UNIVERSITY OF CALIFORNIA
RIVERSIDE

Ensuring Users' Privacy and Security on Mobile Devices

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ioannis Gasparis

March 2017

Dissertation Committee:

Professor Srikanth V. Krishnamurthy, Chairperson
Professor Zhiyun Qian
Professor Chenqyu Song
Professor Michalis Faloutsos

The Dissertation of Ioannis Gasparis is approved:

_____

_____

_____

_____

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor; without his help I would not have been here. Thank you for your continued support and guidance throughut my life in grad school. You not only made me a more knowledgable researcher, but you also helped me become a better person.

Without of my parents' help and constant support, I would not have been here. Thank you for making me who I am, thank you for pushing me to challenge myself, and thank you for giving me everything that I have ever needed to succeed in my life.

Many thanks to the love of my life; you gave me happiness and a reason to fight for during grad school. Thank you for showing me that there are still people worth fighting for.

Chapter two was published in CoNEXT 2013. Chapter three and four are under submission in MobiSys 2017 and USENIX 2017, respectively.

To my juju, my parents and my brother. Thank you for your constant support and love throughout my life.

MONO AEK

ABSTRACT OF THE DISSERTATION

Ensuring Users' Privacy and Security on Mobile Devices

by

Ioannis Gasparis

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2017
Professor Srikanth V. Krishnamurthy, Chairperson

Mobile devices have become increasingly powerful and popular. The number of mobile users is growing fast and people use their phone nowadays for a plethora of things (e.g., e-commerce, mobile banking, e-mail, social, etc). While powerful mobile devices are more convenient for users, the volume and sensitivity of information they contain are crucial to a user's privacy as well as her security. Sometimes the latter is taken for granted (wrongly).

In this dissertation, we propose three major frameworks to protect users' privacy and security on such mobile and highly capable platforms. The three frameworks are as follows.

- **VideoDroid**, an Android framework that allows users to conduct video calls on open Wi-Fi networks while preserving their privacy and minimizing the performance penalties.
- **Droid M+**, an Android framework that helps developers transform their legacy apps to apps that support the newer revocable permission model that was introduced by Google to help users' privacy.
- **RootExplorer**, a fully automated system that is able to detect malware carrying root exploits by leveraging root exploits from commercial one-click root apps to learn the exploits' behaviors and their expected environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices have become the most powerful and convenient computer device. There are more than 8 billion mobile devices that account for more than 3.7 exabytes of mobile network traffic per month. Users use their phone not only for personal use but also for work. It is common for a user to use her phone for e-commerce purposes, mobile banking, as well as for sharing and saving private moments. A compromise of a mobile device can have devastating effects to a user's privacy and security. Consequently, a mobile user's privacy and security is of paramount importance. Because of the way that mobile devices have been designed, to make them secure and privacy aware, a solution has to exist that spans across different layers of a mobile device.

In this dissertation, we consider three layers of security and privacy in a mobile device: (a) the user layer, where the user has the ability to choose her privacy and security level of her mobile device, (b) the developer layer, where the developer should give to the user the option to have privacy and security aware apps, and (c) the provider layer, where a mobile provider has to offer secure and privacy aware apps to the users. The goal of this dissertation is two-fold: (a) to discuss how current methodologies fail to address privacy and security issues for mobile devices, and (b) present three systems that are designed and implemented specifically to solve such issues on the mentioned layers.

The first system is VideoDroid, an Android framework that allows users to conduct video calls on open Wi-Fi networks while preserving their privacy and minimizing the performance penal-

ties. Video transfers using smartphones are becoming increasingly popular. To prevent the interception of content from eavesdroppers, video flows must be encrypted. However, encryption results in a cost in terms of processing delays and energy consumed on the user's device. We argue that encrypting only certain parts of the flow can create sufficiently high distortion at an eavesdropper preserving content confidentiality as a result. By selective encryption, one can reduce delay and the battery consumption on the mobile device. We develop a mathematical framework that captures the impact of the encryption process on the delay experienced by a flow, and the distortion seen by an eavesdropper. This provides a quick and efficient way of determining the right parts of a video flow that must be encrypted to preserve confidentiality, while limiting performance penalties. In practice, it can aid a user in choosing the right level of encryption. We validate our model via extensive experiments with different encryption policies using Android smartphones. We observe that by selectively encrypting parts of a video flow one can preserve the confidentiality while reducing delay by as much as 75% and the energy consumption by as much as 92%.

The second system is Droid M+, a system that helps developers easily retrofit their legacy code to support the new permission model and adhere to Google's guidelines. In Android 6.0 (Marshmallow), Google revamped its long criticized permission model with a new one that prompts the user during runtime, and allows users to dynamically revoke granted permissions. Towards steering developers to this new model and improving user experience, Google also provides guidelines on (a) how permission requests should be formulated (b) how to educate users on why a permission is needed and (c) how to provide feedback when a permission is denied. In this system we perform, to the best of our knowledge, the first measurement study on the adoption of this new revocable permission model on recently updated apps from the official Google Play Store. We find that, unfortunately, (1) most apps have not been migrated to this new model and (2) for those that do support the model, many do not adhere to Google's guidelines. We attribute this unsatisfying status quo to the lack of automated transformation tools that can help developers refactor their code. We believe that Droid M+ offers a significant step in preserving user privacy and improving user experience.

The third and last system is RootExplorer, a fully automated system that is able to detect malware carrying root exploits by leveraging root exploits from commercial one-click root apps to

learn the exploits' behaviors and their expected environment. Malware that are capable of rooting Android phones are arguably, the most dangerous ones. Unfortunately, detecting the presence of root exploits in malware is a very challenging problem. This is because such malware typically target specific Android devices and/or OS versions and simply abort upon detecting that an expected runtime environment (e.g., specific vulnerable device driver or preconditions) is not present; thus, emulators such as Google Bouncer fail in triggering and revealing such root exploits. In this work, we build a system, RootExplorer, to tackle this problem. The key observation that drives the design of RootExplorer is that, in addition to malware, there are legitimate commercial grade Android apps backed by large companies that facilitate the rooting of phones, referred to as root providers or one-click root apps. By conducting extensive analysis on one-click root apps, RootExplorer learns the precise preconditions and environmental requirements of root exploits. It then uses this information to construct proper analysis environments either in an emulator or on a smartphone testbed to effectively detect embedded root exploits in malware. Our extensive experimental evaluations with RootExplorer show that it is able to detect all malware samples known to perform root exploits and incurs no false positives. We have also found an app that is currently available on the markets, that has an embedded root exploit.

Chapter 2

VideoDroid: Resource Thrifty Secure Mobile Video Transfers on Open WiFi Networks

2.1 Introduction

There has been a recent explosion in the number of video transfers from smartphones. In [9], it is reported that there was a fourteen-fold increase in the number of mobile video transfers between 2010 and 2011. Apps such as Facetime are becoming increasingly popular. Preserving the privacy or confidentiality of video transfers from eavesdroppers requires some form of encryption. Encryption however, comes at the cost of increased delays due to processing and battery consumption. Our thesis is that, one does not have to encrypt the entire video stream to be transferred to prevent an eavesdropper from accessing the content. If one were to encrypt only certain appropriately chosen parts of a video stream, there would be a sufficiently high distortion at an eavesdropper, that would protect the confidentiality of the content.

In this work we seek to answer the following questions: (a) What parts of a video stream (or flow) should be encrypted in order to ensure the confidentiality of any eavesdropped content?

and (b) What are the performance benefits (in terms of delay and battery savings) that one can reap, from only encrypting part of the video stream? Towards answering the above questions we develop a mathematical framework that quantifies the effect of the encryption process on the experienced video transfer delay and the expected distortion at an eavesdropper's site. Our framework takes into account both the network (wireless related effects) and video content (slow versus fast motion video) characteristics. We validate the analytical framework via extensive experimentation using Android smartphones. While our analytical framework does not at this time quantify the energy savings from reducing the encryption costs, we quantify the energy savings via experiments.

In more detail, our approach hinges on the insight that different packets in a video flow, carry varied significance with respect to the decoding process. For example, *I*-frames are critical for decoding, and encrypting only these frames (and sending the *P*-frames in the open) could potentially cause the video flow to be significantly distorted and thus, useless at an eavesdropper's site. Encrypting only parts of a video flow can drastically decrease the video transfer delay, as well as provide significant energy savings as discussed above. We consider the possibility of encrypting different types of packets towards determining the strategy that is most effective; each such strategy, wherein we consider encrypting a specific sub-set of packets from a video flow, is referred to as either "the encryption policy", or "the mode of encryption".

Applying our analytical framework in practice: We envision our framework to be used as follows (see Fig. 2.1). The user captures video with her mobile device and initiates the streaming or transfer process. The UI prompts her with the choices available with respect to privacy. The set of choices could include the two extreme cases where all packets are either encrypted or transmitted in the open. A third choice would allow the user to minimize performance penalties (in terms of processing delay and energy consumption) while largely preserving confidentiality. If this option is chosen, the analytical framework is used to determine the appropriate encryption policy. The model is first calibrated with a few sample measurements to estimate scenario parameters; a tool such as AForge [1] can be used to estimate the motion level in the video, while the device capabilities and network conditions are estimated to determine the penalties with each partial encryption choice.

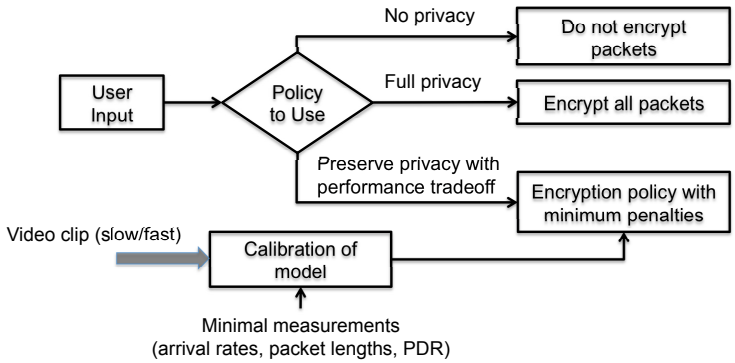


Figure 2.1: Applicability of our framework.

The correct level of encryption is then applied while transferring the video.

In summary, our contributions in this work are:

- We develop an analytical framework that captures the impact of an encryption policy (which packets are encrypted) on key performance metrics of a secure video transfer, viz., the delay due to the encryption process and the distortion at an eavesdropper's site. The framework provides a quick and effective way of quantifying these performance metrics for any given generic encryption process, wireless channel parameters, and the type of video content.
- We implement various encryption policies and demonstrate the effectiveness of only encrypting a part of the video flow via extensive experiments. The experiments also validate our analytical framework. Specifically, by using the Android Native Development Kit (NDK) we implement a mechanism that allows us to deploy different encryption policies on Android smartphones. We evaluate the performance in several scenarios that span differently encoded video streams (various GOP sizes) with different characteristics (slow-motion, fast-motion video streams) and with different encryption policies.

Key results: The key observations from our evaluations are the following:

- By only encrypting parts of a video flow, confidentiality can be preserved while achieving both battery savings and reduced transfer latency. In many cases, these performance benefits are significant; the transfer latency reductions were in some cases 75% and the energy consumed was reduced by 92% in the best case.

- The right strategy in terms of what parts of a video flow to encrypt, depends on the content itself. We find that the encryption of *I*-frames distorts slow motion video more than fast motion video; the encryption of just the *P*-frames distorts fast motion video more than the slow motion video. This is because, rapid changes between scenes in fast-motion videos cause the *P*-frames to carry significant information regarding the content, whereas with slow motion video, these frames do not carry much information.
- Due to the same reasoning as above, with slow-motion video the encryption of the *I*-frames sufficiently protects the content from an eavesdropper; significant savings in cost in terms of the delay and power consumption are possible with this strategy. With fast-motion video, 20% of the *P*-frames need to be encrypted in addition to the encryption of the *I*-frames, to ensure confidentiality. As a consequence, the savings in cost are less significant.

2.2 Background and Related Work

Video Standards and Terminology: Standards such as MPEG-4 [103] and H.264/AVC [186] specify the encoding and transmission of video flows over a network. Typically, the initial video stream has a repetitive structure called Group of Pictures (GOP), which contains an initial *I*-frame followed by *P* and *B* frames (*B*-frames are optional). The size of the *I*-frame varies from GOP to GOP, while the sizes of the *P* and *B* frames differ within and across GOPs. Depending on the Maximum Transmission Unit (MTU) of the network, each frame is segmented into a number of packets that are transmitted over the network[1]. With predictive source encoding, the *I*-frame can be decoded independently of any other information within the GOP and each of the *P* and *B* frames use the *I*-frame as a reference [41]. In this work, we assume an $IPP \ldots P$ encoding structure for each GOP. We refer to the distance between consecutive *I*-frames as the GOP size.

Encryption in Commercial Video Delivery: Various solutions have been proposed for commercial video delivery (not mobile or wireless specific). The HTTP Dynamic Streaming (HDS), developed by Adobe, allows different levels of encryption for the content (low, medium, high).

[1] Thus, one can assume that the number of packets in a GOP is a random variable.

While at a high level it appears that with lower settings only a subset of the frames are encrypted to provide performance improvements during decryption at a receiving client, no details are readily available. In our work, we consider the appropriate frames for encryption during user transfers, with the primary purpose of protecting the content against an eavesdropper.

The HTTP Live Streaming (HLS), implemented by Apple as part of the QuickTime software, also specifies an encryption mechanism that uses AES and a method of secure key distribution based on HTTPS. Finally, the Dynamic Adaptive Streaming over HTTP (DASH) is an MPEG standard (ISO/IEC 23009-1) which enables media content delivery over HTTP. All these platforms operate over HTTP; however each of these uses different segment formats and therefore, to receive the content from each server, a device must support its corresponding proprietary client protocol.

Capturing the impact of wireless links on video: There has been some work on analyzing video communications over wireless links. In [104], the authors model the effect of wireless channel fading on video distortion. Video streaming over a multi-hop IEEE 802.11 wireless network is studied in [119, 144]. These efforts however, do not consider the impact of encryption on video distortion.

In [126] the selective encryption paradigm is discussed and various consumer applications are presented where compression and encryption occur together. However, no analysis is given regarding the behavior of key performance attributes in any case. In [180], the authors focus on wireless sensor networks and propose a scheme to selectively encrypt data based on the channel condition in order to improve video quality. However, in their study they do not take into account the characteristics of the video (slow vs fast motion) or the delay incurred due to the encryption process.

Impact of Encryption on Battery: There are studies on the energy consumption due to encryption on wireless devices. In [61], a comparison of the energy consumption due to common encryption algorithms (AES, DES, 3DES, RC2, Blowfish and RC6) on wireless devices is presented. In [147], an analysis of energy consumption of RC4 and AES algorithms in wireless LANs is provided. However, these works do not consider selective encryption of video as we do here.

Other Related Work: Partial encryption of content in photos is considered in [150]. An

encoding algorithm that extracts and encrypts a small but significant component of a photo, is proposed. In contrast, we consider the structure of a video stream towards preserving its confidentiality.

2.3 Threat Model and Assumptions

Threat model: In this work, we focus on a specific *threat* to privacy arising when transferring video flows or streams from a wireless device over a WiFi connection. Specifically, we consider the capture of content by an eavesdropper who is on the same open WiFi network. This situation arises typically in public places where WiFi is offered for free, e.g., cafes, malls, libraries, airport terminals (e.g., see [5, 8]).

Our *goal* is to secure wireless video transfers from eavesdroppers in a resource efficient way, with respect to the delay and the energy consumption on the wireless device. We expect the user to select the level of protection based on the sensitivity of the content. We *define* a selection policy $\mathscr{P}$ to be (i) the encryption algorithm that is used for protecting the transmitted packets, and (ii) the set of packets to be encrypted. In the extreme case where the content is highly sensitive and no information is to be leaked, all packets are to be encrypted. If the information is not sensitive, no packets are encrypted and this eliminates the performance penalties due to the encryption process. A user may simply seek to distort the video flow at an eavesdropper's site, thus risking the leakage of some information, but preserving the confidentiality to a large extent. Thus, we seek to provide the user with a control mechanism over the protection level for his content by defining various encryption policies. Each such policy results in the encryption of a specific sub-set of packets, (*I*-frame packets, *P*-frame packets, mixture of both) based on the required protection level and the associated performance cost. Depending on the level of distortion induced, an eavesdropper may be able to glean some information from the flow, but not all information; for example, he may determine that the flow is that of a football game, but may be unable to identify the players in the stream[2].

We focus on symmetric key encryption and *assume* that the mutual authentication and the

[2]We reiterate that a user may choose to encrypt all packets in an extreme case.

agreement on the symmetric encryption method has been completed a priori (before the video is to be transferred). We also *assume* that the user has a valid key that has been established either using Public Key Infrastructure (PKI) or the standard Diffie-Hellman (DH) key exchange algorithm. For each encryption policy $\mathscr{P}$, the sender selects the appropriate set of packets to be encrypted. For each of these packets, the payload is encrypted using the symmetric key algorithm defined by $\mathscr{P}$ and the 'a priori established' secret key and transmitted on the wireless uplink. We emphasize that establishment of keys or authentication are not the focus of our work.

We *assume* that an unauthorized eavesdropper using the same WiFi network, is able to overhear transmissions but cannot decrypt packets, encrypted by the sender under $\mathscr{P}$. This affects the video distortion at the eavesdropper, since the encrypted packets cannot be used towards reconstructing the video during the video decoding phase.

We do not consider a traffic analysis attack by the eavesdropper. Specifically, we make no attempt to encrypt the headers that contain information such as IP addresses; while this can be easily accomplished this is not the goal of our work. The eavesdropper may be able to distinguish packets as belonging to either *I*-frames or *P*-frames based on their size or other characteristics. While the sender can obfuscate these features by using techniques such as padding the payload, we do not consider these possibilities in this work.

Other assumptions: Our key idea and approach is agnostic of whether the video flow is transferred using HTTP or RTP, and the transport layer in use. It can be applied with real-time streaming (e.g., Facetime) or for transfers to a server. Facetime uses RTP (or Secure RTP) over either UDP or TCP [11]. Other applications for video transfers (e.g., Google hangouts) also use UDP [6]. Commercial content delivery systems operate over HTTP and therefore, TCP. For tractability (so that we do not have to model TCP behaviors) we assume the use of RTP and UDP in our analysis. However, we experimentally demonstrate in Section 2.6 that our key ideas hold with HTTP/TCP.

We also defer the problem of jointly encoding and encrypting video. Furthermore, we expect that the volume of audio content is going to be much lower than video and thus, all of it can be encrypted. However, we do not consider this here and will explore these issues in future work.

2.4 Our Analytical Framework

In this section, we present our mathematical framework to characterize the effect of an encryption policy $\mathscr{P}$ on (i) the increase in delay due to encryption and (ii) the distortion at an eavesdropper due to a chosen encryption policy. An encryption policy $\mathscr{P}$ defines (i) the symmetric key algorithm that is used for encrypting packets at the sender and (ii) the packets to be selected for encryption.

2.4.1 Packet Success Rate

A key parameter to our delay and distortion analysis is the packet success rate of the network. As discussed above, we consider a WiFi network based on the IEEE 802.11 standard. There are various models (e.g., [34, 74]) that attempt to capture the operations of the IEEE 802.11 protocol. We use the model in [34] to represent the operations of the PHY and MAC layers. The model consists of three sets of equations (representing scheduling, channel access and routing) which are solved through a fixed point method. The solution is an approximation to the packet success rate p_s under the assumption that the traffic at the source nodes are persistent.

2.4.2 Delay

Video transfer delay is important especially for streaming applications. To compute the delay for each packet at the sender, we characterize the process as packets traversing a 2-MMPP/G/1 queue. The arrivals to the queue correspond to reading video file segments from the disk and storing them to a buffer in the memory. Based on the encryption policy $\mathscr{P}$, the server decides whether to encrypt the packet at the head of the queue or not, and then transmits the packet over the channel. This implies that the service time consists of a possible delay due to the encryption process, a backoff time due to possible collisions at the shared medium and the transmission time.

Arrival Process

The arrival process models the time instances where video files segments are read from the local disk and enqueued in a buffer for transmission. A segment that corresponds to an I-frame is typically larger than the MTU of the network and is thus fragmented into several packets with lengths equal to the MTU. On the other hand, a P-frame typically corresponds to a single packet that is much smaller than the MTU of the network. In the first case, the interarrival times of the packets that belong to an I-frame are much smaller compared to those that are associated with the arrival of P-frame packets. Therefore, there is a need to capture the two different phases of the arrival process. A natural choice is the Markov modulated Poisson process (MMPP), which is a doubly stochastic Poisson process where the rates are determined by the state of a continuous-time Markov chain [101].

We use a two-state Markov chain where the rate of transition from state 1 to state 2 is ρ_1 and from state 2 to state 1 is ρ_2. When the chain is in state 1 the arrival rate is λ_1 and the process models the arrival of I-frame packets (small interarrival times). When the chain is in state 2 the arrival rate is λ_2 and the process models the arrival of P-frame packets (larger interarrival times). The MMPP is then parameterized by the infinitesimal generator R associated with the Markov chain and the rate matrix Λ:

$$R = \begin{bmatrix} -\rho_1 & \rho_1 \\ \rho_2 & -\rho_2 \end{bmatrix}, \quad \Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}. \tag{2.1}$$

The equilibrium probability vector π (representing the probabilities of being in state j, $j \in \{1,2\}$) is given by:

$$\pi = \frac{1}{\rho_1 + \rho_2} (\rho_2, \rho_1). \tag{2.2}$$

Service Times

The total service time T of a packet is the time from the moment the packet reaches the server until it is successfully transmitted. This time includes the encryption time $T_e^{(\mathscr{P})}$ in case the packet is scheduled for encryption based on the encryption policy $\mathscr{P}$, the backoff time T_b that the

packet may need to wait at the transmitter due to collisions and the transmission time T_t,

$$T = T_e^{(\mathscr{P})} + T_b + T_t. \tag{2.3}$$

The encryption time $T_e^{(\mathscr{P})}$ of a packet depends on the packet size and the encryption policy $\mathscr{P}$. In a typical GOP structure the initial I-frame is larger than the following P-frames (e.g., an I-frame can be 100 times larger than a P-frame) [41]. Therefore, as discussed earlier, an I-frame is typically fragmented into a sequence of packets that have lengths equal to the MTU of the network, while a P-frame corresponds to a single packet of a much smaller length. Moreover, the use of different encryption algorithms result in different delays and in general, affects the encryption time.

If $q^{(\mathscr{P})}$ is the probability that a packet is selected for encryption under an encryption policy $\mathscr{P}$ and p_I is the probability that a packet belongs to an I-frame, then the distribution $F_e^{(\mathscr{P})}(\cdot)$ of the encryption time $T_e^{(\mathscr{P})}$ is a mixture distribution derived from the distributions $F_{e,I}^{(\mathscr{P})}(\cdot)$ and $F_{e,P}^{(\mathscr{P})}(\cdot)$ of the encryption time $T_{e,I}^{(\mathscr{P})}$ of a packet that belongs to an I-frame and the encryption time $T_{e,P}^{(\mathscr{P})}$ of a packet that belongs to a P-frame, respectively:

$$\begin{aligned}
F_e^{(\mathscr{P})}(\tau) &= P\{T_e^{(\mathscr{P})} < \tau\} \\
&= P\{T_e^{(\mathscr{P})} < \tau, \text{ } I\text{-frame pkt, encrypted}\} \\
&\quad + P\{T_e^{(\mathscr{P})} < \tau, \text{ } P\text{-frame pkt, encrypted}\} \\
&= q^{(\mathscr{P})} p_I P\{T_{e,I}^{(\mathscr{P})} < \tau\} + q^{(\mathscr{P})} (1-p_I) P\{T_{e,P}^{(\mathscr{P})} < \tau\} \\
&= q^{(\mathscr{P})} p_I F_{e,I}^{(\mathscr{P})}(\tau) + q^{(\mathscr{P})} (1-p_I) F_{e,P}^{(\mathscr{P})}(\tau).
\end{aligned} \tag{2.4}$$

We compute the Laplace-Stieltjes transform $H_e^{(\mathscr{P})}(\cdot)$ of $T_e^{(\mathscr{P})}$ by using (2.4) and the sta-

tistical independence of $T_{e,I}^{(\mathscr{P})}$ and $T_{e,P}^{(\mathscr{P})}$:

$$\begin{aligned} H_e^{(\mathscr{P})}(s) &= \int_0^{+\infty} e^{-s\tau}\, dF_e^{(\mathscr{P})}(\tau) \\ &= q^{(\mathscr{P})} p_I \int_0^{+\infty} e^{-s\tau} dF_{e,I}^{(\mathscr{P})}(\tau) \\ &\quad + q^{(\mathscr{P})}(1-p_I) \int_0^{+\infty} e^{-s\tau} dF_{e,P}^{(\mathscr{P})}(\tau). \end{aligned} \tag{2.5}$$

The time T_b corresponds to the time the packet has to wait at the transmitter due to collisions at the MAC layer. The packet is successfully transmitted without any collisions with probability p_s and when this happens $T_b = 0$. Otherwise, the backoff time can be approximated by the sum $\sum_{j=1}^{K} \tau_j$ of independent, exponentially distributed random variables $\{\tau_j,\ j = 1,2\ldots,K\}$ with mean $1/\lambda_b$, each corresponding to a waiting interval after a collision. The number K of collisions experienced by a packet, and therefore, the number of the waiting times τ_j, is distributed according to

$$P\{K = k\} = (1-p_s)^k\, p_s, \quad k = 0,1,2,\ldots. \tag{2.6}$$

The Laplace-Stieltjes transform $H_b(\cdot)$ of T_b can be computed to be:

$$H_b(s) = p_s \frac{\lambda_b + s}{\lambda_b\, p_s + s}, \quad s < \lambda_b. \tag{2.7}$$

The transmission time T_t of a packet depends on the packet size. The distribution $F_t(\cdot)$ of the transmission time T_t is a mixture distribution derived from the distributions $F_{t,I}(\cdot)$ and $F_{t,P}(\cdot)$ of the transmission time $T_{t,I}$ of a packet that belongs to an I-frame and the transmission time $T_{t,P}$ of a packet that belongs to a P-frame, respectively. If p_I is the probability that a packet belongs to an

I-frame, then

$$\begin{aligned} F_t(\tau) &= P\{T_t < \tau\} \\ &= P\{T_t < \tau, I\text{-frame pck}\} + P\{T_t < \tau, P\text{-frame pck}\} \\ &= p_I P\{T_{t,I} < \tau\} + (1-p_I) P\{T_{t,P} < \tau\} \\ &= p_I F_{t,I}(\tau) + (1-p_I) F_{t,P}(\tau). \end{aligned} \tag{2.8}$$

We compute the Laplace-Stieltjes transform $H_t(\cdot)$ of T_t by using (2.8) and the statistical independence of $T_{t,I}$ and $T_{t,P}$:

$$\begin{aligned} H_t(s) &= \int_0^{+\infty} e^{-s\tau} dF_t(\tau) \\ &= p_I \int_0^{+\infty} e^{-s\tau} dF_{t,I}(\tau) + (1-p_I) \int_0^{+\infty} e^{-s\tau} dF_{t,P}(\tau). \end{aligned} \tag{2.9}$$

Assuming the random variables $T_e^{(\mathscr{P})}$, T_b and T_t are mutually independent, the Laplace-Stieltjes transform $H(\cdot)$ of the service time T can be computed from (2.5), (2.7) and (2.9) to be:

$$H(s) = H_e^{(\mathscr{P})}(s) H_b(s) H_t(s), \quad s < \lambda_b. \tag{2.10}$$

Special Cases:

Constant encryption and transmission times: If the encryption times $T_{e,I}^{(\mathscr{P})}$ and $T_{e,P}^{(\mathscr{P})}$ for the packets that belong to an I and a P-frame, respectively, are constant such that:

$$T_{e,I}^{(\mathscr{P})} = \mu_{e,I}^{(\mathscr{P})}, \quad T_{e,P}^{(\mathscr{P})} = \mu_{e,P}^{(\mathscr{P})}, \tag{2.11}$$

then (2.5) becomes:

$$H_e^{(\mathscr{P})}(s) = q^{(\mathscr{P})} p_I e^{-s\mu_{e,I}^{(\mathscr{P})}} + q^{(\mathscr{P})}(1-p_I) e^{-s\mu_{e,P}^{(\mathscr{P})}}. \tag{2.12}$$

Similarly, if the transmission times $T_{t,I}$ and $T_{t,P}$ of the packets that belongs to an I-frame and a

P-frame, respectively, are constant such that:

$$T_{t,I} = \mu_{t,I}, \quad T_{t,P} = \mu_{t,P}, \tag{2.13}$$

then (2.9) becomes:

$$H_t(s) = p_I e^{-s\mu_{t,I}} + (1-p_i)e^{-s\mu_{t,P}}. \tag{2.14}$$

Accounting for minor variations: If we want to account for minor variations of the encryption and transmission times (seen to occur due to minor variations in packet size in our practical experiments described in Section 2.6) about some typical values, we can represent these variations by independent Gaussian random variables, such that:

$$T_{e,I}^{(\mathscr{P})} = \mu_{e,I}^{(\mathscr{P})} + r_{e,I}^{(\mathscr{P})}, \qquad T_{e,P}^{(\mathscr{P})} = \mu_{e,P}^{(\mathscr{P})} + r_{e,P}^{(\mathscr{P})}, \tag{2.15}$$

where $\mu_{e,I}^{(\mathscr{P})}$ is constant and equal to the time needed to encrypt a packet of size equal to the MTU of the network under the encryption policy $\mathscr{P}$ and $\mu_{e,P}^{(\mathscr{P})}$ corresponds to a typical encryption time for a packet that belongs to a P-frame. The quantity $r_{e,I}^{(\mathscr{P})}$ is a normal random variable with zero mean and variance $(\sigma_{e,I}^{(\mathscr{P})})^2$ that represents small variations in the encryption time of a packet that belongs to an I-frame and is selected for encryption. Similarly, $r_{e,P}^{(\mathscr{P})}$ is a normal random variable with zero mean and variance $(\sigma_{e,P}^{(\mathscr{P})})^2$ that represents variations in the encryption time from packet to packet for the P-frames. Clearly, $T_{e,I}^{(\mathscr{P})} \sim \mathscr{N}(\mu_{e,I}^{(\mathscr{P})}, (\sigma_{e,I}^{(\mathscr{P})})^2)$ and $T_{e,P}^{(\mathscr{P})} \sim \mathscr{N}(\mu_{e,P}^{(\mathscr{P})}, (\sigma_{e,P}^{(\mathscr{P})})^2)$.

Representing the transmission times of packets that belong to I and P-frames in a similar way, we have:

$$T_{t,I} = \mu_{t,I} + r_{t,I}, \qquad T_{t,P} = \mu_{t,P} + r_{t,P}, \tag{2.16}$$

where $\mu_{t,I}$ is constant and equal to the time needed to transmit a packet of length equal to the MTU of the network and where $\mu_{t,P}$ corresponds to a typical transmission time for a P-frame packet. The quantity $r_{t,I}$ represents minor random variations in the transmission time of an I-frame packet, modeled as a normal random variable with zero mean and variance $\sigma_{t,I}^2$ and $r_{t,P}$ is a normal random

variable with zero mean and variance $\sigma^2_{t,P}$ that represents minor variations in the transmission time of a P-frame packet. Clearly, $T_{t,I} \sim \mathcal{N}(\mu_{t,I}, \sigma^2_{t,I})$ and $T_{t,P} \sim \mathcal{N}(\mu_{t,P}, \sigma^2_{t,P})$.

Using the representations in (2.15) and (2.16), the Laplace-Stieltjes transforms $H_e^{(\mathscr{P})}(\cdot)$ and $H_t(\cdot)$ of the encryption and transmission times $T_e^{(\mathscr{P})}$ and T_t, respectively, become:

$$\begin{aligned} H_e^{(\mathscr{P})}(s) &= q^{(\mathscr{P})}\, p_I\, e^{-\mu_{e,I}^{(\mathscr{P})} s + \frac{1}{2}(\sigma_{e,I}^{(\mathscr{P})})^2 s^2} \\ &\quad + q^{(\mathscr{P})}\,(1-p_I)\, e^{-\mu_{e,P}^{(\mathscr{P})} s + \frac{1}{2}(\sigma_{e,P}^{(\mathscr{P})})^2 s^2}, \end{aligned} \tag{2.17}$$

$$H_t(s) = p_I\, e^{-\mu_{t,I} s + \frac{1}{2}\sigma_{t,I}^2 s^2} + (1-p_I)\, e^{-\mu_{t,P} s + \frac{1}{2}\sigma_{t,P}^2 s^2}, \tag{2.18}$$

where we used the fact that the Laplace-Stieltjes transform of a normal distribution with mean μ and variance σ^2 is $e^{-\mu s + \frac{1}{2}\sigma^2 s^2}$. Note that we use this second model in our evaluations described in Section 2.6.

2-MMPP/G/1 Queue Model

The delay experienced by each packet at the sender can be estimated by the 2-MMPP/G/1 queueing model described above. An algorithmic approach that solves the n-MMPP/G/1 queue model is given in [95] and refined in [68] for the case $n = 2$. The algorithm describes a numerical procedure that is shown to converge to the solution of the model. It is based on a general method introduced in [135] and applied in [152] to provide a detailed statistical analysis of the N/G/1 queue.

The method takes as input the infinitesimal generator R and the rate matrix Λ of the MMPP and the Laplace-Stieltjes transform of the service time given by (2.10). The algorithm computes the distribution function and the moments of the delay seen by the video packets. In particular, the expected value of the queueing delay W is given by

$$\begin{aligned} \mathrm{E}[W] = \frac{1}{2(1-\rho)} \Big[& 2\rho + \mu^{(2)}\pi\lambda \\ & - 2\mu^{(1)}(y + \mu^{(1)}\pi\Lambda)(R + e\pi)^{-1}\lambda \Big], \end{aligned} \tag{2.19}$$

where $\rho = \pi\lambda\mu^{(1)}$ is the traffic intensity, $\mu^{(1)}, \mu^{(2)}$ are the first and second moments about the origin

respectively, of the service time that can be computed directly from (2.10), λ is the vector with the diagonal elements of Λ, $e = (1,1)^T$ and the vector y is computed by the algorithm.

2.4.3 Distortion

There are two parameters that control the video distortion: (i) the packet *decryption rate* p_d and (ii) the decoder *sensitivity* s. The packet decryption rate represents the probability that a packet is received without errors at a node and that the node is able to correctly decrypt the packet. A legitimate receiver has all the necessary information to correctly decrypt packets from the sender; on the other hand an eavesdropper lacks this capability. Therefore, an eavesdropper can only use packets that the sender has decided not to encrypt towards reconstructing the video, when the latter follows a specific encryption policy $\mathscr{P}$. If we denote by $q^{(\mathscr{P})}$ the percentage of packets encrypted by the sender under an encryption policy, then the decryption rates p_d^l and p_d^e of a legitimate receiver and an eavesdropper, respectively, are: $p_d^l = p_s$ and $p_d^e = (1 - q^{(\mathscr{P})})\ p_s$, where p_s is the packet success rate (recall Section 2.4.1).

The parameter s represents the sensitivity of the decoder to packets that are missing in the receiving stream (either due to interference-induced losses, or in the case of the eavesdropper due to the lack of decryption capabilities). It is the minimum number of packets that the decoder needs to receive (and decrypt) without errors in order to decode the corresponding frame correctly. The sensitivity is associated with the video content itself and specifically with the motion level. When a video flow is characterized by high (or fast) motion, the sensitivity s has a higher value compared to a low (or slow) motion video. This is because in a high motion video flow, the difference between successive frames in the GOP structure is large and the loss of a frame has a higher impact on the overall video quality.

Video Frame Success Rate

We map the packet decryption rate p_d to the video frame (referred to as simply 'frame') success rate P_f, which denotes the probability a frame is successfully received over the wireless link. As was mentioned in Section 2.2, we assume that each GOP has an $IPP\ldots P$-structure.

If n is the number of packets in each frame, then to successfully decode a frame, (a) the first packet of that frame needs to be received without channel-induced errors and successfully decrypted, and (b) the same should hold true for $0 \leq s \leq n-1$ of the remaining $n-1$ packets. The success probability of a frame is given by:

$$P_f = p_d \sum_{i=s}^{n-1} \binom{n-1}{i} p_d^i (1-p_d)^{n-1-i}. \tag{2.20}$$

In general, the *I*-frame is much larger than a *P*-frame. As a result, the frame success probabilities for an *I* and a *P*-frame also differ. We denote by P_I, the success probability of an *I*-frame and by P_P, the success probability of a *P*-frame. We have validated the above model via extensive experiments using the EvalVid tool [3].

Mean Square Error

Let the GOP structure contain $G-1$ *P*-frames that follow the *I*-frame. We consider predictive source coding where, if the i^{th} frame is the first lost frame in a GOP, then the i^{th} frame and all its successors in the GOP are replaced by the $(i-1)^{st}$ frame at the decoder. If the *I*-frame of the GOP cannot be decoded correctly, then the entire GOP is considered unrecoverable and is ignored. In this case, these lost video frames are replaced by the most recent frame from a previous GOP that is correctly received. In all cases, the similarity between the missing frames and the reference frame (substitute frame) affects the distortion [181].

We compute the video distortion as the *mean square error* of the difference between the missing frame and the substitute frame. We have the following cases:

Case 1 – Intra-GOP distortion: The *I*-frame of the current GOP is successfully received. The distortion for the current GOP depends on which, if any, of the *P*-frames of the GOP cannot be decoded without errors. If the first unrecoverable *P*-frame is the i^{th} frame in the GOP, the corresponding distortion is given by [104]:

$$d_i = (G-i)\frac{i \cdot G \cdot d_{min} + (G-i-1) \cdot d_{max}}{(G-1) \cdot G}, \tag{2.21}$$

for $i = 1, 2, \ldots, (G-1)$, where $d_{\max}$ is the maximum distortion when the first frame is lost and d_{min} is the minimum distortion when the last frame is lost. The values of $d_{\max}$ and $d_{\min}$ can be estimated given the probability of a packet loss. The probability P_i that the i^{th} frame is lost is

$$P_i = P_I P_P^{i-1}(1-P_P), \quad i = 1, 2, \ldots, (G-1). \tag{2.22}$$

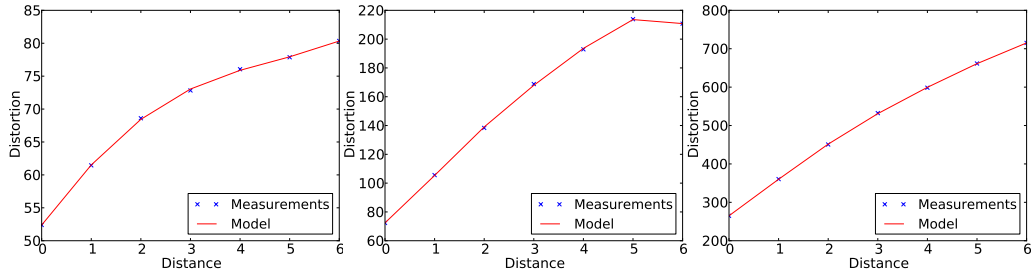
Using (2.21) and (2.22), the expected value of the distortion can be computed to be: $D^{(1)} = \sum_{i=1}^{G} d_i \cdot P_i$

Case 2 – Inter-GOP distortion: The *I*-frame of the current GOP is lost and a frame from a previous GOP is used as the reference frame. Here, the difference between the reference frame and the missing frames determine the distortion.

Similar to the work in [181], we expect to see the motion characteristics of the video affecting distortion. To capture the dependence of the inter-GOP distortion on the motion level of the video we perform a set of experiments and use the collected results to statistically describe this association.

Specifically, we select a set of video streams from [13] and categorize them into three groups according to their motion level: low, medium and high, using the tool in [1]. All video streams have 300 frames each, with a frame rate of 30 frames per second. We use FFmpeg [4] to convert the video streams from the initial, uncompressed YUV format to the MP4 format. Then, we artificially create video frame losses in order to achieve reference frame substitutions from various distances. Finally, we use the Evalvid toolset [3] to measure the corresponding video distortion.

In Fig. 2.2 the dependence of the average distortion on the distance between the missing frame and the substitute is shown for the three categories. In order to use these empirical results in other experiments, we approximate the observed curves with polynomials of degree 5 using a multinomial regression (use of higher degree polynomials does not increase accuracy). In particular, we define the approximate distortion $D^{(2)}$ as a function of the distance d: $D^{(2)}(d) = \sum_{i=0}^{5} a_i d^i$, and compute the coefficients $a_0, \ldots, a_5$, using the regression.



(a) Low motion (b) Medium motion (c) High motion

Figure 2.2: Average distortion with distance.

Case 3 – Initial GOP: The I-frame of the current and all previous GOPs (including the first GOP) are lost. In this case the distortion D is maximized. If $\{D_{\max}^{(1)}, D_{\max}^{(2)}, \ldots, D_{\max}^{(\|\mathscr{G}\|)}\}$, where $\mathscr{G}$ is the set of all GOPs in the video flow, is the set of the maximum distortion values in all GOPs, then $D^{(3)} = \max_{k \in \mathscr{G}} D_{\max}^{(k)}$.

Computing Average Distortion

Suppose the video flow has N GOPs and each GOP consists of an I-frame followed by $G-1$ P-frames. For each GOP of the flow define the state $S_i, i = 1,2,\ldots,N$, such that $S_i \in \mathscr{S} = \{0,1,\ldots,G\}$. The state S_i for the i^{th} GOP indicates which is the first unrecoverable frame in that GOP. Specifically,

$$S_i = \begin{cases} 0, & I\text{-frame is lost,} \\ k, & k^{th}\ P\text{-frame is lost, } 1 \leq k \leq (G-1), \\ G, & \text{none of the frames is lost,} \end{cases} \tag{2.23}$$

for $i = 1,2,\ldots,N$. The initial state for each GOP is G. The transition probability $p_i(G,g)$ of state S_i from G to $g \in \mathscr{S}$ is

$$p_i(G,g) = \begin{cases} 1-P_I, & g = 0, \\ P_I P_P^{k-1}(1-P_P), & g = k, 1 \leq k \leq (G-1), \\ P_I P_P^{G-1}, & g = G, \end{cases} \tag{2.24}$$

for $i = 1,2,\ldots,N$.

To compute the expected value of the distortion for the transmission of the video stream over the wireless channel we need to consider the states of all the GOPs. We define the vector $S = (S_1, S_2, \ldots, S_N) \in \mathscr{S} \times \mathscr{S} \times \cdots \times \mathscr{S}$. The initial state of S is $G = (G, G, \ldots, G)$ and its transition probability $p(G,g)$ to a new state $g = (g_1, g_2, \ldots, g_N)$ is

$$p(G,g) = \prod_{i=1}^{N} p_i(G, g_i). \tag{2.25}$$

The overall distortion for the video stream transmission depends on the final state g. As was discussed earlier in Case 2, the distortion of a GOP may depend not only on missed frames in that GOP but on frames that are missing in previous GOPs as well. Therefore, if D_i is the distortion of the i^{th} GOP, it is a function of the vector g and not only of the i^{th} component of g. We define the random variable $D(g) = (D_1(g), D_2(g), \ldots, D_N(g))$ consisting of the distortions of each of the GOPs of the video flow. Using (2.25) we have:

$$\mathrm{E}[D] = (\mathrm{E}[D_1], \ldots, \mathrm{E}[D_N]) = \sum_g p(G,g) D(g) \tag{2.26}$$

The average distortion that corresponds to the video file is

$$\bar{D} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{E}[D_i]. \tag{2.27}$$

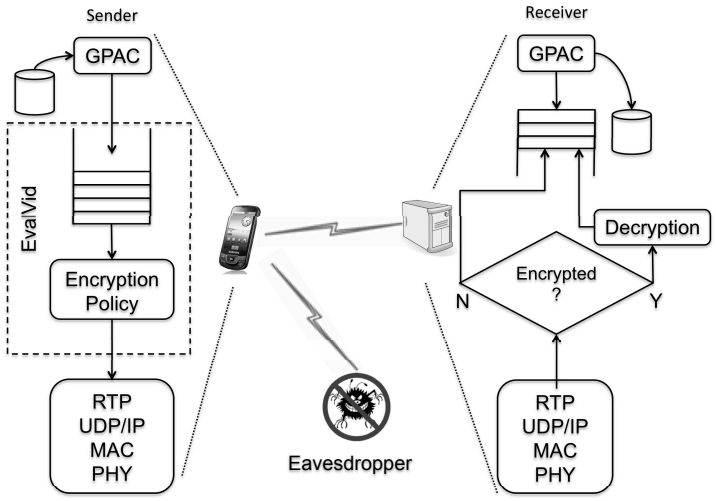


Figure 2.3: Block diagram for sender and receiver.

Mapping Distortion to PSNR

In all the results we present in the sequel, we use the Peak Signal-to-Noise Ratio (PSNR) which is an objective video quality measure [41]. The relationship between distortion and PSNR (in dB) is given by [41]:

$$\text{PSNR} = 20 \cdot \log_{10} \frac{255}{\sqrt{\text{Distortion}}} \tag{2.28}$$

The goal then is to encrypt enough frames to drive the PSNR to be as low as possible at an eavesdropper's site.

2.5 Implementation

We implement a software framework on the Android NDK and SDK that allows us to test various encryption policies. To achieve this we use GPAC [7] and the EvalVid [3] toolset. The former is a cross-platform open source multimedia framework which provides support for creating, parsing and streaming multimedia packaging formats, such as MP4. The latter provides tools for evaluating the quality of video which is transmitted over a network. Our application modifies the EvalVid tool to read and then securely transmit a video stream according to the selected encryption

policy.

With our application, the user selects which video to transmit, the receiver and the encryption policy. The application uses the GPAC library to read the video from the disk into the internal memory. There are two threads that access the memory; the producer thread reads the video segments from the disk and stores them in a queue, and the consumer thread reads the segment from the head of the queue and forwards it to the block of code that implements the encryption policy selected by the user. Our code checks whether the video segment satisfies the encryption selection rule defined by the policy in effect or not. If it does, it uses the GPAC API to encrypt the segment according to the encryption algorithm (AES128, AES256, 3DES) using the *Output Feedback Mode* (OFB). The OFB encryption mode is applied to each segment separately, and therefore a possible error at the receiver does not propagate to the following segments during the decryption process. By default, we assume the use of RTP and UDP; we discuss experiments with HTTP/TCP transfers in Section 2.6.4. Whether the video segment is encrypted or not, it is encapsulated in an RTP packet. In the case that encryption has been performed, the Marker Bit in the RTP header is set denoting the event to the receiver. The RTP packet is transmitted over UDP to the receiver.

Upon the reception of an RTP packet, the receiver checks the Marker Bit in the RTP header to decide if the RTP payload is encrypted. If the Marker Bit is set, the receiver uses the GPAC API to decrypt the packet according to the encryption policy. The received packets are then combined in order to reconstruct the MP4 video file. Fig. 2.3 depicts the operations performed at the sender and receiver.

The eavesdropper (see Fig. 2.3) overhears the transmission on the channel by using `tcpdump` on his rooted phone or laptop. Only the unencrypted packets can be used towards reconstructing the overheard video stream. Because of this, the eavesdropper experiences significantly higher video transmission distortion than that at the legitimate receiver.

Table 2.1: Experimental Setup

Frame Size	CIF (352x288)
GOP Size	30, 50
Video Motion	slow-motion, fast-motion
Encryption Algorithm	AES128, AES256, 3DES
Encryption Level	none, I-frame, P-frame, all
Wireless Devices	Samsung Galaxy S-II, HTC Amaze 4G
Android Version	Ice Cream Sandwich (4.0)

2.6 Evaluation

This section demonstrates the viability of the approach, quantifies the trade-off between the transfer delay, and the distortion at an eavesdropper's site and discusses the impact of the video type (slow vs fast motion) on the mode of encryption needed. Experimental results on the battery savings with different modes of encryption are also presented. Results with HTTP/TCP are presented at the end of the section.

2.6.1 Methodology

We validate our analysis through extensive experiments using smartphones running our Android application over WiFi connections (IEEE 802.11g). Table 2.1 lists the parameters considered.

Wireless devices: All the experiments are repeated on two different smartphones, viz., (i) the Samsung Galaxy S-II that has a 1.2 GHz dual-core ARM Cortex-A9 CPU, an ARM Mali-400 MP4 GPU, and 1 GB RAM and (ii) the HTC Amaze 4G equipped with a 1.5 GHz dual core Qualcomm Snapdragon S3 CPU, Adreno 220 GPU and a 1 GB RAM. Both devices run Android 4.0 (Ice Cream Sandwich).

Strategies tested: Twelve encryption policies are tested; they consist of all possible combinations of three different encryption algorithms and four modes of packet encryption. In particular,

the three symmetric key encryption algorithms that are considered are the AES128, AES256 [174] and 3DES [22]. Due to space constraints we only show the results for AES256 and 3DES. The results for the AES128 encryption algorithm are similar and follow the same trends. The complete set of results is in our technical report [143]. For the packet encryption selection rules, we consider the two extreme cases where either all or none of the packets are encrypted. We also consider the case where only the packets that belong to an *I*-frame are encrypted and the case where only the packets that belong to *P*-frames are encrypted. Finally, we consider encrypting the *I*-frames *and* different fractions of the *P*-frames. We did limited experiments with other possibilities (only partial encryption of *I*-frames) but did not pursue these beyond that since the behavioral results could be extrapolated based on the results that we present here.

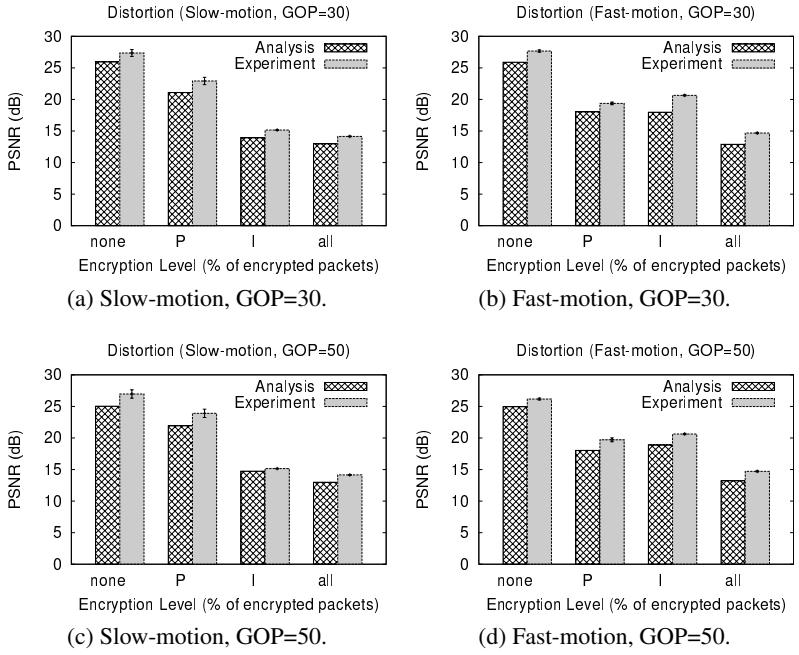
Types of video flows: The experiments are performed on two kinds of video flows: slow-motion and fast-motion video flows. Slow-motion video flows are characterized by slow changes from picture-frame to picture-frame and therefore the size of the *P*-frames in each GOP are typically small (tens to hundreds of bytes). On the other hand, fast-motion video flows contain rapid changes between picture-frames having as a result larger *P*-frames. We use the AForge [1] tool to dynamically categorize the motion level in different parts of the video clip. The motion level of a video flow affects not only the GOP structure (i.e. percentage of *I*-frame and *P*-frame packets in the GOP structure) but also the sensitivity of the video decoder to the packet loss ratio. A fast-motion video flow is more susceptible to packet losses and therefore the distortion at the receiver (or eavesdropper) can be naturally higher compared to the case where a comparable in size, slow-motion video flow is transmitted over the same wireless link. All video flows are of the same picture-frame size (CIF-352x288 pixels) and are encoded using the publicly available `x264` [12] software library and application into different GOP sizes (30, 50 frames).

Experimental methodology: We use the Android application that we have developed to measure the delay due to the encryption and the EvalVid toolset to compute the distortion at the eavesdropper. We also run `tcpdump` on the wireless device to capture the time when each packet is transmitted over the wireless link. EvalVid supports performance metrics such as the Peak Signal to Noise Ratio (PSNR) [41], which we use to represent video quality. Note that the lower the PSNR,

the higher the distortion.

To compute the delay and distortion we follow a sequence of steps: we start with the initial uncompressed video files which consist of a sequence of YUV [102] frames. Using the EvalVid toolset, we transform the YUV format, first to the H.264 format and then to a MP4 video file. Next, we use the Android application we have developed to transmit the video stream to the legitimate receiver. During this phase, we select the set of packets to be encrypted based on the encryption policy that is in effect. We keep track of the time instances at which each packet reaches different parts of our application. These statistics include the time instances when the packet enters and leaves the queue that is shown in Fig. 2.3, the time duration needed to encrypt the packet, in case this packet is selected for encryption, and the time instance when the packet is forwarded to the transport layer. Furthermore, we use `tcpdump` to capture the time instance the packet is transmitted over the wireless link. At the legitimate receiver, all the successfully received packets are used to reconstruct the initial video using the EvalVid toolset. At the eavesdropper, only the successfully received unencrypted packets contribute in the reconstruction of the pilfered video stream. Encrypted packets are treated as erasures. Each experiment is repeated 20 times and the values of the queueing delay and distortion are used to compute the averages and the 95% confidence intervals.

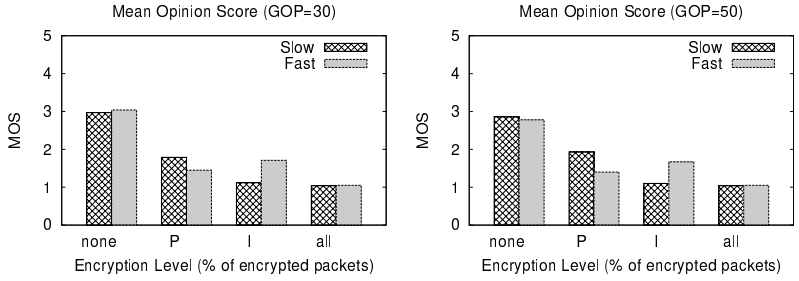
Applying the mathematical framework: We use an initial sequence of events to tune the parameters of our mathematical model. The times of insertion of video segments into the internal queue (see Fig. 2.3) and their type (I, B-frames) are used to estimate the 2-MMPP parameters, R and Λ in (2.1). The sequence of times that are necessary for the encryption of an initial set of packets and the fraction of packets that are encrypted, are used to estimated the mean and variance of the encryption time T_e. Similarly, the observation of the transmission of an initial set of packets can provide estimates for the mean and variance of the transmission time T_t and the parameter λ_b for the backoff time T_b, characterizing this way the service time T in (2.3). Note, the client has access locally to all the necessary information to compute these estimates.



(a) Slow-motion, GOP=30. (b) Fast-motion, GOP=30.

(c) Slow-motion, GOP=50. (d) Fast-motion, GOP=50.

Figure 2.4: Distortion at an eavesdropper's site for slow and fast motion video flows.



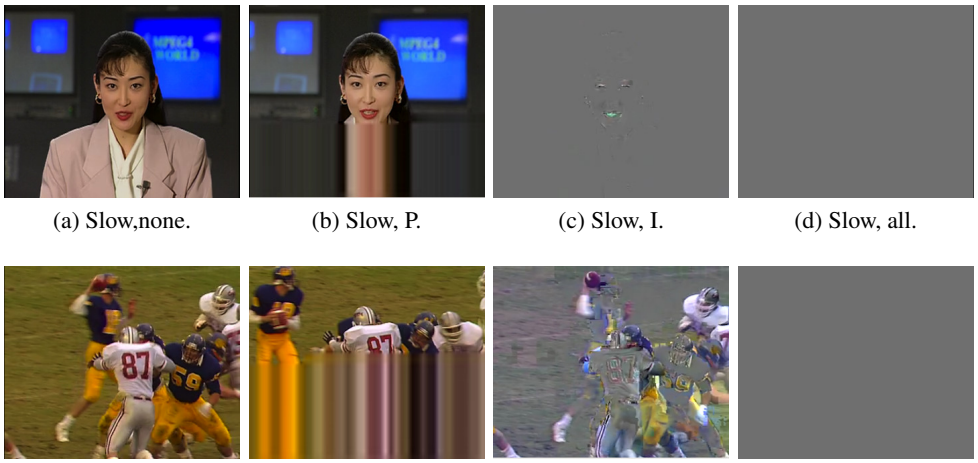
(a) GOP=30. (b) GOP=50.

Figure 2.5: Mean Opinion Score at an eavesdropper's site for slow and fast motion video flows.

2.6.2 Delay vs Distortion

Since the legitimate receiver is capable of decrypting the packets the distortion is only affected by the packet loss ratio on the wireless link. The video distortion at the eavesdropper also depends on the percentage of packets that are encrypted at the sender according to the specific encryption policy that is in use. In order to compute the distortion at each end we use the EvalVid tools to reconstruct the YUV file based on the successfully received and decrypted packets.

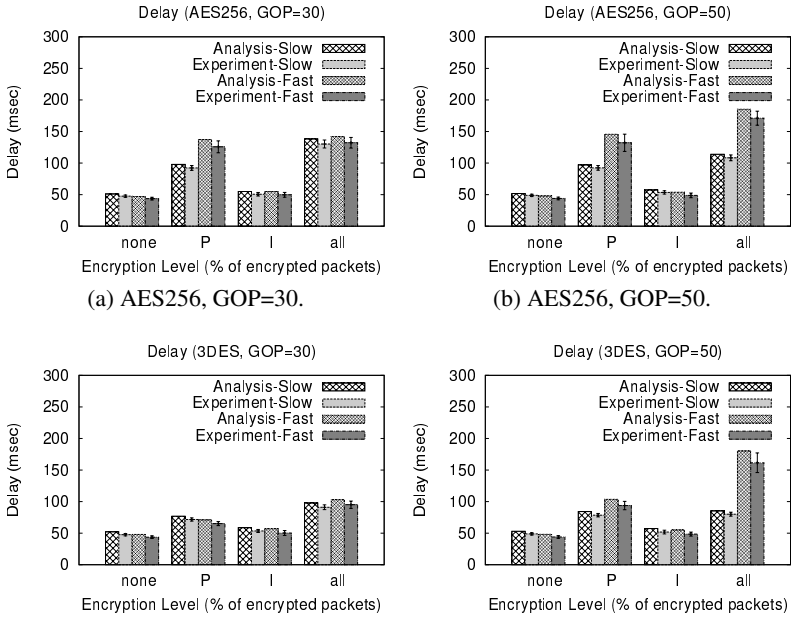
Distortion at an eavesdropper due to the encryption of *I* and *P* -frame packets: The



(a) Slow,none. (b) Slow, P. (c) Slow, I. (d) Slow, all.

(e) Fast, none. (f) Fast, P. (g) Fast, I. (h) Fast, all.

Figure 2.6: Screenshots of video flow at an eavesdropper's site (slow vs fast, GOP=30).



(a) AES256, GOP=30. (b) AES256, GOP=50.

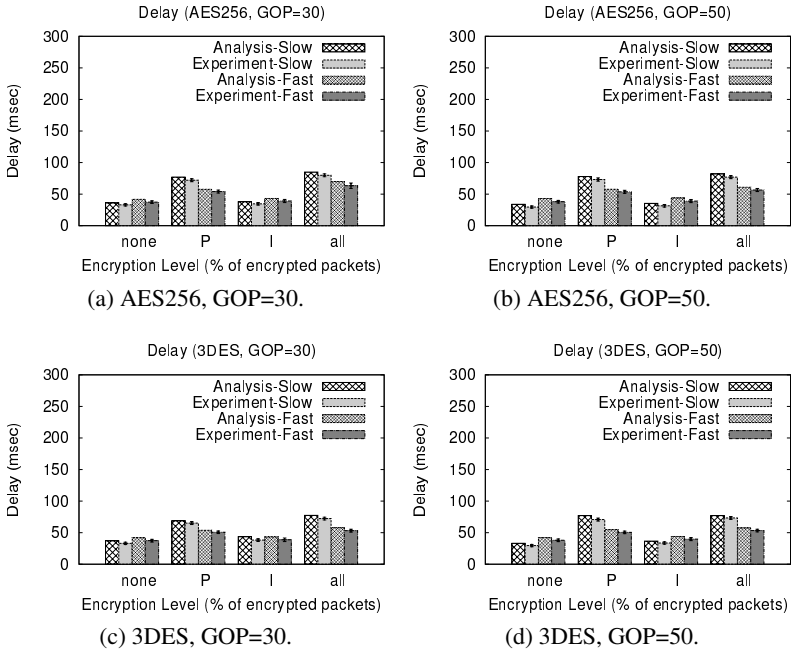
(c) 3DES, GOP=30. (d) 3DES, GOP=50.

Figure 2.7: Comparison of transfer latency in various cases (analysis and experiments with Samsung S-II).

distortion results are shown in Fig. 2.4 for both slow and fast-motion video flows. The factor that determines the video distortion is the percentage of packets that are correctly received and successfully decrypted. The legitimate receiver decrypts all the packets successfully delivered over the channel and the distortion here corresponds to the first bar in each plot, labeled "none". In con-

trast, the eavesdropper experiences higher distortion since it cannot correctly decrypt packets. As a general observation, the analytical results closely match the experimental results. The encryption of *I*-frame packets plays a more significant role in degrading the video quality (up to 80%) at the eavesdropper compared to the case where only *P*-frame packets are encrypted (the largest decrease observed here is 40%). This is to be expected since the *I*-frames carry a lot more information regarding the video content. Moreover, the encryption of *I*-frame packets degrades the video quality at the eavesdropper to a greater extent for the case of slow-motion video (80%) compared to the fast-motion video (30%). This is because the *I*-frames carry most of the information in the former case. The loss of *P*-frames affects video with fast motion to a higher extent, since in this case, these frames carry a lot more information (as compared to slow motion video flows). The Mean Opinion Score (MOS), which is a subjective metric that represents the quality of the video at the eavesdropper's site is given in Fig. 2.5, while Fig. 2.6 contains video screenshots as seen at the eavesdropper's site, for both slow and fast motion video flows. The Mean Opinion Score (MOS) gives a numerical indication of the perceived quality of the received video clip. It is expressed as a number from 1 to 5, where 1 indicates bad quality and 5 the best quality. Although MOS is subjective, there is software that measures the MOS on network transfers. For our experiments we report MOS values as measured by the EvalVid toolset. Note that the MOS drops to the lowest levels (≈ 1) with partially encrypted flows. This essentially implies that the video is practically unviewable by the eavesdropper.

Latencies with *I* and *P* frame encryption: Figures 2.7 and 2.8 show the average delay per packet for each device, GOP size, motion level and encryption policy. A general observation is that the incurred delay when the *P*-frame packets are encrypted is larger than the delay for the case where *I*-frame packets are encrypted. In the case of the HTC Amaze 4G, this delay is almost equal to the extreme case where all the packets in the transmission are encrypted. The same is true for Samsung S-II for the AES256 encryption algorithm, but not for the 3DES encryption scheme. Furthermore, the delay in the case where the *I*-frame packets are selected for encryption is small and close to the delay when none of the packets are encrypted.



(a) AES256, GOP=30. (b) AES256, GOP=50.

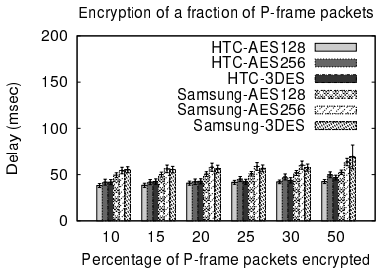
(c) 3DES, GOP=30. (d) 3DES, GOP=50.

Figure 2.8: Comparison of transfer latency in various cases (analysis and experiments with HTC Amaze 4G).

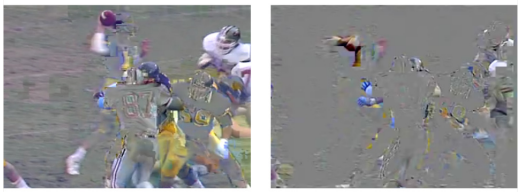
Finer control of protection for fast-motion video: An encryption policy where we encrypt a mixture of *I* and *P* frame packets can provide a finer control over the protection levels of the content. Going back to Figs. 2.4b and 2.4d, we observe that for fast motion video flows, the distortion at the eavesdropper is lower when we encrypt *I*-frame packets compared to the case where *P*-frame packets are encrypted. This is in contrast to what happens in the case of slow motion video flows. However, the delay for encrypting *I*-frame packets is lower than the delay that the *P*-frame packet encryption incurs (given the much larger volume of *P* frames). To achieve a better trade-off between delay and distortion, we examine the case where we encrypt all the *I*-frame packets and a fraction α, of the *P*-frame packets in a GOP for fast motion video. We experiment with different values of α and we show in Fig. 2.9a the corresponding transfer latency for each encryption algorithm and wireless device. Table 2.2 shows the delay, distortion and the Mean Opinion Score for Samsung S-II. We observe that the minimum value of α that provides an almost complete obfuscation of the video flow due to distortion is 20%. For that value of α, the power consumption is 1.48 Watt, while the power consumption is 1.28 Watt when only *I*-frame packets are encrypted (power

Table 2.2: Delay vs distortion.

	Delay	PSNR	MOS
I	48.41 msec	20.65 dB	1.71
I* + 10% *P	53.06 msec	17.8684 dB	1.26
I* + 15% *P	53.90 msec	17.6895 dB	1.24
I* + 20% *P	54.91 msec	17.3359 dB	1.20
I* + 25% *P	55.47 msec	17.1776 dB	1.17
I* + 30% *P	56.51 msec	16.4268 dB	1.15
I* + 50% *P	61.76 msec	16.0106 dB	1.14



(a) Upload latency.

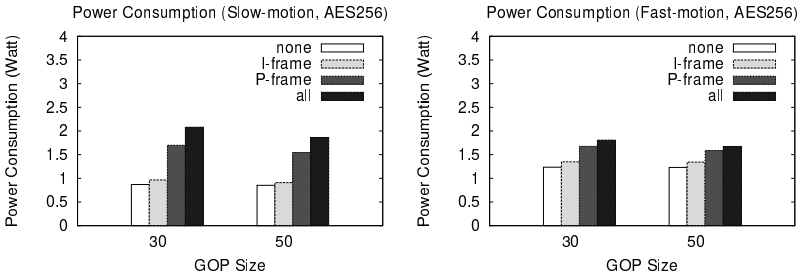


(b) Screenshots for *I* (left) and *I*+20%*P* (right) case.

Figure 2.9: Encrypting all *I*-frame and a fraction of the *P*-frame packets (GOP=30).

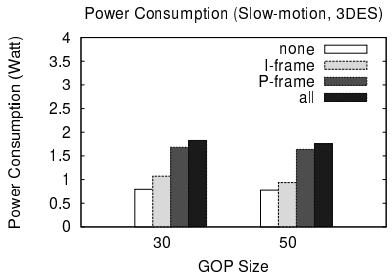
consumption is discussed in detail later in Section 2.6.3). The change in delay due to this additional encryption is only about 6.5msec. Figure 2.9b depicts screenshots at the eavesdropper's site in the case where only the *I*-frames are encrypted (left) and the mixture of *I* and 20% of *P*-frame packets encryption (right).

For slow motion video flows we observe from Figs. 2.4a and 2.4c that encrypting all *I*-frame packets results in a high distortion, to almost make the content invisible, at an eavesdropper's site. In order to save on energy consumption and delay, we examined the case where half of the *I*-frame packets are encrypted. We found that the distortion levels are similar to the case where all the *P*-frame packets are encrypted and thus does not provide adequate obfuscation.

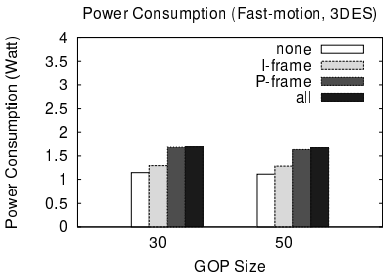


(a) Slow-motion, AES256.

(b) Fast-motion, AES256.



(c) Slow-motion, 3DES.



(d) Fast-motion, 3DES.

Figure 2.10: Power consumption with Samsung S-II.

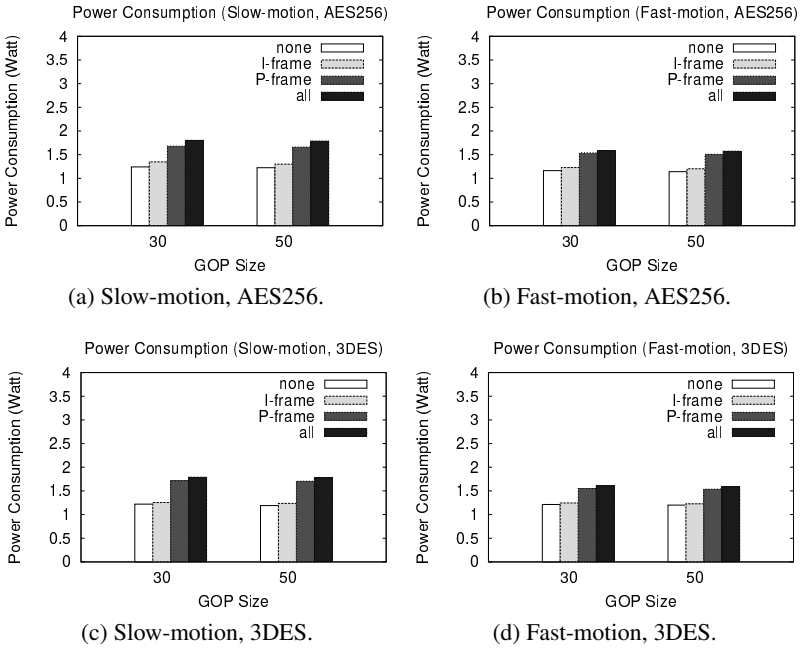
2.6.3 Power Consumption

To compute the power consumption we use the power monitor tool by Monsoon Solutions, Inc. and measure the amount of energy the mobile phone consumes during the video streaming. The reading v from the power monitor is in μAh which we convert into Watts as follows:

$$\frac{v \cdot \text{Voltage} \cdot 3600}{\text{stream duration}} \cdot 10^{-6}; \tag{2.29}$$

the Voltage is set to 3.9 Volts.

Due to the different sizes of the slow and fast motion video flows, we do not compare the power consumption between them; instead, we perform the comparison within the same type (slow or fast motion) of flows but with different encryption policies. The power consumption measurements for the Samsung Galaxy S-II phone are shown in Fig. 2.10, while those for the HTC Amaze 4G are in Fig. 2.11. The results are for slow and fast motion video and for the three encryption algorithms, for each GOP size. As can be seen, when the video stream is unencrypted the energy consumption is the lowest due to the fact that fewer CPU cycles are needed in order to process a



(a) Slow-motion, AES256.

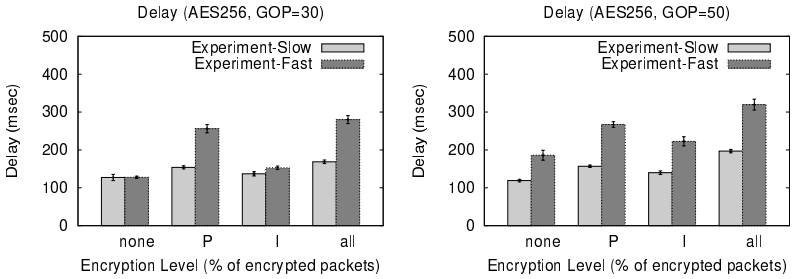
(b) Fast-motion, AES256.

(c) Slow-motion, 3DES.

(d) Fast-motion, 3DES.

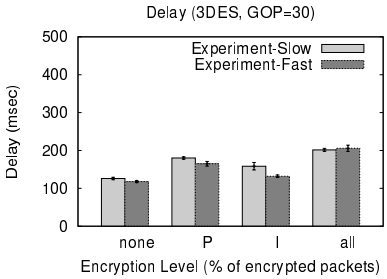
Figure 2.11: Power consumption with HTC Amaze 4G.

frame. On the other hand, a fully encrypted stream consumes the highest amount of energy. Furthermore, more energy is necessary when only the *P*-frames are encrypted compared to the case where only the *I*-frames are selected for encryption. This is so because the overall size of the *P*-frame packets together is larger than the overall size of the *I*-frame packets together. Considering the Samsung S-II, and for a slow motion video, an increase in the power consumption by 140% can be seen comparing the two extreme cases where none of the packets are encrypted and all the packets are encrypted. If only the *I*-frames are encrypted, the increase is only 11%. This translates to a savings of 92%. The power consumption increase for a fast motion video flow is lower, where the largest increase (by 50%) in the power consumption is observed when all the packets are encrypted. For the HTC Amaze 4G the increase in the power consumption is not as steep; the largest increase is by 50% and 38%, for the slow motion and fast motion video, respectively. For fast motion video, when all the *I*-frames and 20% of the *P*-frames are encrypted (to provide almost complete confidentiality), we find the energy savings to be 26% (reduction from 2 Watts to 1.48 Watt).

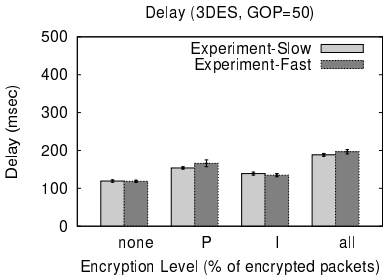


(a) AES256, GOP=30.

(b) AES256, GOP=50.

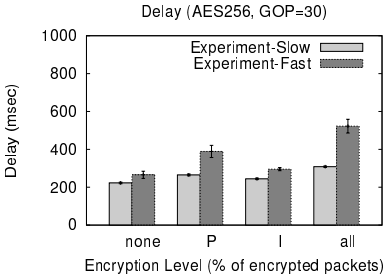


(c) 3DES, GOP=30.

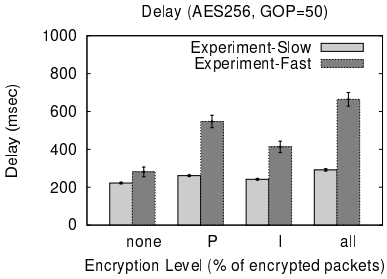


(d) 3DES, GOP=50.

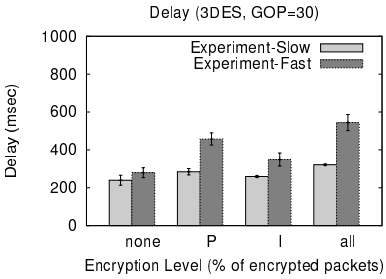
Figure 2.12: Comparison of transfer latency for HTTP/TCP (Samsung S-II).



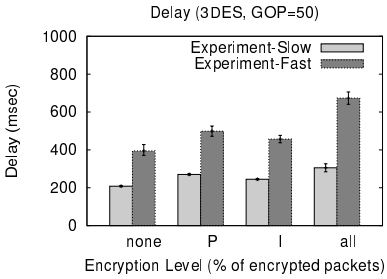
(a) AES256, GOP=30.



(b) AES256, GOP=50.

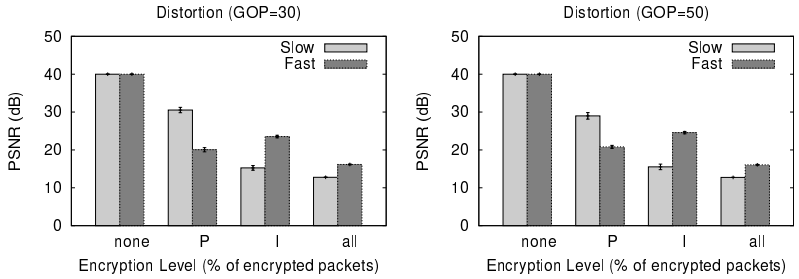


(c) 3DES, GOP=30.



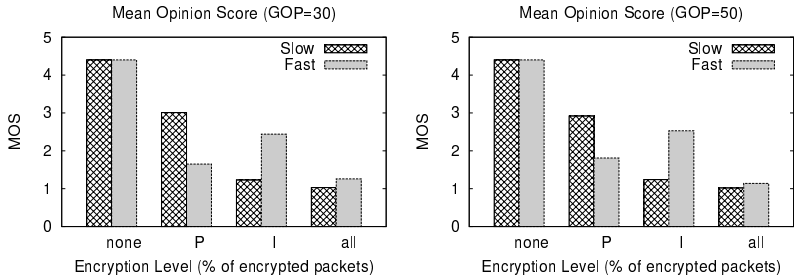
(d) 3DES, GOP=50.

Figure 2.13: Comparison of transfer latency for HTTP/TCP (HTC Amaze 4G).



(a) GOP=30. (b) GOP=50.

Figure 2.14: Distortion at an eavesdropper's site for slow and fast motion video flows with HTTP.



(a) GOP=30. (b) GOP=50.

Figure 2.15: Mean Opinion Score at an eavesdropper's site for slow and fast motion video flows with HTTP/TCP.

2.6.4 Experiments with HTTP/TCP

Next, we experimentally evaluate selective encryption for video traffic based on HTTP/TCP. A Marker bit is used again (in the option header) to indicate whether or not a packet is encrypted. The average delay per packet is shown in Fig. 2.12 and Fig. 2.13 for the Samsung S-II and the HTC Amaze 4G phones, respectively. The distortion and the mean opinion score for both the slow and fast motion video flows are shown in Fig. 2.14 and Fig. 2.15, respectively. The trend that is observed when RTP/UDP are used is also seen for the case of HTTP/TCP. While the latency is slightly higher (due to TCP retransmissions), it is reduced significantly, especially for fast motion video where the volume of packets is more. Since the fraction of packets encrypted remain the same, the energy benefits are identical to that with UDP/RTP; thus, we do not present these plots here.

2.7 Conclusions

Due to the widespread use of smartphones, video transfers over WiFi connections are becoming increasingly popular. We argue that only encrypting parts of a video flow can sufficiently distort the stream at an eavesdropper's site and thus render the content useless; at the same time such approaches can reduce performance penalties in terms of delay and energy. We refer to encrypting different parts of the stream as different modes of encryption. We develop a mathematical framework to characterize the effect of different modes of encryption on the delay at the client and the distortion at an eavesdropper's site. The framework provides an efficient way of determining the volume of video traffic that needs to be encrypted to preserve confidentiality at minimum performance cost. We validate our model via extensive experiments using Android smartphones.

Chapter 3

Droid M+: Developer Support for Imbibing Android's New Permission Model

3.1 Introduction

Application sandboxing and management of permissions to sensitive resources (permission model) are key components of modern mobile operating Droid M+ s for improving the security of individual apps and protecting the users' personal data and privacy. Prior to Android 6.0, Android was using a permission model that asks developers to declare required permissions in the manifest file and at installation time, asks users to either grant all requested permissions or to refuse the installation. Since its introduction, many published studies discuss various limitations of this model (*e.g.* [106], [151]).

The first, frequently criticized aspect of this old Android permission model is the creation of over-privileged apps [65]. Over the years, to enable access to increased functionalities of the Android platform [183], the number of Droid M+ permissions sought, have been boosted from 75 (API level 1) to 138 (API level 25). Lacking sufficient understanding of this permission model,

developers tended to ask for more permissions than necessary. For example, one-third of the 940 apps analyzed by Felt *et al*. are over-privileged [65]. Requesting unnecessary permissions is a severe security problem since attackers can leverage a multitude of combinations of these permissions to compromise the privacy of the user (*e.g*., leaking personal photos over the Internet).

Another problem with the old Android permission model is the lack of flexibility; users can neither grant a subset of all requested permissions, nor revoke granted permissions. A recent user study by Wijesekara *et al*. [187] showed that 80% of the participants would have preferred to *decline* at least one requested permission and one-third of the requested accesses to sensitive resources; this is because of their belief that (a) the requested permission did not pertain to the apps' functions; or, (b) it involved information that they were uncomfortable sharing. The lack of such flexibility has led Android users to either ignore the permission warnings [66], or to not use the app; for example, a recent survey [111] of over 400 adults found that over 60% of the participants decided not to install an app because it required many permissions. Irrevocable permissions also pose privacy concerns to users as apps can retain their access to sensitive sensors (*e.g*., microphone) while running in the background [108].

Wijesekara *et al*. [187] proposed using Nissenbaum's theory of context integrity [139] as a guideline to determine whether accesses to protected resources would violate users' privacy. However, lacking enough *contextual information*, the install-time permission granting model makes it very difficult for normal users to determine why a permission is needed and if the app would violate their privacy [66]. Moreover, they also found that even if the permission is requested during runtime, lacking proper mechanisms to explain why a particular resource was necessary could also lead to incorrect perceptions, and less willingness to grant the permission.

In Android 6.0 (Android M(arshmallow)), Google revamped the Android permission model to solve the aforementioned problems. In short, Android no longer promotes users to grant permissions during install-time; instead, *normal permissions* (*i.e*., no great risk to users' privacy and security) are automatically granted and *dangerous permissions* are requested during runtime. To further streamline the number of requests, dangerous permissions are put into *permission groups* and granting one dangerous permission would automatically grant other permissions in the same

group. To help developers communicate to users why a permission is necessary, Google also added an API to check whether additional explanation might be needed. Finally, users can revoke a granted permission at anytime using the Droid M+ settings. More details on the new permission model are provided in § 3.2; note that the permission model carries over to the next version of Android (Nougat or Android N).

While the new permission model is a significant improvement over the old model in empowering users with more control over their privacy and in making apps more appealing for adoption (recall that asking permissions at install-time may affect users' decisions on installing an app), we find that only a very few apps have effectively migrated to the new permission model. To verify whether this is a general issue for the entire Google Play Store, we conduct, to the best of our knowledge, the first Droid M+ atic measurement study towards answering the following questions:

- How many newly released apps have adopted the new permission model?
- For those that have not, what are the likely reasons?
- For those that have adopted, how well do they adhere to Google's guidelines [87]?

Our analysis results show that, despite a 26.7% market share of Android M, only 23.4% of the apps have adopted the new permission model. Further, our study shows that only 48% of the adopted apps educate users on why a permission is necessary and 5% of them crash if the user denies a request (due to lack of a corresponding callback function).

We attribute cause of this unsatisfying status quo to be the lack of proper development tools. In particular, to migrate to the new permission model, developers have to make non-trivial changes to their existing code. This is especially true if developers want their app(s) to follow Google's guidelines, *i.e.*, properly checking if a permission has been revoked, educating a user in-context on why a permission is necessary, and properly handling instances where a permission request has been denied. In support of this, we found that apps that request fewer permissions better conform to the new model than apps that request more permissions.

To solve this problem and thus improve the security of the Android ecoDroid M+ , we develop a tool set, Droid M+ , to help developers to retrofit their legacy code to the new permission

model. In particular, given the source code of an Android app, our tool will (1) identify different *functionalities* (*i.e.*, context) of the app; (2) identify permissions that are required for each functionality; (3) automatically populate the entry of each functionality with an annotation that allows developers to review the requested permissions and provide corresponding justifications; (4) automatically translate the annotation into real Java code; and (5) provide a default callback function to handle denied requests. In summary, Droid M+ allows developers to easily morph their app(s) to support revocable permissions and adhere to Google's guidelines, with minimal changes to their existing code.

Via extensive evaluations, we demonstrate that Droid M+ can facilitate easy permission revocations as intended by Android M. Although not part of Droid M+ , we discuss how it facilitates a new permission model wherein permissions can be granted only when specific functionalities are invoked; a detailed study of this however, is deferred for future work.

In summary, this work makes the following contributions:

- We perform an in depth measurement study of 7000 top free applications (apps) from Google Play Store and examine the adoption of the new Android permission model. Our study reveal that only 23.4% of the apps support revocable permissions, at least 51% of these apps do not follow the guidelines from Google, and 4.8% of them crash if user denies a permission request.
- We design, implement and evaluate Droid M+ , a tool set that aids developers in easily incorporating Android's new permission model and adhere to Google's guidelines.

3.2 Background and Motivation

In this section, we provide background on the new Android's permission model and Google's guidelines on how permission requests and revocations should be handled.

The Android M Permission Model: Android is a privilege separated OS, in which each app runs with a distinct Droid M+ identity. Fine-grained security protection is provided via a permission mechanism that restricts access to sensitive sensors, users personal data, other services/apps, etc. To access those resources, an app must declare the required permissions in its "manifest file."

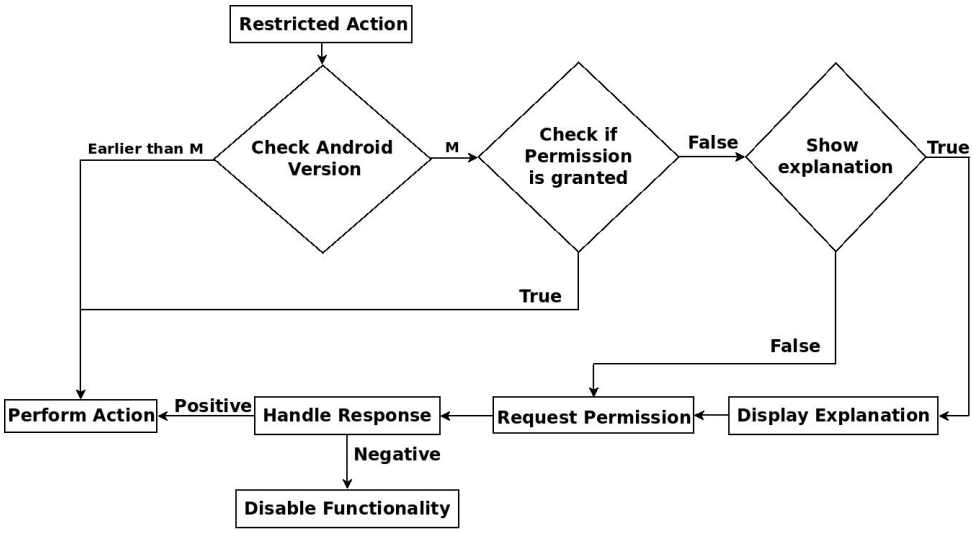


Figure 3.1: The permission workflow of Android M.

What are considered normal permissions, are automatically granted to the app. What are considered dangerous permissions need to be explicitly granted by users; with Android M, these are requested at run-time. Users are prompted with a pop-up box when an app first seeks to use a protected resource (seeks a dangerous permission). Three options are provided: (1) users can grant the permission and the app will retain access to the resource; (2) users can deny this particular request; or (3) for a permission that has been previously denied, a chat box is provided to automatically deny all future requests. If the user denies a permission request, Android M allows the app to either continue running with limited functionality or completely disables its functionalities. Dangerous permissions are further put into permission groups; if one permission in a permission group is granted by the user, the remaining permissions in the same group will be automatically granted. Currently there are the following new permission groups: calender, camera, contacts, location, microphone, phone, sensors, sms, and storage. Android M also allows the user to modify the permissions granted to apps using Droid M+ settings. Note that a user can also revoke permissions for legacy apps (API level < 23); in such cases, the platform will disable the APIs guarded by the permission by making them no-ops, which may return an empty result or a default error.

To provide truly revocable permissions, developers should follow the following steps (Fig. 3.1):

- At each instance when an app needs to invoke API(s) that require a permission(s), the developer should insert a call to explicitly check if the permission(s) is granted. This is key because users may revoke granted permission(s) at anytime, even for legacy apps. Developers can use the `ContextCompat.checkSelfPermission` method from the support library or directly invoke platform APIs to do so.
- Optional but recommended by Google, developers might want to help the user understand why the app needs a permission. Towards this, the developer should specifically write code to display the proper reasoning to the user before requesting a permission.
- If the app does not have the permission(s) in order to complete a restricted action(s), the developer should insert a call to request the permission(s). Developers can do so by calling the `ActivityCompat.requestPermissions` method from the support library or directly invoke platform APIs. If the permission(s) has not been permanently denied, the platform will display a dialog box to the user showing which permission group(s) are requested.
- Since permission requests are asynchronous, a callback method must be provided to handle the results of such requests. Overriding the `onRequestPermissionsResult` method can help this process. Upon being invoked, this method provides a list of the permissions granted and denied.
- The developer should handle both positive and negative responses from the user. In the case of denied permissions, the app could either continue execution with limited functionality or disable the corresponding functionality and explain to the user why the permission was critical.

Google's guidelines: Google offers guidelines for permission management [87], which suggest that permission requests be simple, transparent and understandable. These attributes promote user adoption as shown by the results in [169, 187], *i.e.*, users were more willing to grant permission(s) when requested in-context and with proper justifications. Specifically, Google recommends that permissions critical to the correction functionality of the app (*e.g.*, location to a map app) be requested up-front, while secondary permissions be requested in-context. If the context itself is not self-explanatory (*e.g.*, requesting the camera permission when taking a photo is understandable but the reason for requesting location at the same time lacks clarity), the app should

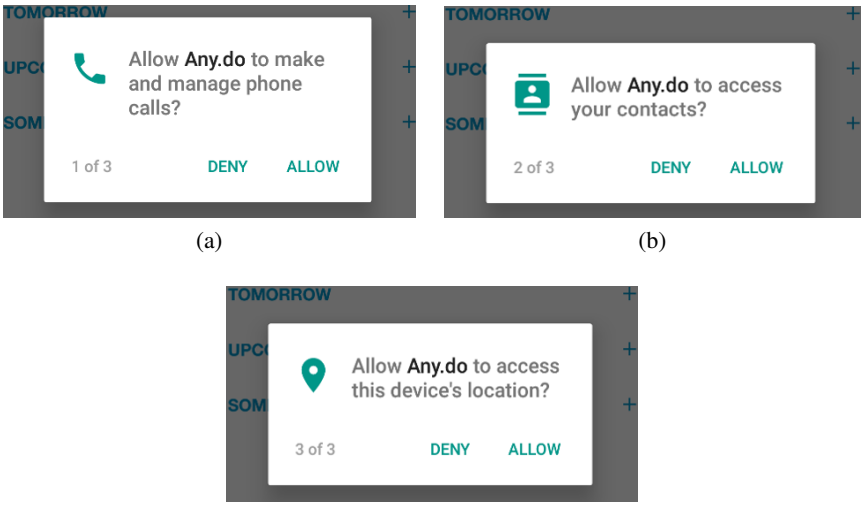


Figure 3.2: Any.do permissions during startup.

provide education to the user about why the permission is requested. The education recommendation also applies to critical permission(s) asked up-front. When a permission is denied, the app should provide feedback to the user and if possible provide other available options. If critical permissions are denied, the app should further educate the user as to why the permission is critical for the app to function and offer a button so that the user can grant it. For secondary permissions, the app should disable the corresponding features and continue providing the basic functionalities.

3.3 Measurement Study

In this section, we present our in-depth measurements on Android apps from Google Play. Our goal is to understand the extent to, and the way in which developers have adopted Android M's new permission models.

To aid discussion, we define what we call functionalities. We define a functionality to be a specific (unique) capability of the app, that can only be performed if one or more revocable permissions have been granted. For example, consider an application that is capable of taking and saving photos to a user's device; this capability is a functionality that needs the CAMERA and

STORAGE permissions. A more rigorous definition of functionalities is provided in §3.4.

Two functionalities are considered different, if they are invoked at different points inside the app code (even if they need the same permissions). For example, when a user searches for restaurants around her current location it is considered a functionality that is different from one wherein, inside the app the user's location is being shared with third parties libraries (e.g. Ad networks); note that this is the case even though both functionalities need and request the same permission (Location). The points within the code that we refer to can either depend on (a) user input (e.g., click a button) or (b) components of the app (e.g. Services, Activities, etc).

As a motivating example, we consider an app `Any.do` [24], one of the most popular to-do apps on Google Play, recently updated in November 2016. To implement its functionalities, this app requires access to the microphone, location, contacts, calendar, the device identifier, and local storage. It is not clear why a to-do app would need all such permissions. The app description page on Google Play does not offer proper information either. Further, although this app does target the new API level and thus should support the new permission model, the way the permissions are requested does not adhere to Google's guidelines. In particular, when the app is first launched, all the permissions are requested up-front (Fig. 3.2). At this time, it is unclear why these permissions are required by a to-do application, and no further explanations are offered (even though they are legitimately used). Given this motivating case, next we perform an in depth measurement study of the top applications on Google Play to understand their structure and how they adopt the new permission framework in Android M.

3.3.1 Measurement Tool

We design and implement a novel tool, the *Revocable Permission Analyzer*, to experimentally quantify via different metrics, the way existing apps are developed using the new permission model of Android M. Its basic functionality involves analyzing Android APKs, generating the call graph of the application under consideration, and marking the methods that contain dangerous permissions or requests for permissions. Note that in Android M and later, the revocable (dangerous)

Permission Group	Permissions
CALENDAR	**READ_CALENDAR, WRITE_CALENDAR**
CAMERA	**CAMERA**
CONTACTS	**READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS**
LOCATION	**ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION**
MICROPHONE	**RECORD_AUDIO**
PHONE	**READ_PHONE_STATE, CALL_PHONE, READ_CALL_LOG WRITE_CALL_LOG, ADD_VOICEMAIL, USE_SIP PROCESS_OUTGOING_CALLS**
SENSORS	**BODY_SENSORS**
SMS	**SEND_SMS, RECEIVE_SMS, READ_SMS RECEIVE_WAP_PUSH, RECEIVE_MMS**
STORAGE	**READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE**

Table 3.1: Dangerous Permissions and permission groups

permissions are the ones that are classified as *PROTECTION_DANGEROUS* (see Table 3.1). It does not include permissions such as *INTERNET*, which are granted automatically at the time of the installation of the application.

The *Revocable Permission Analyzer* first leverages apktool [26], a tool for reverse engineering binary Android apps, to decompile and decode the APK's resources including the manifest file and the UI XMLs file. It uses androguard [23], a tool written in Python for statically analyzing Android APKs, to generate the call graph of the APK. The call graph is a control flow graph, that captures the "calling relationships" between methods in an application. These call graphs are then analyzed using static analysis (using an approach similar to that used in Droid M+ as described in § 3.4.1) to determine how the applications with revocable permissions have been developed and determines if they follow Google's guidelines. The static analysis looks for methods that contain API calls to *checkSelfPermission*, *requestPermissions*, *shouldShowRequestPermissionRationale* and *onRequestPermissionsResult*. By focusing on these API calls, it can examine (a) when the application is requesting or even checking for dangerous permissions, (b) when the application shows a rationale for requesting the permission and (c) what it does after the user responds to the permission request.

As described in § 3.2, Google's guidelines suggest that only permissions that are critical and obvious should be asked up-front, during the launching of the app. It also suggests that de-

velopers should educate the users when they ask non-obvious permissions in-context. *Revocable Permission Analyzer* checks to see how many requests for permissions are invoked when the main "Activity" is launched for a given app. To do so, the algorithm simply conducts a reachability analysis on the call graph where the root node is the *onCreate*, *onStart*, or *onResume* method of the main activity (which are invoked when the activity is launched or resumed). If the root node can call any method *requestPermissions()*, the corresponding permission is said to be requested upfront. For permissions not asked upfront, we conservatively consider them as being asked in-context.

Listing 3.1: Customized Messages

```
1  WrapperMethod(...) {
2    if (ActivityCompat.checkSelfPermission(this, permission) !=
3      PackageManager.PERMISSION_GRANTED) {
4      if (ActivityCompat.
5      shouldShowRequestPermissionRationale(this, permission)) {
6        // display reason
7        ActivityCompat.requestPermissions(...);
8      } else {
9        // display feedback
10     }
11   } else {
12     RealMethod(...);
13   }
14 }
```

Revocable Permission Analyzer is also capable of checking if the developer includes customized messages to educate the users about why permissions are requested. The code snippet shown in Listing 3.1 is an example where customized messages are displayed, as recommended by Google. Specifically, the *shouldShowRequestPermissionRationale()* API call is invoked to check whether customized message needs to be shown; if so, it will display a message to the user and then call the *requestPermissions()*. If *shouldShowRequestPermissionRationale()* is not encountered, then

it is a strong indication that no customized message and education is attempted. In the case when customized messages are indeed shown, we look up the message from the strings.xml resource file and manually evaluate them in terms of how meaningful and informative they are.

3.3.2 Android Applications Dataset

Our measurement study is based on 7000 applications that are obtained by downloading the top free apps from each available category (e.g. Social, Games, etc.) as per Google's Play Store [86] charts from August 2016. We perform the study on a Lenovo Laptop with 4-core Intel i7 CPU and 16GB RAM, running Ubuntu 16.10 and Oracle's 1.8 JDK.

3.3.3 Results and Inferences

Adoption of Android M permission model

A large number of Android M apps do not support revocable permissions. From the 7000 apps, only 1907 are developed for Android M or above (with a targetSdkVersion of 23 or higher). Further only 1638 apps, or 23.4% of the total, actually used the Android M permission management APIs such as *requestPermissions()*. Besides, there are 219 apps out of the 1907 that do not require *any* so called dangerous permissions and thus, in a normal way do not invoke any Android M APIs. We point out that, surprisingly, there are apps like Ringdroid, developed by Google itself, that do not use revocable permissions even though they are developed with the latest Android SDK (see § 2.6 for more details).

As reported in [83], in December 2016, the share of Android users that use Android M and N, is 26.7%; one can expect this percentage to keep growing. Unfortunately, the above result shows that most of the apps do not support revocable permissions. This implies that a user who has a phone with a version of Android that supports the latest fine-grained permission mechanism, will be forced to grant all the permissions to most of the applications; otherwise these applications will likely not function correctly [64].

Permission revocation support is more common in apps that require fewer permis-

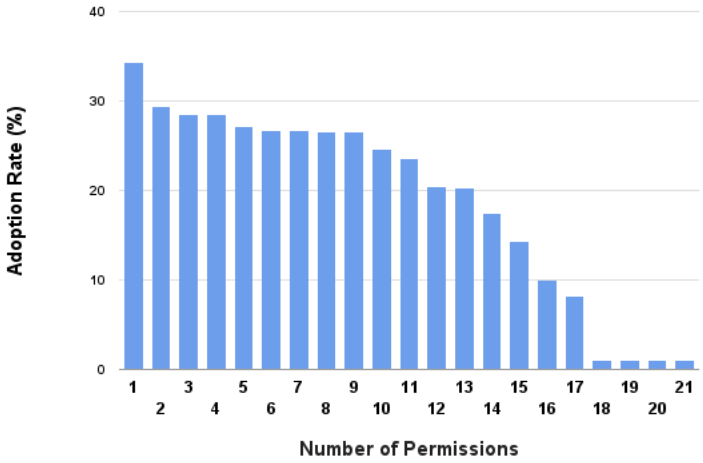


Figure 3.3: Adoption rate per number of permissions.

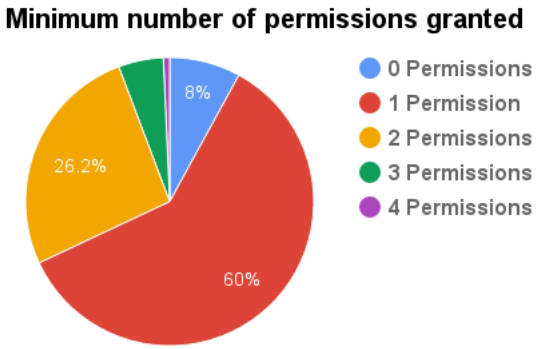


Figure 3.4: Critical permissions that can/should be asked upfront.

sions. The preliminary report also logs data on how many permissions are being asked by the various apps. Fig. 3.3 shows the adoption rate of the Android M permission model with varying total number of dangerous permissions requested. Note that when we say an app has adopted the Android M permission model, we mean that it has invoked one or more Android M permission management APIs such as *requestPermissions()*. It is clear that the more dangerous permissions asked, the less likely developers will make an effort to imbibe revocable permissions. At a higher level, it

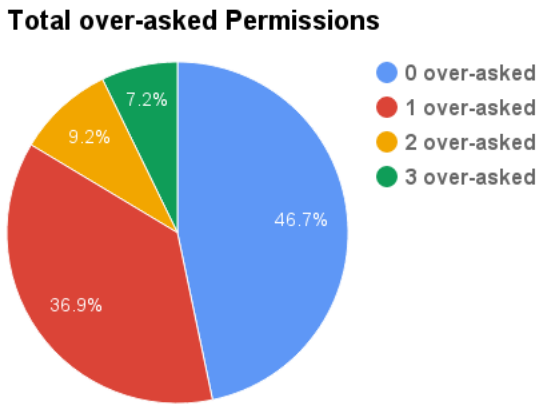


Figure 3.5: Over-asked permissions during launch.

suggests that the lack of friendly developer support could be hindering the adoption of Android M.

User Education

A significant fraction of apps does not provide meaningful explanations for non-obvious permissions. We find that from the apps that request permissions in-context, only 51.5% (803 in total out of 1558 apps) educate the users. An app is educating the user if the message that it provides during the request is meaningful and the user can understand why the requested permission is needed for a given functionality. For example, when the app SONGily [164] requests the storage permission, the following message is provided: 'Permission to write files is required". Clearly this message does not educate the user. Similarly, AskMD [28] provides the following message when a request for accessing the microphone is being made: 'AskMD would like to access your microphone. Please grant the permission in Settings.". Contrary to those apps, theScore [172] provides the following message when it requests access to the user's calendar: 'In order to add events to your calendar, we require the Android Contacts permission. We will not be reading or accessing your information in any way, other than to add the events.". Clearly, this message follows Google's guidelines by properly educating the user why it needs that permission.

Permissions asked upfront are less likely to have meaningful explanations. From the apps that ask permissions upfront, only 178 (39%) educate the user properly. This is a much lower rate compared to permissions that are asked in-context. Permissions that are asked upfront lack the context and it is generally even more important to educate the users about what the permissions are used for. Unfortunately, the results indicate that a majority of the considered apps fail to adhere to the Google's guidelines.

Permissions Asked Upfront vs. In-Context

A significant fraction of apps ask permissions upfront (during the launch of the main activity) instead of in-context. We find that 28% of the apps (458 out of 1638) request permissions during startup, in hope that the users will grant the permissions and thus, they do not need to ask again later on. As discussed later, most apps (1558 out of 1638) still attempt to request the same permissions again in-context even if the permission requests are denied upfront. Often, these permissions are not really critical and the app can still function even without them. To understand if such permissions are critical to an app's functionality (and therefore have legitimate reasons to be asked upfront), we manually check each application's description page in Google Play, including the categories that they belong to and the kinds of functionalities offered. For example, a camera application is expected to request the Camera permission; similarly an app belonging to a category such as travel/navigation can be expected to ask for the location permission. We admit that this approach is subjective; however if we as technically savvy researchers are not able determine why a permission is needed for an app by reading its description, then it is unlikely other users will. Fig. 3.4 shows the distribution of the number of the critical permissions asked upfront by the 458 apps; 60% have only one critical permission while 26.2% require 2 permissions and 5.8% require 3 or more. This shows that in general very few permissions are considered critical and should be asked upfront. Unfortunately, in most cases, apps often ask more permissions than necessary. Fig. 3.5 shows the total number of over-asked permissions that those apps are requesting upfront. Clearly, with respect to more than 52% of the apps, one or more permissions requested upfront are in fact not critical.

Some of the apps expect all permissions to be granted upfront or will simply refuse to run. Interestingly, even though they support the Android M permission model and are using the corresponding APIs, some apps simply expect all the permissions asked upfront to be granted; otherwise they simply refuse to run. We leverage the Revocable Permission Analyzer that checks the statements invoked when a requested permission is refused; if the statements like `Droid M+ .exit(0); or finish();` are encountered, it is evident that the app is simply voluntarily ending its run due to permission revocation.

This style of such an app defeats the purpose of revocable permissions as it does not really intend to support revocation of permissions (even when some of the permissions are not critical). Overall, using this approach, we are able to identify 4.8% or 80 apps out of 1638 apps that ask at least one non-critical permission, and yet refuse to run if such a permission is not granted. The remaining 1558 apps still ask for these permissions (again) in-context, even though they were denied when requested up front.

Summary. Our measurement study demonstrates that only a small percentage of applications that were built on the Android M platform, are using the new permission model. An even smaller number of these applications, unfortunately, adhere to Google's guidelines. One of the possible reasons for developers not fully and properly adopting the new permission model, is the complexity and the work associated with transitioning their apps from the previous Android version for which the app was developed, to the newer version. As users become more privacy conscious [111, 187], following Google's guidelines can be a key factor influencing their choice of apps. Below is a list of our key observations:

- 75% of the top apps supporting Android M API level or above are not in fact utilizing the Android M permission revocation APIs.
- 49.5% and 39% of the permissions asked in-context vs. upfront do not have informative explanations for why the permissions are sought as per the Google guidelines.
- 28% of the apps supporting permission revocations ask permissions upfront and 52% of them ask one or more permissions that are non-critical.

- Some of these apps ($\approx$ 5 %) simply refuse to run if any of the permissions asked upfront are not granted.

3.4 Droid M+ Tool Set

In this section, we describe the design of Droid M+ and its component tools. Droid M+ consists of three major components(see Fig. 3.6). The first component is a static analysis tool that helps developers identify different functionalities (*i.e.*, context) of their apps, what permission(s) each functionality requires, and the right place to request the permission(s). The second part is an annotation Droid M+ that facilitates the easy integration of revocable permissions and conformance to Google's guidelines within existing Android app code. Finally, Droid M+ contains a compiler extension that interprets the annotations and inserts the corresponding code.

3.4.1 Static Analyzer

The static analyzer has three tasks: (1) identify functionalities within an app, (2) identify permission(s) required by each functionality, and (3) identify the right place to annotate. The primary function of the static analyzer is to help developers migrate apps that are developed against an old API level to the new API level ($\geq$ 23). However, apps that are already developed for the new API level can also utilize this tool to help refactor the code, *i.e.*, determine what permissions to request, where to place the requests, and what education message to display with each request.

Identify Control Flow

Before we can do any useful analysis on an app, the tool needs to first parse the source code and generate the corresponding call graph and control flow graph. These are standard techniques which we will not describe in detail. As discussed in the literature (e.g., [160]), there are two challenges worth mentioning. First, point-to analysis [160] needs to be employed in order to generate an accurate call graph. Second, Java reflection needs to be handled to generate the complete call graph. Currently, we do not support the latter but there are ways to statically resolve the Java

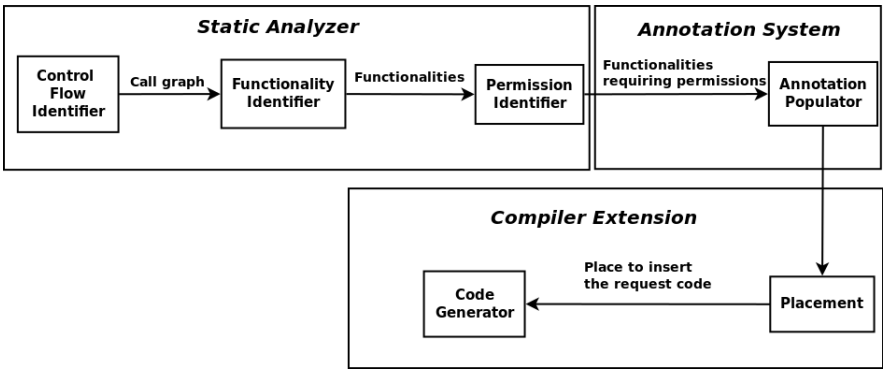


Figure 3.6: Droid M+ architecture.

reflection calls [162], and we plan to incorporate these in the future.

Identify Functionalities

Given a call graph, we define a *functionality* as a collection of nodes in the control flow graphs that are reachable from a single *entry point*. In Android, entry points include activities (*i.e.*, the `onCreate()` method of the `android.app.Activity` class), callback methods for UI interactions (*e.g.*, click of a button), content providers (app data managers), services (background threads), and broadcast receivers (Droid M+ -wide broadcast events). These entry points can be identified by parsing the manifest files and analyzing the code. The reasoning is that these entry points represent user-triggered events, or significant activities that should be made aware to users (*e.g.*, background services). Each functionality should contain a sequence of instructions involving some usage of permissions. We believe it is a natural and reasonable place to request permissions; this practice is aligned with the permission management guidelines by Google and is actually used in many real-world apps that perform in-context permission requests (*e.g.*, WhatsApp Messenger [185]). As most existing static analysis tools for Android already support the identification of all the entry points of an app and building a complete call graph ([23], [165]), we omit the details here.

Identify Permissions

The first step in identifying the required permissions, is to parse the manifest file and find out the target API level of the app. Note that although our tool helps migrate the app to the

new permission model of Android M (API level 23), this step is still necessary for supporting the newer version of the SDK (*i.e.*, if the app's current `targetSdkVersion` is lower than 23, we assume 23; otherwise we use the app's `targetSdkVersion`). The API level is used to map SDK APIs to their required permissions. Specifically, we use PScout [30] to generate the database that maps a permission to a set of SDK APIs that require this permission.

With this mapping information, identifying permissions required by a functionality is straightforward. In particular, given the complete static call graph, we use a standard reachability analysis for Android apps to identify all the potential invocable SDK APIs from the entry point of a functionality. Then we use the permission mapping to generate all the required permission(s) for this functionality.

Third-party Libraries. Some third-party libraries such as advertisement libraries (*e.g.*, AdMob) and analytic services could also access resources protected by permissions (*e.g.*, location). Because these libraries are usually delivered in binary (bytecode) format, we need additional steps and different analysis tools to identify the permission(s) required by these libraries. Specifically, we first collect all calls to the third-party libraries. Then we decompile the byte code of the libraries. Finally, we perform the same reachability analysis starting from the invoked methods to identify all the SDK APIs that may be invoked and map them to the required permissions, which is similar to Stowaway [65]. Note here that Droid M+ currently does not support native libraries.

Populating Annotations

We use Java annotation language (see § 3.4.2) to make it easy for developers to provide an explanation for a permission request. Assuming that a functionality has a single purpose, we ask developers to only provide the annotation once, as all accesses to the same protected API will share the same purpose. Under this assumption, we place the annotation at the entry point of each functionality, with two exceptions viz., background services and libraries. Background services are different since they are not a subclass of `Activity` and thus, cannot invoke the `requestPermission` method to prompt users. Libraries are different for several reasons. First and most importantly, a library may be used in many functionalities, including background thread. Be-

sides, the `onRequestPermissionsResult` method is bound to each Activity, so if a library is used in different Activities, it also creates confusion. Second, it is unreasonable to ask first-party developers to provide explanations for why a third-party library would require a permission(s). And instrumenting libraries distributed in binary format requires additional efforts. As a result, Droid M+ places the annotation at the method where the background services are started (`startService`) and where the library methods are invoked.

3.4.2 Permission Annotations

According to the Google guidelines [87] and previous studies [169], users are more likely to grant a permission if the developer provides an explanation for why the permission is needed. Developers should also provide feedback if a permission request is denied and accompany this feedback with a button leading to the Droid M+ settings for enabling the permission(s). Unfortunately, while it is easy to automatically identify all the required permission(s) of a functionality, automatically generating the corresponding explanations and feedback is much harder. Hence, our current design seeks developers' help for generating the explanations and the feedback. To ease this process, we use the customizable Java annotation Droid M+ [141] to capture the explanations. List. 3.2 provides an example of the annotation we use for declaring dangerous permissions and providing their justifications. Starting with `@Permission`, the annotation includes: (a) a name for the functionality; (b) an array of permissions, where each array element is a tuple `<perm, reason, feedback>`. `perm` denotes the requested permission, `reason` denotes the optional justification, and `feedback` denotes the optional message to be shown if the permission is denied.

For example, the "Attach photo to task" functionality of Any.Do could be annotated as:

Listing 3.2: Permission Annotation.

```
1 @Permission(
2   functionality ={"Attach photo to task"},
3  request={
4   {"READ_EXTERNAL_STORAGE", "Require storage to access your photos.", "You won't be able to
        attach photos."}
```

```
5  })
6 public void fromGallery() {
7    // code
8 }
```

3.4.3 Compiler Extension

We use Droid M+ 's compiler extension to interpret the permission annotations and generate the corresponding code. For each required permission, we use Google's example code [84] as the template towards generating the code:

Listing 3.3: Generated code.

```
1  SuitableMethod (...) {
2    // begin of template
3    if (ActivityCompat.checkSelfPermission(this, perm) !=
4      PackageManager.PERMISSION_GRANTED) {
5      if (ActivityCompat.
6      shouldShowRequestPermissionRationale(this, perm)) {
7        // display reason
8        ActivityCompat.requestPermissions (...) ;
9      } else {
10       // display feedback
11     }
12     return ;
13   } else {
14     WrapperMethod();
15     return ;
16   }
17   // end of template
18 }
19
20 @Override
```

```
21 public void onRequestPermissionsResult(int requestCode, String[] permissions,
22   int[] grantResults) {
23   // length will always be 1
24   if (permissions[0] == permission) {
25     if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
26       WrapperMethod();
27     } else {
28       // display feedback
29     }
30     return;
31   }
32 }
```

Here the `perm`, `reason`, and `feedback` are from the annotation. If the `reason` or the `feedback` is empty, we use the string ‘‘`${functionality} requires ${perm}.`’’. Our compiler extension ensures that the `functionality` cannot be empty.

While populating the template is straightforward, the challenge is determining where the permission should be requested. In [125], Livshits *et al.* proposed four properties for a valid prompt placement: ***(a) Safe***: Every access to the protected resource is preceded by a prompt check; ***(b) Visible***: No prompt is located within a background task or a third-party library; ***(c) Frugal***: A prompt is never invoked unless it is followed by an access to the resource; and, ***(d) Not-repetitive***: A prompt for permission is never invoked if already granted.

In Android M, since a call to `checkSelfPermission` always guarantees the not-repetitive property and we have already annotated background services and calls libraries differently, we will focus on safety and frugality. To be frugal, we want to place the permission request as close to the resource access as possible, which also makes the request more likely to be *in-context*. However, the current design of the Android M's permission model makes it hard to implement this placement strategy. In particular, as already shown in the code template, `requestPermissions` is an asynchronous method and thus, when it is invoked, the execution will not be blocked. Hence when the

execution reaches the next statement, the permission(s) may not be granted yet and invoking the protected API can crash the app. The standard way is to immediately exit the current method after requesting the permission. At the same time, after the user responds to the permission requests, the execution is resumed in the `onRequestPermissionsResult` callback function instead of the statement following `requestPermissions`. The problem is that if local variables are used in the access to the protected APIs, then they will not be accessible in the callback function; similarly and more fatally, if the method that accesses protected API returns a value (*e.g.*, location), then we have no way to return that value to the caller. Due to this problem, we choose to sacrifice some degree of frugality to avoid the need to drastically refactor the code.

Placement Algorithm. For each functionality that has an annotation, we use a placement algorithm to insert the "permission requesting" code. Our placement algorithm is similar to the one proposed in [125], with two key differences. First, as mentioned above, because of framework support, we do not need to consider the non-repetitive constraint. Second, our algorithm does not try to avoid third-party libraries and background services because they are *not* annotated. Instead, we walk up the dominator tree to avoid each method whose return value depends on the protected API and will be used by its caller(s).

First, for each annotated functionality, we initialize a job queue and into which pairs `<sensitive call, current method>` are inserted. Here a sensitive call denotes an invocation to a SDK API, a library method, or a background service that may require permission(s). For each pair in the job queue, we perform a backward search to check if the permission has already been requested. Note that according to Android's documentation [88] because the permission group may change in the future, developers should always request for every permission even though another permission in the same permission group may already be asked. Therefore, when checking for existing permission requests, we do not consider (1) whether a permission within the same permission group has been requested and (2) whether a permission that implies current permission (*e.g.*, `WRITE_EXTERNAL_STORAGE` implies `READ_EXTERNAL_STORAGE`) has been requested.

If a permission has not been requested, we check whether the current method is a suitable method. A method is suitable if (1) it is a void method, (2) its return value has no dependencies on

the sensitive call, or (3) its return value will ever be used. If the current method is not suitable, for each call site of the current method, we push a new job pair `<current method, call site>` into the queue.

Once a suitable method is found, we place the permission request inside the method. We first create a wrapper method that replicates the code from the sensitive call to the end (return) of that branch. If the wrapper method depends on local variables, we use a `map` to store those variables before requesting the permission and retrieve them inside the wrapper method. After creating the wrapper method, we insert the permission request template right before the sensitive call and populate it with correct annotations and the generated wrapper method, as suggested in Lst. 3.3. Note that although some of the code after the template will become dead because the execution will always return before reaching that code, our current design does not try to eliminate it; instead, we rely on the existing dead code elimination pass of the compiler to eliminate this unreachable code.

The above process is repeated until the queue is empty. Note that because the entry point of a functionality is always a void method, this loop is guaranteed to terminate.

Background Services. Droid M+ 's placement algorithm can handle almost all cases, but it cannot handle exported background services. These are services that can be started through a "broadcasting intent". Since such services can be started by the Android framework, if they require permissions, Droid M+ must request the permissions up-front. We identify such services by parsing the manifest file. For any service with attributes `enabled = true` and `exported = true` and requires permission(s), we add the permission requests in the `onCreate` method of the main Activity.

Critical Permissions. Droid M+ currently does not support identifying permissions that are critical and should be requested up-front. For such permissions, developers have to add the requests manually as well as provide proper education on the welcome screen [87]. However, because granted permissions can always be revoked by users at anytime (through Droid M+ settings), the code snippets that Droid M+ inserts are still necessary for the correct functioning of the app.

3.5 Evaluations

In this section, we present the Droid M+ 's evaluations. Our evaluation focus on answering two questions:

- How applicable is Droid M+ i.e., how well can it handle today's apps on the Play Store?
- How good is our permission request placement algorithm?

3.5.1 Implementation

We implement the static analyzer of Droid M+ based on the soot [165] static analysis framework. We use apktool [26] and androguard [23] to analyze existing apps. Annotation interpretation and code insertion is done based on Java JDK 1.8 and using Java parser [105].

3.5.2 Applicability

We design Droid M+ to be a source code level tool set. Unfortunately, as there are only a limited number of open sourced Android apps, we evaluate the applicable of Droid M+ in two ways. First, using RingDroid as a case study, we showcase how Droid M+ would work on real world Android apps. Then we analyze 100 top apps from the Google Play store and quantify how many apps can be handled by Droid M+ .

Case Study: Ringdroid

Ringdroid [154] is an open source app that records and edits sounds. In this case study, we use the commit `955039d` that was pushed on December 2, 2016; actual source code and line numbers can be found in [154]. Although Ringdroid was developed by Google and was targeting the latest Android SDK (API level 25), it surprisingly does not support revocable permissions (built against API level 22). Instead, it just wraps access to protected APIs with a `try` block and catches the thrown `SecurityException`. This makes it a good example to showcase the benefits of Droid M+ .

Ringdroid requires four dangerous permissions: `READ_CONTACTS`, `WRITE_CONTACTS`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_STORAGE`. The static analyzer in Droid M+ , finds 11 functionalities that require permissions. Among them 8 of them are background functionalities and the remaining 3 are associated with button click. 9 requests are finally inserted by Droid M+ (2 of them are redundant). In all the 9 cases, the requests were inserted immediately before the associated sensitive call happens, because the containing methods are all void methods.

- The first functionality is the `onCreateLoader` interface, implemented by `RingdroidSelectActivity` when it returns a `CursorLoader` that requires the `STORAGE` permission. This method is invoked in the background by the Android framework when `LoadManager` related methods are invoked (*e.g.*, in the `onCreate` method at line 151 and 152). Droid M+ insert the requests before line 151 (152 is covered by this request), and the remainder of the code from line 151 to line 187 is replicated in a wrapper method.
- The `LoadManager` is also invoked in the `refreshListView` method. Since this method can be called from other UI events such as `onCreateOptionsMenu`, Droid M+ also placed a request for the `STORAGE` permission at line 528, with lines 528 and 529 replicated in a wrapper.
- In the `onDelete` method invoked from a button click callback function, Droid M+ also inserted a request for the `STORAGE` permission at line 473; the remainder of the function is replicated in a wrapper.
- The `ReadMetadata` method also requires the `STORAGE` permission. However, since it is not part of an Activity, Droid M+ has to move it up to the `loadFromFile` method of the `RingdroidEditActivity`. Droid M+ inserted a request at line 598, with the remainder of the function replicated in a wrapper. Note that inside the `loadFromFile` method, there is a background thread that also requires the `STORAGE` permission. However, as the permission has already been requested, no request is inserted for this thread.
- The `MICROPHONE` permission is also required by the `RecordAudio` method of the `SoundFile`. However, as it is invoked from a background thread, the request is inserted before the creation of the thread at line 755 inside the `recordAudio` method; the rest of the function replicated in a wrapper.

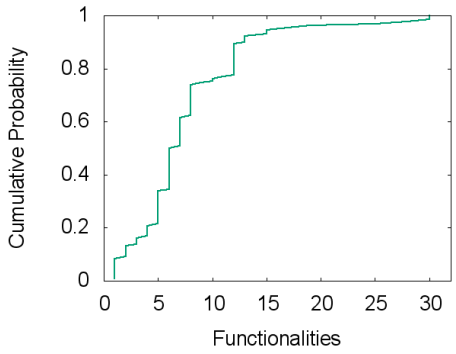


Figure 3.7: CDF of apps versus the number of functionalities that require permission(s).

- Three methods of the `SoundFile` class: `ReadFile`, `WriteFile`, and `WriteWAVFile` require the `STORAGE` permission. But as they are invoked in background threads, namely, one created in the `loadFromFile` method discussed above and another in the `saveRingtone` method, the request is inserted inside the creation method. In particular, this is done at line 1225. The rest of the function replicated in a wrapper.
- Similar to `RingdroidSelectActivity`, the `ChooseContactActivity` implements the `onCreateLoader` interface which will return a `CursorLoader` that requires the `CONTACTS` permission. The loader is initialized in the `onCreate` method. Thus, a request is inserted at line 129 with the code from line 129 to line 139 replicated.
- The `afterTextChanged` method handles UI events. A request to `CONTACTS` permission is inserted to enable `restartLoader`, at line 181; this single line (line 181) is replicated in a wrapper.
- The `assignRingtoneToContact` method, invoked from a button click callback function, also requires the `CONTACTS` permission. A request is inserted at line 154 with the remainder of the function replicated in a wrapper.

The reasons for requesting these permissions are self-explanatory; thus, we omit the annotations that were created.

Extended Measurement

To understand the applicability of Droid M+ to general apps, we perform the following measurements. First, because Droid M+ cannot handle native code that requires dangerous permis-

sions (*e.g.*, write to the external storage), we analyze how many apps from our measurement study (1) contain native code and (2) require external storage read/write permissions[1]. We found that 546 apps from among the 7000 apps match these criteria and thus, might not be handled by Droid M+ .

Next, we analyze 100 top Android M apps that do not have native code and require dangerous permissions. We choose Android M apps to ensure that we can also compare the permission requests placements. From among these apps, we found 698 functionalities that would request a total of 158 permissions up-front. On average, each application has 7 different functionalities that require permission(s); 90% of the apps have 13 "permission requiring" functionalities or fewer. Fig. 3.7 presents the CDF of the number of permission requiring functionalities of the analyzed apps. Among these functionalities that require permissions, 203 are Activities, 232 are UI event handlers, 201 are third-party libraries, and 60 are background threads. Most of the functionalities (579) require only one permission, 98 require two, 21 require three or more.

Regarding request placement, as was discussed in § 3.4, Droid M+ cannot place an annotation in a non-void method whose return value depends on the permission and would be used by its caller(s). To understand how common this situation is, we performed a more conservative measurement – how many non-void methods contain access to protected APIs. We found that 43% of the methods would returns a value and thus, the placement of the permission request may have to moved up to its callers. We also found that 11.3% of these requests might need to be placed on the entry method because all the other methods inside the functionality return a value.

Finally, by applying Droid M+ to these apps, only 48 permissions will be requested up-front (instead of 158), while the remaining will be asked instead, in-context; this corresponds to a decrease of 69.62% in permissions asked up front.

3.5.3 Quality of Request Placement

In this subsection, we evaluate the effectiveness of our request placement strategy (*i.e.*, examine whether the processed app would actually follow Google's guidelines [87]). To demonstrate that this is indeed the case, we first use an existing Android M app that supports revocable

[1]This is the only common permission that can be used within native code.

Group	Permissions
LOCATION	**ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION**
CONTACTS	**READ_CONTACTS, GET_ACCOUNTS**
CALENDAR	**READ_CALENDAR**
PHONE	**READ_CALL_LOG, READ_PHONE_STATE**
MICROPHONE	**RECORD_AUDIO**
STORAGE	**WRITE_EXTERNAL_STORAGE**

Table 3.2: Dangerous permissions requested by Any.Do and their corresponding permission groups.

permissions to show that Droid M+ 's placement would match the placement of the developers, *i.e.*, we are as good as manual placement. Then we reason about why the Droid M+ -processed app would follow the guidelines.

Case Study: Any.do

In this case study, we analyzed the free version of Any.do [24], one of the most popular "To do" and task list apps on Google's Play Store with a total number of installations between 10 and 50 million. This application was updated last on November 21, 2016 and it supports the new revocable permission model. We input the downloaded APK to Droid M+ 's static analyzer and find that the app requires 9 dangerous permissions (see Table 3.2).

Manual Request Placement: Upon launch, the app requests users for permissions for `LOCATION`, `CONTACTS`, and `PHONE`. No explanation or education is provided with regards to any of these. Upon further investigation, we find that none of the requested permissions are critical and the app would continue to function even if no permissions are granted. In addition to these up-front requests, all used permissions are also requested in-context; so they can be dynamically revoked.

- The `LOCATION` permission is used in four functionalities: one for attaching to a scheduled task for location based reminders, two for sharing the location to custom analytics classes, and one in a third-party location-based library.
- The `CONTACTS` permission is used in two functionalities: when the user needs to share a task with her friends and to display the caller of a missed phone call. If the permission is not granted, no feedback is provided and the corresponding functionality is not performed.

- The `PHONE` permission is used when the user seeks to be notified with regards to missed phone calls. If the permission is not granted, feedback "Missed call feature requires phone permission" is provided. It is also being used in the two analytics classes.
- The `CALENDAR` permission is requested in-context and used to cross-reference the tasks with the user's calendar for conflict detection, etc. If the permission is not granted, the feedback "Post meeting feature requires calendar permission" is provided.
- The `MICROPHONE` permission is requested correctly in-context when the user wants to add a voice message to a task that she schedules for later. If the permission is not granted, the feedback "Recording permission is needed" is provided.
- The `STORAGE` permission is also requested correctly in-context when the user wants to add a photo or a document to her scheduled task. If the permission is not granted, the feedback "This feature requires access to external storage" is provided. But this permission is also used by three different third party libraries.

In summary, to support revocable permissions, all permissions are checked before use; however, only 3 (50%) of them are also requested in-context. Moreover, Google's guidelines are not adequately followed: (1) non-critical permissions are requested up-front, (2) no education is provided at all; while admittedly in many cases the reasons for requesting the permissions can be inferred from the functionality, some uses are less intuitive (such as displaying the callers for missed calls); (3) when permissions are not granted, feedback is not always provided and the functionality silently fails (*e.g.*, share with contacts).

Droid M+ 's Request Placement: Because Any.Do does not always request the permission in-context, we cannot do a one-to-one matching with its request placement; instead, we performed two measurements: (1) we checked whether their in-context requests matches Droid M+ 's and (2) we checked whether their in-context checks matches the requests made by Droid M+ .

For the three already in-context permission requests, because they are all inside click handlers, our placement is roughly the same as existing placement. There is a difference due to the fact that, to support ascynchronous permission requests, Any.do's code has already been refactored

so that the protected APIs are all accessed inside `void` methods and the permissions are asked before the invocation of the methods. This makes Droid M+ 's placement one level deeper than existing placement (permissions are asked inside the method). Unfortunately, since we do not have an older version of this app, we cannot verify if without such factoring, our algorithm would have yielded the same placement.

In the remaining places where permissions are checked but not requested, Droid M+ 's static analyzer identified all of those as places where permissions should be requested. From among them, 16 of the accesses are inside third-party libraries and 4 of them are inside background threads. We have also manually checked whether our request placements are in-context and the answer is yes.

Discussion: Although Any.Do is a single case study, we argue that the evaluation result is also applicable to other apps. More specifically, the quality of the request placement is assessed based on whether the request would be *in-context*. In Droid M+ we achieve this goal in two steps: (1) we segment the code into different functionalities based on unique entry points and (2) we try to place the request as close to the permission use as possible. We believe this strategy is effective for the following reasons. First, our functionality identification process essentially segments the app into Activities, UI events handlers, and background threads. Since most UI events handlers are single-purposed and very simple, permissions requests should also be in-context. Activities typically, at most represent a single context; thus, any request within an Activity is highly likely to be in-context. Furthermore, the quality of the placement inside a Activity is further improved in the second step (see above). For background threads, due to the limitation of the Android framework, the request would is less likely to be in-context; however, Droid M+ still improves the quality of the request by supporting the insertion of justifications.

3.5.4 Performance

In the next set of experiments, we seek to quantify the overheads that accompany Droid M+ on the developer side. Towards this, again consider Ringdroid, the open source app that was previously described and modify it to support revocable permissions. We make two constructs; in

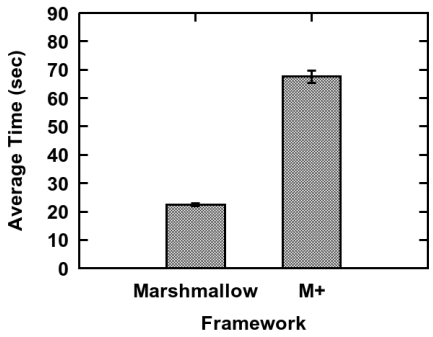


Figure 3.8: Average Compilation Time.

one the API of Android M is used, and in the other Droid M+ is used.

Compilation Overhead: Our case study was performed on Android Studio 1.4.1. The latter was running on a laptop with a quad core Intel Core i7 2.00GHz CPU with 16GB of RAM and a hard drive of 1TB at 5400 rpm. Because of the additional steps, the compilation with Droid M+ can be expected to take longer than with Android M's API. We seek to quantify this increase. We perform a clean before each compilation in order to achieve the maximum overhead that can be incurred. We perform 20 runs and both the average results and the 95% confidence intervals are shown in Figure 3.8. On average, Ringdroid using only Android M compiles in 22.43 seconds while using Droid M+ the times is 67.53. This corresponds to an increase of approximately 201%. Although this overhead is significant, it is experienced only when the app is compiled on the developer's computer (it does not affect the user).

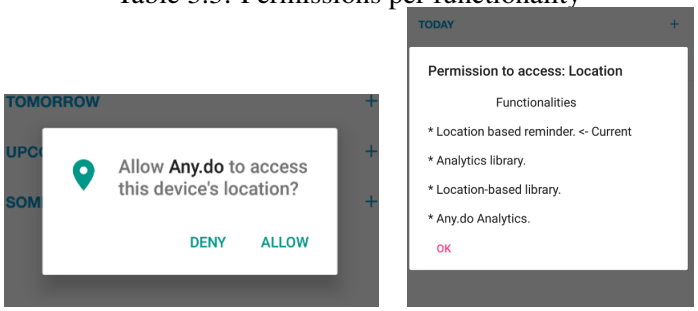
3.6 Discussion

Automated Annotation Extraction. Currently, Droid M+ cannot automatically generate the `functionality`, `reason`, and `feedback` within the annotations as it requires automated reasoning about the context. However, researchers have demonstrated that it is possible to use natural language processing (NLP) over apps' descriptions to infer their functionalities [142]. Using NLP, we may also be able to extract the context information directly from the target app. We plan to explore this direction in the future.

Per-Functionality Permission. Although the new Android permission model is a big

At most one	At least one	5 or less	5 or greater
49%	51%	60%	40%

Table 3.3: Permissions per functionality



(a) (b)

Figure 3.9: Any.do current version vs with Droid M+.

step forward from its old model, it is still not ideal. Specifically, once a user grants a permission, the permission will be shared across all the functionalities that will require this permission. For example, when a user grants the `Location` permission to a "map" app to find her current location, this permission can also be used by third party libraries [192] and violate the user's privacy. A recent study [187] shows that for the "ask-on-first-use" strategy Android M and iOS employ, the participants subsequent decisions on whether to grant a permission would match their first decision, only about half of the time (51.3 %). This observation is also intuitive; for example, in the above example, while the user is willing to grant the `Location` permission to the core functionality of the map app, she may not want the advertisement library to know her current location.

To quantify the extent to which permissions are carried over across functionalities, we analyzed the same 1638 apps that contain revocable permissions (recall Section 3.3). Table 3.3 shows the results. In particular, 51% of the apps share at least one permission across multiple functionalities. For these apps, in 60% of the cases, the permission is shared by up to 5 functionalities; in the remaining cases, the permission is shared between 5 and 20 functionalities.

We discuss two further improvements that can address this problem. Under the constraint of the current Android M permission model, one solution is to provide more education to the user with aggregated explanation messages from multiple functionalities. This solution will offer the

best transparency about all possible uses of any permissions. With Droid M+ , we have in fact implemented this solution and provided an example screenshot shown in Fig. 3.9. If a user chooses to approve the permission use, he or she will be fully aware that the permission is enabled in all functionalities. Of course, the downside of such an approach is that it burdens the user with too much information and puts the onus on her to revoke a permission later to protect her privacy.

A better solution is to extend the "ask-on-first-use" strategy to work on a triplet `<app, functionality, permission>` instead of the pair `<app, permission>`. This means that a permission is approved per functionality. If the same permission is used in several functionalities, a user can independently approve and deny a subset of the same. With the help of Droid M+ , an existing app can be easily ported to this new per-functionality permission model as Droid M+ can already help developers to identify and annotate the different functionalities of their apps, identify the required permission(s) for each functionality, and automatically generate the permission request code. To enforce this fine-grained permission model though, we can either insert additional code to maintain the per-functionality permission status, or require proper support from the Android OS.

3.7 Conclusions

Given criticisms on Android's permission models, Google revamped the model in Android 6.0. In this work, we find via an in depth measurement study that many apps from the Google Play store have either not migrated to the new model, or do not follow Google's guidelines for adopting the model, effectively. We find some evidence that this unsatisfying status quo could be due to the lack of tools that allow developers to easily adopt the new model. Towards addressing this shortfall, we design and implement Droid M+ , a tool set that helps developers refactor their code to adopt the model. We show that Droid M+ can help developers in evolving their legacy code to adhere to Google's guidelines, via case studies and general app evaluations. We also show that the overheads with Droid M+ are very reasonable.

Chapter 4

RootExplorer: Detecting Android Root Exploits by Learning from Root Providers

4.1 Introduction

Android is currently the most popular mobile operating system in the world, with 1.4 billion users worldwide and 87.5% of the market share [163]. Google, contrary to Apple (wrt iPhone), having no complete control over either the hardware or the software of Android phones. On the positive side, this allows many hardware and other third-party vendors to build a competitive, customized, and diverse ecosystem. But on the other hand, the diversity of Android devices also introduces security issues. First, the OS update process varies from vendor to vendor (some are faster than others). For example, at the time of writing, only 29.6% of Android devices on the market have Marshmallow [91], which was introduced nearly 2 years ago. Second, the vendor customization of Android often introduces vulnerabilities at different levels of the software stack including application, OS kernel, and drivers [20, 189, 197, 198, 201]. Consequently, millions of users are exposed to various critical security vulnerabilities that plague such customized, typically

older and unpatched devices [46, 56, 176].

Among all vulnerabilities, arguably the most pernicious are privilege escalation vulnerabilities that would allow attackers to obtain the root privilege – the highest privilege on Android. Such attacks are usually referred to as *root exploits*. Once it has acquired the root privilege, an attacker/malware can bypass the Android sandbox, perform many kinds of malicious activity, and even erase evidence of compromise. For this reason, malware with embedded root exploits are on the rise. Indeed, as apparent in recent news, it has become more and more common that malware found in third party Android markets or even in the official Google Play store, contain root exploits. For instance, in June 2016, Trend Micro reported GODLESS [132] , an Android malware family that uses multiple root exploits to target a variety of devices, affecting over 850,000 devices that were running Android 5.1 or earlier, worldwide. One month later (July), another Android malware dubbed HummingBad was reported to have infected more than 85 million devices and was found in 46 different applications, 20 of which were found on Google Play [100]. In September 2016, a Pokemon Go Guide app spotted in Google's Play Store, was found to contain root exploits as well [157]; the app had accumulated over 500,000 downloads by the time it was spotted and taken down. Considering that Google has already deployed a cloud-based app vetting service viz.,"Google Bouncer" [82], these repeated instances demonstrate that it is both important and challenging to detect malware that carry out root exploits.

An even more concerning fact is that the number of newly discovered privilege escalation vulnerabilities (*e.g.*, kernel vulnerabilities) is also on the rise [15]. Many of such vulnerabilities, such as DirtyCow [56], can even be used to root the latest versions of Android. So it is simply a matter of time before they are leveraged by malware to attack (potentially a large number of) unpatched devices.

In this project, we aim to tackle the challenging problem of detecting malware that employ a variety of root exploits. The key observation that drives our approach is that, in the Android world, it is not just the malware that carry root exploits. There are legitimate and popular Android applications, often called root providers or one-click root apps, that root phones on behalf of users [197]. Many of these apps are commercial-grade and backed by large companies such as Tencent, Qihoo,

and Baidu. They are capable of rooting tens of thousands of different Android devices using a hundred or more root exploits [197]. Note that rooting (as well as jailbreak) is considered legal [49], and users do want to root their phones to remove bloatware or unlock new features that were otherwise not available. These root exploits can serve as valuable resources towards aiding detection since they are highly customized (towards specific devices), reliable, and more importantly are likely to be used as is, by malware developers (discussed later). This means we can take advantage of these exploits to build a system (*RootExplorer*) that automatically extracts signatures from root exploits, and use those signatures for runtime malware detection.

Unfortunately, this seemingly simple strategy is not easy to realize in practice. The big obstacle is that almost all exploits are tailored towards specific Android devices, models, and/or OS versions. Screening apps in an emulator is unlikely to trigger and reveal the exploit, unless the environment matches exactly what the exploit expects. This in turn means that one may need tens of thousands of real Android devices to cover just all *known* root exploits. To overcome this obstacle, *RootExplorer* also learns the environment requirements from the aforementioned commercial root exploits and uses this knowledge to create the "expected" runtime environment so that it is capable of interacting with the exploits to drive their execution (*e.g.*by pretending that a particular vulnerable device exists).

We design, prototype and extensively evaluate *RootExplorer* to detect root exploits present in malware. It consists of (a) an offline training phase where it extracts useful information about root exploits from one-click root apps using behavior analysis, and (b) an online detection phase where it dynamically analyzes apps in specially tailored environments to detect root exploits. We test our prototype with a large set of benign apps, known malware, and apps from third-party app marketplaces. Our evaluations show that *RootExplorer* yields an almost perfect true positive rate with no false positives. *RootExplorer* also found an app that is currently available on the markets, that contains root exploits.

In summary, the contributions are as follows:

- We identify and address the fundamental challenge of detecting Android root exploits that target a

diverse set of Android devices. In particular, we learn from commercial one-click root apps which have done the "homework" for us with regards to (a) what environmental features are sought and (b) what pre-conditions need to be met, for a root exploit to be triggered.

- We design and implement *RootExplorer*, a fully automated system that uses the learning from commercial one-click root apps to detect malware carrying root exploits. Specifically, in an offline phase, it conducts extensive static analysis to understand the precise environment requirements and the attack profile of the exploits. It then utilizes the learned information to construct proper analysis environments and detects attempted exploits.
- We evaluate *RootExplorer* via extensive experiments and find that it can successfully detect all known malware that contain root exploits, including very recently discovered exploits and the ones that are used in other one-click root apps; *RootExplorer* results in no false positives with our test set. Using *RootExplorer*, we also find an app which is currently available on an Android market, that contains root exploits.

4.2 Background & Related Work

4.2.1 Root Exploits and One-Click Root Apps

As mentioned, one-click root apps are very popular among users and they are competing against each other to be able to root more phones and offer more reliable results. One of the reasons that companies develop these apps is that they also develop security apps or app management tools that also require the root privilege to function correctly (*e.g.*antivirus software must have higher privileges than any malware [33]);

Interestingly, the competition between these one-click root apps have driven them to include the most comprehensive and advanced root exploits. For example, in 2015, it was reported that there are 39 families of directly usable root exploits that can be found publicly (with source code or binaries); in contrast, there were 59 families of root exploits found in a popular commercial one-click root apps, including exploits against publicly unknown or zero-day vulnerabilities [197],

and exploits that can bypass advanced defense mechanisms like SELinux [92], Verified Boot [93], etc. On the contrary, although researchers have detected several malware families with root exploits, none of them contain previously unknown exploits [202]. We believe this is because most malware authors, except the so-called state sponsored, do not have the capability to develop new root exploits; hence, they typically only embed exploits that are developed by others (*e.g.*one-click root apps).

While detecting malicious behaviors has been the focus of many prior efforts in the literature, detecting Android root exploits faces unique challenges. One such challenge is that specific preconditions (*e.g.*environment constraints) need to be satisfied in order for such exploits to be triggered; this is hard becuase of the highly fragmented Android hardware and software. Specifically, not only do different phones have different device(s) and corresponding driver(s), even with respect to a universal kernel vulnerability such as the futex bug [72], the root exploit has to be tailored for different phones. This is because the actual kernels on different phones are different (*e.g.*each has a different memory layout). As a matter of fact, one commercial one-click root app contains 89 different exploit payloads for the same underlying futex bug [197]. Consequently, malware carrying root exploits typically have specific environment checks to determine (1) what kinds of vulnerabilities are available and (2) how the attack should be launched. Thus, in order to detect a root exploit, an analysis environment must satisfy the necessary preconditions.

We categorize these preconditions into two corresponding types: (1) *environment checks* and (2) *preparation checks*. Environment checks gather information with regards to the environment such as the device type, model, and operating system versions. For instance, many times a particular malware will check whether it has a matching exploit for the current environment. If so, the specific exploit is selected from either a set of local exploits or a remote exploit database. This process is in fact also used by one-click root apps [197]. Preparation checks verify that the interactions with the underlying operating system are as expected, (*e.g.*a vulnerable device file exists on the system and the driver returns expected results in response to specific commands). The number of preparation checks can be large, depending on the nature and complexity of the root exploits. This makes it difficult to manually prepare the right environment for each root exploit and detect them.

4.2.2 Android Malware Analysis

A relatively large chunk of Android related literature, is on malware analysis and malicious behavior detection. However, most of this literature focuses on detecting malicious behaviors like leaking/stealing private information and financial charging [202]. Unfortunately, no existing work tackles root exploit detection. We roughly categorize such work into three types: static analysis, dynamic analysis, and hybrid analysis.

Static Analysis: Static analysis is used to analyze an Android app's byte code and/or native code without running it inside an emulator or a real device. To detect information/privilege leaks, a set of tools [27, 44, 94, 120, 127, 140, 182] has been developed to perform information-flow analysis. Another popular direction is to model and detect malicious behaviors that are unique to Android. Pegasus [47] uses "Permission Event Graphs" to detect sensitive operations performed without the user's consent. Apposcopy [67] uses "Inter-Component Call Graphs" to detect Android malware. AppContext [195] uses contextual information (UI events and environmental triggers) to check access to sensitive operations. The advantage of static analysis is coverage and efficiency; it may however face problems when analyzing apps with heavy obfuscation. In fact, it has been shown that simple obfuscation techniques or transformations applied to known malware samples can often easily evade static detection by anti-virus software [153].

Dynamic Analysis: Dynamic analysis analyzes an Android app by running it inside an emulator or a real device. Similar to static analysis, many dynamic malware analysis systems also focus on information flow analysis and leak detection [62, 148, 199]. Others use system calls to model and detect malicious behaviors [43, 54, 168, 194]. Because malware can detect that it is being run in an instrumented environment such as an Android emulator [107, 145, 177], researchers have also proposed building sandboxes on real devices [32, 37] for this purpose. Dynamic analysis can usually overcome obfuscation techniques employed by malware, but a malicious behavior can only be detected if it is executed during the analysis. To overcome this, tools have been developed to systematically exercise the functionality of an app in the hope of triggering its malicious behaviors [31, 188].

Hybrid Analysis: Hybrid analysis can be divided into two cagegories. The first category combines static and dynamic characteristics to detect malicious behaviors [184, 191, 203]. The other category utilizes static analysis to guide dynamic analysis [31, 188, 196, 200].

4.2.3 Attack Modeling and Detection

Previous papers on attack modeling and detection mainly focus on filtering remote exploits like those launched by worms [50–52, 114, 123, 136, 161, 179]. Similar to those systems, *RootExplorer* also leverages program analysis techniques like symbolic execution to extract the attack signature. However, there are a few differences. First, due to fragmentation of the Android ecosystem, we do not always have the targeted device, *i.e.*we need to derive both the attack signature and the corresponding environment requirements *without* the corresponding target system. Note that in aforementioned systems, in contrast, analysis is usually performed over the targeted software. Second, for remote attacks, the malicious payload usually contains shellcode; however, in local privilege escalation attacks, shellcode is rarely used – `ret2usr`, `ret2dir`, or direct kernel object modification (DKOM) are more common. Finally, due to polymorphic or metamorphic payloads, finding a good balance between false negatives and false positives is very challenging for network filters. Android root exploits are more difficult to morph (as shellcode is not part of the payload); more importantly, even though it is possible to generate polymorphic exploits, as previously discussed, most Android malware authors are not capable of doing so. For these reason, we decide to pursue our current approach, *i.e.*, derive system-call-based signatures purely from known exploits.

4.2.4 Other Related Work

Android Emulator Evasion: Recent works have shown how easy it is for malware authors to evade the Android emulator. Petsas *et al*. [145] apply three different detection heuristics and manage to detect most Android dynamic analysis tools. Vidas *et al*. [177] derive four different techniques based on differences in behavior, performance, hardware and software components and show how

they can easily detect existing malware scanner tools that are based in emulators. Morpheus [107] is a system that can create up to 10,000 different detection heuristics for Android emulators. As a countermeasure, researchers [133] have begun to use real phones instead of emulators to analyze malware. We design our solution to be operable on both real Android devices and emulators, thereby making this issue orthogonal to our work.

Syscall-based Behavior Modeling: *RootExplorer* uses system-call-based behaviors to model and detect root exploit attempts. Syscall-based behavior modeling has been widely used to model and detect malicious behaviors [35, 117]. Our model is derived from the behavior graph proposed in [117], with adjustments to fit our scenario.

4.3 Threat Model and Problem Scope

The goal of *RootExplorer* is to detect Android apps that carry *root exploits*. Detecting other malicious behaviors is out-of-scope of this work and has been covered by many previous papers (§4.2). We also do not attempt to understand what the malware will do after acquiring the root privilege; we defer such an analysis to future work.

We envision our system to operate in the cloud (similar to Google Bouncer [82]), and that it will scan apps by dynamically executing the samples on real Android devices and/or emulators. For this reason, we restrict the source of the analyzed apps to be either from the official Google Play Store or from third-party marketplaces. We do not consider malware involved in targeted attacks such as APTs.

We assume that malware carrying root exploits can be obfuscated to prevent static analysis, and may be equipped with common anti-debugging/anti-virtualization techniques to detect the analysis environment. They may also download root exploits dynamically from a C&C server only when the desired Android device is detected. For triggering root exploits, we focus on understanding and providing the environment expectations. However, we do not handle malware that depends on specific user inputs (*e.g.*passing a game level) to trigger the root exploit. We believe generating such inputs is orthogonal to this work and has been covered by other projects [31, 188].

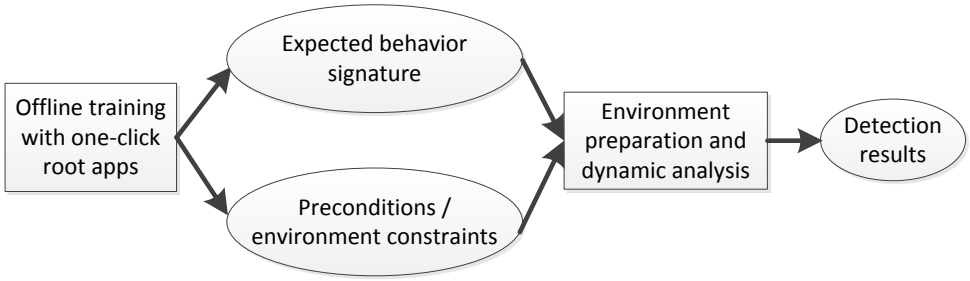


Figure 4.1: System overview

Finally, we focus on detecting root exploits against known vulnerabilities; detecting unknown or zero-day exploits is out of scope of this work. We believe this is a reasonable limitation as no malware that has propagated through app marketplace has been found to contain zero-day exploits.

4.4 *RootExplorer* Overview

Figure 4.1 depicts the operations of *RootExplorer*. There are two key phases: (1) an offline training phase (static analysis) that extracts useful information about root exploits from one-click root apps and, (2) a detection phase (dynamic analysis) that dynamically analyzes apps in specially tailored environments to detect root exploits.

During training, we gather information about as many different root exploits as possible. Since root exploits target *specific* devices, it is not possible to trigger all of their behaviors without proper environments. We thus resort to static analysis. For each exploit, we collect (1) sequence and dependencies of system calls that can lead to a compromise of the device, *i.e.*behavior signature [35, 117], and (2) preconditions for *deterministically* triggering the exploit.

The first step of our offline analysis is to identify a feasible execution path that leads to the success of the analyzed root exploit. We use guided symbolic execution to solve this problem. In particular, we symbolize all external "inputs" to each root exploit and aim to find a shortest feasible path from the entry to the marked successful end point. We build our prototype symbolic execution engine based on IDA pro, which is capable of handling all the instructions and libc functions that

were encountered in the training set of exploit binaries.

From the feasible execution path, we extract the sequence of system calls and the dependencies across system calls from the output of symbolic execution as well. This information is then used to construct the behavior signature. Since we already collect constraints over what information needs to be returned from the system through system calls (*i.e.*preconditions) during symbolic execution, we just consult an SMT solver to provide a concrete instance of satisfying preconditions. Both pieces of information (behavior signature and preconditions) feed directly to the dynamic analysis phase towards preparing the right environment and satisfying necessary preconditions, to trigger and thereby detect various root exploits.

For this purpose, besides utilizing root exploits from one-click root apps, we could in theory utilize the many exploits with PoC code available on the Internet, but they all come in different "sizes and shapes". Some contain source code but often hard code values in certain variables; this renders the exploit suitable only for a specific tested Android device. Some have binaries only, which are obfuscated to prevent direct reuse. Therefore, We choose to work with a popular one-click root app for the purposes of training. The benefits are multi-fold: (1) the quality of exploits is likely very good, as they are offered in commercial products (*e.g.*they don't contain unnecessary steps, and are unlikely to crash the system); (2) there is a rich variety of exploits available (60 families of exploits in our evaluation); (3) the exploits packaged in the same one-click root app are likely to be obfuscated in similar ways, making it possible to de-obfuscate all exploits at once and conduct static analysis on them.

Learning the expected behavior signature: The behavior signature of an exploit is extracted by analyzing the de-obfuscated exploit binaries. While there are many possible models to construct malware signatures in general, we favor system call based behavior signatures; this is because root exploits interact with the operating system through system calls in unique ways to exploit vulnerabilities. To this end, we build our behavior signature largely based on prior work on extracting a malware behavior signature from system calls [35, 117]. This allows our dynamic analysis to keep track of the progress of an exploit and confirm it when all of its steps have been performed. More details are provided in §4.5.

Learning preconditions: As discussed earlier in §4.2, there are two types of preconditions that have to be satisfied with regards to a root exploit in general: environment related and exploit preparation related. Environment preconditions dictate whether the underlying Android device model and kernel version match what are expected by the exploit. After training, our dynamic analysis environment can provide the expected Android device information to trigger an exploit. Normally it is difficult to determine which exploits work against which Android devices (because one needs to ideally test an exploit against real devices). Fortunately, one-click root apps already provide this information to a large degree. Specifically, the one-click root app we studied downloads a different set of exploit binaries depending on the device information that is reported to its backend server. By reverse engineering their protocol, we have effectively built a mapping from a list of more than 20K Android device types (available from [2]) to their corresponding exploits. The assumption is that a one-click root app has a reasonably good idea of which exploits can target which device.

For exploit preparation related preconditions, we give the symbolic constraints collected along the feasible path and ask the SMT solver to construct a concrete satisfying instance such that when replayed during dynamic analysis, can deterministically trigger the analyzed root exploit. For instance, if an exploit expects to open a vulnerable device file successfully, the "input" to the exploit program is the return value of the `open()` syscall, which needs to take a non-negative value according to the symbolic execution. Once we learned such preconditions, our dynamic analysis environment can provide the same expected "input". We will present the detailed design of the symbolic execution in §4.6.

4.5 Behavior Graph Analysis

Since Android malware (especially those that contain root exploits) typically obfuscate their payloads heavily [202], dynamic analysis is the obvious choice over static analysis, for the purposes of detection. However, as discussed earlier, dynamic analysis wrt root exploits is difficult as such exploits target *specific* Android devices. Without the right environment, such exploits are likely to terminate prematurely, thereby preempting detection.

To overcome this hurdle, we leverage de-obfuscated binaries from a one-click root app to extract the behavior signatures of root exploits. A behavior signature is constructed by abstracting the low-level operations in to a high-level behavioral representation [35, 117]. One can check for malware samples that exhibit similar behaviors at runtime and thereby detect the presence of the particular exploits. In the case of root exploits, since they interact with the kernel (or device drivers) in unique ways to exploit an OS vulnerability, we choose to capture behaviors by modeling system call events. Instead of reinventing the wheel, we borrow the system call modeling technique from ANUBIS [117] with slight adjustments. Specifically, we follow the definition of "behavior graphs" [117] that are used to describe OS objects, system calls that operate on these objects and, relationships across system call events (*e.g.*the result of one system call is used as a parameter on another system call).

The behavior graphs are directed acyclic graphs where the nodes represent objects and system calls, and the edges encode (1) the dependencies between objects and system calls, and (2) the dependencies across system calls. Compared to the traditional model of simply looking at a sequence of system calls [98], a behavior graph constrains the order of only dependent operations through an explicit edge (and never constrains independent operations).

While the high-level behavior graph is similar to that proposed in [117], we highlight the main differences here: (1) We statically extract the behavior graph instead of extracting it from a dynamic trace (as is done in ANUBIS). This leads to different requirements as elaborated later. (2) Since we target Android, the system calls are mostly inherited from Linux and are different from Windows.

4.5.1 Generating Training Behavior Graphs

We now describe how we automatically generate the behavior graph statically, by analyzing de-obfuscated ARM root exploit binaries. The system call invocations, and their hard-coded arguments are generally easy to identify. This allows us to know what OS objects are created (*e.g.*a file name), and how they are operated on (*e.g.*Read-only or Read/Write). The main challenge that we face is to extract the dependencies across system calls.

Extracting data dependencies: To extract dependencies across system calls, we look for cases where the arguments for one system call is derived from a previous system call. Previous work [117] utilized taint analysis to derive such dependencies. In our system, since we perform static analysis over de-obfuscated binaries, we take a slight different approach. Specifically, when we use symbolic execution to find a feasible success path, we symbolize all the outputs of system calls. During the analysis, symbolic values are propagated along the execution path. To determine whether a path is feasible, whenever we meet a conditional branch that depends on symbolic value, we consult the solver to see if the corresponding path constraints are solvable. If we consider a symbolic value as tainted, then symbolic execution itself, already constructs the data dependencies between system calls, *i.e.*, if the input argument(s) of a system call is a symbolic value, then it must have a data dependency over one or more previous system calls. More importantly, the symbolic formulas of such input arguments also specify *how* they are depend on each other. Based on this observation, we extract the data dependencies between system calls by simply naming the symbolic values returned by system calls according to the system call names and their sequence in the feasible path (*e.g.*`read2_buf`).

Extracting control dependencies: Symbolic execution does not directly provide control dependencies. To extract such information, we simply conduct a backward analysis. In particular, when outputting the feasible path discovered via symbolic execution, we also mark each control point that *directly* depends on the symbolic value with the system calls that introduced that value. Using the path, we start from the end point and traverse the trace backwards to look for system call invocations (e.g., `BL mmap`). Once we find a system call invocation, we can extract its control dependencies over previous system calls by searching for the closest "tainted" branch that precedes this syscall invocation. Alternatively, we could have used static binary taint analysis to extract both data and control dependencies.

Modeling of libc functions: The exploit binaries in our training set do not generally call the system calls directly (as typical with most native code). Instead, they call the libc functions (in Android, it is called Bionic). Fortunately, most are simply wrappers of system calls and have the same exact semantics. In cases they are not exactly the same, for example, `fopen()` vs. `open()`,

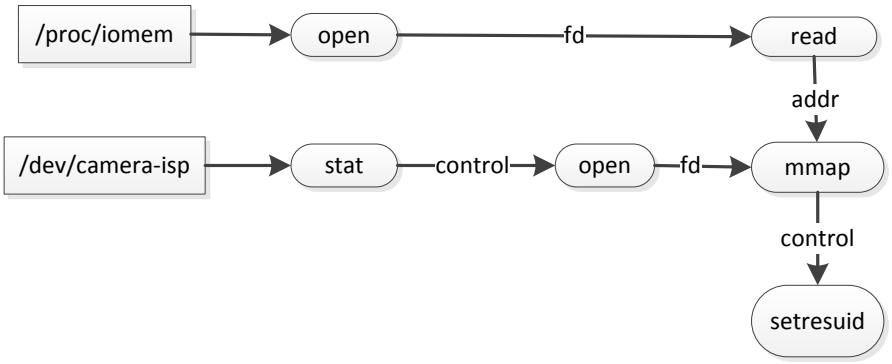


Figure 4.2: Behavior graph for the "camera-isp" exploit.

we model the Bionic version `fopen()` by mapping its arguments and return values to `open()`. Furthermore, we leverage function summaries to model most encountered libc functions that need to be analyzed by symbolic execution.

4.5.2 Examples

Device Driver Exploit: To illustrate our behavior graph analysis, we consider a popular device driver exploit that targets the vulnerable Qualcomm camera driver, "camera-isp". This example is taken directly from our training data set from a popular one click root app. In brief, the vulnerable device driver allows any program to map any part of the physical address space into the user space, which can subsequently allow the disabling of the permission check in `setresuid()` system call. This allows an attacker to change the running process into a root process.

Figure 4.2 represents the behavior graph. The exploit needs to open two separate files, the vulnerable device file `/dev/camera-isp` and the helper file `/proc/iomem` which has the information about where the kernel code is located in the physical address space. Both files are checked with the `open()` system call to ensure that they can be successfully opened. The device file is checked in addition in the begining, via a `stat()` system call, for existence. The exploit then attempts to `mmap()` the kernel code region into the user address space with read/write permissions; however, the exact offset (argument in `mmap()`) is retrieved from the read result of the `/proc/iomem`. After the `mmap()` is successful, the exploit searches for a particular sequence of bytes in the mapped memory

that corresponds to the code blocks for `setresuid()`. Upon locating the code block, it patches the code block by writing to a specific offset, which effectively eliminates the security check in `setresuid()` (the above two steps are invisible in the behavior graph). Then the exploit simply calls `setresuid(0,0,0)` to change the uid of itself to root. Finally, as mentioned earlier, all exploit binaries in our training set, end the execution with a check through `getuid()` to verify that the exploit process has obtained root.

Note that due to space constraints, we do not annotate the graph with the exact arguments (*e.g.*file open with a read/write permission or read-only). We also do not label whether the system call succeeded or not. In most exploits, all system calls need to be successful in order to compromise the system and typically the failure of a system call will immediately result in an abort.

Kernel Exploit: As a second example, we consider Pingpong root [193], one of the most recent generic root exploits that can tartget almost all Android devices released prior to mid-2015. The case also reflects one where the exploit creates multiple processes. In particular, the key exploit logic [78] is conducted in the main process, including `mmap()` at a specific address, and invoking multiple `connect()` calls on the same ICMP socket (we omit the complete behavior graph for brevity). In addition, One or more child processes are created as helpers to construct as many ICMP sockets as possible for padding. Since the `fork()` occurs in a loop (up to 1024 iterations), it is necessary for symbolic execution to identify and choose one feaisble path. Specifically, the analysis output is that as long as the loop is executed once, a feasible exploit path can be constructed. This means that we can simply unroll the loop once and have a new behavior graph constructed for the child process (which is connected to the parent behavior graph via a `fork()` edge). Note that unrolling the fork loop more times is also feasible which will cause identical behavior graphs to be constructed. In this case, all behavior graphs will need to be matched so that we can claim an exploit is detected. It is worthwhile mentioning that the precondition analysis (which will be described in more detail in the next section) is conducted jointly, and will ensure that the first `fork()` will succeed at runtime, thus causing the exploit to match the behavior graph with one child process only.

4.5.3 Using Behavior Graphs in Detection

Once the behavior graphs for different root exploits are generated offline, we are able to use them for detection in a scanner (similar to Google Bouncer). More precisely, by monitoring system call invocations (and arguments), our dynamic analysis environment determines if the behavior of the program under analysis matches any of the learned behavior graphs. The matching algorithm is similar to that in [117]. We only briefly describe the procedure below and the design decisions that were made.

To find a match in the behavior graph, it is necessary to ensure the following: (1) The order of system calls conforms to the dependencies represented in the learned behavior graph. In addition, the dependencies in the behavior graph need to be maintained at runtime as well. This can be checked using dynamic taint analysis [35, 117]. (2) The exact values of the arguments for system calls match (*e.g.*a file opened with read/write permission). For those arguments whose values cannot be determined statically during training, they will simply be considered as wildcard values that can match any value at runtime. (3) A system call's status (either success or failure) matches with the one in the learned behavior graph.

We observe that the root exploits typically have unique inputs to the system via system call arguments, which makes them easy to distinguish from legitimate programs. We therefore relax requirement (1) by only verifying simple dependencies at runtime (*e.g.*a file `read()` depends on the output of `open()`). Such cases can be checked through the OS objects monitored in the kernel, without conducting an expensive taint analysis. For more complex dependencies such as the values obtained through `read()` affecting a system call `mmap()` as shown in Figure 4.2, we only require that the order is the same as constrained on the graph, *i.e.*`read()` happens before `mmap()`. We plan to implement the dynamic taint analysis for stricter dependency enforcement in future work. Alternatively to improve efficiency, we could also apply the optimization proposed by Kolbitsch *et al.* [117].

4.6 Satisfying Exploit Preconditions

It is crucial to build an environment that can satisfy the preconditions expected by root exploits. More importantly, because our behavior graph is constructed over one successful path, if an analyzed app contains root exploits, our dynamic analysis environment must deterministically coerce the app to follow that path, *i.e.*, the app must be made to reveal the same set of malicious behaviors that match the learned signature. This means that whenever the exploit asks the environment for certain results, we must return them as expected.

The problem naturally maps on to the common debugging and testing problem of generating the proper inputs to a program, so that it will reach a particular target statement [50, 55, 128]. Here the target statement is the end point of the root exploit, *e.g.*, the `getuid()` call. And the "inputs" are the system call results, including (1) system call return values and (2) other return results through arguments (*e.g.*, a buffer filled in `read()`). Our solution to this problem is symbolic execution. That is, we symbolize all the "inputs" from system calls and leverage symbolic execution to find the shortest feasible path that can reach the target instruction from the entry point. Once we found such a path, we then ask the SMT solver to generate a concrete instance of the inputs which will be "replayed" during dynamic analysis.

With respect to the system call return values, we consider two types of system calls: (1) Those that return a reference to kernel object, *e.g.*, `open()` and `socket()` returns a file descriptor; and `mmap()` returns the address of the "memory-mapping object". (2) The remaining one (*e.g.*`stat()`) that return either 0 (indicating success) or error code. For type (1), since file descriptors and mapped addresses are determined dynamically by the OS and the constraints are typically simple (just `!= 0`), we symbolize their return values as a boolean during analysis and do not force a specific value during runtime. Instead, we simply choose to force a successful or failure based on the boolean and let the OS to assign the concrete return value. And to allow expected interactions with the corresponding kernel objects, We use "decoy objects" (explained later) instead of tracking those references. For type (2), we just symbolize their returned value normally as bit-vectors and ask the solver to generate a satisfying value.

For system calls that return results through arguments, they are always pointers passed in user programs (*e.g.*read buffer). We use these input pointers to symbolize the corresponding memory content. Going back to the first example exploit in §4.5.2, after reading from the file `/proc/iomem`, the exploit attempts to read the starting physical address of the kernel code. This procedure is illustrated in Figure 4.3. As we can see, the exploit reads the file line by line to look for the constant string "Kernel code". Once the line is located, it retrieves the kernel code base address (through the `getAddress()` call) at the -20 offset relative to the returned buffer of `strstr()`. There are effectively two loops in the program. The first is the `while` loop; the second is inside `strstr()`. In this particular case, the discovered feasible path says that the `while` loop can iterate just once, indicating that we can return the string containing "Kernel code" when the first line is retrieved using the `read()` system call. However, the feasible path also says that the loop in `strstr()` needs to iterate at least 20 times[1]; in other words, "Kernel code" needs to start at line[20]. This is because the `getAddress()` call reads the location at `buf-20`. If `buf` is at the beginning of line, then `buf-20` would be reading something out of bound.

In this case, the address returned from `getAddress()` is not further constrained later, which means that `line[0]` to `line[19]` are unconstrained and can take any value. Therefore, the constraint solver will generate an output for `line` with something like "abcdefghijklmnopqrstKernel code". Further, since the `read()` system call only reads one line, we will place the single line content into the expected file object. There is a similar case later on involving a search through the memory for constants after `mmap()`, which can be resolved similarly.

Decoy Objects: During dynamic analysis, we can provide the preconditions we learned by forcing/faking all syscall results. However, to improve the robustness of our environment (*i.e.*, making it more real), we decided to use decoy objects to provide expected results for operations over certain type of kernel objects. Doing so would allow us to "tolerate" certain operations (*e.g.*, `stats()` that are not observed during our offline learning phase.

Currently we only support three types of decoy objects: files, socket, and device drivers.

[1] In our real implementation, we use function summary to handle all encountered external library calls.

```
fdIo = open("/proc/iomem");
// locate the kernel code offset in physical memory
while ((line = readline(fdIo)) > 0} {
   if((buf = strstr(line, "Kernel code")) != NULL) {
      addr = getAddress(buf);
      break;
   }
}

int getAddress(buf) {
   return atoi[buf-20];
}
```

Figure 4.3: Pseudo code of `proc/iomem` read

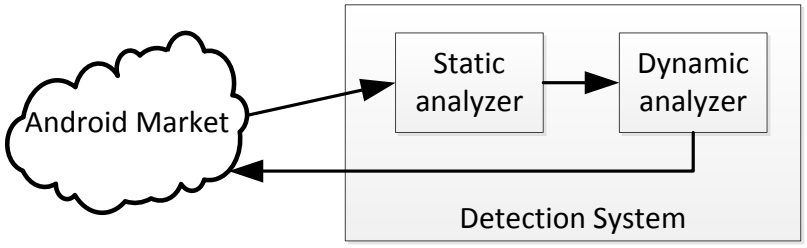


Figure 4.4: Operational model of the detection system

And they are created in two ways. If the target objects such as a vulnerable device driver do not exist in our analysis environment, we simply create a decoy one. If the objects such as `/proc/iomem` already exist in our environment, instead of opening the real file, we "redirect" the file open operation to an alternative decoy object as well, so that we can return the expected content.

4.7 Detecting Root Exploits

Thus far, we have described the *training phase*, where we generate both the behavior graph and the environment constraints. In this section, we provide details about the components of our detection system (*testing phase*). We first present an overview of our system's operational model and then describe its components in detail.

4.7.1 Operational Model

As mentioned earlier, we envision *RootExplorer* to be used as an app vetting tool for Android markets. When a developer submits an app to the market via a web service, we envision the market pushing it to *RootExplorer*, as depicted in Figure 4.4. First, we employ a static analyzer (different from the static analysis during the training phase), which performs several checks to filter apps that are unlikely to contain root exploits (details later). Subsequently, it determines "with which kind of mobile device(s) or emulator(s)," the dynamic analysis will be performed. Upon completion of the dynamic analysis, the detector collects the results and determines if the app contains a root exploit and if so what exploit it is. If the app does have root exploits, it informs the Android market and saves the hash of the app to the central database; otherwise the app is moved either to a different malware scanner (*e.g.*Bouncer) that is orthogonal to our system or for publication in the Android market. The dynamic analyzer can be run on either real phones or Android emulators (or a mix of both), and can be easily integrated into various malware analysis environments as needed.

4.7.2 Static Analyzer

The static analyzer consists of three components as shown in Figure 4.5. The first component is the native code detector. Since almost all root exploits are written in native code (certainly the case for the one-click root app we study), it is natural to check whether the apps contain native code. Specifically, the native code detector does the following checks to filter apps that are extremely unlikely to contain root exploits: (1) Whether the app matches signatures of known malware samples that contain root exploits. If so, we abort any further analysis and flag the malware. (2) Whether it has any native code or has the capability of dynamically loading native code (*e.g.*through the network). If negative, we can safely skip the analysis of this app. (3) If it contains native code, similar to prior work [14], we use a list of custom heuristics to decide if they can *possibly* contain root exploits (*e.g.*whether any dangerous system calls are being called). If negative, we do not analyze the app further.

If the native code detector did not abort the analysis, the app is moved to the device detec-

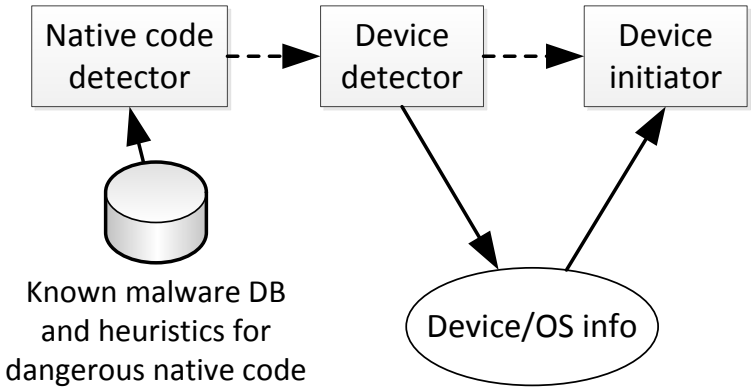


Figure 4.5: Static analyzer

tor. This is responsible for determining "under which environment the app should be dynamically analyzed." The observation is that since malware can embed different exploits targeting different Android devices, they usually contain logic that detects the type of the Android environment. Thus, we look for any such logic that performs checks against hard-coded device types.

The last component is the device initiator, which generates the Android environment based on the output of the device detector. We describe the device detector and the device initiator in more details below.

Device detector: This component parses the decompiled bytecode (using androguard [23]) and finds the methods (A) that contain code that check either the Android version that resides in the static class android.os.Build.VERSION, or the type of the device that resides in the android.os.Build class, or the version of the Linux Kernel (*e.g.*by `Runtime.getRuntime().exec(‘‘uname’’)` and reading the /proc/version file). Furthermore, it finds the methods (B) that either run an executable native file (*e.g.*`Runtime.getRuntime().exec(‘‘/sdcard/foo’’)`) or call a function in a native binary (*e.g.*library files). If there is a program path from the methods that are members of (A) to the ones in (B), it finds which conditions should be satisfied and creates the appropriate Android environment. Similarly, the same procedure is performed in native code. In the case that the native code is obfuscated or even downloaded via a C&C server, the device detector simply picks a few popular candidate device types randomly, with the view that the malware will likely target one or

more popular devices.

Device initiator: Android stores the device information in system files such as `/system/build.prop` and `default.prop`. `/system/build.prop` contains specific information about the mobile device such as the Android OS version, the name and the manufacturer of the device. In addition, there are also system files such as `/proc/version` and `/proc/sys/kernel/*` inherited from Linux that store information about the Linux kernel. When the system boots, the Android's property system reads the information from these files and caches them for future use. The device initiator monitors all such interfaces via which an app can learn about the device and obtain OS information. Since we have collected a database of Android devices from the online repository [2], we know what values to modify in the system files or what to return through the proc interfaces.

4.7.3 Dynamic Analyzer

The dynamic analyzer consists of two parts as illurstrated in Figure 4.6 viz., a Loadable Kernel Module (LKM) and a background service process. The LKM hooks every available system call in the Android Linux Kernel. In addition, it creates a character device that can be opened by only the background service (to prevent malware from tampering with the communication), and with which a communication link is established between *user-land* and *kernel-land*. The LKM tracks only a specific app (under analysis) and its child processes at any moment in time. The background service stores the training models including behavior graphs and environment constraints, as well as the state of the current running app (*e.g.*what part of a behavior graph has been matched and what environment constraints are supposed to be returned next).

Once a hooked system call is called by the app under analysis, the execution is directed to our LKM which then transmits a specially crafted message that contains the system call names as well as their arguments to the background service through the character device. Based on the behavior graph and environment constraints, the background service is responsible for deciding what action is going to be taken, and it returns that action to the LKM. First, it checks the behavior

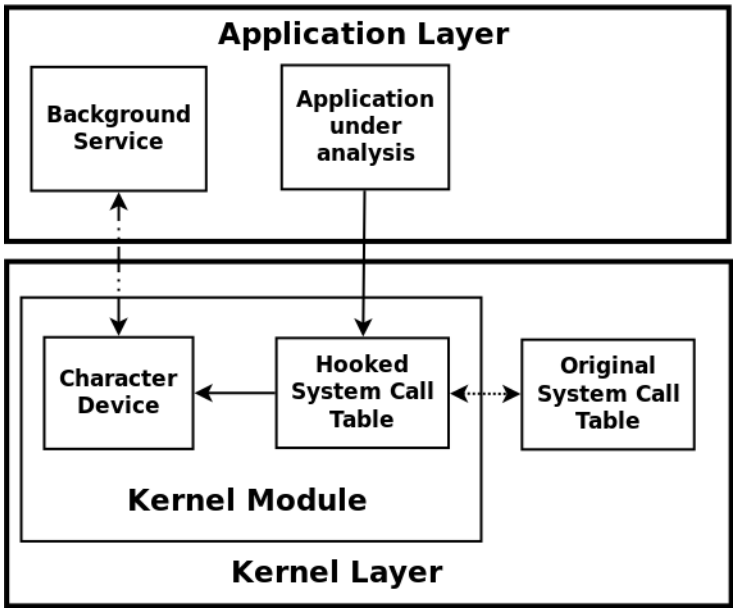


Figure 4.6: Dynamic Analyzer.

graph to see if the system call in question matches any new node. If not, it does nothing and simply instructs the LKM to execute the system call as is. If a new node is matched, it further checks if it is an object creation system call such as `open()` or `socket()`. If so, it deploys a decoy object to satisfy the environment constraints as described in §4.6. Otherwise, if it is a system call that operates on an existing object, the return results will be served from the data prepared ahead of time for the decoy object (*e.g.*file content).

Note that deploying decoy objects has to be done carefully. As mentioned, Android root exploits often need to be adapted to work on different devices, even when they target the same underlying vulnerability. For instance, the device file `/dev/camera-isp` can be exploited slightly differently on different Android phones that all have the vulnerable device driver; this will cause slightly different behavior graphs and preconditions to be generated, *e.g.*the input to a vulnerable device file will look different, and the expected return results from a system call may be different. Therefore, once we have decided to disguise as a particular Android device (*e.g.*Samsung Galaxy S3), we will need to choose the behavior graphs and preconditions accordingly (obtaining such a Android device to exploit binary mapping is discussed in §4.4). Otherwise, the decoy objects we deploy may be for the wrong Android device which will in fact fail to detect the exploit.

4.8 Evaluation

In this section we describe the evaluations of *RootExplorer*. We focus on its effectiveness wrt the following aspects: (1) can it detect synthetic and real malware containing root exploits? (2) does it cause false alarms on benign apps? (3) does it miss malware samples?

4.8.1 Environment Setup

Training parameters: Our training database contains 168 different root exploits that were designed for different devices and were obtained from a commercial one click root app. The number of devices that we can successfully emulate currently based on the root exploit database is 211. We trained with all 60 families of root exploits.

Testing dataset: We have the following categories of apps for evaluation:

1. *Samples that are known to contain root exploits.* This includes publicly distributed exploit PoCs [76, 77, 79, 81] and GODLESS malware [75], and seven other one-click root apps (different from the one we trained with) which also contain many different root exploits.

2. *Samples that* may *contain root exploits.* We obtained a list of 1497 malware samples from an antivirus company, and have crawled 2000 recently uploaded apps between January and February 2017, from four unofficial Android app markets: 7723 [16], ANDROID life [17], Mobo-Market [19] and EOEMARKET [18]. We target third-party markets because they are known to have more malware than the official Google Play [203].

3. *Samples that almost certainly* do not *contain root exploits.* This includes the top 1000 free apps from Google play. As they are extremely popular, it is very unlikely that they contain root exploits. This set is used to evaluate the false positives (if any) with *RootExplorer*.

Analysis Testbed: The experiments are performed on a Lenovo Laptop with a quad core Intel Core i7 2.00GHz CPU, 16GB of RAM, and a hard drive of 1TB at 5400 rpm. We use an Android emulator for analyzing the malware[2]. To make the emulator appear as realistic as possible, it is loaded with real files, such as music, pictures and videos. Furthermore, it contains a call

[2]Even though the system runs on real phones, we choose an emulator based approach since it is easier to run a large set of experiments concurrently.

One-Click App	Exploit
O_1	**/dev/camera-sysram**
O_2	**/dev/graphics/fb5**
O_3	**/dev/exynos-mem**
O_4	**/dev/camera-isp**
O_5	**/dev/camera-isp**
O_6	**/dev/camera-isp**
O_7	**towelroot**

Table 4.1: One-Click apps with the discovered exploits.

log, SMS history and contacts, as well as various installed apps. We have modified the binary image of the emulator, in order to show that it has a real phone number and a real International Mobile Equipment Identity (IMEI) number. Finally, the `build.prop` file (containing the device information) is updated appropriately prior to each experiment. Each app is analyzed starting with a clean image of the emulator in order to avoid any side effects that a previously tested malware app can have on the emulator. A simple micro-benchmark on the `open()` system calls shows that the system call monitor increases the execution time of `open()` by 75%, on average.

Input generator: To achieve as much code coverage as possible when executing an app (in hope that root exploits will be triggered), we leverage DroidBot [80], a lightweight test input generator for Android that generates pseudo-random streams of user events such as clicks, as well as a number of system-level events. DroidBot can generate random events on its own, or generate events based on the manifest file of the app, or can take as input a file with predefined events. In our experiments, we use randomly generated events ("black-box" technique) and events based on the manifest file of the app ("gray-box" technique).

4.8.2 Effectiveness

We evaluate *RootExplorer* against all the test datasets mentioned earlier containing 4497 apps in total. Overall, we do not find any false positives, *i.e.*benign apps are never mistakenly reported to contain root exploits. For the known malware samples, we obtain the ground truth either from the fact that github explicitly states that it is a root exploit, or via cross-validation

Exploit	VirusTotal	RootScanner
diag	**1/57**	✓
exynos	**4/57**	✓
pingpong	**1/57**	✓
towelroot	**3/57**	✓

Table 4.2: Detection rate for debug compilation.

Exploit	VirusTotal	RootScanner
diag	**0/57**	✓
exynos	**1/57**	✓
pingpong	**0/57**	✓
towelroot	**1/57**	✓

Table 4.3: Detection rate for obfuscated compilation.

with VirusTotal and the antivirus company that we work with. Out of 8 known malware families containing root exploits, we do not find any false negative. We describe the details below.

Unit testing: To obtain assurance that the training phase works as expected, We execute the 60 families of root exploits (from the training data) in our dynamic analysis environment and see if they can be detected. Note that this means the training and testing data are exactly the same. If any of these exploits cannot be detected, it indicates that the behavior graphs or preconditions that were prepared are in fact incorrect. The testing results show that all of the exploits are successfully detected.

Detecting other one-click root apps: Testing *RootExplorer* against other one-click root apps allows us to further confirm that the system works well. Since the exploits from these apps may or may not be implemented exactly in the same way as the ones in our training set, being able to trigger and detect them is a promising sign. Table 4.1 lists the first exploit that was caught upon running 7 other one-click root apps on an emulated Samsung Galaxy S3 device. Interestingly, different one-click root apps in fact choose to launch different exploits against our device. For instance, with O_1, we caught an exploit related to the `/dev/camera-sysram` driver, while O_2 and O_3 triggered exploits against `/dev/graphics/fb5` and `/dev/exynos-mem` respectively. The results showcase the effectiveness of *RootExplorer* in detecting a wide variety of exploits.

Detecting Exploit PoCs from the Internet: We next take four proof-of-concept root exploits (with source code) that we can find on github [76, 77, 79, 81], and embed them in a testing Android app we build, that simply roots a phone upon touching a button. We first check the effectiveness of current anti-virus programs against the "malware" we built containing publicly available PoCs. We scan the app using the virusTotal API [178] which contains 57 anti-virus programs

(*e.g.*Trend Micro) capable of scanning Android apps . Table 4.2 shows the detection rates for the case where we compiled the source code with all the debug options on and without any obfuscation, while Table 4.3 shows the results when the compiled binaries are obfuscated using the O-LLVM obfuscator [109] and packed using UPX [175] (both are off-the-shelf tools).

In brief, without obfuscation, all four exploits can be detected by at least one antivirus. However, with simple obfuscation, only exynos (CVE-2012-6422) [70] and towelroot (CVE-2014-3153) [72] can be successfully detected and that too by only one antivirus. On the other hand, *RootExplorer*, by preparing the right preconditions and observing the exploit behaviors at runtime, can detect every exploit regardless of the obfuscation and packing methods.

Detecting GODLESS: GODLESS [132] is a family of malware that employs multiple root exploits, and has caused havoc in the wild since mid-2016. *RootExplorer* is extremely effective in detecting the exploits in the GODLESS malware family. Its source code is largely based on the open source repository on github [75]. Specifically, GODLESS checks the device type against a predefined, populated database of supported exploitable devices. Depending on which device it is running on, it invokes a corresponding, appropriate exploit. The process is iterative. It begins with `acdb` and checks if the device is in the database, and only if so, will continue with the actual exploit. Upon failure, it moves on to next exploit which is `hdcp`, and so on until it has tested the last exploit viz., `diag`. We run GODLESS against 5 different emulated devices to showcase that *RootExplorer* is effective in properly stimulating the right exploit for a device. Table 4.4 shows the results (with the emulated devices). The exploits with code name `msm_camera`, `put_user` and `fb_mem` can be caught using any emulated device; this is because these exploits affect a large number of devices and seemingly, the author of GODLESS does not even know the complete list of devices they affect. Instead, GODLESS simply always tries to execute them without checking the actual device type. Of course, if a target device does not have the vulnerable device file such as `/dev/msm_camera`, the exploit will simply abort and the next exploit is attempted. Since *RootExplorer* is trained to prepare the preconditions for all the exploits at all times including `msm_camera`, it deploys the decoy file `/dev/msm_camera` on demand when GODLESS tries to open it, and can therefore always trigger

	HTC J	Fujitsu Z	Fujitsu X	Galaxy Note	Samsung S3
acdb	✓	✗	✗	✗	✗
hdcp	✗	✓	✗	✗	✗
msm_camera	✓	✓	✓	✓	✓
put_user	✓	✓	✓	✓	✓
fb_mem	✓	✓	✓	✓	✓
perf_swevent	✗	✗	✓	✗	✗
diag	✗	✗	✗	✓	✗

Table 4.4: Emulated devices and corresponding exploits caught by *RootExplorer* in GODLESS malware.

and detect its complete malicious behavior with respect to this exploit.

Detecting Malware in the Antivirus malware dataset and 3rd-party Android Markets: For each app from the 1497 malware samples we received from an anti-virus company and the 2000 apps downloaded from four third-party Android markets, we apply *RootExplorer* for 10 minutes using Droidbot with an emulated Samsung S3 device; the kernel version, build ID, and the model of the device are set to 3.0.31-1083875, JZO54K, and GT-I9300 respectively. Upon booting the emulated device, Droidbot launches the main activity of each app and generates random touch events and system events such as `BOOT_COMPLETED` every second. Meanwhile, our tool runs in the background and analyzes all the system calls that the app uses. To measure the number of false positives and false negatives, we scan those apps with VirusTotal. Among all the apps, *RootExplorer* detected two true positives (and has no false positive).

The first app is named *Wifi Analyzer* from the MoboMarket [19], which was discovered to contain the pingpong root exploit [193] (`md5` ea1118c2c0e81c2d9f4bd0f6bd707538). At the time of writing, the app is still alive on the market. After consulting with VirusTotal and an antivirus company, we confirmed that it is an instance of the `rootnik` malware family [134]. We have reported to the market and are waiting for the app to be removed.

Another detected app is a Flashlight app from the Antivirus malware dataset, containing the `camera-isp` root exploit. It has an `md5` of 1365ECD2665D5035F3AB52A4E698052D. Upon starting, the app tries to access the files `/system/xbin/su` and `/system/bin/su`. *RootExplorer* returns the appropriate errors to make the app believe that it is running on an un-rooted device.

Interestingly, only when DroidBot delivers the `BOOT_COMPLETED` event to the app, the root exploit is triggered. In the beginning, it opens and reads the file `/proc/kallsyms` four times to retrieve the address of certain kernel symbols. After that, it opens the vulnerable `/dev/camera-isp` device file[3]. It subsequently invokes two different `ioctl()` system calls with request types `0xC0086B03` and `0xC0186201` that effectively compromise the driver. As expected, *RootExplorer* deploys the decoy file `/dev/camera-isp` which returns a real file descriptor for `open()`, and success for `ioctl()` (to trick the exploit into believing that it has succeeded). Finally, the exploit performs a `setresuid(0,0,0)` to get root access. At that point, *RootExplorer* successfully finds the root exploit and stops the execution of the app.

In addition to the above two malware samples, VirusTotal also reports three additional malware samples that carry root exploits. We analyzed these cases manually and found that they in fact attempt to download the exploits from a specific URL which is no longer valid. In other words, the exploits are never executed even though the malware may have done it in the past.

4.9 Conclusions

In this project, we tackle the challenging problem of detecting the presence of embedded root exploits in malware. We build a system *RootExplorer*, that learns from commercial-grade root exploits used for benign reasons and backed by large companies such as Baidu and Tencent, and detects such embedded root exploits. Specifically, it carefully analyzes these to determine what environments root exploits expect, and what pre-conditions are to be satisfied in order to trigger them. It uses this information to construct proper analysis environments for malware and can effectively detect the presence of root exploits. Our extensive evaluations shows that it can detect all known malware samples carrying root exploits, and has no false positives. We are also able to detect a root exploit in a malware that seems to have bypassed current vetting procedures, and is available on an Android market.

[3]Note that in this case, the exploit targets a different vulnerability in the same device driver from the example in Section 4.5.

Bibliography

[1] AForge.NET. http://www.aforgenet.com/framework/features/motion_detection/_2.0.html.

[2] Android device inventory. https://www.androiddevice.info/.

[3] EvalVid with GPAC. http://www2.tkn.tu-berlin.de/research/evalvid/EvalVid/docevalvid.html.

[4] FFmpeg. http://ffmpeg.org/.

[5] Firesheep. http://codebutler.com/firesheep/.

[6] Google apps documentation and support. http://support.google.com/a/bin/answer.py?hl=en&answer=1279090.

[7] GPAC. http://gpac.wp.mines-telecom.fr/.

[8] A guide to sniffing out passwords and cookies. http://lifehacker.com/5853483.

[9] Report: Mobile uploads up fourteen-fold. http://www.pcmag.com/article2/0,2817,2392007,00.asp.

[10] Surveillance self-defense. http://bit.ly/deiE0O.

[11] What's APPening wtih Apple FaceTime. http://researchcenter.paloaltonetworks.com/2010/08/whats-appening-with-apple-facetime/.

[12] x264. http://www.videolan.org/developers/x264.html.

[13] YUV CIF reference videos (lossless H.264 encoded). http://www2.tkn.tu-berlin.de/research/evalvid/cif.html.

[14] *Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy*, 2016.

[15] Android security bulletin — january 2017, 2018. https://source.android.com/security/bulletin/2017-01-01.html.

[16] 3rd-party Android Market. 7723 market, 2017. https://goo.gl/iMi4Bo.

[17] 3rd-party Android Market. Android life, 2017. https://goo.gl/hAov2G.

[18] 3rd-party Android Market. Eoemarket, 2017. https://goo.gl/FB0ykP.

[19] 3rd-party Android Market. Mobomarket, 2017. https://goo.gl/tzpjY7.

[20] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX SECURITY*, 2016.

[21] Vijay Kumar Adhikari et al. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *Proceedings of the IEEE Conference on Computer Communications, (INFOCOM 2012)*, Orlando, FL, March 2012.

[22] American National Standards Institute. *ANSI X3.92-1981 American National Standard, Data Encryption Algorithm*, 1981.

[23] Androguard. Androguard, a full python tool to play with android files, 2016. `https://goo.gl/edcClw`.

[24] Any.do. To-do list, task list, 2016. `https://goo.gl/rPpZq8`.

[25] AOP. Aspect oriented programming with spring, 2016. `http://goo.gl/1UnkGS`.

[26] Apktool. A tool for reverse engineering android apk files, 2016. `https://goo.gl/JCh7U7`.

[27] 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. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[28] Askmd, 2016. `https://goo.gl/3D5Vvw`.

[29] Soren Asmussen. *Applied Probability and Queues*. John Wiley & Sons, 1987.

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

[31] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, volume 48, pages 641–660. ACM, 2013.

[32] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security*, pages 691–706, 2015.

[33] Baidu. Shoujiweishi, 2017. http://shoujiweishi.baidu.com/.

[34] John S. Baras, Vahid Tabatabaee, George Papageorgiou, and Nicolas Rentz. Performance metric sensitivity computation for optimization and trade-off analysis in wireless networks. In *IEEE GLOBECOM*, 2008.

[35] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[36] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.

[37] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38. ACM, 2015.

[38] Aggressive advertisers pose privacy risks, 2013. `http://goo.gl/cl8HuK`.

[39] Mobile operating system wars - Android vs. iOS, 2013. `http://goo.gl/V0wbT5`.

[40] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE, 2010.

[41] Alan C. Bovik. *The Essential Guide to Video Processing*. Academic Press, 2009.

[42] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[43] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[44] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[45] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. ipshield: a framework for enforcing context-aware privacy. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 143–156. USENIX Association, 2014.

[46] CheckPoint. Quadrooter: New android vulnerabilities in over 900 million devices, 2016. https://goo.gl/GN6ZwW.

[47] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, page 234, 2013.

[48] Cloc. https://goo.gl/PsfjQP.

[49] U.S. copyright office. Copyright protection and management systems, 2017. https://goo.gl/zpeUtK.

[50] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 117–130. ACM, 2007.

[51] Jedidiah R Crandall, Zhendong Su, S Felix Wu, and Frederic T Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248. ACM, 2005.

[52] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 252–266. IEEE, 2007.

[53] Google Developers. Improving code inspection with annotations, 2016. http://goo.gl/qSE9dh.

[54] Marko Dimjaševic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Android malware detection based on system calls. *University of Utah, Tech. Rep*, 2015.

[55] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, 2014.

[56] DirtyCow. Cve-2016-5195, 2017. https://goo.gl/K8cWEK.

[57] Florin Dobrian et al. Understanding the impact of video quality on user engagement. *Communications of the ACM*, 56(3):91–99, March 2013.

[58] Eclipse. Aspectj, 2016. https://goo.gl/LHLhDv.

[59] Elf manual entry, 2016. https://goo.gl/96WkSL.

[60] Salma Elmalaki, Lucas Wanner, and Mani Srivastava. Caredroid: Adaptation framework for android context-aware applications. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 386–399. ACM, 2015.

[61] Diaa Salama Abdul Elminaam et al. Performance evaluation of symmetric encryption algorithms. *IJCSNS International Journal of Computer Science and Network Security*, 8(12):280–286, December 2008.

[62] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[63] Yliès Falcone and Sebastian Currea. Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 350–353. ACM, 2012.

[64] Zheran Fang, Weili Han, Dong Li, Zeqing Guo, Danhao Guo, Xiaoyang Sean Wang, Zhiyun Qian, and Hao Chen. revdroid: Code analysis of the side effects after dynamic permission revocation of android apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 747–758. ACM, 2016.

[65] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[66] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[67] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[68] Wolfgang Fischer et al. The Markov-modulated Poisson process (MMPP) cookbook. *Performance Evaluation*, 18(2):149–171, September 1993.

[69] Standard for Information Security Vulnerability Names. Cve-2012-4220, 2012. https://goo.gl/JZJHyv.

[70] Standard for Information Security Vulnerability Names. Cve-2012-6422, 2012. https://goo.gl/R7Icm7.

[71] Standard for Information Security Vulnerability Names. Cve-2013-2597, 2013. https://goo.gl/MvLlsN.

[72] Standard for Information Security Vulnerability Names. Cve-2014-3153, 2014. https://goo.gl/R7Icm7.

[73] Harald T. Friis. A note on a simple transmission formula. *Proceedings of the IRE*, 34(5):254–256, May 1946.

[74] Michele Garetto, Theodoros Salonidis, and Edward W. Knightly. Modeling per-flow throughput and capturing starvation in CSMA multi-hop wireless networks. In *IEEE INFOCOM*, 2006.

[75] Github. android_run_root_shell (base for godless), 2017. https://goo.gl/VKSWb6.

[76] Github. Cve-2012-6422, 2017. https://github.com/dongmu/vulnerability-poc/tree/master/CVE-2012-6422.

[77] Github. Cve-2014-3153 aka towelroot, 2017. https://github.com/timwr/CVE-2014-3153.

[78] Github. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[79] Github. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[80] Github. Droidbot, 2017. https://goo.gl/y8ldRA.

[81] Github. exploit for cve-2012-4220 working on zte-open, 2017. https://github.com/poliva/root-zte-open.

[82] Google. Android and security, 2012. https://goo.gl/mo29A4.

[83] Google. Android dashboards, 2016. `https://goo.gl/JXU1t3`.

[84] Google. Android runtimepermissionsbasic sample, 2016. `https://goo.gl/t59Dw9`.

[85] Google. Battery historian, 2016. `https://goo.gl/YvfzCz`.

[86] Google. Google play store, 2016. `https://goo.gl/kN0Nhz`.

[87] Google. Material design patterns – permissions, 2016. `https://goo.gl/QQcfEv`.

[88] Google. Requesting permissions at run time, 2016. `https://goo.gl/0enMi9`.

[89] Google. What is api level ?, 2016. `https://goo.gl/xYXo0T`.

[90] Google. Android ndk, 2017. https://goo.gl/n3uxti.

[91] Google. Dashboards, 2017. https://goo.gl/6BTWw4.

[92] Google. Security-enhanced linux in android, 2017. https://goo.gl/btJ9xb.

[93] Google. Verified boot, 2017. https://goo.gl/LiQm9E.

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

[95] Harry Heffes and David M. Lucantoni. A Markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance. *IEEE J. Sel. Areas Commun.*, 4(6), September 1986.

[96] Hex-Rays. Ida pro, 2017. https://goo.gl/cqgKCM.

[97] Mukesh M. Hira et al. Throughput analysis of a path in an IEEE 802.11 multihop wireless network. In *IEEE WCNC*, 2007.

[98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*

[99] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[100] Hummingbad android malware found in 20 google play store apps, 2016. https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/.

[101] Oliver C. Ibe. *Markov Processes for Stochastic Modeling*. Academic Press, 2008.

[102] International Telecommunications Union. *ITU-R BT.601: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*.

[103] ISO/IEC JTC1/SC29/WG11. ISO/IEC 14496 – Coding of audio-visual objects. `http://mpeg.chiariglione.org/standards/mpeg-4/mpeg-4.htm`.

[104] Michel T. Ivrlač, Ruly Lai-U Choi, Eckehard G. Steinbach, and Josef A. Nossek. Models and analysis of streaming video transmission over wireless fading channels. *Signal Proessing: Image Communication*, 24(8):651–665, September 2009.

[105] Java 1.8 parser and abstract syntax tree for java, 2016. `https://goo.gl/qI1f34`.

[106] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[107] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.

[108] Jaeyeon Jung, Seungyeop Han, and David Wetherall. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 45–50. ACM, 2012.

[109] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[110] Sang H. Kang et al. Two-state MMPP modeling of ATM superposed traffic streams based on the characterization of correlated interarrival times. In *IEEE GLOBECOM*, 1995.

[111] Michelle Atkinson Kenneth Olmstead. Apps permissions in the google play store, 2015. `http://goo.gl/ph7KGk`.

[112] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[113] Yosuke Kikuchi, Hiroshi Mori, Hiroki Nakano, Katsunari Yoshioka, Tsutomu Matsumoto, and Michel van Eeten. Evaluating malware mitigation by android market operators. In *9th Workshop on Cyber Security Experimentation and Test (CSET 16)*. USENIX Association, 2016.

[114] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[115] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. John Wiley & Sons, 1975.

[116] Leonard Kleinrock. *Queueing Systems, Volume II: Computer Applications*. John Wiley & Sons, 1976.

[117] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351–366, 2009.

[118] Harold J. Kushner. *Heavy Traffic Analysis of Controlled Queueing and Communication Networks*. Springer, 2001.

[119] Deer Li and Jianping Pan. Performance evaluation of video streaming over multi-hop wireless local area networks. *IEEE Trans. Wireless Commun.*, 9(1):338–347, January 2010.

[120] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[121] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

[122] Shuying Liang, Andrew W Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 21–32. ACM, 2013.

[123] Zhenkai Liang and R Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222. ACM, 2005.

[124] Linux. objdump - linux man page, 2017. https://goo.gl/6Wf5oS.

[125] Benjamin Livshits and Jaeyeon Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 113–130, 2013.

[126] Tom Lookabaugh and Douglas C. Sicker. Selective encryption for consumer applications. *IEEE Commun. Mag.*, 42(5):124–129, May 2004.

[127] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[128] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, 2011.

[129] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1-2):1–13, 2012.

[130] Linux man page. strace, 2017. https://goo.gl/zG7YIO.

[131] Kamesh Medepalli et al. Towards performance modeling of IEEE 802.11 based wireless networks: A unified framework and its applications. In *IEEE INFOCOM*, 2006.

[132] Trend Micro. Godless mobile malware uses multiple exploits to root devices, 2016. https://goo.gl/qKSCXl.

[133] 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 *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, 2015.

[134] Palo Alto Networks. Rootnik android trojan abuses commercial rooting tool and steals private information, 2015. https://goo.gl/epd1IB5.

[135] Marcel F. Neuts. A versatile Markovian point process. *Journal of Applied Probability*, 16(4):764–779, December 1979.

[136] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Security and Privacy, 2005 IEEE Symposium on*, pages 226–241. IEEE, 2005.

[137] Mladen Nikolic. Measuring similarity of graphs and their nodes by neighbor matching. *arXiv preprint arXiv:1009.5290*, 2010.

[138] Nima Nikzad, Octav Chipara, and William G Griswold. Ape: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 515–526. ACM, 2014.

[139] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[140] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX security symposium*, pages 543–558, 2013.

[141] Oracle. Java se annotations, 2016. `http://goo.gl/g9b0Dh`.

[142] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 527–542, 2013.

[143] George Papageorgiou, John Gasparis, Srikanth V. Krishnamurthy, Ramesh Govindan, and Tom La Porta. Securing mobile video uploads from eavesdroppers with minimum performance penalties: Tech report. `http://www.cs.ucr.edu/~gpapag/tech-report-encryption.pdf`, June 2013.

[144] George Papageorgiou, Shailendra Singh, Srikanth V. Krishnamurthy, Ramesh Govindan, and Tom La Porta. Distortion-resilient routing for video flows in wireless multi-hop networks. In *IEEE ICNP*, 2012.

[145] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[146] Pomelo, LLC. Analysis of Netflix security framework for 'Watch Instantly' service. Technical report, March 2009.

[147] P. Prasithsangaree and P. Krishnamurthy. Analysis of energy consumption of RC4 and AES algorithms in wireless LANs. In *IEEE GLOBECOM*, 2003.

[148] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. On tracking information flows through jni in android applications. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 180–191. IEEE, 2014.

[149] Qualcomm. Lumicast, a guiding light for indoor venues., 2016. `https://goo.gl/tIXHj7`.

[150] Moo-Ryong Ra, Ramesh Govindan, and Antonio Ortega. P3: Toward privacy-preserving photo sharing. In *USENIX NSDI*, 2013.

[151] Amir Rahmati and Harsha V Madhyastha. Context-specific access control: Conforming permissions with user expectations. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 75–80. ACM, 2015.

[152] V. Ramaswami. The N/G/1 queue and its detailed analysis. *Advances in Applied Probability*, 12(1):222–261, March 1980.

[153] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 2014.

[154] Ringdroid. Ringdroid, 2016. `https://goo.gl/MhLqGW`.

[155] Sanae Rosen, Zhiyun Qian, and Z Morely Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[156] Sandvine. Global internet phenomena report, Spring 2011. http://bit.ly/2SihDa, 2011.

[157] SecureList. Rooting pokémons in google play store, 2016. https://goo.gl/Ry7AUw.

[158] Patrick Seeling et al. Video transport evaluation with H.264 video traces. *IEEE Trans. Multimedia*, 14(4):1142–1165, FOURTH QUARTER 2012.

[159] Beomjoo Seo et al. An experimental study of video uploading from mobile devices with http streaming. In *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12. ACM, 2012.

[160] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Maoo. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[161] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, volume 4, pages 4–4, 2004.

[162] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[163] Craig Smith. Android statistics, 2016. https://goo.gl/9Pp6I5.

[164] Songily, 2016. https://goo.gl/fFWI1m.

[165] Soot - a java optimization framework, 2016. https://goo.gl/UsmKcC.

[166] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[167] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. Citeseer, 2012.

[168] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[169] Joshua Tan, Khanh Nguyen, Michael Theodorides, Heidi Negrón-Arroyo, Christopher Thompson, Serge Egelman, and David Wagner. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 91–100. ACM, 2014.

[170] Target. Plan, shop & save, 2016. https://goo.gl/oupAB7.

[171] The right way to ask users for ios permissions, 2014. https://goo.gl/fnxF6i.

[172] thescore: Sports scores, 2016. https://goo.gl/iefw9C.

[173] Location Tracker. Location tracker, 2015. https://goo.gl/X2LuAd.

[174] United States National Institute of Standards and Technology (NIST). *Announcing the Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197.*, November 2001.

[175] Upx, 2017. https://goo.gl/6BkD4i.

[176] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[177] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[178] Virustotal, 2017. https://goo.gl/Fw7yPC.

[179] Helen J Wang, Chuanxiong Guo, Daniel R Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 193–204. ACM, 2004.

[180] Wei Wang, Michael Hempel, Dongming Peng, Honggang Wang, Hamid Sharif, and Hsiao-Hwa Chen. On energy efficient encryption for video streaming in wireless sensor networks. *IEEE Trans. Multimedia*, 12(5):417–426, August 2010.

[181] Yubing Wang, Mark Claypool, and Robert Kinicki. Impact of reference distance for motion compensation prediction on video quality. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN 2007)*, 2007.

[182] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[183] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[184] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414*, 1:5, 2014.

[185] Whatsapp messenger, 2016. `https://goo.gl/W1QcPv`.

[186] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):560–576, July 2003.

[187] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[188] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[189] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *CCS*, 2013.

[190] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914. IEEE, 2015.

[191] Lifan Xu, Dongping Zhang, Nuwan Jayasena, and John Cavazos. Hadm: Hybrid analysis for detection of malware. 2016.

[192] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing permission usage in android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 400–410. IEEE, 2013.

[193] Wen Xu and Yubin Fu. Own your android! yet another universal root. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[194] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

[195] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 303–313. IEEE, 2015.

[196] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[197] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1093–1104. ACM, 2015.

[198] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[199] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 259–270. ACM, 2014.

[200] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[201] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Oakland*, 2014.

[202] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[203] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.