Horus: A Security Assessment Framework for Android Crypto Wallets

Md Shahab Uddin, Mohammad Mannan, and Amr Youssef(✉)

Concordia University, Montreal, QC, Canada
md_din@encs.concordia.ca, m.mannan@concordia.ca, youssef@cisse.concordia.ca

Abstract. Crypto wallet apps help cryptocurrency users to create, store, and manage keys, sign transactions and keep track of funds. However, if these apps are not adequately protected, attackers can exploit security vulnerabilities in them to steal the private keys and gain ownership of the users' wallets. We develop a semi-automated security assessment framework, Horus, specifically designed to analyze crypto wallet Android apps. We perform semi-automated analysis on 310 crypto wallet apps, and manually inspect the top 17 most popular wallet apps from the Google Play Store. Our analysis includes capturing runtime behavior, reverse-engineering the apps, and checking for security standards crucial for wallet apps (e.g., random number generation and private key confidentiality). We reveal several severe vulnerabilities, including, for example, storing plaintext key revealing information in 111 apps which can lead to losing wallet ownership, and storing past transaction information in 11 apps which may lead to user deanonymization.

Keywords: Crypto wallets · Cryptocurrency · HD wallets · Android apps

1 Introduction

Bitcoin is the world's most popular cryptocurrency, its value has recently surpassed US$60,000 [35] and is getting wider adoption in businesses. Other cryptocurrencies like Ethereum, and Litecoin have established their footprints and going strong among cryptocurrency users. The combined market cap of more than 6000 cryptocurrencies has reached a new high of US$1.24 trillion in 2021 [36]. Unfortunately, this massive growth of cryptocurrencies is also attracting malicious actors to find and exploit vulnerabilities in cryptocurrency-related technologies. The cryptocurrency community has witnessed no less than 34 attacks, breaches, and scams in 2020 alone [16,37], and attackers have stolen approximately US$4 billion [34] worth of assets from users.

Crypto wallets, being an ingrained part of cryptocurrency ecosystem, also encounter attacks ranging from deanonymization to password cracking [15,40,43]. Due to their importance, past work analyzed several crypto wallet apps. For

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 120–139, 2021.
https://doi.org/10.1007/978-3-030-90022-9_7

example, He et al. [26] analyzed critical attack surfaces in two wallet apps and demonstrated proof-of-concept attacks. Haigh et al. [25] analyzed the forensic artifacts of seven wallet apps, and designed a Trojan attack by repackaging a wallet app to steal user credentials; they chose to analyze non-HD (Hierarchical Deterministic [49]) wallets, although HD-wallets are gaining adoption in recent years and preferred over non-HD wallets in terms of security and portability.

To perform a comprehensive and scalable analysis of current and popular wallet apps (the number of which is growing, currently in the range of hundreds), we developed a semi-automated test framework, Horus[1]. Our framework combines both static and dynamic analysis of crypto wallet apps and can assess whether industry best practices are being followed in the app implementation. Horus has three major components: (1) *scraper module*, which can collect a large dataset of specific categories of apps from the Google Play Store; (2) *static analysis module*, which looks for API pattern to determine if proper security standards are followed in the app implementation; and (3) *dynamic analysis module*, which searches for key revealing information stored in the app's artifact on the device.

Using these components, we collect 310 wallet apps and analyze them for security risks. We conduct a manual inspection of the top 17 crypto wallet apps in the Google Play Store to understand the apps' security practices and evaluate the security risk associated with them. We use popular open-source vulnerability analysis tools, such as Androbugs[2] and Qark.[3] Syntax-based static analysis tools often fall short to determine vulnerabilities accurately [42]. We have noticed two issues with this approach, (1) a significant number of false-positive vulnerabilities and (2) discovered vulnerabilities are generic and do not represent issues specifically applicable to wallet apps. To complement static analysis tools, we also conduct a dynamic analysis to better understand the apps' workflow, and explore a broader set of vulnerabilities, including plaintext key storage, and exported components. In the final step, we inspect the decompiled apps' code to verify our findings and understand the app's operational mechanism in a greater detail. The main observation that follows from our analysis is that critical security standards applicable for wallet apps are missing in the app's binary, which indicates those standards are not implemented or have not been considered in the apps' development process. Additionally, the apps' key revealing information is handled insecurely (e.g., saved as plaintext or encrypted using poorly chosen encryption configuration). Our framework can accurately identify critical issues in wallet apps, and our automated analysis results are consistent with the manual inspection results.

Several design components of our semi-automated framework take into consideration the implicit and varying nature of the wallet apps. Firstly, a wallet-import step varies significantly from one app to another due to the heterogeneous user interface, and the effort requires to make it a generalized automated step is non-trivial. To overcome this problem, we develop a semi-automated framework

[1] Horus is one of the ancient Egyptian deities. The Eye of Horus is an ancient Egyptian symbol of protection.

[2] https://github.com/AndroBugs/AndroBugs_Framework.

[3] https://github.com/linkedin/qark/.

with a reduced-effort manual step (see Sect. 4.2). Secondly, most apps are still non-HD wallets and do not support wallet portability features. We developed a separate workflow to analyze non-HD wallet apps (see Sect. 4.2). Lastly, some wallet apps are heavily obfuscated which hinders our reverse engineering phase. To circumvent this problem, we focus on artifacts analysis instead of the obfuscated code. This technique also enables us to be compatible with mobile apps developed using hybrid and cross-platform frameworks (e.g., PhoneGap, Flutter) along with native apps developed for Android and iOS. For the purpose of this work, we keep our focus limited to analyzing Android apps.

Contributions. Our contribution can be summarized as follows:

1. We develop a semi-automated framework, Horus, to statically analyze wallet apps (fully automated), and perform dynamic analysis (with limited manual interactions).
2. We conduct automated analysis of 310 crypto wallet apps on the Android platform. Additionally, we inspect the top 17 most popular crypto wallet apps, with combined downloads of 46M+ and in total 84M+ of 310 apps, in Google Play Store to understand the apps' security practices and evaluate the security risk associated with them.
3. We reveal that 111/310 of the apps store key revealing information in plaintext and 18 HD wallet apps store the encryption key without additional protections, i.e., without Android Keystore or Android Hardware Security Module (HSM). Moreover, 11/310 apps store transaction information that can lead to deanonymization. Only 3/310 apps use HSM the best in class secure storage solution to protect key revealing information.
4. We find that the Android user dictionary can be leveraged to derive the mnemonic phrase used to generate the private key. Only 20/310 apps implement custom keyboard to safeguard against this attack.

2 Background and Threat Model

In this section, we provide brief descriptions of some wallet-related terminologies and our threat model.

Crypto wallet apps generally generate new addresses, store private keys securely, and help automate transactions. Some wallets can handle only one type of cryptocurrency (e.g., Bitcoin), and others can handle multiple types of cryptocurrencies. Furthermore, there are two types of wallets: Hierarchical Deterministic (HD) wallet [49], and non-HD wallet. HD wallets organize user accounts from one or more seed values and utilize open-source community-driven protocols to perform each operation, such as generating seed and creating private keys. On the other hand, a non-HD wallet randomly generates keys (i.e., no connection or hierarchy between the keys); such unrelated keys are also known as JBOK (Just-a-Bunch-of-Keys).

Bitcoin Improvement Proposals (BIPs). Several open-source community-driven protocols known as BIP [12] facilitate various crypto wallet functions.

Each proposal is responsible for a specific goal, and the Bitcoin community can propose, rectify, establish, approve or reject proposals by consensus. As of March 13, 2021, there are 140 BIPs [12], but 3 BIPs are primarily relevant for HD-wallet apps. (1) *BIP 32* which defines a tree structure to populate public-private key pairs from a seed. The seed allows the wallet to be interchangeable with different implementations/devices, and implies the wallet does not need to be backed up often; just saving the seed is enough to recreate the tree structure of keys [49]. (2) *BIP 39* which defines both generating a mnemonic phrase and how to create a seed from the phrase, as compared to a hexadecimal random seed, a phrase is easier to remember and store for users. This proposal also defines a list of 2048 common English words to be selected for mnemonic phrases. The chosen words for a mnemonic phrase and their order are needed to regenerate the same seed afterward. The word list is also available in multiple languages [39]. (3) *BIP 44* which defines a syntax to enable a multi-account hierarchy of keys based on BIP 32. The syntax expresses purpose, coin type, account, address index to generate proper keys [38]. The proposal defines how to generate any number of cryptocurrency-specific child keys.

HD Wallet Generation. A mnemonic phrase is generated using the standardized process defined in BIP 39. The most common phrase length is 12 and 24 words. In Android wallets, the 12-word length phrase is prevalent. First, a random sequence of 12/24 words is selected, providing 128–256 bits of entropy [9]. Then, the PBKDF2 (Password-Based Key Derivation Function 2) key derivation function is used to derive a 512-bit seed from the word sequence. This seed can be used to deterministically generate an HD wallet [49]. An HD wallet root consists of a pair of a master private key and a master chain code. Next, a master public key is generated from the master private key. Both the master private key and the master chain code are used to generate child keys using BIP 44 [38]. Note that there is no way to verify the words and their order in a phrase. If the user adds/removes/scrambles the phrase's words, a new seed is generated, leading to some other wallet keys. The phrase, seed, master private/public keys, and master chain code are all key revealing information and should be protected with equal importance.

Threat Model. We assume the attacker can install a malicious app on the victim's device with the following capabilities (or a subset of those). The app can have virtual keyboard permission, or it is a keyboard app itself, which is set default by the user. The malicious app can take a screenshot of the device. We assume the device is not rooted by the user; however, a malicious app can successfully root the device. Attackers can have physical access to the unlocked device for a few minutes. Note that all these capabilities are not required for all our attacks.

3 Dataset Collection

Google Play Store search displays only the popular apps based on its search algorithm, but does not provide a comprehensive list of all the apps matched with

a search term [10,41]. Conversely, a web search engine provides an exhaustive list of apps matched with the provided search term. We develop a specialized search engine scraper module based on a generic scraper tool, Search Engine Scraper,[4] for collecting a large set of apps. Our scraper module can search, parse results, remove duplicate app IDs, and download the APK (Android Application Package) files from Google Play Store automatically. We have used search term site : play.google.com "bitcoin" "wallet".

We use the search engine bing.com due to API restrictions in google.com (up to 100 pages, but multiple test runs from the same IP address may result in the IP being blocked before this limit is reached). We use a user agent value to appear as a desktop browser to the search engine, and introduce a random delay before scraping each new search result page to emulate normal usage and avoid any API restrictions.

We scraped 24,800 search results and found a total of 636 APK links, with 442 unique app IDs. We use PlaystoreDownloader[5] to download the latest version of the app from the play store. We encountered some exceptions during app download, including: some apps are unavailable in our location (Canada), some apps are available only via the early access program, and some apps are incompatible with our device. Eventually, we downloaded 392 apps and found 82 apps were not wallet apps although the term "wallet" appears in their description; such apps include Bitcoin key generator, currency exchange service, etc. We filtered out non-wallet apps and obtained a collection of 310 wallet apps. We shortlisted all the apps with at least 1M+ downloads in Google Play Store (total 17 apps, accessed: 2021-02-03) for further manual inspection due to their high user base, see Table 1.

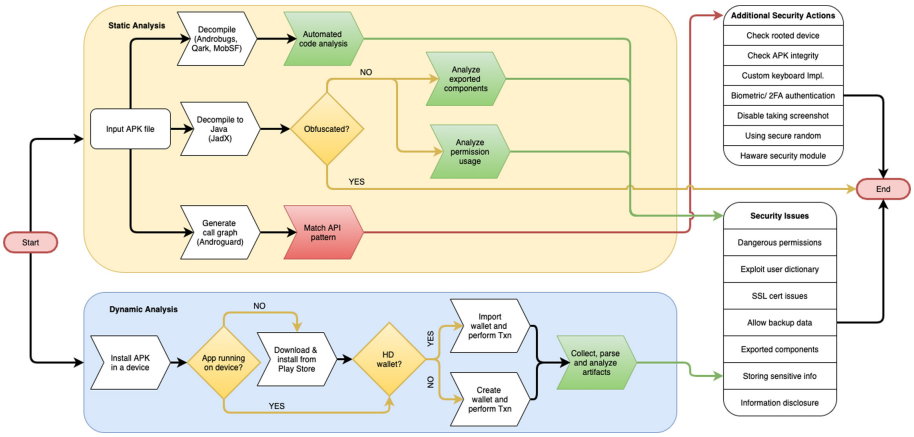

Fig. 1. Overview of proposed framework

[4] https://github.com/tasos-py/Search-Engines-Scraper.
[5] https://github.com/ClaudiuGeorgiu/PlaystoreDownloader.

Table 1. Top 17 most downloaded wallet apps in Google Play Store (accessed: 2021-02-03). ✗ indicates non-HD wallet app and ✓ indicates HD wallet app.

Application ID	Version	Downloads	HD?	Supported coins
asia.coins.mobile	3.5.22	5M+	✗	BTC, XRP, ETH, BCH
co.bitx.android.wallet	7.6.0	5M+	✗	BTC, ETH, XRP, USDC
co.mona.android	3.84.0	1M+	✓	80+ coins
com.binance.dev	1.36.3	5M+	✗	200+ coins
com.bitcoin.mwallet	6.9.10	1M+	✓	BTC, BCH
com.breadwallet	4.7.0	1M+	✓	BTC, XBT, BCH, ETH
com.coinomi.wallet	1.20.0	1M+	✓	125+ coins
com.mycelium.wallet	3.8.6.1	1M+	✓	BTC, ETH, ERC-20
com.paxful.wallet	1.7.1.534	1M+	✗	BTC, USDT
com.polehin.android	3.4.9	1M+	✗	100+ coins
com.unocoin.unocoinwallet	3.4.7	1M+	✗	BTC, ETH, XRP, LTC
com.wallet.crypto.trustapp	1.26.5	5M+	✓	50+ coins
com.xapo	5.3	1M+	✗	BTC
de.schildbach.wallet	8.08	5M+	✗	BTC
org.toshi	23.3.357	1M+	✓	100+ coins
piuk.blockchain.android	8.4.7	10M+	✓	BTC, ETH, BCH, XLM
zebpay.Application	3.12.02	1M+	✗	BTC, ETH, XRP, EOS

4 Horus: Our Analysis Framework

The purpose of developing Horus is to automate the analysis process of wallet apps and discover issues in the apps' implementation. The idea and rationale behind building this framework and design decisions for different wallets are explained in this section. An overview of our evaluation methodology is presented in Fig. 1. There are two modules for app analysis in Horus: *static analysis module* and *dynamic analysis module*.

4.1 Static Analysis Module

The static analysis module of Horus looks for API patterns to determine if a particular functionality or security feature is implemented in the app. For instance, a wallet app must implement a custom keyboard; otherwise, it is vulnerable to user dictionary attacks; see Sect. 5. We verify if the APIs required to implement a custom keyboard in Android are called in the app. If the API calls are found, the functionality is assumed to be implemented in the app.

To determine the API calls in an app, Horus takes an apk file as input and constructs a call graph of the app using Androguard.[6] A call graph contains the class name, method name, descriptors, and access flags. Each node in the

[6] https://github.com/androguard/androguard.

call graph is a method, and the actual calls with arguments are denoted using edges. For root detection, the app checks the execution of su command [29], or the existence of a list of root enabler apps [19] on the device. The call, Runtime.exec() is used to check the existence of su binary and PackageManager.getPackageInfo() is used to check the existence of a root enabler app. The API calls in the app's

Table 2. API signatures to check for feature existence.; the *Type* column denotes the security standard type, the *API Signature* column points to the API calls and individual class names required to implement the security standard, and the *Usage* column indicates the use case of each API call.

Type	API signature	Usage
Root Detection	Runtime.exec()	Execute runtime command, e.g. su
	PackageManager.getPackageInfo()	Get installed app's information
	Os.stat()	System call for runtime command execution
	Os.access()	System call to query installed app
Integrity Check	PackageManager.getPackageInfo()	Get installed app's information
	Context.getPackageCodePath()	App installer file path
	ZipFile.init()	Execute operation on installer file
	RandomAccessFile.init()	Read, write in system file
Custom Keyboard	KeyboardView.setKeyboard()	Replaces default keyboard app
	OnKeyboardActionListener.onKey()	Listeners for input key
	InputMethodService.onCreateInputView()	Callback when the keyboard view is created
	InputConnection.commitText()	Commit the user input to app
	InputMethod	Handles keyboard type depending on input field
Biometric Authentication	BiometricManager	Provides biometric utilities
	BiometricPrompt	Handles biometric authentication
	FingerprintManager	Defines types of authentication (Deprecated)
	BiometricService	Updates system server for fingerprint
	FingerprintService	Updates system server for fingerprint (Deprecated)
Screenshots Disabled	Windows.setFlags()	Uses for full screen access
	View.setDrawingCacheEnabled()	Access to the current view displayed
Hardware Security Module	KeyStore.getInstance()	Returns a keystore object of specified type
	KeyGenParameterSpec.Builder.isStrongBoxBacked()	Requesting Android to use hardware module
	StrongBoxUnavailableException	Exception if the hardware module is absence
Random Generator	SecureRandom	Cryptographically strong random number generator

call graph indicate the app is checking whether the device is rooted. Table 2 depicts the API signatures we use to determine the existence of the security standard in an app. Below, we discuss several security standards that we verify using our static module.

Secure Random: A secure random number generator is crucial for crypto wallet apps. According to BIP 39 [39], to create a mnemonic phrase, the client must use an entropy of length 128–256 bits. A seed is generated from the mnemonic phrase and used to create the master private/public keys. If the random number generator is predictable, it affects the key generation objective. In Android, /dev/urandom [47] is used to generate seed for SecureRandom [7], which is the recommended API to generate a cryptographically strong random number.

Custom Keyboard: Once a wallet app generates a mnemonic phrase, it asks the user to enter the mnemonic phrase for verification. To import an existing wallet into a new app, the user also needs to enter the mnemonic phrase via keyboard. If the user uses a third-party keyboard app, it can capture all user inputs [5], including the mnemonic phrase. Additionally, the app is vulnerable to user dictionary attack. A wallet app should have a custom keyboard triggered while taking any key revealing input and possibly randomize the keyboard's key location [32].

Disabling Screenshot-Taking: This is another critical risk avoidance feature for wallet apps. Any malicious app with the capability of taking a screenshot can capture the screen content during the wallet import phase. Also, a user may take a screenshot of the mnemonic phrase to save it as a backup. This image is saved in the gallery, and any app with reading storage permission can access the file. Crypto wallet apps must call Android API to disable the screenshot-taking feature for sensitive screens.

Two Factor Authentication (2FA): 2FA or biometric authentication should be implemented before performing any sensitive operation. We observe a significant number of wallet apps do not require user registration. An unauthorized user can perform a transaction with a few minutes of physical access to the device, assuming it is unlocked. Some apps require the user to set a PIN code and ask for the PIN code when confirming a transaction. However, a PIN code can be as short as a 4-digits number which can be brute-forced or seen by shoulder surfing. A 2FA should be incorporated in each wallet app as second-layer protection.

Integrity Check: App's signature verification to check integrity ensure that the app has not been tampered with and installed from a legit source (e.g., Google Play Store). Integrity can also be ensured by calculating a hash of the installed APK file and comparing it with the hash of authenticated APK file. The solution is not foolproof and can be bypassed in a repackaged app. However, integrity checking is a widely adopted practice in financial apps and is considered a self-defense mechanism. The wallet app must perform integrity checks as another layer of security before starting to operate.

Root Detection: An app can be made more secure by implementing 2FA or strong encryption; however, no security on the app end works if the device itself is compromised. Malware apps can have root providers included in the binary and may gain root privilege without the user's consent [50]. This privilege can be exploited in many ways, from monitoring activities of other apps to sending key revealing information to a malicious back-end.

Hardware Security Module: Recent Android smartphones have a separate processor, which provides additional security to keep key revealing information safe, called hardware security module [4]. The key revealing information is stored in a secure enclave that is also protected in a rooted device and safe against brute force attack by utilizing rate limiting. The solution is not foolproof; there is still room for information disclosure if the device contains a malicious keyboard app or a clipboard listener. Nevertheless, the hardware security module provides substantial improvements over any other storage solutions and should be used by wallet apps to secure key revealing information.

Manual Inspection. For in-depth inspection of the top 17 wallet apps, we use three state-of-the-art analysis tools, AndroBugs, Qark, and MobSF.[7] The tools look for generic vulnerabilities by following a defined set of rules and patterns and do not require the app to run. All these tools start with decompiling the app code and looking for several vulnerabilities, including: runtime command execution, SSL certificate verification, and webview vulnerabilities. The tools mark the found vulnerabilities with different severity labels (e.g., Critical, Warning, Info). We consider only the critical vulnerabilities.

We use reverse engineering to discover vulnerabilities in the app code by decompiling the APK file. Reverse engineering reveals permissions usage in the app for malicious purposes, logic bombs, Trojan code, etc. Sometimes the app code is obfuscated, by renaming code components, such as folders, classes, variables, into a shorter, unintelligible name [8]. It is difficult to get out any meaningful information from the obfuscated code; we skip the app in such cases. We use APKTool[8] and JD-GUI[9] for our reverse engineering step. Apktool can decode resources from the Android APK file, and JD-GUI is a Java decompiler and code browser.

4.2 Dynamic Analysis Module

In the dynamic analysis module, we look for key revealing information in apps' internal file structures. If the device is rooted or the target app allows backup, malicious apps or actors can capture the target app's internal files. If the internal files contain key revealing information (e.g., plaintext private key), a malicious actor can easily access such information. In general, the master private key is the most critical information that needs to be protected by wallet apps. Other such

[7] https://github.com/MobSF/Mobile-Security-Framework-MobSF.
[8] https://ibotpeaches.github.io/Apktool/.
[9] https://github.com/skylot/jadx.

critical items include the mnemonic phrase, seed, and master chain code—all of which should be protected with equal importance as of the master private key. Note that a mnemonic phrase is used to generate a seed value, which is used to generate its corresponding private key. Similarly, the master public key and any of the child's private key is enough to recreate the master private key [24]. So it is imperative to secure all of the key revealing items along with the master private key. Our goal is to seek answers to the following questions: Are wallet apps storing the above-mentioned key revealing information in plaintext on the device? If encrypted, which encryption algorithm is used? Can we identify the encryption key used to perform the encryption, and if yes, where is the encryption key stored?

HD Wallet Workflow. A HD wallet can recreate the hierarchical tree of keys from a mnemonic phrase. In our HD wallet workflow, we maintain four lists as follows. (1) *Key revealing information:* A list of secret information that can be used to regenerate the master private key. In addition to the master private key itself, this list includes mnemonic phrase, seed, seed hex value, BIP 32 private key, and master chain code. (2) *Candidate encryption key:* Key revealing information should be encrypted before storing on the device. *Candidate encryption key* is a list of all possible encryption keys that wallet apps can use to encrypt key revealing information. We find traces that encryption keys are stored in the app's internal files. Using Horus, we can verify how widespread the practice is to store encryption keys locally. (3) *Cipher key revealing information:* Each item of the *key revealing information* list is encrypted and the resulting ciphertext is encoded using Base64 and then appended to this list. (4) *Search term:* A combined list of *key revealing information* and *cipher key revealing information*. We use this list of items as a search term and look for each item of this list in the wallet's artifact.

To start the dynamic analysis, we install the target app on a rooted device. We generate a mnemonic phrase and import the same phrase in all HD wallet apps. We use a fixed email, username, PIN, password, and phone number in all the apps for account creation, and verification as needed by the app. We append this fixed information in the *candidate encryption key* list as potential encryption keys.

From a mnemonic phrase, Horus gets *key revealing information* list, and by parsing internal files, Horus get *candidate encryption key* list. Horus iterates through the *key revealing information* list and applies encryption algorithms available in the Android platform to encrypt each of the items in *key revealing information* by using items from *candidate encryption key* list as the encryption key. One such operation is as follows, we take the first element of *key revealing information*, encrypt it with AES using the first element of *candidate encryption key*, and add the resulting ciphertext in *search term*. We then repeat the same operation with the next element of *candidate encryption key*, and so on. When done, we use a different encryption algorithm (e.g., Blowfish) and go through the same list of *candidate encryption key* and repeat the process. All the *key revealing information* is also appended as-is to *search term* list because the *key revealing*

information can also be found in internal files without any encryption. Horus reads all the app's internal files to find traces of the elements in *search term*. If any of the *search term* element is found, then the wallet app is exploitable if the internal files are exposed.

Horus generates a mnemonic phrase used for all HD wallet apps throughout the analysis. Horus starts a tcpdump[10] session to capture all network requests and clear the logcat buffer for capturing a new session. At this stage, we import the mnemonic phrase in the app. Then, Horus takes the app ID as input and pulls all internal files, tcpdump generated network dump files, from the connected emulator/device, and reads the files sequentially.

Horus identifies common file extensions (e.g., xml, db, pcap) and uses an appropriate parser to extract the file content. In case of an XML file, Horus parses the XML file and reads the content and appends all the string values in *candidate encryption key*. For the SQLite database file, we develop a parser that lists all the tables in the database and reads all the values in the tables and appends them in the same list. We get a pcap file from tcpdump containing network request logs, and we use a Python library Scapy[11] to parse the pcap file and extract the content. Horus can also parse log files and take fixed email, username, PIN, and password used as input. All the parsed content and input are considered potential encryption keys and listed in *candidate encryption key* list.

Horus reads all the internal artifacts again and runs a fuzzy search, using fuzzywuzzy,[12] for the items listed in *search term*. We record the search result whether *search term* items are found and whether in plaintext or encrypted form. Note that all *candidate encryption key* items cannot be used directly as encryption keys. AES requires blocks of 16 bytes key, whereas a PIN is a four-digit number. Thus, a PIN cannot be used as a key. We use four common hashing algorithms (MD5, SHA1, SHA256, and SHA512) to generate a digest for each *candidate encryption key* and use the digest instead as an encryption key. Throughout our reverse engineering step, we observed that the hashing technique to convert a non-suitable key into a proper encryption key is followed in many wallet apps.

Non-HD Wallet Workflow. A non-HD wallet generates a list of public-private key pairs, and there is no relationship among the keys. Non-HD wallets manage many keys; each public-private key pairs are used for only one transaction. The downside of this approach is that the user needs to take backup regularly, ideally after each transaction. This approach is not convenient and error-prone. Additionally, there is a risk of key-exposure if the backup is not handled with caution. In Horus, non-HD wallet workflow is different from HD wallet workflow because, in an HD wallet, keys can be generated predictably with a known mnemonic phrase, but there is no relation among the generated keys in non-HD wallet. So, we look for key patterns in the internal files in the non-HD wallet workflow. For

[10] https://www.tcpdump.org/.
[11] https://scapy.net/.
[12] https://github.com/seatgeek/fuzzywuzzy.

instance, Bitcoin public addresses start with the character 1 or 3 [44], are 34 characters long, and formatted as Base58; Ethereum public addresses start with 0x and are 42 characters long. We consider different key formats as well. For example, Bitcoin private keys can be in standard 256-bit hex format (64 bytes long), or WIF format [33] (51 bytes long) and start with 5. The pre-requisite steps of performing dynamic analysis on HD wallets are applicable for non-HD wallets except importing the wallet. Instead of importing the wallet, the keys are generated using the app itself. Different apps of course generate different keys; however, the key format is identical. We incorporate a pattern matching regex to look for Bitcoin public/private addresses and their various derivatives [21] in the app's internal files.

Transactions Workflow. For both HD and non-HD wallets, we make a small transaction [13] of a fixed amount to a pre-defined address. Horus looks for the pre-defined receiver's address and the fixed amount value in the wallet app's internal files. After a transaction is completed, the transaction history should not be saved in the device. If the receiver's address and the transaction value are present in the app's internal files, it stores past transactions and can be abused for deanonymization if the app's internal files are exposed.

To emulate wallet transactions, we use Bitcoin testnet [45]. Since not all wallet apps support testnet transactions, we make transactions only testnet compatible wallet apps. To collect testnet coins, we use coinfaucet[13] and mempool,[14] two freely available services to distribute testnet coins.

Manual Inspection. For in-depth analysis of the top 17 wallet apps, we monitor the apps' workflow and response based on the app's interaction. Our goal is to understand the app's workflow and its process to generate and store key revealing information. We observe changes in the app's internal files (e.g., Shared Preferences, Databases, File IO) based on the activity we perform using the app.

An Android app consists of 4 components: Activity, Service, Broadcast Receiver, and Content Provider. Android enforces a sandbox mechanism to protect the components, where no app gains access to other app's components by default. However, an app can export its components and let other apps access the components. If a component, such as a service, is exported and not protected with permissions, then any app can start and bind to the service. Any app on the device can invoke all the exported components in the target app. We manually verify if it is possible to send crafted intent from any other app to activate the exported components in the target app and make it perform the malicious task.

We use two state-of-the-art tools, Drozer[15] and Frida.[16] We use Drozer to list all exported components, universally accessible URIs using which any other app can ask for key revealing information from the target app and SQL Injection attack surface in the app. We use Frida to monitor critical operations in the app,

[13] https://coinfaucet.eu/en/btc-testnet/.
[14] https://testnet-faucet.mempool.co/.
[15] https://github.com/fsecurelabs/drozer/.
[16] https://frida.re/.

e.g., database operations, file IO, and method trace to find out passing arguments and the return value of a method. We also use Frida to trace app logs, bypass SSL pinning, and bypass root detection. We use an additional wrapper tool, House[17] over Frida for ease of use. Both Drozer and Frida are used for monitoring and intercepting app workflow.

5 Experimental Results

We use the LDPlayer[18] emulator, running Android 7.1.2 for all automated analysis, and Alcatel 5041C, running Android 8.1.0 for manual experiments. We use LDPlayer for its faster execution. The two different Android versions we use to cover a large part of contemporary Android phone users. Dynamic analysis requires a rooted device and we root the device using Magisk.[19] We analyze 310 wallet apps using Horus and manually inspect the top 17 most popular apps. In this section, we present some of our main findings and corresponding security risks.

Storing Key Revealing Information: Our dynamic analysis identifies 239 apps (77%) as non-HD wallets, 71 apps (23%) as HD wallets. In total, 111 apps (87 non-HD wallets and 24 HD wallets) store key revealing information in plaintext. In 47/71 HD wallet apps use encryption to store key revealing information; however, 18 of those apps store the encryption key without additional protections. In most cases, where we found an encryption key, the key is located in the SharedPreference file. In other cases, the key is located in a readable internal file, or it is a user-provided PIN (e.g., 4–6 digits). Among the top 17 apps, 3 apps store key revealing information in plaintext, and 4 apps store key revealing information encrypted with a known encryption key from our list of *candidate encryption key*. The best-in-class storage solution in Android is HSM, but only 3/310 apps are using HSM. If HSM is not available, then Android Keystore should be used, which provides the best available solution provided the Android OS itself is not compromised. The user-provided PIN should not be used as the encryption key because it is brute-forceable. 11 of 310 apps store transaction information on the device, leading to deanonymization if the internal files are exposed.

User Dictionary: On all Android devices, the keyboard app uses a user dictionary (database of words, locale, and frequency count) for predictive text inputs. In wallet apps, the mnemonic phrase contains common English words, and most wallet apps take mnemonic phrase input from the user using the default keyboard. The information regarding the words in mnemonic phrase is saved in user dictionary [18]. This dictionary can be abused to predict the mnemonic phrase [28] by extracting frequency information of typed words. Note that the attacker app requires virtual keyboard permission to access the dictionary, and in general, input editor and spellchecker apps ask for this permission. Multiple

[17] https://github.com/nccgroup/house.
[18] https://www.ldplayer.net/.
[19] https://github.com/topjohnwu/Magisk.

apps can have virtual keyboard permission on a device. Only 20 apps out of 310 apps (6%) implement custom keyboards to defend against this attack.

Allow Backup: This is an attribute declared in the AndroidManifest.xml file, and it is true by default. It denotes the app data is backed up upon the app's uninstallation and is restored upon re-install [2]. When enabled, the app data can be backed up using the ADB (Android Debug Bridge) command. It enables an attacker to extract an app's internal files in a non-rooted device within few minutes of physical access. Open-source tools such as Android-Backup-Toolkit[20] can be used to extract the backup and gain access to internal files.

Dangerous Permissions: In the Android ecosystem, some permissions are considered dangerous [3] and require the user's explicit consent before being authorized. Most wallet apps require some common permissions for their functionality (e.g., WRITE_EXTERNAL_STORAGE, READ_EXTERNAL_STORAGE, GET_ACCOUNTS, CAMERA). However, some apps ask for certain privacy-sensitive permissions, such as contact list and record audio, which appear to be non-essential for the app; see Table 3 for such permissions. Each permission has a constant value associated with it, such as android.permission.RECORD_AUDIO for audio recording, which is used to check, request and verify permission from the Android SDK. The constant values of the permissions are stored as a static variable in Manifest.permission class [3]. The presence of the static variable or the associated constant value of a particular permission in the app's codebase conforms the usage of the permission.

Table 3. Unnecessary dangerous permissions usage. Last row provides the total number for all 310 apps. ✗ indicates the app declares the permission requirement in the AndroidManifest file; *Read Profile* allows an app to read the user's personal profile data; *Read Phone State* allows the app to access the device's phone features; *Get Tasks* allows the app to retrieve information about currently and recently running tasks; and *Request Install Packages* allows app to install additional packages on the device.

Application ID	Read contacts	Access coarse location	Access fine location	Read profile	Read phone state	Get tasks	Request install packages	Record audio
asia.coins.mobile	✗	✗	✗					
co.bitx.android.wallet	✗			✗				✗
co.mona.android	✗		✗					
com.binance.dev		✗	✗					
com.bitcoin.mwallet	✗	✗	✗					
com.breadwallet		✗	✗					
com.coinomi.wallet								
com.paxful.wallet					✗	✗		
com.polehin.android								
com.unocoin.unocoinwallet	✗	✗	✗					
com.wallet.crypto.trustapp							✗	
com.xapo	✗	✗	✗	✗	✗	✗		
de.schildbach.wallet								
com.mycelium.wallet		✗						
org.toshi								
piuk.blockchain.android								✗
zebpay.Application						✗		
Total	47	75	81	8	84	36	28	40

[20] https://sourceforge.net/projects/android-backup-toolkit/.

Root Exploitation: In the Android ecosystem, root exploitation is well-known [46]. There are legitimate Android apps available in the Google Play Store that facilitate rooting of phones, referred to as root providers or one-click root apps. In 2016, 85 million devices downloaded such root provider apps, and the devices are soft-rootable [22]. In wallet apps, we find 70 apps out of 310 are checking if the device is rooted before starting its operation; see Table 4. However, none of the apps terminates upon root detection; instead, the app displays a non-blocking alert and lets the user continue using the app.

Table 4. Static module analysis summary. Here, ✓ indicates the existence of security standard implementation in the app and a blank cell indicates the absence of it. Last row provides the total number for all 310 apps.

Application ID	Root detection	Integrity verify	Screenshot disabled	Bio-metric	Custom keyboard	Secure random	Hardware security module
asia.coins.mobile	✓		✓			✓	
co.bitx.android.wallet	✓			✓		✓	
co.mona.android			✓			✓	
com.binance.dev	✓		✓		✓	✓	
com.bitcoin.mwallet	✓	✓	✓		✓	✓	
com.breadwallet		✓			✓	✓	
com.coinomi.wallet	✓	✓				✓	✓
com.mycelium.wallet			✓		✓	✓	
com.paxful.wallet				✓		✓	
com.polehin.android			✓	✓		✓	
com.unocoin.unocoinwallet		✓				✓	
com.wallet.crypto.trustapp	✓		✓		✓	✓	
com.xapo	✓			✓	✓	✓	✓
de.schildbach.wallet						✓	
org.toshi			✓			✓	
piuk.blockchain.android	✓		✓		✓	✓	
zebpay.Application						✓	✓
Total	70	33	44	46	20	292	10

Strandhogg Attack: This attack [17] is applicable when a malicious app that targets a wallet app opens up before the wallet app. When the user taps on the wallet app, the malicious app opens up instead. The malicious app can mimic the wallet app's UI and ask for its PIN, mnemonic phrase, etc. No permission is required for the malicious app, and Android versions 8.0–9.0 are vulnerable to this attack.

Exported Components: Depending on the responsibilities of a component, it may leak information or perform unauthorized tasks. For example, if a content provider is exported, it can reveal sensitive database information to any other app on the device. We find each one of the top wallet apps exports from 3 to 25 components, expanding the attack surface on wallet apps.

6 Discussions

In this section, we discuss Horus's current capabilities and limitations. Horus's static module is based on the app call graph, enabling the framework to survive

the code obfuscation. Android obfuscation techniques work on the app code and not on the SDK APIs. We note that Android API calls remain unobfuscated in the call graph. Our dynamic module is based on the app's artifacts and network trace. The dynamic module does not have any dependency on the app platform and source code. The artifacts analysis technique is equally effective in apps developed using hybrid and cross-platform technologies. Overall, Horus can quickly assess the security standards followed by a crypto wallet app using the static module, and provide a deeper understanding of the app's sensitive data handling process using the dynamic module. However, static analysis tools suffer from obvious limitations. They can only determine whether specific APIs or syntax patterns are present in the source code but cannot indicate whether the implementation is error-free. Depending on the syntax pattern, Horus may indicate that an app has implemented a security feature, but in reality, the implementation may be flawed and may still contain serious bugs.

To make Horus fully automated, we evaluate tools like Monkey[21] and Droidbot[22] to perform signup, import wallet, and complete a transaction. However, in our evaluation, we find that the pseudo-random events that Monkey generates cannot accomplish a set of pre-defined tasks. It is possible to accomplish specific tasks using Droidbot, but it is not well-suited for a generalized workflow. We have to write a customized script for each app. Considering apps' versatility, writing a script for each app beats the purpose of an automated framework, and we settled for a semi-automated solution.

We identify some wallet apps that encrypt key revealing information, use salt with the encryption key. We find hard-coded salt value in the source code and also salt value printed in logs. In Horus we are not automating when salt is used in the encryption key.

Starting with Android 7.0 (API level 24), system-wide private certificates are not accepted in Android. Optionally, an app can explicitly choose to rely on a private certificate [14]. Also, as of Android 9.0 (API level 28), plaintext traffic is not allowed and cannot be configured as well [6]. To overcome this, we use mitmproxy[23] to monitor network traffic and place the mitmproxy certificate in the system certificate folder. Also, we use tcpdump to capture network requests in a pcap file to look for key revealing information in network communication.

7 Related Work

Several wallet app analyses have been carried out in recent years, which expose vulnerabilities and propose new defenses. Volety et al. [48] perform offline brute force dictionary attacks on the mnemonic phrase to gain access to two wallet apps. Guri et al. [23] infect a cold wallet with malicious code during the installation phase and get hold of the private keys. Further, Koerhuis et al. [30] conduct forensic analysis on two popular cryptocurrencies, Monero and Verge, in the

[21] https://developer.android.com/studio/test/monkey.
[22] https://github.com/honeynet/droidbot.
[23] https://mitmproxy.org/.

desktop environment. They analyze the host machine's volatile memory, network traffic, hard disks and find critical artifacts like seed and plaintext passphrase. A similar study is carried out on Bitcoin by Zollner et al. [51].

Several studies also look into the security of Android wallet apps. He et al. [26] demonstrate two attack scenarios by capturing sensitive information from device display using accessibility permissions and obtaining user input via USB debugging. This analysis is conducted on only two wallet apps (not among the top 50 wallets in the Google Play Store). Hu et al. [27] devise 3 proof-of-concept attacks targeting deanonymization, spamming, and violating P2P (peer-to-peer) protocol requirements of Bitcoin. Capturing clipboard values [31] also presents a significant risk to crypto wallet apps, e.g., when importing non-HD wallet keys from another app/device, or when copying mnemonic phrases. Haigh et al. [25] analyze the forensic artifacts of seven wallet apps, and develop a Trojan POC by repackaging a wallet app that can steal the users' passwords. Gangwal et al. [1] use machine learning to identify a wallet app by tracing a user's network activity.

UI deception attacks that include clickjacking, phishing, and activity hijacking [20] in Android, are generally applicable for any wallet app. Bergandano et al. [11] develop a hybrid analysis tool by following OWASP guidelines that analyzes vulnerabilities in varying categories of apps (e.g., wallet, food, social). The work lacks manual inspection; thus, the tool's conclusion is unverifiable and focuses on generic vulnerabilities instead of taking into account any specific nature of the apps.

Contrary to prior studies, we explicitly focused on crypto wallet apps and discovered several new attack surfaces specifically applicable to Android wallets.

8 Conclusion

With the massive growth of cryptocurrencies, the number of threat vectors against crypto wallet apps is also increasing. This puts millions of users at risk if security concerns are not adequately addressed in leading wallet apps. We introduced Horus, a semi-automated framework to analyze and detect security issues in crypto wallet apps. We analyzed 310 apps on the Android platform and discover a unique set of vulnerabilities. Our analysis indicates that security standards are not followed when developing apps, and there are vulnerabilities in the protection of key revealing information. Serious security gaps appear in popular wallet apps, including asking for dangerous permissions without a proper need. Based on our analysis, there is a lack of checks and balances in our understanding of wallet apps' security and their actual implementation. Users should be more vigilant and proactive in evaluating apps before relying on them. Additionally, developers should be better informed about the industry's security standards and strictly adhere to the best practices and recommendations.

References

1. Aiolli, F., Conti, M., Gangwal, A., Polato, M.: Mind your wallet's privacy: identifying Bitcoin wallet apps and user's actions through network traffic analysis. In: 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1484–1491 (2019)
2. Android Developers: Auto backup data. https://developer.android.com/guide/topics/data/autobackup
3. Android Developers: Dangerous permissions. https://developer.android.com/reference/android/Manifest.permission
4. Android Developers: HSM. https://developer.android.com/training/articles/keystore
5. Android Developers: Input. https://developer.android.com/guide/topics/text/creating-input-method.html
6. Android Developers: Network security. https://developer.android.com/training/articles/security-config
7. Android Developers: SecureRandom. https://developer.android.com/reference/java/security/SecureRandom
8. Android Developers: Shrink, obfuscate, and optimize your app. https://developer.android.com/studio/build/shrink-code#obfuscate
9. Antonopoulos, A.M.: Mastering Bitcoin. O'Reilly Media, Inc., March 2021
10. Bankhead, P.: Android developers blog: Improving discovery of apps and games on the Play Store. https://android-developers.googleblog.com/2018/06/improving-discovery-of-quality-apps-and.html
11. Bergadano, F., Boetti, M., Cogno, F., Costamagna, V., Leone, M., Evangelisti, M.: A modular framework for mobile security analysis. Inf. Secur. J. **29**(5), 220–243 (2020)
12. Bitcoin: BIPS: Bitcoin Improvement Proposals. https://github.com/bitcoin/bips
13. Bitcoin: Transactions. https://www.bitcoin.com/get-started/how-bitcoin-transactions-work
14. Brubaker, C.: Android developers blog: Changes to trusted certificate authorities in Android Nougat. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html
15. Cimpanu, C.: Bitcoin wallet update trick has netted criminals more than $22m — ZDNet. https://www.zdnet.com/article/bitcoin-wallet-trick-has-netted-criminals-more-than-22-million/
16. Cimpanu, C.: Hacker group has stolen more than $200m. https://www.zdnet.com/article/cryptocore-hacker-group-has-stolen-more-than-200m-from-cryptocurrency-exchanges/
17. NVD - CVE-2020-0096. https://nvd.nist.gov/vuln/detail/CVE-2020-0096
18. Diao, W., Liu, X., Zhou, Z., Zhang, K., Li, Z.: Mind-reading: Privacy attacks exploiting cross-app keyevent injections. In: European Symposium on Research in Computer Security, pp. 20–39. Springer (2015)
19. Druffel, A., Heid, K.: Davinci: Android app analysis beyond Frida via dynamic system call instrumentation. In: International Conference on Applied Cryptography and Network Security, pp. 473–489. Springer (2020)
20. Fernandes, E., et al.: Android UI deception revisited: Attacks and defenses. In: International Conference on Financial Cryptography and Data Security. pp. 41–59. Springer (2016)
21. ForensicFocus.com: Forensics and Bitcoin. https://www.forensicfocus.com/articles/forensics-bitcoin/

22. Gasparis, I., Qian, Z., Song, C., Krishnamurthy, S.V.: Detecting Android root exploits by learning from root providers. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 1129–1144 (2017)
23. Guri, M.: Beatcoin: leaking private keys from air-gapped cryptocurrency wallets. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 1308–1316. IEEE (2018)
24. Gutoski, G., Stebila, D.: Hierarchical deterministic Bitcoin wallets that tolerate key leakage. In: International Conference on Financial Cryptography and Data Security. pp. 497–504. Springer (2015)
25. Haigh, T., Breitinger, F., Baggili, I.: If I had a million cryptos: Cryptowallet application analysis and a trojan proof-of-concept. In: International Conference on Digital Forensics and Cyber Crime. pp. 45–65. Springer (2018)
26. He, D., Li, S., Li, C., Zhu, S., Chan, S., Min, W., Guizani, N.: Security analysis of cryptocurrency wallets in Android-based applications. IEEE Network **34**(6), 114–119 (2020)
27. Hu, Y., et al.: Security threats from Bitcoin wallet smartphone applications: Vulnerabilities, attacks, and countermeasures. In: Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY 2021), pp. 89–100 (2021)
28. Kachakil, D.: Discovering and exploiting a vulnerability in Android's personal dictionary. https://ioactive.com/discovering-and-exploiting-a-vulnerability-in-androids-personal-dictionary/
29. Kim, T., Ha, H., Choi, S., Jung, J., Chun, B.G.: Breaking ad-hoc runtime integrity protection mechanisms in Android financial apps. In: 2017 ACM on Asia Conference on Computer and Communications Security. pp. 179–192 (2017)
30. Koerhuis, W., Kechadi, T., Le-Khac, N.A.: Forensic analysis of privacy-oriented cryptocurrencies. Forensic Science International: Digital Investigation **33**, 200891 (2020)
31. Li, C., He, D., Li, S., Zhu, S., Chan, S., Cheng, Y.: Android-based cryptocurrency wallets: Attacks and countermeasures. In: 2020 IEEE International Conference on Blockchain (Blockchain). pp. 9–16. IEEE (2020)
32. Ling, Z., et al.: Privacy enhancing keyboard: Design, implementation, and usability testing. Wireless Communications and Mobile Computing 2017 (2017)
33. Maksim, B.: What is wallet import format (wif)? https://allprivatekeys.com/what-is-wif
34. Neal, W.: Cryptocurrency hackers steal $3.8B in 2020. https://www.occrp.org/en/daily/13627-cryptocurrency-hackers-steal-3-8-billion-in-2020
35. Newburger, E.: Bitcoin surpasses $60,000 in record high as rally accelerates. https://www.cnbc.com/2021/03/13/bitcoin-surpasses-60000-in-record-high-as-rally-accelerates-.html
36. Online, FE: Market Cap $1.24T, February 2021. https://www.financialexpress.com/market/bitcoin-rally-takes-market-cap-of-over-6000-cryptocurrencies-to-whopping-new-high-of-1-24-trillion/2189730/
37. Osborne, C.: AT&T dragged to court, over SIM hijacking and cryptocurrency theft — ZDNet. https://www.zdnet.com/article/at-t-dragged-to-court-again-over-sim-hijacking-and-cryptocurrency-theft/
38. Palatinus, M., Rusnak, P.: Official specification of BIP-0044, March 2019. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki
39. Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: Official specification of BIP-0039, February 2021. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

40. Powers, B.: This elusive malware has targeted crypto wallets for a year, January 2021. https://www.coindesk.com/elusive-malware-electrorat-targets-crypto-wallets
41. Rahman, M.: Developers are facing huge drop in new installs after Play Store algorithm changes. https://www.xda-developers.com/developers-huge-drop-new-installs-play-store-algorithm-changes/
42. Ranganath, V.-P., Mitra, J.: Are free Android app security analysis tools effective in detecting known vulnerabilities? Empir. Softw. Eng. **25**(1), 178–219 (2019). https://doi.org/10.1007/s10664-019-09749-y
43. Redman, J.: The $700m wallet crack. https://news.bitcoin.com/the-700-million-wallet-crack-bitcoins-7th-largest-address-is-under-constant-attack/
44. Sedgwick, K.: Bitcoin address formats - Wallets Bitcoin News. https://news.bitcoin.com/everything-you-should-know-about-bitcoin-address-formats/
45. Testnet - Bitcoin Wiki. https://en.bitcoin.it/wiki/Testnet
46. Unuchek, R.: Android: To root or not to root — Kaspersky official blog. https://www.kaspersky.com/blog/android-root-faq/17135/
47. /dev/random - Wikipedia. https://en.wikipedia.org/wiki//dev/random
48. Volety, T., Saini, S., McGhin, T., Liu, C.Z., Choo, K.K.R.: Cracking bitcoin wallets: I want what you have in the wallets. Futur. Gener. Comput. Syst. **91**, 136–143 (2019)
49. Wuille, P.: Official specification of BIP-0032, August 2020. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
50. Zhang, H., She, D., Qian, Z.: Android root and its providers: a double-edged sword. In: 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1093–1104 (2015)
51. Zollner, S., Choo, K.K.R., Le-Khac, N.A.: An automated live forensic and post-mortem analysis tool for Bitcoin on Windows systems. IEEE Access **7**, 158250–158263 (2019)