Trustedcam App

Project Report

Prof. Dr. Volker Skwarek

30.09.2023



Contents

[**Implementation** 3](#_Toc144922526)

[**Eliciting operating and security requirements from end users** 3](#_Toc144922527)

[**Conception of the security architecture** 4](#_Toc144922528)

[**Development of the operating environment** 6](#_Toc144922529)

[**Development of the system software** 6](#_Toc144922530)

[**Emulator Detection Mechanism Overview** 7](#_Toc144922531)

[**Frida Detection Utility Overview** 8](#_Toc144922532)

[**Magisk Detection Utility Overview** 9](#_Toc144922533)

[**Root Detection Utility Overview** 10](#_Toc144922534)

[**Device Fingerprint** 11](#_Toc144922535)

[**Blockchain transaction:** 11](#_Toc144922536)

[**Camera Driver Integrity through Secure Mobile Phone Registration** 13](#_Toc144922537)

[**Fingerprint & GPS:** 14](#_Toc144922538)

[**Blockchain Transaction** 14](#_Toc144922539)

[**Integration and system test** 16](#_Toc144922540)

[**AP6: Beta release** 17](#_Toc144922541)

[**Conclusion:** 17](#_Toc144922542)

**Table of Figures**

[Figure 1: Architecture of Trustedcam App 4](#_Toc144727369)

[Figure 2: Sequence chart of Trusedcam App 5](#_Toc144727370)

[Figure 3: Different status of integrity checks 14](#_Toc144727371)

[Figure 4: Usage of Fingerprint and Get Location button 15](#_Toc144727372)

[Figure 5: Stages of QR code generation 16](#_Toc144727373)

[Figure 6: Etherscan blockchain transaction 16](#_Toc144727374)

[Figure 7: Completion of user verification 17](#_Toc144727375)

# **Implementation**

This chapter consists of all the requirements submitted during the project proposal and the explanation of which the requirements has been fulfilled.

## **Eliciting operating and security requirements from end users**

In general, it can be expected that an untampered mobile phone records videos from the imaging chip with the correct time setting in the system time and the correct location, e.g. using GPS. Therefore, this is the phone's identity regarding imaging. In the previous "TrustedCam" project, an unaltered identity of the camera system can be taken as given as the manufactured delivers hardware and software as an unmodifiable bundle. For mobile phones, it can be easy to program an imaging app, imitating an imaging app but streaming prepared records into the "TrustedCam" system. This mimics the authenticity of a live stream, although a prerecorded stream was used. Our approach is to verify the identity of the mobile camera using different sources, including its publication on a public blockchain. Some of the functions are as follows:

* The raw GPS position is cross-checked with Google's augmented locating service or, e.g. ranges of IP addresses.
* The identity of the mobile phone components is acquired via system information, including the ID-number of the processor and the imaging sensor.
* This information is written to a public blockchain returning the block number, which can be roughly converted into a time stamp. This will be cross-checked with the GPS time and the system time.
* After processing the information, the blockchain writes it into a block delivering an unpredictable, random block hash value.
* The hash value must be printed within a limited time, e.g. 5 minutes, and recorded by the verification app. While recording it, the mobile phone creates a blinking pattern with its flashlight based on the random hash value.

If all tests succeed, the verification extension of the "TrustedCam App" has confirmed the untampered identity of the imaging path of the mobile phone by

* Capturing a visual flash pattern,
* Produced by the app itself,
* Including a QR code with a unique and random block hash,
* which represents the serial numbers of the mobile phone components, which the app itself can cross-check,
* including a time check with the block time of the transaction on the blockchain. The position is the weakest information as this can only be roughly verified with another locating service or the IP address range.

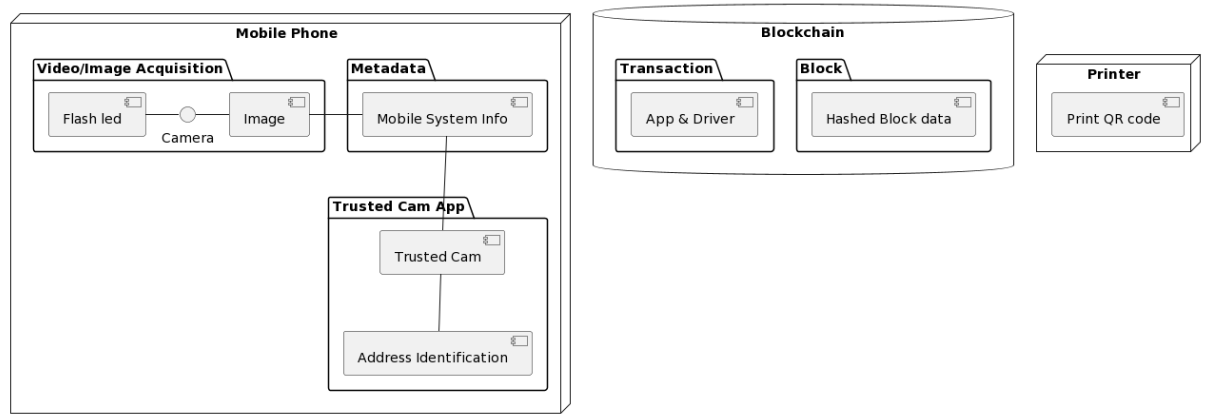
**Conception of the security architecture** 

Figure 1: Architecture of Trustedcam App

The architecture shore in figure 1 is explained below.

**Mobile Phone:**

* Video/Image Acquisition: The camera has two purposes here. One is to capture the image, and the other is to scan the QR code. The camera led flashes based on the QR code.
* Metadata: Mobile system information such as the processor model, the camera chip ID, and the time and location of the captured image are stored.
* Trusted Cam App: Downloaded into the mobile phone, creating smart contracts with available information.

**Block Chain:**

* Transaction: Unique transaction signature for the app and the driver.
* Block: The information created from the image and the metadata are hashed for higher security. The signature and hashed data are pushed into the blockchain.

**Printer:**

* Print QR Code: A QR code created from the hashed block data is printed to be scanned by the mobile camera.

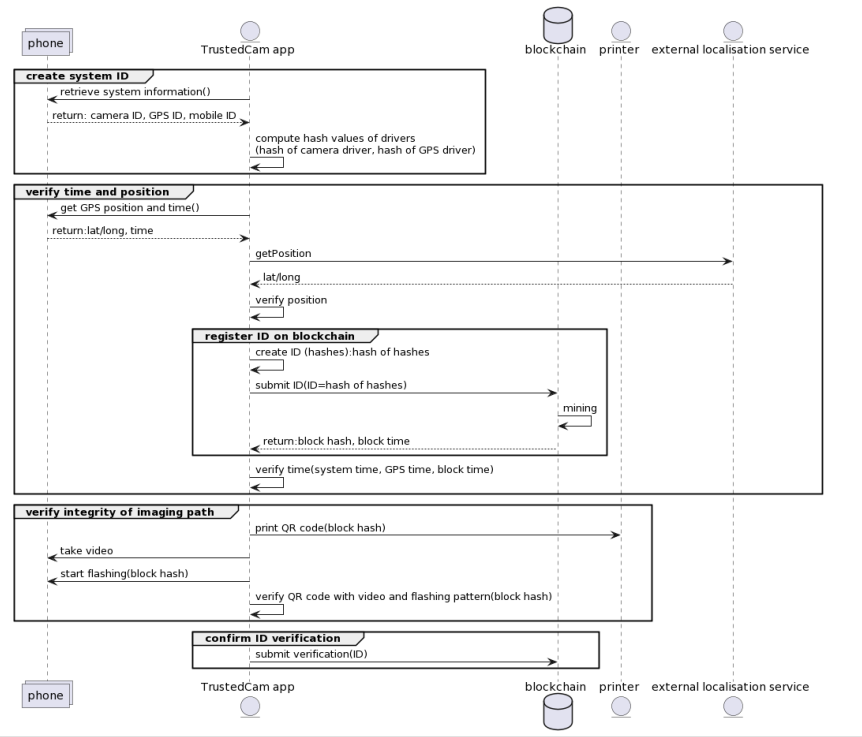


Figure 2: Sequence chart of Trusedcam App

Three levels of security are implemented:

* Level 0: video stream secured for integrity and authenticity by original TrustedCam algorithms,
* Level 1: integrity of drivers for imaging, time and location verified,
* Level 2: consistency of time and location verified for a single recording. In this article, we are mainly describing level-1-security. Level 0 is provided by the algorithms of the former "Trusted Cam"-project, and level 2 is obtained by repeating parts of the steps of level 1 for every single video. This increases security but also effort.

The process is shown in Figure 2: After installing the TrustedCam-app on the mobile phone, the identity of video-critical components will be verified, confirmed and stored on a public repository. For the verification, the following steps are executed:

1. The app accesses the mobile phone system information about the serial numbers of the processor, imaging sensor and GPS receiver if available.
2. The app computes the hash values of the GPS driver and the imaging driver and stores them locally, together with their file sizes and dates.
3. The GPS information is cross-checked with the Google augmented localization service and other position information such as IP ranges.
4. The system time is cross-checked with the GPS time and the derived position from step 3.
5. All information from steps 1 and 2 is the phone's identity for imaging. This is hashed and sent to a smart contract on a blockchain.
6. After mining the transaction, the block time and hash are returned to the app.
7. The app validates the block time with the time derived in step 4.
8. The app prints a random QR code, e.g., with the block hash in step 6, and converts the random value in a flash sequence, which needs to be immediately detected by the app. This step confirms that the path between the optical lens and the app has not been tampered.
9. If step 8 could be verified, this result is again recorded with the smart contract from step 5 on the blockchain using the same identity.

After these steps, the identity (ID) and the verification process are registered on a blockchain. This identity can later be used to verify the integrity and authenticity of a public video stream such as posted to Youtube or Instagram. This ID shall be part of the meta information of the stream but is also encoded by the original TrustedCam algorithms.

## **Development of the operating environment**

The entire code was developed in kotlin language and in android studio. For easy use from the user end, the user interface is developed in such a way that all the details can be seen on the screen and actions are straightforward and simple. The UI of the application can be seen in the figure 4. The “Fingerprint” button is to retrieve the device fingerprint , “Get location” is to get the location and time/date of the mobile phone during registration, “Verify user” to register the phone identity on the blockchain which inturn gives back a QR code based on the blockhash. The blockhash is also displayed on the screen.”Share QR code” button is used to share or print the QR code,”Scan SQ code” is to scan the QR code which verifies the integrity of the camera driver. The details of the blockchain transaction can also be retired by the user by using the blockhash as the reference in the etherscan website. The functionality of each and every button is given in detail under AP4 section.

## **Development of the system software**

In today's digital landscape, where mobile applications handle sensitive data and execute critical functions, ensuring robust security measures is paramount. Android applications are no exception to this need, and developers must employ effective methods to detect potential security threats. This report explores four essential detection mechanisms that empower Android applications to identify and respond to potential security vulnerabilities: Emulator Detection, Frida Detection, Magisk Detection, and Root Detection.

**Emulator Detection:**

Emulators serve as invaluable tools for Android app testing and development. However, distinguishing between emulated environments and genuine devices is essential to tailor testing strategies effectively. The "Emulator Detection" mechanism scrutinizes attributes such as build information, fingerprints, and hardware identifiers to determine if the application is running within an emulator. This enables developers to adapt their testing approaches and ensure comprehensive testing across various device types.

**Frida Detection:**

Frida, a dynamic instrumentation toolkit, can be used for unauthorized monitoring and manipulation of Android applications. The "Frida Detection" mechanism is designed to identify whether the application is being monitored by Frida or connected to a debugger. By employing methods like checking for the presence of the Frida server process and detecting debugger connections, this mechanism enhances the application's security posture and helps prevent potential breaches.

**Magisk Detection:**

Rooting Android devices through tools like Magisk can compromise application security and user data. The "Magisk Detection" mechanism aims to identify devices that have been rooted using Magisk. By inspecting paths associated with superuser binaries, Magisk-related files, and the presence of the Magisk Manager app, this mechanism empowers applications to apply additional security measures and counteract potential threats.

**Root Detection:**

Rooted devices can pose significant security risks due to elevated privileges. The "Root Detection" mechanism offers methods to identify rooted devices. By analyzing build tags, checking for common root-related file locations, and verifying the existence of known "su" binaries, this mechanism helps applications respond proactively to rooted environments, thereby enhancing overall security.

### **Emulator Detection Mechanism Overview**

In the context of Android application development, emulators play a crucial role in testing and debugging software before its deployment onto physical devices. To ensure the accuracy and reliability of an application, it's imperative to distinguish between emulator environments and real devices. To address this requirement, an emulator detection mechanism has been implemented, utilizing specific attributes within the Android operating system. This mechanism aids in identifying whether the application is running within an emulator or on an actual device.

**Purpose and Significance:**

The primary purpose of the emulator detection mechanism is to enhance the testing and quality assurance process by distinguishing between emulator instances and genuine devices. This distinction is significant because emulators emulate the behavior of real devices and may not fully replicate certain hardware or software interactions. Thus, identifying emulator environments enables developers and testers to tailor their approach, considering the limitations and unique characteristics associated with emulators.

**Mechanism in Detail**

The emulator detection mechanism employs a set of checks based on attributes derived from the Android device's build information. The following attributes are scrutinized:

1. Build Brand and Device: Emulators commonly feature the term "generic" in their build brand and device names. The mechanism examines whether the `Build.BRAND` and `Build.DEVICE` properties start with "generic."
2. Build Fingerprint: Emulators tend to possess fingerprints that either start with "generic" or "unknown." The mechanism evaluates whether the `Build.FINGERPRINT` property begins with these patterns.
3. Hardware: Emulator hardware often contains "goldfish" or "ranchu" in their names. The mechanism checks for the presence of these strings in the `Build.HARDWARE` property.
4. Model: Emulators can be identified through specific model strings like "google\_sdk," "Emulator," or "Android SDK built for x86." The mechanism examines whether the `Build.MODEL` property contains these identifiers.
5. Manufacturer and Product: Emulator environments frequently have manufacturer and product names associated with them, such as "Genymotion," "sdk\_google," "google\_sdk," and others. The mechanism verifies the presence of these strings in the `Build.MANUFACTURER` and `Build.PRODUCT` properties.

**Outcome and Application**

Upon conducting these checks, the emulator detection mechanism provides a conclusive determination. If any of the criteria are met, the mechanism considers the environment to be an emulator and returns a positive result. Conversely, a negative outcome implies that the application is likely running on a physical device. This determination empowers developers and testers to adapt their strategies and approaches accordingly, ensuring thorough testing and accurate evaluation in both emulator and device contexts.

**Conclusion**

The implementation of an emulator detection mechanism within the Android application serves as an indispensable tool for ensuring the robustness and reliability of the software. By utilizing distinct attributes inherent to emulator and device environments, this mechanism allows for context-aware testing and assessment. As a result, the emulator detection mechanism significantly contributes to the overall quality assurance process, ultimately leading to a more polished and reliable end product.

### **Frida Detection Utility Overview**

In the realm of Android application security, it's vital to safeguard against potential threats posed by tools like Frida, which can be employed for unauthorized monitoring and manipulation. To ensure the integrity and security of the application, a Frida detection utility has been implemented. This utility employs various techniques to ascertain whether the application is being monitored by Frida or connected to a debugger.

**Purpose and Significance**

The primary purpose of the Frida detection utility is to bolster the security measures of the Android application by detecting the presence of Frida and debugging connections. Frida, a dynamic instrumentation toolkit, enables real-time code manipulation, making it a potent tool for security breaches if used maliciously. Detecting the presence of Frida and debuggers helps ensure that the application's execution environment remains secure and uncompromised.

**Utility Functionality**

The Frida detection utility comprises several key functions:

1. isFridaServerRunning(): This function determines whether the Frida server process is currently running. It executes the 'ps' command with elevated privileges ('su') to list all running processes. The output is then analyzed to identify if the "frida-server" process is present, indicating the presence of Frida. Any exceptions that arise are handled by returning `false`.
2. isDebuggerConnected(): This function utilizes Android's `Debug` class to ascertain whether a debugger is currently connected to the application. It directly employs the `isDebuggerConnected()` function from the `android.os.Debug` class to provide a determination.
3. isFridaDetected(): This function combines the results of the previous two functions. It checks if either the Frida server is running (as determined by the first function) or a debugger is connected to the application (as determined by the second function).

**Outcome and Application**

The Frida detection utility yields a binary outcome: detection or non-detection of potential security threats. If the utility detects the presence of the Frida server or an active debugger, it indicates a potential breach in security. Application of this utility enables developers and security teams to proactively respond to potential threats, enhance security measures, and take appropriate countermeasures.

**Conclusion**

The inclusion of a Frida detection utility within the Android application's security framework stands as a testament to the dedication to robust security practices. By identifying the presence of Frida and debugger connections, the utility aids in preventing unauthorized code manipulation and access to sensitive information. This proactive approach to security ensures that the application's users and data remain protected, even in the face of potential security vulnerabilities posed by external tools and debugging connections.

### **Magisk Detection Utility Overview**

In the landscape of Android security, detecting the presence of root access, especially facilitated through tools like Magisk, is crucial for maintaining the integrity of an application. To uphold the security posture, a Magisk detection utility has been devised. This utility employs multiple methods to identify if the device is rooted through Magisk, thereby allowing for the implementation of necessary security measures.

**Purpose and Significance**

The primary objective of the Magisk detection utility is to reinforce the security framework of the Android application by identifying instances where root access has been achieved using Magisk. Magisk, being a sophisticated tool for systemless root access, can potentially compromise application security. Detecting Magisk usage is of paramount importance to prevent unauthorized modifications and safeguard sensitive data.

**Utility Functionality**

The Magisk detection utility encompasses the following primary functions:

1. isMagiskDetected(): This function consolidates the outcomes of the subsequent methods, returning `true` if any of the Magisk detection methods yield positive results.
2. checkForSuBinary(): This function validates the existence of known superuser (su) binary paths. It verifies whether any of the predefined paths where su binaries are typically located exist on the device.
3. checkForMagiskFiles(): This function identifies the presence of specific Magisk-related files on the device. It checks predefined paths for files associated with Magisk, signaling its presence.
4. checkForMagiskManager(): This function detects whether the Magisk Manager application is installed on the device. It employs the package manager to ascertain if the Magisk Manager package is present, indicating Magisk's potential existence.

**Outcome and Application**

The Magisk detection utility yields a binary outcome: detection or non-detection of Magisk-rooted environments. A positive outcome alerts developers and security teams to the presence of Magisk, facilitating the implementation of additional security measures, such as heightened scrutiny of application behaviour or access restrictions. The utility empowers developers to counteract the potential security risks associated with rooted devices.

**Conclusion**

Incorporating a Magisk detection utility within the Android application's security infrastructure underscores the commitment to robust security practices. By detecting the use of Magisk and its associated artifacts, the utility contributes significantly to preventing unauthorized access and manipulation of application data. The utility plays a pivotal role in maintaining data confidentiality and application integrity, even when confronted with devices that have gained root access through sophisticated tools like Magisk. Through proactive detection and response, the utility effectively enhances the application's security posture.

### **Root Detection Utility Overview**

Ensuring the security and integrity of an Android application requires identifying instances where the device has been rooted, as rooted devices pose potential security vulnerabilities. To address this, a root detection utility has been designed. This utility employs a series of methods to determine if the device is rooted, thus enabling the implementation of appropriate security measures.

**Purpose and Significance**

The primary purpose of the root detection utility is to bolster the security of the Android application by identifying devices that have been rooted. Rooting a device provides users with escalated privileges, which can be exploited to compromise application security. Detecting rooted devices is essential for preventing unauthorized access, data breaches, and potential manipulation of the application.

**Utility Functionality**

The root detection utility encompasses the following core functions:

1. isDeviceRooted(): This function consolidates the outcomes of the subsequent methods, returning `true` if any of the root detection methods yield positive results.
2. checkRootMethod1(): This method inspects the device's build tags to identify the presence of the string "test-keys." The existence of this string suggests that the device may be rooted.
3. checkRootMethod2(): This method scans common root-related file locations to determine whether specific files exist. These files, like "su" binaries, are indicative of rooted devices. It iterates through a predefined list of file paths and checks if the corresponding files exist.

**Outcome and Application**

The root detection utility yields a binary outcome: detection or non-detection of rooted devices. Detecting rooted devices prompts developers and security teams to consider implementing additional security measures tailored to these environments. This could include heightened scrutiny of application behavior, access control, or preventing certain sensitive operations on rooted devices.

**Conclusion**

Incorporating a root detection utility within the security strategy of the Android application underscores the commitment to safeguarding user data and maintaining application integrity. By identifying rooted devices, the utility facilitates the application of context-aware security measures, ensuring that sensitive information remains protected even in potentially compromised environments. This proactive approach to security allows developers to respond effectively to the challenges posed by rooted devices, enhancing overall security posture.

### **Device Fingerprint**

In the domain of Android application development, understanding the distinctive attributes and characteristics of each device is essential for tailored user experiences, security, and optimization. The utilization of the four detection mechanisms detailed above ensures the authentication and integrity of the device fingerprint without susceptibility to any form of tampering or malpractice.

**Purpose and Significance**

The central purpose of the "Device Fingerprint and Unique Device Identification" utility is to provide developers and users with an intricate insight into the individual attributes that set each device apart. A device's fingerprint encapsulates a plethora of attributes, including build information, hardware specifications, and processor details. These unique characteristics form the basis for creating an exclusive device identity, enabling the application to customize experiences and ensure security measures.

### **Blockchain transaction:**

In the rapidly evolving landscape of digital technology, the need for robust security measures has become paramount. With the integration of blockchain technology, innovative solutions are emerging to address various security challenges. One such solution is the storage of device fingerprints in the blockchain, resulting in the generation of a unique device ID through a single transaction. This approach leverages the power of the Ethereum blockchain to ensure secure device identification and enhance overall application security.

**The Role of Device Fingerprint**

A device fingerprint is a unique identifier derived from a combination of hardware and software attributes of a device. This fingerprint encapsulates the distinctiveness of each device, providing a reliable means of identification. Storing device fingerprints in a centralized or cloud-based system can be vulnerable to breaches. However, utilizing the blockchain's immutable and decentralized nature can offer a more secure solution.

**Device Fingerprint in Blockchain**

By storing device fingerprints in the Ethereum blockchain, an unprecedented level of security can be achieved. In our project, we use a utility to interact with the Ethereum blockchain. This utility retrieves the blockchain balance of a specified Ethereum address, and more importantly, enables secure transactions.

**Generating a Unique Device ID**

The process begins with obtaining the device's fingerprint using the `getFingerprint` utility. This fingerprint, unique to each device, serves as an identifier. Subsequently, the device fingerprint is embedded within an Ethereum transaction. The transaction's details are signed using the device's private key, ensuring authenticity and integrity.

**Transaction Execution and Result**

The Ethereum transaction encompasses the transfer of the device fingerprint to a specified address. This action triggers the Ethereum network to process the transaction. The Ethereum network ensures the transaction's validity, security, and integrity. Upon successful execution, the transaction results in the generation of a random number – the unique device ID.

**Advantages and Implications**

Storing device fingerprints in the Ethereum blockchain offers several advantages:

1. Security: The blockchain's tamper-proof nature ensures that stored fingerprints remain secure and unalterable.
2. Decentralization: The decentralized nature of blockchain eliminates single points of failure, reducing the risk of data breaches.
3. Immutability: Once stored, device fingerprints cannot be modified or deleted, providing a trustworthy source of device identification.
4. Authentication: The device ID generated from the blockchain transaction serves as an authentication token, enhancing user access control.

**Conclusion**

Storing device fingerprints in the Ethereum blockchain exemplifies the innovative fusion of device security and blockchain technology. By creating a secure, tamper-proof, and decentralized repository for device fingerprints, this approach elevates device identification to new levels of reliability. With a single transaction generating a unique device ID, this method not only ensures robust security but also exemplifies the transformative potential of blockchain integration in enhancing digital ecosystems. As the technology landscape continues to evolve, solutions like these underscores the profound impact that blockchain can have on modern security paradigms.

### **Camera Driver Integrity through Secure Mobile Phone Registration**

This section delves into a comprehensive process designed to ensure the untampered functionality of a mobile phone's camera driver. This process involves the generation of a random device ID, the creation of a QR code, the utilization of barcode scanning technology, and the transformative power of blockchain integration.

**Generating a Random Device ID and QR Code**

To initiate this process, a unique random device ID is created. This device ID is a one-of-a-kind token that encapsulates the distinctiveness of the mobile device. This random device ID then becomes the foundation for creating a QR code, a compact graphical representation of data. This QR code serves as a secure vessel to encapsulate the random device ID for future validation.

**Utilizing Barcode Scanning Technology**

The next step involves the usage of barcode scanning technology embedded within the mobile application. The mobile device's camera acts as the scanner, capturing the QR code's intricate details. The scanned data is then processed, converting the graphical QR code into a binary representation. This binary form becomes the basis for further transformative operations.

**Conversion to Flash and Integrity Verification**

The binary representation derived from the QR code is transformed into a sequence of flashes emitted by the mobile device's flash unit. This sequence holds the encoded information from the original random device ID. The values present before and after the flash sequence are meticulously recorded and preserved.

**Ensuring Camera Driver Integrity**

The process of camera driver integrity verification entails a meticulous comparison of three crucial values:

1. Random Device ID: The original random device ID generated during the initial phase as a result of successful blockchain transaction.
2. Values Before Flashing: The binary representation extracted from the QR code before transformation into flashes.
3. Values After Flashing: The binary values derived from the flash sequence.

**Completing Verified Mobile Phone Registration**

The validation process culminates when all three values—random device ID, values before flashing, and values after flashing—demonstrate exact alignment. A successful match among these values signifies the untampered status of the mobile phone's camera driver. This verified registration process establishes an unassailable trust in the camera driver's authenticity.

**Enhanced Imaging and Video Capture**

Upon successful registration, the mobile phone's camera is rendered utterly trustworthy. The camera driver's integrity ensures that any photos or videos taken are unequivocally authentic and devoid of any tampering. This paradigm shift in mobile device security empowers users with the confidence to capture moments with the assurance that their content remains unaltered and genuine.

**Conclusion**

The intricate process described above underscores the confluence of advanced technologies—random device IDs, QR codes, barcode scanning, flash transformation, and blockchain integration—to ensure the sanctity of mobile device hardware components. This approach exemplifies how innovation in security can redefine the way we perceive and utilize our mobile devices. By safeguarding camera driver integrity, this process charts a new path in digital authenticity, resonating with a future where trust and transparency reign supreme in the realm of technology.

### **Fingerprint & GPS:**

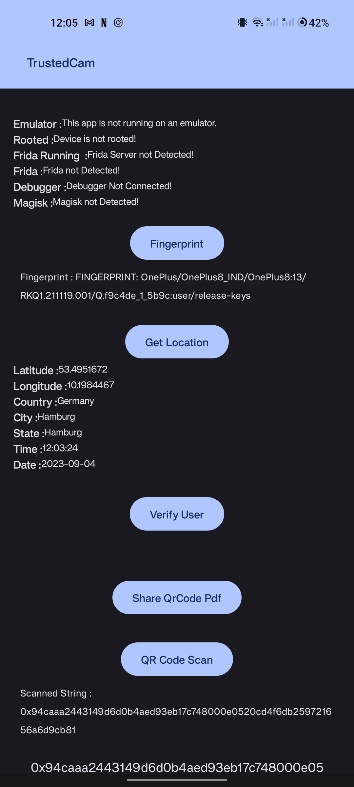


Figure 4: Usage of “Fingerprint” and “Get Location” button

From the screenshot in figure 4, you can observe the fingerprint data, as well as the acquired location and timestamp details gathered by the app.

### **Blockchain Transaction**

Upon clicking the "verify" button, the app triggers a blockchain Ethereum transaction, with the device fingerprint serving as the message. Once the transaction is successfully processed, it generates a blockhash. Subsequently, this blockhash is transformed into a QR code.Then this QR is scanned for verification of the itergrity of the camera driver as explained in the previous sections. The scanned string of blockhash and the actual block has can be seen in the screenshot in the figure 5.

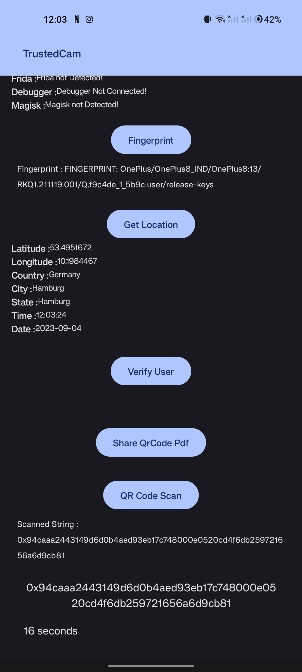
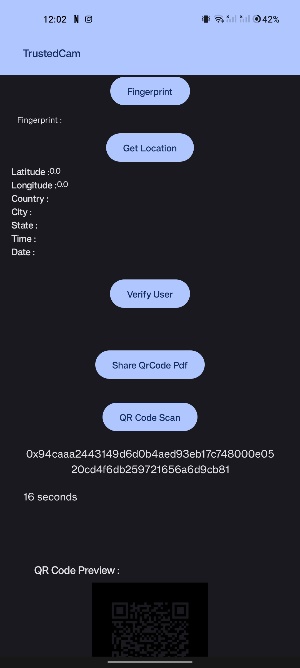
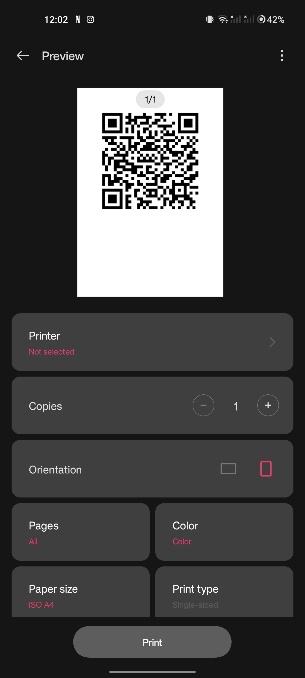


Figure 5: Stages of QR code generation

Details of the transaction on the etherscan website (<https://sepolia.etherscan.io/>).

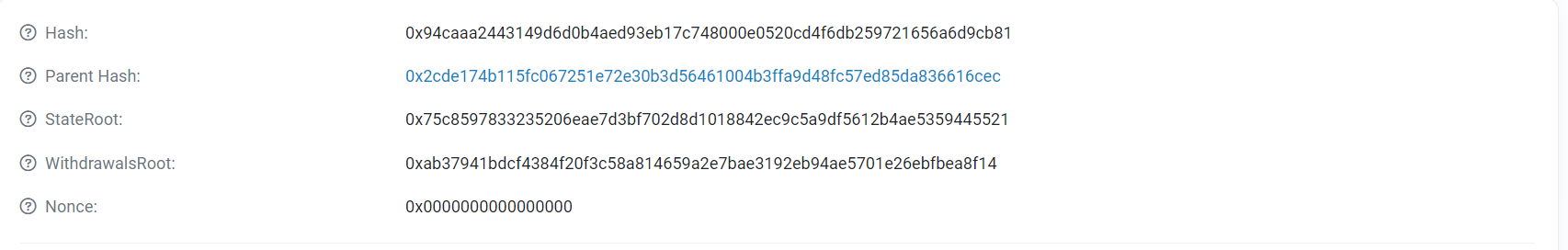
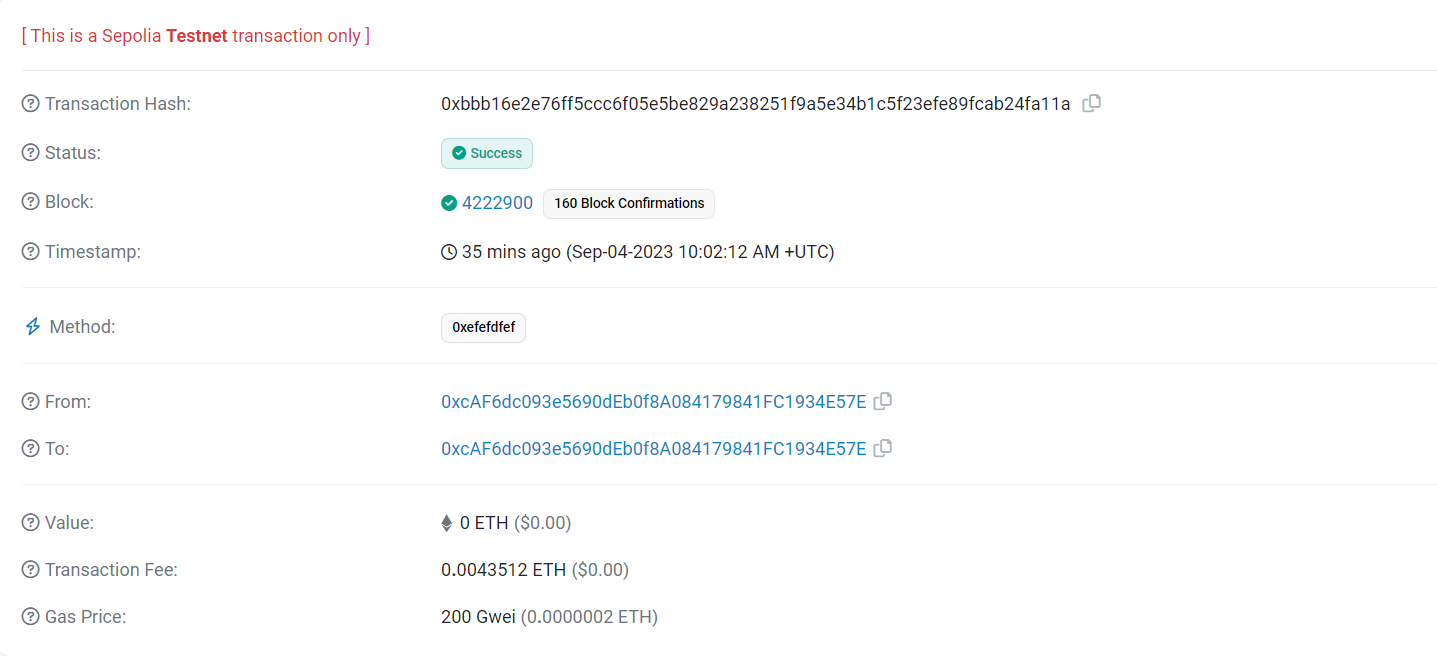


Figure 6: Etherscan blockchain transaction

In the figure 6 the string value against “Hash”, is the blockhash which has been used to generate the QR code.

After successful verification, a dialog box appears, indicating that the user has been "verified." It can be seen in the figure 7. This marks the completion of the registration process of the user.

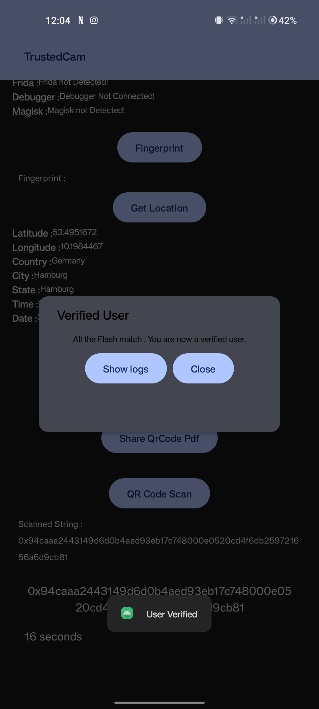


Figure 7: Completion of user verification

## **Integration and system test**

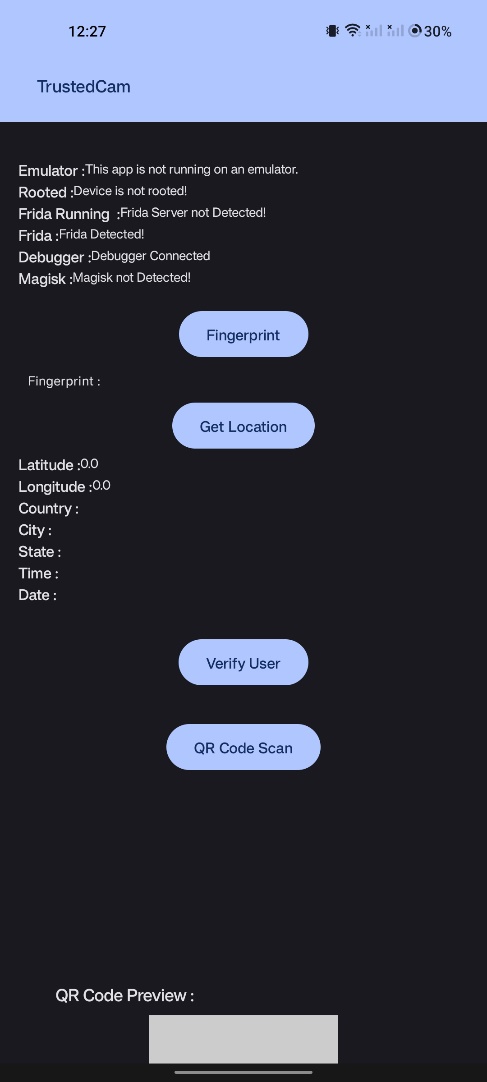
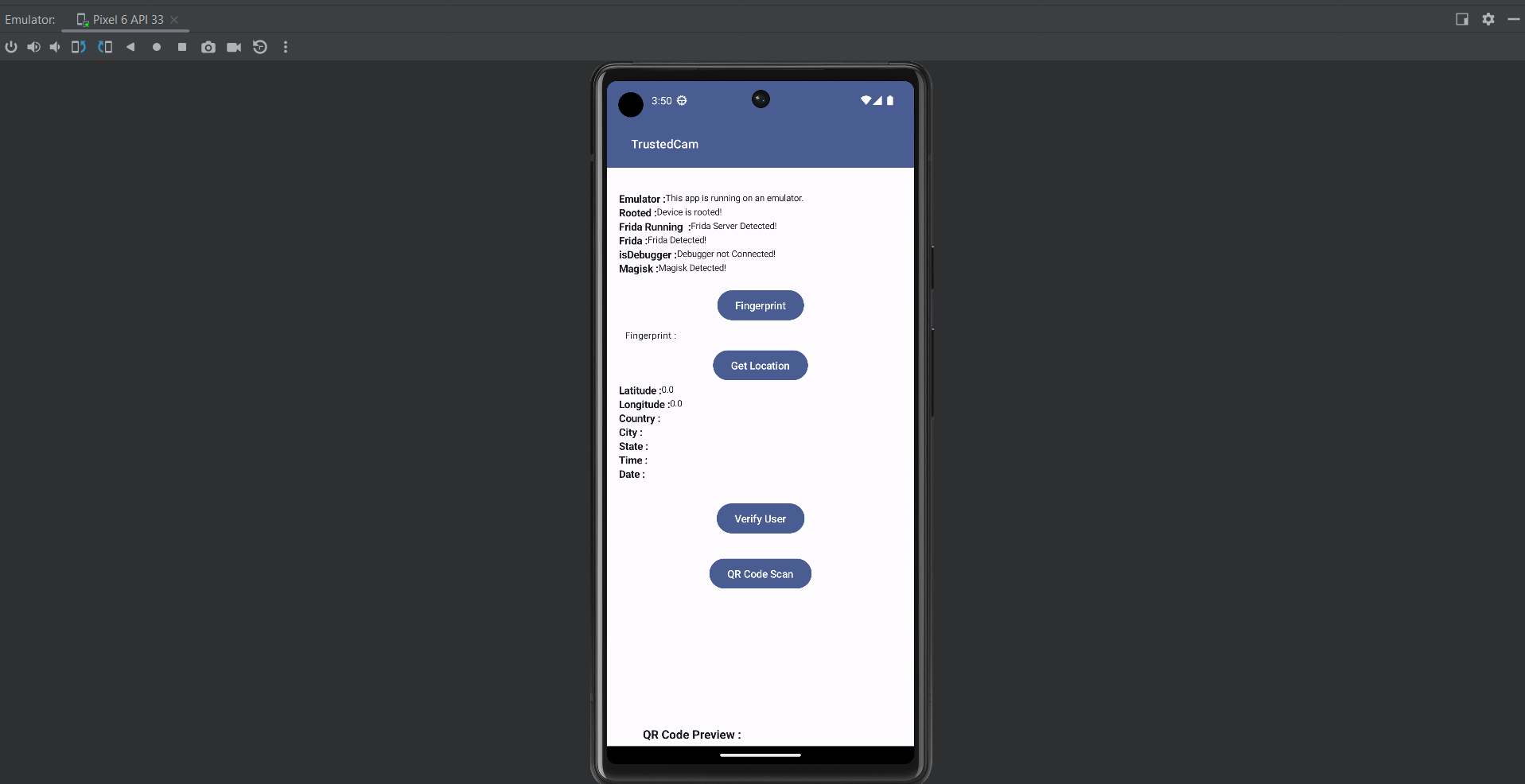
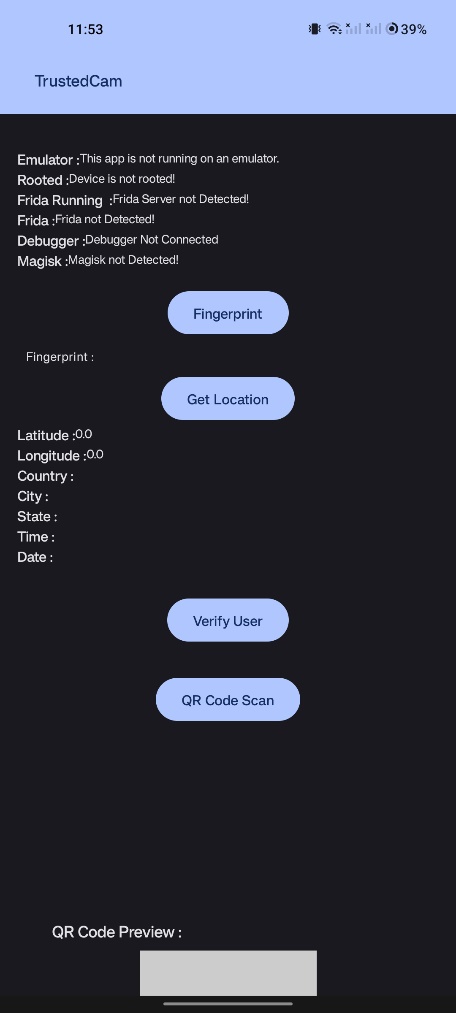


Figure 3: Different status of integrity checks

You can view the status of the integrity checks conducted on the TrustedCam App through the screen display illustrated on the left in the figure 3. In the middle screenshot of figure 3, you can observe the results of the checks when the app was executed on an emulator within Android Studio, with the mobile phone being rooted using Magisk. Additionally, a Frida script was connected to the app, and the app successfully detected its presence. When the TrustedCam App ran on an Android phone with a connected debugger, it also managed to identify the debugger, employing the mechanisms explained in previous sections, which is shown in the right most screenshot of figure 3.

## **Beta release**

The code has been uploaded in github ([suvenisa/Trustedcamapp\_HAW (github.com)](https://github.com/suvenisa/Trustedcamapp_HAW)) for the reference of the public. This contains all the details and codes which can be executed by anyone.

# **Conclusion:**

In conclusion, our approach to verifying the authenticity and untampered identity of mobile phone imaging pathways within the "TrustedCamApp" project is a robust and multifaceted endeavor. We acknowledge the potential vulnerability introduced by apps imitating imaging processes and streaming prerecorded content, leading us to implement a series of identity validation measures. By cross-checking raw GPS positions with augmented locating services and IP address ranges, we enhance the credibility of the device's geographical context. Additionally, we leverage system information, including processor and imaging sensor ID-numbers, to establish a comprehensive identity profile.

The integration of blockchain technology amplifies our verification process, where key identity information is encoded and written onto a public blockchain. By aligning blockchain's block number timestamps with GPS and system time, we fortify the accuracy of the captured information. Our approach culminates in the generation of an unpredictable and random block hash value, which is promptly transformed into a visual flashing pattern using the mobile phone's flashlight. This dynamic pattern carries an embedded QR code, encapsulating the mobile phone component's serial numbers, enabling cross-checks within the app itself.

Successful completion of our tests ensures the unaltered identity of the mobile phone's imaging path. The verification extension of the "TrustedCamApp" captures and analyzes the visual flash pattern, cross-checks serial numbers, and validates the blockchain transaction's block time alignment. While the positioning aspect remains relatively weaker due to its approximate nature, the combination of cross-referenced data, blockchain, and dynamic flashing patterns consolidates the trustworthiness and authenticity of the imaging process. This robust approach establishes a new benchmark in ensuring the credibility of mobile phone imaging and contributes to the broader mission of reinforcing digital integrity in the realm of mobile applications.