Security Breach in Smartphone with Overt and Covert Channels

Team Project

KimLong Nguyen

Yuxi Jing

Department of Computer Science and Engineering

Santa Clara University

Fall 2014

Abstract

In this project, we study how attackers can bypass security features implemented in Smartphone platforms by installing a communication channel between two applications. Application collusion can happen because isolation system enforced in Smartphone Operating Systems can be circumvented.

There are mainly two kinds of application connection routes: overt channel and covert channel. Overt channel can easily be constructed. However, its visibility makes overt channel vulnerable to detection mechanisms. Contrarily, a covert channel requires more elaborate setup. But tracking systems may not be able to expose this type of channel.

A covert channel can be created with the use of file lock. One application can send signal by locking or unlocking a file. The other application can retrieve the signal by checking whether the file is locked or not. Our program applies this method to transfer data between processes.

Acknowledgments

We would like to thank our professor, Dr. Ming-Hwa Wang, for giving us the opportunity to work on this project. Doing research on the topic has been a positive learning experience for us. We also appreciate many valuable suggestions given generously by Dr. Wang. His insightful advices contribute to the success of this project.

TABLE OF CONTENTS

1. Introduction

In this part, we present security vulnerabilities in Smartphone platforms that lead to numerous exploits through overt and covert channels. We then explain how studying this subject helps us to better understand other material taught in our Operating System class. Finally, we analyze how overt channel and covert channel can successfully crack down Smartphone defense mechanism.

1.1. Security Breach in Smartphone

Security has always been a significant concern for operating system programmers. They make special effort to build a platform that can protect system resources and user's private data without relying on third-party applications. More and more sophisticated measures are put in place to secure the system. However, hackers can still find ways to breach Smartphone security walls and execute vicious attracts.

In their study, Marforio and al. [1] show how application isolation imposed by Android platform could be circumvented. The authors implemented different connection channels between applications. All installed channels succeeded in leaking user's data to remote servers.

Even with tighter security control such as system imposed in Windows Phone 7 operating system, the mentioned authors could still pinpoint a possible thread. A communication channel could be constructed using the Media Library. As reading and writing to this resource does not require any interaction from user, colliding applications could secretly pass and receive data through the Media Library.

1.2. Security vulnerability and Operating System

Our effort to find answer to the security problem in mobile phone platforms leads us to precede a

sound analysis on different Smartphone operating systems. This investigation brings us profound knowledge on Security features implied on mobile phone platforms, a major topic in our Operating System course's curriculum.

Our discussion is limited to the security feature that got exploited by communication channels between applications. That is the Permission-Based Access Control structure used by many mobile phone platforms.

In their paper [1], Marforio and al. review different tools Android uses to enforce its Permission-Based Access Control. The main goal of this security mechanism is to control access to system resources and to prevent applications from reaching assets that are not in their stated permission.

When an application is installed, Android provides an API showing a set of permissions required by the application. Permission can be access to private data, like contact list, SMS, etc or access to the internet. After user granted the requested permissions, the system will provide a unique UNIX user ID and group IDs to the application. These IDs are based on permissions allowed by user. User ID imposed which files an application can access. Similarly, group IDs control access to other resources such as device descriptors, socket files, etc. Beside user ID and group IDs, Android also uses a middleware that guards against inter-component communication.

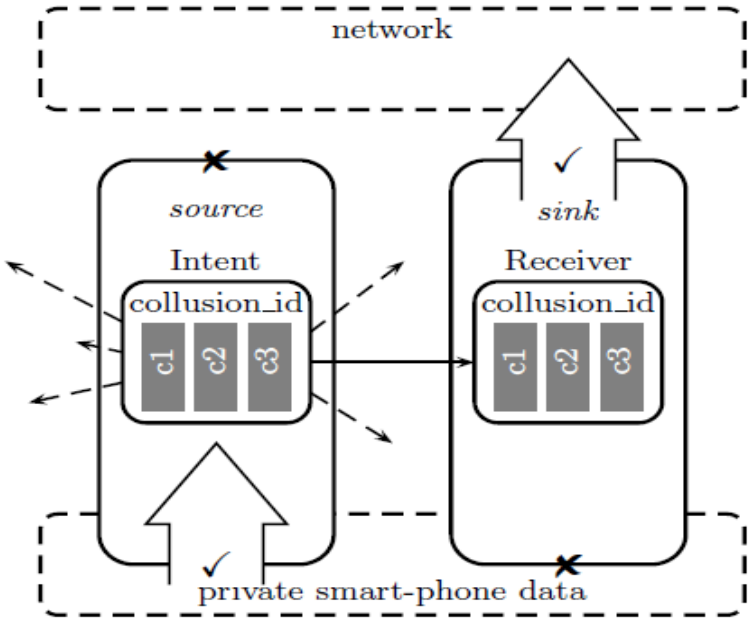
In his paper [3], Nachenberg acknowledges Permission-Based Access Control in Android is a robust system that limits applications to a minimal set of resources required for its operations.

1.3. Overt and Covert Channel

In his thesis [2], Ritzdorf cites a definition of overt and covert channel from Kemmerer [4]. According to the author, in overt channel, an application uses a data container such as a file or a buffet to hide information. The colliding application will then come to the container and get the data.

Marforio and al. [1] illustrate how collusion channel functions in their paper. In their example, an application named Weather who gets permission access to the internet plays the role of the Sink. Sink is the application that receives data. The other application in the channel is the Source which has access to user's private data and sends data to the Sink. They suggest Source can be an application named ContactsManager which has permission access to user's contacts list. Source also can be PasswordsManager which can read user's passwords.

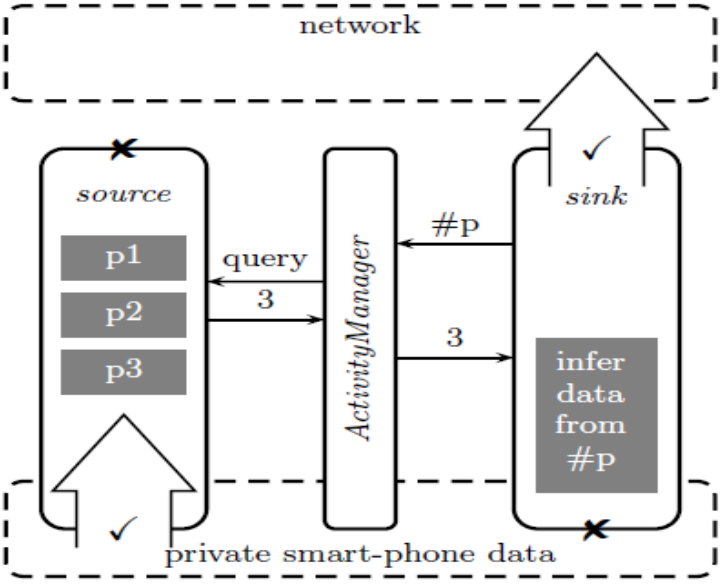
Below is a figure from [1] showing how Source and Sink communicate through an overt channel:



Source hides data in a container called intent. Intent is like a message an application sends out to the whole system. In this example, Source places contacts data c1, c2 and c3 in an intent and broadcast it out. Sink would filter intents and catch the matched one. It collects data from the container. With its right to access to the internet, Sink can leak private data to third-party server through the web.

Contrarily to overt channel, covert channel does not use any data object. Rather, Source manipulates system state, such as file lock or busy flag to signal information to the Sink.

Below is another figure from [1] illustrating communication between two applications through covert channel:



Source application spawns or terminates dummy processes to obtain a total of N processes. N equals to 3 in the above figure. N is the data Source wants to pass to Sink. Sink gets the number of Source's processes through system's API. It then infers the number to get the message. Message is later sent out to the network.

The operation of overt and covert channels reveals how these mediums break the powerful Permission-Based Access Control mechanism. In fact, Sink which does not have permission to access private data ends up getting the data anyhow. Similarly, Source is not given access to the network so it could not release private data out. It still manages to send data to the internet through these channels.

2. Theoretical Bases and Literature Review

Our program will apply the approach of using file lock to transfer data. In this section, we'll first present how file lock mechanism works. We'll then introduce an implementation of the concept. In this case, the authors have successfully built a covert channel in Smartphone using file locks. We'll also show how a covet channel via file lock can be detected. Finally, we'll present our plan to model the concept.

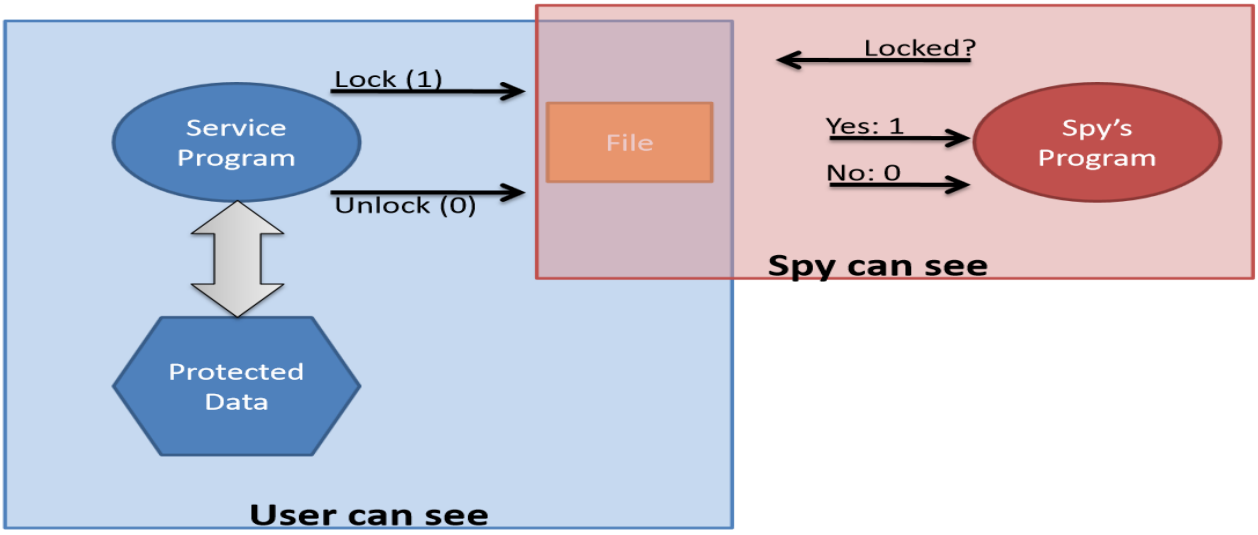
2.1. Concept

The Source application will send data to the Sink one bit, 1 or 0, at a time. The Sink application will collect the sequence of bits, decode it to get the message the Source application wanted to send out.

Before the communication, Source and Sink would agree on a data file as a communication place. The two applications also need to synchronize their acts in an interval time. Source would lock the designated file if it wants to pass 1 to the Sink. The file will be unlocked if Source wants to send out 0.

Later in the same time interval, Sink would try to lock the file. If it fails, it shows that Source has already locked the file. Sink would interpret it as a 1 signal. If Sink successes at locking the file, it knows that Source wants to signal a 0.

Below is a figure from [5] showing how the file locking system operates. The Source is called "Service Program" here. It has access to private data and passes it out one bit at a time by locking or unlocking a file. Both applications have access to this file. The Sink is the "Spy's Program". It receives data one bit at a time by checking whether the targeted file is locked or not.



2.2. Implementation

In the Soundcomber paper [6], the authors show how a covert channel could be built using file locks. In their experience, there were *m* signaling files and one data file involved. Signaling files were used to synchronize the works between Soundcomber (the Source) and the Receiver (the Sink). The data file is used to pass information.

If *m* is equal to 10 then there are 10 signaling files named S1 to S10. Soundcomber will be responsible for the first five signaling files S1 to S5. The Receiver has its five signaling files from S6 to S10. At the beginning of the communication, both Soundcomber and the Receiver lock all of its signaling files. The Receiver also locks itself by trying to lock file S1. Since this file has already been locked by Soundcomber, the Receiver has to wait. In the meantime, Soundcomber sends out the first bit by locking the data file if it's a 1, or unlocking the file if the bit is 0.

When this is done, Soundcomber will release the Receiver by unlocking file S1. It also tries to lock file S6. As this file has already been locked by the Receiver, Soundcomber has to wait this time.

The Receiver who just got released will go to the data file to read the first bit sent by Soundcomber. The Receiver then wakes up Soundcomber by unlocking file S6. The Receiver will also try to lock file S2 which is still locked by Soundcomber. So the Receiver will have to wait while Soundcomber is sending the second bit.

By rotating to send and read bit, Soundcomber and the Receiver are able to communicate through the built system. Using signaling files ensures synchronization between the two programs. In fact, it guarantees that the Receiver only reads data after Soundcomber has sent out a new signal. The signaling scheme also makes sure that Soundcomber only sends out new bit after the Receiver has read the previous bit.

2.3. Detecting Covert Channels employing File Lock

Despite its complex setup, covert channel using file lock can still be exposed by the Shared Resources Matrix technique. This method uses a matrix to track shared resources between processes. Below is a figure from [7] showing how a Shared Resources Matrix looks like:

	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	

Matrix rows are filled with resources. In the case of two applications passing data through file locks, the resources are the file lock and the data. Columns show processes that have access to the resources. An entry of "R" means Read access. "M" stands for Modify access. A process can write, change, create or delete resources with M right.

Both the Service Process (the Source) and the Spy's Process (the Sink) can access the first resource. Regarding to the second resource, only the Service Process has access to it. The Spy's Process cannot read confidential data.

From this Share Resources Matrix, a possible data flow can be identified. Service Process can pass data by modifying the lock to signal Spy's Process. This information flow will be added to the matrix to track for potential data leakage.

From the modified matrix below taken from [7], we can see Spy's Process can read data through the covert channel using file lock.

	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	R

2.4. Our Method

As we can see, covert channels using file lock can be spotted by the Shared Resources Matrix routine. Adding several shared signaling files into the structure as the Soundcomber paper [6] suggested can only make the channel more visible.

In our program, we only use one single shared variable between Source and Sink. Even with just one shared variable, we make sure the communication channel between these two threads remains

as much reliable.

The shared variable is an instance of a class we build. The object serves as a lock to guard the critical section. For Source, its critical area is sending data. As for Sink, its critical section is reading data. The lock assures only one thread can enter critical section at a time. As a consequence, Source and Sink could only operation in the order as planned.

3. Goals

Our goal is to build a covert channel that implements a simpler synchronization system than the setup of *m* signaling files by Soundcomber paper's authors [6].

Our channel needs to guarantee an absolute coordination between Source and Sink. This means there won't be any risk for deadlock or race conditions. Consequently, data stays safe from corruption during transfer.

We also set a target transfer rate better than what achieve by the above authors. Their channel offers a bandwidth of more than 685 bps [6].

4. Methodology

We present in this section some design highlights of our program:

4.1. Input Data

Since Source can only send one bit at a time, any random sequence of 0 and 1 can be a good testing input.

However, we decided to use printable ASCII text as input. This allows us to print out data to verify the reliability of our channel. We'll make sure to include different types of characters, such as number, letter in lower and upper case, special characters: . (dot), _ (underscore), @, etc.

4.2. Process

The communication channel is constructed with two threads. One thread does the work of Source. The other thread is the Sink. A data file is made available for both threads to use as a connection point. Source also gets the message it needs to send to Sink.

For each character of the message, Source computes its binary string. It then sends bit-by-bit to Sink by locking or unlocking the lock file. After sending out last bit of the last character, Source sends a sequence of eight bits of 1 to indicate the end of communication.

Since the binary string 11111111 does not correspond to any ASCII character, Sink won't mistake it as another data sent by Source. Therefore it would stop reading as soon as it got the signal. This mechanism prevents Sink from an infinite wait.

In a loop, Sink will check the lock file to read one bit during iteration. After collecting eight bits, Sink converses the binary string to its equivalent character. If the binary string contains only 1, Sink will get out of loop and ends the execution.

4.3. Algorithm Design

```
/* main takes two command line arguments: a lock file name and a message file
name.  Source will lock/unlock the lock file to signal information.  The content
file contains the message for Source to transfer to Sink.  Main also creates an
instance of sharedLock object.  This lock will be passed to both Source and
```

```
Sink.   This shared variable will keep Source and Sink synchronized */


Algorithm main (args[])
      lockFile = args[0];
      messageFile = args[1];
      SharedLock lock = new SharedLock();
      Source source = new Source(lockFile, messageFile, lock);
      Sink sink = new Sink(lockFile, lock);
      source.start();
      sink.start();
end main


/* This is the run function of Source class. It uses a nested loop to process the
message.  Outer loop go through each character.  Inner loop walks through each
bit of the character binary string.

After loop end, Source sends 8 bits of 1 to signal end of message to Sink.*/


Algorithm run ()
      for ( each char of message)          // outer loop
      begin
            bits = char_to_binary(char);
            for (each bit of bits)         // inner loop
            begin
                  if  bit = 1
                  then
                        lock (lockFile);
                  else
                        unlock (lockFile);
            end
      end                                  // end of message
```

```
      for ( i=0 ; i<5 ; i++)              // send 8 bits of 1
      then
            lock (lockFile);
 end run
```

/* This is the run function of Sink class. In a loop, Sink reads one bit at a time by checking if lockFile is locked or unlocked.

After reading 8 bits, it checks if all bits are 1. If it is, it breaks out of loop. If it's not, it converse the binary string to a character. It then concatenates the new character to message string. It then goes to read next bit*/

```
Algorithm run ()
      while (true)
      begin
            if lock(lockFile) = null  // file already locked by Source
            then
                  bit = 1;
            else                      // get lock (Source did not lock)
                  unlock(lockFile);   // release lock
                  bit =0;

            str += bit;               // add new bit to binary string

            if str.length() = 8       // get all 8 bits
            then
                  if (isAllOne(str))  // if 8 bits of 1: signal end of file
                  then
                        break;        // end loop
```

```
                else
                        c = binary_to_char(str);
                        message += c;  // add new char to message string
                        str = "";      // empty binary string
        end
    end run
```

4.4. Language Used

Our program is written in java.

4.5. Tools Used

We have used Java API's multi-thread and synchronization.

4.6. Output Generation

Sink thread prints out the data it got from Source. This information needs to match the original data in order to prove the transmission channel functions properly.

We have used three different debug modes. When debug is set to 0, program prints out sequence of bits from Source and from Sink. This allows us to verify the accuracy at the bit level.

With debug set at 1, Source and Sink print out the whole message. We then can check if the entire message remains consistent after the transmission.

For debug mode 2, program calculates data transfer rate. To get an accurate rate, Source and Sink only do the sending and receiving data. The threads don't execute any print statement.

5. Implementation

In this part, we'll first present data structures we have developed for our program. We'll start from small, inner structures to the bigger, outer ones. We'll then show the interaction between these structures through a few charts.

5.1. Data Structures

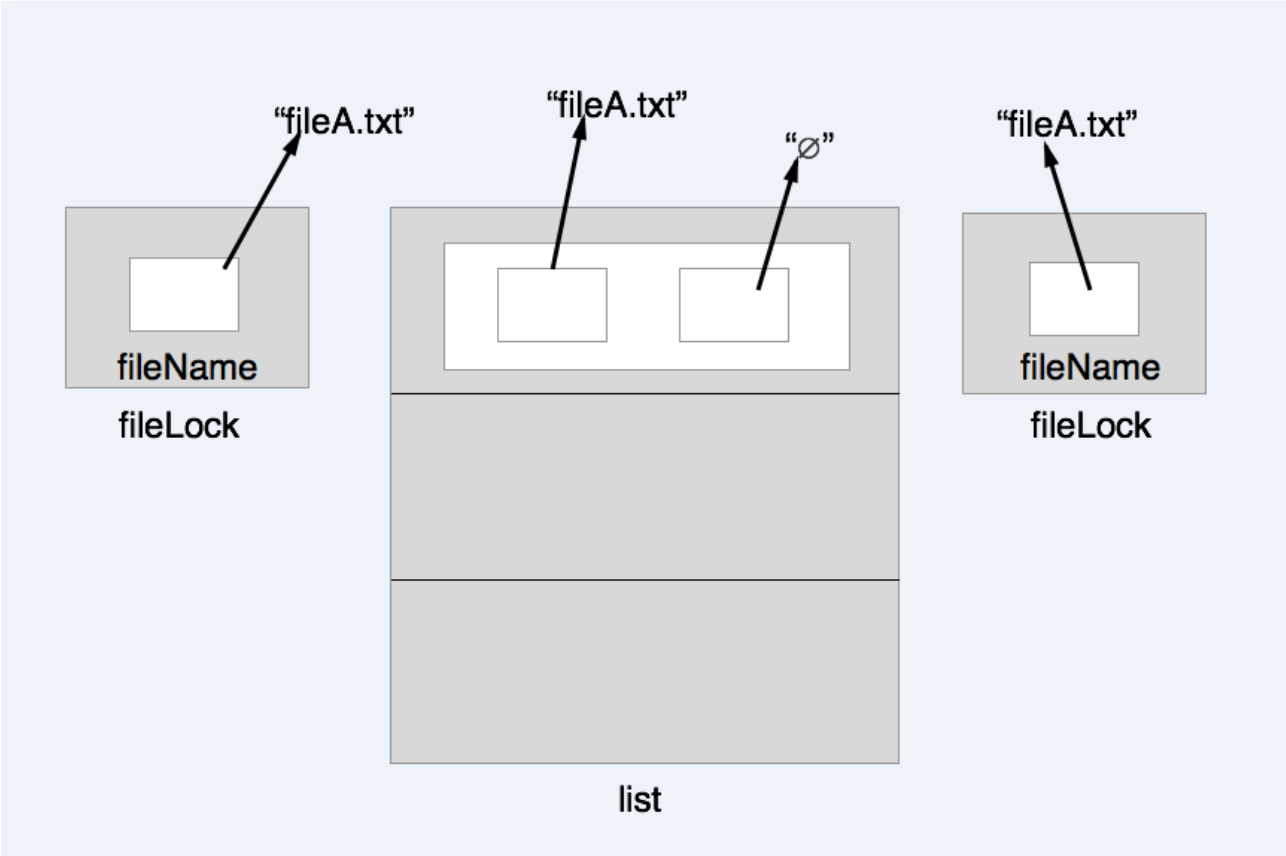
5.1.1. *fielLock* Class

Our original idea was to use file lock as a communication way between two threads. However, we have found out later that file lock was not a good solution for our exercise. Below is a quote from Oracle Docs page [8] advising against the use of file lock in the context of multiple threads:

"File locks are held on behalf of the entire Java virtual machine. They are not suitable for controlling access to a file by multiple threads within the same virtual machine."

Therefore, we proceeded to create a structure that could mimic the functionalities we were looking for in Java FileLock class. Our class was named *fileLock*. Instead of locking a file, Source will now flag a file name to 1 when it sends a bit 1. Flag will be set to 0 by Source when the bit is 0.

Below is the sketch of *fileLock* structure:



fileLock class has a static variable named *list*. This is a hash table that keeps track of flag on each file. Key is file name and value is its flag which could be 0 or 1. *fileLock* also has an instance variable name *fileName* which is the file name.

When an instance of f*ileLock* is created, a new record is added to *list* if the file name is not yet in the list. Since Source and Sink are given the same file name, they refer to the same record in the hash table.

This structure is simple yet it is able to provide an efficient way for Source and Sink to communicate to each other one bit at a time the same way we expected from Java FileLock API.

Figure 1 in the Appendix shows comparison between Java FileLock and our *fileLock* API.

5.1.2. *SharedLock* Class

The *SharedLock* class was created to play the role of a synchronization tool. It has an instance variable name *locked*. *Locked* can be *true* or *false*. It also provides two synchronized methods. One is for Source to call when it wants to go to its critical section. The other one is for Sink to call to enter its critical section.

5.1.3. *TimeAndCount* Class

This class is used to keep track of the time it takes to pass a message and the total number of bits transferred.

main creates two instances of *TimeAndCount*. It passes one to Source so Source could save the time it starts to send out the message and the bits count. Another instance is passed to Sink. Sink will store the time it finishes reading the message and the number of bits it receives.

Based on the start time, end time and bits count, *main* can compute bandwidth of the channel.

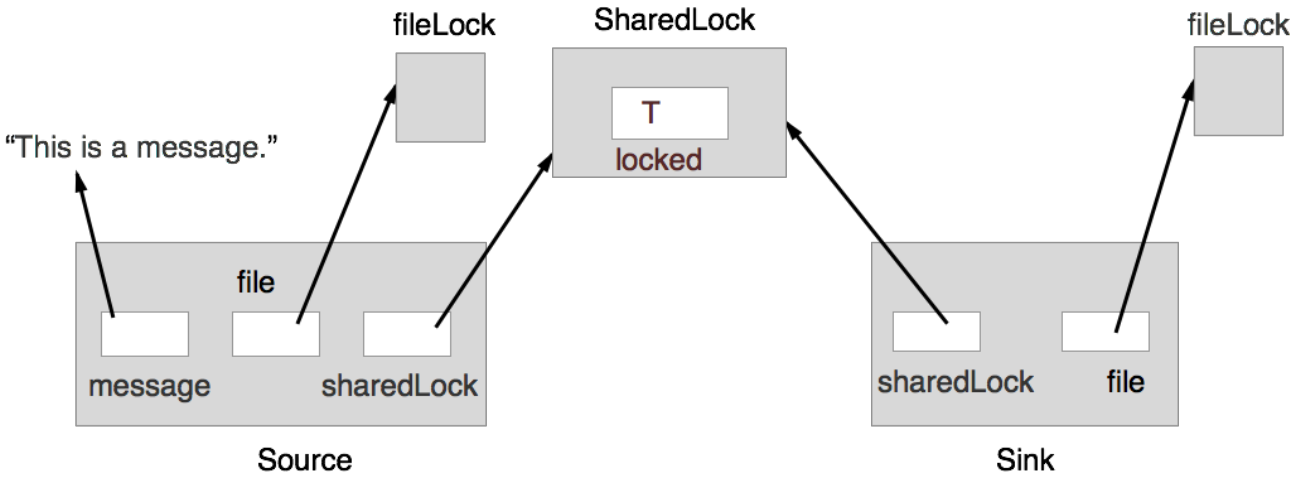
Since this class does not get involved in the transfer, we do not include it in the structure chart to keep the sketch simple and clean.

5.1.4. *Source* Class

Source is a thread that has three instance variables. *messageFile* is the message Source will send to Sink. *file* is an instance of *fileLock* that Source creates with the file name passed by *main*. The last

variable named *sharedLock* is an instance of *SharedLock*. This instance was created by *main* and passed to Source and Sink. Therefore, both threads share this same object.

Below is a figure demonstrating the association between *SharedLock*, Source and Sink.



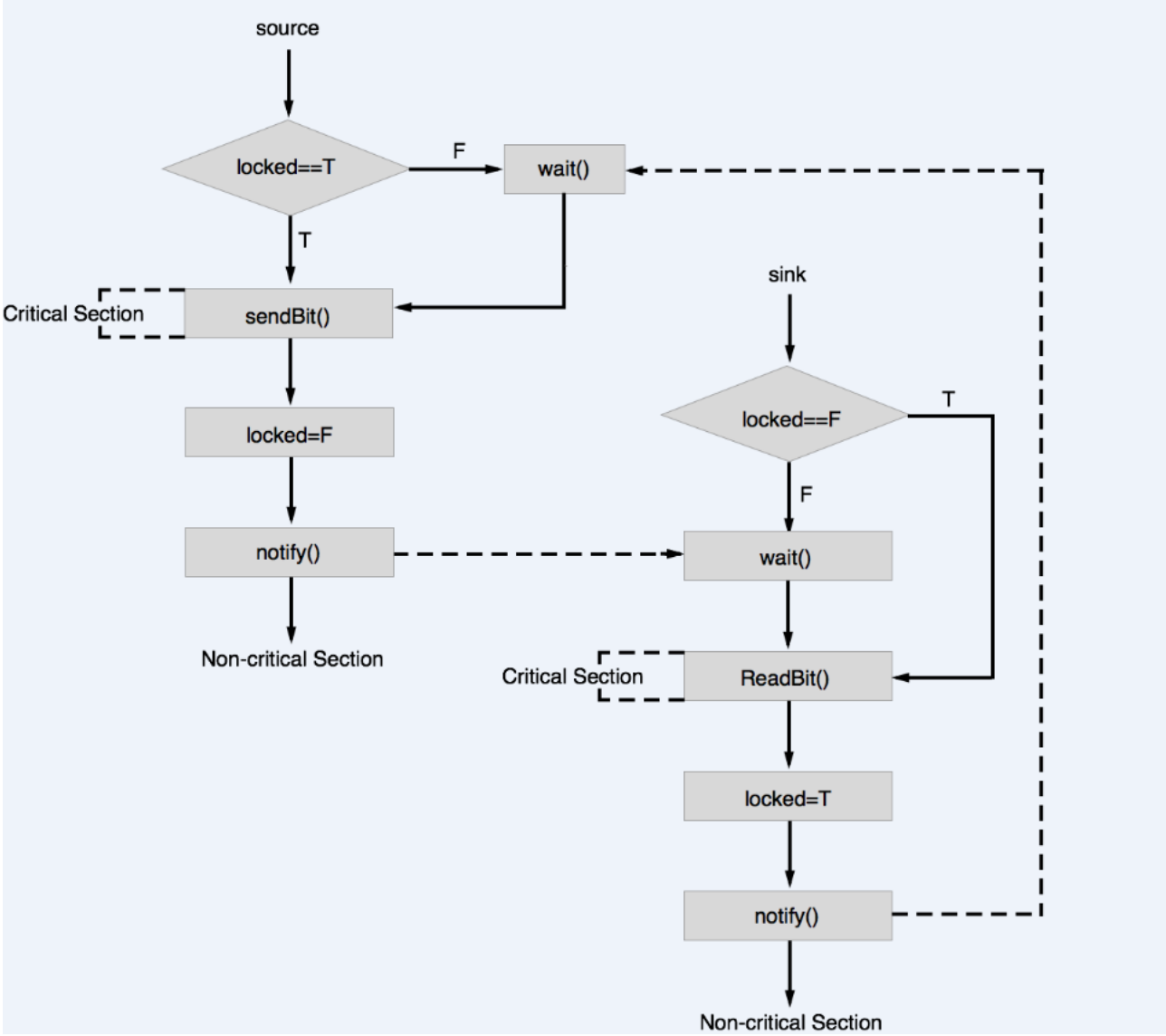
5.1.5. *Sink* Class

Sink is also a thread. Sink only has two instance variables: an instance of *fileLock* named *file* and *sharedLock*, an instance of *SharedLock* that Sink shares with Source.

Figure 2 in the Appendix shows the relationship between all data structures used in the channel.

5.2. Flowcharts

Since we already provided algorithm for *main* function as well as run function of Source and Sink, we'll only present here flowcharts of Source and Sink when each enters their respected critical section. The flowcharts below show a smooth coordination between Source and Sink.



6. Data Analysis and Discussion

In this section, we first present a brief analysis of our program outputs. We then explain how the outputs align with our set goals.

6.1. Output Analysis

We have run our program more than thirty times. Inputs were text files of different lengths and text was made of varieties of characters. We have gotten matched outputs every single time. There were no missed bit, no error bit.

In the Appendix, we provide two output samples. Figure 3 prints input bits alongside output bits for

an easy comparison. Figure 4 shows printout of whole long text message by Source and Sink. The matching of two texts shows that even with long message, the integrity of data remains stable.

Finally, figure 5 displays two sample outputs of data transfer rate. Since rate is only calculated when bit count from Source is matched the count from Sink, the fact that rate is printed proves the number of bits is identical in both threads.

6.2. Output Discussion

Consistent outputs proved that our goals have been achieved. First of all, the coordination between Source and Sink was assured by only one single variable *sharedLock*. This simple setup is also proven efficient.

As we can see from test run outputs, data did not get corrupted during the transfer. This is a proof of the channel's stability.

Additionally, the performance of our model could be explained by the absence of deadlock and race conditions.

In their book [9], Tanenbaum and Bos list four conditions leading to resource deadlock. They are Mutual exclusion, Hold-and-wait, No-preemption and Circular wait. Also per these authors, when one of four conditions is absent, deadlock won't be possible.

In our scheme, circular wait happens when both Source and Sink are waiting at the same time. From the flowchart, we see that Source only waits when *locked* is *false*. As for Sink, it has to wait when *locked* is *true*. Since *locked* cannot be *false* and *true* at the same time, circular wait won't

take place.

The absence of race conditions could be analyzed in the same way. Source can only enter its critical section to send out a bit when *locked* is *true* while Sink can go to read a bit when *locked* is *false*. Since *locked* cannot be two values at the same time, race conditions are avoided.

Figure 7 in the Appendix illustrates how *locked* insures mutually exclusive access to critical sections.

In term of channel's bandwidth, we were able to get an average of 87.5 bits per millisecond. This corresponds to 87,500 bps which is much higher than our set rate of 685 bps.

Figure 6 in the Appendix shows how the average bandwidth of 87.5 was calculated. In this outline, we provide different rates from twenty test run outputs.

7. Conclusions

Our initial intension was to build a covert channel where a Source thread sent data to a Sink thread by locking or unlocking a file.

However, while Java FileLock class is safe to use in different processes, it is not recommended for file synchronization between threads of same process [8]. Therefore, we have developed a class that simulated Java FileLock. Our *fileLock* class could provide functionalities for Source to send a bit of 1 or 0 to Sink.

In order to coordinate Source and Sink operations, we have implemented a class named *SharedLock*. This class provides a mean for synchronization.

As a result, the program has provided steady outputs. This helped to prove that our channel was safe and reliable. In fact, our system was able to prevent cooperating threads from the risk of deadlock and race conditions.

Additionally, the covert channel achieves a noticeably high bandwidth of 87,500 bps.

Furthermore, the solution we proposed could easily be applied to any similar system where two threads take turn to access a common data structure. Only the critical sections need to be updated to reflect the jobs threads are set to accomplish.

While our model was proven adequate, its application is limited to threads in the same process. Because threads share same address space, Source and Sink could share a *SharedLock* object. The two threads were then able to check and read signal from this common object. Each thread knows when to go and when to wait. Without this shared variable, synchronization would not be possible.

In fact, the authors of the Soundcomber paper [6] had to use *m* signaling files as synchronization tool. They had to go through the complexity of managing several files because they run two different applications. Methods using shared variables are not applicable to their model.

Appendix

Figure 1: Comparison between Java FileLock and *fileLock* API

Method	Java FileLock	*fielLock*
tryLock()	-return a FileLock object if file not locked -return *null* if file already locked	-return a random object if flag is 0 -return *null* if flag is 1
release()	-unlock file	-set flag to 0

Figure 2: Relationship between different data structures in the channel.

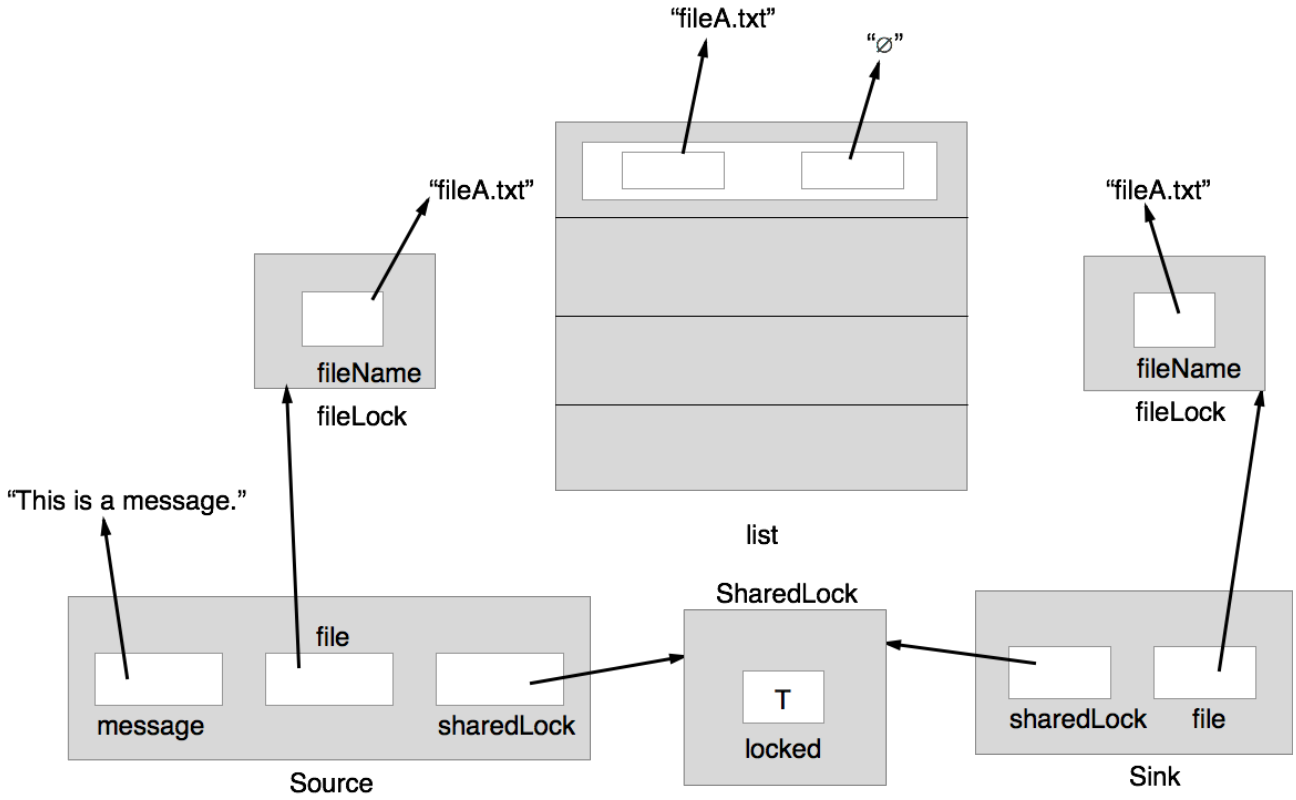
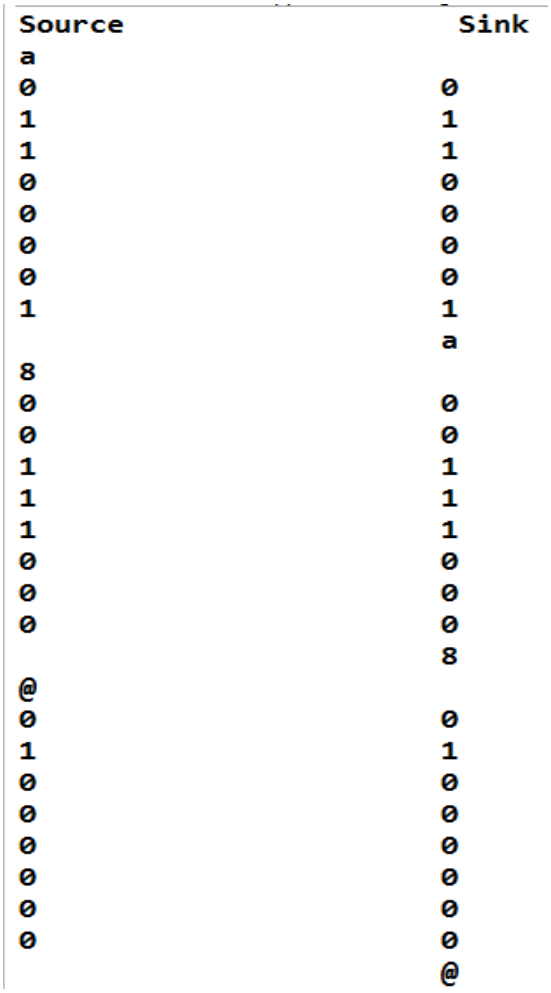


Figure 3: printout bit-by-bit from Source and Sink. All bits are matched.



```
Source                    Sink
a
0                         0
1                         1
1                         1
0                         0
0                         0
0                         0
0                         0
1                         1
                          a
8
0                         0
0                         0
1                         1
1                         1
1                         1
0                         0
0                         0
0                         0
                          8
@
0                         0
1                         1
0                         0
0                         0
0                         0
0                         0
0                         0
0                         0
                          @

T
0                         0
1                         1
0                         0
1                         1
0                         0
1                         1
0                         0
0                         0
                          T
1                         1
1                         1
1                         1
1                         1
1                         1
1                         1
1                         1
1                         1
```

Figure 4: printout long message from Source and Sink. Entire massage is matched.

```
Source:
What a Wonderful World
Louis Armstrong

I see trees of green, red roses, too,
I see them bloom, for me and you
And I think to myself
What a wonderful world.

I see skies of blue, and clouds of white,
The bright blessed day, the dark sacred night
And I think to myself
What a wonderful world.

The colors of the rainbow, so pretty in the sky,
Are also on the faces of people going by.
I see friends shaking hands, sayin', "How do you do?"
They're really sayin', "I love you."

I hear babies cryin'. I watch them grow.
They'll learn much more than I'll ever know
And I think to myself
What a wonderful world

Yes, I think to myself
What a wonderful world


Sink:
What a Wonderful World
Louis Armstrong

I see trees of green, red roses, too,
I see them bloom, for me and you
And I think to myself
What a wonderful world.

I see skies of blue, and clouds of white,
The bright blessed day, the dark sacred night
And I think to myself
What a wonderful world.

The colors of the rainbow, so pretty in the sky,
Are also on the faces of people going by.
I see friends shaking hands, sayin', "How do you do?"
They're really sayin', "I love you."

I hear babies cryin'. I watch them grow.
They'll learn much more than I'll ever know
And I think to myself
What a wonderful world

Yes, I think to myself
What a wonderful world
```

Figure 5: printouts of data transfer rates for 2 different message sizes: 11464 bits and 5432 bits.

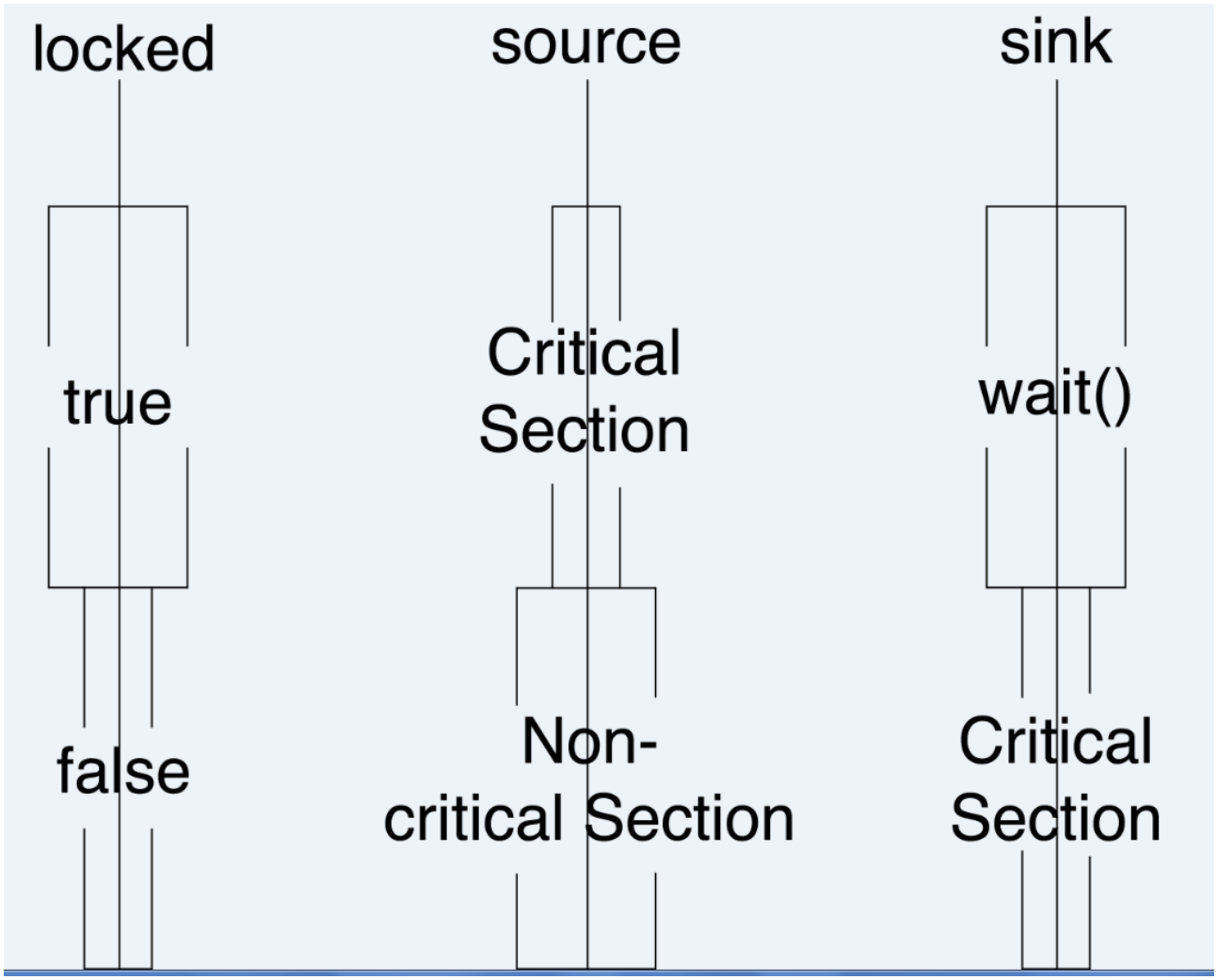
```
---Bits count = 11464
---Transfer time = 109 msec
---Transfer rate= 105 bits/msec


---Bits count = 5432
---Transfer time = 61 msec
---Transfer rate= 89 bits/msec
```

Figure 6: Data transfer rates recoded from 20 test runs with 2 message sizes 11464 bits and 5432 bits.

Bits count	Transfer time (msec)	Transfer rate (bits/msec)	Average rate (bits/msec)
11464	101	113	
	195	58	
	102	112	
	156	73	
	101	113	
	179	64	
	109	105	
	172	66	
	110	104	
	172	66	87.4
5432	54	100	
	86	63	
	55	98	
	86	63	
	62	87	
	70	77	
	55	98	
	70	77	
	55	98	
	47	115	87.6
		Average rate:	87.5 bits/msec

Figure 7: Mutual exclusion to critical section.



References

[1] Claudio Marforio, Francillon Aure´lien, and Srdjan Capkun: Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems. Technical Report 724, ETH Zurich, April 2011.

[2] Hubert Ritzdorf: Analyzing Covert Channels on Mobile Devices. Master Thesis, ETH Zurich, April 2012.

[3] Carey Nachenberg: A Window Into Mobile Device Security - Examining the security approaches employed in Apple's iOS and Google's Android, 2011.

[4] Richard A. Kemmerer: A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later. In *18th Annual Computer Security Applications Conference (ACSAC)*, pages 109–118, December 2002.

[5] Charles P. *Pfleeger,* Shari Lawrence *Pfleeger:* Security in Computing, p. 144, 2003.

[6] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang, Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, February 2011.

[7] Charles P. *Pfleeger,* Shari Lawrence *Pfleeger:* Analyzing Computer Security: A Threat/ vulnerability/countermeasure Approach, p. 514, 2011.

[8] Oracle Docs. https://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileLock.html.

[9] Andrew S. Tanenbaum, Herbert Bos: Modern Operating Systems, 4th Edition, p. 440, 2014.