Virtualizing the Process of Fuzzing for the Testing of QR Codes on Android Applications

Alberto Lazari
Department of Mathematics
University of Padua
Padua, Italy
alberto.lazari@studenti.unipd.it

Elia Scandaletti
Department of Mathematics
University of Padua
Padua, Italy
elia.scandaletti@studenti.unipd.it

Francesco Protopapa
Department of Mathematics
University of Padua
Padua, Italy
francesco.protopapa@studenti.unipd.it

***Abstract*— Most fuzzing methodologies are based on text inputs, that are crafted or randomly created to find bugs and input vulnerabilities in programs. While fuzzing Android applications is not an entirely new field, doing it specifically by using QR code inputs is still a less explored technique. *QRFuzz* by F. Carboni, D. Donadel, and M. Sciacco [1] is one of the first implementations of this kind of testing, however it is limited by its use of a physical device and actual QR codes to scan, that cause difficulties in testing and pose a limit to the automation and scalability of the process.**

In this report, we provide an alternative design to the framework presented in F. Carboni, M. Conti, D. Donadel, and M. Sciacco [2] (*If You're Scanning This, It's Too Late! A QR Code-Based Fuzzing Methodology to Identify Input Vulnerabilities in Mobile Apps*), proving that it is possible to remove any physical device and QR code constraint by using a virtualized environment for the testing process, that can be easily configured in a reproducible and automated way.

We are able to do it by running an Android Virtual Device (AVD), instead of using a real one, that is linked to a static video stream as a virtual webcam. A comprehensive setup system is used to automatically install and configure the toolkit.

***Index terms*—QR code, Fuzzing, Android security, Virtualization, Mobile applications**

I. Introduction

Related works

In recent years, the widespread adoption of Quick Response (QR) codes has introduced new challenges for the security and robustness of mobile applications. QR codes, while they can be used to provide simple and immediate access to information, they also present potential security vulnerabilities that can be exploited by malicious actors, being them just another form of input for an application. The paper "If You're Scanning This, It's Too Late! A QR Code-Based Fuzzing Methodology to Identify Input Vulnerabilities in Mobile Apps." by F. Carboni, M. Conti, D. Donadel, and M. Sciacco [2] made a first effort towards an automated fuzzing-based methodology able to address this problem.

Their proposal required the use of a real smartphone to run the tests on, which limited the possibility of automation and the reproducibility of the results, being them based on various factors that depends on the device, that also causes some hardware-related probems.

Contributions

Our work builds upon the F. Carboni, M. Conti, D. Donadel, and M. Sciacco [2] research, which introduced a fuzzing framework designed to uncover QR code input vulnerabilities in mobile apps. However, to improve the efficiency of the proposed approach with the aim of making the testing more scalable, we propose an alternative design. Our proposal is based on the virtualization of the mobile phone, that allows for a more detailed configuration of the environment and potentially for the creation of multiple parallel testing sessions, without the need of large quantities of dedicated hardware. Our framework also uses a virtual camera, linked to the virtual device, such that it is possible to virtually display QR codes on it, cutting out the hardware-related problems the previous proposal suffers of.

By virtualizing the testing infrastructure, we aim to overcome the limitations of traditional physical devices, providing researchers and developers with a more versatile platform for uncovering potential vulnerabilities in Android applications that provide QR code interactions. The introduced virtualization not only improves the efficiency of the fuzzing process but also simplifies the adoption of this fuzzing methodology for new users.

Report's structure

In the following sections of this report, we present the details of our virtualization approach. We provide a comprehensive overview of the previous work (Section II), critically analyze the key aspects of its original design (Section III), and illustrate the details of our approach (Section IV). Sub-

sequent sections discuss the technological and implementation details of webcam (Section V) and device virtualization (Section VI), how the entire process is automated (Section VII), results of the performed tests (Section VIII) and conclude with insights into future possible actions that could further improve this field (Section X).

II. Overview of previous work

Thestarting point of our project is *QRFuzz* by F. Carboni, D. Donadel, and M. Sciacco [1], which is made of the following components:

- **PC Monitor**: A phisical monitor in which the generated QRs are displayed.
- **Smartphone**: A phisical smartphone in which the app under test is run.
- **Appium server**: An Appium Server is an open-source automation server that facilitates the automated testing of mobile applications across different platforms, including iOS, Android, and Windows. It acts as a bridge between test code and the mobile device or emulator, allowing to interact with the application under test programmatically.
- **QR Code Fuzzer**: The main element of the system that is responsible for coordinating and initiating actions for both the QR Code Generator and the Appium Server, along with managing the entire testing process. The fuzzing operation itself involves a series of steps repeated for each QR code under examination. These steps include navigating to the designated scanning page within the application, scanning the QR code, verifying if the app's normal behavior has been disrupted, and logging the results of the iteration as potential indications of a vulnerability.
- **QR Code Generator**: The QR Code Generator is another crucial component of QRFuzz. Its primary function is to dynamically produce QR codes using a predefined dictionary and display them on the screen. This generator is capable of creating both standard, general-purpose QR codes and custom codes based on an application-specific template.

Workflow

The workflow of the system (Figure 1) can be summarized by the following steps:

1. The QR code displayed on the monitor is scanned by the smartphone.
2. The smartphone sends a scan responce to the QR Code Fuzzer through the Appium server
3. The QR Code Fuzzer notifies a QR change request to the QR Code Generator by updating a shared JSON file.
4. The QR Code Generator generates a new QR and displays it on the monitor.
5. The QR Code Generator notifies the update to the QR Code Fuzzer by changing the JSON file.
6. The QR Code Fuzzer sends a scan request to the smartphone via Appium server.

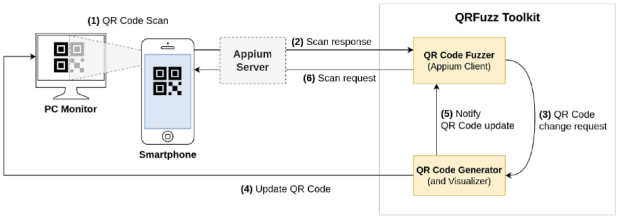

Figure 1: Old architecture

III. Critical Aspects

The architecture presented in the previous section is not scalable since running multiple tests in parallel would require to use multiple smartphones and monitors at the same time. Moreover setting up all the equipment is not trivial and the results of the tests may differ because different devices has been used. These limitations slow the process of finding new vulnerabilities in the apps under test.

Conducting extended tests on applications using a physical smartphone revealed a sporadic issue of camera overheating, which significantly impeded the continuity of the entire testing process. This overheating problem, in addition to slowing down the testing process, also made it necessary to make a distinciton between crashes caused by QRs and crashes caused by camera overheating.

Due to differnt programming languages used in the development of the QR Code Fuzzer and QR Generator, coordinating them is not a straightforward task. To overcome this challenge, a JSON file is used as a means of communication and synchronization between the QR Code Fuzzer and the QR Code Generator. However, relying solely on a JSON file for communication introduces potential drawbacks such as increased latency and complexity, as well as the risk of data inconsistency or loss due to manual handling.

IV. Our Approach

We managed to solve the critical aspects described in the previous section by implementing several key solutions.

Virtualizing the device

Instead of relying on physical smartphones, we employ virtualization techniques. This eliminates the need for physical devices and monitors, thus improving scalability.

Virtualizing the camera

By virtualizing the camera, we create simulated camera interfaces within the testing environment. This allows us to present QR codes for scanning without the limitations imposed by physical camera hardware.

Rewriting the QR Code Generator

We rewrote the QR Code Generator component in Javascript in order to avoid comunication through the file system with the QR Code Fuzzer which was already written in Javascript.

Automation

Automation plays a crucial role in the testing process. We implemented several scripts for initiating the whole environment, virtualizing the camera and running the tests in a completely automated way.

Workflow improvement

The workflow of the whole system (Figure 2) has been improved, we can identify five key phases in it:

1. The QR Code Fuzzer component takes a string as input from a pool of inputs. Using this input string, the QR Code Fuzzer generates a QR code corresponding to the string. Once generated, the QR code is saved in the file system as an image file.
2. The QR code image file saved in the file system is displayed in a virtual camera.
3. QR Code Fuzzer sends a scan request to the Android emulator through the Appium server.
4. The Android emulator receives the scan request from the Appium server, it then accesses the virtual camera where the QR code is displayed and scans the QR code.
5. Once the QR code is successfully scanned by the Android emulator, it generates a scan response which contains log informations and a screenshot. Also in this phase, the communication between the Android emulator and the QR Code Fuzzer occurs through the Appium server.

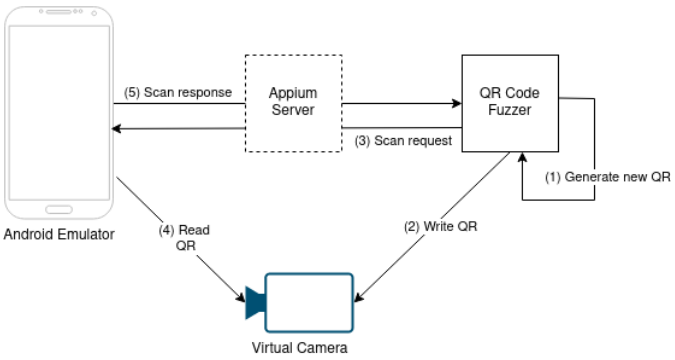

Figure 2: New architecture

V. Webcam virtualization

In order to decide how to virtualize the camera, we considered several tools:

v4l2loopback

v4l2loopback [3] is a lightweight kernel module that provides a simple way to create virtual video devices. It integrates directly with the Linux Video4Linux2 (v4l2) subsystem, moreover it is open source, well known for its reliability and has already been used in several applications requiring virtual camera functionality.

Using v4l2loopback presents some drawbacks, in fact configuration and setup might require some familiarity with Linux kernel modules and system administration, making it less user-friendly for novice users. There may be occasional compatibility issues with specific software or older kernel versions. Finally it is only available on Linux and it does not provide advanced features such as scene composition and effects.

OBS

OBS (Open Broadcaster Software) [4] is a free and open-source app for screencasting and live streaming. OBS offers a wide range of features including scene composition, overlays, transitions, and effects, making it suitable for creating complex virtual camera setups. It is available for Windows, macOS, and Linux and with an intuitive interface and extensive documentation, it is relatively easy to use for new users.

On the other hand OBS is difficult to be used entirely from the command line and even if it is user-friendly, the extensive feature set of the tool can be overwhelming for users looking for simple virtual camera functionality.

GStreamer

GStreamer [5] is a pipeline-based multimedia framework that links together a wide variety of media processing systems to complete complex workflows. GStreamer offers extensive flexibility and customization options and it is available for Windows, macOS, and Linux. Despite that GStream-

er's extensive feature set and complex pipeline syntax may present a steep learning curve for users unfamiliar with multimedia frameworks and while GStreamer has extensive documentation, finding specific information may be challenging due to the framework's complexity.

Selected tool

After carefully considering the advantages and drawbacks outlined above, we decided to use v4l2loopback for our project. Our decision was based from the project's main objective of improving the scalability of QR fuzzing. Relying on a tool that does not provide a good support for the usage from command line like OBS was not an option because it would have significantly limited the whole automation process. We also decided not to go for GStreamer because the complex set of options offered by the tool was not necessary for our purposes.

In the end v4l2loopback was the most appropriate solution for our project. Although configuring the kernel module required an initial investment of time, the process was manageable and well-documented. Moreover, v4l2loopback's straightforward interface aligned with our objectives, allowing us to focus on optimizing the automation process without the distractions of extraneous features. Furthermore, the absence of advanced video editing features within v4l2loopback was not a limit for us. Our only requirement of resizing QR codes could be achieved without the need for additional effects or modifications.

VI. Device virtualization

In order to achieve a good level of scalability and efficient testing in a controlled environment, we opted for device virtualization, leveraging the capabilities of the Android emulator from the Software Development Kit (SDK).

The emulator easily integrates with a virtual camera, a feature that can be implemented through a simple command line switch. This allowed us to simulate scanning scenarios without the need for physical devices. Moreover, emulators do not require a graphical user interface (GUI), enabling tests to be performed on a server environment.

Each testing setup benefited from a clean Android environment, ensuring consistent and reliable results. This reproducibility is crucial for accurate testing and analysis

Virtualization also provides complete control over the Android operating system version and build. This flexibility allows to test across different Android versions and configurations.

Finally device virtualization eliminated the need for physical hardware, reducing logistical complexities and costs. Moreover, by no longer depending on the hardware potential differences in results attributable to varying hardware or vendor-specific configurations are eliminated.

VII. Automation

Automation plays a crucial role in our framework, since many different technologies need to be installed and, once installed, they have to be able to communicate with each other. The original toolkit exclusively offered basic scripts for installing dependencies and initiating application testing, but its functionality was limited to Debian-based systems only. These scripts, however, were constrained by numerous assumptions about the user's system. Notably, the assumptions included the user's platform, the manual initiation of programs, and the user's familiarity with the tool's intricacies—specifically, the need to adjust many non-trivial parameters.

Our automation system is designed with the main goal of leading users toward a fully operational fuzzing environment and initiating the process seamlessly. This is achieved by providing an intuitive setup that accomodates to users regardless of the operating system they are running.

Managing dependencies

To ensure a seamless and user-friendly experience, we implemented a robust dependency management system. Central to this system is the creation of a comprehensive `install` script, designed to orchestrate the installation of various foundamental dependencies for the fuzzing process. This script loads individual setup scripts when needed, each tailored to install a specific program essential to the fuzzing environment.

Our approach goes beyond the traditional one-size-fits-all installation process. Each component within the fuzzing framework is inherently aware of its dependencies and their installation status. In the event of missing dependencies, rather than resulting in a system failure, the components prompt the user to install the necessary prerequisites.

This dependency check mechanism is implemented in a modular and reusable fashion. By adhering to a modular design, our system ensures adaptability to evolving dependencies, making it straightforward to incorporate new programs or updates. The reusability of the dependency check modules enhances maintainability, enabling developers to apply the same mechanism across different components within the fuzzing framework.

In summary, our dependency management system, lead by the `install` script and supported by modular and reusable dependency checks, provides a streamlined and adaptive approach to handling the various software requirements of the fuzzing process. This user-centric design minimizes potential roadblocks, allowing researchers to focus on

the core aspects of their work rather than grappling with intricate installation procedures.

Managing components: simplifying complexity with wrapper scripts

In the intricate landscape of our fuzzing framework, we developed a set of wrapper scripts to simplify user interaction with underlying tools. These scripts not only offer a simplified interface but also handle the intricacies of launching and managing multiple components, critical to the fuzzing process. In this section, we illustrate and showcase these scripts, providing insights into their purpose and internal mechanisms:

`stream`: orchestrating video stream creation

The `stream` script serves as an orchestrator for v4l2loopback and ffmpeg, ensuring the seamless creation of a video stream from a QR-encoded PNG to a virtual video device (`/dev/videoN`). This process involves intricacies in managing these tools, which are inherently complex for manual usage. By providing a simplified interface, `stream` enhances accessibility while managing the correct interaction between v4l2loopback and ffmpeg.

`launch-emulator`: streamlining emulator initialization

For virtual camera integration, the `launch-emulator` script acts as a streamlined interface to the Android emulator. It verifies the availability and status of the specified virtual camera device (`/dev/videoN`) and performs necessary checks to ensure its readiness. This script simplifies emulator launch procedures, making the process user-friendly while ensuring the proper configuration of virtual camera interfaces.

`apk-install`: automating application deployment

This script simplifies the tedious process of application installation, the `apk-install` script automates the download and installation of application packages (APKs). This script ensures that the target applications are readily available for testing, minimizing manual intervention and simplifying the preparation phase of the fuzzing process.

`qrfuzz`: main orchestrator of the system

Core of the fuzzing framework, the `qrfuzz` script is the main orchestrator. This script not only calls upon the previously mentioned `stream` and `launch-emulator` scripts, but also manages their lifecycle as background devices. Furthermore, `qrfuzz` initiates the Appium server, a crucial element for interacting with applications within the emulator. It plays a crucial role by starting the entire system and smoothly transitioning into the fuzzing process.

Overall, these wrapper scripts collectively contribute to a more user-friendly and automated fuzzing environment. They encapsulate the complexities of underlying tools, manage component lifecycles, and simplify interactions, empowering researchers and developers to focus on the core aspects of their work without being distracted by complicated technical details.

Unified test automation management

Previously, executing the tests required separate actions:

- Launching the Appium server in one terminal.
- Initializing the QR Code generator in another terminal.
- Starting the QR Code fuzzer in yet another terminal.

With our automation system, these three components are all managed by the same script and their logs are displayed on the same terminal. This automation process simplifies the test execution, making it more convenient and efficient for users.

VIII. Test results

Since running tests on different applications require to write a distinct inspector for each application, our tests have been run on the “postepay” app. Then we showed that it is possible to automate the testing of applications which require a login by writing an inspector for the “discord” app.

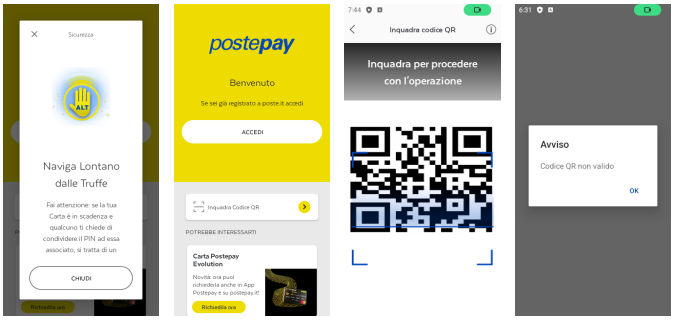

Figure 3: Test screenshots

We run the complete test set of F. Carboni, D. Donadel, and M. Sciacco [1] on the “postepay” app without finding vulnerabilities and our tool was able to scan an average of 7 QRs every minute.

IX. Conclusions

In our work, we showed how the process of fuzz testing mobile application using QR codes described in F. Carboni, M. Conti, D. Donadel, and M. Sciacco [2] can be improved. Our contribution is about the complete virtualization of the testing process, making it significantly more scalable and efficient. Previously, fuzz testing mobile applications with QR codes has been performed by relying on physical devices, which pose several challenges such as difficult setup process, camera overheating issues, and scalability issues.

By virtualizing the testing environment, we effectively eliminated these critical aspects.

Overall, our approach to virtualizing the fuzz testing process not only addressed the critical issues associated with physical devices but also significantly enhanced the scalability, efficiency, and effectiveness of QR fuzz testing enabling researchers and developers to conduct tests at scale.

X. Future work

Future work should move along two main axes: the improvement of the current project and the parallelization of fuzzing.

Improving the project

The actual project can be improved in several different ways. Firstly, expanding the project with additional inspectors would significantly broaden its applicability, enabling it to fuzz a more extensive range of applications. Furthermore, introducing features such as the possibility to utilize a specific subset of dictionaries and adjust logging levels through command flags would enhance the flexibility and customization options for users. Additionally, future development efforts could focus on streamlining the process of downloading the required APKs, therefore streamlining the setup process and enhancing the overall automation.

Parallelizing the fuzzing

Leveraging the abstraction and automation capabilities already integrated into the system, researchers can now readily replicate experiments within containerized environments. This opens the way for creating Docker images of the system, facilitating seamless testing within controlled, deterministic environments. This approach allows to conduct parallel tests through orchestrating services and platforms like Ansible, enabling more efficient and scalable fuzz testing.

References

[1] F. Carboni, D. Donadel, and M. Sciacco, “QRFuzz”. [Online]. Available: https://github.com/spritz-group/QRFuzz

[2] F. Carboni, M. Conti, D. Donadel, and M. Sciacco, “If You’re Scanning This, It’s Too Late! A QR Code-Based Fuzzing Methodology to Identify Input Vulnerabilities in Mobile Apps”, in *International Conference on Applied Cryptography and Network Security*, 2023, pp. 553–570.

[3] “v4l2loopback”. [Online]. Available: https://github.com/umlaeute/v4l2loopback

[4] “OBS”. [Online]. Available: https://github.com/obsproject

[5] “GStreamer”. [Online]. Available: https://github.com/GStreamer/gstreamer