Detecting (Absent) App-to-app Authentication on Cross-device Short-distance Channels

Stefano Cristalli
stefano.cristalli@unimi.it
University of Milan
Milan, Italy

Long Lu
l.lu@northeastern.edu
Northeastern University
Boston, Massachusetts

Danilo Bruschi
danilo.bruschi@unimi.it
University of Milan
Milan, Italy

Andrea Lanzi
andrea.lanzi@unimi.it
University of Milan
Milan, Italy

ABSTRACT

Short-distance or near-field communication is increasingly used by mobile apps for interacting or exchanging data in a cross-device fashion. In this paper, we identify a security issue, namely cross-device app-to-app communication hijacking (or *CATCH*), that affect Android apps using short-distance channels (e.g., Bluetooth and Wi-Fi-Direct). This issue causes unauthenticated or malicious app-to-app interactions even when the underlying communication channels are authenticated and secured. In addition to discovering the security issue, we design an algorithm based on data-flow analysis for detecting the presence of *CATCH* in Android apps. Our algorithm checks if a given app contains an app-to-app authentication scheme, necessary for preventing *CATCH.* We perform experiments on a set of Android apps and show the *CATCH* problem is always present on the whole analyzed applications set. We also discuss the impact of the problem in real scenarios by presenting two real case studies. At the end of the paper we reported limitations of our model along with future improvements.

CCS CONCEPTS

• **Security and privacy → Authentication**; *Mobile and wireless security*; Software and application security.

KEYWORDS

android, data-flow analysis, authentication protocols, mobile security

ACM Reference Format:
Stefano Cristalli, Long Lu, Danilo Bruschi, and Andrea Lanzi. 2019. Detecting (Absent) App-to-app Authentication on Cross-device Short-distance Channels. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3359789.3359814

1 INTRODUCTION

Cross-device communications allow nearby devices to directly communicate bypassing cellular base stations (BSs) or access points (APs) [13, 17, 18] . Such a paradigm can bring many benefits, such as spectral efficiency improvement, energy saving, and delay reduction. Without the need for infrastructure, such a technology enables mobile users (e.g., Android) to instantly share information (e.g., pictures and videos) with each other, even in areas without cellular coverage or access points [21]. It is also becoming an important technology for mobile social networks [12]: friends close to each other can be automatically identified and paired up. Moreover, this technology is used to establish the so-called mobile ad-hoc clouds, which take advantage of unused resources of nearby devices to provide cloud services, such as data and computation offloading. This is also a typical case of IoT environment, where IoT devices communicate with each other on short-distance channels [10].

Several solutions exist for securing cross-device communication. In the Android environment, they allow authentication of devices and communication channels [11, 20]. However, these solutions are not sufficient to protect the entire communication flow. Specifically, the proposed protection system in [11] restricts apps' access to external resources, such as Bluetooth, SMS and NFC, by defining new SEAndroid types to represent the resources based upon their identities observed from their channels. The policies bind an app to a particular device on a specific channel. In this case, a malicious app installed on one device, which is allowed to communicate with a paired phone, can interfere with the communication and inject data on the channel. This problem is due to the fact that the authentication between apps is missing, and such authentication is needed in addition to the device-level and channel-level authentication. One can solve this problem by designing Android access control at system level for preventing an unauthorized access to communication channel (e.g, Bluetooth) during security operations, and removing public resources for stopping side-channel attacks [20]. This, however, makes the system less usable and compatible for the apps that already use the public resources for legitimate purposes. Moreover, these systems do not handle channels such as: SMS, Audio, Wi-Fi and NFC. We name this security issue cross-device app-to-app communication hijacking, or *CATCH.* We argue that *CATCH* is critical and is due to the fact that no APIs or mechanisms

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12.
https://doi.org/10.1145/3359789.3359814

are made available to Android programmers for performing app-level authentication on short-distance channels (e.g., Bluetooth, Wi-Fi-Direct).

In this paper, we study the problem of mutual authentication between two apps running on two different devices and communicating over a short-distance channel. Although such channels already provide device pairing and authentication methods, these methods only operate at the device or channel level. They are oblivious to the apps running on the devices. In this study, we first define the authentication scheme for short-distance channels. We then design a new tool that is able to analyze a given Android app and detect potential *CATCH* vulnerabilities (i.e., the lack of app-to-app authentication). Our tool uses several data-flow analysis techniques and is able to recognize specific *if-statement* conditions in the code related to the authentication scheme. Such particular conditions can be precisely recognized since, in our context, the analyzed authentication model must be performed with some sort of dynamically generated secret (out-of- band authentication) (Section 3.1) that is usually stored in the dynamic memory (e.g. heap, stack). We perform some experiments to show the flexibility of our tool on detecting authentication schemes, even when the target app has been manipulated with the ProGuard obfuscator, one on the most used obfuscators for Android [3]. Our tool can be deployed in several contexts: it can serve as a tool for the developer, or it can scan apps in distributing environments (e.g. Google Play) for detecting potential vulnerabilities on Android apps that communicate by using short-distance channels.

In summary, this paper makes the following contributions:

- We identify a **security problem** called cross-device app-to-app communication hijacking (CATCH), which commonly exists in Android apps that use short-distance channels, and afflicts all the tested Android versions. We perform experiments on a dataset that contains 662 Android apps that use Bluetooth technology, collected in the Androzoo repository.
- We provide a **solution** to the CATCH problem by designing and developing an authentication scheme detector that analyzes Android apps to discover potential vulnerabilities. We tackle several challenges in identifying code boundaries of the authentication scheme, along with the authentication checks.
- We **validate** the results of our system on Android apps with manual analysis, and test its resilience in detecting the authentication scheme. The results show that our approach produces 0% of false positives and false negatives. We also show two case studies on real Android apps.

2 BACKGROUND

In this section we provide the necessary background to understand the security vulnerabilities in Android apps performing peer-to-peer communication.

2.1 Authentication for Cross-device App-to-app Communication

In this paper we study the problem of mutual authentication between two apps running on different devices and communicating over a short-distance channel. Although such channels already provide device pairing and authentication methods, these methods only operate at the channel level: they allow two devices to be paired and mutually authenticated (i.e., establishing a channel) but they are oblivious of the apps running on the devices (i.e., all apps on these devices share this established channel). As a result, when two devices are paired and authenticated at the channel level, it is possible for a malicious app on one device to interfere in a communication on the channel between two legitimate apps.

Currently, most cross-device, peer-to-peer communications channels are authenticated by using an *out-of-band* scheme that works as follows. A user (**requesting user**) A initiates a communication from his device to a nearby device, whose user (**accepting user**) B is then prompted with confirmation. The confirmation is requested either with a secret PIN that B has to communicate to A via a separate channel (e.g., verbally), or as a simple “accept” button presented along with information that enables the identification of the device trying to initiate the communication. These steps are already implemented in Android; one never needs to re-implement authentication for the communication channel. Once authentication has passed, communication can begin. Bluetooth uses encryption to protect the channel. It is important to note two points related to authentication in this scenario:

(1) Authentication occurs via sharing of *out-of-band* information/secret.
(2) Authentication performed on the channel (Figure 1) is not sufficient to guarantee authentication between higher level applications communicating over the channel.

Point 1) is important as a general property of authentications performed in our scenario. The exchanged information needed to confirm authentication is, in practice, visual and verbal contact between the two users, and the out-of-band element is a constant in all this type of authentications. More strongly, we exclude the possibility of authentication being carried out exclusively via information passed on the same channel being authenticated, as a result of previous research [8, 26].

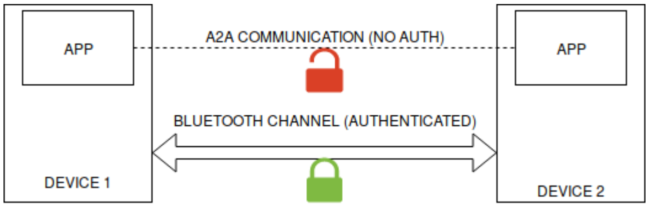

APP
A2A COMMUNICATION (NO AUTH)
APP
BLUETOOTH CHANNEL (AUTHENTICATED)
DEVICE 1
DEVICE 2

Figure 1: Authentication of A2A communication is not guaranteed by channel authentication

To understand why the lack of app-level authentication is dangerous, let us consider the following example (Figure 2): a chat app using Wi-Fi-Direct opens a `ServerSocket`, accepting communication through it and display incoming messages to the final user.

The intended use of the app is to be installed on two devices that communicate with each other in a peer-to-peer fashion. We also consider the presence of a malicious app on one device, this is a common threat model, as shown in [20]. Since the *devices* are authenticated, and not the *apps*, the malicious app has permission to communicate over the channel, as any other app installed on the device. The malicious app can therefore craft custom messages to send to the other device, which are displayed as if they were sent from the original app. If there is no code performing authentication in the benign app, there is no possibility of detecting this sort of action.

Depending on the particular context, there are some scenarios in which the attack can become very dangerous:

- *Phishing*: in cases like the example described above, the malicious app could send phishing material to the other app. The user will be likely to trust and open the content, as he will have no means to distinguish it from benign content sent from the device communicating with him.
- *Malware delivery*: the same system could be used to deliver malware to the user, in the form of malicious files that would trigger a vulnerability upon opening (for instance, a malicious PDF file that targets a vulnerable PDF reader).
- *Exploitation*: if the target app performs internal operations depending on commands received from the communication channel, the malicious app could send commands that could change the execution flow and trigger unwanted behavior. For instance, a command to delete the user data could be issued to a file manager app that accepts operations via Bluetooth device.

It is important to note that other network attacks such as MiTM are difficult to accomplish in this context, since the attacker should be physically close to the devices in order to hijack the communication, and would also need to overcome or bypass the channel protections, such as encryption. For this reason we believe that the attack explained above is the most realistic in this environment. We name the underlying problem common to all these scenarios **cross-device app-to-app communication hijacking**, or *CATCH*. Having established the potential impact of the problem, we aim at building a system for automatic analysis of Android apps, targeted at detecting the presence (or lack thereof) of authentication on particular communication channels. The purpose of our system is to provide a tool that will help app developers to secure their software. Moreover, our detector can also be used as a security scanner on app markets (e.g., Google Play) for detecting potential authentication vulnerabilities.

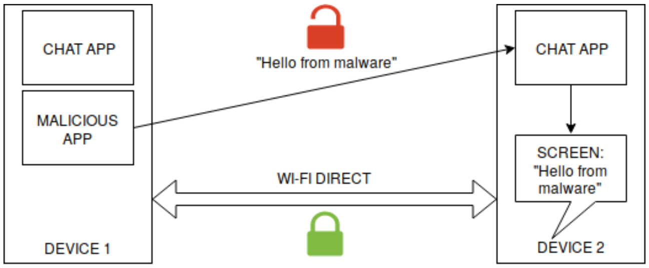


Figure 2: Malicious app sending content to chat app

3 APPROACH OVERVIEW

This section describes the design and structure of our approach. We build our system with the goal of automatically verifying the existence of app-to-app authentication in Android apps. To detect app authentication in an automated way, we mainly face the following challenges:

C1) We need to define a generic scheme that captures the essential logic of app-to-app authentication. Such a scheme is necessary for identifying and evaluating the implementation of authentication in apps. (Section 3.1)

C2) We need to define a strategy for differentiating between an if-statement that does not operate on security critical data and an if-statement that is a part of the authentication scheme. (Section 3.2)

C3) Additionally, the authentication scheme can be implemented in several ways according to the developer experience. This adds an additional layer of difficulty for our analysis, that should be general enough to also capture such cases. (Section 3.2)

We now proceed to illustrate our approach for building an analysis tool that is able to tackle these challenges and provide accurate results in terms of detection.

3.1 Authentication Definition

In this section we define an authentication scheme for cross-device communication in Android environment. More specifically our authentication model considers two devices, D1 and D2, with apps A1 and A2 respectively installed. The two devices establish an authenticated channel, on top of which A1 and A2 initiate a communication. Such form of authentication proposes authenticated information exchange between mobile devices using several methods different than the standard RF channel [26]. These are called out-of-band, side-channels or location-limited channels (LLCs) [28], and include audio, visual, infrared, ultrasound, and other forms of transmission [7, 23, 24, 27]. Such techniques allow the receiver to physically verify the source of the transmission. Using this information, the devices are mutually authenticated, and a secure shared key can be established. More precisely in such an authentication scheme we recognize the following steps:

1. A2 obtains a secret that will be used to authenticate communication. This secret is either generated on device D2, and then communicated to app A1, or it is generated by D1 and then shared with A2. Such a communication uses an *out-of-band* channel which is also called a “human assisted channel”. Such a channel cannot be manipulated by an attacker, and thus it is considered trusted by definition.
2. Once A1 and A2 share the same secret, they can start sending data, using the secret as authenticator. Depending on what the secret is and the application protocol, the data could be encrypted with a key derived from the secret (e.g., *Hash(Secret)*), or the secret could be sent as plaintext along with the data for authenticating the transmission.
3. In both cases (encryption with key derived from the secret, or secret sent with data as a simple pass-phrase/PIN), app A2 needs to perform authentication checks on the received data. In the first case, A2 needs to check that the decryption

operation performed by the secret key is correct, and in the second case A2 needs to check whether the pass-phrase/PIN is correct. These checks must occur before any critical use of the data, otherwise the communication is not authenticated. Only in case the checks are correct, the data is authenticated and the communication can continue.

We mentioned "authentication checks" that are performed in step 3. It is crucial to define what form these controls might assume, in a way that helps us target their recognition in code. Moreover such a definition should be general enough to capture the majority of several forms of the authentication schemes deployed by different developers. We define a **communication** in our model as some exchange of data from A1 to A2, beginning when A2 reads the data from the communication channel. We define a **use** of the data as any operation whose result depends on the data itself. We define an **authenticated use** of the data as any instruction that needs to be authenticated before access to the data. We give the following **definition of authentication** in our model: given a communication over a peer-to-peer channel with exchanged data `D`, an authentication is a condition in code situated between the beginning of the communication and the first authenticated use of `D`, which either: (1) allows the execution to continue, in case `D` is successfully authenticated, or (2) prevents any authenticated uses of `D` every time the authentication is unsuccessful. The internal logic of the authentication checks depends on the context, and is therefore not possible to include it in the definition.

3.2 Detection of Authentication Scheme

For detecting authentication, we first explore the possibility of identifying authentication schemes via the use of particular APIs. If such APIs existed, then we could reduce our analysis to a code reachability problem. This is the case, for instance, of authentication over Unix domain sockets [25]. Unfortunately, we could not find any standard APIs for app-level authentication for the technologies we analyzed. For this reason, we shift our focus on detecting a set of instructions in the code that might indicate the presence of an authentication mechanism. In such a context we must clearly define a strategy for identifying possible authentications once we track the data of our interest. The first step for creating a scalable analysis framework is to identify boundary code points in the application. Such boundary permits to restrict the analysis only to a part of code that potentially could contain an authentication scheme. After the boundary area is identified we can apply further code analysis techniques in order to validate the authentication scheme. In our system the boundary area is defined by two main elements: the *entry* and *exit* points.

More specifically, an *entry point* is an instruction in the code that indicates the start of the communication over the analyzed channel (e.g., data receiving). Given this broad definition, we can recognize multiple entry points in an application for a given communication. For example, In Listing 1 we can see an example of Bluetooth communication in Android app. Since the data is read from the stream at line 13, the instruction represents an entry point. The call `socket.getInputStream()` at line 11 is also an entry point for this communication. We are obviously interested in entry points that help to indicate the start of communication for a specific channel such as Bluetooth. An accurate identification of the entry points for a communication channel will ensure that all possible communications over such channel are identified and targeted by our analysis.

Listing 1: Sample Bluetooth socket communication

```
1  try {
2    socket = mmServerSocket.accept();
3  } catch (IOException e) {
4    Log.e(TAG, "Socket's accept() method failed", e);
5    break;
6  }
7
8  if (socket != null) {
9    InputStream inputStream;
10   try {
11     inputStream = socket.getInputStream();
12     byte[] buffer = new byte[10];
13     inputStream.read(buffer);
14     if (buffer[1] == 10) {
15     writeToFile(buffer);
16     FunctionLibrary fl = new FunctionLibrary();
17     writeToFile(fl.return6());
18   }
19   mmServerSocket.close();
20   } catch (IOException e) {
21     e.printStackTrace();
22   }
23   break;
24 }
```

The end of the boundary is defined by an *exit point*. An *exit point* is represented by the first *authenticated use* of the data coming from the monitored channel. Even though exit points exist for every communication, it is hard to define whether an exit point is an authenticated use of the data or not, since this is a semantic property of an use. As an example, the use of line 15 in Listing 1, where the data is written to file, may or may not be an authenticated use, depending on what the file is used for. If it is a log file used simply for debugging purposes, and virtually never checked unless an error occurs, then it is not important that authentication necessarily occurs before such point. On the other hand, if the data defined into the file is part of the main flow of the app protocol, then authentication must necessarily occur in order to avoid untrusted and potentially dangerous data in the file.

Due to this ambiguity of the use of the data, we design a detection strategy that is not dependent on exit points. In particular we design an algorithm (Algorithm 3.1), based on program analysis techniques, that performs data and control flow analysis. The algorithm starts computing the Control Flow Graph (CFG) and Data Dependency Graph (DDG) for each analyzed app (line 7-8). Both graphs are necessary to find out the relationships between data of our interest and the condition statements that depend on such data. Then, for each node in the CFG, the system determines whether it is an entry point by using function `isEP`. This function uses a pre-defined table based on function signatures related to a specific communication channel (Section 4.2). If no entry points are found, the result NO AUTH NEEDED is returned (lines 9-12). In all the other cases, each node in the DDG is analyzed. If the node represents a condition in the code (function `isCondition`), then the system checks if there exists a path in the DDG that connects an entry point to the conditional node (lines 16-17).

Algorithm 3.1: Authentication detection

```
1   input: APK app
2   output: NO AUTH NEEDED |
3           NO AUTH FOUND |
4           POSSIBLE AUTH FOUND
5
6     entry_points ← []
7     cfg ← computeCFG(app)
8     ddg ← computeDDG(app)
9    foreach node in cfg
10     if isEP(node) then entry_points.add(node)
11   end
12   if entry_points == [] then return NO AUTH NEEDED
13
14   foreach node in ddg
15     if isCondition(node) then
16       foreach ep in entry_points
17         path ← findPath(ep, node, ddg)
18         if path != null
19         then
20           if isCheckConstant(node, ddg) == false
21           then return POSSIBLE AUTH FOUND
22         endif
23       end
24     endif
25   end
26
27   return NO AUTH FOUND
```

If such a path exists, it means that we possibly found an authentication scheme. However it is still possible to obtain false positives: simple sanity checks or other controls on data would be all erroneously identified as authentication. In order to reduce the number of false positives among conditions that are candidate for authentication, the algorithm applies a *constant propagation* technique. Technically speaking, such technique is using reaching definition analysis results. In particular, if a constant value is assigned to a variable, and such variable is not modified before a point `P` in code, then the variable has a constant value at `P` and can be replaced with the constant.

In our context, since the analyzed authentication model must be performed with some sort of dynamically generated secret (out-of-band authentication, Section 3.1) that is usually stored in the dynamic memory (e.g., heap, stack), by using constant propagation we can discard all the conditions that use constant values in their comparison, as they certainly do not represent authentication on data. Constant propagation is a very powerful technique for our analysis, and it helps to reduce the false positives to 0% in our experiments as we will show in Section 5.2.

4 SYSTEM IMPLEMENTATION

We now discuss our practical implementation choices for the algorithm presented in the Section 3.2, by describing the technical details of our system.

4.1 Overview

We implemented our system on top of the Argus-SAF framework [29]. The framework offers various tools for analyzing Android apps, such as the generation of the CFG and DDG that we need in our algorithm. Also, the framework translates Dalvik bytecode into an intermediate representation (IR), called Jawa, on which our algorithm performs the analysis. In particular, various conditions in code, including `while` and `for` loops, `if` statements and exception `try/catch` blocks, are all translated into `if` statements in the intermediate representation. The CFG and DDG built by the framework contain nodes that map to single Jawa instructions, making it possible to have the fine-grained, instruction-level information that we need in our algorithm for targeting conditions. Furthermore, Argus-SAF permits inter-component modeling, meaning that transitions between components such as Android intents are integrated in the graphs. These features made possible for us to explore the application code together with the graphs built by Argus-SAF on top of it. Our system is composed of three main components: (1) Graphs Builder, (2) Path Finder and (3) App-to-app Authentication Finder. Our framework accepts an app in input (as an APK file), and outputs either that no authentication has been found, or a list of specific instructions in code that may contain authentication checks.

- The *Graphs Builder* starts the Argus-SAF analysis on the APK. The framework applies four sequential steps: (1) the Jawa IR is generated from the Dalvik bytecode, then (2) an environment model of the Android system is generated. This is crucial to capture the control flow and interactions between components, such as the dispatch of intents between activities. (3) At this point, Argus-SAF builds an inter-component control flow graph (ICFG) of the whole app. At the same time, it performs data flow analysis and builds an inter-component data flow graph (IDFG) on top of the ICFG. (4) Finally, the framework builds a data dependency graph (DDG) on top the IDFG. We mainly use information from this graph in our analysis. The information of our interest is extracted in the Graph Builder by using classes `ComponentBasedAnalysis` and `InterComponentAnalysis` for extracting the CFG and DDG. The graphs are then passed to the next component.
- The main goal of the second component, *Path Finder*, is to locate areas in the code where an authentication scheme may exist. This is done by identifying data flows for the protocol of our interest, and performing reaching definitions analysis to see if any conditional statement operates on data read from the channel that we are inspecting. The Path Finder component traverses the CFG received from Graphs Builder, and marks entry points for the analyzed channel based on a predefined list of method signatures. It then finds all conditional statements, which is accomplished by extracting all the nodes of type `IfStatement` in Argus-SAF. At this point, it is possible to perform reaching definition analysis, to check

whether there is at least one conditional statement using a variable that was earlier defined as data read from the channel. The DDG obtained from Graphs Builder contains all the information to perform this search: definition-use pairs map to edges in the graph, so Path Finder traverses it in order to find possible authentication paths. It sends the discovered paths, if any, to the last component.

- *App-to-app Authentication Finder* applies further checks to the paths received from Path Finder, in order to exclude false positive results by recognizing checks against constant values. In particular, it analyzes the `if` statements in the Jawa IR, which can be divided into two types: (1) comparisons between two variables, (2) comparisons between a variable and a constant. The system immediately discards the conditions of the second type from our search, as they certainly do not represent the authentication scheme that we look for (see Section 3.2). For conditions of the first type, our system uses constant propagation to determine if one of the two variables in the condition is a constant. It walks up the DDG from the `IfStatement` to their definition, reconstructing the value-history of the variables from their initialization. If the last-assigned value to either of the two variables (before the `IfStatement`) is a constant, then we are in the same case of type-two conditions, and the path is again discarded for the same reasons.

4.2 Choice of Entry Points

In our implementation we focused on Bluetooth, since it is the most used technology in Android apps for short-distance communication. Wi-Fi-Direct is still not very common among the Android apps, in fact in the dataset that we analyzed we only found a few samples (10) of it. To show the security issue of CATCH applied on Wi-Fi-Direct channel we analyzed one of these apps as a case study (Section 6.2). However, the core of our analysis is orthogonal to any communication channel. The only part that can change among different channels is the identification of the entry points. For Bluetooth communication based on `BluetoothSocket`, we found two possible entry points (i.e., where a `BluetoothSocket` stream starts receiving data): `BluetoothSocket.getInputStream` and `InputStream.read`. A typical Bluetooth communication flow involves the former function, called to obtain an `InputStream` object, followed by an invocation to the latter function. In the DDG of an application containing this type of communication, the instructions operating on data read from the channel are linked to both functions. It would appear that `InputStream.read` is the best choice for an entry point: semantically, it actually represents the point in which the data from the stream enters the control flow of the app. However, given the general use of class `InputStream` outside the context of Bluetooth communication, this choice led to many false positives in practical experiments. For this reason, the choice of `BluetoothSocket.getInputStream` worked much better as definition for our entry point for Bluetooth. Although it is an instruction *preceding* the actual read operation of data from the Bluetooth stream, it uniquely identifies our protocol of interest. Moreover in all the communication flows that we observed in Bluetooth apps operating on BluetoothSockets, the functions are always used in pairs.

5 EXPERIMENTAL EVALUATION

In this section we present and discuss the results about the experiments we performed to validate our system.

5.1 Preliminary considerations

In order to test the efficacy of our algorithm we need to collect a balanced dataset that contains both positive (i.e. apps with authentication at application level) and negative samples (i.e. apps without authentication). In our analysis we noticed that the security problem of CATCH afflicts all the Android apps using Bluetooth in our dataset. For this reason the dataset is unbalanced. To test our system under such conditions, we divided the experiments into two main categories: (1) a dataset analysis on APKs retrieved from a research repository [5], aiming at confirming the efficacy of the algorithm on negative samples; (2) a targeted analysis on custom apps built by applying code transformation techniques (e.g. obfuscation) for proving that the authentication scheme is correctly detected by our algorithm.

5.2 Dataset analysis of Android apps

To evaluate the efficacy of our system, we ran tests on a large number of APKs collected from the Androzoo repository [5]. The Androzoo dataset contains more than three million unique Android apps, crawled from several Android markets: Google Play, Anzhi and AppChina. In our experiments we pre-filtered APKs from the dataset and selected non-obfuscated apps that use Bluetooth. We decided to focus only on Bluetooth apps considering the amount of manual analysis we performed during the design of our algorithm, which could help us as a ground truth for validating our results. We started analyzing a total of 210,425 APKs, randomly chosen from the Androzoo repository. In order to select the appropriate Bluetooth APKs we applied the following filter: check if an app (1) requires the Bluetooth permissions in the manifest file; (2) contains certain libraries and classes related to Bluetooth (e.g., `BluetoothSocket`). The filter produced a total of 2,739 APKs.

We then applied a second filter where we exclude the obfuscated apps since it is quite hard validate them at this first step. For this filtering we focus on the ProGuard obfuscation tool, which is the free software most commonly used by developers, and it is referred in the Android Documentation [3]. In particular, we implemented some heuristics for recognition based on the typical class names (e.g. `a.class`) produced by ProGuard in obfuscated APKs. This filter selected a total of 942 APKs from the initial set of 2,793, which means that the majority of the apps in our dataset, almost 70%, use ProGuard for code obfuscation. After running our algorithm, we discovered that 704 of the selected apps do not have any entry point for Bluetooth communication in the CFG. This happens in cases where Bluetooth functionality is imported in some library/classes, but never used in the code, so the instructions that we would mark as entry points for our analysis never appear in the CFG/DDG. We also excluded such APKs from our dataset. Finally, we obtained a number of 238 APKs, suitable for our analysis and evaluation.

We then performed our first experiment. We ran our system on the 238 APKs without constant propagation enabled. This experiment shows the important role of the constant propagation technique on reducing false positives. It shows that 26 APKs out of 238 are found positive (i.e., about 11% of the APKs are potentially performing authentication on data read from Bluetooth sockets) and the rest are found negative (i.e., not performing authentication). they never applied any checks on data received form the Bluetooth channel.

In our second experiment, we enabled the constant propagation technique and we ran our system on the same set of 238 APKs. In this case we observed that all of the positive samples found previously were actually false positives (i.e., they used one constant value in the parameter of the if statement marked as possible authentication). This result shows that no app in the dataset performs app-to-app authentication when using Bluetooth.

At this point, we manually investigated the negative cases to check for any false negatives. To this end, for validating our results we chose a sample of 20 random APKs from our dataset of 238 APKs and we manually analyzed them. We observed that all of them receive data from the Bluetooth channel, but they never apply any checks on such data before using it. Our manual analysis found 0 false negatives. Our experiments show that 100% of the analyzed APKs in our dataset which perform Bluetooth communication using Bluetooth sockets are potentially vulnerable to the CATCH attack model.

5.3 Dataset composition

We analyzed the composition of our dataset to make sure that we did not run tests on sample/unused/abandoned apps. We sampled 300 APKs (containing permissions/classes for Bluetooth) from our dataset, and performed a manual analysis by searching them on Google Play. We found that about 30% of the apps were present on this market. We classified the apps by category, depending on their description. The vast majority of apps belongs in the following categories:

- *Game apps*, where Bluetooth is used for playing peer-to-peer
- *IoT apps* for specific devices, where Bluetooth is used to send and receive data from the controlled device or sensors
- *Business apps*, using Bluetooth to send data from smartphone to computer, or again smartphone to device

Other categories with less APKs included health apps, used for communicating with medical devices, cryptocurrency-related apps, and smart home management apps.

5.4 Targeted analysis

For our second analysis, we built a custom app using Bluetooth. It only performs these basic operations: it reads from a `BluetoothSocket` when the user triggers an action, and it displays any received content on screen. We then patched the app to include a basic authentication scheme fitting our model: upon starting, the app generates a random secret PIN of 4 digits, and shows it on the screen. This secret needs to be communicated out-of-band to other apps interacting with ours (e.g., verbally to another user wanting to send data). When reading from the `BluetoothSocket`, the app first expects to receive the PIN in plaintext, in the first four bytes read from the socket's `InputStream`. If the PIN matches the one generated by the app, the communication is accepted; otherwise it is rejected and the user is informed of the event. We found that our algorithm correctly predicts the possible presence of authentication.

We ran another test to check if changes introduced by common optimization and obfuscation tools would impact our algorithm. In this case we validate the obfuscation transformation since we can check the ground-truth provided by our application. In particular, we used the ProGuard tool [2] on our sample app, since it is the most commonly used by developers and it is recommended in the Android documentation [3]. ProGuard performs a series of transformations aiming to remove unnecessary code, and renames types and variables to hinder reverse engineering. We ran ProGuard on both versions of our test app (with and without authentication). We found that the transformations introduced by this tool do not impact the detection capabilities of our algorithm, which correctly discriminates the apps' behavior. In particular, we observed the following results:

- Sample app without authentication and ProGuard disabled, the system returns `NO AUTH FOUND`.
- Sample app without authentication and ProGuard enabled, the system returns `NO AUTH FOUND`.
- Sample app with authentication and ProGuard disabled, the system returns `POSSIBLE AUTH FOUND`.
- Sample app with authentication and ProGuard enabled, the system returns `POSSIBLE AUTH FOUND`.

5.5 Analysis of obfuscated APKs

Our results from the targeted tests indicate that ProGuard transformations do not affect the precision of our tool in the detection of authentication. For this reason, we decided to run our tool on ProGuard-obfuscated APKs from our dataset. We selected the 1797 obfuscated APKs that were initially discarded, and filtered them for Bluetooth use and appearance of entry points in the CFG/DDG as we did for non-obfuscated ones. This process yielded a total of 424 APKs, which we analyzed (combined with the previous experiments, we have a total of 662 APKs analyzed that use Bluetooth technology). 100% of the APKs were identified as negative (i.e., not containing authentication) by our tool, with constant propagation enabled. To validate this result, we manually analyzed 15 APKs, randomly chosen from the obfuscated APKs dataset. Since our tool indicates where the entry points are located in the CFG, and what the possible authentication paths have been analyzed, we were able to manually validate the absence of authentication checks, confirming that our heuristic approach is not only powerful enough for detection, but also that it is resilient to the obfuscation techniques.

5.6 Performance Analysis

In this section we report the time needed for each phases of our analysis.

Time Threshold. One of the main critical point for our analysis is how to set a time threshold for building the CFG and DDG in Argus-SAF, since the computational complexity explodes for large applications, and the system is not able to construct the entire graphs within reasonable time. After this threshold is hit while analyzing a single component in an APK, Argus-SAF will stop its analysis and move to the next component. In order to set up a correct time threshold we need to be sure that the constructed CFG and DDG include the Bluetooth entry points and the authentication checks (if present). To this end, we performed some experiments on APKs collected in our dataset. In particular, for each analyzed app we first built the graphs by setting a certain time threshold T, and we then search for Bluetooth entry points inside the computed CFG. Afterwards, we compute the number of nodes that are dominated by the entry point node in the graph that represents the number of instructions that can potentially include the authentication scheme. We start with a threshold of T = 30 sec., and then increase the value to T = 60 sec. and T = 120 sec. By comparing the different results, we notice two important things: (1) for any entry point, both the number of reachable nodes in the CFG and the number of data dependency nodes in the DDG are sufficient to contain a potential authentication scheme. More in details we found on average more than 10,000 instructions that are dominated by the entry point and the CFG reachability from a single entry point to any node in the graph is always above 99%, an expected result given by the inter-component connections in Android code. (2) The variation of the results between the three runs is minimal, that it means that we generally do not miss any important information that would have been considered adding more time of analysis. For this reason we chose a threshold of 30sec. for our experiments.

Time of Analysis. For our tool a use case would be code validation where the detector could serve as a pre-release tool to check for unauthenticated communication. In such a context the tool should perform its analysis in a short-time. In this direction we perform several experiments that show the overhead of the analysis. In particular the experiments were performed on a laptop running Ubuntu Linux 17.10, with a Intel Core i7-6700HQ CPU (2.60GHz) and 16 GB of RAM. We specifically measured the time taken to analyze the 26 apps that were found positive by the first version of our system (without constant propagation). The average time spent for modeling the APK in Argus-SAF is 5 minutes, while the average running time of our algorithm on the generated graphs is 2 minutes, giving a total average time of 7 minutes. Although the variance is high, we think that even the worst-case execution time is suitable for the use cases we designed, considering that the release of an app is not an instantaneous process, and that an average of 10 minutes is a feasible testing time for an automated developing pipeline of Android apps. Moreover we can decrease the time threshold for building graphs from 2 minutes to 30 sec. and gain more efficiency by reducing the average time from 7 to 5 minutes in total.

6 CASE STUDIES

In this section, we present two real attacks case studies that we select from our dataset in which our analyzer gave negative results. Such applications are representative of the common type of applications that can be used in peer-to-peer communication environment: (1) chat app, (2) data sharing app. We will now discuss the attack implementation, and the engineering effort required for its setup and execution along with its own limitations.

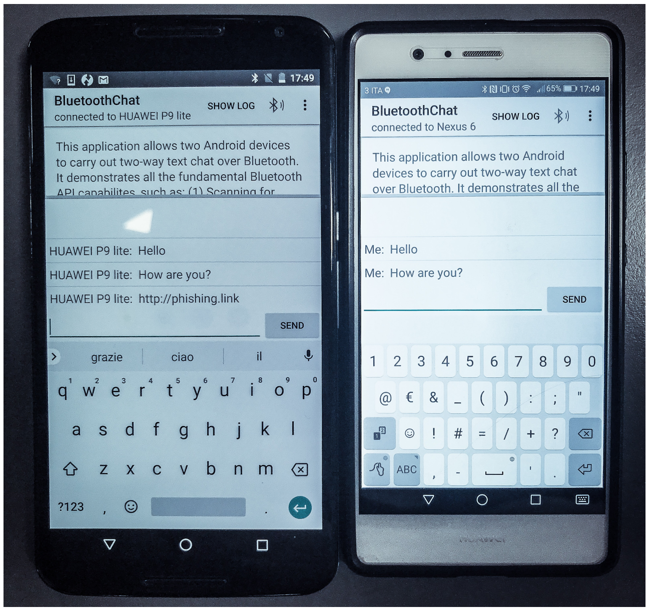
6.1 Data injection on BluetoothChat

We target the Android BluetoothChat app [1]. This app is a working example of peer-to-peer chat that is affected by CATCH problem, since it does not implement any app-level authentication scheme. The BluetoothChat app gives the user the possibility to scan for nearby devices, connect to one of them by using RFCOMM identifier, and then send text messages via Bluetooth. In this attack scenario we will describe a *data injection* attack to a remote device.

Attack Preparation. To accomplish a successful attack we need to satisfy two preliminaries requirements: (1) the malicious app needs to recognize the presence (i.e., installation) of the target application on the device. (2) the malicious app needs to detect when the target application is opened and run on the device. These two states, installed and opened, allow the malicious app to identify a potential active connection between BluetoothChat applications on different devices.

In order to detect the presence of the target app, the malicious app can retrieve a list of installed apps by querying the `PackageManager` object. Such operation is not privileged and it can be executed by any app installed on the device. For the Bluetooth Chat sample, the malicious app can detect the installation of it just looking at the package name. Once the presence of the vulnerable app has been identified, the next step for the malicious app is to exploit a legitimate communication for spoofing content and deliver the attack payload. However, this may happen at unpredictable time since the malicious app does not know when a remote communication will be activated. While it is possible for the malicious app to continuously try to exploit the communication by using polling technique, this is not desirable from the attacker's perspective since it creates suspicious events that can be detected. The best result would be achieved if the malicious app could monitor the vulnerable app, and perform the attack only at the appropriate time. While it is very difficult to fully monitor the behavior of other apps from another app [30], a possible way to partially achieve the result is to monitor the list of open apps, obtainable via the `ActivityManager` class, specifically with the `getRunningAppProcesses` method. Again, this information can be requested to the Android system by any app without any specific privilege; the malicious app can continuously poll this list, and try the attack when the communication is open and running in foreground.

Payload Running. If the attacker has satisfied the previous two requirements the attack can be performed successfully. In particular for the BluetoothChat case, the attacker needs to install a malicious app on one of the two devices that performs the communication. The communication protocol over Bluetooth is implemented with BluetoothSocket, with a RFCOMM identifier for the chat service. If the attacker knows the identifier, his malicious app can send messages to the app on the other device, which will be indistinguishable from benign ones. In this case, the app is open source, so the RFCOMM identifier is embedded inside the application, and a simple manual investigation can reveal it. Once the attacker knows the identifier he can perform data injection on the remote device, and send a message to a remote application. The impact of this data injection is potentially high especially if the receiver trusts the sender and for instance she is opening any forwarded links, which in this case could lead to phishing pages controlled by the attacker. The following figure shows an example of hijacked communication in BluetoothChat. The first two messages are written by the user on the device Huawei P9 Lite, while the third is sent by our malicious app; the receiving user on device Nexus 6 will be unable to distinguish the malicious messages.



BluetoothChat
connected to HUAWEI P9 lite

This application allows two Android devices to carry out two-way text chat over Bluetooth. It demonstrates all the fundamental Bluetooth API capabilities, such as: (1) Scanning for

HUAWEI P9 lite: Hello

HUAWEI P9 lite: How are you?

HUAWEI P9 lite: http://phishing.link

BluetoothChat
connected to Nexus 6

This application allows two Android devices to carry out two-way text chat over Bluetooth. It demonstrates all the

Me: Hello

Me: How are you?

6.2 Data injection on Wi-Fi Direct +

In this second case study we focus on real-world app that use Wi-Fi Direct + [4] collected from the Google Play market. This file sharing app has more than 500,000 downloads, it is constantly being updated, and it has a paid version, Wi-Fi Direct + Pro. This information definitely indicates that the app is relevant for our analysis. Since this apps does not implement and app-to-app authentication as revealed by our tool we can perform a data injection attack.

App Functionalities. Wi-Fi Direct + offers the possibility to share file between two Android devices, via Wi-Fi-Direct protocol. After performing Wi-Fi-Direct pairing, on one device the user should select the option for receiving files. At this point, his device is entering in listening mode for incoming connections. When the user on the other device selects the option for sending a file. A success dialog is then displayed on the receiving device and the file is transferred.

Payload running. After pairing has been established, the app on the receiving device opens a `ServerSocket`, and accepts connections on it. If a malicious app on the sending device tries to connect to the socket, we have a typical CATCH scenario, in which the receiving app is not able to distinguish legitimate and malicious data. For the attack to succeed, some technical details have to be considered. The attacker needs to study the Wi-Fi Direct + protocol used to send files in order to replicate it without errors, then he has to build a malicious app for sending files with this protocol. At this point, by activating the sending of a file from the malicious app at the right time, as explained in the previous case study, the attacker is able to inject data in the communication with another device. With Wi-Fi Direct + in particular, the useful time window for data injection is reduced in comparison to the BluetoothChat case study; this is because Wi-Fi Direct + on the receiving device will accept only one file before closing the communication channel, as opposed to BluetoothChat, which keeps listening for incoming messages. This is a problem for the attacker: if the benign app sends its file first, then the file sent by the malicious app will not be accepted (race condition). If the opposite happens instead, the benign file will be rejected, and the sending user will be notified of an error. Depending on the situation, the users might verbally communicate and establish that something suspicious happened. This risk is always present, but it is greatly reduced in cases such as BluetoothChat, where no error messages are displayed to the users. Although we recognize this problem for the attacker in some cases, we have to also consider the situations in which the users are not able to identify the attack. For instance, the sending user receiving an error message may think of a bug in the app, especially if the receiving user confirms that the file has been correctly received (his app will display the correctly received malicious file). In our experiments we were able to perform the attack successfully. We were able to run the malicious app and send a malicious file without causing any alarm on the target device.

7 DISCUSSION

In this section we discuss about limitations of our analysis along with impact of the problem that we found out.

7.1 Impact of the problem

From our research, it is clear that high-level, app-to-app authentication is almost never present in Android apps that communicate on channels such as Bluetooth. Aside from the results of our algorithm on the large dataset, we could not manually find apps performing this type of authentication for the specific channels of our interest. We postulate that this is due to a lack of awareness in programmers, who build their code relying on sources such as the official Android documentation, and the network Stack Overflow for learning how to use a particular technology (e.g. Bluetooth). It is common to reuse sample code snippets from these sources with minimal adapting [19]; since they seem not to address the problem we are stating in any way, the issue is propagated, and any app using these technology is potentially vulnerable. It is worth noting that the actual impact of the vulnerabilities, as well as the difficulty of hypothetical attacks, greatly depend on the functionalities of the specific app being attacked, and need to be evaluated on a case-by-case basis. The evaluation of the general danger introduced by the lack of app-to-app authentication on a large scale is out of the scope of this research. We think that generally, vulnerabilities based on apps accepting unauthenticated content would be medium-impact (as in permitting phishing and/or DOS at best), but we cannot exclude the existence of particular apps where it would be possible to obtain more severe effects (e.g. arbitrary code execution).

Listing 3: Threads in Bluetooth communication

```
// Main thread code
new ReadThread().start()

...

// ReadThread code
public void run() {
...
  if (socket != null) {
  InputStream inputStream;
  try {
    inputStream = socket.getInputStream();
    inputStream.read(buffer);
    // We expect authentication happening here,
    // not in a separate thread
}
```

7.2 Limitations of our analysis

From the experiments, our system shows excellent performance in detecting the presence of CATCH vulnerabilities in Android apps. However, the results have to be considered together with the limitations of our technique. Our analysis suffers from the general limitations of static analysis. One of these limitation concerns the precision of the model of apps control flow. Argus-SAF is not able to handle particular intra-component and inter-component transitions, such as ones performed with reflection, and it cannot correctly model concurrency [29]. In practice, reflection is not commonly used by Android developers to perform normal tasks such as transitions between Activities. Instead concurrency is definitely present in peer to peer apps; to avoid blocking input/output, separate threads are typically spawned on demand to handle read/write operations on the channel (this applies to both Bluetooth and Wi-Fi Direct). In case of authentication, we expect to see controls on data read from the channel immediately after a read operation, following our authentication model. So, while it is true that it would be a problem to correctly model authentication flows that involved concurrent operations, there is no reason to expect authentication occurs in a separate thread in reality (see Listing 3). To further validate our results (Section 5.2) we manually analyzed 20 Android apps from 662 dataset apps and we check whether threads functions defined in the apps include any authentication scheme (false negative results). Our manual analysis shows that no one of the app functions threads analyzed contained any authentication scheme.

8 RELATED WORK

To the best of our knowledge, we are the first to explore the potential dangers associated with the lack of app-to-app authentication in Android apps. However, previous research has made important contributions in related areas.

Security of Android communication channels. Previous work highlighted the problems existing in Android device-to-device communications [11, 20]. In particular, Demetriou et al. [11] studied several channels of communication in Android (such as Bluetooth, SMS, Internet and audio) showing that the security model of Android does not offer adequate measures for protecting certain secrets. To address this problem, they build a security system, called SEACAT, to enforce fine-grained protection on the above resources. Our work continues the exploration of missing security features in Android, and how apps can be vulnerable if developers make the wrong assumptions about the security of the underlying system.

Security models of peer-to-peer protocols. Claycomb and Shin formally studied the problem of authentication in mobile devices [8], and use BAN logic to prove that device authentication using a single communication channel is not possible. We consider this result when building our model: in particular, this justifies our assumption of the secret exchange happening out-of-band. Shen et al. focus on Wi-Fi Direct technology, studying its security and discussing related best practices [26]. Again, importance of out-of-band channels to obtain authentication is highlighted, and used in the implementation of a secure Wi-Fi Direct protocol.

Using static analysis for detecting authentication. Static analysis techniques have been extensively used in previous work, for instance for detecting malicious application logic on Android or Web application[9, 15, 22], and for detecting privacy leaks in both iOS [14] and Android [16] apps. Closely related to ours is the work of Shao et al., which studies the presence of authentication in the use of Unix domain sockets on Android [25]. We followed the same choice of tools used to perform the analyses, favoring Argus-SAF [29] (formerly *Amandroid*) over FlowDroid [6] because of its superior handling of inter-component communication. An important difference is that we could not model our problem with as a standard taint-analysis reachability search, so we had to build our custom data dependency analysis on top of the tools provided by Argus-SAF.

9 CONCLUSION

In this paper, we have shown the extension and potential impact of *CATCH* vulnerabilities in Android apps, providing a threat model and specific definitions for the problem, as well as manual and automated analysis for experimental evaluation. Our main contribution, the automated system for APK analysis, is a first line of defense against human error, and could be used to identify vulnerable apps. Nevertheless, it is clear that the problem can be effectively solved only by raising awareness among Android developers, by providing them with appropriate documentation explaining the dangers of *CATCH*, as well as APIs and libraries to correctly perform high-level app-to-app authentication on peer-to-peer communication channels.

ACKNOWLEDGMENTS

This project has received funding by the Italian Ministry of Foreign Affairs and International Cooperation (grant number: PGR00814). The project has also received funding by the US Army Research Office (grant number: W911NF-17-1-0039).

REFERENCES

[1] [n.d.]. Android BluetoothChat Sample. https://github.com/googlesamples/android-BluetoothChat. Accessed: 2019-01-18.

[2] [n.d.]. ProGuard. https://www.guardsquare.com/en/products/proguard. Accessed: 2018-11-19.

[3] [n.d.]. Shrink your code and resources. https://developer.android.com/studio/build/shrink-code. Accessed: 2018-11-19.

[4] [n.d.]. WiFi Direct +. https://play.google.com/store/apps/details?id=com.netcompss_gh.wifidirect. Accessed: 2019-02-15.

[5] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[7] W. R. Claycomb and D. Shin. 2009. Secure device pairing using audio. In *43rd Annual 2009 International Carnahan Conference on Security Technology*. 77–84. https://doi.org/10.1109/CCST.2009.5335562

[8] William R. Claycomb and Dongwan Shin. 2011. Extending Formal Analysis of Mobile Device Authentication. *J. Internet Serv. Inf. Secur.* 1 (2011), 86–102.

[9] Andrea Continella, Michele Carminati, Mario Polino, Andrea Lanzi, Stefano Zanero, and Federico Maggi. 2017. Prometheus: Analyzing WebInject-based information stealers. *Journal of Computer Security* 25 (02 2017), 1–21. https://doi.org/10.3233/JCS-15773

[10] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P'19)*.

[11] Soteris Demetriou, Xiao-yong Zhou, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A Gunter. 2015. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources.. In *NDSS*.

[12] Mianxiong DONG, Takashi Kimata, Komei Sugiura, and Koji ZETTSU. 2014. Quality-of-Experience (QoE) in Emerging Mobile Social Networks. *IEICE Transactions on Information and Systems* E97.D (10 2014), 2606–2612. https://doi.org/10.1587/transinf.2013THP0011

[13] K. Doppler, M. Rinne, C. Wijting, C. B. Ribeiro, and K. Hugl. 2009. Device-to-device communication as an underlay to LTE-advanced networks. *IEEE Communications Magazine* 47, 12 (Dec 2009), 42–49. https://doi.org/10.1109/MCOM.2009.5350367

[14] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In *NDSS*. 177–183.

[15] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 377–396.

[16] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer, 291–307.

[17] Y. Li, S. Su, and S. Chen. 2015. Social-Aware Resource Allocation for Device-to-Device Communications Underlaying Cellular Networks. *IEEE Wireless Communications Letters* 4, 3 (June 2015), 293–296. https://doi.org/10.1109/LWC.2015.2410768

[18] Jiajia Liu, Yuichi Kawamoto, Hiroki Nishiyama, Nei Kato, and Naoto Kadowaki. 2014. Device-to-device communications achieve efficient load balancing in LTE-Advanced networks. *Wireless Communications, IEEE* 21 (04 2014). https://doi.org/10.1109/MWC.2014.6812292

[19] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *2015 IEEE Symposium on Security and Privacy*. 692–708. https://doi.org/10.1109/SP.2015.48

[20] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. 2014. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android.. In *NDSS*.

[21] Phond Phunchongharn, Ekram Hossain, and Dong In Kim. 2013. Resource allocation for device-to-device communications underlaying LTE-advanced networks. *IEEE Wireless Communications* 20, 4 (2013), 91–100.

[22] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1120–1136. https://doi.org/10.1145/3243734.3243785

[23] M. K. Reiter, J. M. McCune, and A. Perrig. 2005. Seeing-Is-Believing: Using Camera Phones for Human-Verifiable Authentication. In *2005 IEEE Symposium on Security and Privacy (S&P'05)(SP)*, Vol. 00. 110–124. https://doi.org/10.1109/SP.2005.19

[24] N. Saxena, J. . Ekberg, K. Kostiainen, and N. Asokan. 2006. Secure device pairing based on a visual channel. In *2006 IEEE Symposium on Security and Privacy (S P'06)*. 6 pp.–313. https://doi.org/10.1109/SP.2006.35

[25] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z. Morley Mao. 2016. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 80–91. https://doi.org/10.1145/2976749.2978297

[26] W. Shen, B. Yin, X. Cao, L. X. Cai, and Y. Cheng. 2016. Secure device-to-device communications over WiFi direct. *IEEE Network* 30, 5 (September 2016), 4–9. https://doi.org/10.1109/MNET.2016.7579020

[27] Dongwan Shin et al. 2006. Using a two dimensional colorized barcode solution for authentication in pervasive computing. In *Pervasive Services, 2006 ACS/IEEE International Conference on*. IEEE, 173–180.

[28] Dirk Balfanz Smetters, Dirk Balfanz, D. K. Smetters, Paul Stewart, and H. Chi Wong. 2002. Talking To Strangers: Authentication in Ad-Hoc Wireless Networks.

[29] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Trans. Priv. Secur.* 21, 3, Article 14 (April 2018), 32 pages. https://doi.org/10.1145/3183575

[30] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao yong Zhou, and XiaoFeng Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android.. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 915–930. http://dblp.uni-trier.de/db/conf/sp/sp2015.html#ZhangY0ZW15