FuzzQR: QR Code Fuzzer Toolkit for Green Pass Checkers and Smartphone Apps

Federico Carboni, Mariano Sciacco
Department of Mathematics
University of Padova
Padova, Italy
{federico.carboni, mariano.sciacco}@studenti.unipd.it

Abstract—**In recent years, QR codes become popular for different applications such as Green Pass checking, used for COVID-19 swab and vaccination verification. The personal QR code is scanned with a smartphone application that certifies whether the Green Pass is valid or not. Though, QR codes can be manipulated by adding malicious code inside the payload. Hence, we developed a toolkit, FuzzQR, for testing multiple QR codes in an automated way on a particular set of fuzzing strings that may crash the target app. In our experiments, we tested *VerificaC19* on Android with 5 dictionaries of words containing symbols and ASCII characters. Our tests on 344 words showed that our toolkit correctly scanned 98.6% of the given QR Codes, with an average scan time of 3-4 seconds. Moreover, FuzzQR can be adapted for applications on both Android or iOS and custom dictionaries of fuzzing strings.**

Index Terms—**QR Code, Fuzzing, Green Pass, COVID-19, Android, toolkit**

I. Introduction

In the last decades, one-dimensional barcodes have been one of the most used solution for saving small pieces of information quickly, while being easy to scan, thus giving access to a huge number of applications. The main use of one-dimensional barcodes is found in shop articles, where there is the need of storing small information such as the identification number, thus helping to match an item with a record in a database. Nevertheless, the amount of information that one can save might not be enough, especially if someone wants to store lots of data without having a strong dependency with a centralized database. Here comes two-dimensional barcodes, also known as QR codes (*quick response codes*), that are a simple solution used to fit data in small 2D-representation of symbols and shapes. This solution guarantees a reliable way to store data and a quick approach for the end user to read this kind of information with just a smartphone or a dedicated scanner. There are plenty of apps able to read QR codes, such as bike sharing apps, social media apps and shopping apps. Even though simplicity is the winning point of this technology, QR codes are more complex and can contain more data compared to one-dimensional barcodes. For this reason, everything that is behind some shapes and symbols might be dangerous, depending on the application used. In fact, QR codes can be an attack vector as far as the payload is malicious for the mobile app that scanned specific QR codes.

In this period of time, the whole world is fighting against a pandemic disease, due to SARS-COV-2 virus, that required swab and vaccination control in many public places to prevent the spread of the virus for the safety of people. Hence, a good solution is found in the green pass, which is an official document entirely composed of a QR Code that is generated after a vaccine dose or a swab with a variable expire time depending on the type and the law according to the region where it is used.

The green pass can be validated by scanning it through a smartphone application, but what would happen if the payload is invalidated and contains malicious code? We looked at some literature regarding malicious code in QR codes like [11], [1] and [12]. Based on this premise, we decided to develop a **fuzz testing toolkit for Android/iOS apps**, called **FuzzQR**, that permits to scan multiple QR codes with malicious data and see whether it is possible to make an app crash or alter the normal behavior.

In this way, it is possible to test the green pass checker security using an automated testbed, in order to detect possible exploits that can be a risk for health frauds. Moreover, it is possible to change the type of app under analysis by adjusting the script in a simple way. Therefore, our contribution in this field is reported as follows:

- Design and implementation of a fuzz testing toolkit compatible with Android and iOS apps that can be used to create a testbed working autonomously to scan QR code and report results.
- Security analysis of the QR code payload used for the Green Pass document, with focus on possible malicious code.
- Code analysis and testbed results of *VerificaC19*, a green pass checker app officially used in Italy, picked as main example for our tests.

II. Background

A. QR Code Technology

As mentioned, the QR Code is a type of bi-dimensional barcode that can be easily read by mobile and electronic devices. They store information as a series of pixels in a square-shaped grid. Since many smartphones have already built-in QR reader apps, they are often used in advertising and marketing campaigns. More recently, QR codes have been used to identify and validate Green Passes by checking whether a person has a negative swab or a valid vaccination.

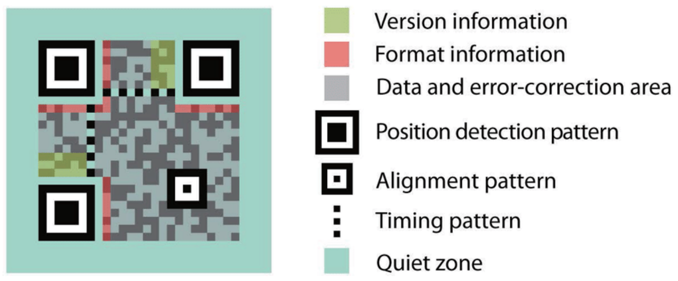


Fig. 1. QR Code structure (Version 2) - *Source:* [8]

There are many different types of QR codes, some of the most used are:

- **Standard**: As shown in Figure 1, it has three positioning markings and a variable number of alignment markings to adjust the orientation. The area surrounding the markings is reserved for the format and version info. There is an L-shaped sequence of 1 and 0 in the upper-left part of the QR, that is used from the reader to identify the size of each individual pixel. It requires also a white "quiet zone" all around the square. This type of QR code can store up to 4296 alphanumeric characters (in a square of 177 x 177 modules);
- **Micro QR code**: requires only one position detection square, but it can store only 35 characters;
- **iQR code**: the code is rectangular, it can store up to 40,000 numeric characters, the error correction can be up to 50%;
- **Atzec code**: the data is encoded in concentric square rings, with a bulls-eye detection pattern at its center. It does not need a quiet zone, and can contain 3067 characters.

Due to their popularity, QR codes can also be used for malicious purposes. Moreover, they are intended to be machine-readable, so humans cannot distinguish between benign and malicious codes, since the payload is hidden to the user. For example, as reported in [11], an attacker can insert a custom URL inside a QR code and trick the user with a cloned website, using a *phishing* approach. In the same way, the attacker can create a QR code for a payment, redirecting to a different bank account, thus committing *fraud*. In these two cases, the app scanning the QR code should work without problems. On the other hand, there might be attacks aimed at breaking the app's normal behavior. For example, an attacker can create a QR code that performs *SQL injection* [10], *command injection* or *buffer overflow* once it is read by the app. These last cases can be prevented if the developer checks in advance the input through a process of sanification. Our work is particularly focused on finding the cases where there is no proper input sanification: the goal is to generate malicious QR codes that cause unexpected behaviors into the mobile application under attack (e.g. app crashes).

B. VerificaC19 app

The mobile application *Verifica-C19* is the official choice suggested by the Italian Government for the verification of the Green Pass. The app is a fork of the European one, *EU Digital COVID Certificate Verifier*, and it makes use of two additional repositories. The final structure of the project is the following:

- **it-dgc-verificaC19-android** [4]: the starting point of the app, essentially the application user interface;
- **dgca-app-core-android** [2]: the EU core, that contains all the decoding and the decompressing functions needed by the app;
- **it-dgc-verificac19-sdk-android** [5]: the SDK, that contains the logic of the application, the procedures to perform the network operations and the data storage classes.

Moreover, the QR code detection and decoding phase includes also some additional third-parties libraries, that could potentially contain themselves some string-related vulnerabilities:

- **zxing**: for the QR code scanning;
- **retrofit2**: for the network calls;
- **gson**: for the JSON serialization / deserialization.

The first step performed by the app, when activated, is to use the *xzing* library to detect the QR code from the camera view and convert it into a string (as described in the *CodeReaderFragment*). After that, the string code is used to initialize a *VerificationViewModel* object. Finally, here the string is decoded to verify the authenticity of the COVID certificate. More in detail, the *decode* function performs the following operations:

1) checks and remove the prefix *"HC1:"*;
2) decodes the string from the *Base45* format;
3) decompresses using *Zlib*;
4) decodes the bytes according to the *COSE* specification (CBOR Object Signing and Encryption);
5) verifies that the packet contains the *kid* field;
6) validates the JSON content [3] using the schema *"JsonSchema.kt"*;
7) decodes the *CBOR* object;
8) obtains the certificate from the repository using the *kid*;
9) validates the certificate using the validation function in *"VerificationCryptoService.kt"*;
10) checks that it is not black-listed.

C. Fuzz Testing Technique

In this field, the use of the **fuzz testing technique** can be very important in order to perform several tests with different payloads, thus forcing an app to crash under particular malicious code. The type of malicious code can be different, ranging from XSS attacks [15] to SQL injection [10], but also complex strings or symbols that cannot be correctly processed. For this reason, there are several tools that have been developed in order to perform tests regarding not only motion or click events on the screen, but also *activities* and *intents*, like *DroidFuzzer* [16] and *Caiipa* [13]. Going deeper, there is not a lot of literature regarding QR code fuzzing techniques used in the context of smartphone applications. As for now, there are for sure tools that are used to dynamically generate QR codes with malicious code that can be manually scanned by a human being. The malicious code can be injected in many ways depending on how the app parses and reads the content of the QR code. Though, at the time of this paper, we did not find any work related to this approach.

III. Method

A. Introduction

Once analyzed how the Green Pass is generated, we developed a script able to create QR codes with fuzzing strings, similar to the real ones with valid data. This is essential in order to test the higher number of possible steps, because the *VerificaC19* app returns from the checking function with a negative result as soon as one of them fails. After generating the QR codes, we automated the reading and result monitoring phases. The final composition of our tool is the following.

1) The **Fake Green Pass Generator** handles the generation of the QR codes. Unfortunately, we haven't been able to generate codes passing the last step of the checks (§II-B, step 9) because it requires a valid encryption using one of the private-keys held by authorized health institutes;
2) the **QR Code Visualizer** shows the QR codes on the computer screen, using a JSON file to synchronize with the mobile device;
3) the **Appium Test Automator** controls the mobile device, controlling the interactions with the app's UI, the screenshots and all the actions that would otherwise be performed manually by the user.

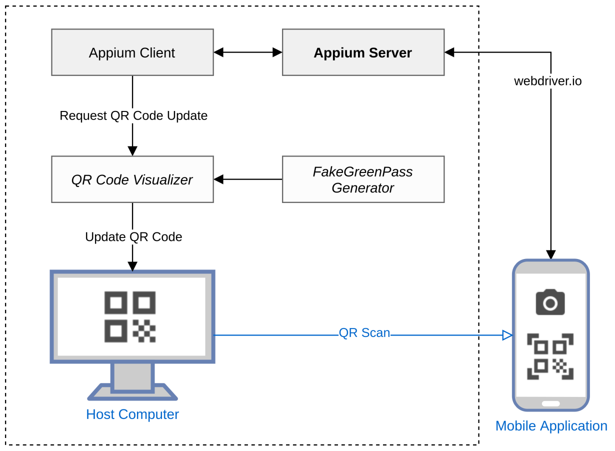


Fig. 2. Toolkit components overview

B. Fake Green Pass Generator

The generation of the Green Pass QR Codes is a crucial part of the tool. To do this, we took inspiration by two existing python scripts and developed a new one with the characteristics we needed. The first one is the tool *QRGen* [9]: it automatically generates malformed QR codes using different fuzzing strings organized in dictionaries. The second one is *Green Pass Generator* [6]: it implements the procedures to construct plausible Green Passes in a (mostly) correct format. Initially, our program takes as argument the dictionary file of strings to be injected into the Green Pass (or one of the pre-existing ones). Then, it creates a set of malicious Green Pass data, one for each word of the dictionary. In particular, the data objects contain the fuzzing string into the *Name* field of the Pass. Then, the Green Passes are compressed and encoded using the procedure explained in §II-B, in reverse order. Finally, we generate the QR codes using the Python library *PyQRcode* [14]. The generated images can either be saved to the `genqr` directory or directly displayed using the *QR Code Visualizer* module.

C. QR Code Visualizer

In order to visualize the QR code during the test phase, we implemented a Python script that creates and updates a UI window with the generated QR code. The script uses the fake green pass generator module to create a malicious QR code based on words in a dictionary. In particular, the behavior of the QR code visualizer is the following:

1) Loop through a list of generated payloads from the *Fake Green Pass Generator*.
2) Extract one payload from the list and generate a *bitmap* image, that is the raw format to be visualized inside the program window.
3) Once the QR code is visualized, check whether *Appium* required a new QR code through a common JSON file.
   - If *Appium* required a new QR code, the script picks the next item from the list and generate the new QR code to visualize.
4) Once the list is empty, wait for 10 seconds and exit the program.

Fig. 3. Example of QR code visualization

By iterating in this way, we can coordinate the *Appium* script to perform some actions inside the application and update the QR code automatically after performing all the tasks required (scan, collect the result, and so on).
During the execution, the visualizer (Figure 3) reports inside the terminal the payload used for the QR code, as well as the

name of the test executed. If something goes wrong everywhere with the QR code generation, the script automatically reports an error. For the QR code generation, it is also possible to configure the error correction (default is *low*) as well as the scale of the image inside the UI window.

D. Appium Framework

In order to perform actions inside the target app, we decided to use *Appium* [7], which is a popular testing framework used for both iOS and Android. This framework comes with three main components:

- **Appium Server**: this is mandatory, and it is used to actually make the script perform the actions required by a client.
- **Appium Client**: our custom script used to send required action following a precise iteration and to coordinate with the QR code visualizer.
- **Appium Inspector**: this component is optional and can be used to register and save the sequence in which we want to perform the iteration in our app (e.g. click a button, take a screenshot, go back and repeat).

The client component is responsible to invoke elements from the app executed on a smartphone connected over USB to the computer. Once is executed, the client send a request to the server using the *webdriver.io* library to perform click events or timeouts depending on the sequence of commands we want to do.

IV. Experiments

A. Experiment Setup

To verify the functionalities of our toolkit, we prepared a testing environment, composed by software and hardware components. A software-only approach should theoretically be feasible, but the simulated phone camera we tried, using the *Android Emulator Device* (based on the open-source software *Qemu*) and the *OBS Virtual Camera*, experienced very frequent crashes and errors. Beyond that, the objective of our experiment is to test the possible vulnerabilities of the Green Pass checking application *VerificaC19*, the app we took as example for our study. To obtain meaningful results, we need first to choose some fuzzing strings to insert into the data field of the payload. We selected the following dictionaries for a total of 344 words, because we thought that they could possibly have an impact with the technologies and the programming languages involved in the checking process:

- *jsondict* and *jsondict2*: both contains possible JSON injection string that might cause problem during the parsing phase of the payload;
- *mixed*: this contains strings and arrays with different dimension;
- *symbols*: here we used complex and composed symbols in ASCII;
- *emoji-mix*: in this dictionary we used a combination of different emojis mixed with ASCII symbols.

Regarding the toolkit configuration, the software components used for the experiments are:

- the **Fake Green Pass Generator** modules;
- the **QR Code Visualizer** to show the QR codes synced with the app;
- an **Appium Server** instance;
- the **Android Debug Bridge** (ADB) to communicate with the phone, whether it is a physical or a simulated device;
- the **NodeJS Appium Client** we wrote, that controls the app actions on the device.
- the **APK file** of the app under test, for us *VerificaC19*. Note that, in order to make Appium launch an activity, it needs to be settled as `"exported"` in the manifest file of the app.

The hardware component we settled up to overcome the emulator problems is structured as follows (Figure 4).

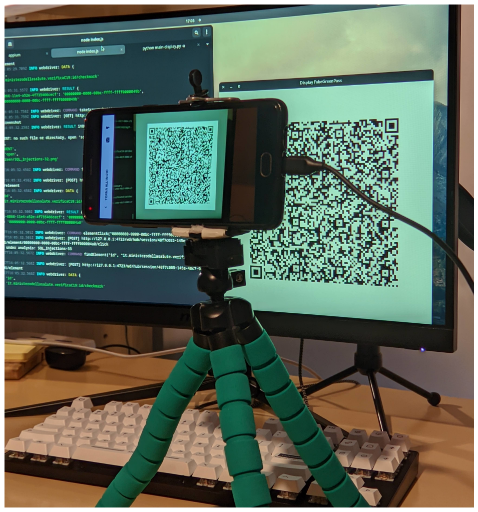


Fig. 4. Testbed setup

- A One Plus 3 smartphone, with Android 9.0 installed;
- A tripod, to adjust the phone height, so that it directly faces the PC monitor;
- A USB cable to connect the device through the ADB interface with the computer and the *Appium Server*.

The first step is to attach the smartphone to the computer and make sure that it is correctly recognized by the ADB Server (using the command `adb devices`). Then, we need to start the *Appium Server* into the host computer that will keep listening for new clients. We open the Python script *Fake Green Pass Generator*: it will display an initial mock QR code that will be substituted with the generated images whenever it will read the value `"status": 1` of the *fuzzer.json* synchronization file. Finally, with the phone camera pointing the QR on the monitor, we can launch the NodeJS client script for controlling the app behaviors. As previously described, the role of the controller is to install and launch the app, press the "scan" button and wait for a QR code to be read. When this happens, the script takes a screenshot, updates the *fuzzer.json* file and goes back to the QR scanning activity. If for any reason the QR code is not found after 10 seconds, the script saves the image information into a log file and continues with the other images.

B. Experiment Results

We performed our experiments with different kind of dictionaries in order to execute fuzz testing with multiple symbols and strings. We decided to save each result for every dictionary tested, meaning that we took a screenshot after scanning the QR code generated with the malicious payload, and then we reported any possible anomalies of the app inside a log file.

	Valid QR	Unrecognized QR	Total words
jsondict	43	0	44
jsondict2	34	0	34
mixed	115	4	119
symbols	127	1	128
emoji-mix	19	0	19

TABLE I
EXPERIMENTS RESULTS IN WORDS ANALYZED.

As shown in Table I and Figure 5, the scanning performed by the toolkit is overall always valid, even in a long-run test such as the *symbols* experiment. The unrecognized QR codes in *mixed* and *symbols* are due to their size in terms of payload. A QR code generated with a huge payload might cause the camera of the smartphone to be out of range (being visually bigger). Moreover, it can take longer time to process and so it is skipped by the toolkit.

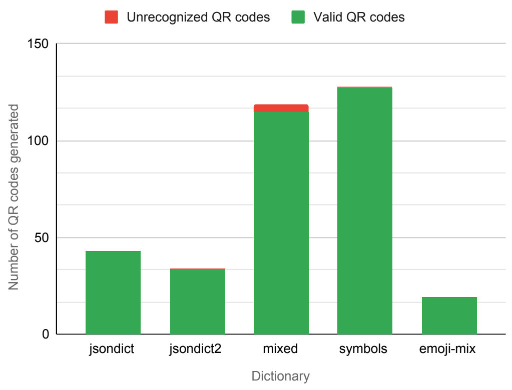


Fig. 5. Experiments result using 5 different dictionaries

In our results, we saw that the actual experiment with a tripod and a smartphone connected through a USB was correctly working. In addition, with over 300 QR codes tested, our toolkit was able to save almost every result page of *VerificaC19* (98.6% in total). As just said, we had a few of them that the script was not able to recognize, but in that case it is possible to retry those manually. We encounter no further problems in letting the toolkit process automatically and autonomously the fuzzing experiments once executed, therefore we can say that the toolkit should work even with larger dictionaries.
Based on the execution time of each experiment as shown in Table II, we had an average time to process each word of about *3.26 seconds* over a total of 5 different sessions with different number of words. Considering the overall results, we should point out that in each iteration the camera is turned on, and the reaction time relies on the hardware, so it might be different from smartphone to smartphone. Moreover, QR code recognition can be challenging in case of big payloads or high error correction rate.

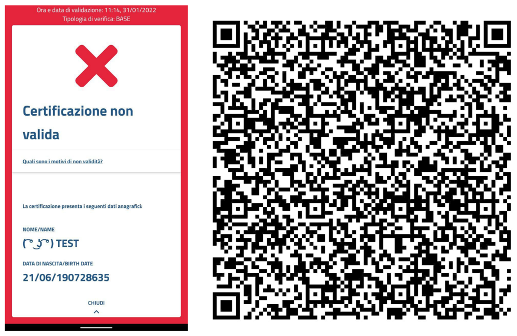
	Total seconds	Total words	Avg. sec. per words
jsondict	118s	44	2,68s
jsondict2	115s	34	3,38s
mixed	418s	119	3,51s
symbols	465s	128	3,63s
emoji-mix	59s	19	3,11s

TABLE II
AVERAGE COMPUTED TIME FOR EACH EXPERIMENT EXECUTION.

After testing all the words from the dictionaries, we were unable to make the application crash even after bypassing the initial validation checks. In fact, the signature verification process does not allow us to go further, so for now we were only able to show and change the name and the birthday in the payload, without changing the behavior of the application in the code.

C. Further Experiments

After completing the previous experiments, we searched for other vulnerabilities on different data fields of the Green Pass. In particular, we tried to change different fields, like the state of the vaccination or the swab type, but without success. Then, we tried to change the birthday field that is a string, formatted as `YYYY-MM-DD`. We discover that if we change this field with an unfeasible date, the result is parsed to a realistic one: for example, if we insert `2022-01-32`, the date displayed on the app is `2022/02/01`. Moreover, if we insert non-numerical characters inside the date, the app recognizes only the first part with numbers. Finally, if we insert a huge number in the correct format inside this field (e.g. over 2 billion as the year), the final result will be an invalid date corresponding to the highest supported date (i.e. `21/06/190728635`, as shown in Figure 6).


Ora e data di validazione: 11:14, 31/01/2022
Tipologia di verifica: BASE

Certificazione non valida

Quali sono i motivi di non validità?

La certificazione presenta i seguenti dati anagrafici:

NOME/NAME
(͡° ͜ʖ ͡°) TEST

DATA DI NASCITA/BIRTH DATE
21/06/190728635

Fig. 6. Screenshot of VerificaC19 with relative QR code and result

V. FUTURE WORKS

During the working period, we figured out what could be the possible future improvements to our project.

A. More Dictionaries

First of all, for what concerns the fuzzing dictionaries, we tested the app *VerificaC19* with only few malformed strings, without reaching any meaningful results in terms of bypassing the app controls. It is possible that we missed some particular payload that is more suitable for the application or languages involved, so a new investigation and string selection could succeed.

B. Test on Other Applications

Another possible differentiation from our work could be to change the application under test. Indeed, while *VerificaC19* is the official Green Pass verification app, many others have been developed and are currently in use. Some of that may not have the same robustness or safety. For example, some of the most used alternative apps in Italy are *COVID Certificate Check*, *EU Certificate Scanner Green Pass*; *Green Pass Italiano*. On the other hand, this toolkit can also be used for a different kind of apps, like for example *mobile sharing* apps, where the use of a QR code is required to use vehicles.

C. Parallelized Test Benches

We also thought of some ways to speed up the testing process. Given that the scanning phase is not instantaneous and that there are hundreds of possible fuzzing strings to test, it could be useful to parallelize it somehow. In particular, it should be possible to use different smartphones, connected to one single computer running multiple instances of the *Appium Server*. Hence, the *QR Code Visualizer* module should also be modified in order to generate more Green Passes simultaneously, each one synced with the correspondent device.

D. VM Implementation

As for another possible future work, there is the possibility to realize a Virtual Machine implementation of the toolkit, meaning that the need of a real smartphone to perform fuzz testing is no longer required. Though, this implementation should use a virtual video device that shows the QR code as a simulated camera in the smartphone emulator. Hence, this might be a reasonable solution to improve parallelism, since multiple instances of the same VM can be spawned and the experiment can be executed in parallel. In our research, before adopting our hardware solution, we found out that using *v4l2loopback*, which is a Linux library used to simulate the camera device, this implementation should work. On top of that, this solution requires high hardware resources to run 2 or more emulators at the same time.

VI. Conclusion

During our research, FuzzQR has been developed as a toolkit to perform QR code fuzzing tests over a smartphone application. In this way, we can automatically try the QR recognition part and see if malicious code can lead the app to errors or even to crash. This kind of attack can be very dangerous for the user privacy, as well as user credentials information, since in some apps like mobile sharing ones it might be possible to use QR code as vector attack. In our experiments, we tried to scan a malicious payload placed inside a Green Pass to make the scan checking app crash. Though, even if we could not make the app crash, we were able to build this toolkit that can be used in different contexts. Our experiments showed that with QR codes containing a small payload, the actual time to perform a single scan might take from 2 to 4 seconds, including result reporting (e.g. screenshots). Therefore, we can test many different dictionaries containing malicious words, so that there is no need to make this process manually. Although, if we use big payloads or higher error correction for the QR code generation, the smartphone camera might have some problems in focusing the correct target. On the other hand, FuzzQR is enough modular to be extended for different behaviors, and also it works for both Android and iOS apps.

References

[1] Andrey Averin and Natalya Zyulyarkina. Malicious qr-code threats and vulnerability of blockchain. In *2020 Global Smart Industry Conference (GloSIC)*, pages 82–86, 11 2020.

[2] EU Digital COVID Certificates. Eu digital covid certificate app core - android. https://github.com/eu-digital-green-certificates/dgca-app-core-android, 2021.

[3] EU Digital COVID Certificates. Eu digital covid certificate overview and specifications. https://github.com/eu-digital-green-certificates/dgc-overview, 2021.

[4] Ministero della Salute. Verificac19 app source code on github. https://github.com/ministero-salute/it-dgc-verificaC19-android, 2021.

[5] Ministero della Salute. Verificac19 sdk on github. https://github.com/ministero-salute/it-dgc-verificac19-sdk-android, 2021.

[6] Michele Federici. Green pass generator. https://github.com/ps1dr3x/greenpass-generator, 2021.

[7] JS Foundation. Appium.io. https://appium.io/, 2022.

[8] Zhongpai Gao, Guangtao Zhai, and Chunjia Hu. Qr-code structure image. In *The Invisible QR Code*, 10 2015.

[9] h0nus. Qr gen. https://github.com/h0nus/QRGen, 2019.

[10] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.

[11] Peter Kieseberg, Sebastian Schrittwieser, Manuel Leithner, Martin Mulazzani, Edgar Weippl, Lindsay Munroe, and Mayank Sinha. *Malicious Pixels Using QR Codes as Attack Vector*, pages 21–38. Trustworthy Ubiquitous Computing, 01 2012.

[12] Katharina Krombholz, Peter Frühwirt, Thomas Rieder, Ioannis Kapsalis, Johanna Ullrich, and Edgar Weippl. Qr code security – how secure and usable apps can protect users against malicious qr codes. In *2015 10th International Conference on Availability, Reliability and Security*, pages 230–237, 2015.

[13] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530, 2014.

[14] Michael Nooner. Pyqrcode. https://github.com/mnooner256/pyqrcode, 2016.

[15] Germán E. Rodríguez, Jenny G. Torres, Pamela Flores, and Diego E. Benavides. Cross-site scripting (xss) attacks and mitigation: A survey. *Computer Networks*, 166:106960, 2020.

[16] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing and Multimedia*, MoMM '13, page 68–74, New York, NY, USA, 2013. Association for Computing Machinery.