An Android-Based Covert Channel Framework on Wearables Using Status Bar Notifications

Kyle Denney, A. Selcuk Uluagac, Hidayet Aksu, and Kemal Akkaya

Abstract Covert channels circumvent security measures to steal sensitive data undetectable to an onlooker. Traditionally, covert channels utilize global system resources or settings to send hidden messages. This chapter introduces covert channels and focuses on a novel covert channel on Android-based Internet of Things (IoT) devices. Particularly, we were able to make a covert channel using notifications a user gets from everyday applications. The chapter will also present this covert channel by discussing the framework, evaluating the performance, and demonstrating the functionality and flexibility of the proposed model.

1 Introduction

By 2020, it is estimated that there will be 50 to 100 billion devices connected to the Internet [8, 9, 18]. All of these devices traditionally communicate *overtly* using established communication protocols (i.e., TCP/IP), but it is possible to have these devices communicate *covertly* without detection.

Covert communication is the art of using typical communication standards and transforming them into a way that only two parties are aware of the message. This is different from encryption where the message is disguised in a way where an onlooker cannot discern the meaning. With covert channels, the onlooker is completely unaware the message exists in the first place.

This chapter will discuss how covert channels occur in computer systems. We start with a general overview of covert channels in Sect. 2. In Sect. 3, we discuss the timeline of covert channels and how they developed into their current state. Section 4 highlights a covert channel model we introduce that works on Android-based IoT systems, written in a way to show how covert channels may be developed in any computer system. Section 5 discusses the results of our proposed covert

K. Denney (✉) · A. S. Uluagac · H. Aksu · K. Akkaya
College of Electrical and Computer Engineering, Florida International Univeristy, Miami, FL, USA
e-mail: kdenn016@fiu.edu; suluagac@fiu.edu; haksu@fiu.edu; kakkaya@fiu.edu

© Springer Nature Switzerland AG 2018

M. Conti et al. (eds.), *Versatile Cybersecurity*, Advances in Information Security 72, https://doi.org/10.1007/978-3-319-97643-3_1

channel on a live testbed. We discuss possible ways to eliminate our covert channel in Sect. 6. Finally, Sect. 7 discusses where future development in covert channels may be headed–discussing both future attack models and future defenses to covert channels.

2 Background

Covert channels [2, 4, 16] are a way to steal sensitive information in manners undetectable to a third-party onlooker. Essentially, these channels abuse freely-available resources in a system to create novel means of communications. For instance, a *storage covert channel* [16] can alter values of a system resource using a predetermined codebook. A seperate process, knowing the codebook, can detect these changes and interpret the encoded message. This allows processes to work together to send messages surreptitiously within the system. The rest of this section will describe the history and traditional methodology behind covert channels.

2.1 Traditional Threat Model

Covert channels were first described by Lampson [13] to show how processes that were not intended for communication can be used to send hidden messages. Over time, various forms of covert channels were introduced [1–4, 16, 20]. This subsection introduces the concept of covert channel and describes how the different types of channels may be used by attackers in networks.

The process introduced by Simmons is a model we still use today to describe covert channels [19]. He describes a process similar to prisoners attempting to escape a prison. In this model, as shown in Fig. 1, Alice and Bob are two prisoners that plan to escape their confinement under the watch of a warden, Wendy. Alice and Bob must come up with a way to communicate without Wendy finding out. If Wendy discovers their communication, Alice and Bob are thrown into solitary confinement. In order to escape prison, Alice and Bob must communicate with messages that look innocent to Wendy, but actually contains information about escaping.

Fig. 1 Covert channel conceptual model [16]

This analogy can be translated to how covert channels work. Two applications must be able to send hidden information between each other without the warden, generally a firewall or antivirus software, discovering the communication. To hide the communication, covert channels generally use resources that are universally given to processes to use at will. Traditionally, there are two ways to communicate using these resources: *timing-based* or *storage-based*.

2.2 Types of Covert Channels

Timing-based covert channels communicate by using the time to send messages. A covert mechanism commonly used in these channels is the systematic delaying of packets being sent across the network. One covert sender will send packets across the network and delay them intentionally. The amount of the delay corresponds to a symbol that is part of the message. The covert receiver will detect these symbols and interpret the message.

On the other hand, *storage-based covert channels* work by altering values in system resources. Storage-based channels operate in intervals that are known between the two processes – typically a time with the least chance of user-interference. When the channel is operating, the covert sender process will change the value of a resource, like the volume setting, to correspond to a symbol in the message. The covert receiver process will read the changes in the values and interpret the message from there.

2.3 Motivations for Using Covert Channels

As with most security concepts, covert channels can be used offensively and defensively. For example, an attacker may establish a covert channel on a system in order to steal data undetected to the system administrators. Imagine the attacker implanting a covert channel on a military base: once established, he can use this covert channel to steal military secrets from the base without high-command knowing his presence.

On the other hand, covert channels are very important for defensive purposes. Take a military unit lost in enemy lands, for instance. Their entire mission can be compromised once the enemy knows of their presence. If the unit uses traditional communication methods to relay information, the enemy would be able to uncover their existence. Even if the unit uses encryption, the enemy could infer the intentions of their messages and detect the unit. However, through the use of covert channels, the unit will be able to remain undetected to the enemy while gaining crucial information from their commanders.

The fact that covert channels can be used for either purpose makes it especially important for research to be done in creating new covert channels and uncovering ways to mitigate existing covert channels.

3 Use Cases of Covert Channels

Covert channels are used throughout computer networks to send messages surreptitiously. Typically, if a system can connect to a network, a covert channel can be developed. In this section, we look at the fields that can be affected by covert channels and examples for each field. We start with the first covert channels and work our way up the timeline to covert channels on today's systems.

As mentioned before, covert channels were first introduced by Lampson in 1973 [13]. His work, "A Note on the Confinement Problem" focused on monolithic systems. He noted that it was possible to obtain information on who called an individual program by utilizing resources and system calls not designed for information sending. At the time, Lampson's work only defined covert channels operating on a single system or mainframe; using the channel to uncover information that was privileged and inaccessible to users. However, it quickly became understood that these covert channels could be used across multiple networked devices.

Once computer systems became networked together, covert channels increased their potential [20]. Researchers examined that it was possible to hide information in packet headers to discretely send information. The first instance of a timing channel was introduced by Girling [12] where he noted it was possible to systematically delay packets to hide information. From there, it quickly became realized that it is possible to create a covert channel on practically any networking protocol as one can either: (1) create a timing channel by systematically delaying the protocol packets or (2) create a storage channel by hiding information in the protocol packet headers.

Since a covert channel can be introduced on practically any networked device, they can have a potentially massive impact on the world. With 50 to 100 billion devices connected to the Internet in the near future [8, 9], the amount of data transferred in covert channels will be massive. For instance, the covert channel showcased in this chapter uses the notification class on Android used to notify a user of an incoming message from an application. Since notifications are used universally on all Android devices, this covert channel impacts the full spectrum of Android devices – including IoT devices that utilize the Android Wear operating system. As of May 2017, Google announced it has over 2 billion active Android devices running in the world [15]. This means the covert channel we discuss has a high potential to affect millions of users.

4 A Novel Covert Channel over Android-Based Notifications

In this section we introduce our proposed covert channel model. We show how it is possible to create a covert channel using the notifications that show on an Android system when one receives an SMS message or email.

Starting with an introduction to the Android operating system, we then move on to how the notifications themselves are created on an Android device. We showcase how one can manipulate these notifications to create both a timing and storage channel. Going step-by-step through the creation of both models, we also show how a covert channel may be developed on any system or protocol.

4.1 Android OS

The Android operating system is based on Linux, with much of the security designs and features being extensions of Linux. The main Android platform is a Java Virtual Machine running on top of the Linux kernel. What makes Android different from Linux is that each application running on Android is awarded the same privilege level – meaning that every application, including Google applications, have the same limitations and privileges.

When an application is running, it is given a unique user ID with unique privileges it can access from the system (these are awarded by the user in the form of permissions). For instance, applications that are given the Wi-Fi permission are allowed to use the phone's Internet connection – all others are barred from this access.

Since each application is its own user on the device, applications are not allowed to talk to each other directly. This technique is called *sandboxing* and is a typical way to increase the security of a device. Essentially, sandboxing works by giving each application a unique user ID. That user ID is then paired with its own set of permissions and system resources and then applications are not allowed to share information between each other. To get around this, additional permissions must be granted based on the type of communication the two applications would like to perform.

Covert channels can attempt to subvert these sandboxing methods [10, 11]. Applications may form either a timing-based or a storage-based covert channel with each other in order to share their own privileged information. One application can write information by altering a system resource while the second application can read that information as the resource is changed. Since system resources are universally given to applications, these attacks circumvent the security measures Android employs.

4.1.1 Android Notifications and Android Wear

The covert channel framework that is presented in this paper uses the *notification* mechanism available on Android-based wearables. Notifications are messages that are displayed to the user to give information about an application that is running. Examples of notifications would be a new SMS message, notice that a file finished downloading, or information about a new system update for the device itself. In this sub-section, the structure of a notification is described and the functionality of notifications across devices is introduced as it is an important concept to understand the covert channel technique introduced in this paper.

Traditionally, the status bar holds all of the notifications that appear on an Android smart phone. When an application needs to tell the user of something important, it creates a notification and displays it on the status bar. The basic structure of a notification includes a notification ID which is used by the smart device to distinguish different notifications, a title that displays at the top of a notification, and the text that gives more information to the user.

Starting in Android 4.3, support for Android wearables was introduced [5]. Using the Android Wear application, a smartphone can sync with a smartwatch. The two devices can then share information such as notifications between each other and make a more user-friendly experience by being more integrated with the user's life.

In order for notifications to be pushed from the phone to the wearable (or vice versa), a notification listening service on the wearable needs to be implemented. When the user gives the BIND_NOTIFICATION_LISTENER_SERVICE permission to an application, the application is allowed to access information about every notification that appears on the status bar. If the user has a wearable device synced with their smart device, the permission allows for all notifications across both devices to be read. *Since all notifications appear on both devices, it does not matter which device the application is installed on for it to read the notifications*.

Note that many applications on the Android marketplace today utilize this permission. Applications can sync with many devices and send notifications from a phone to other personal devices such as a computer or tablet. Moreover, with a simple search for 'notification reader' application on the Google Play Store, a myriad of applications appear that advertise a more user-friendly experience by sharing notifications across devices. Unfortunately, when an application advertises that devices can work together, the user is more inclined to give permissions such as the BIND_NOTIFICATION_LISTENER_SERVICE to the application.

4.2 Threat Model

With this information in mind, we can use these notifications in Android systems to develop a covert channel. First, we must establish the threat model we will be using for this covert channel. Imagine a rogue employee at a company who wishes to steal company secrets. He can sync his Android watch to his work tablet and establish

a covert sending application on the tablet and a covert receiving application on the watch. From there, he can make a covert channel that sends messages by hiding the data in notifications. The tablet can create notifications and the watch will read these notifications and infer the hidden data from within. Below we will describe methods to create this covert channel.

4.3 Covert Channel Framework

The covert channel that is introduced in this paper uses Android Wearable notifications to send information between applications on the same device, or even across multiple devices. We first describe the general framework, then introduce how we use the framework to make both storage-based and timing-based covert channels.

The first application we make is the covert sending application. The covert sender is responsible for creating notifications that will be used for the covert channel. It creates notifications with specific notification ID values that have pre-established meaning.

The second application, the covert receiver, is responsible for reading these notifications and determining the hidden message. This receiving application is assumed to have the BIND_NOTIFICATION_LISTENER_SERVICE permission enabled, allowing it to read notifications on the Android system. The receiver can read the notification ID values and infer the hidden message by comparing it to the pre-established codebook.

Below we describe how this can be done using both a timing-based and a storage-based covert channel methodology.

4.3.1 Previous Work

We have previously published works with this covert channel framework [6, 7]. In past works, our framework was only a storage channel framework. In this chapter, we expand on the framework to function as both a timing and storage channel. Additionally, we improve on the storage channel model to have a higher throughput, which we will analyze in later sections.

4.3.2 Timing-Based Framework

Here we describe how a timing-based covert channel can be achieved using notifications.

In the timing-based framework, the sending and receiving applications first have to establish a timing codebook. For our purposes, we established the receiver to look for a notification at every t intervals. We also created 3 notification ID values to be read: $start$, $stop$, and $message$. These ID values are arbitrary as long as both applications know what the corresponding values are.

To establish the covert communication, the sending application creates a notification with the *start* value. When the notification is read by the receiving application, it starts a timer that reads for notifications every t intervals.

The sender, with a timer of its own, then takes the message it wants to send and converts it into its hexadecimal data. The sender then reads the current data value in the stream and waits for the appropriate t intervals (e.g., if the data value is A, the sender waits 10 t intervals; the sender also waits a full 16 t intervals if the data value is 0). When the sender has waited the appropriate time, it creates a notification with the *message* value and then moves on to the next data value in the stream and repeats the process.

When the receiving application reads the *start* notification, the application starts looking for notifications with the *message* value every t intervals. If the application does not detect a notification, it increases a counter. This counter counts up until either a notification is finally read or the counter counts up to 16. The value of the counter whenever it is reset is added to the receiving application's message stream.

When the message is complete, the sending application creates a notification with the *stop* value. The receiving application reads this notification and stops its timer and counter. The receiver then stores the entire message it collected and the covert message is complete.

Figure 2 describes the covert channel model for the timing channel framework. The sending application creates notifications with specific time intervals in between. Then, the receiving application reads these altering timing values to interpret the hidden message.

4.3.3 Storage-Based Framework

The framework for a storage-based covert channel is similar to the timing-based framework. Here, we again use a *start* and *stop* notification to initialize and terminate the covert message respectively. However, this time, the *message* notification varies in value.

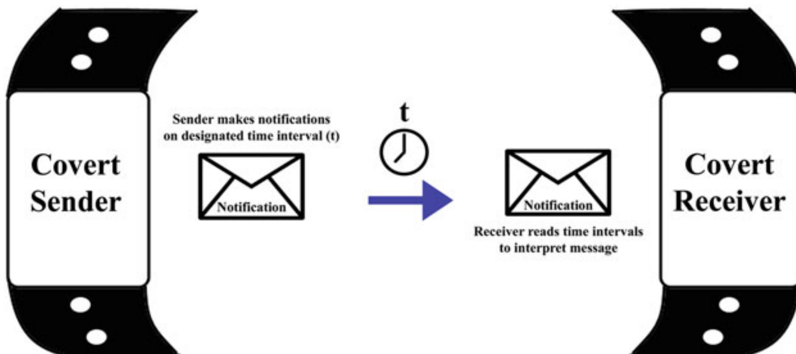


Fig. 2 Design of the timing-based covert channel

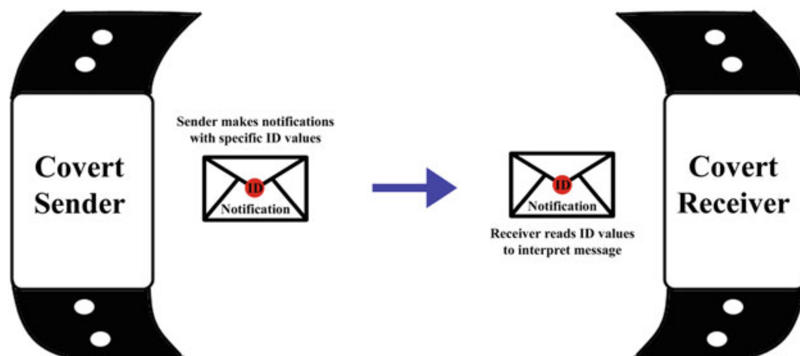


Fig. 3 Design of the storage-based covert channel

Once the covert message has started, the covert sender reads the current value in the message stream and inputs that value directly into the *message* ID value. The receiver then reads the value of the notification and adds it to its message stream.

Figure 3 describes the covert channel model for the storage channel framework. The sending application creates notifications with ID values corresponding to the predetermined codebook. Then, the receiving application reads these altering ID values to interpret the hidden message.

5 Success of the Covert Channel

In this section, we demonstrate the feasibility and functionality of the novel wearable covert channel framework and evaluate its performance on real wearable devices focusing on the two covert channel techniques introduced in the framework.

In our model, a rogue employee (Bob) may use this covert channel set up to steal sensitive data from a work phone by using a paired smart watch. Bob can install the covert sender on his company phone and the covert receiver on the watch. Bob can use the covert sender to encode the sensitive data into either notification-based model–creating notifications on his work phone. The Android Wear application will then send any notifications made on Bob's work phone to his paired smart watch. The smart watch will then receive any notifications sent from the Android Wear application.

By installing the covert receiver on the smart watch, it is possible for Bob to listen to all the incoming notifications on the smart watch. The covert receiver then picks up the message originating from the covert sender on the work phone, storing the sensitive data on the smart watch without the company's knowledge.

To test the functionality and the feasibility of this covert channel model, a Samsung Galaxy S5 was used as the work phone and a Sony SmartWatch 3 was used as a wearable device. The Galaxy phone holds a 2.5 GHz quad-core processor with 2 GB of RAM and the watch a 1.2 GHz quad-core processor and 512 MB of RAM. The devices used in our testbed are shown in Fig. 4.

Fig. 4 First covert channel technique testbed (left: Samsung Galaxy S5; right: Sony SmartWatch 3)

5.1 Throughput Analysis

In this subsection, we analyze the throughput of each covert channel model. By calculating how quickly notifications can be sent, we will be able to calculate how much data can be sent by both models. From there, we can determine which model has a higher throughput by comparing the two results.

5.1.1 Timing-Based Throughput

To have a high throughput, we need as low of a time interval between notifications sent as possible. To measure this, we first need to calculate how quickly the phone can make notifications as well as how fast the watch can read incoming notifications. From there, we can calculate the shortest time interval notifications can be sent at where the message is still discernible.

First, we measured how quickly the phone can make new notifications. We found that our phone could make about 19 notifications per second. To test how quickly the watch could read notifications, we had the phone repeatedly send notifications as quickly as it could (19 notifications per second) for an extended period. On the watch end, we simply made a counter that incremented every time it processed a notification from the phone. The ratio of how many notifications the watch processed to how many notifications were actually made gives us how many notifications per second the watch is capable of receiving. In our case, we found the watch was able to process just over 6 notifications per second, which we round down to 6 to ensure every notification is read during transmission.

By inverting our ratio of 6 notifications per second, we find that the lowest time interval between two notifications being processed is 1/6th of a second. That means our t value for transmission is also 1/6th of a second.

To calculate our final throughput, we assume the slowest possible message sent (a series of straight zeros) making new data being sent every 2.6 s. Dividing the amount of data sent (16 bits) by the maximum sending rate (2.6 s) gives us our final throughput of 6.15 bps. As Claudio et al. discuss [14], even a low bit rate covert channel is enough to share “reasonable amounts of data on the smartphone.” Given the timing channels they analyze are 3.70 and 4.88 bps, we can conclude that ours at 6.15 bps is also sufficient.

5.1.2 Storage-Based Throughput

The throughput of the storage-based model is much easier to calculate. Here, we had to simply take how quickly notifications are being sent by how much data can be sent per notification.

We have our speed of 6 notifications per second from analyzing the throughput of the timing-based model before, so we only have to determine how much data can be sent per notification. Notification ID values use the unsigned integer type, making each notification sent worth 4 bytes of data. This gives us around 24 bytes per second (192 bps) for the throughput of the storage-based model; exponentially faster than the timing-based model.

5.2 Covert Analysis

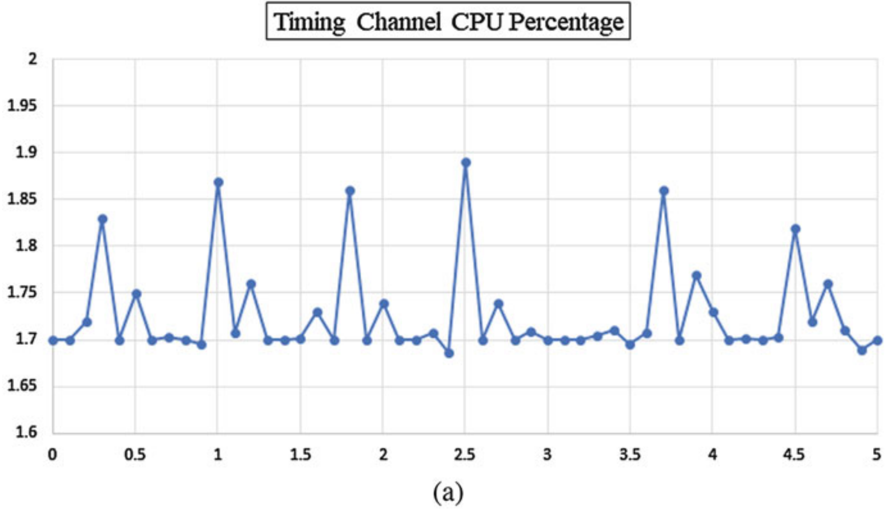
In this subsection, we discuss how covert each model actually is. We do this by analyzing how much system resources each model takes up as well as having a brief discussion on how easily an onlooker may uncover the communication through simple observation.

5.2.1 Pattern Recognition

To hide the communication occurring in our proposed covert channel, we use the *cancel()* function provided to notification creation. This function allows an application to remove a notification it created from the status bar of Android device. We call this function immediately after the covert sender makes a notification for transmission to ensure that an onlooker cannot physically see the notifications appear on the smart device. However, if one looks at the system logs, all the notifications still appear and communication can continue unaltered.

Since the notifications are still present on the system, it is still possible for a warden-type application to detect the covert communication by analyzing incoming notifications on the smart device.

To counter this warden, we can implement measures to mask the patterns one can discern from the covert communications. For instance, instead of the storage model sending notifications at a set pace, we can randomize the time between notifications as that will have no impact on the transmission. For the timing model, we can randomize the time interval after every transmission. Essentially, any measure that can change the transmission pattern without affecting the actual transmission can be introduced to our model to prevent a warden from recognizing the communication occurring.



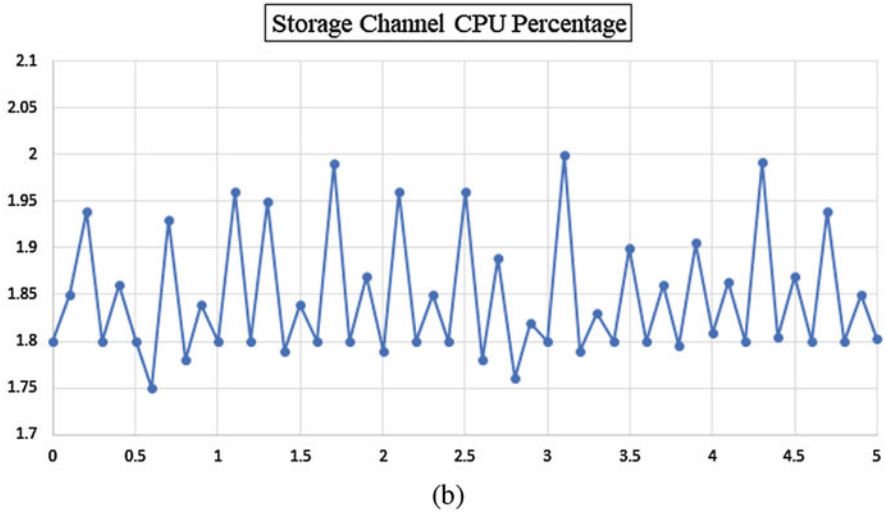


Fig. 5 CPU readings of (**a**) Timing and (**b**) Storage covert channel models

5.2.2 CPU Usage

We set up a simple test to see if there is any noticeable CPU usage from the covert sender. We monitored the CPU usage on our Galaxy device from the covert sender while the sender relayed a message to the covert receiver. Figure 5 has the CPU percentages of the covert sending application for both the timing and storage channel models (with A referring to the timing model and B referring to the storage

Table 1 Analysis summary of storage and timing channels

	Timing	Storage
Throughput	6.15 bps	192 bps
Data per notification	1 bit	4 bytes
CPU usage (average)	1.75%	1.85%

model, respectively). For simplicity's sake, we set the notification rate to 1/5th of a second and set the length of the test over 5 s to read the patterns more clearly.

As expected, the CPU usage spikes whenever a notification is created. This can be clearly seen in the timing model, as there is a heartbeat pattern whenever a notification is created: a large spike for the creation of the notification and a smaller spike for when the *cancel()* function is called to wipe the notification from the phone. While not as clear, this is still present in the storage model.

5.3 Analysis Summary

To conclude the analysis portion of our work, we include a table to compare and contrast the efficacy of both the timing and storage channel frameworks. Table 1 compares the two frameworks in their throughput, amount of data encapsulated in each notification, and average CPU usage respectively.

6 Discussion and Prevention

In this section, we discuss the pros and cons of both covert channel models. We also discuss methods to prevent the proposed covert channels and how effective these prevention models can be.

6.1 Prevention

In fact, there are no current solutions to prevent this proposed covert channel. However, this chapter proposes solutions that diminish, or stop entirely, the success of this covert channel.

A simple way to stop the proposed storage channel is to change the way the notification class handles notification ID values. Instead of allowing an application to create its own ID value for a notification, Android OS should randomly assign an ID value to every created notification. The new notification would still be attributed to the appropriate application, but it prevents this covert channel from operating properly.

However, this does not stop the timing channel from operating. Another solution to prevent both models is through simple intrusion detection. An antimalware software installed on an Android device could be programmed to detect newly created notifications. If one application is creating an unreasonable number of notifications, it would be flagged as a rogue application and proper authorities would be notified.

6.2 Comparison

Overall, the proposed storage channel model is a more well-rounded model compared to the timing channel. Below, we discuss the various criteria we used to come to this conclusion.

In our throughput analysis, we established that the storage channel is capable of sending more information at once. At its maximum throughput, the timing channel was only capable of 6.15 b/s, which is minuscule compared to the storage channel at 20 B/s. From the CPU usage section above, we established that it is simple to spot the pattern of an application sending repeated notifications over time. The timing channel model especially falls prey to this pattern recognition as there is no simple way to break up a system dependent on time. For the storage channel, we can simply send the notifications over a longer span of time, thereby making it harder for an onlooker to discern a pattern. However, the timing channel has more potential for longevity. It is rather simple for Google to shutdown the capability for the storage channel to operate by forcing notifications to have randomized ID values. There is no similar option for Google to take to stop the timing channel, as notifications will always have to be sent at times specified by the calling application. Therefore, the operability of the timing channel is practically ensured in all future Android releases.

7 Trends and Future Work

Since covert channels can exist on practically any computer system, we need to know what new technologies can be used to establish covert channels and what we can do to prevent them from occurring. This section will look at what new research in covert channels may look like in the near future.

7.1 Imminent Threats

As noted, practically any new technology that can be networked can be used to create covert channels. However, we note that current popular technology trends will have the most focus for covert channel development.

The covert channel we showcase only works on Android devices at the moment, but notifications exist on practically all modern devices from smart phones, to personal computers, to IoT devices. In order to improve this covert channel, one would need to develop a universal system that allows the covert channel to work on notifications from all of these devices.

As the case with our covert channel model, more covert channels will be introduced in IoT technologies. Similarly, we believe that covert channels in Cyber-Physical Systems (CPS) will be especially popular due to the nature of a CPS (e.g., stealing information about the energy grid undetected would be especially valuable).

Another research area – especially with its increased popularity in recent months – is uncovering covert channels in Bitcoin and other altcoins. Due to the history of Bitcoin, it is likely that covert channels are currently well-established to communicate in the criminal undergrounds. Research here would be done to prove how these covert channels can operate in order to work on the harder problem: detecting and diminishing the covert channel.

7.2 Future Defenses

Typically, once a covert channel is detected, it is very simple to stop it from occurring. One can simply add a detection system that flags when covert communications are likely happening and stop the process that is operating on the covert channel. However, future research can be done to help automate the detection instead of relying on manual eradication.

As mentioned earlier, covert channels are likely well-established in the Bitcoin network. The real challenge would be detecting and eradicating covert channels currently in place. Due to the nature of Bitcoin and how anonymous each transaction is, it would be more difficult than a traditional network to detect a covert channel in place. Research in this area would be finding ways to determine patterns in blockchain transactions in order to detect possible storage or timing channels in the blockchain network.

Neural networks and machine learning techniques can be used to determine typical covert channel patterns and then implemented on computer systems to automatically flag potential covert channels. Work on this concept was introduced by Shrestha et al. [17] to automatically detect timing channels, but this idea may be extended to include storage channels.

8 Conclusion

Covert channels are an effective means to send information undetected. They have widespread use throughout the Internet and can be used in a variety of ways. As more and more technologies are introduced, the need to understand how covert channels operate is important in order to prevent information from being stolen in these new technologies.

This chapter introduced covert channels and how they are traditionally implemented. We then showcased a model for a timing and storage channel in Android-based IoT systems that work on real and current implementations. We conclude this chapter with discussing where future trends in covert channel development and prevention are headed.

Acknowledgements This work is partially supported by the US National Science Foundation (Awards: NSF-CAREER-CNS-1453647, NSF-1663051, REU-CNS-1461119). The views in this document are of the authors, not of the funding agencies.

References

1. Al-Haiqi, A., Ismail, M., Nordin, R.: A new sensors-based covert channel on android. The Scientific World **2014** (2014)
2. Ambrosin, M., Conti, M., Gasti, P., Tsudik, G.: Covert ephemeral communication in named data networking. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14 (2014)
3. Caviglione, L., Gaggero, M., Lalande, J.F., Mazurczyk, W., Urbanski, M.: Seeing the unseen: Revealing mobile malware hidden communications via energy consumption and artificial intelligence. IEEE Transactions on Information Forensics and Security **11**(4), 799–810 (2016). DOI 10.1109/TIFS.2015.2510825
4. Chandra, S., Lin, Z., Kundu, A., Khan, L.: Towards a systematic study of the covert channel attacks in smartphones. SECURECOMM 2014 (2014)
5. Comstock, J.: 1 in 5 americans owns a wearable, 1 in 10 wears them daily. Mobi Health News (2014). URL http://mobihealthnews.com/37543/pwc-1-in-5-americans/-owns-a-wearable-1-in-10-wears-them-daily/
6. Denney, K., Uluagac, A.S., Akkaya, K., Bhansali, S.: A novel storage covert channel on wearable devices using status bar notifications. In: 2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC), pp. 845–848 (2016). DOI 10.1109/CCNC.2016.7444898
7. Denney, K., Uluagac, A.S., Akkaya, K., Saputro, N.: Demonstration of a novel storage covert channel on android smartwatch using status bar notifications
8. Drucker, P.F.: Internet of Things position paper on standardization for IoT technologies (2015). URL http://www.internet-of-things-research.eu/pdf/IERC_Position_Paper_IoT_Standardization_Final.pdf
9. Evans, D.: The Internet of Things: How the next evolution of the Internet is changing everything (2011). URL https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf
10. Gasior, W., Yang, L.: Exploring covert channel in android platform. In: Cyber Security (CyberSecurity), 2012 International Conference on (2012)

11. Gasior, W.C., Yang, L.: Network covert channels on the android platform. Cyber Security and Information Intelligence Research 2011 (2011)
12. Girling, C.G.: Covert channels in lan's. IEEE Transactions on Software Engineering **SE-13**(2), 292–296 (1987). DOI 10.1109/TSE.1987.233153
13. Lampson, B.W.: A note on the confinement problem. Communications of the ACM **16** (1973)
14. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51–60. ACM (2012)
15. Popper, B.: Google announces over 2 billion monthly active devices on android (2017). URL https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users
16. Radhakrishnan, S., Uluagac, A., Beyah, R.: Realizing an 802.11-based covert timing channel using off-the-shelf wireless cards. In: Global Communications Conference (GLOBECOM), 2013 IEEE, pp. 722–728 (2013)
17. Shrestha, P.L., Hempel, M., Rezaei, F., Sharif, H.: A support vector machine-based framework for detection of covert timing channels. IEEE Transactions on Dependable and Secure Computing **13**(2), 274–283 (2016). DOI 10.1109/TDSC.2015.2423680
18. Sikder, A.K., Aksu, H., Uluagac, A.S.: 6thsense: A context-aware sensor-based attack detector for smart devices. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 397–414. Vancouver, BC (2017)
19. Simmons, G.J.: The prisoners' problem and the subliminal channels. In: D. Chaum (ed.) Advances in Cryptology, pp. 51–67. Springer US (1984)
20. Zander, S., Armitage, G., Branch, P.: A survey of covert channels and countermeasures in computer network protocols. IEEE Communications Surveys Tutorials **9**(3), 44–57 (2007). DOI 10.1109/COMST.2007.4317620