractive as the computing power of these devices increases. In particular, the *New York Times* reported in late 2017 about the SDK *Alphonso*, which exhibits alarming parallels to the SilverPush SDK [192].

Nonetheless, during a rough analysis of five applications containing the library, we found no evidence suggesting the exposure of sensitive data, such as the phone number. Still, the case of Alphonso once again demonstrates how close legitimate and malicious use of the tracking functionality can be. Additionally, it shows that tracking based on audio fingerprints can also impact the privacy of mobile device users.

3.8 RELATED WORK

Ultrasonic cross-device tracking touches different areas of security and privacy. We review related approaches and concepts throughout this section.

MOBILE DEVICE FINGERPRINTING While classic web browser fingerprinting is characterized by a vivid area of research in the last years [152], there is only a small number of works that examine mobile devices. A straightforward adoption of browser fingerprinting methods is not possible due the highly standardized nature of mobile devices [105]. Nevertheless, Hupperich et al. recently demonstrated the feasibility to fingerprint the mobile web browser as well [105]. Furthermore, Kurtz et al. showed how personalized device information such as the list of installed apps or the most-played music songs also provide an effective way to fingerprint an iOS device without any user permission [123].

Another approach is to leverage unique physical characteristics from device sensors such as the camera [88], the accelerometer [31] as well as the microphone and speakers [8, 31, 55, 225] for fingerprinting. Although the resulting hardware fingerprints are highly unique due to their random character, their computation is expensive and requires access to the sensor for a certain amount of time.

While these works aim at fingerprinting one device, the studied ultrasonic side channel enables an adversary to track a user across her multiple devices, her visited locations as well as to obtain her media usage.

COVERT ACOUSTIC COMMUNICATION Different researchers have demonstrated the feasibility to communicate covertly in the ultrasonic range with just standard loudspeakers and microphones [60, 64, 102, 126, 218]. The considered scenarios, however, differ from our study. First, these authors mainly focus on bypassing security mechanisms and bridging the "air gap" between isolated computer systems. Second, the ultrasonic communication is usually conducted in a quiet environment, whereas ultrasonic user tracking demands a high robustness that can compensate different environmental noise.

3.9 CHAPTER SUMMARY

In this chapter we have discussed the privacy threats of ultrasonic tracking for mobile device users. Using this technology, malicious applications are able to monitor users' TV viewing habits, track their visited locations, and deduce their other devices. Furthermore, even side channel attacks on Bitcoin or Tor users become possible. In the end, malware authors might obtain a detailed, comprehensive user profile using solely the device's microphone.

By analyzing prominent examples of commercial tracking technologies, we gained insights about their current state and the underlying communication concepts. The case of SilverPush emphasizes that the step between spying and legitimately tracking is rather small. While the technology behind SilverPush shares essential similarities with legitimate solutions, it mainly differs in that the user is unaware of the tracking functionality performed by SilverPush in the background.

Throughout our empirical study, we confirm that audio beacons can be embedded in sound, such that mobile devices spot them with high accuracy while humans do not perceive the ultrasonic signals consciously. Moreover, we spot ultrasonic beacons from Lisnr in music and Shopkick beacons in 4 of 35 stores in two European cities, proving that the technology is already actively used by companies in the wild. While we do not find indication of ultrasonic tracking in TV media, we are able to detect the SilverPush SDK in 234 Android applications, which have been collected between December 2015 and January 2017.

As a reaction to our findings, Google removed all remaining applications containing the SilverPush SDK from GooglePlay in May 2017. However, even though SilverPush might not pose a privacy threat anymore, the underlying technology can still be misused by malware authors in the future. While the detection tool used for this study can in principle be extended to also detect other characteristic code regions, it requires manual effort to identify and extract them for each new malware family individually.

Instead, we present a new learning-based approach in the next chapter, which automatically identifies characteristics of arbitrary malware families, thus allowing the detection of unknown malware instances without needing to craft the required signatures manually.

4 LEARNING-BASED MALWARE DETECTION

In the previous chapter, we discussed a sophisticated malware which uses an ultrasonic side channel to transmit information imperceptibly, without the users' knowledge. To detect this malware family, we built an efficient tool that allows scanning for members of it in a large number of applications. However, before being able to derive proper signatures, the detection tool first requires the malware analyst to manually identify characteristic code regions in some initial samples of this family. Consequently, the approach is only feasible when trying to detect samples of a small number of malware families, but it is not suitable for detecting Android malware in general.

We propose a learning-based method...

Therefore, we are looking for a more general solution that enables us to detect Android malware without the need for much manual effort. To this end, we consider machine learning techniques, as these have already been successfully applied for intrusion and malware detection before [e.g., 156, 168, 173]. However, in contrast to previous approaches, we want to perform the classification directly on the device, without requiring the user to send applications to an external server. This constraint has mainly two reasons. First, uploading applications to an external server is not always possible and can even lead to high costs for the user. Second, requiring users to send their applications to an external analysis server theoretically enables the operator of this server to derive sensitive information on device usage. This can possibly have severe privacy implications.

...which allows the detection of Android malware...

To tackle the described challenges, we present Drebin, a lightweight method for Android malware detection that infers detection patterns automatically and enables identifying malware directly on the smartphone. Drebin performs a broad static analysis, gathering as many features from an application's code and manifest as possible. These features are organized in sets of strings (such as permissions, API calls, and network addresses) and embedded in a joint vector space. As an example, an application sending premium SMS messages is cast to a specific region in the vector space associated with the corresponding permissions, intents, and API calls. This geometric representation allows Drebin to identify combinations and patterns of features indicative of malware automatically using machine learning techniques. For each detected application the respective patterns can be extracted, mapped to meaningful descriptions and then provided to the user as explanation for the detection. Aside from detection, Drebin can thus also provide insights into the identified Android malware samples.

...directly on the mobile device.

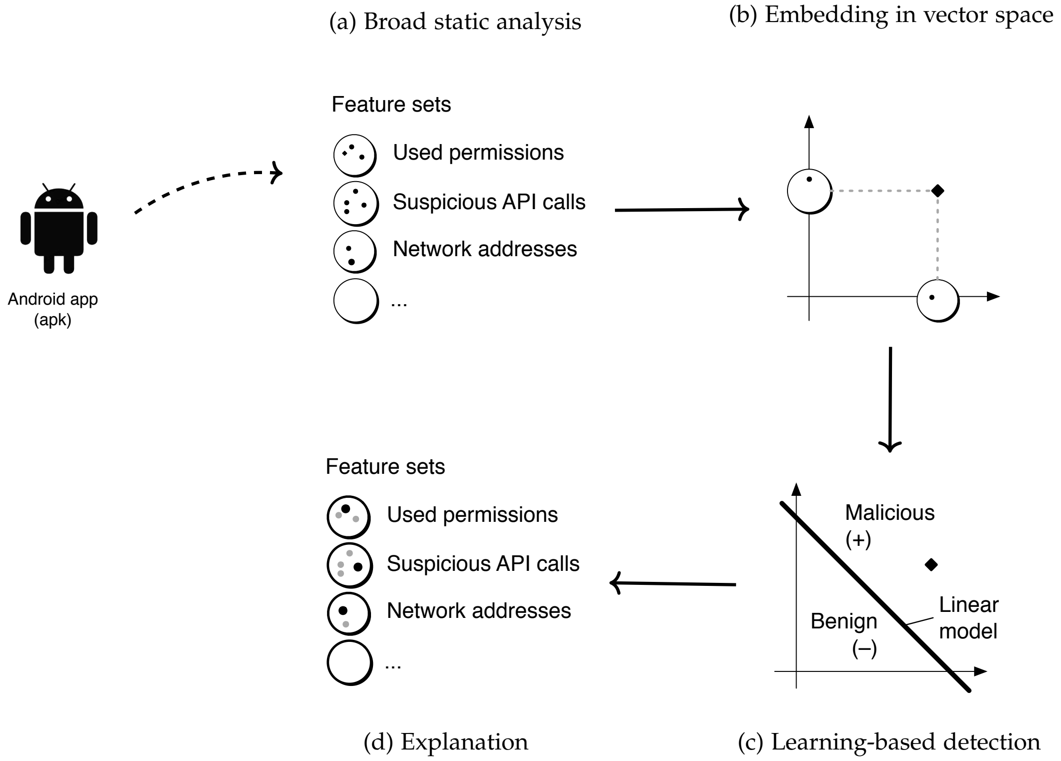


Figure 17: Schematic depiction of the analysis steps performed by Drebin.

In summary, we make the following contributions to the detection of Android malware in this paper:

- *Effective detection.* We introduce a method combining static analysis and machine learning that is capable of identifying Android malware with high accuracy and few false alarms, independent of manually crafted detection patterns.

- *Explainable results.* The proposed method provides an explainable detection. Patterns of features indicative of a detected malware instance can be traced back from the vector space and provide insights into the detection process.

- *Lightweight analysis.* For efficiency we apply linear-time analysis and learning techniques that enable detecting malware on the smartphone as well as analyzing large sets of applications in reasonable time.

We provide an extensive evaluation of Drebin *in the next chapters.*

In this chapter, we describe the details of our approach. An extensive evaluation of its detection capabilities on actual Android malware is provided in the next chapter. During the assessment, we also point to some limitations of the method, which might be subject to further research. Note, for instance, that Drebin builds on concepts of static analysis and thus cannot rule out the presence of obfuscated or dynamically loaded malware on mobile devices. Due to the broad analysis of features, however, the method raises the bar for attackers to infect smartphones with malicious applications and strengthens the security of the Android platform, as demonstrated by our evaluation in Chapter 5.

4.1 METHODOLOGY

To detect malicious software on a mobile device, DREBIN requires a comprehensive yet lightweight representation of applications that allows determining typical indications of malicious activity. For this purpose, it employs a broad static analysis which extracts feature sets from different sources and analyzes these in an expressive vector space. This process is illustrated in Figure 17 and briefly outlined in the following:

Our method performs a broad static analysis...

a) *Broad static analysis.* In the first step, DREBIN statically inspects a given Android application and extracts different feature sets from the application's manifest and dex code (Section 4.1.1).

b) *Embedding in vector space.* The extracted feature sets are then mapped to a joint vector space, where patterns and combinations of the features can be analyzed geometrically (Section 4.1.2).

c) *Learning-based detection.* The embedding of the feature sets enables us to identify malware using efficient techniques of machine learning, such as linear Support Vector Machines (Section 4.1.3).

d) *Explanation.* In the last step, features contributing to the detection of a malicious application are identified and presented to the user or analyst for explaining the detection process (Section 4.1.4).

In the following sections, we discuss these four steps in more detail and provide the necessary technical background of the analysis.

4.1.1 *Static Analysis of Applications*

As the first step, DREBIN performs a lightweight static analysis of a given Android application. Due to the hardware limitations of mobile devices, the static extraction of features needs to run in a constrained environment and still complete its extraction process within a reasonable amount of time. If the analysis takes too long, the user might skip the ongoing process and refuse the overall method. Accordingly, it becomes essential to select features which can be extracted efficiently.

...and extract features from 8 different sets.

We thus focus on the manifest and the disassembled dex code of the application (see Chapter 2), which both can be obtained by a linear sweep over the application's content. We provide details on the implementation later in this chapter. To allow for a generic and extensible analysis, we represent all extracted features as sets of strings, such as permissions, intent filters, and API calls. In particular, we extract the following 8 sets of strings.

This includes features from the manifest file...

FEATURE SETS FROM THE MANIFEST Every Android application contains a manifest file called *AndroidManifest.xml*, which provides data supporting the installation and later execution of the application. During the analysis process, DREBIN extracts the following feature sets from this file:

S1 *Hardware components:* This first feature set contains requested hardware components. If an application requires access to the camera, touchscreen, or the GPS module of the mobile device, these features should be specified in the manifest file. Requesting access to specific hardware clearly has security implications, as the use of specific combinations of hardware often reflects harmful behavior. An application which has access to GPS and network modules can, for instance, collect location data and send it to an attacker over the network.

...like the requested permissions of an application.

S2 *Requested permissions:* One of the most essential security mechanisms introduced in Android is the permission system. Permissions are actively granted by the user at installation time and allow an application to access security-relevant resources. As shown by previous work [71, 173], malicious software tends to request certain permissions more often than innocuous applications. For example, a high percentage of current malware sends premium SMS messages and thus requests the `SEND_SMS` permission. We therefore gather all permissions listed in the manifest in a feature set.

S3 *App components:* There exist four different types of components in an application, each defining different interfaces to the system: *activities*, *services*, *content providers*, and *broadcast receivers*. Every application can declare several components of each type in the manifest. The names of these components are also collected in a feature set, as the names may help to identify well-known components of malware. For example, several variants of the so-called DroidKungFu family share the name of particular services [see 109–111].

S4 *Filtered intents:* Inter-process and intra-process communication on Android is mainly performed through *intents*: passive data structures exchanged as asynchronous messages and allowing information about events to be shared between different components and applications. We collect all intents listed in the manifest as another feature set, as malware often listens to specific intents. A typical example of an intent message involved in malware is BOOT_COMPLETED, which is used to trigger malicious activity directly after rebooting the smartphone.

FEATURE SETS FROM DISASSEMBLED CODE Android applications are developed in Java and compiled into optimized bytecode for the Dalvik virtual machine. This bytecode can be efficiently disassembled and provides DREBIN with information about API calls and data used in an application. We use this information to construct the following feature sets:

Additional features are gathered from the dexcode...

S5 *Restricted API calls:* The Android permission system restricts access to a series of critical API calls. Our method searches for the occurrence of these calls in the disassembled code to gain a deeper understanding of the functionality of an application. A particular case, revealing malicious behavior, is the use of restricted API calls for which the required permissions have not been requested. This may indicate that the malware is using root exploits in order to surpass the limitations imposed by the Android platform.

S6 *Used permissions:* The complete set of calls extracted in S5 is used as the ground for determining the subset of permissions that are both requested and actually used. For this purpose, we implement the method introduced by Felt et al. [77] to match API calls and permissions. In contrast to S5, this feature set provides a more general view on the behavior of an application, as multiple API calls can be protected by a single permission (e.g., sendMultipartTextMessage() and sendTextMessage() both require that the SEND_SMS permission is granted to an application).

S7 *Suspicious API calls:* Certain API calls allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. As these calls can especially lead to malicious behavior, they are extracted and gathered in a separate feature set. This includes, for instance, the usage of SMS functionality. Moreover, we also consider functions which are known to be used by malware for obfuscation, such as the use of encryption and reflection calls. Table 3 lists the collected API call types along with some common examples. Note that the list should ideally be updated regularly to keep up with the evolution of malicious software (see Section 5.2.5).

...including API calls known to be frequently used by Android malware.

S8 *Network addresses:* Malware regularly establishes network connections to retrieve commands or exfiltrate data collected from the device. Therefore, all IP addresses, hostnames, and URLs found in the disassembled code are included in the last set of features. Some of these addresses might be involved in botnets

Suspicious behavior	**Indicative API calls**

Access of sensitive data	`getDeviceId()`
	`getSubscriberId()`
	`...`

Network communication	`execHttpRequest()`
	`setWifiEnabled()`
	`...`

Usage of SMS functionality	`getMessageBody()`
	`sendTextMessage()`
	`...`

Execution of external code	`Runtime.exec()`
	`DexClassLoader.loadClass()`
	`...`

Possible obfuscation	`Cipher.getInstance()`
	`reflect.Method.invoke()`
	`...`

Table 3: Examples of API calls considered as suspicious.

and thus present in several malware samples, which can help to improve the learning of detection patterns.

4.1.2 *Embedding in Vector Space*

As most learning models operate on numerical vectors...

Malicious activity is usually reflected in specific patterns and combinations of the extracted features. For example, a malware sending premium SMS messages might contain the permission `SEND_SMS` in set S_2, and the hardware component `android.hardware.telephony` in set S_1. Ideally, we would like to formulate Boolean expressions that capture these dependencies between features and return true if a malware is detected. However, due to the large amount of data, it is infeasible to infer these Boolean expressions from real-world data in practice.

As a remedy, we aim at capturing the dependencies between features using concepts from machine learning. As most learning methods operate on numerical vectors, we first need to map the extracted feature sets to a vector space. To this end, we define a joint set S that comprises all observable strings contained in the 8 feature sets

$$S := S_1 \cup S_2 \cup \cdots \cup S_8. \tag{15}$$

We ensure that elements of different sets do not collide by adding a unique prefix to all strings in each feature set. In our evaluation, the set S roughly contains up to 3.2 million different features (see Section 5.2.5).

Using the set S, we define an $|S|$-dimensional vector space, where each dimension is either 0 or 1. An application z is mapped to this space by constructing a vector $\varphi(z)$, such that for each feature s extracted from z the respective dimension is set to 1 and all other dimensions are 0. Formally, this map φ can be defined for a set of applications $\mathcal{Z}$ as follows

$$\varphi : \mathcal{Z} \to \{0,1\}^{|S|}, \quad \varphi(z) \mapsto \big(I(z,s)\big)_{s \in S} \tag{16}$$

where the indicator function $I(z,s)$ is simply defined as

$$I(z,s) = \begin{cases} 1 & \text{if the application } z \text{ contains feature } s \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

Applications sharing similar features lie close to each other in this representation, whereas applications with mainly different features are separated by large distances. Moreover, directions in this space can be used to describe combinations of features and ultimately enable us to learn explainable detection models.

...we need to map the extracted feature sets to a vector space.

Let us, as an example, consider a malicious application that sends premium SMS messages and thus needs to request certain permissions and hardware components. A corresponding vector $\varphi(z)$ for this application looks like this:

$$\varphi(z) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{array}{l} \cdots \\ \left.\begin{array}{l}\texttt{HARDWARE::android.hardware.wifi} \\ \texttt{HARDWARE::android.hardware.telephony}\end{array}\right\} S_1 \\ \cdots \\ \left.\begin{array}{l}\texttt{REQ\_PERMISSION::SEND\_SMS} \\ \texttt{REQ\_PERMISSION::DELETE\_PACKAGES}\end{array}\right\} S_2 \\ \cdots \end{array}$$

At first glance, the map φ seems inappropriate for the lightweight analysis of applications, as it embeds data into a high-dimensional vector space. Fortunately, the number of features extracted from an application increases linearly with its size. That is, an application z containing m bytes of code and data includes at most m feature strings. As a consequence, only m dimensions are non-zero in the vector $\varphi(z)$—irrespective of the dimension of the vector space. It thus suffices to only store the features extracted from an application for sparsely representing the vector $\varphi(z)$, for example, using hash tables [47] or Bloom filters [30].

4.1.3 *Learning-based Detection*

In the third step, we apply machine learning techniques for automatically learning a separation between malicious and benign applications. The application of machine learning spares us from manually constructing detection rules for the extracted features.

A linear SVM learns a hyperplane...

While several learning methods can be applied to learn a separation between two classes, only a few methods are capable of producing an efficient and explainable detection model. We consider linear *Support Vector Machines (SVMs)* for this task. In short, a linear SVM determines a hyperplane that separates the training points of two classes in feature space with maximal margin. In our case, one of these classes is associated with malware, whereas the other class corresponds to benign applications. An unknown application is classified by mapping it to the vector space and determining whether it falls on the malicious (+) or benign (−) side of the hyperplane. Note that a detailed description of the algorithm and its mathematical background is described in Chapter 2.

To stay compliant with the previously discussed notation, the detection model of a linear SVM corresponds to a vector $\boldsymbol{w} \in \mathbb{R}^{|S|}$, which is perpendicular to the hyperplane and specifies its direction in feature space. The corresponding detection function $f : \mathcal{Z} \to \mathbb{R}$ is then given by the term

$$f(z) = \langle \varphi(z), \boldsymbol{w} \rangle = \sum_{s \in S} I(z, s) \cdot w_s \qquad (18)$$

and returns the orientation of $\varphi(z)$ with respect to $\boldsymbol{w}$. That is, $f(z) > b$ indicates malicious activity, while $f(z) \leqslant b$ corresponds to benign applications for a given threshold b.

...that separates both classes with maximum margin.

To compute the function f efficiently, we again exploit the sparse representation of the map φ. Given an application z, we know that only features extracted from z have non-zero entries in $\varphi(z)$. All other dimensions are zero and do not contribute to the computation of $f(z)$. Hence, we can simplify the detection function f as follows

$$f(z) = \sum_{s \in S} I(z, s) \cdot w_s = \sum_{s \in z} w_s. \qquad (19)$$

Instead of an involved learning model, we finally arrive at a simple sum that can be efficiently computed by just adding the weight w_s for each feature s in an application z. This formulation enables us to apply a learned detection model on a smartphone and also allows us to explain results obtained by the Support Vector Machine.

4.1.4 *Explanation*

In practice, a detection system must not only indicate malicious activity, but also provide explanations for its detection results. It is a common shortcoming of learning-based approaches that they are black-box methods [182]. In the case of DREBIN, we address this problem and extend our learning-based detection, such that it can identify features of an application that contribute to a detection. Moreover, an explainable detection may also help researchers inspect patterns in malware and gain a deeper understanding of its functionality.

DREBIN provides explanations for its decisions.

By virtue of the simple detection function of the linear SVM, we are able to determine the contribution of each single feature s to the function $f(z)$. During the computation of $f(z)$, we just need to store the largest k weights w_s shifting the application to the malicious side of the hyperplane. Since each weight w_s is assigned to a certain feature s, it is then possible to explain why an application has been classified as malicious or not. This approach can be efficiently realized by maintaining the k largest weights w_s in a heap during the computation of the function $f(x)$ [47].

It extracts the most significant features...

After extracting the top k features by their weights, DREBIN automatically constructs sentences that describe the functionality underlying these features. To achieve this goal, we design sentence templates for each feature set which can be completed using the respective feature. Table 4 lists these templates. For features frequently observed in malware, such as the permission `SEND_SMS`, we provide individual descriptions.

Feature set	Explanation
S_1 Hardware features	`App uses %s feature %s.`
S_2 Requested permissions	`App requests permission to access %s.`
S_3 App components	`App contains suspicious component %s.`
S_4 Filtered intents	`Action is triggered by %s.`
S_5 Restricted API calls	`App calls function %s to access %s.`
S_6 Used permissions	`App uses permissions %s to access %s.`
S_7 Suspicious API calls	`App uses suspicious API call %s.`
S_8 Network addresses	`Communication with host %s.`

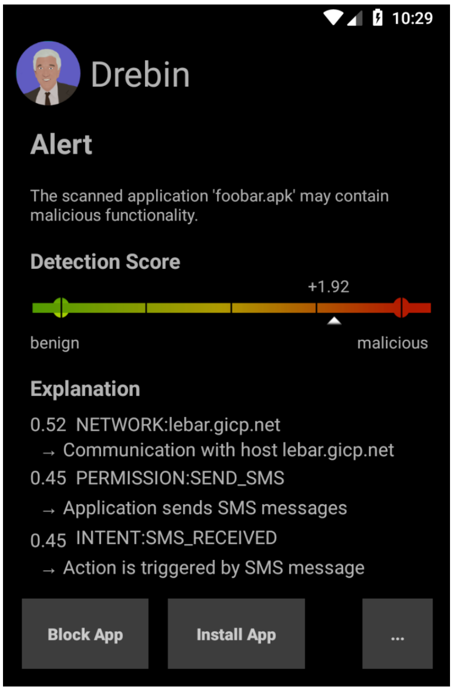
Table 4: Templates for explanation.

...and uses them to derive descriptions.

For most of the feature sets, the construction of sentences from the templates in Table 4 is straightforward. For example, for the hardware features we make use of their naming scheme to construct meaningful sentences. For instance, DREBIN presents the sentence *"App uses hardware feature camera."* to the user, if an application uses the `android.hardware.camera` feature.

Similarly, we provide explanations for requested and used permissions. The explanation for a permission can be derived from the Android documentation which provides proper descriptions—at least for all system permissions. We slightly modify these descriptions in order to present meaningful explanations to the user. However, due to the fact that application developers are able to define custom permissions we also provide a generic sentence which is presented to the user if no proper description exists. We follow the same approach for the restricted API calls that build on the use of certain permissions. For all other feature sets the templates are directly filled with either the feature's name or a corresponding placeholder.

An example of an explanation generated by Drebin is shown in Figure 18. The presented sample belongs to the GoldDream family. Drebin correctly identifies that the malware communicates with an external server and sends SMS messages. The application requests 16 permissions during the installation process. Many users ignore such long lists of permissions and thereby fall victim to this type of malware. In contrast to the conventional permission-based approach, Drebin draws the user's attention directly to relevant aspects that indicate malicious activity. Furthermore, Drebin presents a score to the user which tells him how confident the decision is. As a result, the user is able to decide whether the presented functionality matches his expectation or not.



Alert

The scanned application 'foobar.apk' may contain malicious functionality.

Detection Score

Explanation

0.52 NETWORK:lebar.gicp.net
→ Communication with host lebar.gicp.net
0.45 PERMISSION:SEND_SMS
→ Application sends SMS messages
0.45 INTENT:SMS_RECEIVED
→ Action is triggered by SMS message

Figure 18: Example of an explanation.

In addition to the benefit for the user, the generated explanations can also help researchers discover relevant patterns in common malware families. We discuss this aspect in more detail in Chapter 6.

4.2 DISCUSSION

While the presented method is already feasible when running on Desktop computers, it requires additional considerations to run it directly on a mobile device. In the following, we discuss several problems along with possible solutions to address the hardware limitations of smartphones. Finally, we give some details on the implementation we used for the evaluation of DREBIN in Chapter 5 and Chapter 6.

OFFLINE LEARNING In our implementation, we do not learn a detection model on the smartphone due to its computational costs. Moreover, a large data set is needed to train the classification model, which cannot be stored directly on the device. Instead, we train the Support Vector Machine offline on a dedicated system and only transfer the learned model w and threshold b to the smartphone for detecting Android malware.

The classifier is trained on an external server.

MODEL SIZE However, as the number of available applications used to train the classifier increases, so does the number of available features, and hence also the size of the resulting classification model. Although the SVM only selects a small fraction of the available features, the model will still exceed an acceptable size at some point. Fortunately, there exist two techniques which allow an effective reduction of the model size—*feature selection* and *feature hashing*. Note that both methods can even be combined to further reduce the size of the detection model.

The size of the resulting model can be reduced...

The first option to limit the model size is *feature hashing* [177]. This technique is related to the concept of Bloom filters [30]. In order to apply feature hashing, we can define a new mapping function $\widehat{\varphi} : \mathcal{Z} \to \{0, 1\}^N$ that maps an application z into an N-dimensional vector space, where $N \leqslant |S|$ restricts the model size. For this purpose, $\widehat{\varphi}$ uses a hash function $h : S \to \{1, \ldots, N\}$, which assigns a natural number to each feature string s. Besides the dimensionality reduction, the technique also allows mapping each application independently into the feature space. Note, however, that h is not an injective function and thus different features might get associated with the same dimension. Although h can be selected such that it minimizes the probability of such a collision, too many collisions occur if $|S|$ becomes too large. Consequently, no meaningful explanation can be derived from the model anymore. In this case, performing a feature selection poses a better option.

...using feature hashing...

When applying *feature selection* techniques, only the most discriminant features of the training data are picked to train the model. There exist different methods to select these features [98, 99, 120]. For instance, we examine the impact of two different methods for feature

...or feature selection.

selection in Chapter 6. While the first one selects the features with the largest weights in w, the second approach uses the concept of regularization for feature reduction (see Chapter 2). We will discuss both of them in detail in Chapter 6. Furthermore, we demonstrate how much the number of features can be reduced with these techniques.

IMPLEMENTATION To allow for efficient processing of applications directly on the device, we implement a prototype of DREBIN based on the tools *aapt (Android Asset Packaging Tool)* and *DexDump*. In particular, we use aapt to extract the feature sets S_1-S_4 from the manifest file and apply a modified version of DexDump for the extraction of the remaining feature sets. We evaluate the efficiency of this prototype implementation in Section 5.3.

For all other experiments in this thesis, we use a re-implementation of DREBIN based on a modified version of the *Androguard* [61] toolbox. While this implementation is slower than the original tool, it has the advantage that Androguard is a commonly used software and under constant development. Thus, it is compatible with newer Android versions which, for instance, provide new mappings between permissions and API calls [2, 18, 77]. Furthermore, it allows adding new features to DREBIN with little effort. Note that we slightly modified Androguard such that it can also inspect some (obfuscated) applications where otherwise the analysis usually fails. In addition, we use *GNU Parallel* [191] to compensate for the run-time overhead introduced by Androguard.

4.3 RELATED WORK

Several researchers have already proposed Android malware detection systems before DREBIN. In the next chapter, we compare our approach to some existing ones that can also be applied directly on the mobile device. For a better understanding of the discussions provided throughout the next chapter, we give a more detailed description of these approaches in the following. Afterward, we discuss further existing work on Android malware detection.

KIRIN uses 9 security rules to check for suspicious characteristics.

KIRIN This is one of the first approaches to Android malware detection and was already presented back in 2009 by Enck et al. [71]. At that time, the most recent Android version was 1.1, and only a few malicious applications were known. It was, however, already foreseeable that mobile malware might become a severe threat in the near future. As a result, the authors proposed a lightweight certification method for Android that allows the identification of potentially dangerous functionality in mobile applications. For this purpose, the method uses 9 distinct security rules, which mainly check whether an application requests suspicious combinations of permissions. For instance, it categorizes an application as potentially dangerous, if the app requests the permissions *RECEIVE_SMS* and *WRITE_SMS* at the same time. As in the case of Drebin, these checks can be performed directly on the phone.

Nonetheless, while Kirin is a very lightweight yet effective approach, it suffers from the drawback of requiring manual effort to derive its security rules. In contrast, Drebin infers these rules automatically. Consequently, it makes sense to perform a direct comparison of both detection approaches.

RCP In 2012, Sarma et al. [173] presented an approach to automatically infer risk signals based on *Rare Critical Permissions* (RCP) from a given set of Android applications. These risk signals, in turn, allow determining whether an unknown application requests a critical permission or a combination of critical permissions that are unusual to occur in legitimate apps. In total, the authors consider 26 permissions as critical which allow applications to access particularly sensitive or security-relevant functionalities of the device.

RCP checks an application for rare critical permissions.

In contrast to Kirin, but similar to Drebin, the method does not rely on static rules but automatically infers them from a training dataset. Nonetheless, it only uses a small subset of permissions that need to be manually defined in advance.

PENG The last approach we use for comparison was proposed by Peng et al. [156] in 2012. The authors use risk scoring functions to measure the risk of an application. Similar to Kirin and RCP, the risk score is based on the requested permissions of an application, i.e., it increases with the number of permissions an application requests. However, the extent to which a specific permission influences the resulting risk score may vary, depending on the underlying distribution that an application is considered to belong to. For estimating these distributions, the authors propose different Naive Bayes models. These models mainly differ in their ability to weigh certain critical permissions or to consider different application categories within the used dataset.

This approach assigns a high risk to applications that request many permissions.

For our evaluation, we implement the approach of Peng et al. using an SVM instead of a Naive Bayes classifier. The SVM shows similar results to those reported in the original paper. Besides, it allows examining the impact of the additional feature sets considered by Drebin on the overall detection performance.

4.3.1 *Further Related Work*

The analysis and detection of Android malware has been a vivid area of research in the last years. In addition to the previously discussed approaches, further concepts and techniques have been proposed to counter the growing amount and sophistication of this malware. An overview of the current malware landscape is provided in the studies of Felt et al. [76], Zhou & Jiang [223], and Wei et al. [202].

DETECTION USING STATIC ANALYSIS The first approaches for detecting Android malware have been inspired by concepts from static program analysis. Several methods have been proposed that statically inspect applications and disassemble their code [e.g., 13, 71, 73, 77, 97]. For example, the method Stowaway [77] analyzes API calls to detect overprivileged applications and RiskRanker [97] statically identifies applications with different security risks. Similarly, Chen et al. propose the tool MassVet [44], which allows scanning for repackaged malware in large amounts of data. Finally, there also exist common open-source tools for static analysis, such as Smali [87] and Androguard [62], which enable dissecting the content of applications with little effort.

In contrast to other static approaches,...

Our method DREBIN is related to these approaches and employs similar features for identifying malicious applications, such as permissions, network addresses, and API calls. However, it differs in two central aspects from previous work: First, we abstain from crafting detection patterns manually and instead apply machine learning to analyze information extracted from static analysis. Second, the analysis of DREBIN is optimized for effectiveness *and* efficiency, which enables us to inspect applications directly on the smartphone.

...DREBIN is optimized for effectiveness and efficiency.

DETECTION USING DYNAMIC ANALYSIS Another branch of research has studied the detection of Android malware at run-time. Most notably, are the analysis system TaintDroid [72] and DroidScope [217] that enable dynamically monitoring applications in a protected environment, where the first one focuses on taint analysis, and the latter allows the introspection at different layers of the platform. While both systems provide detailed information about the behavior of applications, they are technically too involved to be deployed on smartphones and detect malicious software directly.

Analyzing applications at run-time...

As a consequence, dynamic analysis is mainly applied for offline detection of malware, such as scanning and analyzing large collections of Android applications. For example, the methods Mobile Sandbox [183], DroidRanger [224], VetDroid [221], AppsPlayground [163], Andrubis [131], CopperDroid [190], and Monet [186] have been successfully applied to study apps with malicious behavior on different datasets. Google operates a similar detection system called Bouncer. Such dynamic analysis systems are suitable for filtering malicious applications from Android markets. Due to the openness of the Android platform, however, applications may also be installed from alternative sources, such as web pages and memory sticks, which requires detection mechanisms that operate on the smartphone.

ParanoidAndroid [161] was one of the few detection systems that employed dynamic analysis and can spot malicious activity while running on the smartphone. To this end, a virtual clone of the smartphone is run in parallel on a dedicated server and synchronized with the actions of the device. This setting allows for monitoring the behavior of applications on the clone without disrupting the functionality of the real device. The duplication of functionality, however, is involved and with millions of smartphones, operating ParanoidAndroid at large scale is technically not feasible in practice.

...is still infeasible directly on the mobile device.

4.3.1.1 *Detection using machine learning*

The difficulty of manually crafting and updating detection patterns for Android malware has motivated the use of machine learning. Several methods have been proposed that analyze applications automatically using learning methods [e.g., 22, 156, 173]. As already discussed, Peng et al. [156] apply probabilistic learning methods to the permissions of applications for detecting malware. Similarly, the methods Crowdroid [38], DroidMat [209], Adagio [91], MAST [43], DroidSIFT [219], MaMaDroid [137], and DroidAPIMiner [1] analyze features statically extracted from Android applications using machine learning techniques. Closest to our work is DroidAPIMiner [1], which provides a similar detection performance to DREBIN on Android malware. However, DroidAPIMiner builds on a k-nearest neighbor classifier that induces a significant runtime overhead and impedes operating the method on a smartphone. Moreover, DroidAPIMiner is not designed to provide explanations for its detections and therefore is opaque to the practitioner.

Most learning-based methods are computationally expensive...

Overall, previous work using machine learning mainly focuses on accurate detection of malware. Additional aspects, such as the efficiency and the explainability of the detection, are not considered. We address these aspects and propose a method that provides an effective, efficient, and explainable detection of malicious applications.

...and thus do not run directly on mobile devices.

4.4 CHAPTER SUMMARY

Android malware is a fast-growing threat. Classic defenses, such as anti-virus scanners, increasingly fail to cope with the amount and diversity of malware in application markets. While approaches like DroidRanger [224] and AppPlayground [163] support filtering malicious applications from these markets, they induce a run-time overhead that is prohibitive for directly protecting smartphones. Other approaches, such as the detection tool presented in Chapter 3, require manual effort to derive proper signatures that allow the detection of malware on the device. As a remedy, we introduce DREBIN, a lightweight method for detecting Android malware. DREBIN combines concepts from static analysis and machine learning, which enables it to better keep pace with malware development. In the following chapter, we compare its detection performance to related approaches and examine whether its run-time is low enough, such that DREBIN can be applied directly on mobile devices.

5 PERFORMANCE EVALUATION

In the previous chapter, we presented a machine learning-based approach capable of detecting Android malware directly on mobile devices. We named our method Drebin. While common detection methods still rely on manually crafted signatures, our approach is, in contrast, able to automatically infer characteristic patterns from malicious applications. To demonstrate the efficacy of Drebin, we conduct a comprehensive analysis, including the examination of its detection capabilities and run-time performance. In summary, we conduct the following experiments:

1. *Detection performance.* We evaluate the detection performance of Drebin on different datasets and compare it with related approaches, including popular anti-virus products. The datasets used throughout the evaluation contain samples of several years, covering the time span between 2010 and 2017.
2. *Run-time performance.* In the second step, we analyze the runtime performance of a prototype implementation of Drebin on several mobile devices. The mobile devices have different hardware configurations in order to ensure that Drebin also runs on older devices with limited computational power.

5.1 EVALUATION DATA

We start our analysis by comparing the detection performance of Drebin against the related methods [71, 156, 173] that we discussed in the previous chapter. For all experiments, we consider datasets that contain Android applications over several years. Table 5 provides an overview of all dataset, including the number of samples per class and the respective time range.

We evaluate our method using Android applications from 2010 to 2017

Dataset	**#Malicious**	**#Benign**	**Time Period**
Drebin$_{\text{ORIG}}$	5,557	123,430	2010 – 2012
Drebin$_{\text{AVC}}$	6,013	93,635	2010 – 2012
AMD	24,553	93,635	2010 – 2016
Silverpush	234	93,635	2010 – 2017

Table 5: Overview of datasets used for the evaluation of Drebin.

In the following, we provide a detailed description of all listed datasets and discuss why they are relevant for our evaluation.

DREBIN-ORIG DATASET This denotes the dataset used in Arp et al. [8] and has been one of the largest datasets used for the evaluation of a learning-based approach back in 2014. To construct this dataset, we acquired 131,611 applications containing benign as well as malicious applications. The samples have been collected between August 2010 and October 2012. In detail, the dataset contains 96,150 applications from the official Google Play store, 19,545 applications from different alternative Chinese Markets, 2,810 applications from alternative Russian Markets, and 13,106 samples from other sources, such as Android websites, malware forums, and security blogs. In addition, the dataset includes all samples from the *Android Malware Genome Project* [223].

The first dataset contains around 130,000 different applications...

To distinguish malicious from benign applications, we sent each sample to the VirusTotal service and flagged the applications based on the output of ten common anti-virus scanners (AntiVir, AVG, BitDefender, ESET, F-Secure, GData, Kaspersky, McAfee, Sophos, and Symantec). We scrutinized all samples as malware which had been flagged by at least two scanners. Subsequently, we removed samples labeled as adware from our dataset, as this type of software is in a twilight zone between malware and benign functionality.

The final dataset contains 123,430 benign applications and 5,557 malware samples. Note that there are slight differences compared to the numbers reported in Arp et al. [8]. These differences arise, since the reimplementation of Drebin is unable to analyze a small number of applications.

...including roughly 5,600 malicious Android apps.

Following the described labeling procedure, we sought to ensure that our data is (almost) correctly split into benign and malicious samples. However, as we discuss throughout this section, drawing a distinct line between malicious and benign applications is often difficult, since anti-virus scanners do not follow common labeling schemes [106]. Instead, they often even change their decision over time [92, 143].

DREBIN-AVC DATASET In late 2017, we rescanned the complete Drebin$_{\text{ORIG}}$ dataset to check whether samples initially considered as false positives (i.e., samples only flagged by a single scanner) have been flagged by additional scanners in the meantime or indeed turned out to be false positives.

Surprisingly, we experience significant growth in the number of samples that are flagged by at least one scanner. In particular, we find that 35,352 applications are now flagged at least once, when considering all anti-virus scanners available on VirusTotal. In consequence, we revise our labeling procedure and scrutinize all samples as malicious using a threshold of 10 anti-virus flags. As proposed by Lindorfer et al. [130], we regard all 21,299 samples with less than 10 flags as suspicious and analyze them separately in Section 5.2.6. Note that

The second dataset is derived from the first one...

a large fraction of these samples are only flagged by a single scanner and thus are likely false positives. Still, we regard none of these 14,034 samples as benign, as we cannot be completely sure whether these do indeed not exhibit any malicious functionalities.

Moreover, we again filter samples that have been flagged as potentially unwanted programs (PUPS) by the popular labeling tool AVCLASS [119], leaving us with a total of 6,013 malicious samples. The vast majority of the remaining samples—5,040 applications—are already part of the $\text{DREBIN}_{\text{ORIG}}$ dataset. The difference occurs due to the different labeling approach used by AVCLASS and the additional information provided by VIRUSTOTAL.

In summary, the dataset consists of 6,013 malicious samples, 93,635 benign samples, and 21,299 suspicious samples. The vast majority of these newly flagged samples can, presumably, be considered as grayware. The reasons for the significant growth in the number of this type of application are manifold. First of all, a large number of these samples have been flagged only by a couple of anti-virus scanners and are possibly false positives. Second, Google has changed the *Google Play* policies several times within the past few years. As an example, samples containing advertising libraries that make use of push notifications have been banned from Google Play in September 2013 [116, 179]. Therefore, these applications have also been flagged by several anti-virus scanners since then.

...by using more recent labeling information retrieved in 2017.

ANDROID MALWARE DATASET The *Android Malware Dataset* (AMD) has been composed by Wei et al. [202] and contains 24,553 malicious samples. The applications belong to 71 different malware families and have been collected by the authors between 2010 and 2016.

The authors only select a malicious application for their dataset if it has been flagged by at least 50% of the 55 anti-virus scanners available at VIRUSTOTAL. Furthermore, at least 50% of the anti-virus scanners that flagged a sample as malicious, had to agree on the malware family the sample belongs to. Otherwise, the sample was removed from the dataset. Using this approach, the authors ensure that the dataset only contains samples which have a very low probability of being false positives.

The third dataset contains around 25,000 malicious samples from 2010 to 2016.

Unfortunately, the AMD dataset does not contain any benign samples. Therefore, we use the benign samples from $\text{DREBIN}_{\text{AVC}}$ to evaluate the detection performance of DREBIN on this dataset. The resulting dataset finally comprises a total of 24,553 malware samples and 93,635 benign applications.

SILVERPUSH DATASET Initially, we have motivated the development of a learning-based detection method for Android malware by pointing to the inherent limitations of signature-based approaches. An example of such a signature-based approach has been presented

Finally, we prepare a dataset to examine the detection of the Silverpush family.

in Chapter 3. While this method allows us to detect samples reliably which exhibit specific characteristics, it requires much effort to craft the necessary signatures manually. In this chapter, we would like to examine whether Drebin can extract the characteristic patterns required to detect these samples automatically. To this end, we compose a dataset consisting of all 234 samples of Silverpush and all 93,635 benign samples of the $\text{Drebin}_{\text{AVC}}$ dataset.

5.1.1 *Discussion*

Each of the four datasets is considered to shed light on a different aspect of the detection capabilities of Drebin. In particular, we use the $\text{Drebin}_{\text{ORIG}}$ dataset to verify that the reimplementation of our tool still provides results comparable to its original implementation. Additionally, we use the $\text{Drebin}_{\text{AVC}}$ dataset to account for more recent labeling information we retrieved from the VirusTotal service in late 2017. The VirusTotal reports show that a significant number of anti-virus engines follow unstable labeling schemes that change over time. Therefore, we also conduct experiments using the AMD, which solely consists of malware samples with a very low probability of being false positives. Finally, we prepare a distinct dataset to examine the detection performance of the Silverpush family.

5.2 DETECTION PERFORMANCE

Equipped with the necessary data, we can examine the detection capabilities of DREBIN. To this end, we start the evaluation by comparing its detection performance with related learning-based approaches and common anti-virus solutions in Section 5.2.1 as well as 5.2.2, respectively. Subsequently, we assess its detection performance of different malware families in Section 5.2.3, also considering in Section 5.2.4 the case where no samples of a malware family have been available during training of the classifier, i.e., when DREBIN has to output a decision for entirely unknown malware families. Afterward, we analyze its detection performance over time, using a large dataset that covers multiple years (see Section 5.2.5). In Section 5.2.6, we finally inspect its detection score for samples on which many anti-virus scanners disagree in their decision, making it difficult to confidently assign these samples to one of the two classes.

We evaluate the detection capabilities of DREBIN *in six different scenarios.*

5.2.1 *Comparison with Related Approaches*

We begin our evaluation by comparing DREBIN with related static approaches on all previously discussed datasets. In particular, we compare the detection performance against Kirin [71], RCP [156], and the approach proposed by Peng et al. [156]. For a detailed description of these approaches, we refer the reader to Chapter 4, where we discussed these methods in detail.

EVALUATION SETUP For this experiment, we randomly split the datasets into a training set D_{train} (66%) and a test set D_{test} (33%). The detection model and the respective parameters of DREBIN are examined on the training data, whereas the test set is only used for measuring the final detection performance. We repeat this procedure ten times for each dataset and average the results. For interested readers, we provide a detailed description of the training procedure in the following paragraph.

TRAINING STEP Instead of performing a k-fold cross-validation or performing a time-based split of the training data, we follow a slightly different training approach. In particular, we split the training data D_{train} into two datasets of equal size—a (new) training set $D_{\text{train},0}$ and a validation set $D_{\text{train},1}$. We train a linear SVM on the training data $D_{\text{train},0}$ using different parameters. More precisely, we vary the cost and weight parameters in the range of $C = 10^{-1}, \ldots, 10^3$ and $W = 10^0, \ldots, 10^4$, respectively. At the end of the training step, we select the parameter combination that yields the best bounded $\text{AUC}_{0.01}$ (see Chapter 2) on the validation set $D_{\text{train},1}$. Using the selected parameter combination, we train a model on the full training data D_{train}, i.e., the

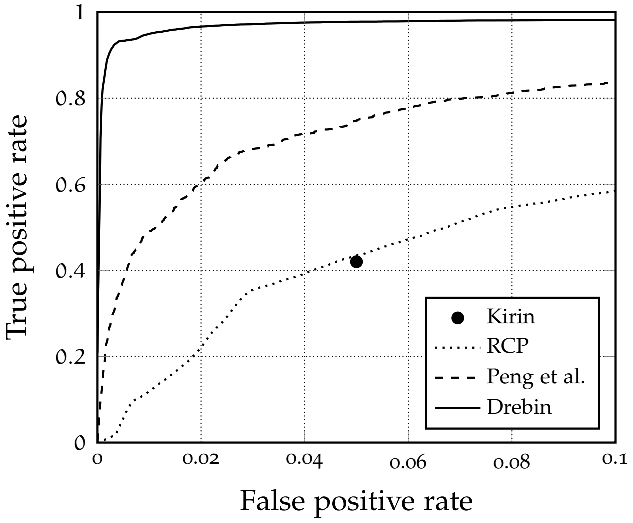


Figure 19: ROC curve comparison for $\text{Drebin}_{\text{ORIG}}$ dataset.

concatenation of training and validation set. Finally, we evaluate the detection performance of the resulting model on the remaining test data D_{test}.

Drebin *clearly outperforms all related approaches...*

RESULTS An overview of the results for all considered methods and all datasets is listed in Table 6. The results show that Drebin clearly outperforms all related approaches, yielding detection rates between 92% and 99% at a false positive rate of 1%. Note that this corresponds to one false alarm when installing 100 applications on the mobile device. In contrast, the related approaches provide lower detection rates between 8% and 94% at the same false positive rate. Kirin even exhibits higher false positive rates between 4 to 5% depending on the dataset. This can be seen in Figure 19 and Figure 20, which show the ROC curves for $\text{Drebin}_{\text{ORIG}}$ and $\text{Drebin}_{\text{AVC}}$, respectively. Overall, Kirin and RPC have obvious limitations detecting the malicious applications, since they only use a subset of the available requested permissions.

Dataset	**KIRIN**	**RPC**	**Peng**	**Drebin**
$\text{Drebin}_{\text{ORIG}}$	0.42 ± 0.01	0.12 ± 0.01	0.49 ± 0.03	0.95 ± 0.01
$\text{Drebin}_{\text{AVC}}$	0.38 ± 0.01	0.11 ± 0.01	0.53 ± 0.02	0.92 ± 0.01
AMD	0.44 ± 0.00	0.08 ± 0.00	0.66 ± 0.04	0.99 ± 0.00
Silverpush	0.63 ± 0.06	0.08 ± 0.04	0.94 ± 0.02	0.99 ± 0.01

Table 6: Detection rates for learning-based approaches. Except for Kirin, all listed results refer to a false positive rate of 1%.

However, all related methods show that requested permissions are important evidence for malicious activity. In the case of the Silver-

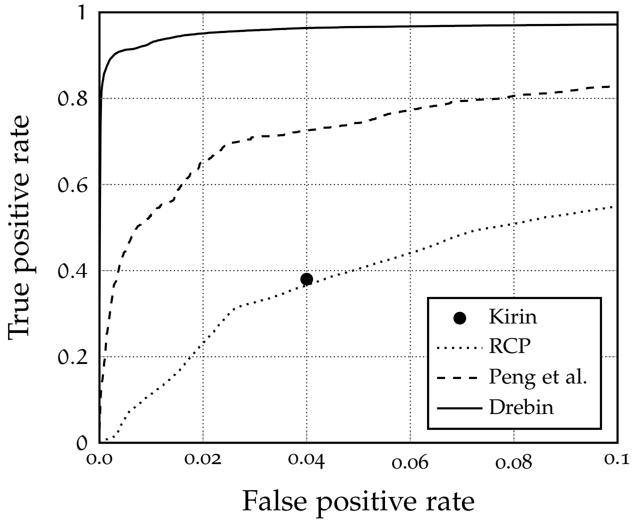


Figure 20: ROC curve comparison for DREBIN$_{\text{AVC}}$ dataset.

PUSH family, requested permissions like *RECORD_AUDIO* can already provide strong evidence to identify this family, as the results for Peng et al. indicate. Nonetheless, even when considering all available permissions, the approaches often still lack crucial information to be able to achieve an acceptable accuracy in many cases. The excellent performance of DREBIN results from the combination of different feature sets used to model malicious activity. These sets include requested permissions but also contain other relevant characteristics of applications, such as suspicious API calls, filtered intents, and network addresses. As a result, DREBIN detects the samples of the SILVERPUSH family almost perfectly, without the manual effort of crafting specific signatures like the approach used in Chapter 3.

...by combining several different types of features.

Dataset	Precision	Recall	F1-Score	TPR$_{0.01}$
DREBIN$_{\text{ORIG}}$	0.90 ± 0.02	0.93 ± 0.01	0.92 ± 0.00	0.95 ± 0.01
DREBIN$_{\text{AVC}}$	0.93 ± 0.02	0.91 ± 0.02	0.92 ± 0.00	0.92 ± 0.01
AMD	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00
SILVERPUSH	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01

Table 7: Detection rates of DREBIN on different datasets.

Table 7 provides further results of DREBIN's detection performance in terms of precision and recall. For these metrics, we consider a classification threshold of 0. Both metrics also confirm the good performance results of the classifier and two interesting insights:

First, all metrics underline that DREBIN can detect the malicious applications of the AMD dataset extremely reliably, yielding a detection rate of about 99%. Although the different ages of the benign and malicious dataset might contribute to this result, their impact on the detection performance should almost be negligible, as both classes have a considerable time overlap. Instead, the purity of the malicious data seems to make it particularly easy for the detection model to distinguish between benign samples and malware in this case. In particular, since a significant fraction of anti-virus scanners had to agree in their decisions throughout the labeling step, most samples of the AMD dataset likely contain very distinct malicious patterns. In consequence, it becomes easier for DREBIN to distinguish these samples from legitimate applications.

The second insight is the impact of the relabeling step between $\text{DREBIN}_{\text{ORIG}}$ and $\text{DREBIN}_{\text{AVC}}$ on the detection performance. Overall, the detection rates between $\text{DREBIN}_{\text{ORIG}}$ and $\text{DREBIN}_{\text{AVC}}$ only slightly differ, as Figure 20 shows. However, the false positive rate decreases significantly, which also results in a better precision on the $\text{DREBIN}_{\text{AVC}}$ dataset. For comparison, when considering a false positive rate of 0.1%, DREBIN is able to detect 79.10% and 84.07% of the malware in $\text{DREBIN}_{\text{ORIG}}$ and $\text{DREBIN}_{\text{AVC}}$, respectively. The absence of grayware in $\text{DREBIN}_{\text{AVC}}$ presumably leads to this decrease in the false positive rate. Thus, similar to the AMD dataset, it becomes easier for the detection methods to distinguish between malware and benign samples.

5.2.2 *Comparison with Anti-Virus Scanners*

Although DREBIN shows a better performance compared to related learning-based approaches, in the end it has to compete with anti-virus products in practice. Thus, we conduct an additional experiment which provides a direct comparison of DREBIN with several common anti-virus products.

We compare our approach to five popular anti-virus scanners...

EVALUATION SETUP For the comparison, we pick the five scanners that yielded the best detection performance on $\text{DREBIN}_{\text{ORIG}}$ in 2012. The detection performance of these scanners was determined by using the results obtained from the VIRUSTOTAL service when uploading the samples to the service for the first time. By following this approach, we are able to determine whether an anti-virus scanner has already been capable of detecting the malicious samples at the time these samples have been collected by us.

...using the VirusTotal service.

Throughout the evaluation, we examine three different malware datasets. Besides the malicious samples of the two already discussed datasets $\text{DREBIN}_{\text{ORIG}}$ and $\text{DREBIN}_{\text{AVC}}$ (see Section 5.1), we also use the malicious samples provided by the Malgenome project [223]. We choose a false-positive rate of 1% for DREBIN, which we think is sufficiently low for practical operation.

RESULTS The results for all five anti-virus scanners are shown in Table 8. Overall, the detection rate of the anti-virus scanners varies

	DREBIN	AV1	AV2	AV3	AV4	AV5
$\text{DREBIN}_{\text{ORIG}}$	94.50%	96.41%	93.71%	84.66%	84.54%	78.38%
$\text{DREBIN}_{\text{AVC}}$	92.23%	87.44%	88.24%	91.35%	88.92%	89.31%
Malgenome	98.21%	98.63%	98.90%	98.28%	98.07%	98.66%

Table 8: Detection rates of DREBIN and anti-virus scanners.

considerably. While the best scanners detect more than 90% of the malware in the $\text{DREBIN}_{\text{ORIG}}$ dataset, the worst performing scanner yields a detection rate which is more than 15% lower than that of the best performing one. On the $\text{DREBIN}_{\text{ORIG}}$ dataset, DREBIN provides the second best performance with a detection rate of 94.5% and outperforms 4 out of the 5 scanners. This observation is remarkable since, due to our test setting, at least two scanners should be able to detect each malware sample. Therefore, all samples have to be known for a certain amount of time, and most anti-virus scanners should be equipped with a corresponding signature.

DREBIN *even outperforms popular anti-virus scanners in many cases...*

When considering the $\text{DREBIN}_{\text{AVC}}$ dataset, which has been relabeled using more recent labeling information, the detection results for the anti-virus scanners vary only insignificantly. In this case, DREBIN even yields the best detection performance, slightly outperforming all anti-virus scanners. This result indicates that the selected anti-virus scanners are likely optimized towards a very low false positive rate, since none of them is close to a perfect detection rate, although the considered samples have been flagged by at least 10 anti-virus scanners and are known for several years (see Section 5.1). In contrast, on the Malgenome dataset, most anti-virus scanners achieve better detection rates than DREBIN, since these samples belong to a popular malware dataset and have been public for a longer period of time. Hence, almost all anti-virus scanners provide proper signatures for this dataset. Note that, when rescanning the malicious samples of the $\text{DREBIN}_{\text{ORIG}}$ dataset in late 2017, the results for this dataset are comparable to those obtained for the Malgenome dataset.

In all fairness, it needs to be mentioned that the false-positive rates of anti-virus scanners are in general lower than the false-positive rate of 1% we consider for DREBIN. However, the average user only installs some dozens of applications on her device. According to Google [96], the average number of installed applications per smartphone in the U.S. has been 35 in 2016. In consequence, we consider a false-positive rate of 1% still acceptable for operating DREBIN in practice. Moreover, as we further discuss in Section 5.2.6, even anti-virus scanners do not have a common malware definition. Consequently, there exists a large margin of applications where the decision of whether a particular application should be considered as malicious or as a false positive becomes somewhat ambiguous.

...and its false positive rate is low enough for practical application.

DISCUSSION The previous experiments show that DREBIN is capable of deriving meaningful patterns from malicious applications automatically, yielding detection results comparable to those obtained by common anti-virus products. While DREBIN has a slightly higher false positive rate than current anti-virus solutions, the classification model allows—in contrast to the signature-based approach of anti-virus scanners—the detection of previously unknown malicious samples. Unlike DREBIN, anti-virus products often require multiple weeks or even months until their signature databases get updated, thus leaving mobile devices vulnerable in this time frame. While a lower false positive rate is indeed crucial when scanning thousands or millions of applications, we argue that a false positive rate of 1% is sufficient in case of mobile devices, since only a few dozen applications are installed on the device by average users.

5.2.3 *Detection of Malware Families*

Analyzing the detection rate for different malware families...

An important aspect when evaluating the detection performance of a malware detection method poses the balance between the different malware families in the dataset [170]. In particular, if the number of samples of certain malware families is much larger than that of other families, the detection result mainly depends on these families. To address this problem, one can use the same number of samples for each family. However, this leads to a distribution that significantly differs from reality. Instead, we evaluate the detection performance for each of the 20 largest malware families in each dataset separately. We again start our analysis using the $\text{DREBIN}_{\text{ORIG}}$ dataset.

...ensures that not only large families can be detected.

DREBIN-ORIG FAMILY DETECTION Figure 21 illustrates the detection performance of DREBIN for each family, while Table 9 lists the available number of samples per family. DREBIN is able to reliably detect all families with an average accuracy of 93.82% at a false-positive rate of 1%. In particular, seventeen families show a detection rate of more than 90%, where five of them can even be identified perfectly (H, I, O, P, Q). There is only one malware family which cannot be reliably detected by DREBIN. This family is Gappusin (R) [148]. Although it is in many cases possible to extract features which match the description of the Gappusin family—amongst others the hostname of the external server—there are too few malicious features to identify the samples as malware. Gappusin mainly acts as a downloader for further malicious applications and thus does not exhibit common malicious functionality, such as theft of sensitive data.

DREBIN-AVC FAMILY DETECTION As mentioned in Section 5.1, the $\text{DREBIN}_{\text{ORIG}}$ dataset contains many samples which have been considered as benign in previous experiments but are now flagged as

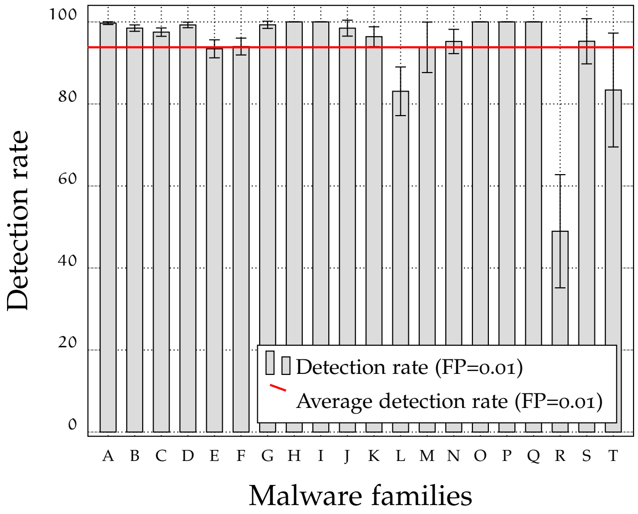


Figure 21: Detection per family in DREBIN$_{\text{ORIG}}$.

Id	**Family**	**#**	*Id*	**Family**	**#**	*Id*	**Family**	**#**
A	FakeInstaller	925	*H*	Kmin	147	*O*	MobileTx	69
B	DroidKungFu	667	*I*	FakeDoc	132	*P*	FakeRun	61
C	Plankton	625	*J*	Geinimi	92	*Q*	SendPay	59
D	Opfake	613	*K*	Adrd	91	*R*	Gappusin	58
E	GingerMaster	339	*L*	DroidDream	81	*S*	Imlog	43
F	BaseBridge	330	*M*	LinuxLotoor	70	*T*	SMSreg	41
G	Iconosys	152	*N*	GoldDream	69			

Table 9: Largest malware families in DREBIN$_{\text{ORIG}}$.

malicious by several anti-virus scanners. In the following, we repeat the same experiment as conducted on DREBIN$_{\text{ORIG}}$ using the relabeled dataset. Afterward, we compare the result.

Most malware families are detected reliably by DREBIN.

The family names and the number of samples for each family in DREBIN$_{\text{AVC}}$ can be found in Table 10. Figure 21 depicts the detection performance of DREBIN for each family. Note that the malware families differ between DREBIN$_{\text{AVC}}$ and DREBIN$_{\text{ORIG}}$, due to the altered labeling procedure. The average detection performance for the families slightly increases to 94.98%. Furthermore, it is noteworthy that none of these families exhibits a detection rate of less than 85%. We conclude that the relabeling approach lead to more stable family labels, allowing DREBIN to better distinguish between the two classes in this dataset.

The largest family with a low detection performance of only 45.57% is *SMSAgent* [15]. The family contains 19 samples and is ranked on place 32. When examining the members of this family in more depth, we notice that only 13 of the 19 available samples request a permission related to SMS functionality. Most of these samples contain ad-

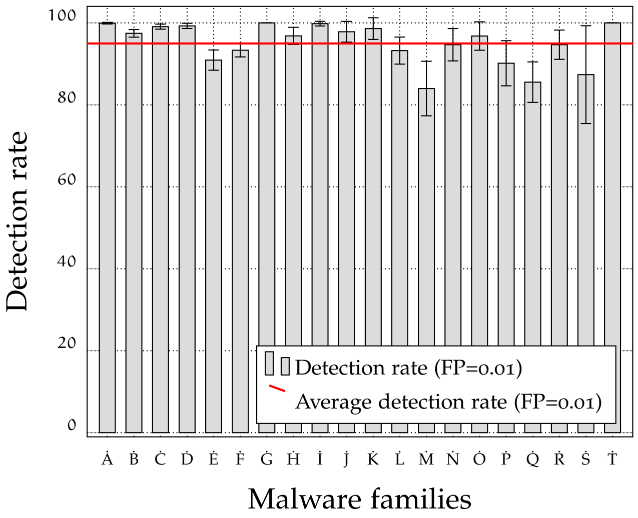


Figure 22: Detection per family in Drebin$_{AVC}$.

Id	**Family**	**#**	*Id*	**Family**	**#**	*Id*	**Family**	**#**
Ȧ	fakeinst	883	*Ḣ*	iconosys	201	*Ȯ*	fujacks	67
Ḃ	droidkungfu	682	*İ*	kmin	149	*Ṗ*	pjapps	65
Ċ	opfake	603	*J̇*	boxer	114	*Q̇*	lotoor	62
Ḋ	plankton	601	*K̇*	geinimi	100	*Ṙ*	imlog	49
Ė	ginmaster	365	*L̇*	fakeapp	94	*Ṡ*	steek	46
Ḟ	basebridge	334	*Ṁ*	droiddream	82	*Ṫ*	fakenotify	43
Ġ	fakeflash	240	*Ṅ*	golddream	72			

Table 10: Largest malware families in Drebin$_{AVC}$

vertising libraries and are possibly flagged by generic heuristic signatures. However, among the remaining 13 applications are two particularly interesting ones[1]. In both cases, Drebin classifies them as benign, since it can only extract a handful of mainly unsuspicious features. In contrast, both samples are flagged by around 25 anti-virus scanners. When analyzing these samples, we discover that both applications are loading code at run-time from an APK file that is hidden inside the *assets* folder. The hidden application, in turn, contains the malicious functionality. This example demonstrates an inherent drawback of Drebin, since it solely relies on static analysis for its decision. Thus, it is susceptible to this kind of obfuscation where code is loaded dynamically at run-time. As a remedy, it is possible to extend Drebin such that it looks for *apk* and *dex* files hidden inside an app and then analyzes them separately in this case [184]. We provide a detailed discussion of the limitations of Drebin in Section 5.4.

Some samples of the SmsAgent *family hide their malicious payload successfully.*

[1] 0fcccc5d9f3f3e0cf9d559ea203318f06feb6037443e441db5c7d2688285b005
13c5a348d44a11aba143e821ec5c0257fad89bb642d3c2ed4753cd811146ccb5

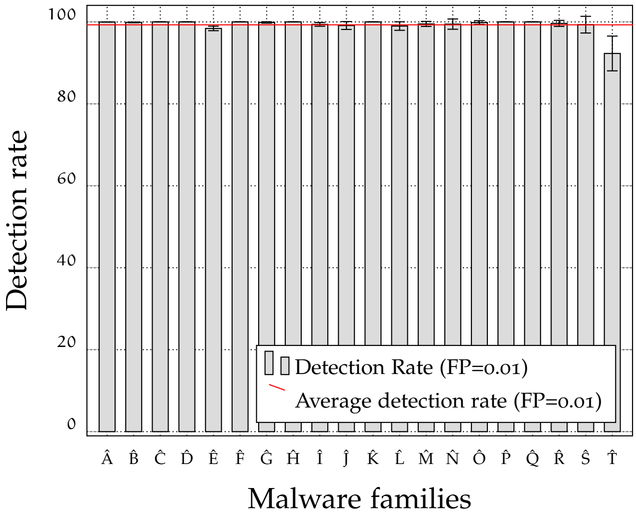


Figure 23: Detection per family in AMD.

Id	**Family**	**#**	*Id*	**Family**	**#**	*Id*	**Family**	**#**
Â	Airpush	7843	*Ĥ*	BankBot	648	*Ô*	Triada	210
B̂	Dowgin	3385	*Î*	Jisut	547	*P̂*	Minimob	203
Ĉ	FakeInst	2172	*Ĵ*	KungFu	546	*Q̂*	Kyview	175
D̂	Mecor	1820	*K̂*	Lotoor	328	*R̂*	SlemBunk	174
Ê	Youmi	1300	*L̂*	RuMMS	310	*Ŝ*	SmsKey	165
F̂	Fusob	1276	*M̂*	Mseg	235	*T̂*	SimpleLocker	165
Ĝ	Kuguo	1199	*N̂*	Bogx	215			

Table 11: Largest malware families in AMD.

AMD FAMILY DETECTION Figure 23 shows the results for the *Android Malware Dataset*, while Table 11 lists the respective families. As can be seen in the plot, none of the 20 largest families has a detection rate below 92%. We deduce that the excellent detection performance on this dataset results from the labeling procedure, which led to a very pure malware dataset [202]. Surprisingly, there are only four malware families in the complete dataset where DREBIN yields a detection rate below 90%, i.e., *Opfake* (88.83%), *FakeUpdates* (73.81%), *Tesbo* (69.44%), and *Fobus* (60.00%). The latter three consist of only 4 to 5 samples, thus making it difficult for DREBIN to generalize their characteristics appropriately, as these have to be derived from only two training samples on average.

Most of the AMD *families can be detected extremely reliably...*

In general, however, we cannot observe a dependency between the detection rate and the size of the malware families. Instead, there are many families in the AMD which also contain very few samples but can be detected very well by our approach. We conclude that it highly depends on the diversity of the members within a particular malware family whether it is possible to derive meaningful patterns

...even when only few samples are available for training.

for its detection or not—mostly independent of the number of available samples of a family.

To gain a deeper understanding of this issue, we perform additional experiments in the next section. These show how DREBIN performs in general when no or only a few samples of a particular malware family are available during training.

5.2.4 *Detection of Unknown Families*

The underlying learning algorithm of our approach is supervised, i.e., DREBIN uses labeled benign and malicious data to derive its detection model. Thus, it is essential to assess how many samples of a family need to be known to detect this family reliably. To study this issue, we conduct two additional experiments where we limit the number of samples for a particular family in the training set.

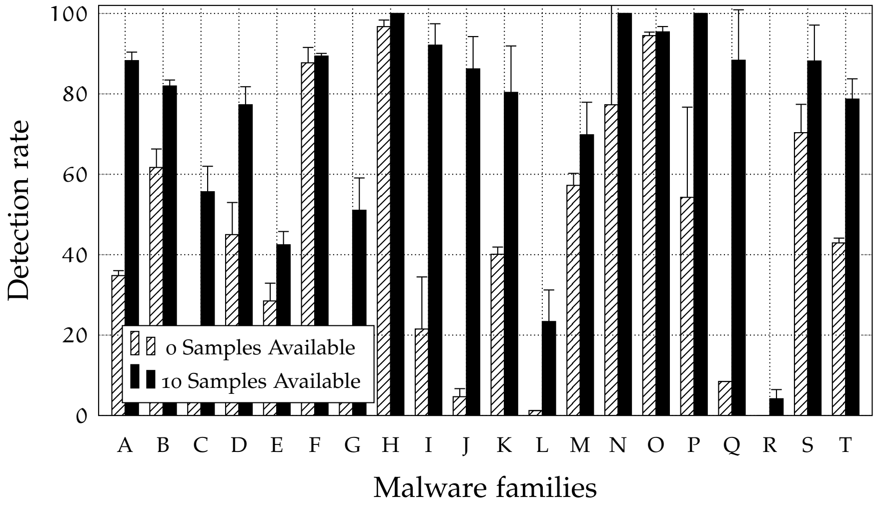


Figure 24: Detection of unknown families for $\text{DREBIN}_{\text{ORIG}}$.

How much is the family detection rate affected if no or only a few samples are available during the training phase?

EXPERIMENTAL SETUP For each of the three datasets considered in the previous section, we conduct two additional experiments to examine how well DREBIN can generalize even if only a few samples of a specific malware family are available. In the first experiment, we provide no samples of the family, corresponding to a malware strain that is utterly unknown to the classifier. In the second experiment, we put 10 randomly selected samples of the family back into the training set, thus simulating the starting spread of a new family. Each experiment is repeated 10 times, and the results are finally averaged.

DREBIN-ORIG FAMILY DETECTION The results of the two experiments on $\text{DREBIN}_{\text{ORIG}}$ are presented in Figure 24. If no samples are available during the training step, it becomes rather difficult for our method to detect a family, since the SVM cannot discriminative patterns in advance. However, only very few samples are necessary to generalize the behavior of most malware families. With only 10 samples in the training set, the average detection performance increases by 30 percent on average. Three families can even be detected perfectly in this setting. The reason for this is that members of certain families are often just repackaged applications with slight modifications. Due to the generalization which is done by the SVM, it is therefore possible to detect variations of a family, even though only a very small set of samples is known.

The detection rate drops significantly, if no samples are provided during training...

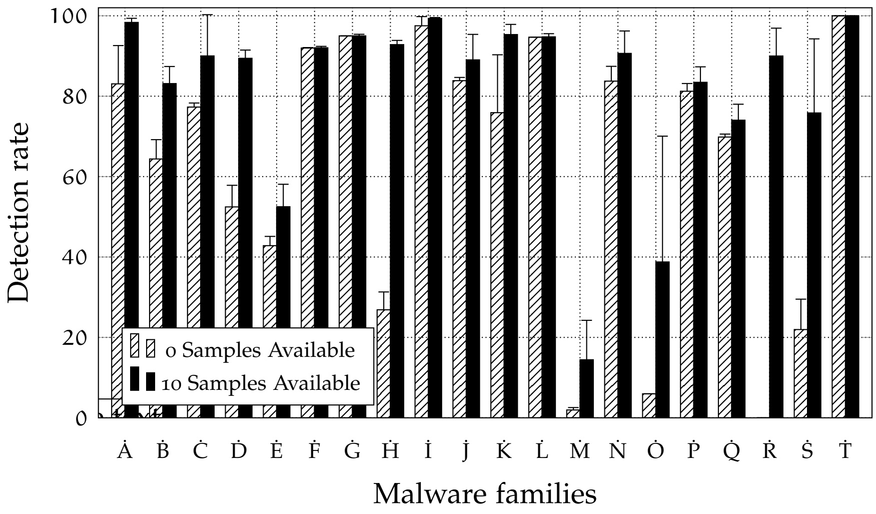


Figure 25: Results for leave-one-out experiments on Drebin$_{AVC}$.

DREBIN-AVC FAMILY DETECTION Figure 25 depicts the results on the Drebin$_{AVC}$ dataset. Like for Drebin$_{ORIG}$, the experiment first shows a decrease in the detection rate if no samples are provided. Similarly, the average performance increases again by 20% when only ten samples of a malware family are available for training.

In addition, the results also exhibit interesting differences. For example, Drebin provides a detection performance of 0% on Drebin$_{AVC}$ if no members of the family *imlog* (Ṙ) are available during training. In contrast, it shows acceptable results for the same family on Drebin$_{ORIG}$ (S) in the same setting. When comparing the number of samples of *imlog* available in both datasets, we notice that Drebin$_{AVC}$ contains 6 additional samples of this family (see Table 9 and Table 10). In this case, the relabeling procedure assigned additional samples to this family. These samples initially enabled Drebin to still extract characteristic patterns from the training data and, in turn, achieve a reasonable detection performance on the test set. Using the relabeled data, however, these samples are not available during training anymore, thus leading to a lack of detection for this particular family.

...but increases again if only a few samples are put back into the training set.

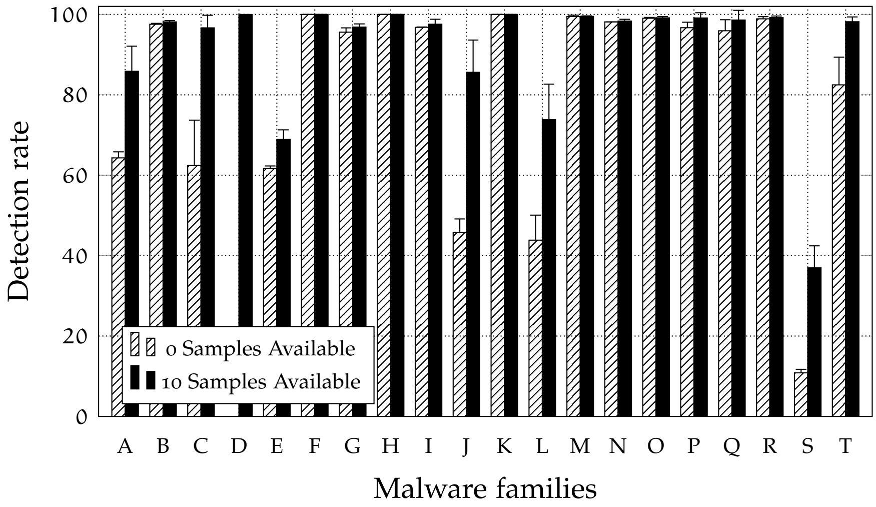


Figure 26: Results for leave-one-out experiments on AMD.

Overall, Drebin *has problems detecting completely unknown families...*

AMD FAMILY DETECTION Finally, we perform the same experiments using the AMD dataset. The results obtained on this dataset are particularly interesting, since Drebin provides almost perfect results on this dataset in the experimental setup presented in Section 5.2.3.

Figure 26 shows the results for the detection of unknown families. Surprisingly, even if we consider a setting where no samples of a particular malware family are available, Drebin shows a good detection performance for many of these families. We suspect that these families partially share similar malicious characteristics, like, for instance, the sending of SMS to premium services. Hence, these characteristics allow Drebin the detection of these families.

However, there also exist several families for which this observation does not apply. In particular, the malware family *Mecor* (D) is an example for this. The family cannot be detected when no samples are available but is detected almost perfectly as soon as only 10 samples are provided for training. When analyzing this case, we find that the family sends sensitive data to an external command and control server [203], whose URL is an essential feature for Drebin to identify this family. Consequently, Drebin requires some samples of this family to extract characteristics specific to this family and achieve reliable detection of Mecor.

DISCUSSION In the conducted experiments, we have examined the detection capabilities of Drebin if no or only a few samples of a family are available during training of the classification model. For all three considered datasets, we experience a significant drop in detection rate if no samples of a family are available throughout the training step. However, as soon as only 10 samples are provided, the average detection rate increases significantly by 14% (AMD) to 33% (Drebin$_{\text{ORIG}}$). Table 12 summarizes the results.

# Samples	Average detection rate ($\text{TPR}_{0.01}$)		
	DREBIN$_{\text{ORIG}}$	DREBIN$_{\text{AVC}}$	AMD
0	0.42 ± 0.31	0.63 ± 0.33	0.77 ± 0.30
10	0.75 ± 0.42	0.82 ± 0.22	0.92 ± 0.15

Table 12: Average detection results for largest families.

Interestingly, we observe significantly different results for some of the malware families depending on the considered dataset. By analyzing these cases, we notice that these differences mostly occur due to different labeling procedures. In particular, even a small number of updated labels can have a significant impact on the detection rate of a specific malware family. This observation further indicates that often only a few samples can make the difference whether DREBIN can detect a malware family or not. This is an important insight, since malware evolves and anti-virus analysts find new malware variants on a regular basis. As a result, DREBIN might not always be able to detect a new malware strain immediately but requires some samples to infer proper detection patterns. In the next section, we thus analyze the actual impact of time on the detection performance of DREBIN in more detail.

...but only needs a small number of family members to infer proper patterns for detection.

5.2.5 *Detection of Malware over Time*

Previously, we have evaluated the detection performance of DREBIN without considering the time dependency within the data set. This experimental setting implicitly follows the assumption that the underlying distribution is stationary, i.e., it does not change over time. In practice, however, malware analysts discover new malware families on a regular basis, and therefore the distribution continuously changes — a phenomenon referred to as *concept drift* in machine learning [112, 180]. As the results obtained in Section 5.2.4 indicate, DREBIN might not always be able to reliably detect members of newly occurring malware families. To examine the impact of time on the detection results, we therefore conduct additional experiments:

How does the evolution of malware affect the detection performance?

1. We examine the detection rates of DREBIN over time using applications from 2010 to 2015. In particular, we check for each year the detection performance of a classification model that has been trained on the available data from the previous years.
2. We compare the performance of the results to the performance obtained when training and testing on samples of the same year. This way, we retrieve a direct comparison between the results obtained on a stationary dataset and a distribution that changes over time.

For the evaluation, we use roughly 450,000 Android applications...

DATASET To evaluate the detection performance of our approach over time, we combine the DREBIN$_{\text{ORIG}}$ malware dataset [8] and the AMD dataset [202]. We use these two malware datasets, as both have been widely studied by several researchers and contain malicious samples ranging over multiple years. In addition, we use 431,551 benign samples that we have collected between 2010 and 2015, including the benign samples of the DREBIN$_{\text{AVC}}$ dataset.

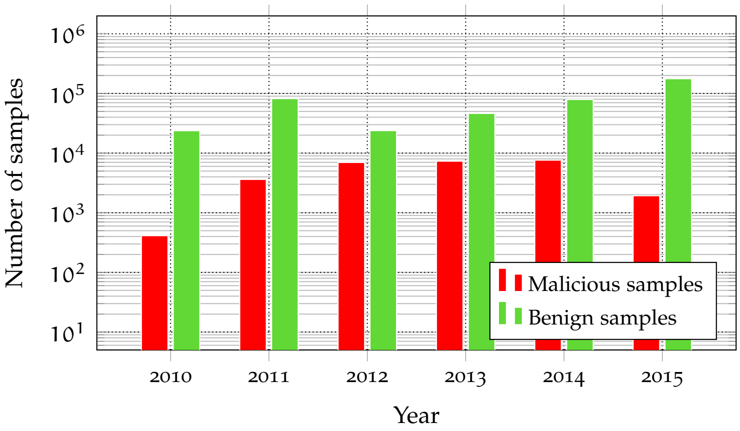


Figure 27: Time distribution of the full dataset.

...ranging from the year 2010 to 2015.

For all samples, we try to obtain their creation date by extracting the timestamp from the *classes.dex* file of each application using the *aapt* tool. We remove samples from the dataset for which this approach yields an implausible result, i.e., if the timestamp does not lie between 2010 and 2016. Moreover, we discard all applications from 2016, since we do not have enough benign data for this year. Our final dataset consists of 27,872 and 431,551 malicious and legitimate samples, respectively. Figure 27 shows the distribution of samples for these years. Note that the numbers are plotted logarithmically, as they differ significantly between the years.

EXPERIMENTAL SETUP Using the discussed data set, we evaluate the detection performance of our approach throughout two different settings. In the first setting, we randomly sample distinct training and test sets from the dataset of a particular year and evaluate the detection performance of the resulting classifier, i.e., we use a stationary distribution, which does not change over time. In contrast, in the second setting, we evaluate a classification model on the data of a particular year where the classifier has been trained and calibrated using all available data from the previous years, i.e., we use a presumably non-stationary distribution that changes over time. For each of these two settings, we conduct five independent runs and average the results. In the following, we refer to the first scenario as *stationary* and to the second scenario as *time-based*.

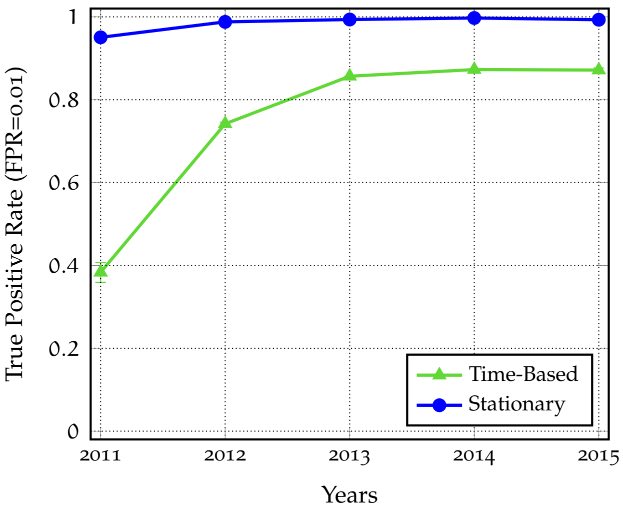


Figure 28: Detection results for DREBIN over time.

RESULTS Figure 28 shows the results for both considered scenarios. In the stationary scenario, DREBIN achieves very good results with detection rates between 95% to almost 100% at a false positive rate of 1%. For the years 2011 and 2012, we obtain slightly worse results than for the subsequent years. This observation can be explained by the circumstance that the data of these years contains the malicious samples of the $\text{DREBIN}_{\text{ORIG}}$ dataset. As already discussed in Section 5.2, these samples are more difficult to detect for DREBIN than those of the AMD dataset, thus slightly affecting the detection performance. Nonetheless, for the stationary scenario, the impact is negligible for the overall detection performance.

While DREBIN *achieves good detection rates on the stationary distribution...*

The differences become apparent when considering the *time-based* setting. While for all years the detection rate decreases, a significant drop of almost 60% can be observed for the year 2011. Note, however, that the underlying classifier has been trained solely on the data from 2010. This dataset contains only 413 malware samples, thus presumably making it difficult for the SVM to generalize enough malicious characteristics from the data. As more data becomes available in the subsequent years, the detection capability of DREBIN improves, yielding detection rates between 74% and 87%. While these results are still significantly below the detection rates obtained in the stationary setting, we assume that they can be further improved by continuously retraining the classifier.

...a significant drop occurs when the time dependency is considered.

The detection performances increases again as more data becomes available.

Overall, it highly relies upon the particular malware families that occur over time whether DREBIN can detect them or not. In particular, the detection rate might immensely vary, depending on the available training data, i.e., if it allows DREBIN to learn the required patterns in advance. As demonstrated in Section 5.2.4, in some cases even new malware families can be detected if they share salient characteristics with already known families. In case of malware families that yield

Setting	Year	Precision	Recall	F1-Score	$TPR_{0.01}$
Stationary	2011	0.93 ± 0.01	0.95 ± 0.01	0.94 ± 0.00	0.95 ±0.01
	2012	0.98 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ±0.00
	2013	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ±0.00
	2014	0.99 ± 0.00	1.00 ± 0.00	0.99 ± 0.00	1.00 ±0.00
	2015	0.98 ± 0.01	0.99 ± 0.00	0.98 ± 0.00	0.99 ±0.00
Time-Based	2011	0.89 ± 0.00	0.35 ± 0.00	0.50 ± 0.00	0.38 ±0.02
	2012	0.99 ± 0.00	0.74 ± 0.00	0.85 ± 0.00	0.74 ±0.00
	2013	0.98 ± 0.00	0.86 ± 0.00	0.92 ± 0.00	0.86 ±0.00
	2014	0.99 ± 0.00	0.87 ± 0.00	0.93 ± 0.00	0.87 ±0.00
	2015	0.92 ± 0.00	0.87 ± 0.01	0.91 ± 0.01	0.87 ±0.01

Table 13: Detection performance over time.

completely novel characteristics, however, Drebin will most likely fail to detect them.

DISCUSSION Initially, Android malware detection methods have been evaluated without considering the time dependency within the data. However, in recent years, several researchers have started to study the impact of concept drift on the classification results of Android malware detection systems [7, 112]. In this section, we have conducted additional experiments in order to examine the effect of concept drift on the detection performance of Drebin. Similar to the results presented by other researchers, we also notice that the overall detection rate decreases. However, Drebin is often still able to achieve good results, depending on the fraction of newly occurring malware families within the test data, i.e., the extent of the concept drift. It is therefore difficult to provide concrete numbers how much concept drift impacts the overall detection performance in general, as this highly depends on the available data.

Overall, the essence of supervised learning algorithms is to extract and learn patterns from a given set of training data that allow distinguishing between multiple classes. Consequently, it becomes challenging—if not impossible—for these algorithms to identify completely unknown patterns not available during training. As a remedy, sophisticated feature design and continuous retraining of the classifier can help to alleviate the problem of concept drift, but can also not solve it completely.

5.2.6 *Detection of Suspicious Applications*

In the previous sections, we have performed different experiments on data that could be split into legitimate and malicious applications with high confidence. When describing the datasets in Section 5.1, however, we have already mentioned that a significant fraction of the samples in the Drebin$_{\text{ORIG}}$ dataset is now flagged by at least one anti-virus scanner—even though these samples have initially been considered legitimate. Throughout this section, we analyze these *suspicious samples* further and discuss the gained insights.

We notice that only a small number of anti-virus scanners consider these samples as malicious. That seems surprising, given the fact that these applications are already known for several years by now. At first glance, it might thus be reasonable to declare them as false positives and ignore the malware flags of the small number of anti-virus scanners. Nonetheless, labeling these applications as benign is problematic, as it remains unclear whether these applications do exhibit any malicious characteristics or not. As a remedy, users could therefore decide on their own, whether or not to trust a particular application. To support users in their decision, the classification score of Drebin might be helpful in case of doubt. In the following, we therefore examine the informative value of this decision score.

EVALUATION SETUP To examine the expressiveness of the score, we first label all samples of the Drebin$_{\text{ORIG}}$ dataset as malicious that have been flagged by at least one anti-virus scanner, i.e., we assume the worst case that all flagged samples are indeed malicious. Next, we split the dataset into training and test sets using the following approach:

- *Training dataset.* For training, we use all malicious samples with at least 10 anti-virus flags. Thus, the training set consists of 14,053 malware samples and 62,423 randomly selected benign samples from the full dataset.
- *Test dataset.* For testing, we consider all malware samples with less than 10 anti-virus flags combined with the remaining benign applications. Hence, the test set contains 21,299 malicious and 31,212 benign samples.

Using this setup, we apply the classifier on the test set and sort the samples according to their assigned classification score. Finally, we compare the scores with the number of anti-virus flags. We repeat this procedure ten times and average the results.

RESULTS Figure 29 shows the results of this experiment. The samples are sorted using their normalized classification score that ranges between 0 and 1, i.e., samples with a higher score are more likely to be malicious according to Drebin. The plot shows a strong correlation between the number of anti-virus flags and the score assigned by Drebin. In particular, we see exponential growth in the number of anti-virus flags for samples with a score close to 1.

The classification score correlates with the number of anti-virus flags

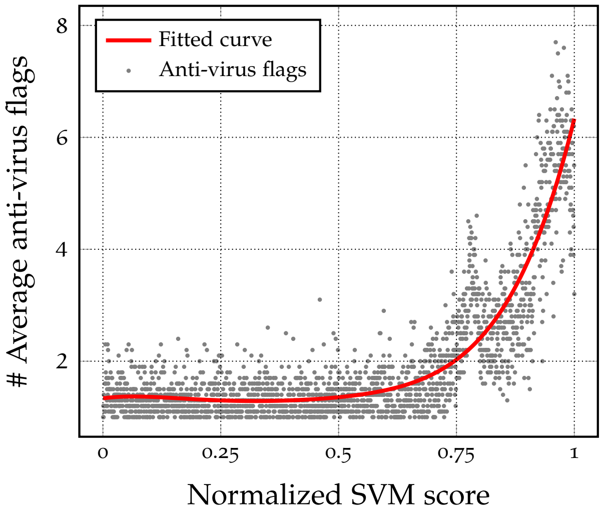


Figure 29: Evaluation of detection performance on suspicious samples, i.e., applications that are flagged by less than 10 but at least one anti-virus scanner.

The unexpected peak at a score of 0.8..

Note, however, that the plot also exhibits an unexpected peak for a group of samples with a score of about 0.8. To examine this issue, we randomly select 10 of these samples and perform a static and dynamic analysis of them. In our analysis, we find that all of these applications[2] are quiz games, sharing most of their code base. Besides, they all contain the same advertising SDKs, namely *Mobclix* and *Pontiflex*. The dynamic analysis exposes the network communication with both advertising networks. Apart from displaying advertisements to the user, some device information is sent to the servers of these advertising companies.

..most likely originates from false positives.

Nonetheless, even though this information also includes the IMEI of the device, it does not appear to be particularly malicious. We conclude that these applications are mainly flagged due to the presence of the advertising SDKs. This is also reinforced by the fact, that all of these samples are considered to belong to the *Mobclix* family by several anti-virus products. Interestingly, different versions of these advertising libraries also occur in many benign samples. Consequently, we also analyze three randomly selected benign samples. We find that these samples exhibit a similar behavior than those samples flagged as malicious.

In conclusion, we cannot tell with full certainty whether or not these samples contain malicious functionality. Two reasons, however, highly indicate that they cannot be regarded as actual malware. First of all, the analyzed samples contain only a small amount of code and we thus consider it unlikely that we missed the malicious functionality. Instead, the matching anti-virus signatures are possibly not specific enough and thus result in false positives [208]. Second, the

[2] The full list of *SHA256* hashes can be found in Section A.1

samples have only been flagged by less than 10 anti-virus scanners, although they have already been available for a long time. For comparison, the malicious samples included in the original Drebin$_{\text{ORIG}}$ dataset are now flagged by 40 scanners on average.

In summary, this specific example shows that the line between a malicious and legitimate application can be blurry. Hence, the output score of Drebin can in many of these cases help decide whether an app exhibits suspicious functionalities. In case of doubt, the user should not install it on the device. In some cases, however, the line becomes too thin, making it challenging for Drebin to distinguish between both cases.

5.2.7 *Discussion*

In the experiments conducted throughout this section, we have focused on the evaluation of Drebin's detection capabilities. To this end, we have examined the detection performance in six different scenarios using various datasets. Overall, Drebin shows good performance results with detection rates of more than 90% at 1% false positives in most cases, thus clearly outperforming several related approaches and even some popular anti-virus scanners. Furthermore, we have analyzed that its good detection capability is not only limited to large malicious families within the available data but even holds for families with only a few samples in many cases.

Nonetheless, we have also identified the possible limitations of our approach throughout the experiments. As a first limitation, we notice that it becomes difficult for Drebin to detect some malware families if no members of these families are available in the training data. This insight is of particular interest, since new malware families arise over time, thus changing the underlying distribution and possibly affecting the detection performance of the classifier. The continuous change of the underlying distribution is a common problem in machine learning referred to as concept drift.

Fortunately, we could also observe that Drebin often requires only very few samples in order to extract and generalize characteristic patterns of a malware family. While this property does of course not solve the problem that the underlying distributions vary over time, it indicates that continuous retraining of the classifier can be beneficial to limit their impact on the overall detection performance.

5.3 RUN-TIME PERFORMANCE

We measure the run-time of our prototype app on five different smartphones.

Besides the detection performance of a malware detection system, also its run-time performance has a direct impact on its applicability in practice. This becomes even more critical if the system is supposed to run on minimal hardware, such as mobile devices, as in the case of DREBIN. In consequence, we also perform an analysis of DREBIN's run-time performance on different mobile devices in this section. While the computing power of mobile devices is rapidly increasing, it is still limited compared to regular desktop computers. Consequently, a detection method that is supposed to run directly on these devices has to carry out its task very efficiently.

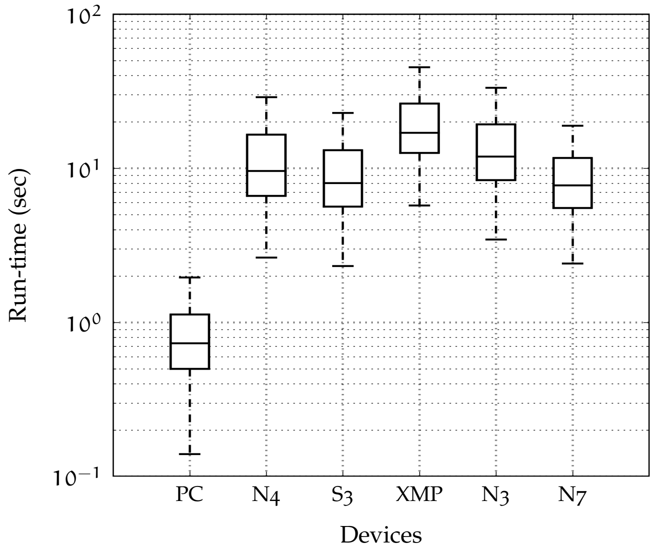


Figure 30: Run-time performance of DREBIN.

Even on older smartphones...

To analyze the run-time of DREBIN we implement a standalone Android application that receives a learned detection model and is able to perform the detection process directly on the smartphone. Using this application, we measure the run-time of DREBIN on different devices using 100 randomly selected popular applications from the Google Play Store. For this experiment, we choose devices which cover various widespread hardware configurations, including four smartphones (Nexus 4, Galaxy S3, Xperia Mini Pro, and Nexus 3), a tablet (Nexus 7), and a regular desktop computer (PC).

...the classification takes only tens of seconds.

The results are presented in Figure 30. On average, DREBIN is able to analyze a given application in less than 15 seconds on the five smartphones. Even on older models, such as the Xperia Mini Pro, the method is able to analyze the application in roughly 20 seconds on average. Overall, no analysis takes longer than 1 minute on all devices. On the desktop computer (2.26 GHz Core 2 Duo with 4GB RAM) DREBIN achieves a remarkable analysis performance of 750 ms

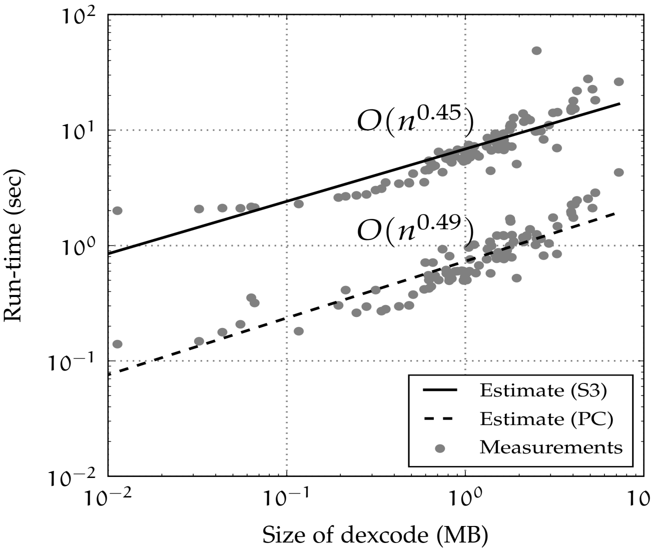


Figure 31: Detailed run-time analysis of Drebin.

per application, which enables scanning 100,000 applications in less than a day.

A detailed run-time analysis for the desktop computer and the Galaxy S3 smartphone is presented in Figure 31, where the run-time per application is plotted against the size of the analyzed code. Surprisingly, on both devices Drebin attains a sublinear run-time, that is, its performance increases with $O(\sqrt{m})$ in the number of analyzed bytes m. Apparently, the number of features does not increase linearly with the code and thus larger applications do not necessarily contain more features to analyze.

From this evaluation, we conclude that Drebin does not only reliably detect malicious applications but is furthermore able to perform this task in a time that clearly meets practical requirements.

5.4 LIMITATIONS

The previous evaluation demonstrates the efficacy of our method in detecting recent malware on the Android platform. However, Drebin cannot generally prohibit infections with malicious applications, as it builds on concepts of static analysis and lacks dynamic inspection. In particular, transformation attacks that are non-detectable by static analysis, for example, based on reflection and bytecode encryption [see 164, 215, 222], can hinder an accurate detection. To alleviate the absence of a dynamic analysis, Drebin extracts API calls related to obfuscation and loading of code, such as `DexClassLoader.loadClass()` and `Cipher.getInstance()`. These features enable us to at least spot the execution of hidden code—even if we cannot further analyze it.

Besides advanced obfuscation techniques...

In combinations with other features, Drebin is still able to identify malware despite the use of some obfuscation techniques.

To avoid crafting detection patterns manually, we make use of machine learning techniques for generating detection models. While learning techniques provide a powerful tool for automatically inferring models, they require a representative basis of data for training. As we have seen throughout this chapter, the quality of the detection model of Drebin critically depends on the availability of representative malicious and benign applications. However, gathering recent malware samples requires some technical effort. Fortunately, methods for offline analysis, such as DroidRanger [224], AppsPlayground [163], and RiskRanker [97] might help here to automatically acquire malware and provide the basis for updating and maintaining a representative dataset for Drebin over time. Using recent data to continuously update the classification model allows limiting the impact of concept drift, which can otherwise significantly lower the detection performance of Drebin, as demonstrated in Section 5.2.5.

...also targeted attacks against the classification model can impact the detection rate.

Another limitation that follows from the use of machine learning is the possibility of mimicry and poisoning attacks [e.g., 151, 157, 197]. While obfuscation strategies, such as repackaging, code reordering or junk code insertion do not affect Drebin, renaming of activities and components between the learning and detection phase may impair discriminative features [164, 222]. Similarly, an attacker may succeed in lowering the detection score of Drebin by incorporating benign features or fake invariants into malicious applications [151, 157]. Although such attacks against learning techniques cannot be ruled out in general, the thorough sanitization of learning data [see 50] and frequent retraining on representative datasets can limit their impact. Furthermore, we present several improvements to the underlying optimization algorithm of Drebin, which can help to increase its robustness towards these attacks, in the next chapter.

5.5 RELATED WORK

In this chapter, we have performed an extensive evaluation of Drebin and also pointed to several pitfalls that should be considered when designing malware detection experiments. Since the Android malware detection field is a relatively novel strain of research, best practices for designing experiments are continuously discussed and improved in this area. In the following, we give a brief overview of some work in this field.

First of all, the work of Rossow et al. [170] has significantly inspired the design of the experiments presented in this chapter. In their paper, the researchers systematically analyze the experimental design of several popular papers on malware detection and describe prudent practices for the design of these. The guidelines should, in turn, allow other researchers to avoid different flaws that Rossow et al. could identify throughout their study. While other research groups already discussed some of these pitfalls before [121, 128, 182], Rossow et al. have been the first to systematically analyze them and to provide guidelines how to design prudent experiments. We have taken these guidelines into account when designing the experiments presented in this chapter. For example, we have examined the detection performance of malware families separately in Section 5.2.3.

Furthermore, other researchers have more recently begun to study the impact of time on the detection performance of Android malware classifiers [e.g., 112, 130, 137]. One of the first works that demonstrates the impact of time on the detection performance of Android classifiers has been presented by Allix et al. [7]. In particular, the authors demonstrate that the performance of classifiers often significantly decreases over time, as malware evolves continuously. As a reaction to this finding, Mariconti et al. [137] propose the system MaMaDroid, which keeps its detection capabilities for long periods of time. To this end, the system utilizes sequences of abstracted API calls, which are less likely to change over time. However, even their approach cannot fully compensate the impact of concept drift. Thus, continuous retraining of the classifier remains mandatory, which is often computationally expensive.

Hence, in order to avoid unnecessary retraining of classifiers, several researchers have proposed methods that allow detecting concept drift within data [e.g. 59, 112, 113, 134, 180]. An early detection of concept drift helps to decide when a classifier requires retraining. For instance, Maggi et al. [134] present techniques to detect concept drift in web applications, such that learning-based intrusion detection can be retrained only if necessary. More recently, Jordaney et al. [112] propose the *Transcend* framework to identify concept drift in classification tasks. The framework is applicable for different learning algorithms and thus not limited to Android malware detection. Using their approach, they were able to show that the performance decay of different methods, including Drebin, can be significantly reduced by selecting the right time to retrain the classifier.

5.6 chapter summary

Our evaluation has shown that our proposed machine-learning based approach achieves remarkable detection results on the considered datasets. In most experiments, Drebin detects more than 90% of the malicious samples at a low false positive rate of only 1%, thus outperforming signature-based approaches in many cases. While anti-virus scanners generally exhibit a low false positive rate, they have difficulties to detect unknown malicious samples, even if those belong to an already known strain of malware. In contrast, Drebin detects these samples in most cases very reliable. Its detection capabilities only meet their limitations if utterly novel malware families arise, whose characteristics cannot be learned in advance. Frequent retraining of the classifier should, however, alleviate the impact on the detection rate in many cases.

In the second set of experiments, we have also examined the run-time performance of Drebin on different mobile devices. Overall, the analysis of applications takes less than 15 seconds on average and even on ancient devices never longer than 1 minute per application. We consider these run-times to be sufficient for practical usage.

6 MODEL ANALYSIS AND EXPLAINABILITY

Throughout the previous chapters, we were able to show that our proposed learning-based detection method allows deriving a reliable classification model for Android malware detection. However, it remains an open question which of the extracted features are particularly useful and significantly impact the classifier's decision. Analyzing these features cannot only yield essential insights into common malicious patterns, but also ensures that the classifier's decisions are not based upon unwanted artifacts in the underlying data. In other words, a careful feature analysis helps us to assure that the underlying model does not suffer from overfitting towards noise.

Conversely to Drebin, many learning-based approaches are deployed as black-box systems [182]. While this lack of transparency might impede the effort of adversaries to run attacks on the model, it also hinders users and analysts to understand the classifier's decisions. Thus, an analyst might be unable to identify possible flaws in the training and prediction process. This lack of interpretability can, in turn, have serious implications for the overall classifier performance and understanding of the underlying distributions.

Hence, a detection system should maintain good interpretability, stability and be robust against possible attacks at the same time. In this chapter, we examine these properties for Drebin in more detail. To this end, we perform an extensive analysis of the classification model, which can be divided into the following steps:

1. *Explainability.* In the first step, we examine the patterns learned by the classifier. In detail, we consider several of the malware families available in two popular datasets, namely Drebin and AMD, and analyze the most relevant features determined by the algorithm, i.e., features having the highest impact on the classifier's decision.

2. *Model analysis.* In the second step, we examine different properties of the model that provide a better understanding of its inner workings, ultimately allowing us to improve its robustness against targeted attacks further. More precisely, we analyze the impact of different regularizers on the overall stability and selected support vectors of the resulting model.

3. *Attacks on the model.* Finally, the robustness of the classification model towards targeted attacks are considered. Based on the obtained insights, we improve the detection system to make it less prone towards such attacks.

6.1 EXPLAINABILITY

Learning-based detection systems are often deployed as black-box systems, whose decisions cannot be interpreted by users or even experts. This lack of interpretability can in some cases even have serious consequences, thus making such systems not suitable in various application fields. In case of a malware detection system, false alarms might lead to a bad reputation of an affected company whose applications have mistakenly been flagged as malicious. Thus, it is crucial for such a system that its decisions are interpretable.

Fortunately, the underlying machine learning algorithm of Drebin, a linear SVM, allows an interpretation of the obtained results. This is due to the fact that each component w_i of the model vector $\boldsymbol{w}$ can be associated with a specific feature. In the following, we use this property of the classifier to derive the patterns, which led to its decisions in the experiments discussed in the previous chapter. To this end, we examine the following two cases:

1. *Malware types.* Malicious applications can be grouped into different types, like *ransomware* or *SMS malware*. Fortunately, the authors of the AMD dataset provide this information for their gathered samples. We check whether the features extracted by the algorithm match these descriptions.

2. *Malware families.* We analyze a set of famous malware families present in the Drebin dataset. For these families, we determine the most relevant features selected by the learning algorithm and discuss whether these match common descriptions of antivirus vendors.

Before we provide an in-depth discussion of the most relevant features, let us first describe how these features are determined throughout the evaluation.

EXPERIMENTAL SETUP In both scenarios, the procedure of how we obtain the most relevant features is identical for malware families and malware types, respectively. In particular, we first select all samples of the test set belonging to a certain malware family or type. For each feature vector $\boldsymbol{x}$, we then sort its available features $s \in \boldsymbol{x}$ according to their weights w_s assigned by the SVM. Afterward, we pick the ten features with the highest malicious scores. We repeat this procedure for each sample of the considered malware family or type. Finally, the results are averaged and the ten most highly ranked features are picked for discussion. These features are presented to the user with high probability, when scanning an app of the considered malware family.

6.1.1 *Feature Analysis of Malware Types*

The AMD dataset provides information about salient characteristics of the malware families. In the following, we consider different types of malware families and compare the top-ranked features with the functionality provided by the authors of AMD. For these experiments, we examine the models trained on the dataset described in Chapter 5, containing the 24,553 malicious samples of the AMD dataset and 93,635 benign samples.

RANSOMWARE The AMD dataset contains six ransomware families, where the five largest are listed in Table 14. Instead of encrypting files on the device, these families mainly use screen overlays to block the mobile device and hinder users from accessing their data. A feature related to this behavior is the permission *SYSTEM_ALERT_WINDOW*, which has to be granted to the malware in advance, allowing it to display such overlay screens [see 24, 86, 131].

Mobile ransomware often uses display overlays instead of encrypting data on the device.

Malware family	**# Samples**	**Feature Rank**		
		1.	⩽ 5	⩽ 10
Fusob	1,277	✓	✓	✓
Jisut	560		✓	✓
SimpleLocker	173			
Koler	69	✓	✓	✓
Roop	48	✓	✓	✓

Table 14: The permission *SYSTEM_ALERT_WINDOW* often indicates ransomware functionality. The table shows the rank of this feature for the five largest **ransomware** families in the dataset.

In five of these six families, this permission is ranked among the top 5 features and in three cases even on the very top. The only exception is the family *SimpleLocker*, since this malware uses encryption instead of screen overlays to block users from accessing their data. However, no features related to this functionality can be found within the top 10 features. In this case, more advanced features are needed that allow deriving additional insights into the malicious behavior.

SMS-TROJANS A large fraction of malicious applications uses SMS functionality to steal money or sensitive information from unwitting users. Within the AMD dataset, a total of 15 families are known to use SMS functionality for malicious purposes, such as sending SMS messages to premium services.

SMS-Trojans abuse the messaging functionality for different purposes,...

In 11 of these families, the *SEND_SMS* permission is ranked on the very top, including the five largest families presented in Table 15.

Malware family	# Samples	Feature Rank		
		1.	⩽ 5	⩽ 10
FakeInst	2,172	✓	✓	✓
RuMMS	402	✓	✓	✓
SmsKey	165	✓	✓	✓
Gumen	145	✓	✓	✓
Leech	128	✓	✓	✓

Table 15: The *SEND_SMS* permission is often requested by **SMS-Trojans**, thus making it a valuable feature to detect this kind of malware.

...such as sending SMS to expensive premium services.

Moreover, the feature is ranked number 2 for the remaining families. Overall, this example shows that malware still often abuses SMS functionality, thus making the request of the respective permission a valuable feature for its detection.

Malware authors try to get root privileges on the device...

ROOT MALWARE The most dangerous type of malware tries to obtain root privileges, therefore enabling it, in the worst case, to perform arbitrary actions on the device. In total, there exist 23 families in the AMD dataset that yield this kind of behavior. However, while 15 of those families explicitly request the user for granting root privileges, the remaining eight families even try to obtain these privileges through root exploits.

Malware family	# Samples	Feature Rank		
		1.	⩽ 5	⩽ 10
Fusob	1,277		✓	✓
BankBot	648		✓	✓
DroidKungFu	546		✓	✓
RuMMS	402		✓	✓
Lotoor	329	✓	✓	✓

Table 16: Features like the intent filter *DEVICE_ADMIN_ENABLED* or the invocation of */system/bin/su* indicate that an application tries to gain **root access** on the device.

...in order to perform arbitrary actions.

From a total of 15 families requesting admin privileges, the intent filter *DEVICE_ADMIN_ENABLED* is listed among the top 5 features in 12 of them. In case of the family *SimpleLocker*, it is even ranked on the very top. Instead of requesting admin privileges, multiple families also try to exploit known vulnerabilities of the Android OS in order to escalate their privileges. We consider the suspicious calls */system/bin/su* and *Runtime.exec()* as indicative for this behavior. In seven of these eight families, at least one of these features is listed among the top 10 features. In the case of the family *Lotoor*, the *su* command is even ranked on the very top.

6.1.2 *Feature Analysis of Malware Families*

In addition to the analysis of different malware types, we examine the most relevant features for several popular malware families in the next step. To this end, we consider the 5,557 malicious samples from the Drebin$_{\text{ORIG}}$ dataset as well as the 234 samples from the Silverpush dataset. Overall, we discuss the results of four malware families which exhibit different malicious characteristics.

The descriptions are mainly based upon malware reports from other researchers or anti-virus vendors. In some cases, however, we conduct an additional static analysis to retrace the background of some of the ranked features further. For these cases, we provide the *SHA256* hashes of the analyzed samples as foot notes.

DROIDKUNGFU This family is a strain of malware root access on the device. The malware has been first reported in 2011 [111] and is particularly interesting, since it has evolved over time [109, 110], i.e., the authors added several obfuscation techniques, such as encryption of malicious payload. Despite these efforts to disguise the malicious characteristics, Drebin is still able to detect samples of this family reliably, as has been demonstrated in Chapter 5.

DroidKungFu tries to gain root access..

Top 10 features		
Feature s	Feature set	Weight $\lvert w_s \rvert$
`SIG_STR`	S_4 Filtered intents	0.60
`getNetworkInfo()`	S_5 Restricted API calls	0.37
`system/bin/su`	S_7 Suspicious API calls	0.35
`getDeviceId()`	S_5 Restricted API calls	0.32
`getSubscriberId()`	S_5 Restricted API calls	0.26
`Runtime.exec()`	S_7 Suspicious API calls	0.25
`BATTERY_CHANGED_ACTION`	S_4 Filtered intents	0.25
`ACCESS_FINE_LOCATION`	S_6 Used permissions	0.20
`getLine1Number()`	S_5 Restricted API calls	0.17
`getSubscriberId()`	S_7 Suspicious API calls	0.12

Table 17: Top 10 features of the **DroidKungFu** family.

The ten most relevant features are listed in Table 17. The malware tries to exploit several vulnerabilities in earlier Android versions to gain root access and steal sensitive data from the device. Its intention to gain root access is reflected by the feature *system/bin/su* and *Run-*

...and steal sensitive data from the device.

time.exec(). Features like *getLine1Number*, *getSubscriberId*, and *getDeviceId* indicate that the malware tries to access sensitive data, i.e., the phone number, the IMEI, and the IMSI of a device. The two intents *BATTERY_CHANGED_ACTION* and *SIG_STR* are filtered by a broadcast receiver component, which is part of many DroidKungFu samples. Both intents are used to trigger malicious functionality when the application is running in the background.

Besides, newer variants of the malware[1] decrypt and install another APK file hidden in the host malware. Looking for the call *getNetworkInfo* leads to the class containing the decryption routine. The API function itself is invoked within a helper function to check whether an infected device has access to the internet.

FakeInstaller sends SMS to expensive premium services...

FAKEINSTALLER Another malware family whose members steal money from users by sending expensive premium SMS. The family has been widely distributed back in 2012 [172]. Its name is derived from the fact, that the members of this family hide their malicious code inside repackaged versions of legitimate applications, thus tricking users into installing them.

Top 10 features		
Feature s	Feature set	Weight $\lvert w_s \rvert$
`SEND_SMS`	S_2 Requested permissions	0.80
`sendTextMessage()`	S_5 Restricted API calls	0.25
`READ_PHONE_STATE`	S_2 Requested permissions	0.14
`BOOT_COMPLETED`	S_4 Filtered intents	0.12
`WAKE_LOCK`	S_6 Used permissions	0.12
`READ_SMS`	S_2 Requested permissions	0.11
`PHONE_STATE`	S_4 Filtered intents	0.06
`getLine1Number()`	S_5 Restricted API calls	0.06
`SEND_SMS`	S_6 Used permissions	0.05
`getDeviceId()`	S_5 Restricted API calls	0.04

Table 18: Top 10 features of the **FakeInstaller** family.

...and hides its payload inside of legitimate apps.

During the installation process, the malware sends SMS messages to premium services owned by the malware authors. Even on the first sight, four of the extracted features indicate that the malware uses SMS functionality. Like the member of the *DroidKungFu* family, a large fraction of these samples also collects sensitive information, such as the telephone number.

Furthermore, some variants present a notification to the user that a critical update is available each time the device finished its boot-

[1] `25be589140f73949124f08759ab5bb57b126396f1401e3bfbfdc5e5c056e0d03`
`f4ccbe2c567e1f7fd9eaca2a92d5230956aa8b5a2d9b7918656a6a7a913aaaa9`
`af33315dfbf3ed5a0d28e8ed03a79d20c3b77b16129cca9439bb4cbddca076e2`

ing process [140]. To this end, the malware registers a broadcast receiver that waits for the intent message *BOOT_COMPLETED*. Moreover, the permission *WAKE_LOCK* is listed, since several variants use the deprecated Android Cloud to Device Messaging (C2DM) service, allowing the malware authors to remotely push notifications onto the mobile device [84].

GOLDDREAM The members of this family are trojans, which monitor an infected device, collect sensitive data, and record information from received SMS messages and incoming phone calls [108, 187]. This data is later sent to an external server owned by the malware authors.

GoldDream collects sensitive data from the device...

Top 10 features		
Feature s	Feature set	Weight $\lvert w_s \rvert$
`lebar.gicp.net`	S_8 Network addresses	0.37
`PHONE_STATE`	S_4 Filtered intents	0.36
`SMS_RECEIVED`	S_4 Filtered intents	0.35
`getDeviceId()`	S_5 Restricted API calls	0.30
`getSubscriberId()`	S_5 Restricted API calls	0.30
`INSTALL_PACKAGES`	S_2 Requested permissions	0.30
`sendTextMessage()`	S_5 Restricted API calls	0.24
`SEND_SMS`	S_2 Requested permissions	0.23
`getSubscriberId()`	S_7 Suspicious API calls	0.16
`ACCESS_FINE_LOCATION`	S_6 Used permissions	0.14

Table 19: Top 10 features of the **GoldDream** family.

The intent filter *SMS_RECEIVED* directly hints us to the reading of SMS messages. Moreover, the intent filter *PHONE_STATE* indicates that the malware is interested in the information that incoming calls are received. Most of the listed features are related to the collection of sensitive data, such as the IMEI or IMSI. After the malware has collected sufficient data, it sends the data to an external server and waits for additional commands. The hostname of this command and control server is ranked on top of the feature list.

..and waits for further commands from the malware authors.

SILVERPUSH As has already been discussed in Chapter 3, samples belonging to this family listen in the background for ultrasonic beacons and send sensitive information to the company's server. The most expressive features of this family determined by DREBIN are listed in Table 20.

Also in the case of Silverpush...

Many of the relevant features derived by DREBIN are connected to known characteristics of *Silverpush*. In particular, the API call *AudioRecord.init()* as well as the used permission *RECORD_AUDIO* di-

Top 10 features		
Feature s	Feature set	Weight $\|w_s\|$
`getSimCountryIso()`	S_7 Suspicious API calls	0.10
`getLine1Number()`	S_5 Restricted API calls	0.09
`Base64()`	S_7 Suspicious API calls	0.09
`AudioRecord.init()`	S_5 Restricted API calls	0.06
`getConnectionInfo()`	S_5 Restricted API calls	0.05
`getAccountsByType()`	S_5 Restricted API calls	0.04
`RECORD_AUDIO`	S_6 Used permissions	0.04
`app.silverpush.co`	S_8 Network addresses	0.04
`app.silverpush.co/V2/exp`	S_8 Network addresses	0.04
`GET_ACCOUNTS`	S_6 Used permissions	0.03

Table 20: Top 10 features of **Silverpush** family.

...the features point to characteristic functionalities.

rectly link to the recording functionality of these applications. Furthermore, the URL of the company's server is listed in the top features. The remaining features are mainly API calls used by the samples to access sensitive data on the device.

6.1.3 *Discussion*

Overall, the analysis of the most relevant features shows that the SVM indeed selects meaningful features automatically, which can be linked to essential characteristics of many malware types and families. As an example, the SVM assigns high weights to features that are connected to the rooting of a device or the sending of premium SMS.

However, although these features can even help regular users to better understand analyzed applications in some cases, they still often require expert knowledge for full comprehension. Thus, more abstraction is still needed to present meaningful explanations to users. A simple way of achieving this, pose predefined description templates as discussed in Chapter 4 to which relevant features can be mapped. A drawback of this approach is the manual effort needed to continuously keep these descriptions up-to-date. In consequence, further research is required to derive meaningful descriptions from the extracted features automatically.

6.2 MODEL ANALYSIS

We have shown that DREBIN can extract meaningful patterns from malicious applications automatically, thus allowing users and analysts to understand the classifier's decisions. In this section, we illuminate various aspects of the underlying model related to its generalization capabilities and overall size, as these properties can have a significant impact on its applicability in practice. In detail, we examine the following aspects:

1. We analyze the support vectors upon which the classification model is built. These provide information on the overall generalization performance of the resulting model and its expected growth over time.

2. Moreover, we examine the impact of different regularizers, as these can also affect the overall size, generalization performance, and stability of the resulting model.

6.2.1 *Support Vector Analysis*

We start our examination of the underlying model by considering the data points of the training set which define the hyperplane and give the SVM its name—the *support vectors*. In particular, we analyze the number of these vectors in more depth, since it does not only allow us to conclude about the overall size of the resulting model, but can also give us an estimate on the boundary of its generalization error at the same time.

The number of support vectors of an SVM...

According to the SVM learning theory, the relation between the expected value of the generalization error and the number of support vectors is given by [37, 67]

$$E_n[\mathit{error}] \leqslant \frac{E_n[N_s]}{n}, \tag{20}$$

where N_s denotes the number of support vectors selected out of a training set of n samples. The SVM only uses these N_s vectors for the description of the hyperplane. Note that it is possible to remove any of the non-support vectors from the training set, without changing the resulting classification model. In consequence, the ratio between the average number of support vectors N_s and the number of available training samples n allows estimating the generalization capabilities of the SVM for a specific problem domain, i.e., the sensitivity of the classifier towards stochastic noise.

...allows drawing conclusions about its generalization performance.

DATASET To examine the number of support vectors, we take a closer look at the classification models obtained in the *time-based* scenario discussed in Section 5.2.5.

RESULTS Figure 32 shows the results of these experiments. Note that, although the training size grows significantly over time, the number of support vectors increases much slower.

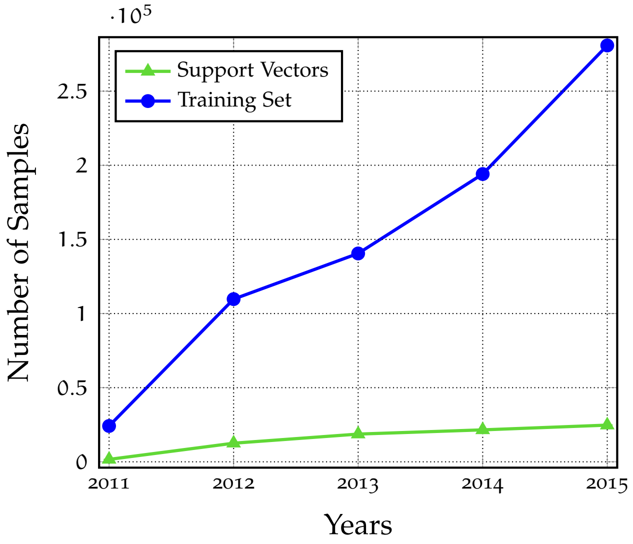


Figure 32: Ratio of samples in training data and selected support vectors.

The SVM selects roughly 10% of the training samples as support vectors.

On average, the SVM selects about 10% of the available training data as support vectors. From this result, we conclude that the SVM indeed generalizes particular characteristics from the training data, instead of simply memorizing the data points of the training set. Thus, it only needs a fraction of the available samples to distinguish between both classes. Moreover, the resulting classifier shows a good detection performance, as we have already examined in Chapter 5.

6.2.2 *Regularization*

The choise of the regularizer...

Previously, we have considered the standard SVM formulation with L2-regularization. Although using this formulation leads to a model with good detection performance, the resulting classification model $\boldsymbol{w}$ is derived through a linear combination of roughly 10% of the training dataset. As a result, its storage requirements tend to grow with the number of available training samples.

As a remedy, we examine an alternative formulation of the SVM, which uses an L1-regularizer instead. This formulation usually leads to sparser models and should thus require less storage. In the following, we evaluate how much the number of features can be reduced when using this formulation. Furthermore, we discuss possible drawbacks that should be considered when preferring one formulation over the other.

L1-REGULARIZATION Before we discuss our evaluation, let us first recapture the differences between using a L1-regularization and a L2-

regularization. Formally, the optimization problem is slightly changed when using an L1-regularizer [74]:

$$\min_{\boldsymbol{w}} \|\boldsymbol{w}\|_1 + C \sum_{i=1}^{n} \max(0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i))^2 . \tag{21}$$

This formula looks very similar to the one presented in Chapter 2. Only the penalty term on $\boldsymbol{w}$ has been replaced by a 1-norm $\|.\|_1$. To give an intuition why this often leads to a sparser solution for $\boldsymbol{w}$, consider Figure 33, which schematically depicts the difference between an L1- and L2-regularization.

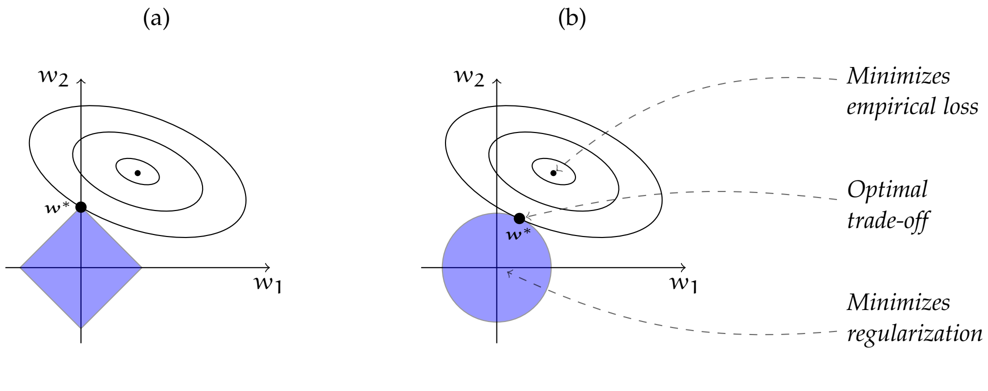


Figure 33: Schematic depiction of difference between (a) L1-Regularization and (b) L2-Regularization.

In both cases, an optimization problem is solved that aims to find proper model parameters $\boldsymbol{w}^*$, such that an optimal balance between the costs caused by misclassification of training data and the regularization penalty is achieved. In both figures, the ellipses depict the contour of the quadratic loss function, while the blue shape refers to the regularization term. Due to the shape of the L1-regularizer, the chance that the best trade-off can be found on one of the axis is significantly higher compared to the L2-regularization term. In our example, the optimal point can be found on the w_2 axis, thus pushing the value for w_1 towards zero. The same principle holds if $\boldsymbol{w}$ contains more dimensions, such as in the case of DREBIN.

...can significantly affect the model sparsity.

6.2.2.1 *Model sparsity*

To evaluate the impact of regularization on the sparsity of the resulting model $\boldsymbol{w}$, we use the same evaluation setup as discussed in the previous section. For brevity, we refer to the SVM with L1-regularizer as *L1-SVM* and to the L2-regularized version as *L2-SVM*.

Figure 34 shows the results of this experiment. In detail, the plots depict the total number of features, and how many of them are selected by the L1- and L2-regularized SVM, respectively. The total number of available features increases between 2011 and 2015 from

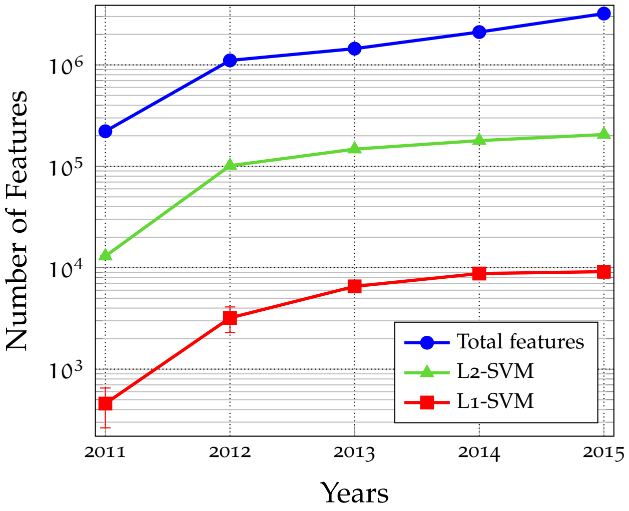


Figure 34: Number of selected features for L1- and L2-regularized SVM.

The L1-SVM reduces the number of features to 0.3%...

221,623 to 3,205,207 features. The L2-SVM already reduces the number of features significantly, selecting roughly 6% of the available features. Thus, only 13,015 and 205,038 features remain in 2011 and 2015, respectively. However, the L1-SVM even reduces this number further, yielding an average number of only 9,147 features in 2015. This means, in consequence, that the L1-SVM only selects about 0.3% of all available features.

6.2.2.2 *Detection rate*

The feature reduction provided by the L1-regularized SVM raises reasonable doubts whether the resulting model still yields an acceptable detection performance compared to the L2-SVM. Therefore, we also examine the respective detection rates and compare them to the results obtained with the L2-regularized classifier.

..without having too much impact on the detection rate.

Table 21 lists the results for this experiment. Surprisingly, the detection performance does not significantly decrease when using the L1-SVM. Instead, the classifier even shows a better performance for the year 2011. Only the variance of the detection performance is in general larger compared to the L2-SVM—especially for 2011 and 2012.

A possible explanation for this observation might be that the L2-regularizer tends to distribute the weight more evenly among features that often occur in combination. In contrast, the L1-regularizer instead tends to pick only one of them, thus reducing the overall number of features. As a result, the classification score of an application can differ significantly, depending on the presence or absence of an important feature.

Regularizer	Year	Precision	Recall	F1-Score	$TPR_{0.01}$
L1	2011	0.85 ± 0.03	0.46 ± 0.05	0.59 ± 0.04	0.46 ±0.05
	2012	0.99 ± 0.01	0.75 ± 0.08	0.85 ± 0.05	0.75 ±0.08
	2013	0.99 ± 0.00	0.82 ± 0.01	0.90 ± 0.00	0.82 ±0.01
	2014	0.99 ± 0.00	0.82 ± 0.01	0.90 ± 0.01	0.83 ±0.01
	2015	0.96 ± 0.00	0.87 ± 0.01	0.91 ± 0.01	0.88 ±0.02
L2	2011	0.89 ± 0.00	0.35 ± 0.00	0.50 ± 0.00	0.38 ±0.02
	2012	0.99 ± 0.00	0.74 ± 0.00	0.85 ± 0.00	0.74 ±0.00
	2013	0.98 ± 0.00	0.86 ± 0.00	0.92 ± 0.00	0.86 ±0.00
	2014	0.99 ± 0.00	0.87 ± 0.00	0.93 ± 0.00	0.87 ±0.00
	2015	0.92 ± 0.00	0.87 ± 0.01	0.91 ± 0.01	0.87 ±0.01

Table 21: Detection rates for different regularizers.

6.2.3 *Discussion*

Let us summarize the results of this section. At the beginning of the section, we considered the number of support vectors, since it allows estimating the generalization performance of the SVM, i.e., whether the algorithm indeed learns the overall concept of the underlying distribution instead of just memorizing the samples in the training set. In case of Android malware detection, the obtained results show a significant reduction in the number of samples necessary to describe the underlying model. However, the number of support vectors tends to grow together with the overall size of the training set.

Hence, we conclude that the number of support vectors also has an important practical implication in our application field: A smaller number of support vectors leads to sparser classifiers which, in turn, require less disk space on the device. As we figured out during further experiments, the choice of a regularization term significantly affects the size of the resulting classification model.

In particular, when using an L2-regularizer, the SVM selects on average about 6% of the available features in the training dataset, while using an L1-regularizer even further reduces this number to only 0.3%. Thus, the model derived with the L1-SVM requires much less disk space compared to the L2-SVM. At the same time, comparable detection rates can be achieved, making the L1-model a good alternative when detecting malware directly on the device.

6.3 ATTACKS AGAINST MACHINE LEARNING

With increasing application of learning-based systems...

By now, we have demonstrated in several experiments that the linear SVM algorithm is able to generalize common malicious patterns, thus enabling it to identify variants of known malware families reliably. The underlying algorithm, however, has not initially been designed with security in mind. In particular, we assumed that the malware authors do not specifically craft their malicious samples such that they can circumvent a machine learning based detection system. With the growing popularity and deployment of machine learning techniques in security applications, attackers will likely try to attack these systems. In this section, we discuss possible attacks against DREBIN and present effective countermeasures.

Attacks against machine learning are not a novel research field, but have already been studied by many researchers within the last two decades [104, 132, 154]. Moreover, the interest in this field has grown significantly in the past few years, especially since machine learning algorithms have become ubiquitous in a wide range of different application fields, including autonomous driving [127, 226], recommendation systems [49, 95], or malware detection [114, 129].

6.3.1 *Attack Scenarios*

While many different attacks on machine learning based systems exist, a large fraction of them can be assigned to two attack classes. These classes mainly differ regarding the point in time at which attackers may try to influence the detection system:

...also grows the possibility of attacks against them.

- *Poisoning Attacks.* In this type of attack [25, 151, 197], adversaries try to attack the learning algorithm throughout the *training phase*. To this end, they manipulate the training data in order to impair the performance of the resulting model. Although this kind of attack can indeed significantly affect the accuracy of machine learning based systems, it also requires a powerful adversary who controls the underlying distribution from which the training data is sampled from.

- *Evasion Attacks.* In contrast to poisoning attacks, evasion attacks attempt to circumvent machine learning based systems at *test time* [83, 157]. The attackers manipulate malicious samples such that these are misclassified by the system as legitimate applications. In many cases, it is assumed that the attacker does neither have access to the (complete) training data nor to the classification model.

Overall, we consider it unlikely, though not impossible, to face an attacker that has the capabilities assumed in the poisoning attack scenario. Thus, we leave the development of proper defenses against poisoning attacks to future research and focus on evasion attacks instead, as the required prerequisites are more likely to occur in practice. In order to examine this kind of attack in more detail, we first define a theoretical framework, which allows us to analyze these attacks in more detail. Using this framework, we then evaluate the liability of Drebin to these attacks and discuss possible defenses.

6.3.2 *Evasion Attacks*

In the following, we perform a stepwise derivation of a framework that enables us to examine the robustness of the classification model. For the sake of simplicity, we make various assumptions that are to the advantage of the attacker, though unlikely to hold in practice. Ultimately, this attack model should allow us to get an upper bound for the attacker's capabilities and develop effective defenses. In particular, we assume the following attack scenario:

Crafting proper defenses against strong attackers..

1. We consider an attacker with perfect knowledge, i.e., an attacker who knows the full classification model w. This assumption is rather unlikely to hold in practice, since the attacker would need to have background knowledge of all feature used throughout the training phase of the model. However, if it is possible to find appropriate defense strategies against this kind of attacker, these defenses should also be effective against adversaries with less background knowledge.

2. We assume that the attacker can manipulate all features with the same effort. In practice, however, this assumption might often not be correct, as we will discuss in more detail later in this section. Consider, for instance, the permissions requested by an app. In most cases, these cannot be removed by an adversary without restricting the app's functionality at run-time.

..makes the algorithm also robust against realistic attackers

Before presenting the theoretical framework to evaluate evasion attacks against Drebin, we first discuss the possibilities and challenges an attacker faces when trying to manipulate a malicious application such that it affects the score of the decision function.

feature manipulation Since Drebin operates on binary vectors, the adversary can either add or remove features in order to circumvent the detection system. In the following, we discuss the challenges of both possibilities.

Adding features is in general feasible, in particular, when injecting manifest features. For instance, requesting additional permissions does not impact any existing application functionality. For the dex code, the attacker can safely introduce information that is not actively

Adding features is often easier for an adversary..

executed, by adding code after *return* instructions (*dead code*) or with methods that are never invoked. Listing 3 shows an example of a method that introduces a URL feature, but is actually never called by the application.

```
.method public addUrlFeature()V
 .locals 2
 const-string v1, "http://www.example.com"
 invoke-direct {v0, v1},
    ↪ Ljava/net/URL;-><init>(Ljava/lang/String;)V
 return-void
.end method
```

Listing 3: Smali code for adding a URL feature.

However, this only applies when such information is not directly executed by the application, and could in principle be stopped at the parsing level by analyzing only the methods belonging to the application call graph. Given that the device has enough computational power to conduct such an analysis, attackers would be enforced to change the executed code, which adds further constraints. For example, when adding an API call to a dex code method that is executed by the application, attackers need to pay attention that they do not introduce undesired artifacts that influence the semantics of the original program. Accordingly, injecting a large number of features may not always be feasible.

..than removing features

Compared to the addition of features, their removal is even more complicated. For instance, removing permissions from the manifest is often not possible, as this can limit the applications functionality. The same holds for intent filters. Some application component names can be changed but the attacker must ensure that the component names in the dex code are changed accordingly.

With respect to the dex code, multiple ways can be exploited to remove its features. For instance, it is possible to hide network addresses by storing them as encrypted strings and decrypting them at run-time using additionally introduced functions. However, this should be done by avoiding the addition of features that are already used by the system and indicate malicious functionality. Similarly, it is possible to hide suspicious and restricted API calls, for instance, by invoking them at run-time through reflection. In this case, the attacker also needs to be careful not to introduce other calls that might increase the suspiciousness of the application.

For the reasons mentioned above, performing a fine-grained evasion attack that changes many features may be difficult in practice, without affecting the malicious application's functionality. In addition, another problem for the attacker is getting to know precisely which features should be added or removed without having access to the classification model or the training data. This becomes particularly a problem to the attacker if the classifier is continuously retrained, as is the case of DREBIN.

EVASION ATTACK ALGORITHM With the discussed background knowledge on the attacker's capabilities at hand, we can finally describe a framework to examine the robustness of the classifier. In particular, we assume that the attacker tries to perform a minimal number of modifications to an application z such that the resulting sample $z^\star$ gets misclassified by the classification function f. Formally, this procedure can be described with:

The attacker yields to modify a malicious sample..

$$z^\star = \arg\min_{z' \in \Omega(z)} f(\mathbf{x}'), \tag{22}$$

where $\mathbf{x}'$ denotes the feature vector of a modified sample z'. This application, in turn, has been derived from its unaltered version z, by applying a modification from the space of all valid modifications $\Omega(z)$. Since we consider a linear classification model, this equation simplifies further to

$$\boldsymbol{z}^\star = \arg\min_{z' \in \Omega(z)} \boldsymbol{w}^\top \mathbf{x}' . \tag{23}$$

Here, $\boldsymbol{w}$ describes the linear classifier. Using this principle formulation we can finally design a more specific evasion attack on DREBIN and examine its impact on the detection performance of the underlying classifier. Given the assumption that attackers can remove and add arbitrary features to evade the classifier, they can perform the following steps to circumvent a binary classifier as is used by DREBIN. In the first step, all weights stored in the model vector $\boldsymbol{w}$ are sorted in descending order of their absolute value $w_{(1)}, \ldots, w_{(d)}$ with $|w_{(1)}| \geqslant \ldots \geqslant |w_{(d)}|$. In the next step, we can modify the feature vector $\mathbf{x}$ of an application for $k = 1, \ldots, d$:

..such that it gets misclassified as benign

- if $x_{(k)} = 1$ and $w_{(k)} > 0$, we set $x_{(k)}$ to zero;
- if $x_{(k)} = 0$ and $w_{(k)} < 0$, we set $x_{(k)}$ to one;
- else $x_{(k)}$ is left unmodified.

Note that the attacker yields to perform as little modifications as possible and thus only continues his attack until $\mathbf{x}$ is flagged as benign by the classification model or the maximum number d of modification is reached.

RESULTS Figure 35 shows the results for the L1-SVM and the L2-SVM. Both classifiers have been trained on the DREBIN$_{\text{ORIG}}$ dataset as described in Chapter 5 using the exact same splits for training, validation, and testing to ensure comparability. In absence of any attacker,

The modification of a small number of features..

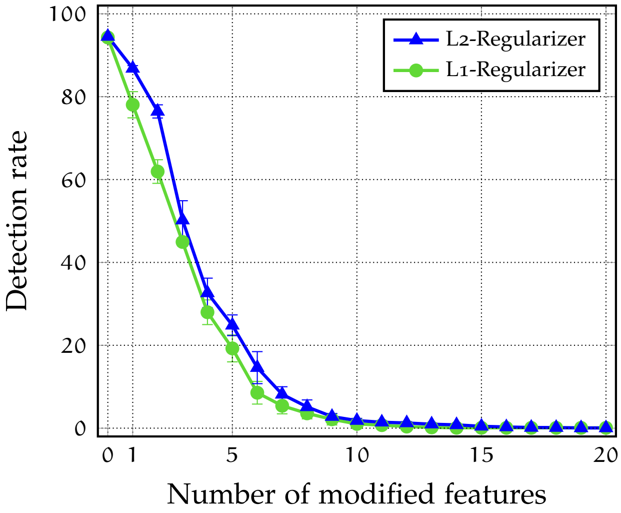


Figure 35: Comparison of L1- and L2-regularized SVM under attack.

both classifiers achieve high detection rates. However, as soon as the attacker starts to modify even a small number of features, the detection performance drops significantly for both classifiers. In particular, when modifying just a single feature, the detection rate for the L1-SVM and the L2-SVM drops by 16% and 8%, respectively. Moreover, one can observe that the L1-SVM is more prone to the attack and its performance thus drops faster than for the L2-SVM.

..already leads to a significant drop in detection rate

The reason for this observation is not surprising, since the regularization with the L1-norm results in a classification model that has much fewer features than the L2-regularized model. While this is a valuable property regarding disk space efficiency, it has the drawback of resulting in a less stable model, as the absence or presence of a particular feature can have a high impact on the resulting classification score of the detection system.

The L1-SVM is more prone towards these attacks than the L2-SVM

This interpretation is also confirmed by the weight distribution of both models, as is depicted in Figure 36. The L1-regularized model contains on average only around 5,000 features, while the L2-SVM has more than 100,000 feature weights. Note that, however, most of the weights of the L2-SVM are close to zero. The weights of the L1-SVM are much less equally distributed than those of the L2-regularized version, resulting in a model more prone to evasion attacks, as has been confirmed throughout our experiments.

Overall, it is necessary to find a good trade-off between the number of selected features, interpretability, and stability of the resulting model. Since the L2-SVM requires much more features, but the L1-SVM results in a less stable model, the question remains whether it is possible to train a classifier that combines the strengths of both models. To this end, we examine the Sec-SVM in the next section, which promises a good trade-off between detection performance, interpretability, and model stability.

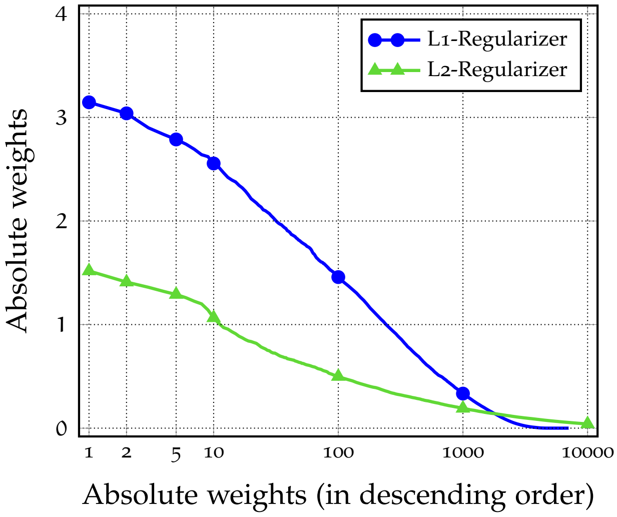


Figure 36: Weight distribution of L1- and L2-regularized SVM.

6.3.3 *Defenses against Evasion Attacks*

The evasion attacks discussed in the previous section have shown that an attacker is already able to evade the classifier of DREBIN by modifying a small number of features of an Android application. Although even the modification of few features might be challenging in practice (see Section 6.3.2), it seems necessary to improve the algorithm such that these attacks require much more effort on the attackers' side.

An unequal distribution of features weights...

The main problem, why these attacks are successful, lies in the unequal distribution of feature weights in the underlying classification model w. Thus, it becomes possible for an attacker to evade the classifier by just modifying a small number of features with high weights. In the following, we discuss two approaches to alleviate the impact of evasion attacks and compare them with the unmodified version of the algorithm.

...makes the method prone towards evasion attacks.

MULTIPLE CLASSIFIER SYSTEM The first approach is based on the combination of the output of multiple, independently trained Support Vector Machines as proposed by Biggio et al. [146]. The individual classifiers are trained using different, randomly selected features. This is a common technique known as the *random subspace method*, which is also part of several other popular learning methods like, for instance, random forests [35]. Furthermore, the training of the classifiers is conducted using different data sets, uniformly sampled from the complete training set with replacement (also a standard technique known as *bootstrapping* [69]). Finally, the obtained outputs of all classifiers are averaged and used for the overall decision (a technique known as bootstrap *agg*reat*ing*, short *bagging* [34]).

Using this approach leads to more evenly distributed weights and thus alleviates the impact of evasion attacks. Note, that averaging the output of all classifiers is equivalent to using a linear classifier whose weights are the average of the weights of the base classifiers, respectively. With this simple trick, the computational complexity at test time remains thus equal to that of a single linear classifier [146]. Thus, the approach only needs more computational resources during training, as it requires to train multiple classifiers in parallel.

The Sec-SVM sets boundaries for the feature weights

THE SEC-SVM ALGORITHM The original SVM algorithm solves the optimization problem without considering an adversary who tries to evade the resulting classifier. To overcome this drawback of the original SVM formulation, we modify the algorithm such that the optimization constraints bound the feature weights into a meaningful interval defined by the vectors $\boldsymbol{w}^{lb}$ and $\boldsymbol{w}^{ub}$. The formulation then becomes:

$$\min_{\boldsymbol{w}} \quad \tfrac{1}{2}\boldsymbol{w}^{\top}\boldsymbol{w} + C\sum_{i=1}^{n} \max\left(0, 1 - y_i \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_i\right) , \tag{24}$$

$$\text{s. t.} \quad w_k^{lb} \leqslant w_k \leqslant w_k^{ub} , \, k = 1, \ldots, d . \tag{25}$$

Therefore, this optimization problem just differs from the original SVM formulation by the presence of a box constraint on $\boldsymbol{w}$. The lower and upper bounds on $\boldsymbol{w}$ are defined by the vectors $\boldsymbol{w}^{lb} = (w_1^{lb}, \ldots, w_d^{lb})$ and $\boldsymbol{w}^{ub} = (w_1^{ub}, \ldots, w_d^{ub})$, which are selected during training with a suitable procedure [57].

EVALUATION SETUP We evaluate the effectiveness of the proposed defenses using a dataset comparable to the DREBIN$_{\text{ORIG}}$ dataset. The data contains 121,329 benign and 5,615 malicious samples, including all malware samples from the DREBIN$_{\text{ORIG}}$ dataset. The samples have been labeled using the VIRUSTOTAL service. In particular, samples flagged by at least 5 antivirus scanners are considered as malicious and only samples without any flags are labeled as benign. During the training step, we randomly select 30,000 applications from this dataset and use them to learn a classification model. All remaining samples are used for testing. This procedure is repeated 10 times and the obtained results are finally averaged.

When running Drebin on the given dataset, more than one million features are extracted by the static analysis. However, as discussed in the previous section, even when using an L2-regularizer, most of these features get assigned a weight close to zero and thus do not have significant impact on the classifier's decision. For computational efficiency, we retain only the most discriminant d' features with the highest values on the training data. Throughout our evaluation, we noticed that using only $d' = 10,000$ features does not significantly affect the accuracy of Drebin. This is consistent with the recent findings of Roy et al. [171], as it is shown that only a very small fraction of features is significantly discriminant, and usually assigned a non-zero weight by the SVM learning algorithm. For the same reason, the sets of selected features turned out to be the same in each run.

For the sake of a fair comparison among different SVM-based learners, we set $C = 1$ for all classifiers and repetitions. For the MCS-SVM classifier, we train 50 base linear SVMs on random subsets of 80% of the training samples and 50% of the features, as this ensures a sufficient diversification of the base classifiers, providing more evenly-distributed feature weights. The bounds of the Sec-SVM are selected through a 5-fold cross-validation. In particular, for each feature weight w_i we optimize the two scalar values $(w_i^{ub}, w_i^{lb}) \in \{0.1, 0.5, 1\} \times \{-1, -0.5, -0.1\}$.

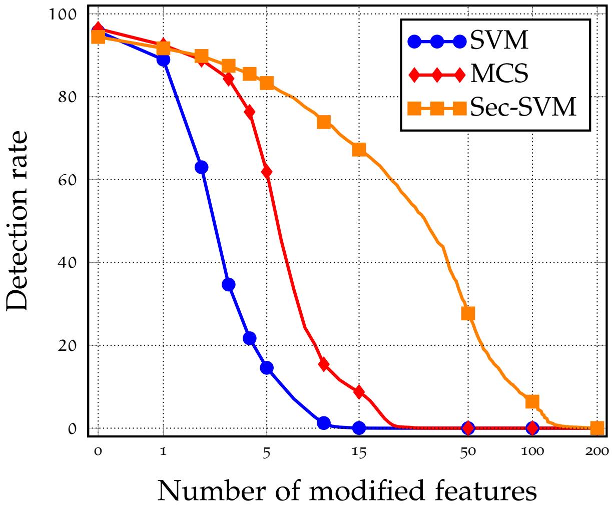


Figure 37: Comparison of different classifiers under attack.

Sec-SVM and MCS-SVM show a higher robustness against the attack.

results We perform an evasion attack using the approach described in Section 6.3, except for the difference that the adversary can only add but not remove features from the manifest file. The underlying idea behind this restriction is the fact that it is difficult for an adversary to remove features like the requested permissions in practice (see Section 6.3.2).

The results of our experiments are depicted in Figure 37. In the absence of an attacker, all methods show similar performance and yield a detection rate of about 94% to 96% at a false positive rate of 1%. The Sec-SVM performs slightly worse than the other two algorithms, while the MCS-SVM yields the best results. However, the detection performance of all classification models differs significantly in the presence of an attacker.

In particular, for the vanilla SVM algorithm, the decrease in detection performance is similar to the results presented in the previous section. For instance, when only two features are modified by the attacker, the detection performance already decreases to about 63%. This result lies in between the detection rates obtained with the L1-

SVM and L2-SVM presented in the previous section, which yield a detection performance of 62% and 76%, respectively. Note that the SVM examined in this experiment is L2-regularized, but trained on a subset of the 10,000 most relevant features. In consequence, it is not surprising that the obtained results are quite similar to those achieved with the L1-regularized SVM.

The Sec-SVM improves the classifier robustness significantly..

The other two approaches show higher robustness against the performed evasion attack. Using the MCS-SVM, the attacker already needs to modify five features to reduce the detection performance to 62% and around 20 features to push it towards zero. In contrast, the Sec-SVM still detects about 62% of the malicious samples when 20 features have been modified by the attacker. Overall, the robustness of the classifier is doubled when using an MCS-SVM and even tenfold when applying the Sec-SVM.

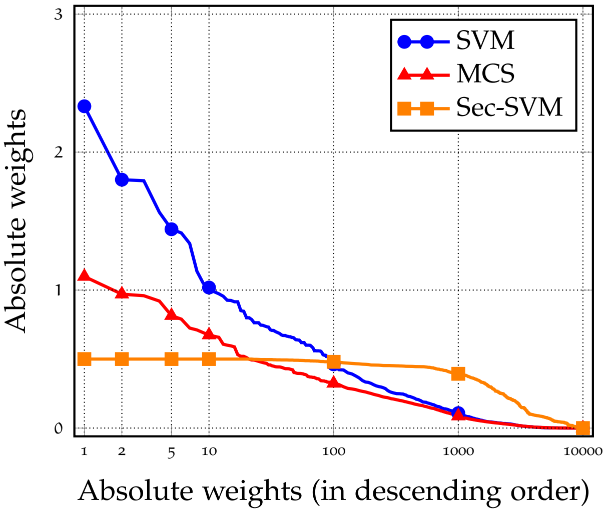


Figure 38: Comparison of weight distribution of different classifiers.

..by distributing the weights more evenly.

The reason for the increased level of security lies in the more evenly distributed weights, as can be derived from Figure 38. Note that the Sec-SVM exhibits on average a maximum absolute weight value of 0.5. This means that, in the worst case, modifying a single feature leads to a decrease of 0.5 of the classification score. In contrast, the decrease is significantly higher for the MCS-SVM and the SVM, with maximum values of approximately 1 and 2.5, respectively. It is thus apparent that it requires a larger number of modified features to evade the Sec-SVM than is needed for the other two detection methods.

6.3.4 *Discussion*

With the growing number of application fields for machine learning algorithms, the risk for these systems to be attacked increases at the same time. Hence, it is crucial to have possible attack scenarios in mind when designing such systems.

In this section, we discussed possible evasion attacks on DREBIN and how they can affect the detection performance of the system. To this end, we first examined the capabilities attackers might have in practice and proposed an attack framework that allows evaluating the impact of such an attack on the classifier. Unfortunately, we experience a significant drop in the detection rate when an attacker can manipulate features that have been assigned a high weight by the classifier. When using an L1-regularizer, the detection performance decreases even faster than with the L2-SVM.

As a remedy, we examine the performance of a slightly changed formulation of the SVM that yields to distribute the weights more evenly among the available features. As a result, the success of the attacker is significantly affected, since more features need to be manipulated until a sample is misclassified. In particular, an attacker has to manipulate ten times more features compared to the unprotected case. Moreover, the performance of the underlying system is only slightly affected and yields detection rates comparable to those obtained when using the original formulation of the SVM.

Of course, even though the Sec-SVM significantly raises the bar for attackers, it still gets to its limits with increasing capabilities of an adversary. Furthermore, it is not capable of dealing with more advanced obfuscation strategies, such as encryption or reflection, as it still relies on static analysis. Therefore, it also has the inherent weaknesses of the original approach.

6.4 RELATED WORK

ATTACKS AGAINST MACHINE LEARNING Machine learning has become an important technique for solving tasks in many domains, including security-critical ones, such as spam detection [100, 150], intrusion detection [122, 167, 201], and malware detection [51, 114, 169, 206]. However, the underlying machine learning algorithms of these systems have initially been designed under the assumption that training and test data follow a stationary distribution. As a result, these systems are often vulnerable to well-crafted attack targeting the underlying machine learning algorithm [20, 21].

Several researchers presented attacks against learning-based systems [e.g., 40, 54, 185, 213, 228]. One of the first works in this area has been done by Dalvi et al. [52], who developed a formal framework to evaluate evasion attacks. Using this framework, they performed a successful attack against spam classifiers and strengthened the learning-algorithm accordingly. In their paper, Dalvi et al. focussed on evasion attacks, which often have the underlying assumption that the attacker has perfect knowledge of the model. While this assumptions seems unrealistic at first glance, there exist several works that demonstrate how machine learning models can be reconstructed by an attacker [132, 193, 200]. Therefore, Biggio et al. [26] proposed a framework to systematically examine evasion attacks by adversaries with varying background knowledge. More recently, Szegedy et al. [188] successfully demonstrated a perturbation attacks against Deep Neural Networks (DNNs). In this attack, some of the pixels within an image are imperceptibly modified such that the content of the image gets misclassified by the model.

In addition to evasion attacks, also different poisoning attacks have been proposed in recent years [e.g., 25, 82, 107, 151, 157, 211]. As an example, Perdisci et al. [157] performed a successful attack against worm signature classifiers by injecting noise into the training data. Similarly, Xing et al. [211] demonstrated how the recommendation systems of popular Web services can be manipulated by injecting specific information into users' profiles.

In response to the growing number of attacks, many researchers also suggested defenses against attacks on machine learning models [e.g., 58, 107, 154, 181]. Related to the defense discussed in this chapter, Demontis et al. [58] analyzed how the selection of the regularizer can improve the robustness of SVM classifiers towards evasion attacks. Furthermore, the examination of defenses against evasion attacks specifically targeting Deep Neural Networks is currently a vivid research field. For instance, Papernot et al. [154] proposed a defense against this kind of attack by obfuscating the gradients of neural networks. Not even a year later, however, Carlini and Wagner [42] were able to show that the defense is ineffective. As a reaction to their finding, other researchers presented further defenses [see 155, 212]. Unfortunately, it has already been shown that many of them do also not provide proper protection [12, 41]. Thus, it is likely that this cat-and-mouse game will continue in the near future, hopefully resulting in more robust learning methods.

EXPLAINABILITY In order to effectively thwart the success of attacks against learning-based systems, it is essential to understand the decision-making process of the underlying models. However, while explaining the decision of linear models is straightforward [8, 168], it is difficult to derive meaningful explanations for decisions made by non-linear classifiers. Therefore, different approaches have been proposed to close this gap [e.g. 17, 19, 118, 141, 165]. As an example, Koh et al. [118] utilized influence functions from statistics to identify training points that had the highest impact on the resulting decision. Instead of deriving explanation from the training data, Ribeiro et al. [165] used interpretable approximations of a classifier to understand its decisions. Similarly, Bach et al. [17] proposed a solution that allows understanding the decisions of non-linear image classification systems, including SVM-based systems. Specifically, their approach produces heatmaps that visualize the contribution of single pixels to the overall decision, thus enabling human experts to verify the decisions of the machine learning system.

Furthermore, there exist several works that focus specifically on improving the explainability in the Android malware domain [e.g., 80, 153, 220]. In particular, Pandita et al. presented the framework WHYPER [153], which employs Natural Language Processing (NLP) techniques to improve application descriptions. To this end, it automatically identifies sentences in an application description that justify the requested permissions of the considered application. Remotely related, Zhu et al. described their system FeatureSmith, which automatically engineers proper feature sets for Android malware detection by mining scientific papers for relevant malware features. Using these features, they were able to build a classifier that has a detection performance comparable to DREBIN. Moreover, Feng et al. presented the tool ASTROID [80]. ASTROID constructs signatures for the detection of malware families automatically. For this purpose, it identifies common subgraphs within the inter-component call graphs of malware family members [79]. Similar to DREBIN, the authors showed that these signatures allowed them to conclude about inherent characteristics of these families. Most recently, Melis et al. [141] proposed a gradient-based method that determines the most relevant features even for non-linear models, i.e., generalizing the approach presented in Section 6.1.

6.5 CHAPTER SUMMARY

In this chapter, we have conducted an extensive analysis of the underlying classification model of DREBIN, including the interpretability of its decisions and the overall stability of its underlying classification model. Besides, we have discussed attacks on the model and also presented suitable defenses. In the following, we summarize the results of each of these aspects individually.

At the beginning of this chapter, we have analyzed the features that had the highest impact on the classifier's decision. For this purpose, we have used two popular malware datasets, namely DREBIN and AMD, and have examined the decisions of the classifier for several popular malware families. Overall, we notice that the features selected by the classifier for its decision, resemble common knowledge about these families in many cases. However, even though the features can already be useful for malware analysts, further research is needed to improve the explanations of the classifier further.

In the second step, we have inspected the overall model stability along with the number of features selected by the classifier. We find that the SVM provides good detection results, while requiring only a small amount of support vectors to achieve this performance. Thus, we conclude that the method is indeed suitable for detecting Android malware in general. Besides, we find that the number of required features can be reduced further by using an L1-regularization, without significantly affecting the overall detection performance of the classifier. Therefore, the size of the required classification model remains small, even when derived from a dataset containing hundred of thousands of data points.

Finally, we have also examined the classifier's robustness towards evasion attacks. Throughout our experiments, we have noticed that attackers can spoof the learning model by changing certain features of a malicious samples such that it gets misclassified as benign. Fortunately, an attacker requires precise knowledge of the underlying model or training data, which should often be infeasible in practice. Moreover, we could identify other constraints that attackers are faced with. Still, we aimed to improve the robustness of the underlying algorithm towards this kind of attack. As part of a thorough analysis, we have found that the unequal distribution of feature weights favors the success of an attacker. As a corollary to this finding, we have modified the SVM algorithm such that it distributes its weights more evenly, leading to a classification model that is more robust than the original version of the algorithm.

7 CONCLUSION

Mobile malware poses a threat to the security and privacy of smartphone users. In recent years, the number of malware for the Android operating system has grown significantly, thus rendering common signature-based approaches often ineffective in detecting new malware instances reliably. Research on novel technologies to effectively counter this threat is therefore of paramount importance for the security of mobile devices.

In this thesis, we have provided insights into recent developments in the Android malware landscape, especially discussing a new malware that uses an ultrasonic side channel to spy on unwitting smartphone users. Based on the knowledge we have gained during our research, we were able to develop a new method for malware detection that does not suffer from typical drawbacks of signature-based approaches. Specifically, our proposed method combines concepts of static analysis and machine learning, thus allowing the automatic derivation of suitable detection patterns from a large number of applications. As a result, the approach can even detect unknown malware instances in many cases.

In this thesis, we have proposed a new method...

Throughout an extensive evaluation, we have shown that the proposed method outperforms several related approaches, including popular anti-virus scanners. In particular, it offers high detection rates with only a few false positives and, contrary to other learning-based approaches, runs directly on the mobile device. Another advantage of the proposed method is that it is not a black-box system, but instead provides explanations of its decisions to the user.

...that detects mobile malware with machine learning techniques.

In summary, we have shown that machine learning techniques can help to improve the security of mobile devices. Although our approach cannot completely prevent the threat of mobile malware, it raises the bar for malicious actors to compromise mobile devices significantly. Still, more research is needed to improve malware detection systems further.

7.1 SUMMARY OF RESULTS

In the following section, we summarize the main results of this thesis in more detail. Afterward, we discuss open questions that require further research.

CHAPTER 1 The first chapter provides information about the significant increase in the number of Android malware samples found in recent years. While in the early days of Android, malware for this platform was rare and thus mainly of academic interest [175], antivirus vendors nowadays find thousands of malicious samples every day. Unfortunately, existing solutions for Android malware detection can often not identify unknown samples, and thus turn out to be ineffective in many cases. In consequence, there is an urgent need for new methods that allow reliable detection and provide the ability to handle large amounts of data. A promising research direction is therefore the utilization of machine learning techniques, as these allow the automatic derivation of characteristic structures and patterns from large datasets. The development of such a learning-based method has been the ultimate goal of this thesis.

Chapter 1 discusses the motivation of this thesis.

Chapter 2 In this chapter, we have equipped ourselves with the necessary background knowledge to stepwise develop a learning-based method for Android malware detection. Specifically, we have introduced basic concepts of the Android operating system in the first part of this chapter, mainly focussing on some of the considerations behind its security design. Moreover, we have discussed current solutions for Android malware detection along with their benefits and drawbacks. In the second part of the chapter, we have presented basic concepts of machine learning theory, including the idea of *regularization* and the mathematical fundamentals behind the SVM algorithm.

Chapter 2 provides some background information.

Chapter 3 Equipped with the necessary background knowledge, we have presented our findings on ultrasonic side channels in Android applications. In particular, we have investigated three commercial solutions that use ultrasonic beacons for various purposes. During the inspection of the inner workings of different apps, we have been able to identify several malicious characteristics within the tracking technology of one of these companies. Most importantly, these applications carry the functionality to listen for ultrasonic beacons in the background without the user's knowledge or consent. Therefore, they can potentially monitor users' TV viewing habits, track their visited locations, or deduce other devices of a user.

Chapter 3 discusses an ultrasonic side channel...

To examine the prevalence of this technology in the wild, we have developed two different tools. The first tool allows scanning for applications that carry the ultrasonic tracking functionality; the second tool detects ultrasonic signals in common media, such as audio and video files. We have used these tools to analyze several hours of multimedia data and more than 1.3 million applications. While we have been able to identify 234 applications that carry the respective tracking functionality to spy on unwitting users, we could not find any ultrasonic beacons of the respective company. Instead, we have found ultrasonic beacons from the two other companies in various media files and 4 European stores. However, in contrast to the illicit use of

...that has already been misused by malware.

this technology, users are aware of the tracking in these cases. In summary, the results of our study show that the technology has already been actively used and still offers potential to be misused by malware authors in the future.

It is noteworthy that the tool we used for scanning the applications has a serious drawback. Although it is very efficient in detecting specific code regions within a large number of applications, it requires manual effort to identify the necessary code regions in advance. Consequently, it is not feasible for detecting Android malware in general.

CHAPTER 4 To overcome the limitations of the detection method discussed in Chapter 3, we have proposed a new method in this chapter. Our method, called DREBIN, is based on concepts of static analysis and machine learning, which enables it to keep better pace with malware development than signature-based methods. In addition, it runs directly on the mobile device. To detect mobile malware, DREBIN first extracts various static information from an application, including, for instance, the permissions that an app requests. In the second step, it maps the application into a high-dimensional vector space. Finally, it classifies the app as malicious or benign by applying a detection model on the app's vector representation. For this purpose, it uses a high-dimensional hyperplane as a classification model that separates benign and malicious data in that vector space. In contrast to many other common detection methods, DREBIN provides explanations for its decisions, which can help users to understand its assessment on analyzed applications.

In Chapter 4, we present a new method to detect mobile malware.

CHAPTER 5 This chapter provides an extensive evaluation of the detection capabilities and the run-time performance of DREBIN. For the evaluation, we have used several datasets, which contain more than 400,000 different Android applications in total.

In the first part of this chapter, we have examined the detection performance of DREBIN and compared it against various other solutions for Android malware detection, including popular anti-virus scanners. DREBIN outperforms related approaches in most cases, achieving detection rates between 92% and 99% at a low false positive rate of only 1%. Interestingly, the approach achieves a detection rate of 99% for the malware family discussed in Chapter 3, thus proving its effectiveness in deriving useful detection patterns automatically. We have only noticed that the approach has problems in detecting malware families if no members of a malware family are available during training. In this case, the learning algorithm is unable to extract crucial detection patterns from the training data. However, the detection performance significantly increases in most cases, as soon as only a few samples of a malware family become available for training.

We evaluate its detection and run-time performance in Chapter 5.

In the second part of the chapter, we have evaluated the run-time performance of Drebin. To this end, we have implemented a prototype application, which has enabled us to measure the average time Drebin needs to output a decision. In an experiment in which we have tested the analysis time of 100 different applications on five different devices, we have found that it takes less than 15 seconds on average to output a decision for an application. Moreover, even on very outdated devices, it never takes longer than a minute. We consider these run-times to be sufficient for practical usage.

Chapter 6 In this chapter, we have performed a comprehensive analysis of the detection model of Drebin. Specifically, we have analyzed the interpretability and robustness of the underlying classifier.

In Chapter 6, we have examined its interpretability...

In the first part, we have examined the most relevant features of different malware families selected by the learning algorithm, and have compared them with common knowledge about these families. Overall, we have found that the selected features often reflect important characteristics of the considered families, and pose a promising direction for future research.

..., generalization capabilities,...

In the second part of the chapter, we have analyzed the generalization capabilities of the detection model. In particular, we have estimated the generalization performance of the classifier by analyzing the selected number of support vectors. The SVM only selects roughly 10% of the data points as support vectors, thus indicating an excellent generalization performance. By using an L1-regularizer, we have been able to reduce the number of features further, i.e., the SVM only selects about 0.3% of the available features when adapting the optimization problem accordingly.

...and how to harden it against attacks.

In the third part, we have examined the robustness of the underlying model towards possible attacks. For this purpose, we have assumed an attacker who tries to evade the detection system by modifying the features of a malicious application, such that it gets classified as benign. In multiple experiments, we have demonstrated that a strong attacker with perfect knowledge of the underlying model only needs to modify a small number of features to circumvent the classifier. To prevent the success of such an attacker, we have shown that the feature weights of the classification model need to be distributed more equally. Therefore, we have discussed how the optimization problem of the SVM can be modified accordingly, resulting in the Sec-SVM formulation. The Sec-SVM shows much higher robustness and still detects about 62% of the malware even if many features have been modified by the attacker.

7.2 LIMITATIONS AND FUTURE WORK

In this thesis, we have shown that machine learning techniques can help to significantly improve malware detection systems. However, while the obtained results are promising, there is still space for improvement and future research. We discuss some possible research directions in the following section.

CROSS-PLATFORM TRANSFER While we have solely focussed on the Android operating system throughout this thesis, it should be possible to adapt the proposed methods to other operating systems. This is of particular importance, as malware authors and advertising companies already target users across different platforms and devices (see Chapter 2 and Chapter 3). Hence, transferring technologies between different operating systems is crucial to ensure a comprehensive protection of mobile device users.

SOFTWARE VULNERABILITIES More research is needed to enhance the security of the Android operating system in general, as this can significantly lower the risks induced by malware. For instance, several critical vulnerabilities in the Android operating system have been identified by researchers in the past [e.g., 66, 78, 86], and many others are most likely still undetected. These vulnerabilities are, in turn, used by malicious applications to escalate their privileges on infected devices [94]. This is possible, as Android still suffers from strong fragmentation and thus vulnerabilities often remain unpatched (see Chapter 2). In recent years, various methods have been proposed to speed up the finding of vulnerabilities [e.g., 14, 158, 178, 216]. However, only few consider the peculiarities of Android [e.g., 86].

DYNAMIC ANALYSIS Throughout the evaluation, we were able to show the efficacy of our method in detecting malware on the Android platform. However, DREBIN cannot generally prevent infections with malicious applications, as it builds on concepts of static analysis and lacks dynamic inspection. By extending the approach with dynamic analysis techniques, it should even be possible to detect malicious behaviour despite the use of advanced obfuscation techniques, such as Java reflection or dynamic code loading. Several researchers [e.g., 130, 224] have already successfully shown that a combination of static and dynamic analysis techniques can improve the detection performance of such a system. Moreover, several sophisticated de-obfuscation techniques have been proposed, which could also help to improve the detection capabilities of DREBIN [e.g., 46, 85, 162].

Unfortunately, the run-time overhead induced by these techniques is currently still too high to run such a method directly on the device. In the future, however, smart devices will most likely offer enough computational power to extend Drebin with this functionality. Until then, the problem can at least be alleviated by adding proper features that are difficult to hide by malicious applications, as has already been demonstrated by other researchers [e.g., 6, 89, 184].

MODEL ROBUSTNESS As we have discussed in Chapter 5, malware continuously changes over time, thus leading to changes in the underlying distributions. In machine learning, this phenomenon is referred to as concept drift. The reasons for the occurrence of concept drift are manifold, ranging from malware evolution (e.g., adaption to new Android versions) to even targeted evasion attacks against the detection systems of mobile devices.

To lower the impact of concept drift, the detection model of Drebin requires periodic retraining to keep pace with the development of mobile malware. However, the number of retraining steps should be minimized mainly for two reasons. Firstly, retraining a learning model is often computationally expensive. Secondly, users have to download updated models every time, thus possibly resulting in high costs for the user. To alleviate the impact of concept drift, Jordaney et al. [112] have proposed a framework that allows detecting when a classification model has become outdated. While detecting concept drift is an important step, further research is needed to build more robust models that stay up-to-date for longer periods of time, like, for instance, the approach proposed by Mariconti et al. [137]. Extending Drebin with more robust features therefore remains essential future work.

The extension of Drebin with more robust features will, however, not be sufficient to solve this problem completely, as also the robustness of the underlying learning-algorithms needs to be examined and improved further. We have demonstrated that targeted attacks against the detection model are possible and might become a serious threat for learning-based systems in the near future. While we have shown that it is possible to improve the robustness of the models against evasion attacks significantly, we did not, for instance, consider *poisoning attacks* in this thesis. Furthermore, it remains an open questions whether and how the use of non-linear classification models can improve the security of these systems.

EXPLAINABILITY The last point concerns the explainability of the underlying detection model. Specifically, as learning-based systems often suffer from false positives, it is essential that users understand their decisions. Otherwise, users are unable to decide whether or not an application might indeed exhibit malicious characteristics. Unlike other learning-based detection systems, Drebin already provides explanations for its decisions to the user. However, these are likely still too technical for regular users in many cases. A promising research direction to produce more intuitive explanations might, for instance, offer the field of topic modeling. By using techniques like Latent Dirichlet Allocation (LDA) [29], it might become possible to output possible behaviors of an application, instead of just presenting individual features to the user.

A APPENDIX

A.1 SUSPICIOUS APPLICATIONS

This table contains the list of samples analyzed for the experiments in Section 5.2.6.

MALICIOUS SAMPLES Analyzed samples that have been flagged by less than 10 anti-virus scanners.

```
02575e14ae866a42d36b23a0d30567dd0afb630a06bf0d6297655fa08f7abcc1
0272c5e34f68d2d4d960fdbf8212dbdd51ae1c7961b7bcb113ad2be9909392e9
0475c2f3f0d5bff8bc2f056a8ab44f7d5eb13260cc425946c2863c19af794a82
049458f5665388ba455b129d0ff4bf17ad63f8b990bec1534794a0f5aa28181c
05350bd8feeec42ec586f94e9accd330f553bdbcc3cbb714c6b1598032aa79ac
05c179ff9e579efdc6cc4266e27618136e2983e31372b156278149bef6c1cba7
06f3db64723727d1bf68344311fb85cf668cf9aeac4b69584af621fe38108b5e
07be6853d807f133098224e8e1e4ed29ee8080b598113b76d506f98648b385b9
07ea68f0ecabe5e05dc6f12c9b0cb37f3368700f66c64a25283765414271de2b
09197ce2875d7fa37b437d84d59f5d5e550e9ff6500acb38a07b9aa15bf45bc5
0ae9a3a5e123ec3005e9e8bc5023157a890b71493b095e48878de681f21d28aa
0b4d3e416a60f663e8edcc4a566023220bd795bbd7d3318cda3f2c9871f8c357
0bdb7f20af2ab6a2a6139d18b16b50e47a1799845b36ccd3399cd7c385aa352f
0c1c27b6d1c22dcc03f8f2cddfbe25a59cc3b599669e8e1eb08aae4d39a972e2
0d80a4d42e383a2080895b5b0ab2c78360081bc9bc9247d083efa62b268c976b
10aace8e5f1f555087fb44001521b67817917bb8d83e0dfe679d9fc526007ad7
```

BENIGN SAMPLES Analyzed samples that have not been flagged by any anti-virus scanner.

```
9ec1e7cfb035524e85cbc049a123d8f9afcb41979d56d967e1ca7c22c1e94009
b45400124c4137a6e6e4207901b7ffec96ea5048a41d6a07698fa80d31d803f8
e4b985cda2832ea04a19f6fe85084fb76a734742b5d4b5eb40912ea8cd9ce91a
```

BIBLIOGRAPHY

[1] Yousra Aafer, Wenliang Du, and Heng Yin. "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android." In: *Proc. of Int. Conference on Security and Privacy in Communication Networks (SECURECOMM)*. 2013.

[2] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. "Precise Android API Protection Mapping Derivation and Reasoning." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2018.

[3] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[4] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick D. McDaniel, and Matthew Smith. "SoK: Lessons Learned from Android Security Research for Appified Software Platforms." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2016.

[5] Tomi T. Ahonen. *Installed base of smartphones by operating system from 2015 to 2017 (in million units)*. `https://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/`. (last visited Mar. 13, 2019). 2018.

[6] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. "DroidNative: Automating and optimizing detection of Android native code malware variants." In: *Computers & Security* 65 (2017).

[7] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "Are Your Training Datasets Yet Relevant? - An Investigation into the Importance of Timeline in Machine Learning-Based Malware Detection." In: *Engineering Secure Software and Systems (ESSoS)*. 2015.

[8] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2014.

[9] Daniel Arp, Erwin Quiring, Christian Wressnegger, and Konrad Rieck. "Privacy Threats through Ultrasonic Side Channels on Mobile Devices." In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017.

[10] Daniel Arp, Erwin Quiring, Tammo Krueger, Stanimir Dragiev, and Konrad Rieck. “Privacy-Enhanced Fraud Detection with Bloom filters.” In: *Proc. of Int. Conference on Security and Privacy in Communication Networks (SECURECOMM)*. 2018.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps.” In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2014.

[12] Anish Athalye, Nicholas Carlini, and David Wagner. “Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples.” In: *Proc. of Int. Conference on Machine Learning (ICML)*. 2018.

[13] 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*. Proc. of ACM Conference on Computer and Communications Security (CCS). 2012.

[14] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. “AEG: Automatic Exploit Generation.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2011.

[15] Avira. *Android.SmsAgent.YW.Gen*. `https://www.avira.com/en/support-threats-summary/tid/145007/threat/Android.SmsAgent.YW.Gen`. (last visited Mar. 13, 2019). 2016.

[16] John Aycock. *Computer Viruses and Malware*. Springer Publishing Company, Incorporated, 2010.

[17] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, Wojciech Samek, and Oscar Deniz Suarez. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation.” In: *PLOS ONE* 10.7 (2015).

[18] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. “On Demystifying the Android Application Framework: Re-visiting Android Permission Specification Analysis.” In: *Proc. of USENIX Security Symposium*. 2016.

[19] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert Müller. “How to Explain Individual Classification Decisions.” In: *Journal of Machine Learning Research (JMLR)* 11 (2010).

[20] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. "Can Machine Learning Be Secure?" In: *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2006.

[21] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. "The security of machine learning." In: *Machine Learning* 81.2 (2010).

[22] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. "A methodology for empirical analysis of permission-based security models and its application to Android." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2010.

[23] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. "Static detection of malicious code in executable programs." In: *International Journal of Requirements Engineering* (2001).

[24] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. "What the App is That? Deception and Countermeasures in the Android User Interface." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[25] Battista Biggio, Blaine Nelson, and Pavel Laskov. "Poisoning Attacks against Support Vector Machines." In: *Proc. of Int. Conference on Machine Learning (ICML)*. 2012.

[26] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. "Evasion Attacks against Machine Learning at Test Time." In: *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. 2013.

[27] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[28] David M. Blei. "Probabilistic Topic Models." In: *Communication of the ACM (CACM)* 55.4 (2012).

[29] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation." In: *Journal of Machine Learning Research (JMLR)* (2003).

[30] B.H. Bloom. "Space/time trade-offs in hash coding with allowable errors." In: *Communication of the ACM (CACM)* 13.7 (1970).

[31] Hristo Bojinov, Dan Boneh, Yan Michalevsky, and Gabi Nakibly. *Mobile Device Identification via Sensor Fingerprinting*. arXiv preprint. 2014.

[32] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers." In: *Proc. of the Annual Workshop on Computational Learning Theory (COLT)*. 1992.

[33] Andrew P. Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms." In: *Pattern Recognition* 30.7 (1997).

[34] Leo Breiman. "Bagging Predictors." In: *Machine Learning* 24.2 (1996).

[35] Leo Breiman. "Random Forests." In: *Machine Learning* 45 (2001).

[36] Bill Brenner. *Android malware anti-emulation techniques*. `https://news.sophos.com/en-us/2017/04/13/android-malware-anti-emulation-techniques/`. (last visited Mar. 13, 2019). 2017.

[37] Christopher J. C. Burges. "A Tutorial on Support Vector Machines for Pattern Recognition." In: *Data Mining and Knowledge Discovery* 2 (1998).

[38] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: Behavior-Based Malware Detection System for Android." In: *Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. 2011.

[39] Zhenquan Cai and Roland H.C. Yap. "Inferring the Detection Logic and Evaluating the Effectiveness of Android Anti-Virus Apps." In: *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*. 2016.

[40] Alejandro Calleja, Alejandro Martín, Héctor D. Menéndez, Juan E. Tapiador, and David Clark. "Picking on the family: Disrupting android malware triage by forcing misclassification." In: *Expert Systems with Applications* 95 (2018).

[41] Nicholas Carlini and David Wagner. "Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2017.

[42] Nicholas Carlini and David Wagner. "Towards Evaluating the Robustness of Neural Networks." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2017.

[43] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. "MAST: Triage for Market-scale Mobile Malware Analysis." In: *Proc. of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*. 2013.

[44] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. "Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale." In: *Proc. of USENIX Security Symposium*. 2015.

[45] Mihai Christodorescu and Somesh Jha. "Static Analysis of Executables to Detect Malicious Patterns." In: *Proc. of USENIX Security Symposium*. 2003.

[46] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[47] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[48] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks." In: *Machine Learning*. 1995.

[49] Paul Covington, Jay Adams, and Emre Sargin. "Deep Neural Networks for YouTube Recommendations." In: *Proc. of the ACM Conference on Recommender Systems (RECSYS)*. 2016.

[50] G. Cretu, A. Stavrou, M. Locasto, S.J. Stolfo, and A.D. Keromytis. "Casting out Demons: Sanitizing Training Data for Anomaly Sensors." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2008.

[51] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. "Zozzle: Fast and Precise In-Browser JavaScript Malware Detection." In: *Proc. of USENIX Security Symposium*. 2011.

[52] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. "Adversarial Classification." In: *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*. 2004.

[53] Damballa. *State of Infections Report: Q4 2014*. Tech. rep. Damballa, 2015.

[54] Hung Dang, Yue Huang, and Ee-Chien Chang. "Evading Classifiers by Morphing in the Dark." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2017.

[55] Anupam Das, Nikita Borisov, and Matthew Caesar. "Do You Hear What I Hear? Fingerprinting Smart Devices Through Embedded Acoustic Components." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2014.

[56] Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

[57] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. "Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection." In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2017).

[58] Ambra Demontis, Paolo Russu, Battista Biggio, Giorgio Fumera, and Fabio Roli. "On Security and Sparsity of Linear Classifiers for Adversarial Settings." In: *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop (S+SSPR)*. 2016.

[59] Amit Deo, Santanu Kumar Dash, Guillermo Suarez-Tangil, Volodya Vovk, and Lorenzo Cavallaro. "Prescience: Probabilistic Guidance on the Retraining Conundrum for Malware Detection." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2016.

[60] Luke Deshotels. "Inaudible Sound as a Covert Channel in Mobile Devices." In: *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*. 2014.

[61] Anthony Desnos. *Androguard*. `https://github.com/androguard`. (last visited Mar. 13, 2019). 2018.

[62] Anthony Desnos and Geoffroy Gueguen. "Android: From Reversing to Decompilation." In: *Proc. of Black Hat Abu Dhabi*. 2011.

[63] Google Developers. *Requesting Permissions at Run Time*. `http://developer.android.com/training/permissions/requesting.html`. (last visited Mar. 13, 2019).

[64] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. "Exfiltrating Data from Android Devices." In: *Computers and Security* 48 (2015).

[65] Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. *Android Hacker's Handbook*. Wiley Publishing, 2014.

[66] Joshua Drake. *Stagefright: Scary Code in the Heart of Android*. Black Hat USA. 2015.

[67] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[68] Peter Eckersley. "How Unique is Your Web Browser?" In: *Proceedings on Privacy Enhancing Technologies (PETS)*. 2010.

[69] Bradley Efron. "Bootstrap Methods: Another Look at the Jackknife." In: *The Annals of Statistics* 7.1 (1979).

[70] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014.

[71] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. "On lightweight mobile phone application certification." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2009.

[72] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[73] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security." In: *Proc. of USENIX Security Symposium*. 2011.

[74] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. "LIBLINEAR: A Library for Large Linear Classification." In: *Journal of Machine Learning Research (JMLR)* 9 (2008).

[75] Tom Fawcett. "An introduction to ROC analysis." In: *Pattern Recognition Letters* 27.8 (2006).

[76] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A Survey of Mobile Malware in the Wild." In: *Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. 2011.

[77] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android permissions demystified." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2011, pp. 627–638.

[78] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. "Permission Re-delegation: Attacks and Defenses." In: *Proc. of USENIX Security Symposium*. 2011.

[79] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. "Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis." In: *Proceedings of the ACM Symposium on Foundations of Software Engineering (FSE)*. 2014.

[80] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. "Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[81] Kevin Finisterre. *SilverPushUnmasked*. `https://github.com/MAVProxyUser/SilverPushUnmasked`. (last visited Mar. 13, 2019).

[82] Prahlad Fogla and Wenke Lee. "Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2006.

[83] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. "Polymorphic Blending Attacks." In: *Proc. of USENIX Security Symposium*. 2006.

[84] Fortinet. *Android/FakeInst.C!tr*. `http://fortiguard.com/encyclopedia/virus/3928646`. (last visited Mar. 13, 2019). 2012.

[85] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. "TriggerScope: Towards Detecting Logic Bombs in Android Applications." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2016.

[86] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2017.

[87] Jesus Freke. *Smali - A disassembler for Android's DEX format*. `https://github.com/JesusFreke/smali/wiki`. (last visited Mar. 13, 2019).

[88] Jessica Fridrich. "Sensor Defects in Digital Image Forensic." In: *Digital Image Forensics: There is More to a Picture Than Meets the Eye*. Springer, 2013.

[89] Joshua Garcia, Mahmoud Hammad, and Sam Malek. "Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2018).

[90] Gartner. *Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018*. `https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/`. (last visited Mar. 13, 2019). 2018.

[91] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Structural Detection of Android Malware using Embedded Call Graphs." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2013.

[92] I. Gashi, B. Sobesto, S. Mason, V. Stankovic, and M. Cukier. "A study of the relationship between antivirus regressions and label changes." In: *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 2013.

[93] AV-TEST GmbH. *Security Report 2015/16*. 2017.

[94] Kaspersky Lab GmbH. *Mobile Malware Evolution 2016*. 2017.

[95] Carlos A. Gomez-Uribe and Neil Hunt. "The Netflix Recommender System: Algorithms, Business Value, and Innovation." In: *ACM Transactions on Management Information Systems (TMIS)* 6 (2015).

[96] Google/Ipsos. *How people discover, use, and stay engaged with apps*. `https://think.storage.googleapis.com/docs/how-users-discover-use-apps-google-research.pdf`. (last visited Mar. 13, 2019). 2016.

[97] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. "RiskRanker: scalable and accurate zero-day android malware detection." In: *Proc. of International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. 2012.

[98] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection." In: *Journal of Machine Learning Research (JMLR)* 3 (2003).

[99] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. "Gene Selection for Cancer Classification using Support Vector Machines." In: *Machine Learning* 46 (2002).

[100] Thiago S. Guzella and Walmir M. Caminhas. "A review of machine learning approaches to Spam filtering." In: *Expert Systems with Applications* 36.7 (2009).

[101] Petter Hallmo, Arne Sundby, and Iain WS Mair. "Extended High-frequency Audiometry: Air- and Bone-conduction Thresholds, Age and Gender Variations." In: *Scandinavian Audiology* 23.3 (1994).

[102] Michael Hanspach and Michael Goetz. "On Covert Acoustical Mesh Networks in Air." In: *Journal of Communications* 8.11 (2013).

[103] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. "A Dual Coordinate Descent Method for Large-scale Linear SVM." In: *Proc. of Int. Conference on Machine Learning (ICML)*. 2008.

[104] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. Doug Tygar. "Adversarial Machine Learning." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2011.

[105] Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. "On the Robustness of Mobile Device Fingerprinting: Can Mobile Users Escape Modern Web-Tracking Mechanisms?" In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2015.

[106] Médéric Hurier, Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[107] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. "Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[108] Xuxian Jiang. *Security Alert: New Android Malware – GoldDream – Found in Alternative App Markets*. `https://www.csc2.ncsu.edu/faculty/xjiang4/GoldDream/`. (last visited Mar. 13, 2019). 2011.

[109] Xuxian Jiang. *Security Alert: New DroidKungFu Variant – AGAIN! – Found in Alternative Android Markets*. `https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu3/`. (last visited Mar. 13, 2019). 2011.

[110] Xuxian Jiang. *Security Alert: New DroidKungFu Variants Found in Alternative Chinese Android Markets*. `https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu2/`. (last visited Mar. 13, 2019). 2011.

[111] Xuxian Jiang. *Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets*. `https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html`. (last visited Mar. 13, 2019). 2011.

[112] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. "Transcend: Detecting Concept Drift in Malware Classification Models." In: *Proc. of USENIX Security Symposium*. 2017.

[113] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. "Approaches to Adversarial Drift." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2013.

[114] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. "Revolver: An Automated Approach to the Detection of Evasive Web-based Malware." In: *Proc. of USENIX Security Symposium*. 2013.

[115] Karen Kay. *Fancy that hat Rihanna's wearing on TV? Shazam wants to help you track it down*. `https://www.theguardian.com/technology/2013/mar/30/shazam-app-tv-viewers-advertisers`. (last visited Mar. 13, 2019).

[116] Quentyn Kennemer. *Spammy ads in the notification bar die this week as Google's latest Play Store changes take effect*. `http://phandroid.com/2013/09/30/google-play-notification-ads-policy/`. (last visited Mar. 13, 2019). 2013.

[117] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection." In: *Proc. of USENIX Security Symposium*. 2014.

[118] Pang Wei Koh and Percy Liang. "Understanding Black-box Predictions via Influence Functions." In: *Proc. of Int. Conference on Machine Learning (ICML)*. 2017.

[119] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. "Certified PUP: Abuse in Authenticode Code Signing." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[120] L. F. Kozachenko and N. N. Leonenko. "Sample Estimate of the Entropy of a Random Vector." In: *Problems of Information Transmission* 23.2 (1987).

[121] Balachander Krishnamurthy and Walter Willinger. "What Are Our Standards for Validation of Measurement-based Networking Research?" In: *ACM SIGMETRICS Performance Evaluation Review (PER)* 36.2 (2008).

[122] Christopher Krügel, Thomas Toth, and Engin Kirda. "Service Specific Anomaly Detection for Network Intrusion Detection." In: *Proc. of ACM Symposium on Applied Computing (SAC)*. 2002.

[123] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. "Fingerprinting Mobile Devices Using Personalized Configurations." In: *Proceedings on Privacy Enhancing Technologies (PETS)* 2016.1 (2016).

[124] Google LLC. *Android Security 2016 Year In Review*. 2017.

[125] Google LLC. *Android version market share distribution among smartphone owners as of September 2018*. `https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/`. (last visited Mar. 13, 2019). 2018.

[126] Hyewon Lee, Tae Hyun Kim, Jun Won Choi, and Sunghyun Choi. "Chirp Signal-Based Aerial Acoustic Communication for Smart Devices." In: *IEEE Conference on Computer Communications (INFOCOM)*. 2015.

[127] Jesse Levinson et al. "Towards fully autonomous driving: Systems and algorithms." In: *Proc. of IEEE Intelligent Vehicles Symposium (IV)*. 2011.

[128] Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter. "On Challenges in Evaluating Malware Clustering." In: *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2010.

[129] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem A. Beyah. "Acing the IOC Game: Toward Automatic Discovery and Analysis of Open-Source Cyber Threat Intelligence." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2016.

[130] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. "MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis." In: *Proc. of the IEEE Annual Computer Software and Applications Conference (COMPSAC)*. 2015.

[131] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors." In: *Proc. of the Int. Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. 2014.

[132] Daniel Lowd and Christopher Meek. "Adversarial Learning." In: *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*. 2005.

[133] Christian Lueg. *Cyberangriffe auf Android-Geräte nehmen stark zu*. `https://www.gdata.de/blog/2018/11/31254-cyberangriffe-auf-android-gerate-nehmen-stark-zu`. (last visited Mar. 13, 2019). 2018.

[134] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. "Protecting a Moving Target: Addressing Web Application Concept Drift." In: *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2009.

[135] Davide Maiorca, Giorgio Giacinto, and Igino Corona. "A Pattern Recognition System for Malicious PDF Files Detection." In: *Proc. of the Int. Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*. 2012.

[136] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. "Stealth attacks: An extended insight into the obfuscation effects on Android malware." In: *Computers & Security* 51.C (2015).

[137] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[138] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. "(Sp)iPhone: Decoding Vibrations from Nearby Keyboards Using Mobile Phone Accelerometers." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2011.

[139] Vasilios Mavroudis, Shuang Hao, Yanick Fratantonio, Federico Maggi, Christopher Kruegel, and Giovanni Vigna. "On the Privacy and Security of the Ultrasound Ecosystem." In: *Proceedings on Privacy Enhancing Technologies (PETS)* (2017).

[140] McAfee. *Android/FakeInstaller.L.* `https://home.mcafee.com/virusinfo/`. (last visited Aug. 1, 2018). 2012.

[141] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. "Explaining Black-box Android Malware Detection." In: *European Signal Processing Conference (EUSIPCO)*. 2018.

[142] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. "Gyrophone: Recognizing Speech from Gyroscope Signals." In: *Proceedings of USENIX Security Symposium*. 2014.

[143] Brad Miller et al. "Reviewer Integration and Performance Measurement for Malware Detection." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[144] N. Miramirkhani, M. Appini, N. Nikiforakis, and M. Polychronakis. "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2017.

[145] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Limits of Static Analysis for Malware Detection." In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2007.

[146] "Multiple classifier systems for robust classifier design in adversarial environments." In: *Int. Journal of Machine Learning and Cybernetics (IJMLC)* 1.1 (2010).

[147] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. "BareDroid: Large-Scale Analysis of Android Apps on Real Devices." In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2015.

[148] Naked Security. `http://www.naked-security.com/malware/Android.Gappusin/`. (last visited Mar. 13, 2019). 2012.

[149] Arvind Narayanan and Vitaly Shmatikov. "Robust De-anonymization of Large Sparse Datasets." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2008.

[150] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I. P. Rubinstein, Udam Saini, Charles Sutton, J. D. Tygar, and Kai Xia. "Exploiting Machine Learning to Subvert Your Spam Filter." In: *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. 2008.

[151] James Newsome, Brad Karp, and Dawn Xiaodong Song. "Paragraph: Thwarting Signature Learning by Training Maliciously." In: *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2006.

[152] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2013.

[153] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *Proc. of USENIX Security Symposium*. 2013.

[154] Nicolas Papernot, Patrick Drew McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2016.

[155] Nicolas Papernot, Patrick D. McDaniel, Arunesh Sinha, and Michael P. Wellman. "SoK: Security and Privacy in Machine Learning." In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018.

[156] 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: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2012.

[157] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M.I. Sharif. "Misleading Worm Signature Generators Using Deliberate Noise Injection." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2006.

[158] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[159] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware." In: *Proc. of the European Workshop on System Security (EUROSEC)*. 2014.

[160] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android

Applications." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2014.

[161] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. "Paranoid Android: Versatile Protection For Smartphones." In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2010.

[162] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2016.

[163] V. Rastogi, Y. Chen, and W. Enck. "AppsPlayground: Automatic Security Analysis of Smartphone Applications." In: *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*. 2013.

[164] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. "DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks." In: *Proc. of ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*. 2013.

[165] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?: Explaining the Predictions of Any Classifier." In: *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*. 2016.

[166] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems." In: *Transactions of the American Mathematical Society (AMS)* 74.2 (1953).

[167] Konrad Rieck. "Machine learning for application layer intrusion detection." PhD thesis. Berlin Institute of Technology, 2009.

[168] Konrad Rieck, Tammo Krueger, and Andreas Dewald. "Cujo: Efficient Detection and Prevention of Drive-by-download Attacks." In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2010.

[169] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. "Learning and Classification of Malware Behavior." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2008.

[170] Christian Rossow, Christian Dietrich, Chris Gier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. "Prudent Practices for Designing Malware Experiments: Status Quo and Outlook." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2012.

[171] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. "Experimental Study with Real-world Data for Android App Security Analysis using Machine Learning." In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2015.

[172] Fernando Ruiz. *'FakeInstaller' Leads the Attack on Android Phones*. `https://securingtomorrow.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones/`. (last visited Mar. 13, 2019). 2012.

[173] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. "Android permissions: a perspective combining risks and benefits." In: *Proc. of ACM symposium on Access Control Models and Technologies (SACMAT)*. 2012.

[174] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. "Item-based Collaborative Filtering Recommendation Algorithms." In: *Proc. of the International World Wide Web Conference (WWW)*. 2001.

[175] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Hendrik Clausen, Osman Kiraz, Kamer Ali Yüksel, Seyit Ahmet Çamtepe, and Sahin Albayrak. "Static Analysis of Executables for Collaborative Malware Detection on Android." In: *Proc. of IEEE International Conference on Communications (ICC)*. 2009.

[176] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[177] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S.V.N. Vishwanathan. "Hash Kernels for Structured Data." In: *Journal of Machine Learning Research (JMLR)* 10 (2009).

[178] Y. Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016.

[179] Zarah Simon. *Adwares. Are they viruses or not?* `http://androidmalwareresearch.blogspot.de/2012/07/adwares-are-they-viruses-or-not.html`. (last visited Mar. 13, 2019). 2012.

[180] Anshuman Singh, Andrew Walenstein, and Arun Lakhotia. "Tracking Concept Drift in Malware Families." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2012.

[181] Charles Smutz and Angelos Stavrou. “When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2016.

[182] R. Sommer and V. Paxson. “Outside the Closed World: On Using Machine Learning for Network Intrusion Detection.” In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2010.

[183] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. “Mobile-sandbox: Having a Deeper Look into Android Applications.” In: *Proc. of ACM Symposium on Applied Computing (SAC)*. 2013.

[184] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. “DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware.” In: *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*. 2017.

[185] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daumé III, and Tudor Dumitras. “When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks.” In: *Proc. of USENIX Security Symposium*. 2018.

[186] Mingshen Sun, Xiaolei Li, John C.S. Lui, Richard T.B. Ma, and Zhenkai Liang. “Monet: A User-oriented Behavior-based Malware Variants Detection System for Android.” In: *IEEE Transactions on Information Forensics and Security (TIFS)* (2016).

[187] Symantec. *Android.Golddream*. `https://www.symantec.com/security_response/writeup.jsp?docid=2011-070608-4139-99&tabid=2`. (last visited Mar. 13, 2019). 2011.

[188] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. “Intriguing properties of neural networks.” In: *International Conference on Learning Representations (ICLR)*. 2014.

[189] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[190] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. “CopperDroid: Automatic Reconstruction of Android Malware Behaviors.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2015.

[191] O. Tange. “GNU Parallel - The Command-Line Power Tool.” In: *;login: The USENIX Magazine* (2011).

[192] New York Times. *That Game on Your Phone May Be Tracking What You’re Watching on TV*. `https://www.nytimes.com/2017/12/28/business/media/alphonso-app-tracking.html`. (last visited Mar. 13, 2019). 2017.

[193] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Stealing Machine Learning Models via Prediction APIs." In: *Proc. of USENIX Security Symposium*. 2016.

[194] Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. "FlashDetect: ActionScript 3 Malware Detection." In: *Proc. of Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2012.

[195] V.N. Vapnik. *Estimation of Dependences Based on Empirical Data [in Russian]*. Nauka, 1979.

[196] V.N. Vapnik and A.Y. Chervonenkis. *Theory of Pattern Recognition: Statistical Problems of Learning [in Russian]*. Nauka, 1974.

[197] Shobha Venkataraman, Avrim Blum, and Dawn Song. "Limits of Learning-based Signature Generation with Adversaries." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2008.

[198] Kaveh Waddell. *Your Phone Is Listening – Literally Listening – to Your TV*. `http://www.theatlantic.com/technology/archive/2015/11/your-phone-is-literally-listening-to-your-tv/416712/`. (last visited Mar. 13, 2019). 2015.

[199] Avery Li-Chun Wang. "An Industrial-Strength Audio Search Algorithm." In: *International Symposium on Music Information Retrieval (ISMIR)*. 2003.

[200] Binghui Wang and Neil Zhenqiang Gong. "Stealing Hyperparameters in Machine Learning." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[201] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. "Anagram: A Content Anomaly Detector Resistant To Mimicry Attack." In: *Proc. of International Symposium on Recent Adances in Intrusion Detection (RAID)*. 2006.

[202] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. "Deep Ground Truth Analysis of Current Android Malware." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2017.

[203] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. *Trojan-Spy.Mecor.1*. `http://amd.arguslab.org/families/Mecor/variety1.html`. (last visited Mar. 13, 2019). 2017.

[204] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. "Android Permissions Remystified: A Field Study on Contextual Integrity." In: *Proc. of USENIX Security Symposium*. 2015.

[205] Michelle Wong and David Lie. "Tackling runtime-based obfuscation in Android with TIRO." In: *Proc. of USENIX Security Symposium*. 2018.

[206] Christian Wressnegger, Frank Boldewin, and Konrad Rieck. "Deobfuscating Embedded Malware Using Probable-Plaintext Attacks." In: *Proc. of Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2013.

[207] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. "A Close Look on n-Grams in Intrusion Detection: Anomaly Detection vs. Classification." In: *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*. 2013.

[208] Christian Wressnegger, Kevin Freeman, Fabian Yamaguchi, and Konrad Rieck. "Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks." In: *Proc. of ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*. 2017.

[209] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. "DroidMat: Android Malware Detection through Manifest and API Calls Tracing." In: *Proc. of the Asia Joint Conference on Information Security (ASIA JCIS)*. 2012.

[210] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. "Effective Real-Time Android Application Auditing." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[211] Xingyu Xing, Wei Meng, Dan Doozan, Alex C. Snoeren, Nick Feamster, and Wenke Lee. "Take This Personally: Pollution Attacks on Personalized Services." In: *Proc. of USENIX Security Symposium*. 2013.

[212] Weilin Xu, David Evans, and Yanjun Qi. "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2018.

[213] Weilin Xu, Yanjun Qi, and David Evans. "Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2016.

[214] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART." In: *Proc. of USENIX Security Symposium*. 2017.

[215] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. "Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique." In: *IEEE Transactions on Information Forensics and Security (TIFS)* 12.7 (2017).

[216] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2014.

[217] Lok-Kwong Yan and Heng Yin. "DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis." In: *Proc. of USENIX Security Symposium*. 2012.

[218] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. "DolphinAttack: Inaudible Voice Commands." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2017.

[219] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2014.

[220] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. "Towards Automatic Generation of Security-Centric Descriptions for Android Apps." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[221] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2013.

[222] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems." In: *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2013.

[223] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2012.

[224] 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: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2012.

[225] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. "Acoustic Fingerprinting Revisited: Generate Stable Device ID Stealthily with Inaudible Sound." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2014.

[226] Zhe Zhu, Dun Liang, Songhai Zhang, Xiaolei Huang, Baoli Li, and Shimin Hu. "Traffic-Sign Detection and Classification in the Wild." In: *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[227] eMarketer. *Number of smartphone users worldwide from 2014 to 2020 (in billions)*. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`. (last visited Mar. 13, 2019). 2019.

[228] Nedim Šrndic and Pavel Laskov. "Practical Evasion of a Learning-Based Classifier: A Case Study." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2014.