Physical Media Covert Channels on Smart Mobile Devices

Ed Novak Yutao Tang Zijiang Hao Qun Li
The College of William and Mary, Williamsburg, VA
{ejnovak,yytang,hebo,liqun}@cs.wm.edu

Yifan Zhang
Binghamton University, Binghamton, NY
zhangy@binghamton.edu

ABSTRACT

In recent years mobile smart devices such as tablets and smartphones have exploded in popularity. We are now in a world of ubiquitous smart devices that people rely on daily and carry everywhere. This is a fundamental shift for computing in two ways. Firstly, users increasingly place unprecedented amounts of sensitive information on these devices, which paints a precarious picture. Secondly, these devices commonly carry many physical world interfaces. In this paper, we propose information leakage malware, specifically designed for mobile devices, which uses covert channels over physical "real-world" media, such as sound or light. This malware is stealthy; able to circumvent current, and even state-of-the-art defenses to enable attacks including privilege escalation, and information leakage. We go on to present a defense mechanism, which balances security with usability to stop these attacks.

Author Keywords
Physical Media; Covert Channel; Sensors; Privacy; Security; Smart Mobile Device

ACM Classification Keywords
C.2.0 Computer-Communication Networks: General – Security and Protection; D.4.6 Operating Systems: Security and Protection

INTRODUCTION

Since the introduction of the iPhone in 2007 mobile computing using smartphones and tablets has exploded and is still climbing [21]. There is currently a plethora of Windows, Android, and iOS devices available to consumers containing a wide assortment of physical sensors. Consumers have come to expect smart mobile devices as common place. They are now carried everywhere and relied on daily. This is proving to be a fundamental shift for computing in two ways. First, these devices carry an unprecedented amount of sensitive information such as contact information, passwords, GPS location traces, financial information, and personal attributes such as sexual orientation and health data. Secondly, smart mobile devices are commonly equipped with a variety of physical interface hardware such as accelerometers, cameras, vibration motors, speakers, and microphones. Users of smart devices are expected to be conscious of a myriad of security *best practices* to protect this data, such as using unique and strong passwords for their various services, enforcing reasonable permission allocations to applications, and protecting highly sensitive information with encryption (or keeping this information off the device). These sort of defense schemes are largely lost on users [6], and can be difficult to implement for developers [7, 26]. In this paper we preemptively propose a new form of malware, tailored specifically to mobile computing, along with a first step defense mechanism in hopes to alert the community of this potential threat and spur future research.

This work is interesting because we can exploit several properties of mobile computing. Firstly, the abundance of sensitive information found on these devices is highly valuable to attackers. Passwords are an obvious example, but attackers may also try to learn users' addresses and financial information, which can be used to steal their identity. Users' political views or sexual orientations, can be used by oppressive governments or organizations. Their personal traits and interests can be used to answer common second factor "security questions" such as their childhood home, or pet's name. Secondly, we exploit the variety of physical world sensors to establish several unique covert channels, which we refer to as "physical media covert channels" (PMCC). Because of the architecture currently in place in mobile operating systems, we show that we can easily design malware, using PMCC, to appear benign, fooling both non-expert humans and software systems seeking to eliminate malicious software; making it a variant of a Trojan.

Designing and implementing the PMCCs we use in our malware is difficult, because we must achieve two goals at the same time. First, speed. Previous work has shown that as little as 100 bits per second is enough to pose a serious threat [41], we show that at least one of our covert channels can achieve a few thousand bits per second. Secondly, stealth. The stealth of the covert channels is very important so they are difficult to defend against. They must go unnoticed by the user, and appear to be benign from the point of view of the software. Additionally, defending against these channels is challenging, because we must differentiate between benign and malicious sensor use without interrupting key features or annoying the user with confirmation dialogs.

Our attacks overcome not only the current, widely deployed defense mechanisms, but also the defense schemes posed in recent literature. Most recent work in this area can be divided into three areas: taint-analysis, elaborate security policy mechanisms, and application market curation. Taint-analysis can be used to identify sensitive information as it flows through an application and notifying the user or stopping this sensitive information from leaving the device and being leaked. Sensitive information sources are marked as such (e.g., the user's contact list) and sinks are identified that will ultimately leak this information (e.g., the Socket API). Security policies can be written by users, or automatically, that disallow certain types of malicious behavior. Market curation techniques aim to identify and remove malicious applications from the market before users even have a chance to install it. Our attacks overcome all of these approaches because of our physical medium covert channels, which are novel and difficult to differentiate from benign behavior accurately. Our defense scheme provides a framework, that can be extended in the future, to identify and defend against variants of the malware we propose.

In this paper we identify a new malware specifically designed for mobile devices. PMCCs used in the malware are constructed using the sensors found on smart devices. We show that our malware can easily leak sensitive user information, such as bank statements, or location traces despite any current defense mechanism. Building such a system introduces several challenges including achieving a high bit rate, and low detectability (stealth). We go on to propose a defense scheme against these attacks, which can be implemented by the operating system provider, that aims to enable taint tracking over these channels. To summarize, in this paper we make the following contributions:

- We propose a new class of covert channels for smart mobile devices that utilize real world interfaces (e.g., the vibration motor and the accelerometer). We generalize these as "physical medium covert channels" and demonstrate that they can achieve varying levels of stealth, and speed.
- We design, and implement five example PMCCs, which utilize ultra-sound, physical vibration, light, and the user themselves. We spend extra effort on our ultrasound channel to show that with some engineering effort we can achieve a very high transmission rate.
- We use our PMCCs to design a new variant of Trojan Horse malware, which appears to be benign but actually leaks sensitive user information. We give a specific example called "Jog-Log."
- We propose and implement a novel defense scheme that takes a framework approach. The defense aims to propagate taint information across these channels, while still maintaining high usability of benign applications.
- We evaluate prototypes of each covert channel and the defense mechanism. We show that our ultrasound covert channel achieves 3.71kbps, and our gyroscope channel is very stealthy. Our defense scheme maintains high usability for the user, while stopping all of the proposed attacks with low overhead.

RELATED WORK

Covert channels have a rich academic history [18, 23]. Butler Lampson first described "the confinement problem" [17], in which one entity (the client) must trust another (the server) with some data to perform some calculation. Ideally, the server would be confined, and unable to transmit or store the data. Mr. Lampson, however, was unable to envision all the possible ways that the service may transmit this data. To put the work in this paper in the context of the confinement problem, our work adds new channels by which a confined server can transmit data to itself, or to another process.

Covert channels on smartphones have been studied previously [5, 32], but to the best of the author's knowledge, we are the first to propose covert channels that use physical media, rather than internal / virtual media such as processor workload, or file size.

There are several works [22, 32, 34, 36, 37] that attempt to steal some data from the user, utilizing a variety of different internal / virtual covert channels to perform privilege escalation, and circumvent taint-tracking analysis, similar to our work. Our contribution in this paper is unique in that it is the first to use PMCCs. These channels are particularly dangerous because they are much more difficult to identify as malicious. We also propose a robust defense.

Works on defense mechanisms for the attacks similar to what we propose in this paper can be broken into three categories. In the first, researchers propose that we replace sensitive data with "mock" data when feeding it to untrusted applications [2, 15]. These systems do protect user information, but they erode the quality of benign applications and are intrusive to users, who must make either a few broad decisions or many small decisions to protect their data.

The second category of defense is taint tracking analysis, [3, 9, 11, 16]. In these systems, sensitive information APIs are marked as "sources." Variables that store this information (e.g., microphone data, user contacts) are marked as "tainted." The tainted data are followed through the execution of the program, tainting other data that they effect explicitly (direct assignment). If tainted information reaches a sensitive sink (e.g., Internet Socket, IPC, etc.), the flow is stopped and/or the user is notified.

Machine learning is proposed in [30] to identify sources and sinks during run-time. Their system, "SuSi", relies on supervised learning, which means a list of known (manually tagged as malicious or benign) Android APIs are used to train a classifier. The system classifies based on features such as method name (i.e., “get...”, “put...”), and method call return type. These taint-tracking systems cannot be naively adapted for use to defend against our covert channels because we cannot treat every invocation of a sensor API with sensitive information, to be malicious. An application may use a device benignly at one time, and maliciously at another time. For example, an incoming SMS will almost certainly activate the speaker or vibration motor or both benignly.

The third defense mechanism is alternate permission systems, [10, 19, 33, 38, 40]. These systems are not a good solution to the attack proposed in this paper, because any user interaction with sensitive sources is interrupted with some GUI prompt. Because so many covert channels can be built, and sensitive information is commonly accessed on these devices, these prompts would arise far too frequently. Many systems propose more fine-grained permissions, [4, 8, 25, 43]. These systems rely on the user to construct policies, which will stop malicious use cases, but allow benign ones. We argue that writing an effective policy is too difficult for users.

Our defense scheme is similar to the recent work “ASM” [14]. Both systems hook into key system events, and then allow application developers or security researchers to define specific actions that can be taken on these events. We are not able to leverage this work directly in our implementation because the current ASM implementation does not include the devices we focus on in this paper such as the sensors, speaker and microphone, or the vibration motor. Our contribution, which ASM encourages, is the specifics of the treatment once these events occur.

THREAT MODEL

We assume the attacker is remote. He or she does not have any physical access to the user’s device and cannot alter the operating system, or any other software, already running on the user’s device. The attacker’s goal is to convince the user to run their malicious code, which is used to obtain some information about the user. By “attain” we mean that the attacker’s code accesses some sensitive information, and transmits it, over the Internet, to an attacker controlled host machine.

In order to send information out over the Internet in the malicious application, without requesting the Internet permission, the attacker can exploit a simple capability leak, which is a form of a *confused deputy attack* [11] using the web browser. Android applications can ask the browser to open URLs on their behalf without declaring the Internet permission. The attacker can include some CGI parameters (e.g., attackerhost.com/collector?usersecret=val) to transmit sensitive information to their own controlled host. Although there are protections against these attacks [3, 20], they are not widely deployed. Furthermore, the attacker can always just declare the Internet permission and communicate with their host directly, although this is less stealthy to the user.

Creating Applications The attacker is able to implement Android applications which are, from the point of view of the adversary, normal applications. The attacker does not have to provide the source code to these apps, distributing only .apk files, which are essential .zip archives of java byte code and application assets such as image files and text files containing localized strings.

In order to convince the user to install these applications, the attacker can employ a few well studied techniques. Specifically, to circumvent application market curation techniques [1, 45, 46, 47], a malicious application developer can employ obfuscation such as utilizing native code, java reflection, remote code downloading, identify malware-detection environments [27], or other more sophisticated techniques [28, 31, 37]. This allows the application to appear in application markets, drastically increasing the number of people that will run it.

To be more specific, the attacker aims to build a trojan, which by definition, appears to be benign from the point of view of the user, but actually performs some malicious activity. The attacker can even implement complete benign functionality to produce a convincing trojan.

Timing The attacker can employ standard Android timing mechanisms such as the “ScheduledExecutorService”, in order to execute certain portions of their code at certain times of day (to increase stealth), and to ensure that two different components (possibly in two different applications) run at the same time to facilitate covert channel transmission.

Accessing Sensitive Information As mentioned previously, there are many pieces of information that may be considered sensitive on a typical smart mobile device. This information can be organized into three forms based on how it is accessed.

First, applications maintain sensitive information and may expose it for use by other applications. For example, the contact manager application maintains contact information for other people. Typically individual contacts can be “shared” via an interface between the contact app and another app (e.g. the SMS app may be able to access an individual contact to send the phone number of that contact to a friend). Typically this type of information is guarded with some permissions. For example, to read the phone records of the user, the application must declare the “READ_CALL_LOG” permission, which is displayed to the user at install time. For first party applications such as the phone call app, a permission is always necessary, for third party applications it is less common.

Secondly, certain hardware devices can provide sensitive information immediately, such as the GPS radio. These devices are guarded by permissions, but other sensors, which are not traditionally thought of as sensitive, do not require permissions to access, such as the accelerometer and gyroscope.

Third, sensitive information can be extrapolated from these sources. For example, by polling the GPS sensor regularly the attacker can likely learn the user’s home address and work address by examining where they are late at night and during the day on work-days. These sort of inference or extrapolation attacks are plentiful in recent research [19, 34, 36].

Attacker Limitations The attacker cannot access the hardware devices, such as the accelerometer and microphone, except by going through the provided Android APIs. This is

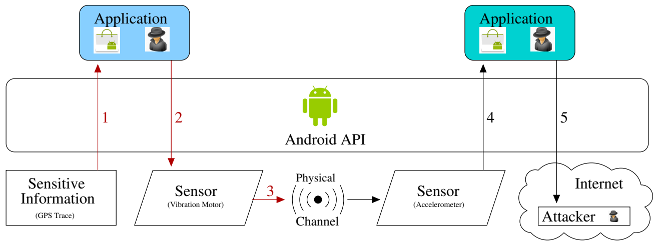


Figure 1. Information flow using physical covert channels.

enforced already in commodity Android due to user permissions, and SELinux. This means that the attacker must abide by the permission system that governs these APIs. The attacker does not have root level access on the user's device.

We do not focus on the case where the attacker tries to use some physical medium covert channel to communicate with another proximal device. Although it may be possible to form some sort of ad-hoc network out of nearby adversary and attacker controlled devices, we consider this scenario to be largely impractical and unlikely.

TROJAN HORSE MALWARE DESIGN

In this section we present a general trojan horse malware, specifically designed for commodity smart mobile devices that utilizes PMCCs. Our malware aims to appear benign to users and software systems, while actually leaking sensitive user information to an attacker controlled host over the Internet. A high level view of the malware can be seen in Fig. 1. The attack works in four phases. First the malware is installed by the user on their personal device. Then it accesses some sensitive information, such as the user's GPS location trace. Third, the malicious application component encodes the information and sends it out over a physical device. In Fig. 1 the vibration motor is used as an example. At the same time, a second attacker controlled application component is used in the background to gather samples from the corresponding sensor. This second application component decodes the signal from the sensor, obtaining the original data (untainted). Finally, the receiver application component sends this information to some attacker controlled host on the Internet.

By "laundering" the information over a PMCC, we circumvent the current, widely adopted permission systems on mobile devices, which try to isolate the data of each application in respective sandboxes. Our channels allow applications to share information without the OS or even current state-of-the-art defense schemes, (including those based on taint-tracking [3, 9, 11, 16]), being alerted. Because none of the systems in recent literature (to the best of our knowledge) account for transmission over physical media, our covert channels pose a novel threat.

Example Trojan Application — Jog-Log

An example trojan application might be a fitness app, which helps users track their running progress. The attacker's goal is to attain the user's home address. The application implements a simple "jogging journal" which determines when, for how long, and where a user jogs to help them track their progress. The application is closed source and is submitted to a well known application market.

The user, interested in jogging, finds the app in the market and installs it. Because they are weary of their information being leaked, they run the taint-droid system on their device [9]. When they want to begin a run, he or she starts the application and the GPS radio is used by the app to track the run. The application declares the permission to access the GPS radio, but it does not request the Internet permission. The application also requests access to the microphone to allow the user to add simple "voice notes" to their journal entires (e.g., "I was greeted with a beautiful sunrise this morning thanks to my new jogging hobby!").

Later, at night when the user is sleeping, the application uses the ScheduleExecutorService to wake up and use a PMCC to transmit the location information gathered earlier during the most recent run. Specifically, the speaker is used to produce an ultrasonic signal. At the same time, the microphone is used to decode the signal in a second component (part of the same application, running simultaneously in a second thread). Because the information has been laundered over a PMCC, the original taint-tags associated with location have been lost. Then, the second component forms a URL with the attacker's host as the domain, and the user's location as a CGI parameter and asks the browser to open this page.

The attacker sets up a special web server to respond to these requests by recording the CGI parameters in a file associated with the IP address of the user. Once the data is at the attacker's host, the attacker can find the street address nearest the GPS coordinates logged at the start and end of the user's runs, which is likely their home address.

Attack Variations

An attacker must control two application components, (e.g. a foreground activity and a background service), one sender and one receiver, to implement a covert channel. These components can be part of the same application, to circumvent taint-tracking analysis, or two different applications, to circumvent taint-tracking *and* the current, widely deployed permission system and information sand-boxing. If the attacker chooses to use two independent applications, they must convince the user to install both of them. To achieve this, the attacker can craft these applications so they are related, with some combined functionality and do in-app cross promotion. This is common practice with many developers in the Android ecosystem. The "Go Launcher" [35] and "Yahoo Weather" [44] apps serve as just two prominent examples.

In our Jog-Log example, the attacker may instead choose to place the voice note feature in a second, stand alone application and do in-app cross promotion. If the user installs both apps, then the attacker can use the covert channel to move the location information from Jog-Log into the voice recording app using a PMCC, and then use the voice recording app to transmit the data to their host. In this case, the voice recording app can declare the Internet permission, and still the user will not expect that there is any way that their location information (in the Jog-Log app with no Internet permission) could possibly be sent out over the Internet.

If the developer chooses to implement two components in the same application, then the malware can defeat current state-of-the-art defense techniques, but will be less stealthy to users, because it must present all of the corresponding permissions together at install time. Fortunately, it has been shown that the permission system is commonly abused by users and developers [26], so this is not a large concern.

If the attacker is a large enterprise that distributes smart mobile devices, they can include the malware via pre-installed applications. This approach is much simpler because the attacker does not need to be concerned with market curation, or convincing users the permissions required do not pose a threat. Pre-installed applications are also significantly more difficult for users to remove, due to both technical and potential legal challenges. However, few attackers have control of an enterprise or a similar level of influence over which apps a user has installed.

COVERT CHANNEL DESIGN

The foundation of our newly proposed malware, is our novel physical media covert channels. In this section we discuss our implementation of five such channels, to increase the attack surface, making the defense more difficult, easing the task of creating seemingly benign malware, and to show that the different channels have different strengths.

Each channel utilizes different permissions. We summarize what Android permissions are used in Table 1. For the remainder of this paper, we will use the row numbers of this table to refer to specific permissions. It is important to note that permissions (4) and (5) are actually members of Android's "uses-feature" tag, not the "uses-permissions" tag, which are *optionally* used by the developer.

All of our channels are stealthy to software, because until now, none of them are considered to be able to transmit and receive information. They have varying stealth in regards to the user, which we detail in each channel subsection.

Ultrasound We put extra effort into the development of our ultrasound covert channel to show that these physical medium covert channels have potential for relatively high bit-rates. This channel uses the speaker and microphone found on smart devices. The main idea is we send very-high frequency, modulated sound waves (above 18kHz) in packets from the device speaker to the microphone. Permissions (1) and (3) are used.

A high level view of the ultrasound covert channel can be seen in Fig. 2. Here the attacker can control one malicious application (housing both the modulator and demodulator), or two applications, where the demodulator is separated out. The malicious sender generates packets in the "Modulator" module and sends sound data over the device's speaker. The receiver uses the microphone to record this sound. The signal is first parsed by the hail listener, which finds the starting point. Then, using the Fourier Transform (FFT) module, the malicious receiver can recover the data.

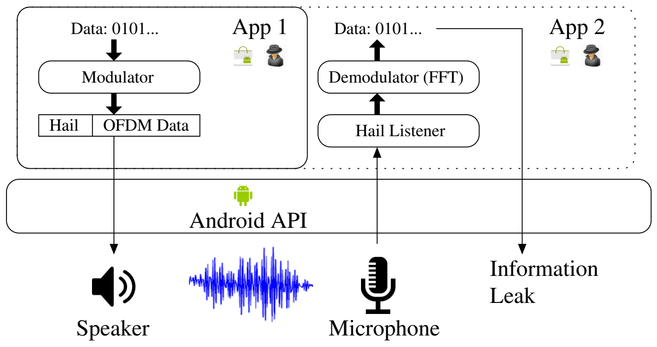


Figure 2. High level module view of the ultrasound covert channel.

We choose to use 18khz - 22kHz, because this spectrum is inaudible to humans (stealthy), below the Nyquist frequency for most smart devices, and has relatively low background noise [29]. In order to remain inaudible, but still achieve a high bit-rate, we choose to work in the frequency domain. We generate data segments 882 samples long, each of which is the sum of many modulated sub-carrier frequencies.

$$s_i = \sum_{f=18kHz}^{22kHz} sin(2 * \pi * i * \frac{f}{F_s} + \theta) : \forall i \in [1 - 882] \quad (1)$$

Each sub-carrier (of which there are 70, spaced 50Hz apart) encodes a binary phase value and a binary amplitude (19% or 100%). We reserve four sub-carriers for calibration (18k, 19k, 20k, 21k), which are used to reduce error when recovering bits. This affords a maximum theoretical throughput of roughly 6.5kbps, which is high compared to traditional covert channels, [41]. The receiver can use the FFT, to determine the amplitude and phase of each sub-carrier frequency present in a given packet $S = \{s_1, s_2, ..., s_i\}$. This is commonly referred to as orthogonal frequency division multiplexing (OFDM). When designing our system we used the following parameters. Packets of length $i = 882$ samples (20ms), $\Gamma = 50$Hz sub-carrier width, and sample rate $F_s = 44.1$kHz. We use 3.5kHz of spectrum from 18kHz to 21.5kHz.

Speaker and Accelerometer The speaker on most smart devices can cause the entire device to vibrate, if the tones are played with a loud enough volume. We use the speaker as a sender, playing standard Dual-Tone Multi-Frequency (DTMF) tones, and the accelerometer to measure the vibration of the phone, which will resonate with the tones being played. We then perform binary amplitude shift keying to modulate the data. DTMF codes are used because smart phones are designed to produce them well, and they are less conspicuous in this context. Permissions (1) and (4) are used. Although this channel is slower, and less stealthy than our ultrasound channel, we present it to demonstrate that seemingly arbitrary sensors can be combined to form a channel.

Vibration and Accelerometer Our vibration channel uses the vibration motor (6), normally used for silent notifications, as the sender, and the accelerometer (4) as the receiver. Again, binary amplitude shift keying is used for modulation. The stealth of this channel can be increased, by transmitting immediately after benign vibration events. In this way, the

	Permission	Hardware	Description
1	MODIFY_AUDIO_SETTINGS	Speaker	Change volume
2	CAPTURE_AUDIO_OUTPUT		Direct access to speaker buffer (e.g., record phone call)
3	RECORD_AUDIO	Microphone	Use microphone
4	android.hardware.sensor.accelerometer	Accelerometer	Use accelerometer
5	android.hardware.sensor.gyroscope	Gyroscope	Use gyroscope
6	VIBRATE	Vibration Motor	Use vibration motor
7	WAKE_LOCK	Screen	Prevent device from locking automatically
8	CAMERA	Camera	Take pictures
9	FLASHLIGHT	Camera Flash	Use lamp
10	READ_PHONE_STATE	Phone	Learn if phone is locked or a call is active
11	INTERNET		Open network sockets

Table 1. Summary of various permissions used by our covert channels

user may mistake the transmission for a long notification or may not notice at all.

Flash and Camera In this channel the camera's flash (9) is used to send data and the camera (8) is used to receive it. We simply turn the lamp on and off and run some simple image processing on the captured image preview from the camera to transmit data. The average brightness of the image should be much higher if the lamp is on, compared to when it is off. This channel has several draw-backs. Firstly, the camera can only be used by an application if a preview is shown in the foreground. Secondly, the camera light is very obvious and suspicious to even non-savvy users. Thirdly, this channel relies on some assumptions about physical orientation and environment. We show in our evaluation that the channel works even when the device is placed camera-down on a flat surface, which increases the practicality and stealthiness toward users.

User and Gyroscope The main idea of this covert channel is much different from the others. Here, we tilt the phone, and measure this action using the gyroscope (5). Bits are encoded in the angle of the device over time. To transmit the bits, we fool the user into tilting the device in the correct sequence by implementing a simple "endless running" game. In the "endless running" genre, an avatar moves down a track collecting items and avoiding obstacles. The user is tasked with tilting the device (or swiping) left and right to move the character on and off several different tracks. Endless running games typically feature randomly generated tracks, with the challenge being how long the user can avoid the obstacles. We simply generate the track according to the bit stream to be transmitted. This channel is highly stealthy in that the user is interacting directly with the device, and will have no idea the channel is being used. In fact, the user is part of the channel!

We make a novel generalization of this concept we call "user-sensor" covert channels, which fool the user into taking specific actions (touching areas of the screen, pressing hardware buttons, etc.), which act as data transmission symbols. User-sensor channels present many unique challenges such as handling inevitable user errors, and clever game design, which are out of the scope of this paper.

DEFENSE

The attacks presented in the first half of this paper demonstrate the need for a new robust defense mechanism to protect sensitive data stored on smart mobile devices. Because we propose PMCCs which can be built from several different physical interfaces, a prudent defense cannot apply blanket or coarse grain rules, such as prompting for user input when these interfaces are used, or blocking their use all together. Applying traditional defense schemes naively, like TaintDroid [9], or elaborate policy based security, would severely decrease the usability of the device, as the physical interfaces are commonly used benignly.

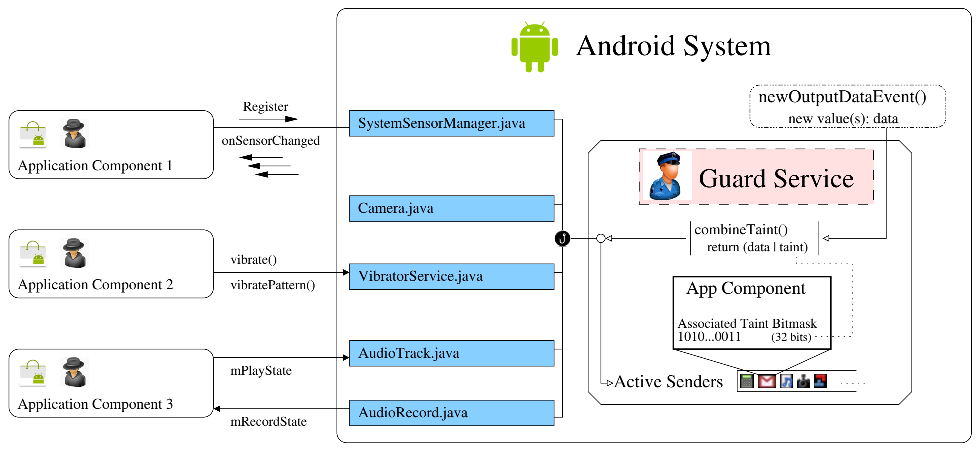
Our proposed defense system architecture, illustrated in Fig. 3, is comprised of two main components or stages. The *detection stage* aims to maintain an always up to date record, during run time, of which application components are using which potential covert channel devices. If two potential channel devices are being utilized simultaneously, we propagate the taint tag from the data on the sender side, to the data on the receiver side. Thereby extending the existing taint tracking solutions. Then, in the *treatment* stage, we provide some additional protections according to a configuration file.

Stage One - Detection

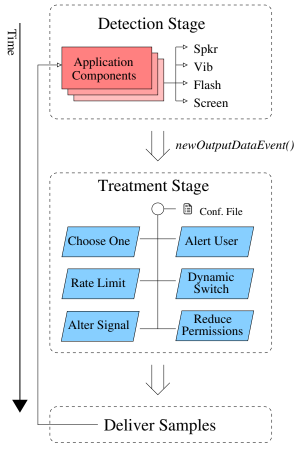
Fortunately, as illustrated in Fig. 1, any application that is trying to transmit using a covert channel, must use an API provided by the operating system to access the hardware device. In order to detect the API use, we add a system service called the "GuardService" to the operating system. The GuardService maintains an "active senders" list of components that are utilizing sender devices (e.g., the speaker). The elements in the "active senders" list are used to store the taint-tags of the data being transmitted. Because we extend the existing TaintDroid implementation, this taint information is a 32-bit bitmask.

The GuardService exposes three methods; *.add(component, device, tag)*, *.remove(component, device)*, and *.lookupTag(component, device)*. The first method adds a component, and a taint tag, to the "active senders" list, the second method removes components from the list, and the third returns the taint tag from the given component and device pair if it is in the list. We modify the Android system device APIs to automatically call these methods.

The Android device APIs are modified so that when a component begins using a sender device, the *.add()* method is called, and the taint-tag from the data flowing to the device is stored in the GuardService. The API is also modified so that samples flowing from receiver devices (e.g., the microphone) are intercepted. The *.lookup()* method is called in these places to retrieve the taint-tag from any possible corresponding and active sender devices in the "active senders" list. Then, the taint tag is propagated (bitwise OR as shown in combineTaint() in



(a) Defense System Architecture. Various system API classes are altered to hook into the "GuardService" Android system service. The GuardService detects potential channels and applies the configured treatment.



(b) Defense System Framework.

Figure 3. Architecture and Framework for our defense system. Applications that use sender devices are recorded. When there is a new data event for some hardware, the taint tag is propagated by our service, before the samples are delivered to the appropriate application component(s).

Fig. 3 (a)) to the new receiver device data. Finally, the API is modified to call the *.remove()* method when the sender devices are no longer being used.

When the malicious code uses a covert channel, the previous work breaks down. By doing this, our system allows the taint information to traverse from the sender side to the data on the receiver side, as depicted in Fig. 3. And, by using taint-tracking analysis, we can ignore benign instances when physical devices are being used, but there is no sensitive information flow.

Stage Two - Treatment

To improve on this, we present several potential treatment methods in a second stage. As illustrated in Fig. 3 (b), when a covert channel is detected, our system intervenes and reads a user controlled configuration file to determine how to act. Allowing different treatments is the crux of our framework, which allows us to include the best ideas from recent literature, as well as leave the system open for improvement in the future, when new covert channels are discovered.

Alert The User The weakest choice of treatment is for the system to alert the user of a possible information leak.

Choose One One device is turned off (the samples are dropped) for the duration the other device is in use. Which device is allowed and which is prohibited is a tunable parameter.

Dynamically Switch Allow only one device to operate transiently based on a simple threshold. For example, a video conferencing application may access the speaker and microphone concurrently. However, in typical conversation it is uncommon that both parties are actually speaking at the same time. Based on which signal is stronger (i.e. RMS for audio signals), we allow only one to flow, as illustrated in Fig. 4. By switching dynamically between the two sensors, we guarantee the covert channel cannot be utilized, because there is never a time when both sensors are active, but benign applications are still usable.

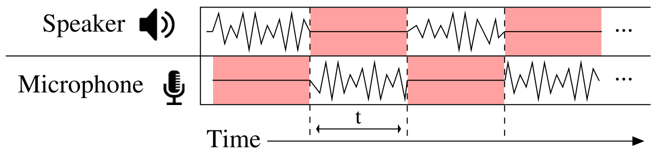


Figure 4. "Dynamically Switch" treatment method. The speaker and the microphone cannot be accessed at the same time. The active device is chosen in each time window by calculating which root mean squared (RMS) is larger. The other device is silenced.

Rate Limit For the flash and camera channel, the two devices must be used at the same time in order to take pictures in low light conditions. Instead of prohibiting access to one device, one possible solution is to limit the rate at which that device can be used.

Altering the Signal For all of these channels, we can increase the error rate the attacker achieves by changing the signal. For example, filtering out ultrasonic audio, removing samples to create random pauses, or inserting noise in the signal. However, we must be careful not to effect benign behavior. Clever approaches may be possible, but unfortunately, must be tailored specifically to the encoding scheme of the attacker, as well as the signal processing of the benign application(s).

Reduce Permissions There are several ways we can defeat the channel by altering the permissions of the application components involved. For example, each component can be re-assigned the *intersection* of the permissions of both temporarily.

Technical Challenges and Implementation Details

We implement a prototype of our defense scheme on Android by downloading and modifying the Android Open Source Project (AOSP) source code. For taint tracking, we extend the existing Taint-Droid system [9].

By default, none of the Android device API classes will notify our GuardService when devices are in use, which is necessary in order to propagate taint information over the covert channel. We modify the API for each device to hook into our "GuardService" accordingly. But, a trivial approach cannot be taken. Because the devices vary greatly, the source code for these classes is complex, and there are many potential pitfalls. In the following we analyze the API of each of the sender devices and detail how we implement the hook.

Flash For the camera and flash covert channel, we describe how to transmit data using the flash, but there are actually several parameters that can be used to encode data such as the image resolution, which are provided in a *Camera.Parameters* class. When the camera is opened, the *GuardService.add()* method is called and we propagate the taint tag of the entire class instance to the "active senders" list. *GuardService.lookup()* is called when a picture is taken or preview frames are delivered and *GuardService.remove()* is called when the camera is closed.

Speaker For covert channels using the speaker, the developer can implement the speaker API in one of two modes; a streaming mode and a static mode. In the static mode, the developer first instantiates an AudioTrack class instance, and writes any sound data using the *.write()* method, which writes an array to an internal buffer. Later, the developer calls a *.play()* method which instructs the hardware to actually play the signal previously written. Alternatively, the developer can choose to implement streaming mode, in which the developer calls the *.play()* method first, and then, after some time, the *.write()* method, which will instantly generate the sound.

Our system places *GuardService.add()* and starts a timer on one of these methods appropriately depending on the mode; *.play()* in static mode and *.write()* in streaming mode. We can estimate the duration that the speaker will be making noise based on the sample rate and the size of the array(s) passed to *.write()*. The timer, of the same duration, is used to call *GuardService.remove()*. A similar, timer-based, approach is taken for the vibration motor, which also provides an API method that returns immediately.

User and Game For this channel, the sender appears to be the user, but actually, the user responds to the information presented on the screen by the malicious game. Because the attacker has implemented a game for this channel, it is safe to assume that they will implement a canvas element to draw the game graphics. The Android canvas element exposes an *onDraw* method to the developer, which is called rapidly to update the current frame on the screen. We can propagate the taint tags from the variables used in *onDraw* that are used in helper functions such as *drawRect()* and *drawArc()*.

Limitations – Implicit Flow Taint Tracking

As we mentioned previously, we leverage the existing TaintDroid work for dynamic taint tracking analysis [9]. This system only tracks *explicit* flows, meaning direct assignments from one variable to another as shown in Fig. 5. However, in this case, it is very likely the attacker will use *implicit* flows to encode the sensitive information, as shown in Fig. 6.

```
s = api.getSensitiveInfo();
a = s;
a = s.getCharAt(3);
networkAPI.transmit(a);
```

Figure 5. Examples of explicit information flow.

```
signalSampleList = new List();
s = api.getSensitiveInfo();
for bit in s.toBinary(){
  if(bit == 0){
    signalSampleList.addSilence(500);
  }
  else{
    signalSampleList.addNoise(500);
  }
}
vibrationAPI.vibrate(signalSampleList);
```

Figure 6. Example of implicit information flow.

To overcome this challenge, we propose a method inspired by previous work [24]. We identify these implicit flows using static analysis and include implicit flow taint propagation rules following a simple heuristic: If a branch depends on a tainted value, then we should propagate the taint tag to the variables assigned inside the branch. Our modifications are done in three stages. First, at install time, the application is decompiled from the dex files to Java code. Then, we use static analysis to automatically find implicit flow blocks. We re-write the application code to propagate taint information into the variables in these blocks. Then, in the final stage, we recompile the application, re-generate the .apk file, and install it on the users device. To reduce false positives, we only apply our implicit flow propagation if the output variable data depends on the entirety of the sensitive input data.

This approach is not perfect, malware has a long history of resisting such techniques. We leave more robust automatic application re-writing, implicit flow taint tag propagation, and code block analysis to future work.

EVALUATION

In our system we implement five covert channels, which utilize physical mediums (e.g., sound, vibration). We implement all five covert channels as single user-space applications. The two application scenario, with one sender and one receiver will not have any effect on transmission speed, errors, or stealth, so we did not implement it. The ultrasound channel was evaluated on an LG-C800 smartphone. The speaker / accelerometer, vibration / accelerometer, and light / camera channels were tested on a Samsung Nexus S and the game / gyroscope channel was tested on a Google Nexus 4. The dynamic switching defense evaluation was done on the Nexus S and LG-C800. The taint tracking experiments were done with a Galaxy Nexus.

Covert Channels

To evaluate the ultrasound covert channel, we implemented an android application that generates ultra-sound packets, as described previously. We transmitted 1000 bits in twelve iterations and plotted the percentage of bits recovered incorrectly in Fig. 7. We can see that the total error percentage and phase

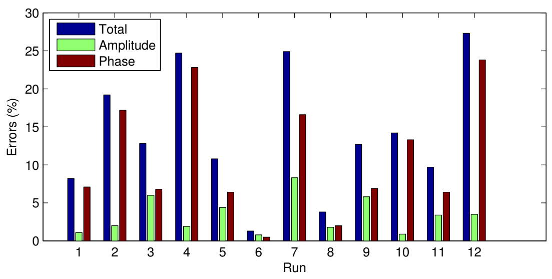


Figure 7. Ultrasound Bit Error Rate. Phase and Amplitude correspond to bit errors from demodulating these sub-carrier attributes respectively.

error percentage have high variance (6.98, and 6.01 respectively). Trial number six shows that, occasionally, the channel can achieve very low error rates. Amplitude error has lower variance (0.57) and shows a lower mean (3.3%). Using both phase and amplitude, we can achieve a bit rate of 6.5kbps with relatively high mean error (14%), and by utilizing amplitude only, we can achieve a lower rate of 3.25kbps that is more consistent.

We performed a short audibility user study to show that our ultrasound scheme is stealthy. A random string of 1000 bits was encoded and transmitted three times in the presence of ten individuals between the ages of twenty and thirty. The experiments were held in two different meetings, with a volume of $\frac{6}{16}$. Participants were not notified beforehand of the experiment and were questioned shortly afterwards about having heard anything. Unanimously, nobody was able to hear our system being used. Even after being told, and actively listening, untrained users are unable to hear the ultrasound emitted.

Speaker & Accelerometer We implemented the speaker and accelerometer covert channel described previously and we used it to transmit 1024 random bits, ten times, at a rate of 2bps to measure the bit error rate. Because we don't have robust synchronization, our demodulator sometimes inserts extra (incorrect) bits or drops bits. We count the total number of bits in error as the sum of the number of inserted, dropped, or incorrectly decoded (flipped) bits. Each type is only counted as one error, even though missing a bit will propagate errors through the rest of the bit stream. For this channel, the recovery is very good; we had only four bit errors (0.039%). Six of the runs transmitted with no errors at all.

Vibration Motor & Accelerometer To evaluate this channel we transmitted 1024 random bits ten times and recorded the bit errors, by type, in Fig. 8. This channel has a low error rate and achieves 2bps throughput. Similar to the Speaker and Accelerometer channel, we can see that there are only a few bits which were incorrectly decoded, the rest of the errors are due to improper synchronization.

Camera & Flash We transmitted 1024 bits using the channel ten times and measured the bit error rate and transmission time. The experiments were done in a dark room (similar to when the user may be sleeping), which can be seen in Fig 9.

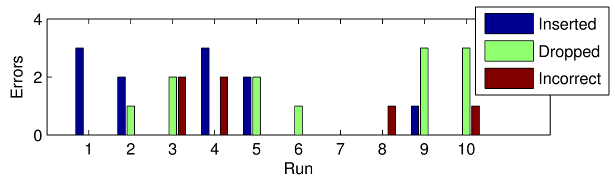
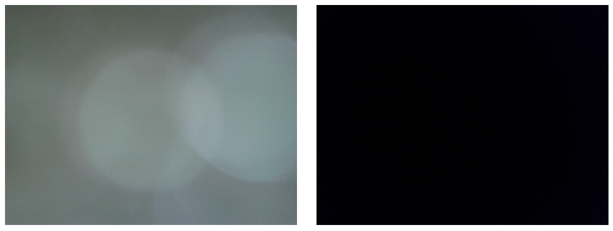


Figure 8. Vibration bit error types

Over all ten trials, there were no bit recovery errors and the transmission time was very consistent at 664 seconds (± 2). The bit rate we achieve is 1.5bps. The limiting factor here is the camera preview, which takes about one second and must be started and stopped each time to change the flash's state. We also performed a simple test to confirm that the camera



(a) Camera Down, Flash (b) Camera Down, No Flash

Figure 9. Camera images captured with and without flash. Images gathered when the phone was camera down, on a desk.

can be in different orientations and environments. We placed the phone flat on the ground, in a poorly lit area under a desk, held upright in a dark room at night, and flat, camera down on a desk (Fig. 9). We transmitted eight bits using the channel each time and we were able to recover all of them with no errors in every scenario. We can attribute this success to the physical design of camera hardware on the phone (Samsung Nexus S). There is a small gap between the actual sensor, and the back of the phone (they are not flush), which allows some light to travel from the flash to the camera lens, even when the camera is facing down on a table. This greatly improves the stealthyness of the channel, as the user will not notice the flash when blocked by the desk.

User / Gyroscope We implemented a simple game in which the user is instructed by a foreground application to rotate the phone in one of the three axes (6 different symbols). A background service measures the gyroscope at the same time to decode the bits. The user is tasked with finishing each rotation task in as short a time as possible. We played this game ten times and transmitted twenty symbols each time. On average, the user is able to rotate the phone in 1.02 seconds, making for a bit rate of roughly 2.5bps. There were no bit errors whatsoever.

Comparison In Table 2 we compare the average case error rate and speed of each covert channel. In order to remove transmission errors entirely, we can use hamming codes to correct single packet errors and retransmit packets containing more than one error [39]. We estimate the effective speed after hamming codes are applied in the "Eff. Speed" column.

Defense

Channel	Error	Speed	Eff. Speed
Speaker/Mic	14.4%	6.5kbps	3.71kbps
Spkr/Mic (Amp Only)	3.3%	3.25kbps	2.73kbps
Speaker/Accelerometer	0.04%	2bps	1.99bps
Vibration/Accelerometer	0.28%	2bps	1.94bps
Light/Camera	0%	1.5bps	1.5bps
User Game/Gyroscope	0%	2.5bps	2.5bps

Table 2. Comparing best case speed and average error rate of various covert channels before error correction coding (Speed) and after (Eff. Speed).

Channel	Delivery Time (no active channels)	Delivery Time (active channel)
Ultrasound	37.5ms	38.5ms
Speaker/Accel.	5ms	7.5ms
Vibration/Accel.	4.9ms	6.1ms
Light/Camera	8.36ms	10.1ms

Table 3. Overhead

For our defense system, effectiveness and usability are top concerns. Unfortunately, performing a robust effectiveness evaluation is difficult, because there is no known or cataloged malware in the wild that, to the best of our knowledge, takes advantage of PMCC as described in this paper. Therefore, we can only test the system on our own contrived examples, in all of these cases, the system was able to identify active covert channels, and propagate the taint-tag information. Our system currently breaks down in situations involving implicit flows. Currently, we do not have a robust solution for taint-tracking through implicit flows, but in the future, we plan to implement a better solution, at which time an effectiveness evaluation will be more interesting.

To measure the usability, we record the time it takes for the system to deliver new samples to the various devices (overhead), the time added by our application re-writing to support implicit flows (overhead), and the use-ability of benign applications under our dynamic switching treatment technique.

Taint Propagation

We implemented our defense by leveraging the existing TaintDroid implementation [9]. Therefore, the memory overhead is identical to their work. The stack is basically doubled in size, due to the extra space needed to store the taint values (except for arrays which share one taint-tag for the entire structure). However, when new samples are delivered, our system introduces some time overhead, because we must lookup and propagate (bitwise OR) the taint value from the sender data to the receiver data. We implemented the ultrasound, speaker + accelerometer, vibration + accelerometer, and the light + camera channels. We measured the average time for the *dispatchSensorEvent()* to finish delivering the data with and without our system running over ten runs. The results are presented in Table 3.

Implicit Flow Propagation

To measure the overhead introduced by our implicit flow taint propagation solution, we manually re-wrote a representative if statement in an Android app using the d2j decompiler, and the Google provided application packaging tools. The body of the if statement contains a for loop, similar to Fig. 6, a primitive variable assignment, an array element assignment, a function call, and a custom object construction. To ensure the taint values are propagated within the implicit flow code bodies, we multiply the to-be-tainted variables by a tainted variable with the value one. By doing this, we guarantee that the variable will receive the taint tag (through the direct assignment taint-droid rule), but also that the variable data will not by modified (multiplying by one has no effect). In 25 trial runs of this if statement, our re-written version, which propagates taint tags into all of this data, introduced only 1ms of overhead on average.

Dynamic Switching

To evaluate the effectiveness of this defense treatment, we implemented a very simple speaker / microphone covert channel. The speaker makes some audible sound and the microphone records this sound. Loud and quiet periods correspond to "1" and "0" bits respectively, with symbols 0.5s wide. We sent the bit sequence 01010011 twice; once using our defense and once without. We plotted the data the microphone measured in Fig 10. We can see that when the defense is running

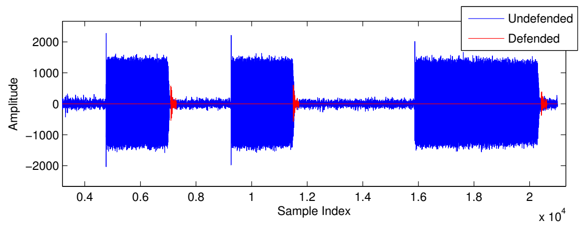


Figure 10. Samples gathered from the microphone while transmitting from the speaker. The data with the defense scheme running is plotted in red.

(plotted in red), the symbols are almost completely erased. The small tail at the end of each symbol is the result of the slow propagation time of sound and hardware introduced latency. We consider this problem to be negligible, because the tails are very brief, and conceal the transmission of consecutive "1s".

We also used a VoIP app (Skype), to measure usability. We instrumented our system on one phone and made a phone call to a second phone. Both users were able to hold a brief ten second conversation without any words being dropped or misunderstood.

CONCLUSION

In this paper we present an attack on mobile smart devices that leverages physical media covert channels to enable privilege escalation and ultimately leak sensitive user information. We also present a novel defense technique that balances usability with the security and privacy concerns raised by this attack.

ACKNOWLEDGMENTS

We would like to sincerely thank the anonymous, Ubicomp 2015 reviewers. Their insightful comments and feedback helped us shape this paper. The W&M team was supported in part by NSF grant CNS-1117412 and the CAREER Award CNS-0747108.

REFERENCES

1. Aafer, Y., Du, W., and Yin, H. Droidapiminer: Mining api level features for robust malware detection in android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)* (Sydney, Australia, September 25-27 2013).

2. Beresford, A. R., Rice, A., and Skehin, N. Mockdroid : trading privacy for application functionality on smartphones. *HotMobile '11 Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (2011), 49–54.

3. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards taming privilege-escalation attacks on android. *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (2012).

4. Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, USENIX Association (Berkeley, CA, USA, 2013), 131–146.

5. Chandra, S., Lin, Z., Kundu, A., and Khan, L. Towards a systematic study of the covert channel attacks in smartphones. *Proceedings of the 10th International Conference on Security and Privacy in Communications Networks, SecureComm* (2014).

6. Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. The tangled web of password reuse. *NDSS* (February 2014), 23–26.

7. Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An empirical study of cryptographic misuse in android applications. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (2013), 73–84.

8. Enck, W. Defending users against smartphone apps: techniques and future directions. *Proceedings of the 7th international conference on Information Systems Security (ICISS'11) 7093* (2011), 49–70.

9. Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, USENIX Association (Berkeley, CA, USA, 2010), 1–6.

10. Fragkaki, E., Bauer, L., Jia, L., and Swasey, D. Modeling and enhancing android's permission system. In *ESORICS*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*, Springer (2012), 1–18.

11. Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)* (2012).

12. Han, H., Liu, Y., Shen, G., Zhang, Y., and Li, Q. Dozyap: Power-efficient wi-fi tethering. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, ACM (New York, NY, USA, 2012), 421–434.

13. Han, H., Sheng, B., Tan, C., Li, Q., and Lu, S. A measurement based rogue ap detection scheme. In *INFOCOM 2009, IEEE* (April 2009), 1593–1601.

14. Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. Asm: A programmable interface for extending android security. In *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association (San Diego, CA, Aug 2014), 1005–1019.

15. Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. 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*, CCS '11, ACM (New York, NY, USA, 2011), 639–652.

16. Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. Android taint flow analysis for app sets. In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2014)* (June 2014). To appear.

17. Lampson, B. W. A note on the confinement problem. *Commun. ACM 16*, 10 (Oct. 1973), 613–615.

18. Lipner, S., Jaeger, T., and Zurko, M. E. Lessons from vax/svs for high-assurance vm systems. *Security Privacy, IEEE 10*, 6 (Nov 2012), 26–35.

19. Livshits, B., and Jung, J. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, USENIX Association (Berkeley, CA, USA, 2013), 113–130.

20. Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM (New York, NY, USA, 2012), 229–240.

21. Lunden, I. 6.1b smartphone users globally by 2020, overtaking basic fixed phone subscriptions. http://www.bloomberg.com/news/articles/2012-10-17/smartphones-in-use-surpass-1-billion-will-double-by-2015, June 2015.

22. Marforio, C., Francillon, A., Capkun, S., Capkun, S., and Capkun, S. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Tech. Rep. 724, Department of Computer Science, ETH Zurich, 2011.

23. Millen, J. K. Covert channel capacity. *IEEE Symposium on Security and Privacy 1987* (1987).

24. Min Gyung Kang, Stephen McCamant, P. P., and Song, D. Dta++: Dynamic taint analysis with targeted control-flow propagation. *NDSS'11* (2011).

25. Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in android. *2009 Annual Computer Security Applications Conference* (2009).

26. Orthacker, C., Teufl, P., Kraxberger, S., Lackner, G., Gissing, M., Marsalek, A., Leibetseder, J., and Prevenhueber, O. Android security permissions - can we trust them? In *MobiSec* (2011), 40–51.

27. Percoco, N. J., and Schutle, S. Adventures in bouncerland. https://www.youtube.com/watch?v=-Kcy-ldh5h0, July 2012.

28. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. *NDSS* (February 2014), 23–26.

29. Rajalakshmi Nandakumar, Krishna Kant Chintalapudi, V. N. P., and Venkatesan, R. Dhwani : Secure peer-to-peer acoustic nfc. In *Proceedings of ACM SIGCOMM*, ACM (New York, NY, USA, 2013).

30. Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing android sources and sinks. *NDSS* (February 2014), 23–26.

31. Rastogi, V., Chen, Y., and Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on 9*, 1 (Jan 2014), 99–108.

32. Ritzdorf, H. Analyzing covert channels on mobile devices. Master's thesis, ETH Zurich, 2012.

33. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. *2012 IEEE Symposium on Security and Privacy* (2012), 224–238.

34. Schlegel, R., Zhang, K., and Zhou, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)* (2011), 17–33.

35. Team, G. L. D. Go launcher ex application. https://play.google.com/store/apps/details?id=com.gau.go.launcherex&hl=en, May 2015.

36. Templeman, R., Rahman, Z., Crandall, D., and Kapadia, A. Placeraider: Virtual theft in physical spaces with smartphones. *NDSS* (Sept. 2012).

37. Wang, T., Lu, K., Lu, L., Chung, S., and Lee, W. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, USENIX Association (Berkeley, CA, USA, 2013), 559–572.

38. Wang, Y., Hariharan, S., Zhao, C., Liu, J., and Du, W. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, ACM (New York, NY, USA, 2014), 25–36.

39. "Wikipedia". Hamming code — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Hamming_code, November 2013.

40. Wu, C., Zhou, Y., Patel, K., Liang, Z., and Jiang, X. Airbag : Boosting smartphone resistance to malware infection. *NDSS* (February 2014), 23–26.

41. Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USENIX Association (Berkeley, CA, USA, 2012), 9–9.

42. Xu, F., Tan, C., Li, Q., Yan, G., and Wu, J. Designing a practical access point association protocol. In *INFOCOM, 2010 Proceedings IEEE* (March 2010), 1–9.

43. Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USENIX Association (Berkeley, CA, USA, 2012), 27–27.

44. yahoo. Yahoo weather application. https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.weather&hl=en, May 2015.

45. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, CCS '13, ACM (New York, NY, USA, 2013), 1043–1054.

46. Zhang, M. Appsealer : Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. *NDSS* (February 2014), 23–26.

47. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (November 2013).