



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

---

Stefano Cristalli, Danilo Bruschi, Long Lu, Andrea Lanzi  
December 13, 2019

University of Milan Italy Northeastern University Boston US

1. Introduction
2. Cross Device Authentication Scheme
3. Approach Overview
4. Experimental Evaluation
5. Case Studies
6. Discussion
7. Conclusion & Future works

# Introduction

---

- Cross-device communications allow nearby devices to directly communicate bypassing cellular base stations (BSs) or access points (APs) (e.g. **spectral efficiency improvement, energy saving, and delay reduction**, etc.)
- Without the need for infrastructure, **such a technology enables mobile users (e.g., Android) to instantly share information (e.g., pictures and videos)**
- Such technology is also predominant in **IoT environment** where a mobile device is directly connected to the embedded system.

# Current Solutions

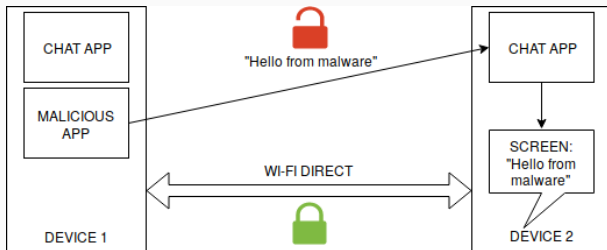
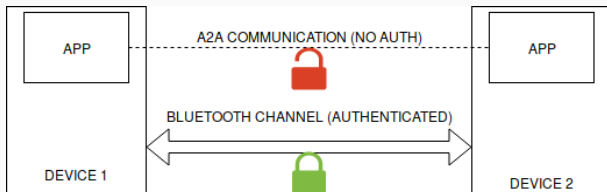
- Several solutions exist for securing cross-device communication. In the Android environment, they allow **authentication of devices and communication channels**.
- Others solutions **restricts apps access to external resources, such as Bluetooth, SMS and NFC**, by defining new SEAndroid types to represent the resources.
- Moreover such **solutions are not able to address several communication channels such as: SMS, Audio, Wi-Fi and NFC** due to of missing important information for the detection purpose.

- 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 version.
- We provide a solution to the CATCH problem by **designing and developing an authentication scheme detector** that analyzes Android apps to discover potential vulnerabilities
- **Validate the results of our system on Android apps** with manual analysis, and test its resilience in detecting the authentication scheme.

# Cross Device Authentication Scheme

---

# Cross-device Authentication Scheme





# Threat Model & Attack

- The attacker is able to install a malicious app on the mobile's victim phone.
- 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.
- Depending on the particular context, there are some scenarios in which the attack can become very dangerous: **Phishing, Malware delivery, Exploitation.**

# Approach Overview

---

# Challenges

- We need to define a **generic scheme that captures the essential logic of app-to-app authentication**.
- 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**.
- 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 (e.g., obfuscation).

# Authentication Scheme Definitions

- 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.

# Detection Strategy

We define two main check points for our algorithm:

- An **entry point** is an instruction in the code that indicates the start of the communication over the analyzed channel (e.g., data receiving)
- An **exit point** is represented by the first authenticated use of the data coming from the monitored channel.

# Detection Algorithm

## 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
```

# Reducing False Positive

Costant Propagation Technique....

# Experimental Evaluation

---



To test our system we divided the experiments into two main categories:

- A dataset analysis on APKs retrieved from a research repository, aiming at confirming the efficacy of the algorithm on negative samples;
- 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.

# Dataset Composition

- We ran tests on a large number of APKs collected from the **Androzoo repository**, crawled from **several Android markets: Google Play, Anzhi and AppChina**.
- 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 **Bluetooth filter** and we obtained a total of **2,739 APKs**.
- Obfuscation 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**.
- we discovered that 704 of the selected apps do not have any entry point for Bluetooth communication in the CFG. we obtained a number of **238 APKs, suitable for our analysis and evaluation**.

# 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 200 APKs (containing permissions/classes for Bluetooth) from our dataset, and performed a manual analysis by searching them on Google Play.
  - **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/health apps**, using Bluetooth to send data from smartphone to computer, or again smartphone to device

- We run our algorithm on **238 Apps without constant propagation** we obtained 11% of false positive. With constant propagation enable we reach **0% of false positive**.
- We built a custom app using Bluetooth that use out-of-band authentication and we applied Proguard transformation on the app. **We found that our algorithm correctly predicts the possible presence of authentication**
- We decided to run our tool on ProGuard-obfuscated APKs from our dataset a total of 662 APKs. **100% of the APKs were identified as negative (i.e., not containing authentication) by our tool.**

# Performance Analysis

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. we set up three Threshold 30, 60, 120 sec.

- For any entry point, 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.**
- The **variation of the results between the three runs is minimal**, that it means that we generally do not miss any important information.
- 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**.

# Case Studies

---

# Data injection on BluetoothChat

text.....

text.....



## Discussion

---

text.....

## Conclusion & Future works

---

# Conclusion

text.....

**Thank you for attention**

---

**Questions?**

---