computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

AN EXTENDED DISCUSSION ON A HIGH-CAPACITY COVERT CHANNEL FOR THE ANDROID OPERATING SYSTEM

Timothy Heard [1)], Daryl Johnson [2)]

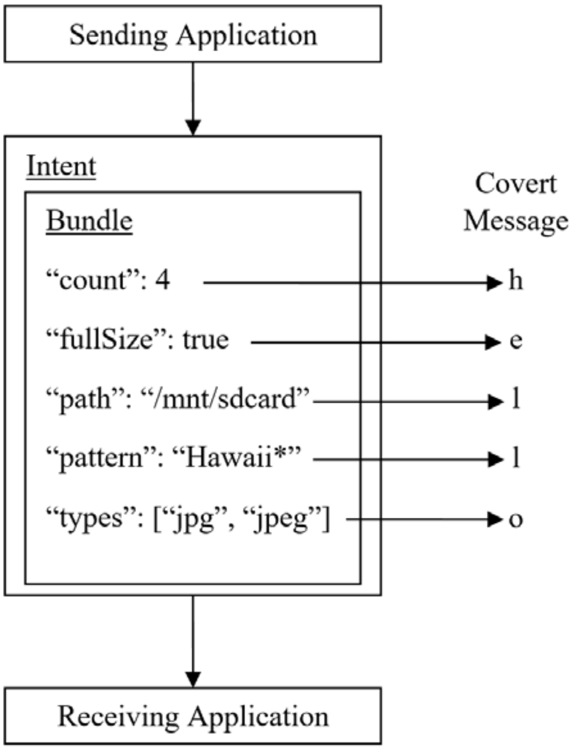
[1)] tjh2430@rit.edu
[2)] daryl.johnson@rit.edu

Abstract: In “Exploring a High-Capacity Covert Channel for the Android Operating System” [1], a covert channel for communicating between different applications on the Android operating system was introduced and evaluated. This covert channel proved to be capable of a much higher throughput than any other comparable channels which had been explored previously. This article will expand on the work which was started in [1]. Specifically, further improvements on the initial covert channel concept will be detailed and their impact with regards to channel throughput will be evaluated. In addition, a new protocol for managing connections and communications between collaborating applications purely using this channel will be defined and explored. A number of different potential mechanisms and techniques for detecting the presence and use of this covert channel will also be described and discussed, including possible counter-measures which could be implemented.

Keywords: Android, covert channel, mobile security.

1. INTRODUCTION

“Exploring a High-Capacity Covert Channel for the Android Operating System” [1] introduced a new covert channel for inter-application communication on Android which had a higher bandwidth than other similar covert channels which had previously been explored [2]–[9]. This channel is capable of encoding messages being used for some form of malicious activity within apparently legitimate traffic[1] between applications by using the Android Intent messaging framework, which is a foundational part of the Android application architecture [10]. Specifically, the channel takes advantage of a special key-value store which can be included within these Intent messages (called a Bundle [11], [12]) by imposing significance on the types of values stored combined with a pre-arranged key ordering as shown in Fig. 1.



The primary use of covert channels such as this one (i.e. covert channels for communicating discreetly between different Android applications) is to perform an application collusion attack [2]. This type of attack allows multiple applications to effectively pool their different permissions which have been approved by the user[2], in order to create dangerous combinations of permissions without the user approving, or even being aware of, that

[1] Or even actually legitimate traffic.

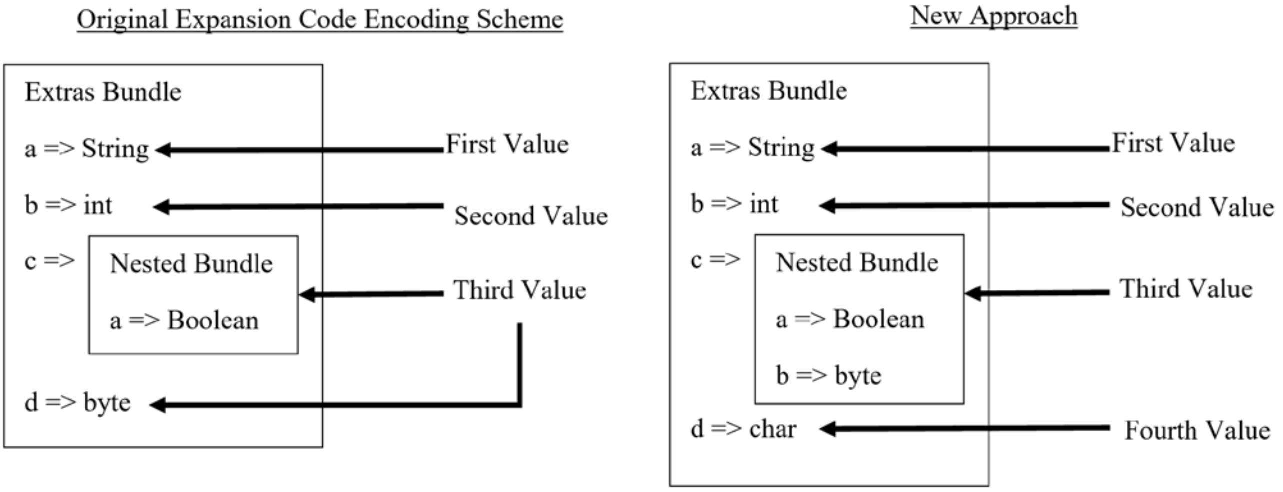
[2] Android requires users to approve the set of permissions required by an application before that application can be installed.

combination. For example, combining the permissions for a photo viewing application and an internet-enabled game would allow a user's pictures to be sent off of the device to an attacker.

Fig. 1 – Covert message encoded in Intent Bundle.

Although this channel achieved a much higher bandwidth (with regards to the speed of transmitting covert message bits) than other similar covert channels, the bandwidth can be further improved, which will be detailed in Section 1. The subsequent section will introduce a dynamic protocol for communication with colluding application which uses these improvements. Section 3 will then discuss different possible approaches for detecting this channel as well as some potential counter-measures which could further reduce the detectability of the channel, followed by a conclusion.

Fig. 1 – New expansion code encoding strategy.



2. INCREASING CHANNEL BANDWIDTH

The original implementation [1] increased the "alphabet" (i.e. set of unique symbols) available through the use of nested Bundles which were used to represent expansion code values. However, only one expansion code was used out of the 21 expansion codes which are possible when using a single nested Bundle containing a single entry. Additionally, only six of the 21 possible values in this expansion set were used, leaving a significant opportunity for improvement.

In order to make full use of the available value space, the encoding scheme used for this channel had to be modified. Specifically, the original static mapping of specific textual characters to the different available Bundle-value data types was abandoned and replaced by a conversion of message data into a single bit-string. These message bits were then divided into a series of "message fragments" (collections of bits) which were individually encoded onto the channel carrier (an Android Intent). With a set of 21 expansion codes, 22 unique sets of symbol/ message fragments were available (the base set of values plus 21 expansion sets), yielding a new alphabet containing 462 unique values (i.e. bit fragments), a more than 17-fold increase over the original 27-character alphabet.

The capacity of the channel was further increased by modifying the approach used for encoding expansion code values. In the original implementation, expansion codes were encoded into a nested Bundle using the same datatype-to-value mapping used for regular data values. That expansion code value was then applied to the next entry in order within the root Bundle. By encoding the value that the expansion code applies to within the same nested Bundle, an additional expansion code value can be achieved. This was accomplished by using an implicit expansion code of one for nested Bundles containing a single key-value pairing (see Fig. 2). This minor modification yields to total of 22 expansion codes and 23 value sets (each containing 21 values) for a total of 483 unique values.

This increased symbol set allowed for the encoding of one 8- to 9-bit message fragment per key-value pairing (within the root Bundle). Specifically, the binary values 00000000 through 11111111 and 100000000 through 111100010 were able to be encoded as message fragments[3]. Table 1 shows an example of this.

While this approach provides a potential increase

[3] 0 through 482 in decimal

in channel bandwidth, the “jaggedness” of the message fragments (i.e. the fact that message fragments do not have a uniform bit length) increases the complexity involved in reconstructing the contents of the covert message. The primary problem with this is that, as currently described, if the final fragment for a given covert message contains fewer than 8 bits it cannot be effectively represented. The most obvious solution is to pad the final fragment with zeroes, which can be seen in Table 1. However, doing so introduces a new problem: the receiver is unable to differentiate between zero bits used for padding and significant zero bits in the last fragment.

In order to solve this problem, when encoding a covert message onto an Intent Bundle, the first value encoded should be the number of significant bits contained in the last fragment of the message. Using this approach, the receiver would be able to reconstruct the exact bit sequence which was sent while maximizing the use of the available value space.

Table 1. Message Fragmentation Example.

Message	Message Bytes	Message Fragments	Expansion Code	Base Value
´	11000010	110000101	18	11
	10110100	01101000	4	20
h	01101000	110100001	19	18
i	01101001	101001110	15	19
´	11000010	00010101	1	0
	10110100	101000000	15	5

The original channel can be further improved by adding an additional dimension: multiple Intent actions. As mentioned in [2], the action field within an Intent can be used to implement a similar covert channel, with different pre-defined action strings mapping to specific values (0 and 1 for example). By combining these two covert channel concepts, a number of further enhancements are made possible.

First, this allows for an even larger alphabet since each action string could be mapped to a distinct set of 483 values[4]. Therefore, the total number of unique values possible for a single Bundle entry would be given by the following equation:

$$v = a * (e + 1) * b, \qquad (1)$$

where ‘a’ is the number of different actions available for sending data, ‘e’ is the number of expansion codes, ‘b’ is the number of values in the base value set, and ‘v’ is the total number of unique values which can be represented. An example of this encoding scheme is shown in Table 2.

An important consideration with such an approach is that the action strings used should not be

Table 2. Message Segmentation with Multiple Data Actions

*a = 4, e = 22, b = 21, v = 4 * (22 +1) * 21 = 1932 unique values*				
Message	**Bytes**	**Fragments**	**Segments**	
´	11000010	"A" => 11000010101	Action = "data-0"	"C" => 0001101000
	10110100	"B" => 10100010101	Min Value = 0	"D" => 0110010100
T	01010100	"C" => 0001101000	Max Value = 482	"F" => 0010111010
h	01101000	"D" => 0110010100		"I" => 0110010000
e	01100101	"E" => 10000001110		"J" => 0011000100
	00100000	"F" => 0010111010		"N" => 0100000011
q	01110001	"G" => 10110100101		"O" => 0011001101
u	01110101	"H" => 10001101101		
i	01101001	"I" => 0110010000	Action = "data-1"	No Values in Range
c	01100011	"J" => 0011000100	Min Value = 483	
k	01101011	"K" => 11100100110	Max Value = 965	
	00100000	"L" => 1111011101		
b	01100010	"M" => 11011011100	Action = "data-2"	"B" => 10100010101
r	01110010	"N" => 0100000011	Min Value = 966	"R" => 10100000000
o	01101111	"O" => 0011001101	Max Value = 1448	"E" => 10000001110
w	01110111	"P" => 11101111000		"G" => 10110100101
n	01101110	"Q" => 11000010101		"H" => 10001101101
	00100000	"R" => 10100000000		"L" => 1111011101
f	01100110			
o	01101111		Action = "data-3"	"P" => 11101111000
x	01111000		Min Value = 1449	"A" => 11000010101
´	11000010		Max Value = 1931	"Q" => 11000010101
	10110100			"K" => 11100100110
				"M" => 11011011100

[4] Assuming the use of the 22 expansion codes and a base set of 21 values as described above.

ones which will cause a user-visible event to occur (such as ACTION_IMAGE_CAPTURE [13]). It is also worth noting that this does introduce additional complexities since values which span the different action-based value sets must be encoded in separate Intents, which could result in higher message fragmentation and require a larger number of Intents to send a given message. However, in certain circumstances sending smaller Intents may be more desirable if it is a better fit for the expected behavior of the involved applications.

The second, and potentially greater benefit of this encoding strategy is the natural support for distinct data and control channels, which can be accomplished by designating one action string for sending control messages and one or more other actions for sending data-carrying Intents. An important consideration with this approach is that there needs to be a way to ensure the proper ordering of the different message segments (i.e. Intents) and also to indicate when all of the segments for the current message have been received. This can be accomplished by reserving two additional Bundle key positions to serve as metadata fields: one for the number of segments contained in the current message and another for the segment number of the current Intent.

By implementing the enhancements to the original encoding scheme described in this section, significant improvements were seen. Specifically, by modifying the scheme to make full use of the available alphabet (i.e. value set), increasing the number of expansion codes available, and using multiple data-channel actions, throughputs of up to 2,048,000 bits per second were observed -a more than 5800% increase over the highest throughput seen with the original implemation (which was 34,996 bits per second [1])[5].

This data was collected by sending a number of different-sized messages using a variety of configurations of the covert channel and observing the amount of time required to transmit the message. Each of these tests was repeated multiple times and then the average of these transmission times was used to calculate the throughput. Table 3 gives the complete results of this testing[6].

3. CONNECTION MANAGEMENT PROTOCOL

In addition to being able to transmit and receive covert messages over this channel, applications which desire to use it for performing a collusion attack will need a strategy for discovering and establishing connections with each other. Leveraging the improvements detailed above, a dynamic application discovery and connection negotiation protocol can be defined.

Similar to the covert channel itself, this protocol would require colluding applications to have certain elements established at build time (i.e. before being installed on a device). Specifically, they would need to have a pre-shared control/broadcast action string and an agreed upon covert message format (including what types of messages will be allowed). With these elements defined, applications would be able to be notified when a new another collaborating application is installed, exchange data, and become aware of colluding applications being removed from the device entirely using this covert channel.

The first phase of this protocol, connection establishment, centers around the application installation process. Upon being installed on a device, an application would send a broadcast Intent using the pre-arranged control action “address”.

Note that this broadcast should not be an ordered broadcast since delivery is not guaranteed in that case [14], [15]. The Intent would contain a set of metadata about the sending application’s capabilities encoded using the covert channel, with this metadata specifying what sensitive user information the application has access to (based on its declared permissions), what off-device exfiltration mechanisms it can provide access to, and the set of control and data actions it has receivers registered for.

This initial broadcast Intent would be received by any previously installed application which is setup to collaborate with the new application. The existing application or applications would then extract and store the received metadata in a persistent connection state table and respond with their own metadata, potentially using a different Intent action (if one was specified in the initial broadcast message).

Upon receiving a response, the new application would update its own state table, at which point a connection would have been established. This would provide the basis for a “source-sink” relationship as described in [2], wherein one application serves as a source of information and the other (the sink) provides a means for transporting that information off of the device.

If no responses are received, the application would assume that there are not any other colluding applications installed on the device (i.e. this application is the first one to be installed) and wait to receive an announcement broadcast over the

[5] All of these numbers apply to the time required for transmission of covert data and do not include the time required to encode and decode the data.

[6] Results generated by running the tests on a Toshiba AT7-C running version 4.4.2 of the Android OS.

Table 3. Evaluation of Channel Improvements

Message Size (Bytes)	Number of Expansion Codes[7]	Number of Actions	Number of Unique Values	Average Throughput (Bits per Second)
256	0	1	21	179,200
		25	525	231,564
		100	2100	217,600
	1	1	42	166,400
		25	1050	256,000
		100	4200	256,000
	2	1	63	174,933
		25	1575	256,000
		100	6300	256,000
	22	1	483	81,067
		25	12075	256,000
		100	48300	256,000
1024	0	1	21	204,800
		25	525	1,024,000
		100	2100	1,024,000
	1	1	42	209,920
		25	1050	1,024,000
		100	4200	1,024,000
	2	1	63	221,867
		25	1575	1,024,000
		100	6300	1,024,000
	22	1	483	129,829
		25	12075	1,024,000
		100	48300	972,800
2048	0	1	21	274,286
		25	525	**2,048,000**
		100	2100	**2,048,000**
	1	1	42	227,556
		25	1050	**2,048,000**
		100	4200	**2,048,000**
	2	1	63	163,840
		25	1575	**2,048,000**
		100	6300	**2,048,000**
	22	1	483	97,995
		25	12075	**2,048,000**
		100	48300	**2,048,000**

covert channel from another application. Note that if this application has the ability/permissions to act as a source application, it could still perform malicious activities while waiting for another application to be installed, such as collecting, processing, and/or aggregating user data in preparation for exfiltration.

Once a covert connection has been established, the second phase of the protocol, data exchange, would begin. Two important message types which could be used during this phase of the protocol are requests for specific information and requests to exfiltrate data. Requests for information would most reasonably be sent from “sink” applications (i.e. those capable of transmitting data off the device) to a source application (one with access to information of interest to the attacker) and should specify what type of data the request is interested in[8]. The request could also include additional parameters such as details on how the data should be delivered (what format, how fast, which action or action strings to use, etc.) as well as any filters which should be applied.

Conversely, requests to exfiltrate information would generally be sent from a source application to a sink application, although this type of message could be used by another sink application in order to gain access to an exfiltration strategy which it does not itself have the necessary permissions for (e.g.

[7] The base value-set size for all test runs was 21.

[8] The types of data which the source application has access to would have been provided in that application’s initial broadcast message.

using the internet instead of SMS or vice-versa). Such requests could similarly include additional quantifiers. For example, the requester could specify which exfiltration strategy to use (if more than one is available[9]), including what medium to use, the rate of transfer, whether to use a discreet transfer approach such as steganography or a network covert channel, and/or an encryption key to use before transmitting the data.

The third phase focuses on closing established connections which are no longer valid because the other destination application has been uninstalled. There are two basic ways to approach this problem. The first is to have every involved application setup their Intent filters to accept the ACTION_PACKAGE_REMOVED action, which will allow them to be notified upon removal of any other application [16]. Note that this requires the application name which will be used by the operating system in this broadcast to be included in the initial announcement message used for establishing covert connections.

The second option would be to require every covert message to be acknowledged by the receiving application (using the covert channel). In that case, given the high reliability of this channel (the authors have not seen any cases of an Intent failing to be delivered), if an acknowledgement is not received within a set amount of time, the destination can be assumed to be invalid (due to removal from the device) and that entry can be removed from the state table. The disadvantage is that this approach is noisier (effectively doubles the number of covert messages which must be sent) and it introduces a greater delay in closing invalid connections. However, it could help avoid certain methods of detection, which will be discussed more in the next section.

4. DETECTION

At the time of this writing, the Android operating system does not inherently provide any mechanism for detecting or protecting against application collusion attacks of any kind, to say nothing of being able to detect the use of covert channels to conduct such attacks. As a result, the only barrier to successfully conducting this type of attack is for the attacker to find a way to get a user to install multiple applications they have created. Such a person would likely publish their various malicious applications using multiple developer accounts to give the appearance that they are completely unrelated. Leveraging a dynamic discovery protocol such as the one described in the previous section, this could be a very effective approach for successfully executing application collusion attacks.

In addition, this approach makes it difficult to detect the malicious intent of these colluding applications without introducing significant enhancements on the current Android security model. The first step would be to implement measures capable of detecting inappropriate and potentially dangerous flows of information between different applications. After this, it would be necessary to further improve these measures to be covert-channel aware in order to detect application collusion attacks which use that vector.

Focusing specifically on collusion attacks which leverage the covert channel described in this article, there are a number of different approaches which could be taken to detect the use of this specific covert channel. Two general categories of detection techniques, static and dynamic analysis, will be discussed below.

With regards to static analysis[10], the most obvious route is to inspect the declared Intent filters of the installed applications and, using this information, construct a relationship graph between all of these applications as seen in [17]. This could serve as a reasonable starting point for more detailed analysis by providing a set of applications which may be involved in a collusion attack. This relationship graph could be further enhanced by also considering what types of Intents each application will send based on an analysis of the application source code[11]. However, given that Intents are explicitly designed to be sent between different applications, this is not sufficient to definitively determine the existence or non-existence of this channel.

Another possible static analysis technique would be to analyze the application code for instances of logic for encoding information within Bundle value data-types. While this is a far more definitive strategy, there would be a risk of both false positives and false negatives, especially given the possibility of deliberate obfuscation of this logic. A different aspect of the application logic which could contribute to such analysis would be the use of explicit Intents, which target a specific application by name and will be received by that application regardless of its Intent filters. Such Intents could be used to circumvent analysis which only looks at each application’s declared filters [10].

[9] This information would have been provided in the initial broadcast from the targeted sink.

[10] Analyzing the application artifacts without executing the program.

[11] This would either have to be provided by the developer, which is unlikely, or decompiled from the Android Application Package (APK) file for the application.

In the realm of dynamic analysis, a number of other approaches are available which can be further categorized based on what level of privileges they require (user or system). The options at the user level are obviously more limited than at system level, with the tradeoff being that such techniques are much more straightforward to implement as they do not require rooting the device or modifying any part of the Android operating system.

The user-level approach which is most likely to provide a benefit with regards to detecting this covert channel would be Intent "sniffing" (collecting and analyzing Intent traffic on the device). This would provide the opportunity to examine the contents of Intents that have been sent for any indication that this channel is being used. These indicators could include unusually large Intent Bundles, Bundles containing an unusually high number of key-value pairs, Bundle keys which appear to be randomly generated, an abnormal degree of Bundle nesting, and/or an unusual number of Intents being sent to and/or from a particular application. Statistical analysis techniques might also be able to applied to the contents of these Intents to look for unexpected or uncommon patterns, focusing on the data-types within each Intent Bundle.

It is important to note that all of these indicators would first require a baseline to be established for the system in order to create definitions for "unusual" and "abnormal" in the context of the applications which are installed on the device[12]. Also worth noting is the fact that this channel could be tuned in order to avoid these indicators at the cost of decreased bandwidth. For example, the keys could be generated using a dictionary of words which fit the overt purpose of both the sending and receiving application. Another shortcoming of this approach is that there is no way to guarantee that all of the Intent traffic on the device will be seen since the sniffing application would only receive Intents which match its Intent filter, limiting it to Intent types that its creators are aware of. Beyond this, Intents which are not sent as a broadcast will only be received by one application, and if more than one application has a matching Intent filter, the user will be prompted to select an application to use. This requires the user to be aware of what this prompt means as well as the reason for selecting the sniffer application. The more likely scenario is that the added noise will be viewed as an annoyance by users who will eventually ignore, or even remove, the sniffer.

In contrast, a monitoring application operating at the system (root) level would be able to ensure that it receives a copy of every Intent sent on the system without requiring any on-going involvement from the user[13]. This would enable a complete picture of the system Intent traffic to be constructed, including an accurate baseline which could (and should) be adapted as applications are installed and removed. At this point, a greater confidence could be placed on any anomalies detected.

Performing monitoring from within the operating system also opens more avenues for reacting to any possible uses of this channel which are detected. One of the simplest and least obtrusive options would be to notify the user of the event and allow them to take any action they choose. Going beyond this, the monitoring application could also incorporate the functionality of an intrusion prevention system (IPS) in addition to that of an intrusion detection system (IDS) by isolating or even removing suspect applications automatically.

Leveraging this concept, such a system could also perform active analysis in addition to passive monitoring. Specifically, once the system baseline has been established, the monitoring program could isolate the installed applications one at a time (i.e. prevent any Intent traffic to or from the application) and then examine how the baseline changes. First, the Intent traffic which the isolated application attempts to send could be analyzed to look for changes in that applications Intent traffic pattern in reaction to not receiving Intent messages from any other applications. Second, since the application would not have been uninstalled at this point, any colluding applications would have to assume that it is still installed until they send a message to it, which would be noted by the monitoring application. Of course, there is no guarantee about the timing of such a message. Also, such messages could easily be sent as a broadcast designed to match the Intent filter of applications not involved in the channel in addition to the actual intended receiver, which would further increase the difficulty of definitively proving the use of this channel through this technique alone.

Ultimately, it may be difficult for any single analysis technique to reliably identify the presence and use of this covert channel. However, it should be possible to combine multiple analysis techniques, with each contributing input into an algorithm for flagging potential use of the channel based on this information. This, in effect, creates a concept of a "trust score" for each application, with each input

[12] It would also be possible to create a baseline from a survey of the Intent traffic patterns of a sufficient number of different applications. [17], [18] may also be useful starting points.

[13] Note that this would require modification of the operating system code for handling Intents.

(i.e. analysis technique) affecting the total score based on its weight (how effective and reliable that technique is when used on its own). The decision algorithm responsible for combining the various inputs could then be little more than a weighted calculation of the trust score for each application followed by a check against a pre-defined trust score threshold.

5. CONCLUSION

As has been previously stated [1], [2] and has been emphasized here, application collusion attacks provide a simple yet effective means to circumvent the existing Android security system. Covert channels can provide a powerful means of executing such attacks, even in the presence of additional security controls which attempt to look for them. The channel detailed in this article is particularly potent given its high bandwidth relative to other similar covert channels and its ability to easily support a flexible and adaptable inter-application control structure[14].

That being said, a number of potential options for addressing this particular danger do exist. Further evaluation is needed in order to determine which of these techniques will be the most effective and then, most importantly, the results of these efforts must be applied to the Android security system in order to better protect the users of that system.

6. REFERENCES

[1] T. Heard, D. Johnson, and B. Stackpole, Exploring a high-capacity covert channel on the Android operating system, 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Warsaw, Poland (September 24-26, 2015), vol. 1, pp. 393–398.

[2] Hubert Ritzdorf , Analyzing Covert Channels on Mobile Devices, Master Thesis, available online on http://e-collection.library.ethz.ch/eserv/eth:5608/eth-5608-01.pdf, accessed June 2016.

[3] S. Chandra, Z. Lin, A. Kundu, and L. Khan, Towards a systematic study of the covert channel attacks in smartphones, International Conference on Security and Privacy in Communication Networks, Beijing, China (September 24-26, 2014), pp. 427–435.

[4] W. Gasior and L. Yang, Network covert channels on the Android platform, Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research, Oak Ridge, Tennessee, USA (October 12-14, 2011), p. 61.

[5] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones, 18th Annual Network & Distributed System Security Symposium, San Diego, California, USA (February 6-9, 2011), vol. 11, pp. 17–33.

[6] A. Al-Haiqi, M. Ismail, R. Nordin, A. Al-Haiqi, M. Ismail, and R. Nordin, A New Sensors-Based Covert Channel on Android, Scientific World Journal, volume 2014 (2014), available online on http://www.hindawi.com/journals/tswj/2014/969628/, accessed June 2016.

[7] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, Analysis of the Communication Between Colluding Applications on Modern Smartphones, Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, Florida, USA (December 3-7, 2012), pp. 51–60.

[8] J.-F. Lalande and S. Wendzel, Hiding Privacy Leaks in Android Applications Using Low-Attention Raising Covert Channels, 2013 Eighth International Conference on Availability, Reliability, and Security (ARES), Regensburg, Germany (September 2-6, 2013), pp. 701–710.

[9] W. Gasior and L. Yang, Exploring Covert Channel in Android Platform, Cyber Security (CyberSecurity), International Conference on, Washington, D.C., USA (December 14-16, 2012), pp. 173–177.

[10] Intents and Intent Filters | Android Developers, Android Online Documentation, available online on https://developer.android.com/guide/components/intents-filters.html, accessed June 2016.

[11] Intent | Android Developers, Android Online Documentation, available online on https://developer.android.com/reference/android/content/Intent.html, accessed June 2016.

[12] Bundle | Android Developers, Android Online Documentation, available online on https://developer.android.com/reference/android/os/Bundle.html, accessed June 2016.

[13] MediaStore: ACTION_IMAGE_CAPTURE | Android Developers, Android Online Documentation, available online on

[14] Admittedly, this could be accomplished using virtually any other similar covert channel, but the availability of distinct communication links for different types of messages based on different action strings reduces the amount of overhead required.

https://developer.android.com/reference/android/provider/MediaStore.html#ACTION_IMAGE_CAPTURE, accessed June 2016.

[14] BroadcastReceiver | Android Developers, Android Online Documentation, available online on https://developer.android.com/reference/android/content/BroadcastReceiver.html, accessed June 2016.

[15] Context: sendBroadcast | Android Developers, Android Online Documentation, available online on https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent), accessed June 2016.

[16] Intent: ACTION_PACKAGE_REMOVED | Android Developers, Android Online Documentation, available online on https://developer.android.com/reference/android/content/Intent.html#ACTION_PACKAGE_REMOVED, accessed June 2016.

[17] K. O. Elish, D. Yao, and B. G. Ryder, On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions, IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, San Jose, CA, USA (May 18-20, 2015).

[18] Common Intents | Android Developers, Android Online Documentation, available online on https://developer.android.com/guide/components/intents-common.html, accessed June 2016.

Timothy Heard

Software Engineer (SDE),
Amazon Web Services

Bachelor of Sciences,
Software Engineering
Rochester Institute of
Technology, 2014

Professional Interests

Computer Security
Software Development
Cloud Computing

Programming Languages

Java
Ruby
C
C++
Erlang
Python

Professor Daryl Johnson is an Associate Professor in the Computing Security department at Rochester Institute of Technology. He received his MS in Computer Science from RIT in 1987. He has developed over thirteen and co-developed over twenty three new courses in networking, security and systems administration as well as redesigning and contributing to many others. He has been a principal in the creation of three departments and seven degrees. Most of his attention over the last two decades has been in the areas of Computer and Network Security with a focus on Covert Communication, Botnet C&C and vulnerabilities in SCADA/ICS/IOT. He has authored over a fifty papers in the security area.