AN ALGORITHM AND IMPLEMENTATION TO DETECT COVERT CHANNELS AND DATA LEAKAGE IN MOBILE APPLICATIONS

By

BAILEY BRIAN NOTTINGHAM

_____

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 1 9

Approved by:

_____

Dr. Saumya Debray
Department of Computer Science

Abstract

As the popularity of the Android Operating System and mobile devices continue to rise, there is a critical need to ensure the sensitive information contained on these devices remains private. Covert channels pose a threat to the Android Operating system by communicating stealthily over channels not intended as a source of communication. The surreptitious channels make it difficult for Android's current security mechanisms to detect the presence of covert communication. Covert channels pose a significant risk to user's privacy because sensitive information requiring explicit permission can end up in applications without the consent of the user. In this thesis, we present an algorithm general enough to detect covert communication in mobile devices as well as desktop and laptop-based devices. We put our algorithm to the test by implementing and testing on real-world applications present in the google play store, malware samples, and samples taken from geographical regions known to produce spyware. We successfully detected covert communication on a suite of Android applications built to communicate covertly and found applications passing sensitive information through Android's interprocess communication mechanisms.

Introduction

At the time of this writing, the Android Operating system runs on a vast number of smartphones, tablets, televisions and has roughly 88% of the worldwide market share [17]. People are swiftly finding these devices as their primary source of computing. The prevalence of this form of computing has resulted in these devices containing a substantial amount of personal data such as photos, credit card information, and user's location. As the mobile computing world continues to grow, there is an urgent need to make sure that sensitive information remains private to the user and only available to the applications granted permission. Current research dealing with collusion examines data leakage across single applications. TaintDroid is a dynamic analysis tool that examines the flow of information as an application runs [10]. FlowDroid and AndroidLeaks are static analysis tools that examine information flow without the need to execute them [1] [13]. This research focuses on examining the flow of sensitive information to network calls and interprocess communication mechanisms. These kinds of leaks require a single application to have permission to generate sensitive information and leak information over the network or through an intent. Covert Channels propose a massive risk to the security model set in place by the Android operating system by bypassing application permissions. These channels expose the operating system's permissions-based model and provide an application with data that the user has not explicitly granted that application to possess. An application that has permission to sensitive information can leak the information to another application via a covert channel that has network permissions to leak the data across the network. It is increasingly vital to develop methods to detect covert communication to understand how common this form of attack is in the real world.

In this paper, we discuss a general algorithm to detect covert channels. This algorithm is general enough to apply to more than just applications on the Android Operating System. The same techniques can apply to applications on other mobile operating systems such as iOS and applications on the desktop and laptop-based operating systems (Windows, MacOS, Linux). Later on, we discuss an actual implementation of this algorithm on real-world Android applications and discuss our results.

Background

Android Operating System

Android's security model is built upon the Linux Kernel and takes advantages of its security features. Android deploys applications in a sandbox with a dedicated data directory for each application to read and store it's relevant data [9]. For an application to control access to hardware devices, Internet connectivity, data, or operating system services the operating system requires the application to obtain permission [9]. Users grant applications permissions upon the first launch of a new application or during installation. Once granted permission, the permission is allowed until the user revokes it. Users prompted with an application needing explicit permission are not always directly told why that permission is necessary and how it will be used [13]. A messaging app might require a user to permit access to contact information, which seems reasonable considering the context of the application, but it could also use that data and transfer it stealthily through the use of a covert channel.

Covert Channels

Covert channels are mediums within an operating system that are not meant to be used as data transfer mechanisms but can be exploited to transfer data [16]. Covert Channels are a security vulnerability that allows information to leak to an unauthorized application [3]. Covert Channels bypass the security mechanisms of operating systems by using these mediums to send small amounts of data at a time. The ability to hide communication makes it extremely difficult for the operating system to catch. A covert channel is best described by "The Prisoners' Problem and the Subliminal Channel" analogy [25]. In this analogy, two prisoners can send messages to each other, but a warden monitors each message. The prisoners want to plan an escape, so they communicate in a way that appears utterly innocent to the warden but contains information regarding their advised escape plan. This way, the prisoners can communicate information not allowed to be communicated without the warden ever knowing. Covert Channels in applications act similarly and are either classified as timing or storage based [3].

Storage-based covert channels communicate by changing the state of a resource that is available to the two applications trying to collude [3]. The

resource may require permissions, so it is necessary that applications have them granted explicitly [12]. Timing-based covert channels operate based on the specific time intervals of different fired events [12]. An example of a storage-based covert channel in the Android software development kit resides in the changing of the state of a mobile device's volume settings [5]. The sender has permission to access the device info of the device, and both the sender and receiver have permissions to access the volume settings. The sender application, which has access to the volume settings and device information will store the device's phone number into the music channels volume settings by using the setStreamVolume() method in the Android application programming interface [2]. This method call will change the state of the music channel to be the phone number of the device. The receiver will now use the getStreamVolume() method in the Android application programming interface to get the current stream volume of the music channel [2]. The receiver is now in possession of information that it does not have permission to and has elevated android's security model. Finding these kinds of method invocations in Android applications is the intuition of our algorithm and implementation.

It is possible that covert communication can happen over more than just two applications. In the case of covert communication happening over three applications, Application 1 will sneak data over a channel to Application 2, which will retrieve it. Application 2 will then sneak that same information either using the same covert channel or by using a different covert channel to Application 3. Application 3 will then retrieve this information and may leak it.

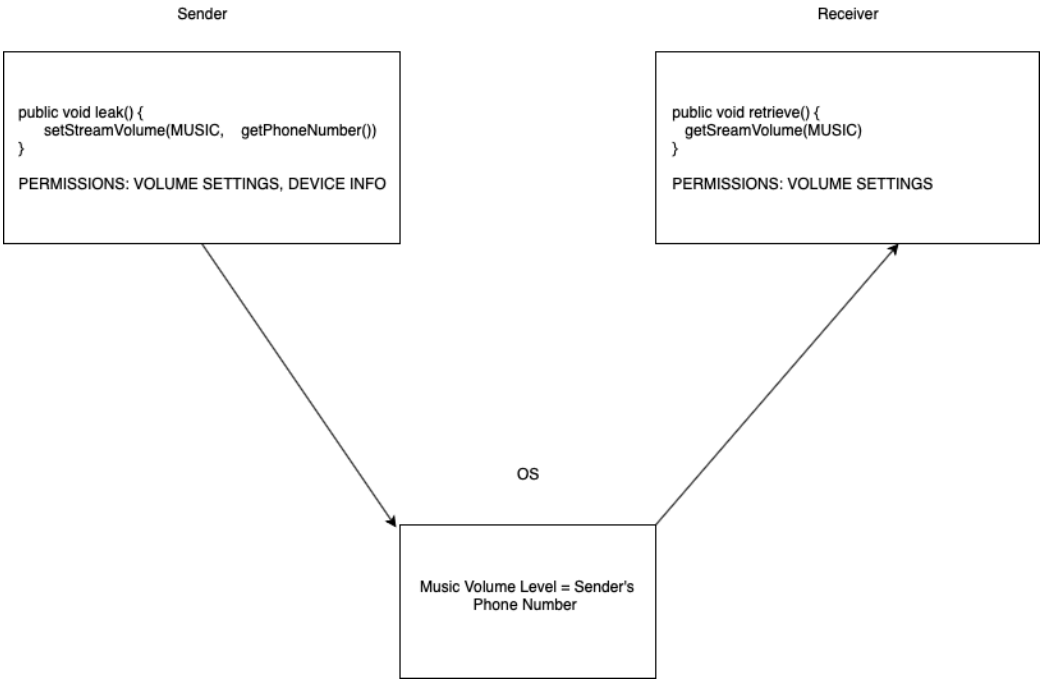

Figure 1: Storage-based covert channel on the volume settings of a device

Program Analysis

Program analysis is a field of computer science that uses algorithms to analyze the behavior of programs [7]. Program analysis techniques are most commonly used in computer security and compiler optimizations [7]. The analysis performed on programs consists of static program analysis and dynamic program analysis [7]. Dynamic program analysis analyzes a program while it is executing, and static program analysis analyzes the source or the object code of a program without the need to run it [19]. Static program analysis techniques are necessary for our covert channel detection algorithm. Later on, in our implementation, we use WALA created by IBM [28]. WALA is an open source Java library that is capable of performing static analysis on Android application packages. Android application packages are the package file format used for distributing applications on the Google Play store and various other application stores [28]. WALA provides us with the functionality to create call graphs, slice program statements, and use control flow graphs to traverse through a program's various execution paths from an

Android application package.

Call Graph

A call graph is a graph that represents functions in a programming language invoking other functions. Nodes in the graph represent functions, and the edges represent the caller to the callee relationship [22]. Having this representation is important because it gives insight into the execution paths of a program.

Intermediate Representation

An Intermediate Representation is an abstract machine language that represents the source code of a program but is not specific to a particular machine [14]. WALA represents the instructions in each node of a call graph as an intermediate representation similar to Java Bytecode. Variables are represented in static single assignment form. Each variable that exists in a call graph node is assigned once and defined before used [6]. WALA numbers each variable in the intermediate representation with a unique identifier starting from one [28]. The intermediate representation also contains a symbol table that consists of all the variables encountered in the call graph node.

Control Flow Graph

A control flow graph is a graph that represents the control flow in a program [26]. A Node represents a basic block. A basic block is a block of instructions that contain no branching [26]. Nodes branch off into edges that represent different possible flows of control to other basic blocks [26]. WALA represents the instructions of a call graph node by a control flow graph. WALA provides a traversal algorithm of the control flow graph by using depth-first search that traverses through each basic block once in the order they appear from the start of the graph. Control flow graphs may contain loops that represent a loop in a program. Loops take up a majority of the execution time of a running program, so it is essential to identify them. Dominator nodes can help identify the presence of loops. A dominator node is a node N that dominates a node M if all paths to M pass through N [11]. After computing the dominators, we can find all the back edges. Back edges from a Node N to Node H is a back edge if H dominates N. Therefore, N is

the entrance of the loop, and the edge jumps back to the beginning of the loop [11]. The body of the loop is then N, H and all the predecessors of N up to H [11].

Program Slicing

Program statements represent a specific action to be carried out in a program. WALA represents statements as *normal*, *parameter caller* and *callee*, and *return caller* and *callee* [30]. *Normal* statements represent instructions in the intermediate representation. *Parameter caller* and *callee* are statements that represent the flow of a variable passing between two functions. *Return caller* and *callee* statements represent the flow of a variable coming from the return value of a function. Program slicing computations gather a set of statements that affect the values of a variable from a specific statement [30]. Slicing is useful because it can remove any unnecessary statements when analyzing the flow of a specific statement. WALA provides the functionality to compute forward and backward slices. Forward slices will gather a set of statements that affect a specific statement after the execution of that statement [30]. Backward slices will gather a set of statements that affect a specific statement before the execution of that statement [30].

Taint Analysis

Taint analysis is a type of analysis that follows a source variable to a sink variable and taints variables that rely on a direct data manipulation operation from the source or other tainted variables [23] [15]. If the source or a tainted variable is in a data manipulation operation, the result variable from that operation becomes tainted [1]. The analysis continues propagating tainting along the execution path. After this analysis, we can determine if the variable sink had a direct impact on the variable source.

Constant Propagation

Programmers commonly assign constant values to variables that are constant on all possible executions of a program [29]. Constant propagation is most commonly used to optimize and remove unnecessary instructions during compile time by attempting to propagate constant values as far forward

through a program as possible [29]. Instructions that add two constant values together can be added together during compilation, and that result can further propagate [29].

Code Obfuscation

Code obfuscation is the art of making code difficult for humans to understand without changing the overall behavior of a program [24]. There are various uses for code obfuscation, but we are interested in code obfuscation that makes it difficult to understand the flow of data through a program, and that can make constant propagation or taint analysis difficult. Figure 2 shows a simple example of two code blocks that share the same behavior. The original code's behavior is easy to understand, but the obfuscated code is challenging to understand because of the conditional statements. In both cases, the setStreamVolume() call uses the channel STREAM_ALARM, but the obfuscated block is difficult to see and for analysis tools to calculate.

Original Code

```
int streamChannel = AudioManager.STREAM_ALARM;
setStreamVolume(streamChannel, getLocation());
```

Obfuscated Code

```
int streamChannel;
int obfuscator = 100;
if(obfuscator < 101) {
  streamChannel = AudioManager.STREAM_MUSIC;
}
if(obfuscator < 102) {
  streamChannel = AudioManager.STREAM_RING;
}
if(obfuscator < 103) {
  streamChannel = AudioManager.STREAM_ALARM;
}
setStreamVolume(streamChannel, getLocation());
```

Figure 2: An example of code obfuscation

Regular Expressions

Regular expressions are a pattern of characters that are used to define a set of strings [20]. Regular expressions are equivalent to finite automata [20]. Since regular languages are expressed as regular expressions, and regular languages are closed under intersection, an intersection of two regular expressions can be computed [4]. To do this, convert both regular expressions to a deterministic finite automaton, compute the product automaton of each other, and make the final states of the product automaton be the pairs consisting of the final states of both deterministic finite automatons [4]. Converting this DFA back to a regular expression yields a regular expression consisting of the intersection of the two regular expressions. In our implementation we use Brics Automaton Java Library to compute the intersection of two regular expressions [18].

Methodology

Algorithm

As seen in the example in Figure 1, a covert channel was possible by setting the stream volume of a particular channel. In order for sensitive information to flow covertly from one application to another two possibilities must exist between the two applications. First, both applications must set and read the volume from the same channel. In Figure 1, the stream type determines the covert channel. Second, sensitive information flows along the execution path to a leaking method invocation. Leaking and Retrieving methods are methods available in an application programming interface that can set or retrieve a shared resource in the operating system. In our previous example, setStreamVolume() was the leaking method because it could set the volume level for a particular audio channel and getStreamVolume() was the retrieving method because it could retrieve the current volume level for that same audio channel. Finding the leaking and retrieving method invocations in applications, determining the channel used by these method invocations, and figuring out if sensitive data flows into the leaking method invocation outlines the main ideas of this algorithm.

Our algorithm has two distinct phases. The first phase, shown in Algorithm 1 finds all the leaking and retrieving method invocations that have already been determined ahead of time due to research on the various application programming interfaces present in the application's operating system. Sensitive method invocations are then checked to see if the generation of data is tainted to impact the data parameter in a leaking method invocation directly. A set of channels is then computed for each leaking and retrieving method invocation. After this phase, we can recognize if any applications are leaking sensitive information. The second phase, shown in Algorithm 4 computes the possibility of covert communication between multiple applications. This algorithm requires methods that can communicate covertly and generate sensitive information as input. Covert communication methods consist of a pair of a leaking method that directly corresponds to a retrieving method. The leaking and retrieving methods must identify which parameters are used to represent the channel, and which parameters store data. Sensitive information methods contain single methods that generate sensitive information. These methods can be found by researching covert channels related to a particular system and by examining the libraries. Our algorithm also requires a

list of applications for analyzing.

Algorithm 1: Phase 1 of Covert Channel Detection. Analyzing Apps

---

```
List<CovertChannels> CovertChannelList = Tagged Pairs (Leak
    to Retrieve) Covert Channel Methods
For every App to be analyzed
   App.CallGraph = generateCallGraph(App)
   App.LeakingStatements =
      findAllLeakingStatementsInCallGraph(App,
      CovertChannelList)
   App. RetrievingStatements =
      findAllRetrievingStatementsInCallGraph(App,
      CovertChannelList)

   For all LeakingStatements in App
      List<Statements> backwardSlice = computeBackwardSlice
         (LeakingStatement)
         For all Statements in backwardSlice
            if Statement represents LeakingStatement
               Approximate value of covert channel
                  parameter
               Store result in LeakingStatemet
            if Statement represents Parameter Passing
               Approximate value of the passed parameter
               Set value in Symbol Table for the calle's CG
                   Node
            if Statement represents Return Value Passing
               Approximate value of return value
               Set value in Symbol Table for CG Node
         LeakingStatement.sensitiveDate = Check if
            Sensitive Data flows into a LeakingStatement
   For all RetrievingStatements in App
      List<Statements> backwardSlice = computeBackwardSlice
         (RetrievingStatement)
      List<Statements> forwardSlice = computeForwardSlice(
         RetrievingStatement)
```

```
        For all Statements in backwardSlice
            if Statement represents RetrievingStatement
                Approximate value of covert channel
                    parameter
                Store result in RetrievingStatement
            if Statement represents Parameter Passing
                Approximate value of the passed parameter
                Set value in Symbol Table for the calle's CG
                    Node
            if Statement represents Return Value Passing
                Approximate value of return value
                Set value in Symbol Table for CG Node
        RetrievingStatement.setLeak = Check if retrieved
            data flows to a LeakingStaement
Write all App data to a file
```

Our algorithm begins by looping over every app that is to be analyzed and computes a call graph representation of it. Using the call graph representation of our app, we can traverse through our call graph by visiting every node and storing every invocation statement of a leaking or retrieving method present in the app. These statements lists are used for further analysis and give us an overview of the types of leaking and retrieving methods contained in an app. After generating leaking and retrieving lists, we need to examine each statement and try to approximate the value of the channel parameter. Our algorithm further analyzes each leaking and retrieving statement by looping over each statement and computing a backward slice from it. The result of this slice now gives us an overview of the statements that have had a direct effect on our statement of interest. These statements may represent parameter passing, return values from called functions, or normal statements that represent instructions in an IR.

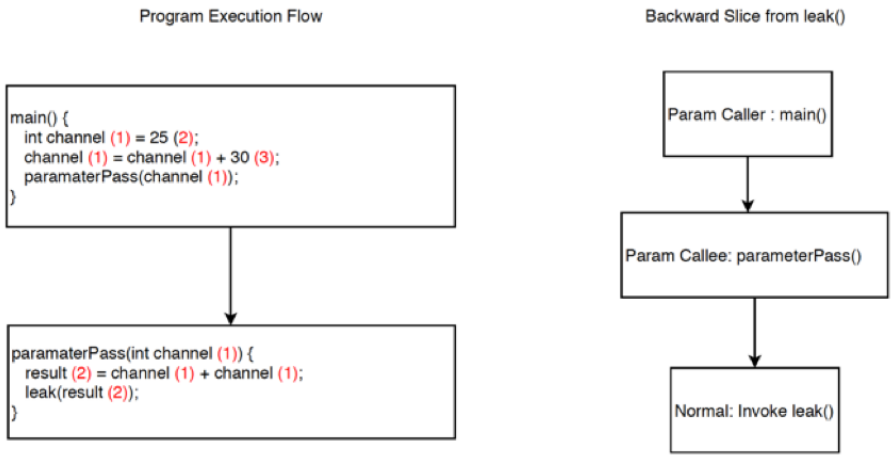

Figure 3: Backward Slice Example

Our algorithm traverses over each statement in the resulting slice and propagates values forward. In Figure 3, a program begins in the function main() that passes a variable into the function paramaterPass() which eventually gets leaked into a function named leak(). In this example, the invocation of leak() was our statement of interest, and a backward slice was computed from there. As we traverse over the result statements the first statement is a parameter pass from main() to the function paramaterPass(). We perform constant propagation to unveil that the parameter being passed into parameterPass() is the value 55. This value sets the unique identifier of that parameter in paramaterPass(). In figure 3, the red numbers are the unique identifiers given to the variables. Variable 1 in paramaterPass() sets the value 55 in its symbol table. The last statement in the result set is a normal invoke statement on our leaking statement. We perform constant propagation on this call graph node and find that the parameter in paramaterPass() was set to 55, and used on an operation that adds 55 + 55 to the variable result. The value 110 is the value associated with the variable result with the unique identifier 2. The tagged channel parameter in the leaking statement leak() is the first and only parameter which corresponds to the variable 2. We store this result with the leaking statement and have now identified the channel as having the value 110.

To perform constant propagation as we did in Figure 3, we need to per-

form the propagation based on the variable type we are trying to evaluate. Algorithm 2 and 3 show the algorithms for performing constant propagation on a call graph node with the result being either an interval or regular expression. If the parameter is numerical, the result will be an interval of possible values the channel may be. If the parameter is a regular expression, the result is a regular expression that represents all the possible strings a parameter may be. Using constant propagation our algorithm attempts to propagate forward constant values to our statement of interest. The algorithms for numerical values and strings essentially follow the same process but have different operations performed on them. The first part of the algorithm gets the variable identifier of our variable of interest. The variable of interest might be the return value of a function call or a parameter of a function. This is decided based on the specific statement from the slice result representing return values or parameter passing. Once we have acquired the unique identifier, we first check if the variable is already constant in the symbol table. If so we either add it into an interval or regular expression and return that result. If the variable is constant upon checking it, there is no need to perform constant propagation on the entire call graph node. If we find that the variable is not constant, we perform constant propagation to approximate the value of this variable. To do this, we first compute the dominators and back edges in the control flow graph to find the basic blocks that represent the body of a loop. We tag all instructions that reside in the body of a loop because the fact that an instruction is inside a loop body implies that this instruction can execute more than once.

Algorithm 2: Constant Propagation Analysis on Numerical Values in a CG Node

---

```
setInterval()
   int specified Variable = Unique ID of variable of
      interest (Parameter, Return Value)
   if specifiedVariable is already constant
      specifiedVariableValue = Constant Value of Variable
         obtained from Symbol Table
      return [specifiedVariableValue]
   ControlFlowGraph cfg = Control Flow Graph from IR of CG
      Node
```

```
Compute Dominators and Back Edges and tag instructions
   that are inside the body of a loop
Map<int, Interval> variableIDToValueMapping

for each BasicBlock through traversal of cfg
   for each instruction in BasicBlock
      if instruction is a numerical operation
         if instruction.operator is addition
            Interval result = execute operation
            if instruction is inside loop body
               if result grows larger after each loop
                  execution
                  add [result, ∞] to map
               else
                  add [−∞, result] to map
            else
               add [result] to map
         if instruction.operator is subtraction
            Interval result = execute operation
            if instruction is inside loop body
               if result grows larger after each loop
                  execution
                  add [result, ∞] to map
               else
                  add [−∞, result] to map
            else
               add [result] to map

         . . . (continues for multiply, divide, modulo)

         if instruction.operator is modulo
            Interval result = execute operation
            if instruction is inside loop body
               if result grows larger after each loop
                  execution
                  add [result, ∞] to map
               else
                  add [−∞,result] to map
            else
               add [result] to map
```

```
List<int> definitions = Check if specified parameter
    variable has multiple definitions
Interval result
for each defintion
    if definition is constant
        fetch constant value from symbol table and add to
            result
    else
        fetch result from map and add to result
return result
```

Algorithm 3: Constant Propagation Analysis on String Values in a CG Node

---

```
setRegexO
    int specifiedVariable = Unique ID of variable of
        interest (Parameter, Return Value)
    if specifiedVariable is already constant
        specifiedVariableValue = Constant Value of Variable
            obtained from Symbol Table
        return Regex.add(specifiedVariableValue)
    ControlFlowGraph cfg = Control Flow Graph from IR of CG
        Node
    Compute Dominators and Back Edges and tag instructions
        that are inside the body of a loop
    Map<int,Interval> variableIDToValueMapping

    For each BasicBlock through traversal of cfg
        For each instruction in BasicBlock
            if instruction is a string operation
                if instruction.operator is concatenation
                    Regex result = execute operation
                    if instruction is inside loop body
```

```
                Append kleene on all disjunct strings in
                    Regex
            else
                add result to map
        if instruction is a invoke operation from Java.
            Lang.String
            if method call is toUppercase()
                execute method on all strings in regex
                add result to map
            if method call is toLowercase()
                execute method on all string in regex
                add result to map

        . . . (continues for methods in Java.Lang.String
            that have return type String)

List<int> definitions = Check if specified parameter
    variable has multiple definitions
Regex result
For each defintion
    if definition is constant
        fetch constant value from symbol table and add to
            result
    else
        fetch result from map and add to result
return result
```

Next, we traverse through the control flow graph of basic blocks and every instruction contained in a block. Every instruction we encounter we perform the associated operation and store the result in a map containing the variables new value. Operations for numerical values include addition, subtraction, multiplication, division, and modulus and operations on string values include concatenation and methods performed in the java.lang.string package. We propagate all constant values forward and evaluate as many operations as possible that have constant inputs.

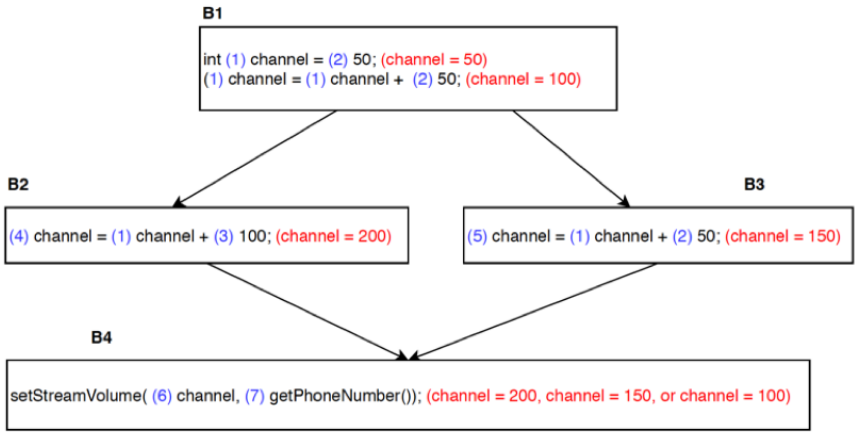

Figure 4: Simple Example of Constant Analysis on Numerical Types.

In Figure 4, an example shows constant prorogation on numerical types. The numbers in blue represent the unique identifiers of variables and the red shows the values of the variables at different points of execution. In block 1, channel is initialized to 50 in the first instruction and used in an addition operation that adds channel plus 50. We perform this addition operation on channel and store the result in the map with the unique identifier as the key and the value as the result of the operation. In this case, identifier 1 sets the map with the value 100. The control flow graph then branches out into two basic blocks. Block 2 performs an addition operation on the variables 1 and 3. By looking up the current value of 1 in the map we find it is 100, and by looking up the value of 3 in the symbol table, we find it is 100. Operating gives a result of 200 in variable 4. The traversal continues on block 3 and then 4 until we get to our leaking invoking instruction. The parameter tagged as a channel in the function setStreamVolume() has the unique identifier 6. We check if this variable has had multiple definitions across the control flow graph and find that 6 defines 4,5, and 1. We now check if these values have been computed in the map or are currently constant in the symbol table to achieve our interval of [100, 150, 200]. When execution reaches our invoking instruction, our channel can either be 100, 150 or 200.

It might be possible that an instruction lives inside the body of a loop

that executes more than once. In this case, we have the instruction tagged from our previous efforts of computing the dominators and back edges in the control flow graph. If computations on constant numerical values in the body of the loop exist, after a single iteration, we can determine if these variables are growing bigger or smaller. If the variable is getting larger, we will compute the channel as an interval between the variable before it enters the body of a loop to infinity. If the variable is getting smaller, we will compute the channel as an interval between the variable before the loop to negative infinity. If the computations on constant string values are present in the body of the loop, the regular expressions associated with each variable appends a Kleene star to each string separated by the or operation. More complex approximations can be performed within the body of a loop to achieve a better approximation. Improving our approximations is left for future work.

At this point in the algorithm, we have now computed approximations for the values of the channels in each leaking and retrieving statement. The next important piece of our algorithm is to determine if each leaking statement has sensitive information flow to the data parameter and if each retrieving statements data parameter flows to another leak. To do this, we use taint analysis to taint the flow of a value from its generation to its end. Taint analysis is used on the return value of a sensitive method invocation to follow the flow of this value and check if it ends up in the data parameter of a covert channel. Taint analysis is used on the return value of a retrieving method invocation to follow the flow of this value to check if it ends up in the data parameter of another leaking statement that could be leaking out this value again. Taint analysis is performed on each leaking and retrieving statement, and the results are stored with each statement.

Our algorithm is now in phase 2 shown in Algorithm 4. All the analysis needed to detect the presence of covert channels has been computed for every app and every leaking and retrieving statement within that app. Our algorithm now writes all of the computed data as a file for each application. Writing data to a file makes the detection more scalable because new apps analyzed in the future already have a set of apps computed.

Algorithm 4: Phase 2 of Covert Channel Detection. Detecting Covert Collusion

```
List<App> apps = Read All Analyzed App Data from Files
List<Node> nodes = Compute Covert Channels between Apps
DisplayResults(nodes)
```

To detect the presence of covert channel communication, we read all the data previously computed back into memory. A graph data structure is used to represent the leak and retrieve between multiple applications. The nodes V in each graph represent an application, and the edges E represent the leak and retrieve from one application node to another application node. The edges contain metadata indicating the intersection result and the sensitive data leaked if present. Every application initializes as a node representing that application. Each application checks all of its leaking statements by computing an intersection between its leaking statement and other applications retrieving statements. Checking each leaking statement in an app with its corresponding retrieve in all other apps computes the edges between the application node and all other nodes. If the intersection yields something other than the empty set, collusion between the two applications is possible, and an edge between the two applications forms. If the channels contain intervals, the intersection yields the values contained in both intervals. If the channels contain regular expressions, the intersection yields a regular expression that refers to a set of at least one string that contains the strings represented in both regular expressions. For every edge added to the node, the same process repeats for the end applications to check if any additional nodes leak and retrieve data from it. Repeating the process for each application added along an edge considers the case where collusion could be happening over more than two applications. Our intervals of channels are approximations using constant propagation. It is possible that an interval may contain values that would not be present during runtime of the application that causes an over approximation. Specific conditional statements or loops in an application may never execute or do not execute as often as we approximate them too. In this situation, the graph would construct an edge as a false positive.

This analysis is evaluated for each statement in each app to create graphs. After the construction of the graph, we use a depth-first search algorithm to look at all the possible paths through a graph starting at a specific node. Running this on every application node, we can see if an application is leaking information covertly and the path of applications that this follows.

The runtime of this algorithm is $O(N * g(s) * f(c) * L * b(x)^2 * S^2 * t(s)^2 * R * w(x))$. N represents the number of Apps to be analyzed, $g(s)$ represents a function of the runtime of constructing a call graph with app size s, $f(c)$ represents a function of the runtime of traversing through every call graph node and instruction in each node for a call graph with c nodes, L represents the number of leaking statements present in app N, $b(x)$ represents a function of the runtime of computing a backward slice for each statement, S represents the number of statements in the slice $b(x)$, $t(s)$ represents the traversal through the call graph node of statement s, R represents the number of retrieving statements present in app N, and $w(x)$ represents a function of the runtime of computing a forward slice for each statement. The runtime consists of numerous products because for every app N multiple computations are performed.

Implementation

The concept of having methods in the Android software development kit set a value in a shared resource and retrieve the resource is common because of the number of features, functions, and resources available on an Android device. Examples include setting the volume of a particular audio stream or turning the vibration mechanism on and off. The shared resources and timing mechanisms in the Android software development kit make the operating system vulnerable to information leakage by covert communication because these resources can employ into communication mechanisms. We implemented a working version of our covert channel detection algorithm on Android applications.

We have identified methods from the Android software development kit that can be used to leak information, retrieve information, and generate sensitive information. We discuss how we found these methods in the evaluation section of this thesis. We define sensitive information as methods in the Android software development kit that generate information that is only accessible if the user grants permission to access that information. Not all methods in the Android application programming interface refer to a specific

channel by a parameter. For example, Android's logging class provides different methods for writing logs to different log files. The files that logs are written to depend on the specific method used. In these situations, we do not need to perform constant propagation to compute a channel. Some methods may contain multiple method invocations that contain the data and channel parameters. For example, to compute the channel for createNewFile(), we must find the method invocation that creates this file object and perform constant propagation on that parameter.

To perform static analysis on Android applications, we use WALA created by IBM [28]. WALA provides us with the functionality to create call graphs, slice program statements, and use control flow graphs to traverse through a program's various execution paths [28]. With the use of WALA for all the needed program analysis tools, we implemented our algorithm in Java.

Figure 5 shows two simple Android applications using a covert channel. This covert channel changes the volume level of a specific stream on the Android device. It can be used to communicate covertly by having two applications pass data by setting and retrieving the volume level.

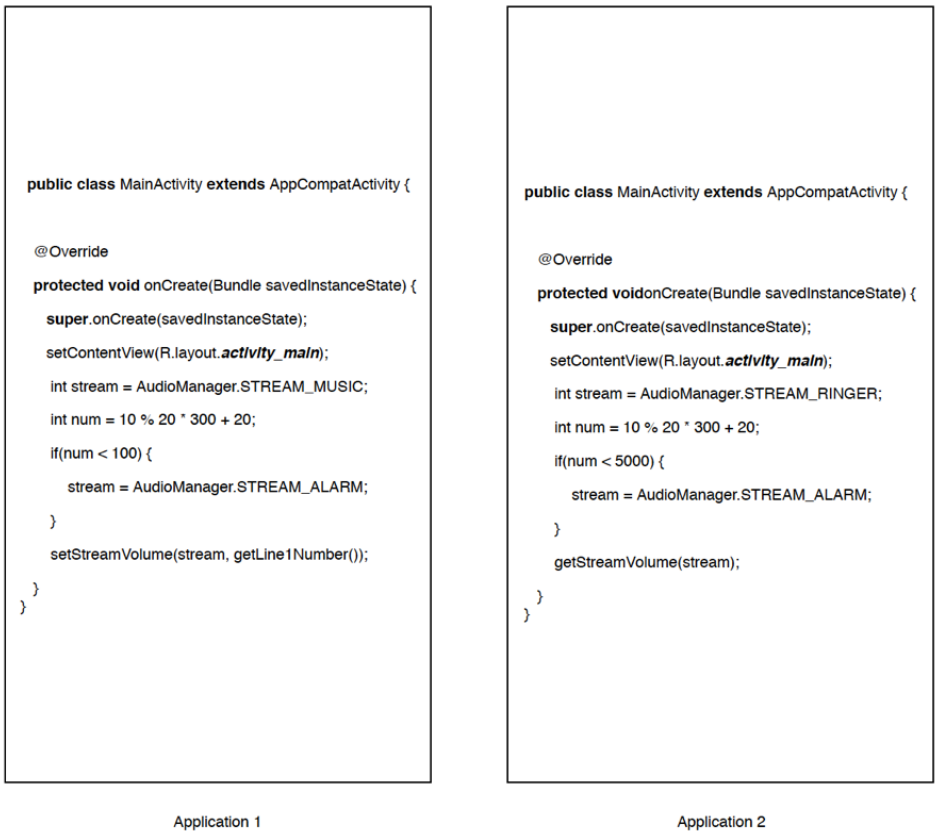

Figure 5: Two Android Applications using a Covert Channel

After we generate call graphs for these applications, application 1 contains the leaking statement setStreamVolume(), and application 2 contains the retrieving statement getStreamVolume(). We perform constant propagation on each statement and gather an approximation of intervals. Performing taint analysis on the statements, we find that setStreamVolume() gets its data from getLine1Number() which contains the phone number of the device. The statement getStreamVolume() does not leak its data once it is retrieved. Figure 6 shows the leaking and retrieving statements for each app and its associated data with each one.

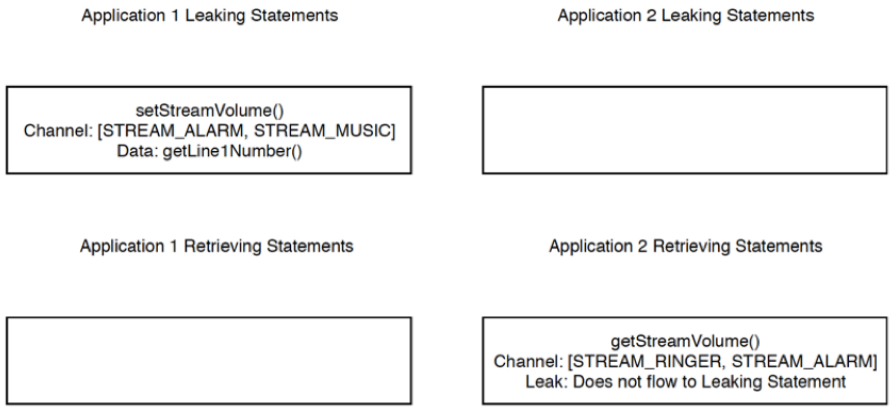

Figure 6: Leaking and Retrieving Statements for each App

After analyzing both apps, we begin to compute the graphs. Application 1 contains the statements setStreamVolume() and application 2 contains its corresponding retrieving statement in getStreamVolume(). Computing the intersection of interval approximations shows that there is an intersection between them, the interval [STREAM_ALARM, STREAM_ALARM]. Therefore, a covert channel is possible between these two applications and a path is constructed. Figure 7 shows the graph and result.

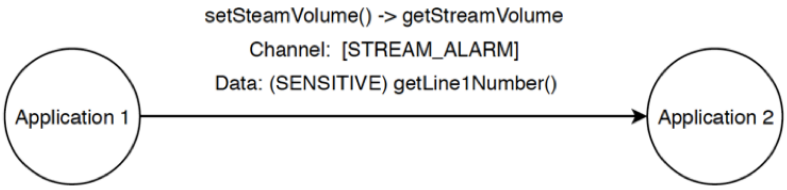

Figure 7: Graph and Final Result

Evaluation

We evaluated our implementation on a variety of Android applications from various sources. Our algorithm relied on a distinct set of tagged leaking, retrieving, and sensitive method calls from the Android SDK. In order to create a list of tagged methods, we used SUSI, which is a machine learning framework to automatically categorize methods in the Android SDK as leaking methods or methods that retrieve sensitive information [21]. This framework was helpful because it resulted in less time manually inspecting the Android API for leaking and sensitive methods. This framework only categorizes sensitive and leaking method calls, so we had to manually inspect if tagged leaking calls were able to be retrieved by some retrieving method. The appendix contains a table of the methods used for our evaluation.

In some cases, leaking methods were unable to communicate covertly to another application because a shared resource is not present. An example of this is the Android API call to send SMS messages. Sensitive information that flows into an SMS message is dangerous if the user is unaware, but another application has no way of picking this sent message up. Our analysis can detect information leakage, so these types of tagged leaks remain in our analysis.

We tested our implementation against Steganocc, which is a set of two applications containing working covert channels [5]. One application covertly sends data, and the other application covertly receives it. After analyzing these two applications, we were able to identify the seven covert channels across the two applications: Volume Settings, Intent Type, Unix Socket Discovery, File Lock, File Existence and File Size [5].

We also analyzed 402 applications from apkpure.com [8] and 513 malware samples from virustotal.com [27]. The site apkpure.com is a third-party app repository containing free applications for download. The site virustotal.com is an online malware repository that contains detected android application malware. We were not able to find any covert communication happening over these applications. In the malware samples from Virus Share, we located 26 applications that contained data leaks. After evaluating these leaks, we noticed that the longitude and latitude, IMEI number, phone number, MAC address, and SQL query calls were the most sensitive sources flowing to a leaking method. The most common leaking methods wrote to different log locations and passed data through Intents.

Leaking Apps

App	Leaking Method	Sensitive Method
1	i()	getLongitude() , getLatitude()
2	putInt()	getDeviceId()
3	e()	execSQL()
4	d()	getMacAddress()
5	i()	getLine1Number()
6	i()	getLine1Number()
7	putString()	getMacAddress()
8	d()	getLongitude(), getLatitude()
9	putString()	getLine1Number()
10	putString()	getLine1Number()
11	putString()	execSQL()
12	putString()	getLine1Number()
13	d()	getDeviceId()
14	putString()	getLine1Number()
15	e()	getLine1Number()
16	e()	getLongitude(), getLatitude()
17	d()	getLongitude(), getLatitude()
18	i()	getDeviceId()
19	putString()	getLine1Number()
20	e()	getDeviceId()
21	putString()	getLine1Number()
22	i()	getLongitude(), getLatitude()
23	putString()	execSQL()
24	v()	getMacAddress()
25	i()	getLongitude(), getLatitude()
26	v()	getDeviceId()

SHA256 Hashes for each app and additional information on the methods are listed in the Appendix

Conclusion and Related Work

In this paper, we created an algorithm that can be used to detect covert channel communication. The algorithm uses static program analysis to find leaking and retrieving method invocations and computes data flows to the method invocations. We implemented this algorithm to analyze Android applications and were successful on a set of created covert channel applications, SteganoCC [5]. A drawback of our algorithm is that it can only find covert channels that are known and tagged by a previous examination of the Android SDK. Applications may be using a covert channel that is not well known to the research community. In the future, we hope to uncover more covert channels and create our own set of applications that communicate covertly.

Our application currently makes approximations on the possible set of channels. Creating more complex constant propagation techniques to deal with code obfuscation to approximate the number of loop iterations and evaluate conditional statements is left for future work. These enhancements could overall improve the accuracy of our analysis.

Related work in covert channel detection deals with data leakage through interprocess communication and network calls in Android applications rather than covert communication. TaintDroid is a dynamic analysis tool that examines information flow in real time as an application executes [10]. It does not consider communication over more than one application. FlowDroid and AndroidLeaks are similar to TaintDroid but the analysis is performed statically [1] [13]. TaintDroid, FlowDroid, and AndroidLeaks are more interested in whether sensitive information generated is leaked out of the device by interprocess communication and network calls. Our algorithm considers leakage over more than just a single application and shows that it can be successfully used to detect covert communication before it happens.

References

[1] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[2] *AudioManager*. URL: https://developer.android.com/reference/android/media/AudioManager.html#summary.

[3] Serdar Cabuk, Carla E Brodley, and Clay Shields. "IP covert timing channels: design and detection". In: *Proceedings of the 11th ACM conference on Computer and communications security*. ACM. 2004, pp. 178–187.

[4] *Closure Properties of Regular Languages*. URL: http://infolab.stanford.edu/~ullman/ialc/spr10/slides/rs2.pdf.

[5] *Covert Channels for Android*. URL: http://steganocc.gforge.inria.fr/#method's-research-paper.

[6] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[7] Isil Dillig. *A Gentle Introduction to Program Analysis*. Jan. 2014. URL: https://www.cis.upenn.edu/~alur/CIS673/isil-plmw.pdf.

[8] *Download APK free online downloader*. URL: https://apkpure.com/.

[9] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. 1st. San Francisco, CA, USA: No Starch Press, 2014. ISBN: 1593275811, 9781593275815.

[10] William Enck et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5.

[11] *Finding Loops in Control Flow Graphs*. URL: http://pages.cs.wisc.edu/~fischer/cs701.f14/finding.loops.html.

[12] Wade Gasior and Li Yang. "Exploring covert channel in android platform". In: *2012 international conference on cyber security*. IEEE. 2012, pp. 173–177.

[13] Clint Gibler et al. "AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale". In: *International Conference on Trust and Trustworthy Computing*. Springer. 2012, pp. 291–307.

[14] *Intermediate Representation*. URL: http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf.

[15] Suman Jana. *Taint Tracking*. URL: http://www.cs.columbia.edu/~suman/secure_sw_devel/taint_tracking.pdf.

[16] Butler W. Lampson. "A Note on the Confinement Problem". In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615. ISSN: 0001-0782. DOI: 10.1145/362375.362389. URL: http://doi.acm.org/10.1145/362375.362389.

[17] *Mobile OS market share 2018*. Aug. 2018. URL: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[18] Anders Møller. *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. http://www.brics.dk/automaton/. 2017.

[19] Anders Møller and Michael I Schwartzbach. "Static program analysis". In: *Notes. Feb* (2012).

[20] Mitsunori Ogihara. *Regular Expression*. 2012. URL: http://www.cs.miami.edu/home/ogihara/csc527/new01-3.pdf.

[21] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A machine-learning approach for classifying and categorizing android sources and sinks." In: *NDSS*. Citeseer. 2014.

[22] Barbara G Ryder. "Constructing the call graph of a program". In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 216–226.

[23] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.26. URL: http://dx.doi.org/10.1109/SP.2010.26.

[24] Monirul I. Sharif et al. "Impeding Malware Analysis Using Conditional Code Obfuscation". In: *NDSS*. 2008.

[25] Gustavus J. Simmons. "The Prisoners' Problem and the Subliminal Channel". In: *Advances in Cryptology: Proceedings of Crypto 83*. Ed. by David Chaum. Boston, MA: Springer US, 1984, pp. 51–67. ISBN: 978-1-4684-4730-9. DOI: 10.1007/978-1-4684-4730-9_5. URL: https://doi.org/10.1007/978-1-4684-4730-9_5.

[26] Tim Teitelbaum. *Control Flow Graphs*. Mar. 2008. URL: https://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec24.pdf.

[27] *VirusShare.com*. URL: https://virusshare.com/.

[28] *wala/WALA*. Mar. 2019. URL: https://github.com/wala/WALA.

[29] Mark N Wegman and F Kenneth Zadeck. "Constant propagation with conditional branches". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.2 (1991), pp. 181–210.

[30] Mark Weiser. "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method". In: *PhD thesis, University of Michigan* (1979).

Appendix

Sensitive Methods

Class	**Method**
TelephonyManager	getDeviceId()
Location	getLongitude()
Location	getLatitude()
LocationManager	getLastKnownLocation()
BluetoothDevice	getAddress()
WifiInfo	getMacAddress()
Locale	getCountry()
WifiInfo	getSSID()
CellIdentityGsm	getCid()
CellIdentityGsm	getLac()
AccountManager	getAccounts()
TimeZone	getTimeZone()
TelephonyManager	getSubscriberId()
TelephonyManager	getLine1Number()
TelephonyManager	getSimSerialNumber()
CdmaCellLocation	getNetworkId()
Location	getBearing()
CdmaCellLocation	getSystemId()
ContentResolver	query()

SQLiteDatabase	execSQL()
HttpEntity	getEntity()
StatusLine	getStatusLine()
Intent	getByteExtra()
Intent	getCategories()
Intent	getComponent()
Intent	getDataString()
Intent	getDoubleExtra()
Intent	getFloatExtra()
Intent	getIntentOld()
Intent	getIntExtra()
Intent	getLongExtra()
Intent	getShortExtra()
Bundle	getDouble()
Bundle	getFloat()
Bundle	getInt()
Bundle	getLong()
Bundle	getShort()
Bundle	getString()
AudioRecord	read()
PackageManager	queryIntentActivities()
PackageManager	queryIntentServices()
PackageManager	queryBroadcastReceivers()
PackageManager	queryContentProviders()
SharedPreferences	getDefaultSharedPreferences()

Leaking and Retrieving Methods

Leaking Method	**Retrieving Method**
AudioManager setStreamVolume(channel, data, _)	AudioManager getStreamVolume(channel)
Intent new Intent(channel/data)	String substring(channel)
ServerSocket new ServerSocket(channel)	ServerSocket new ServerSocket(channel) ServerSocket.close()

File File = new File(channel) RandomAccess RA = new RandomAccessFile(File) FileChannel FileChannel = RA.getChannel() FileChannel.tryLock()	File File = new File(channel) RandomAccess RA = new RandomAccessFile(File) FileChannel FileChannel = RA.getChannel() FileLock FileLock = FileChannel.tryLock() FileLock.release()
File File = new File(channel) RandomAccess RA = new RandomAccessFile(File) RA.setLength(Data)	File File = new File(channel) RandomAccess RA = new RandomAccessFile(File) RA.size()
File File = new File(channel) File.createNewFile()	File File = new File(channel) File.delete()
AudioManager setVibrateSetting(channel, data)	AudioManager getVibrateSetting(channel)
AudioManager adjustStreamVolume(channel, data, _)	AudioManager getStreamVolume(channel)
AudioManager setStreamMute(channel, _)	AudioManager isStreamMute(channel)
AudioManager setRingerMode(data)	AudioManager getRingerMode()
BluetoothAdapater enable()	BluetoothAdapater disable()
WifiManager setWifiEnabled(_)	WifiManager isWifiEnabled()
Settings.System putFloat(_, channel, data)	Settings.System getFloat(_, channel)
Settings.System putInt(_, channel, data)	Settings.System getInt(_, channel)
Settings.System putLong(_, channel, data)	Settings.System getLong(_, channel)

Settings.System putString(_, channel, data)	Settings.System getString(_, channel)
Bundle putBinder(data)	Bundle getBinder(data)
Bundle putBoolean(channel, data)	Bundle getBoolean(channel)
Bundle putBundle(data)	Bundle getBundle()
Bundle putByte(channel, data)	Bundle getByte(channel)
Bundle putChar(channel, data)	Bundle getChar(channel)
Bundle putDouble(channel, data)	Bundle getDouble(channel)
Bundle putFloat(channel, data)	Bundle getFloat(channel)
Bundle putInt(channel, data)	Bundle getInt(channel)
Bundle putLong(channel, data)	Bundle getLong(channel)
Bundle putParcelable(channel, data)	Bundle getParcelable(channel)
Bundle putShort(channel, data)	Bundle getShort(channel)
Bundle putString(channel, data)	Bundle getString(channel)
Bundle putAll(data)	
Log d(channel, data)	
Log e(channel, data)	
Log i(channel, data)	
Log v(channel, data)	

Log w(channel, data)	
Log wtf(channel, data)	
OutputStream write(data)	
FileOutputStream write(data)	
Writer write(data)	
OutputStreamWriter append(data)	
Intent setClassName(_, data)	
Intent setComponent(data)	
Intent putExtra(_, data)	
MediaRecorder setVideoSource(data)	
MediaRecorder setPreviewDisplay(data)	
SmsManager sendTextMessage(_, data, _, _)	
SmsManager sendDataMessage(_, data, _, _)	
SharedPreferences.Editor putFloat(channel, data)	SharedPreferences getFloat(channel)
SharedPreferences.Editor putDouble(channel, data)	SharedPreferences getDouble(channel)
SharedPreferences.Editor putLong(channel, data)	SharedPreferences getLong(channel)
SharedPreferences.Editor putInt(channel, data)	SharedPreferences getInt(channel)
SharedPreferences.Editor putString(channel, data)	SharedPreferences getString(channel)

SHA256 Hashes of Leaking Apps

1. 00d862306f9c02aa954151ebc5814900c8f0c9876cf504b5358aaa1477481f81
2. 010da2454c0cd65e142934d666a94f8cdb0f3917d9a65976b9d7bcea5e0a6db0
3. b6782ac05a95b678b1f16fb4f53b8ea973964db8262c8a549ec10a9b34f9242c
4. b69c7afb953245e759e1cfc47a5ecde31c910813aba7bcb8abe5e0fc3f4ad88d
5. b816e856fced1c1ef52ce69024ca9c2d0d0305c6f511b4961ac3b68c21c4e57b
6. b88bf4cbc332a0ae7f1cd9db9834d86da5d5c5a6b4f030a1a2af42e283f3eab1
7. b91a5898e9000efeb79c7b1fe93498a95665b9b930a617107c35748a117590c9
8. b9e0fa5ce5b5cea6017608ad4e6593f6ef5c79ad92adced0afac2c21f378da7c
9. bbba7abbc40b2fceb822652085af3d6f73244d3a4b704091bfbc138a808cae67
10. bbcd4ef155798c3d846a457b4d3745a9b403e0b3cf660a306978da2f90f9a8fd
11. bc422ea47da0f9083b29db75740d2cb11d0c51a4c0c5e1abf4c022836b5e73fc
12. bc78bf28cec347ce8d8fd2f06f9509920b577bd97b9ec2acc5f37b528a356858
13. bcbef3534363bb1b1ac48ddefecbe0ccdd84b3d3d13060398c737b97a2dc9cde
14. 0491eee8d0ca37393dc23b4bcbc4a54b7caa63fa2d0f3554f477dd531dea6290
15. c8a67bd34b738c8f3335f9896d39c850573711bcab86fe2622e106bab9860cae
16. cce6c3ac5372d7c046900d17a614694cfd8d2e75574959eccadda247cc1df751
17. cef6af10868fbd5d9bcdb9333f72b4bc0ff9c3e775e2ba9ae5fc7c75b6085649
18. cf39ba0da55ec9d04c2a9597b637bfb0f1a9ef4c67bbd314906d475fdb043506
19. cf76e8f8d0cfac263bfc1df1441bd9a47981af85a76a640214141b2342747102
20. 1b182b9bd79c247dfdaee9be7820db55dc4c4c50e18b50cb95fa95aa738e252b
21. d6256495eb154306c5cac3ec55bb648b670e1980bb1369f8a29ec92c79d1d3fa
22. d6a38272f38a6834a457c589c8f7b351203d45a8350aa29a065871e7853b6f65

23. b507d548ecc6552ebca8fc37483e3c19a93e7c6997b6cf9091544b0cb21cfadb
24. b2be361abe0c4d78335a30ff97ba3a1c132b0057a2fb01c9f7e09fcf9bc40e96
25. b1dbdb96db0ece8b42968dfbcf8c4340eb549c246c39ea21fd34845e66d5639b
26. b0435e57793d2e7774537166500bfe2bbd3f7852b210e95dfb834c636261f00c