Automated Software Engineering (2024) 31:32
https://doi.org/10.1007/s10515-024-00430-3

# WalletRadar: towards automating the detection of vulnerabilities in browser-based cryptocurrency wallets

**Pengcheng Xia[1] · Yanhui Guo[1] · Zhaowen Lin[1] · Jun Wu[1] · Pengbo Duan[1] · Ningyu He[2] · Kailong Wang[3] · Tianming Liu[4] · Yinliang Yue[5] · Guoai Xu[6] · Haoyu Wang[3]**

Received: 29 December 2023 / Accepted: 3 March 2024 / Published online: 31 March 2024


## Abstract

Cryptocurrency wallets, acting as fundamental infrastructure to the blockchain ecosystem, have seen significant user growth, particularly among browser-based wallets (i.e., browser extensions). However, this expansion accompanies security challenges, making these wallets prime targets for malicious activities. Despite a substantial user base, there is not only a significant gap in comprehensive security analysis but also a pressing need for specialized tools that can aid developers in reducing vulnerabilities during the development process. To fill the void, we present a comprehensive security analysis of browser-based wallets in this paper, along with the development of an automated tool designed for this purpose. We first compile a taxonomy of security vulnerabilities resident in cryptocurrency wallets by harvesting historical security reports. Based on this, we design WalletRadar, an automated detection framework that can accurately identify security issues based on static and dynamic analysis. Evaluation of 96 popular browser-based wallets shows WalletRadar's effectiveness, by successfully automating the detection process in 90% of these wallets with high precision. This evaluation has led to the discovery of 116 security vulnerabilities corresponding to 70 wallets. By the time of this paper, we have received confirmations of 10 vulnerabilities from 8 wallet developers, with over $2,000 bug bounties. Further, we observed that 12 wallet developers have silently fixed 16 vulnerabilities after our Conflict of interest. WalletRadar can effectively automate the identification of security risks in cryptocurrency wallets, thereby enhancing software development quality and safety in the blockchain ecosystem.

**Keywords** Cryptocurrency · Non-custodial wallets · Browser extensions · Automated security analysis · Vulnerability detection · Data leakage

Pengcheng Xia and Yanhui Guo have contributed equally to this work.

Extended author information available on the last page of the article

## 1 Introduction

Cryptocurrencies have captured the attention of a large number of investors in recent years due to their potential economic value. According to the report Crypto Market Sizing Report H1 (2023), the number of cryptocurrency owners has crossed the 500 million milestone by the first half of 2023. Such a substantial user base has made the cryptocurrency market highly active. As the number of novice investors swells, there is a growing need for user-friendly software to help them engage with the blockchain, leading to the emergence of *cryptocurrency wallets*. Based on whether the credentials are stored with a centralized third party or in the hands of users, wallets can be classified as custodial and non-custodial, respectively. The recent collapse of FTX FTX to start U.S. bankruptcy proceedings, CEO to exit (2022), a provider of custodial wallet services, has highlighted the risks of centralized control, leading to a surge in interest in non-custodial wallets, which offer full control and enhanced security of digital assets. In particular, browser-based non-custodial cryptocurrency wallets are gaining traction due to their immediate accessibility and straightforward interface. For instance, browser-based wallets like Metamask MetaMask (2023), Phantom Phantom (2023), and Coinbase Coinbase Wallet (2023) have achieved significant success, with over one million downloads. These wallets function as browser extensions and facilitate easy account creation with access to a comprehensive range of blockchain functionalities.

*Tall trees catch much wind.* Although users can fully control their assets with non-custodial wallets, the responsibility of securely managing credentials also falls on them. This heightened responsibility comes with its own set of challenges, as the security landscape of non-custodial wallets is constantly evolving. In recent years, some incidents have revealed vulnerabilities in even the most reputed non-custodial wallets Slope Wallet Incident Update (2022); Check Point Research Detects (2022); CPR (2022); Security Threat Exposed (2022); Trinity Attack Incident Part 1 (2022). These incidents have disrupted the notion of non-custodial wallets as fully secure, often resulting in financial losses for users. For example, the Slope Wallet incident led to the leakage of 9,231 wallets' private keys and the loss of about $4.1 million due to a vulnerability in the wallet's handling of sensitive information Slope Wallet Incident Update (2022). Besides, several malicious software specifically targets browser-based wallets Hodlers beware! (2022); Double trouble (2023); LummaC2 Stealer (2023). They typically exploit vulnerabilities in wallets to access user credentials, aiming to steal their cryptocurrencies. For instance, by accessing the vulnerable local storage of browser-based wallets, the LummaC2 Stealer was able to steal sensitive information from over 60 wallets on 10 browsers including Chrome and Firefox.

Thus, it is urgent to identify the vulnerabilities of non-custodial wallets and prevent the attacks that exploit them. Indeed, several vulnerability detection tools have been developed by the research community. However, they primarily focus on the security analysis of mobile applications. For instance, Li et al. Li et al.

(2020) explored the attack surface of Android cryptocurrency wallets and found that due to flaws in Android system design and careless development, security issues could expose users' private keys and phrases, risking the financial safety of millions. Uddin et al. Uddin et al. (2021) developed a semi-automated framework for assessing the security of Android cryptocurrency wallet apps, revealing critical vulnerabilities in key storage and transaction privacy across numerous applications.

To the best of our knowledge, the vulnerabilities in browser-based wallets have not been systematically investigated and there is also a lack of automated tools for detecting these vulnerabilities. There are still some questions that the blockchain community is unaware of. Firstly, *which are the major vulnerabilities affecting these wallets?* Since blockchain techniques and their corresponding wallets are developing rapidly, the vulnerabilities are also continuously evolving, making it difficult to create a comprehensive and up-to-date list. Secondly, *how to reliably and automatically detect these vulnerabilities?* Vulnerabilities common in traditional web applications may manifest differently in browser-based wallets, which could diminish the effectiveness of existing detection methods. Moreover, no automated tools are available to detect newly emerging vulnerabilities dedicated to browser-based wallets. Lastly, *to what extent do these vulnerabilities exist?* While there are isolated reports of vulnerabilities, a comprehensive understanding remains unclear regarding the characteristics of these vulnerabilities, the security level of these wallets, and the security awareness among their developers.

**This work.** We take the first step to characterize and detect vulnerabilities in browser-based wallets. By summarizing security reports of security companies and bulletins of wallet providers, we first create a taxonomy of 6 types of vulnerabilities in browser-based wallets (see **Section** 3.1), including traditional web vulnerabilities that also appear in traditional websites but often have a new form of manifestation and new emerging vulnerabilities targeted at cryptocurrency wallets that tend to have more severe security impacts. To detect these vulnerabilities, we propose a hybrid approach that combines static and dynamic analysis on browser extensions to accurately and automatically detect vulnerabilities in browser-based wallets (see **Section** 4). The evaluation of 96 popular browser-based wallets shows that our framework can operate on 90% of the wallets automatically with high accuracy (see **Section** 5). During this evaluation, 70 wallets (73% of all wallets) were found to have 116 vulnerabilities. These findings underscore a concerning trend: numerous developers of these wallets overlook crucial security mechanisms, such as password policy and credential storage, among others. At last, our impact analysis suggests that these vulnerabilities may influence more than 9.2 million wallet users. However, most wallet developers still have not fixed these issues and do not give enough attention to them. Through our vulnerability Conflict of interest, we assisted 20 different wallets in fixing a total of 26 vulnerabilities. This effort was acknowledged by 8 wallet developers and resulted in $2,000 in bug bounties.

In summary, we make the following main research contributions in this paper:

- *We take the first step to create a taxonomy of vulnerabilities in browser-based cryptocurrency wallets.* Through a comprehensive survey of existing security

reports and a detailed analysis of popular applications, a taxonomy has been created for 6 types of browser-based wallet vulnerabilities, including traditional ones (i.e., clickjacking, cross-site scripting, defective password policy) and new emerging ones (i.e., demonic vulnerability, redundant storage, defective cryptography).

- *An automated vulnerability detection framework is implemented to identify vulnerabilities in browser-based cryptocurrency wallets.* We developed WalletRadar, a framework that combines static and dynamic analysis for accurately identifying vulnerabilities in browser-based wallets. The evaluation shows that the framework can be automated on more than 90% of the wallets and achieves a high accuracy.
- *We systematically characterize vulnerabilities in browser-based cryptocurrency wallets.* This work has revealed that the vulnerabilities are prevalent in browser-based wallets and the developers lack attention to them. WalletRadar has found that 70 out of 96 tested browser-based wallets are vulnerable. The subsequent impact analysis reveals that more than 9.2 million users face the risk of information leakage and financial loss due to these vulnerabilities. We have received confirmations of 10 vulnerabilities from 8 wallet developers, with over $2,000 bug bounties. Further, we observed that 12 wallet developers have silently fixed 16 vulnerabilities after our Conflict of interest.

## 2 Background

### 2.1 Cryptocurrency wallets

Cryptocurrencies, originally developed as a component of blockchain's reward mechanism, play a critical role within the blockchain system. The first cryptocurrency Bitcoin Bitcoin (2022) was released in 2009, and to date, there are over 23 thousand cryptocurrencies worldwide CoinMarketCap (2023). Following the surge in cryptocurrency popularity in 2017 From (2017), individuals have flooded into cryptocurrency markets to acquire or trade cryptocurrencies. For most people without technical experience, using a cryptocurrency wallet is essential to perform transactions on a blockchain platform. This wallet, which can be either a software or hardware tool, facilitates the storing and trading of cryptocurrencies by interacting with blockchain ledgers. However, unlike conventional wallets that physically store fiat currency, cryptocurrency wallets do not directly store digital assets. Since cryptocurrencies inherently exist as transaction data within the blockchain ledgers, the wallet validates the user's cryptocurrency holdings by retrieving the user's transaction information corresponding to their unique addresses (i.e., blockchain accounts).

According to how keys are stored, cryptocurrency wallets can be generally classified into two categories: (*i*) *custodial wallets*, which depend on a centralized third party for key storage, and (*ii*) *non-custodial wallets*, which store keys locally. Non-custodial wallets have been gaining popularity due to their enhanced personal

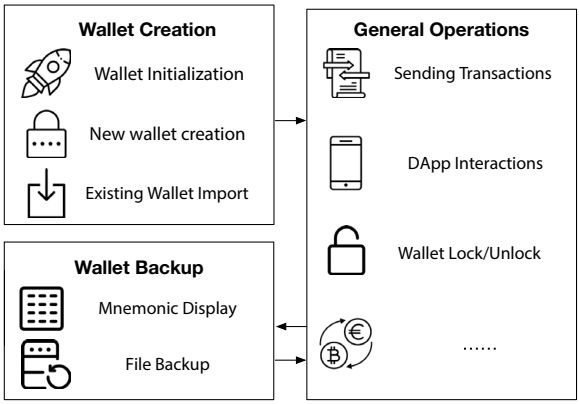



**Fig. 1** The general workflow of the browser-based cryptocurrency wallets

security and direct ownership of assets. They also align with the decentralization principle that is at the heart of the cryptocurrency paradigm.

Non-custodial wallets include browser-based wallets, desktop wallets, and mobile wallets. Existing research Sai et al. (2019); Hu et al. (2021) has revealed a range of security issues associated with mobile wallets and their operating environments. Browser-based wallets, despite their considerable user base owing to their online accessibility, lack systematic security analysis. One such browser-based wallet, MetaMask, has an impressive user base exceeding 10 million. The wallet application can be downloaded from the browser extension store and operated locally, with the user data also stored in the local browser, providing users with full control. However, browser-based non-custodial wallets including MetaMask are not exempt from security threats, and vulnerability-related incidents are frequently reported Slope Wallet Incident Update (2022); CPR (2022); Security Threat Exposed (2022). Therefore, our research primarily focuses on such wallets, aiming to unearth the distinct security issues prevalent in this class of wallets, create a taxonomy derived from these vulnerabilities, and develop methodologies for their identification.

### 2.2 General workflow of a browser-based wallet

As shown in Fig. 1, the main function of a browser-based wallet usually includes wallet creation, wallet backup, and other general wallet operations.

(*i*) **Wallet Creation**: After the wallet's initial launch and traversing the start page, users can opt either to create a brand-new wallet or import an existing one. When creating a new wallet, users need to set a password, which is used to unlock the wallet. Then the wallet will generate a pair of keys, i.e., a private key and a public key, and further output the wallet address based on the public key. In addition, the wallet will automatically generate a string of words for the user (called "mnemonics") to recover the wallet. As for integrating an existing wallet, users should provide the original private key or mnemonics and set the wallet password to complete the wallet import.

(*ii*) **Wallet Backup**: In the wallet backup process, users are firstly required to provide a password to pass the identity authentication. Following this, the wallet will display the mnemonic or private key in plaintext for users to copy and back up elsewhere. Additionally, a few wallets offer methods to back up these sensitive data to files. In such cases, the wallet might require users to provide another temporary password to encrypt the backup file, thereby ensuring its security.

(*iii*) **General operations**: Wallets may be involved in other general operations, like transaction initiation, and decentralized applications (DApps) interactions. In these operations, wallets serve as a medium to help users communicate with the blockchain in terms of transaction activities. For security reasons, when it comes to sensitive operations, e.g., wallet backup, the wallet mandates a password-based user authentication, i.e., users are required to give the password to unlock and perform these sensitive operations.

## 3 Vulnerability taxonomy

### 3.1 Generating the taxonomy

To understand and identify vulnerabilities that are inherent to browser-based cryptocurrency wallets, we first propose a taxonomy. Specifically, to conduct a comprehensive analysis, we focus on behaviors in each stage of the lifecycle of wallets, as shown in Fig. 1. In other words, we concentrate on sensitive operations at wallet creation, backup, and unlock. Note that, since the security of blockchain interactions, like transaction processing, primarily lies on the blockchain side, this work does not consider the threats of these activities.

We aim to divide the vulnerabilities into the following two aspects: (*i*) *Traditional web vulnerabilities*. Given that web extensions fundamentally operate on web pages, gathering traditional web vulnerabilities and considering their applicability is required. This encompasses a range of vulnerabilities like injection, XSS, access control flaws, and clickjacking. (*ii*) *New emerging cryptocurrency wallet vulnerabilities*. Even though the scope of the first type is extensive, vulnerabilities that may have a limited impact on traditional web applications could pose a more serious threat in browser-based wallets. Therefore, inspired by existing literature on mobile wallets He et al. (2020); Li et al. (2020); Uddin et al. (2021), this category consists of some specific vulnerabilities like sensitive data management, flaws in the storage process, and improper use of cryptographic methods, which could cause sensitive data leakage or even financial loss in browser-based cryptocurrency wallets.

Thus, in addition to sources directly related to browser-wallet security, we also considered literature on general web security to ensure our taxonomy's robustness. This approach allowed us to include insights from both blockchain-specific vulnerabilities and those common in broader web applications, providing a comprehensive view of the security landscape for browser-based wallets. In specific, we compile a list of the security vulnerabilities in browser-based wallets from related research Hu et al. (2021); Sai et al. (2019), best-industrial practice guidelines Crypto

**Table 1** The vulnerabilities of browser-based cryptocurrency wallets

| Category | Description |
|---|---|
| Clickjacking | Overlaying phishing pages to trick users |
| XSS | Injecting harmful scripts into pages |
| Defective password policy | Permitting weak, easily cracked passwords |
| Redundant storage | Unnecessary storage of sensitive data |
| Demonic vulnerability | Insecure caching of sensitive keys |
| Defective cryptography | Use of weak cryptographic methods |

Wallet Security (2022); A Guide to Wallet (2022); OWASP Top 10 (2022) and security reports MetaMask Security Monthly (2022); The Wild Crypto World in 2022 (2022); Year 2022 in Review (2023). We systematically reviewed each source to gather information on vulnerabilities specific to browser-based wallets. This involved assessing the characteristics and impacts of each vulnerability to decide its relevance to browser-based wallets. For accuracy and to ensure no detail was overlooked, two authors independently checked the classifications. The vulnerabilities are finally divided into two categories we mentioned above, as shown in Table 1.

## 3.2 Traditional web vulnerabilities

### 3.2.1 Clickjacking

Clickjacking is a vulnerability that employs visual deception Calzavara et al. (2020); Huang et al. (2012); Wu et al. (2016). In the context of cryptocurrency wallets, attackers overlay a transparent wallet homepage, extracted from the target browser-based wallet, on a phishing webpage carefully crafted by them. When careless users interact with the deceptive page, their wallets will be manipulated. This can lead to unauthorized fund transfers or sensitive data leakage. In contrast to the traditional clickjacking defenses, which incorporate headers like Content Security Policy (CSP) and X-Frame-Options to restrict browsers from rendering embedded pages in responses, browser extensions rely on their configuration files to control access from external websites. Thus, if the wallet's main HTML page is listed under the "web_accessible_resources" configuration in the "manifest.json" file, external pages can access the main page, potentially introducing clickjacking vulnerability.

### 3.2.2 Cross-site scripting (XSS)

XSS attacks exploit web functions that render dynamic content Gupta and Gupta (2017); Kumar et al. (2022a, 2022b). Attackers usually insert malicious JavaScript code into HTML pages to manipulate the rendered content. When it comes to browser-based cryptocurrency wallets, there are two notable impacts, i.e., *automatic page manipulation* and *unauthorized access to local storage* (such as localStorage

and indexedDB). While the former can lead to unauthorized fund transfer, the latter could result in the leakage of encrypted data.

Web extensions usually provide users with dynamic notifications, automatic redirection, and other features through the HTML Document Object Model (DOM). After the HTML DOM is loaded, cryptocurrency wallets often alter the DOM to specific page contents. If these alterations involve sensitive functions, they may result in DOM-based XSS vulnerabilities.

#### 3.2.3 Defective password policy

Requiring users to set a complex password is crucial. In browser-based cryptocurrency wallets, all the information in users' wallets is encrypted and stored locally. If a weak password is adopted, attackers may attempt brute force attacks, causing users to lose control of their wallets. A CheckPoint report Check Point Research Detects (2022) points out that when attackers obtain locally stored cryptographic wallet information through malicious means, they can brute-forcedly try 95 passwords per second on a 4-core Intel Core i7 CPU, which is sufficient to exploit a weak 6-digit password.

### 3.3 New emerging cryptocurrency wallet vulnerabilities

#### 3.3.1 Redundant storage

Redundant storage vulnerabilities arise when wallets store sensitive information or related intermediary processing results in local storage. This can significantly lower the barrier for attackers to access sensitive information. It usually occurs in the locking or unlocking stage of cryptocurrency wallets. If wallets store intermediate decryption results in the browser, attackers who can exploit this data (e.g., through an XSS attack) can reverse-engineer and reproduce the decryption sequence. To fully understand this vulnerability, consider a typical wallet unlocking process as an example.

(*i*) **Data Retrieval**. The wallet receives the password entered by the user and retrieves the locally stored encrypted wallet data.

(*ii*) **Key Generation**. Using passwords directly for wallet authentication can make the system vulnerable to brute-force and rainbow table attacks, as it heavily relies on password strength. A common practice for authentication is to create a decryption key from the password using methods with hash iterations (called Password-Based Key Derivation Function, PBKDF), which adds an extra layer of complexity and security Turan et al. (2010); Visconti et al. (2019). To generate a decryption key, the wallet either uses the user's raw input for hashing iterations or initially hashes the user's input, then employs the obtained password hash to further hashing iterations (with another hashing method).

(*iii*) **Data Decryption**. With the generated key, the wallet tries to decrypt the encrypted data. If it succeeds and unveils the plain wallet data, the user is navigated to other pages for the following operations.

During our observations, we found that some wallets would carelessly store sensitive data, such as the hash calculated from the password (step *ii*), in local storage even after unlocking. Note that the implementation of the unlocking process is transparent, which can be easily obtained by auditing the front-end JavaScript files. Thus, after obtaining this hash, they just need to reproduce the iterative hash process to generate the decryption key (step *ii*) and then use the key to unlock the wallet (step *iii*). The decrypted wallet data usually contains mnemonics or private keys, which gives attackers full control of the wallet.

#### 3.3.2 Demonic vulnerability

Demonic vulnerability is another storage-related vulnerability. Unlike the redundant storage vulnerability, which results from the inherent wallet design, the demonic vulnerability is rooted in browser caching mechanisms, leading to the unintentional local caching of sensitive data. According to the report Security Notice (2022), the early versions of Metamask held mnemonics in the HTML "textarea" tag when importing wallets. As an inherent browser mechanism, browsers are designed to cache textual data from active tabs to preserve the current state of the page, allowing for faster access and retrieval later. Thus, such sensitive information would be saved to the local disk due to the caching mechanism. Given the importance of mnemonics in the aspect of cryptocurrency wallets, this is a significant security concern. Numerous wallets in the market employ similar implementations to display sensitive data. As these implementations can be found in various functions, including wallet imports, wallet creation, mnemonic display, and private key display, the potential impact of this kind of vulnerability is extensive.

#### 3.3.3 Defective cryptography

Cryptographic algorithms, central to the functionality of browser-based wallets, are pivotal for sensitive operations like wallet creation and identity authentication, as highlighted in §3.1. This reliance on cryptography underlines the importance of their proper implementation in securing wallet data, which is the reason we put it in the "new emerging cryptocurrency wallet" category. In creating our taxonomy, we noticed a knowledge gap among some wallet developers regarding the optimal adoption of these algorithms, leading to significant cryptography-related issues.

One common issue is the insufficient iterations of the PBKDF2 algorithm (mentioned in §3.3.1). We observed wallets employing as few as 100 or 5, 000 iterations, falling short of the recommended 10, 000 and optimal 310, 000 rounds. This inadequate iteration count makes wallets more vulnerable to brute-force attacks, a risk that escalates with weak passwords. Additionally, the choice of encryption patterns is critical. The use of AES-CBC mode, for example, poses risks due to its lack of integrity checks. A more secure alternative like AES-GCM, which offers both confidentiality and integrity, would be preferable.

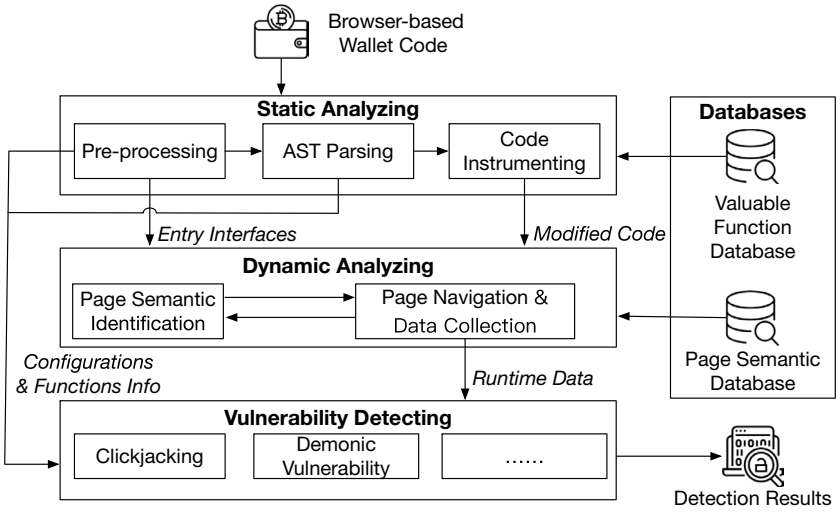



**Fig. 2** The detection workflow for WalletRadar

## 4 The design of WalletRadar

In this section, we present WalletRadar, an automated vulnerability detection framework specifically designed for identifying vulnerabilities in browser-based wallets. According to our taxonomy, we aim to detect all six kinds of vulnerabilities.

We seek to design a hybrid approach that combines static and dynamic analysis. On the one hand, clickjacking, XSS vulnerabilities, and defective cryptography in browser-based wallets can be easily identified through an efficient static analysis. For example, identifying defective cryptography can be done via filtering the signatures of adopted cryptography algorithms. Due to the simplicity of identifying these vulnerabilities, the inherent false positive issue can be reduced as much as possible. In contrast, static analysis is not suitable for detecting the other three vulnerabilities. Since requirements and implementations for password policies vary across wallets, conducting dynamic testing on password input is more appropriate. Also, demonic vulnerabilities and redundant storage vulnerabilities require monitoring the dynamically changing webpage information and data in local storage. Therefore, we propose another set of dynamic analysis methods to identify these vulnerabilities.

### 4.1 Approach overview

Figure 2 provides a general overview of WalletRadar. The framework includes three core phases: (*i*) static analyzing, (*ii*) dynamic analyzing, and (*iii*) vulnerability detecting.

Specifically, WalletRadar takes the code base of a to-be-verified wallet as input. In the static analyzing phase, WalletRadar firstly performs necessary pre-processings. It not only deobfuscates and reformats the given wallet, but also tries to extract entry interfaces of it. Based on the beautified and readable source

code, WalletRadar parses the corresponding Abstract Syntax Trees (ASTs). With the help of the *valuable function database*, WalletRadar can filter the ASTs of suspicious functions out. Meanwhile, WalletRadar collects static features, like the locations and some hard-coded parameters of target functions. To obtain the runtime data, WalletRadar has to conduct the necessary instrumentation before each function invocation. In the dynamic analyzing phase, based on the entry interfaces and the instrumented source code, WalletRadar dynamically deploys the wallet in our local testing environment. According to the *page semantic database*, WalletRadar can simulate the user interactions with pages, and collects the runtime data (like local storage). Finally, in the vulnerability detecting phase, according to the data collected from the above two phases, WalletRadar can efficiently and effectively identify all six types of vulnerabilities.

### 4.2 Static analyzing

The static analyzing phase is responsible for collecting static features and conducting instrumentation for the following dynamic analyzing phase. It consists of three stages, i.e., *pre-processing*, *AST parsing*, and *code instrumenting*.

#### 4.2.1 Pre-processing

As we stated in §4.1, pre-processing is mainly responsible for two things: *beautifying*, and *static features extracting*. Thus, WalletRadar firstly tries to deobfuscate and reformat the source code by js-beauty beautify-web (2022), a well-known and widely adopted formatting tool. Then, to extract features, WalletRadar parses the configuration files (e.g., "manifest.json") of the wallets. To be specific, we focus on the fields related to entry interfaces, like the "background" field, which defines scripts or pages that run persistently in the background; "action" defines the pages displayed when users open the wallets; "content_scripts" defines scripts that are injected and executed on web pages supported by the wallets; and "web_accessible_resources" defines resources within the wallet that can be accessed by other web pages. The configuration items will be directly sent to the vulnerability detecting phase for further analysis. The extracted possible entry interfaces will be sent to the dynamic analyzing phase.

#### 4.2.2 AST parsing

Based on the implementation of the wallet, AST parsing will filter suspicious functions out and parse them into AST format. Specifically, WalletRadar invokes Esprima Esprima (2022), a widely-used JavaScript parser, to obtain the AST for all functions. Then, according to the *valuable function database*, which is collected from related research Hu et al. (2021); Sai et al. (2019) and reports Year 2022 in Review (2023); MetaMask (2023), WalletRadar can identify potentially vulnerable APIs as well as their corresponding AST through simple but effective regex matching. Two types of APIs in the valuable function database are concerned,

```
function UnlockExample(x, y, z) {
    function process(temp) {
        ...
    }
    function unlock(a, b) {
      // The code snippet being injected to collect the parameters
      //collect();
      var c = process(a);

      function unlocklog(d) {
        console.log(d);
      }
      const decrypted = CryptoJS.AES.decrypt(c, b);
    }
}
```

**Fig. 3** An example of a decryption function

i.e., *cryptographic algorithm functions*, and *DOM manipulating functions.* The cryptographic algorithm functions deal with sensitive data handling, like generating decryption keys through key derivation functions or decrypting wallet data (e.g., "AES.decrypt" and "crypto.subtle.deriveKey"). On the other hand, DOM manipulating functions are typically used for providing user notifications or facilitating automatic webpage redirections (e.g., "document.write" and "window.location.replace").

To identify corresponding AST of these APIs, for cryptographic algorithm functions, we need to find the accurate locations of them and extract hard-coded parameters in them. Thus, an extended forward search is initiated to start from the outermost function of the current file, targeting the matched function, and to record the search path encountered during this process. If certain cryptographic algorithm parameters are hard-coded in the code, they are recorded during this step. This step is essential as it can be challenging to obtain these parameters through dynamic code instrumentation, which is primarily intended for capturing dynamically passed variables within functions. This approach ensures that no critical information of related functions is overlooked, thereby enhancing the accuracy of our analysis. As for DOM manipulating functions, to check whether their data sources can be modified, WalletRadar first needs to find their data source functions. Thus, a taint backtracking is performed from the matched function to the data source function on the AST of the source code. Similarly, WalletRadar will record the search paths for subsequent vulnerability detection.

### 4.2.3 Code instrumenting

Code instrumentation is for better collecting runtime data in the following dynamic analyzing phase. Therefore, as we mentioned in Sect. 4.2.2, two types of functions are kept: *cryptographic algorithm related* and *DOM manipulating related*. Since we are concerned with parameters that might be dynamically passed into cryptographic functions, such as keys and ciphertext, and the data flow of DOM functions has already been obtained through AST parsing, we only conduct code instrumentation on cryptographic algorithm related functions. Specifically, based on preliminary

experimental results, we find that cryptographic algorithm related functions are usually enveloped by another layer of functions, WalletRadar instrument the location of this enveloping function for further parameter collection. Take a decrypting function under the cryptographic algorithm related category as an example. As we can see from Fig. 3, at Line 13, it invokes the actual AES decrypting function with parameters `c` and `b`, where `b` is passed through argument directly and `c` is derived from another argument `a`. When identifying the function (i.e., `unlock`) that wraps the target function, WalletRadar performs a local data flow analysis to locate its parameters (i.e., `a`) related to the decryption function, establishing such a data dependency relation. Then, we instrument a `collect`, our self-defined function, before the assignment to `c` at Line 7. Consequently, through the instrumented function, we can obtain the values that are passed to the actual decryption function.

### 4.3 Dynamic analyzing

This phase is intended to capture critical runtime data from the wallets, like local storage data, function innovation details, and dynamic HTML content, which is impossible to collect by the static analysis. All collected data will be used for the final vulnerability detection.

#### 4.3.1 Page semantics identification

To conduct an effective runtime information collection, WalletRadar must first identify the functionalities of the current page, i.e., the semantics of the page. As the concrete operating processes of different wallets vary, it is non-trivial to implement a set of general page semantics identification methods. Thus, against sensitive operations, like password setting and mnemonic import, we preliminarily conduct a manual investigation to find common pages that these browser-based wallets may share. Then, we build a semantics database based on the page patterns, like specific text keywords and HTML elements. Taking advantage of this semantics database, WalletRadar is able to efficiently identify the functionality of the current page during the dynamic analysis, and perform the corresponding following operations. We then detail the *manual investigation on key pages* and *semantics database build* in the following.

**Manual Investigation on Key Pages.** The manual investigation process involves examining each wallet's user interface, noting down the standard and sensitive features such as wallet creation, wallet import, password setting, and wallet backup. By cataloging these pages, we were able to define a set of generic templates that represent the core functional pages across different wallets. After the analysis, 13 key pages remain that can cover all sensitive operations in a browser-based wallet, as shown in Fig. 4. These thirteen pages, following the wallet's usage flow, can be divided into five stages: wallet initialization, new wallet creation, existing wallet import, general operations, and wallet backup. The functionalities of these pages are shown in Table 2. As we can see, except for two optional reminder pages, most pages adhere to basic sequential order. In particular, under clearly defined stage

**Table 2** The 13 key pages and their functionalities

| Stage | Page | Functionalities/Page content |
|---|---|---|
| Wallet initiation | Start page | Entry point with welcome information |
| | Data collection reminder | User agreement and privacy policy |
| | Wallet creation preparations | Select wallet creation method |
| New wallet creation | Password setting | Set wallet access password |
| | Security reminder | Displays security best practices |
| | Mnemonic display | Shows backup mnemonic phrase |
| Existing wallet import | Import method selection | Choose wallet import method |
| | Mnemonic import | Wallet recovery via mnemonic |
| General operations | Home page | Main interface with functionalities |
| | Wallet unlock | Unlocking the wallet interface |
| Wallet backup | Wallet setting | Wallet configuration settings |
| | Password verification | Verify password for backup |
| | Wallet backup | Interface for wallet backup |

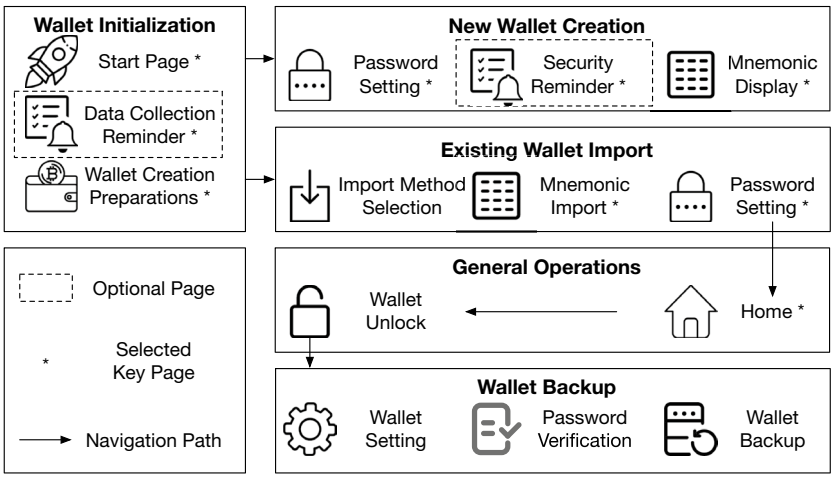



**Fig. 4** The 13 key pages we select and the navigation path during dynamic analyzing. Note that, the password setting page appears twice

sequences, some pages consistently appear after specific ones. For instance, when a user clicks the “Lock" button on the home page, the wallet will navigate to the unlock page. Hence, predictable pages are excluded, and the others are selected as pages requiring the construction of semantic features. Note that, during analysis, we find many wallets merge mnemonic import and password setting features on a single page (referred to as “wallet setup page”) when importing mnemonics, and we add this page to our semantic library. At last, 9 key pages, marked with a “*” symbol in Fig. 4, have been selected for constructing their semantics.

**Table 3** The keyword semantics for the wallet setup page

| Group No | Semantic words or phrases |
|---|---|
| 1 | Import, input, provide,.. |
| 2 | Recovery phrase, mnemonics, seed phrase,.. |
| 3 | Password, credential, PIN,.. |
| 4 | Create, enter, type in,.. |
| 5 | Repeat, confirm, verify,.. |

**Semantics Database Build.** To enable efficient semantics identification on pages, we build a semantics database. Specifically, each row of the database is organized as a key and a series of features, where the key is the corresponding functionality, and the features are composed of several metrics of the functionality, including specific text keywords and HTML elements. For example, mnemonic import is one of the major functionalities of the wallet setup page. Thus, we first build the keyword features of the functionality. The Term Frequency-Inverse Document Frequency (TF-IDF) method, a statistical measure used in text mining and information retrieval Qaiser and Ali (2018), is adopted. This method helps to identify how important a word is to a document in a collection or corpus. Applying TF-IDF to 20 popular wallets, we calculate the frequency of words appearing during the mnemonic import. Subsequently, based on our understanding of sensitive operations, we frame the primary keyword semantics of the functionality in an "Action + Object" format. This involves selecting specific verbs that capture the essence of the functionality's activities (e.g., "import" and "input"), paired with nouns that represent the subjects of these actions (e.g., "recovery phrase" and "mnemonics"). Under this guideline, the keyword semantics for each functionality comprise a few groups of words or phrases. Besides, we also consider interactive HTML elements within the functionalities, such as an input box or 12–24 consecutive input boxes for mnemonic phrase entry during the mnemonic import process.

Table 3 displays the keyword semantics we established for the wallet setup page, encompassing five unique keyword groups: two related to the wallet import and three associated with the creation of a new password. A functionality is identified with a specific semantic if it matches at least one keyword or phrase in each group of the functionality and contains the requisite HTML elements. Furthermore, identifying the semantics of a page implies recognizing the semantics of all its functionalities. Following the rules, a page with keywords like "import", "mnemonic", "password", "enter", "repeat" and corresponding input fields for mnemonics and passwords is identified as a wallet setup page.

### 4.3.2 Page navigation & runtime data collection

With the help of the semantics database, WalletRadar needs to perform appropriate actions according to the current page's functions and elements in order to traverse the wallet's main features while concurrently collecting runtime data. Besides, as we

mentioned in §4.3.1, pages often display in a certain order. Determining navigation paths for these pages can enhance the efficiency of the dynamic analysis. While navigating through the pages, the functions instrumented in the static analyzing phase will be triggered to collect runtime data, like local storage data, function innovation details and dynamic HTML content. We will detail the *page navigation* and *runtime data collection* in detail in the following.

**Page Navigation.** Specifically, two navigation routes are designed, according to the lifecycle of a wallet, as shown in Fig. 4. One regards the wallet as a newly created one and the path is mainly composed of wallet creation and mnemonic display. Another path assumes the wallet is imported, i.e., including mnemonic import, password setting, wallet unlock and wallet backup. Specific operations for each page are set through the Selenium framework Selenium (2022), a famous framework for automating web browsers. Except for the "mnemonic display page" and "mnemonic backup page", which are respectively the ends of two navigation paths, we customize the operation scripts for the remaining 11 pages. Upon arriving at a particular page, the corresponding script is triggered. Although pages with different semantics will be navigated in different manners, the script is designed to follow a general strategy. Firstly, the script locates the interactive elements with their labels, such as buttons, checkboxes and input fields, etc., which helps WalletRadar gain an overall understanding of the current page and facilitate further user action simulations. Secondly, it manages pop-ups and checkboxes by clicking on prompts like "continue" and this reflects real-world scenarios where users frequently encounter and interact with such elements for confirmations or agreements. Thirdly, it comes to the input fields of the current page. The script systematically populates them with predefined credentials, such as usernames and passwords. This step is particularly important for testing specific pages like the password setting or mnemonic import pages, as it allows WalletRadar to test these functionalities as expected and advance the testing path accordingly; in cases where labels correspond to fields not covered by our predefined credentials (such as wallet nicknames), the script will generate and input random strings to help complete the functionality of the page. At last, the script seeks to navigate to the next pages by engaging elements like "Next" or "OK", facilitating WalletRadar in successfully completing the predefined navigation route.

**Runtime Data Collection.** During these processes, the instrumented code is triggered to collect runtime data and sensitive data generated during the interaction, like wallet passwords and mnemonics. Besides, during the page navigation, WalletRadar takes advantage of the periodic call function in JavaScript to continuously monitor the changes of the current HTML page and local storage (including LocalStorage, SessionStorage, IndexedDB and local session files). The monitor is performed once a second whenever modifications occur.

Take the mnemonic import page as an example to illustrate how the page navigation and runtime data collection are conducted, the process of which is shown in Fig. 5. On this page, the major interactive element the WalletRadar locate will be a certain number (12, 15, or 24) of input boxes for entering mnemonics (12 boxes in the example), or a single text box to receive all the mnemonics separated by spaces; next, after handling pop-up windows and checkboxes, WalletRadar fills in the input

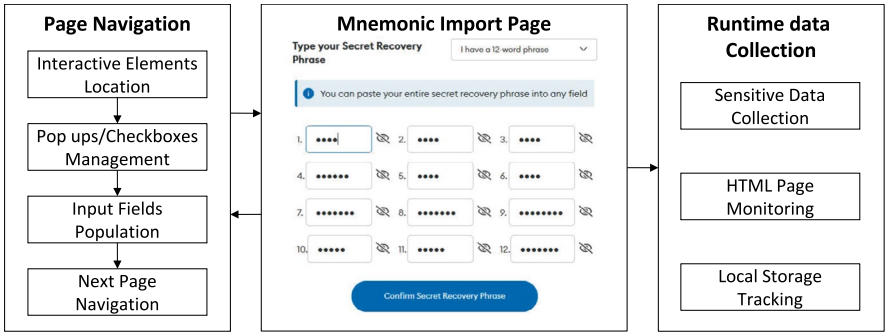



**Fig. 5** The page navigation and runtime data collection on a mnemonic import page

boxes or the text box with the mnemonic words generated during the previous wallet creation; for other non-sensitive information (such as the wallet name), random strings are generated and entered. The import process is completed by pressing the confirm button (blue button in the exmaple). Due to code instrumentation during the static analyzing phase, WalletRadar will capture the HTML code of the front-end page during mnemonic import and log the local storage data after the mnemonic input. After the wallet import, the wallet will navigate to the home page and then progress to test the wallet unlock page.

### 4.4 Vulnerability detecting

In the vulnerability detecting phase, WalletRadar takes advantage of information collected from the previous static analyzing and dynamic analyzing phases to identify potential vulnerabilities hidden in the given wallets. Specifically, WalletRadar integrates six rule-based detectors, corresponding to the six vulnerabilities mentioned in §3.1. We argue that WalletRadar can be easily extended by implementing detectors based on the collected information. The detecting strategies are detailed in the following.

**Clickjacking.** As depicted in §3.2.1, clickjacking in browser-based wallets is related to the "web_accessible_resource" item in the "manifest.json" configuration file. If sensitive HTML pages, such as the home page and transaction page, are present in the configuration item according to the static analysis, it can be concluded that these web pages can be accessed by external websites (including phishing sites) and the wallet is exploitable to this vulnerability.

**XSS Vulnerability.** According to the guidance provided by PortSwigger Dom-based XSS (2022), during the static analyzing phase, WalletRadar traces back from DOM-manipulating functions to the data source. If the data source function at the end of this trace is found in the valuable function database and is identified as being susceptible to external modification, it signifies a possible DOM-based XSS vulnerability. This scenario highlights a risk where the content within the wallet's web pages could be externally altered or compromised, making the wallet vulnerable to XSS attacks.

**Defective Password Policy.** In the dynamic analyzing phase, when encountering a password setting page, the WalletRadar will try to test a set of passwords on the page, which start from the weakest one (e.g., "123") to the relatively strong one (e.g. "Weasdxz@a142"). It will then record the weakest password that finally passes the password setting of one wallet. According to the CheckPoint report Check Point Research Detects (2022), if the password is composed of only 6 or fewer digits, the strength of this password is considered insufficient for supporting the security of the browser-based wallet.

**Redundant Storage.** During the dynamic analyzing phase, WalletRadar continuously records all intermediate data. Thus, this detector compares the data stored in the local storage and the intermediate data generated during the decryption. If the intermediate data can be matched to part of the local storage data, the wallet may leak sensitive information, i.e., it is vulnerable to redundant storage vulnerability.

**Demonic Vulnerability.** This detector focuses on the textual elements (e.g., "textarea" tags) in front-end HTML pages. To detect demonic vulnerabilities, the detector scans for specific textual elements on HTML pages during sensitive operations, particularly focusing on those that hold plaintext mnemonics or private keys. If these elements are found and corresponding plaintext data is also present in the browser's local storage, the wallet is flagged for demonic vulnerability.

**Defective cryptography.** If the detector finds that the wallet applies PBKDF2 with less than 10K rounds or uses inappropriate methods like AES-CBC mode, the wallet is believed to have a defective cryptography vulnerability.

## 5 Evaluation

In this section, we conduct a comprehensive evaluation on WalletRadar to characterize the vulnerabilities in browser-based wallets in the wild.

### 5.1 Research questions & experimental setting

In this paper, we are interested in the following questions:

**RQ1 Is WalletRadar efficient and effective in detecting these vulnerabilities?**

**RQ2 What are the characteristics of vulnerable wallets in the wild?**

To answer RQ1, since there are no existing datasets for browser-based wallets, we collect a dataset from the Chrome Web Store Chrome Web Store (2023). We manually inspect these samples to build a reliable benchmark to evaluate the efficiency and effectiveness of WalletRadar. To answer RQ2, based on the detection results, we analyze the characteristics of these vulnerabilities, evaluate their impacts and track the developers' responses and remediation efforts.

**Table 4** The top 10 browser-based wallets that most users download

| Name | Version | Users | Supported blockchains |
|---|---|---|---|
| Metamask | 10.14.0 | 10 M+ | Ethereum, Polygon,.. |
| Phantom | 22.9.6 | 2 M+ | Solana, Ethereum,.. |
| Ronin Wallet | 1.23.1 | 1 M+ | Ronin |
| Binance wallet | 2.13.7 | 1 M+ | BNB Chain, Ethereum,.. |
| Coinbase | 2.30.2 | 1 M+ | Ethereum, Avalanche,.. |
| Keplr | 0.11.1 | 900K+ | Osmosis, Mars,.. |
| Station | 3.1.0 | 600K+ | Terra |
| Argent X | 5.2.0 | 600K+ | Ethereum |
| TronLink | 3.26.9 | 500K+ | Tron |
| Martian Wallet | 0.2.2 | 500K+ | Sui, Aptos |

**Experimental Setting.** We implement WalletRadar based on Python3. In specific, WalletRadar utilizes js-beautify beautify-web (2022) for code formatting, Esprima Esprima (2022) for AST parsing of JavaScript, and the Selenium framework Selenium (2022) for invoking automated scripts. Other analytical components, including automated runtime scripts, valuable function databases, and detection rules, are all designed independently. The following experiment is performed on a laptop with the Intel Core i7-12700 H@2.3GHz Processor and 16 G RAM. The Selenium framework is operated on a Chrome web browser (version:102.0.5005.189) to test browser-based wallets dynamically.

**Dataset Collection.** We collect browser-based samples from the Chrome Web Store, one of the most widely adopted and well-known platforms for browser extensions. There are 618 results when searching "blockchain wallet". To produce an effective result, we conduct a filtering process based on some criteria. First, as our subject is the browser-based non-custodial wallet, other types of wallets are not considered. Second, these wallets need to be popular. Thus, we keep the wallet with more than 3K users. To cover as many use cases as possible, we consider wallets that support either a single or multiple blockchains. At last, we have collected 120 samples in total. However, we find some of them are not fully functional, such as failure at wallet creation or mnemonics import. Thus, based on user comments on the web store, we removed the samples with bad reputations. Consequently, 96 samples are regarded as candidates, corresponding to multiple blockchains like Bitcoin, Ethereum, and Solana. Considering the limited number of samples and the absence of established vulnerability ground truth in our dataset, we opt for a comprehensive manual analysis of each sample. To build a trustworthy benchmark, the manual labeling is conducted independently by two experienced authors who are familiar with typical vulnerability signatures in browser-based wallets. By doing so, we precisely identify and classify relevant vulnerabilities, thereby creating an accurate benchmark dataset to guide future analyses and comparisons.

**Dataset Overview.** In summary, these samples have been downloaded at least 23 million times in total, accounting for about 97% of total downloads in the search results of "blockchain wallet" on the Chrome Web Store. This indicates that a

**Table 5** The detection results of 96 samples

| Category | # of Vulnerabilities | False positives | False negatives |
|---|---|---|---|
| Demonic vulnerability | 55 | 0 | 0 |
| Defective password policy | 20 | 0 | 0 |
| Redundant storage | 18 | 0 | 0 |
| Clickjacking | 13 | 0 | 0 |
| Defective cryptography | 2 | 0 | 4 |
| XSS | 2 | 0 | 2 |
| Total | 110 | 0 | 6 |

browser-based non-custodial wallet is the major choice of blockchain users. The top 10 browser-based wallets according to download times are shown in Table 4. As we can see, they have at least 500K users and support for more than 10 blockchains in total, highlighting their widespread popularity and versatility in catering to diverse blockchain platforms and user needs.

### 5.2 RQ1: Efficiency & effectiveness

In this section, we evaluate the efficiency and effectiveness of WalletRadar on the collected 96 samples when identifying vulnerabilities.

#### 5.2.1 Automation efficiency test

We performed automated tests on these 96 samples using WalletRadar. Among them, 9 samples only completed the static analyzing phase because of their unique attributes during creating wallets, importing wallets, etc., preventing the full completion of automated dynamic operations. For example, one wallet may require a long press to export mnemonic phrases while another directs users to an external website during page navigation, both scenarios disrupting typical test workflows. However, the remaining 87 samples still successfully completed all the automated analysis processes, achieving a 90.6% automation completion rate. We also manually intervened on the 9 samples that failed to complete the dynamic analyzing phase to ensure they underwent full vulnerability detection. In the automation testing process, the execution time for each sample was approximately 8 min, with around 5 min for the static analyzing phase and about 3 min for the dynamic analyzing phase. This suggests that WalletRadar's automation testing is relatively swift, owing to its clear execution paths and operations.

#### 5.2.2 Evaluation of detection results

The detection results on 96 samples are shown in Table 5. As we can see, a total of 110 vulnerabilities in 70 samples (73% of all tested samples) are identified by WalletRadar. The demonic vulnerability becomes the most serious problem among these wallet extensions, which accounts for 57% of all tested samples. This

may result from the situation that many wallet extensions forked the early versions of the famous wallet Metamask, which previously contained a demonic vulnerability. The other three slightly less serious issues are defective password policy (20), redundant storage (18), and clickjacking (13). This suggests that both wallet-specific vulnerabilities and traditional web vulnerabilities are prevalent on wallet extensions. Among them, 53% (37) of the wallets have one vulnerability and the wallet with the most vulnerabilities has 5 distinct issues except defective cryptography.

As for the false alarms, we define that a false positive happens when WalletRadar flags a non-existent vulnerability, and a false negative occurs when it misses an actual vulnerability. To ensure a precise evaluation of WalletRadar's detection capabilities, the evaluation process involves a detailed review of the target wallets' code and functionalities to identify known vulnerability patterns and compare them against the tool's findings. After the evaluation, we discover that there are no false positives for all six vulnerabilities. We speculate this is due to the characteristics of these vulnerabilities and the efficiency of WalletRadar. Leveraging the comprehensive valuable function database at the outset enhances the accuracy of our detections, complemented by the use of specific static rules that precisely identify unique vulnerability patterns. Additionally, our accurate dynamic testing methods distinguish between true vulnerabilities and normal behaviors. Moreover, we can easily observe that WalletRadar cannot fully detect XSS and defective cryptography vulnerabilities. After manual verification, we discover that the four defective cryptography cases include two "instances of insecure AES usage" and two instances of "insufficient iterations". The cryptographic parameters of these four samples are generated dynamically within closures or local scopes, instead of being hardcoded, thus they evade the static analysis. The dynamic instrumentation, designed to capture parameters at runtime, also failed in these cases because the parameters were generated within specific execution contexts or transient states that were not active or accessible during the instrumentation phase, preventing their capture and analysis. As for the two false negatives in the XSS vulnerability, we find that the data sources and sink points are spread across different files. Therefore, the AST-based analysis method struggles with such inter-file detection, leading to these oversights.

**RQ1 Answer:** Our experiments, conducted on 96 widely-used cryptocurrency wallets, demonstrate that our tool, WalletRadar, can automatically complete the detection process on over 90% of these wallets, with the capacity to cover all wallets when supplemented by manual intervention. Furthermore, WalletRadar exhibits high accuracy in our collected dataset, characterized by the absence of false positives and a minimal incidence of false negatives. In summary, WalletRadar proves to be both efficient and effective in identifying vulnerabilities.

### 5.3 RQ2: Characteristics of vulnerable wallets in the wild

In this section, we first characterize how these vulnerabilities spread in the wild. Then, we evaluate the impact directly brought from them.


**Fig. 6** The demonic vulnerability in P* wallet

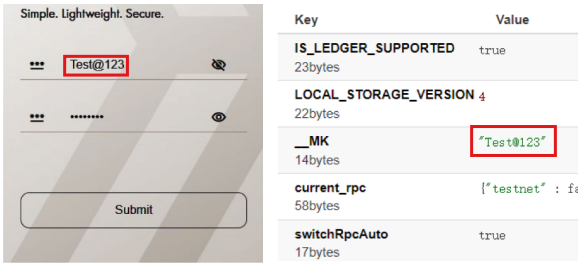
### 5.3.1 Studies on real-world vulnerabilities

**Demonic Vulnerability.** During our evaluation, it was found that 55 wallets use textual HTML tags to store sensitive information when backing up, importing or displaying mnemonics and private keys. Take P* Wallet[1] in Fig. 6 as an example, when a user displays the private key, the wallet uses textual tags to store the private key, and this data will be cached in the local storage, posing a threat of sensitive data leakage. The way to fix this vulnerability is to use 12–24 input boxes with "input" tags and a "password" attribution in an HTML page to accommodate the user's mnemonic or private key. When this implementation is adopted, the user needs to perform multiple inputs, which may be cumbersome and require developers to invest more effort in mnemonic input optimization.

**Defective Password Policy.** Among the 20 samples with password policy flaws, 1 sample requires a minimum of 4 digits, 9 samples do not require password complexity, and 10 samples require a minimum of 6 digits. It is worth mentioning that among the 96 samples, only 29 samples have the minimum password complexity requirement of mixed letters and numbers. In real-world scenarios, there is a high likelihood of weak passwords in browser-based cryptocurrency wallets, resulting in more accessible brute-force cracking and endangering the security of users' wallets. For building a secure cryptocurrency wallet, it is wise to require a minimum of 8-digit password with a mixture of numbers and letters.

**Redundant Storage.** All cases of this vulnerability are found in the process of unlocking the cryptocurrency wallet, and the evaluation results show that this

[1] Due to ethical considerations, we have anonymized the names of the wallets, displaying only their initial capital letters.




(a) The password setting page and the local storage of H* Wallet

```
async function C(e, t) {
    const {encrypted: a, nonce: n, salt: r, iterations: i,
      , u = s().decode(a)
      , m = s().decode(n)
      , _ = s().decode(r)
      , f = await x(t, _, i, d)
      , p = o().secretbox.open(u, m, f);
    if (!p)
        throw new Error(l.ZP.t("error_invalid_password"));
    const g = c.from(p).toString();
    return JSON.parse(g)
}
```

(b) The decryption function of S* Wallet

**Fig. 7** The redundant storage vulnerability in two wallets

vulnerability is severe in actual situations. 18 samples demonstrated varied manifestations of this vulnerability:

(*i*) Embedding the key required by the decryption function within the code.

(*ii*) Storing all raw materials needed for the decryption function in local storage.

(*iii*) Using symmetric encryption based on a timestamp to save the user's password in local storage, where the timestamp is stored in plaintext.

(*iv*) Encrypting and storing the user's password in local storage using a fixed key in symmetric encryption, with the fixed key embedded within the code.

(*v*) Storing the user's plaintext password directly in local storage.

(*vi*) Storing the initial hashing result of the user's password in local storage (without using it for decryption).

The first five cases can directly result in users losing control of their wallets if an attacker gains access to the stored content. In the sixth case, although the wallet stores the initial hash result (e.g., SHA-512) of the user's password in local storage, it does not use the hash for decryption. Instead, when the user enters their password, the wallet performs the same hashing process and compares the two hash results for password verification. Although this practice cannot directly lead to the decryption of the user's wallet, it enables an attacker to extract the hash and use it for brute force cracking, posing a severe threat to the user's wallet. Figure 7a is an example where the H* Wallet directly saves the user's password with plaintext in the local storage, making the wallet vulnerable to attacks. For S* Wallet, although it saves the processed password in the local storage instead of plaintext ones, the data can be directly used in its decryption function shown in Fig. 7b to get the mnemonics, which is also an insecure practice.

All six cases are related to poor practices of wallet developers when they deal with sensitive data storage and the fifth case is the worst one, which fully exposes

```
<iframe src="chrome-extension://{extension id}/phishing.html?href=chrome-
    extension://{extension id}/wallet.html" width="100%" height="100%">
</iframe>
```

**Fig. 8** The code implementation of the clickjacking in S* wallet

the credentials. It is recommended to keep the decryption process in real-time and not save any sensitive data in local storage just for ease of use.

**Clickjacking.** 13 wallet samples were found with clickjacking vulnerabilities, which indicates that such vulnerabilities in the actual situation are also very prominent. We have identified two situations where these cryptocurrency wallets have such vulnerabilities: the introduction of the “wallet home page" and the “security reminder page" in the “web_accessible_resources” configuration. Some wallets directly introduce the “wallet home page" in configuration, and this kind of implementation makes the home pages of these wallets vulnerable to hijacking attacks, causing users to unwittingly download malware, visit malicious web pages or provide sensitive information. Besides, while some wallets also provide users with some security detection services (phishing detection, etc.) and click-jumping functions, they inadvertently introduce this type of vulnerability as a consequence. Figure 8 shows a basic example that exploits the vulnerability. The code can be embedded in a phishing website to overlay the wallet’s phishing warning page, inducing users to jump to the wallet home page and perform sensitive operations.

The original purpose of introducing files in the “web_accessible_resources” configuration is to expose the resources (such as images) for external web pages to access. However, when introducing HTML files, other external pages can directly access the web pages and there is a possibility that these HTML pages are embedded in a phishing website, resulting in clickjacking. The wallet developers need to avoid adding HTML files to this configuration or ensure that the HTML files added do not contain key wallet functions such as importing wallets, sending transactions, etc.

**Defective Cryptography.** In cryptographic practices, it is generally advised to employ algorithms like argon2, scrypt, and PBKDF2 with a higher number of iterations to derive keys based on users’ passwords. In our study, while most wallets adhere to the recommended 10, 000 iterations for PBKDF2, three samples fall below this standard, using fewer than 5, 000 iterations, one of which is depicted in Fig. 9. Only five samples demonstrated exceptional security with 310, 000 iterations. This suggests that most wallets generally meet cryptographic standards but often do not implement the highest level of security practices. Combined with inadequate password policies, this shortfall in implementing the highest security standards increases the risk of brute-force attacks.

Besides, the majority of wallets employ AES for symmetric encryption and decryption. We discovered only three instances that used the less secure CBC mode, while the rest followed best practices by using GCM or CTR mode.

**XSS Vulnerability.** Among the 96 samples, only 4 samples were detected with DOM-based XSS vulnerabilities (one of whose code is shown in Fig. 10). These vulnerabilities occurred in HTML pages associated with clickjacking and were

```
function a(e, t) {
    var o = n.utf8ToBuffer(e)
      , i = n.base64ToBuffer(t);
    return r.crypto.subtle.importKey("raw", o, {
        name: "PBKDF2"
    }, !1, ["deriveBits", "deriveKey"]).then((function(e) {
        // Generate the key for decryption iteratively
        return r.crypto.subtle.deriveKey({
            name: "PBKDF2",
            salt: i,
            iterations: 5e3,
            hash: "SHA-256"
        }, e, {
            name: "AES-GCM",
            length: 256
        }, !1, ["encrypt", "decrypt"])
    }
    ))
}
```

**Fig. 9** The code snippet that has a defective cryptography vulnerability

```
window.onload = function() {
    if ("/phishing.html" === window.location.pathname) {
        // Extract the "hostname" parameter and assign to the variable "e"
        const {hostname: e} = h();
        // Write the parameter value directly into the HTML page
        document.getElementById("esdbLink").innerHTML = '<b>To read more about
            this scam, navigate to: <a href="https://etherscamdb.info/domain/'
             + e + '"> https://etherscamdb.info/domain/' + e + "</a></b>"
    }
}

function h() {
    // Parse the hash in the current URL
    const e = window.location.hash.substring(1);
    return o.parse(e)
}
```

**Fig. 10** A code snippet that has an XSS vulnerability

related to the security reminder function provided by the wallet, indicating that exposed HTML pages (introduced in the “web_accessible_resources" configuration) are more likely to have XSS vulnerabilities in the browser-based cryptocurrency wallets. Notably, although the code of these samples suggests that they are vulnerable to XSS attacks, further manual inspection reveals that their adoption of the Content Security Policy (CSP), the added layer of security, mitigates the impact of this kind of vulnerability.

### 5.3.2 Impact

Based on the analysis result, we find 70 vulnerable wallets with at least 9.2 million downloads. Among these wallets, the most popular one achieves more than a million downloads and 23% of these wallets have more than 100K users. If these wallets are attacked, a large number of users will be exposed to the risk of information leakage and even financial losses. Thus, when we finished the analysis of these 70

**Table 6** The current status of identified vulnerabilities in browser-based wallets

| Vulnerabilities | # of Vulnerabilities | Fixed | Confirmed |
|---|---|---|---|
| Demonic vulnerability | 55 | 18 | 6 |
| Defective password policy | 20 | 6 | 3 |
| Redundant storage | 18 | 3 | 1 |
| Clickjacking | 13 | 6 | 0 |
| Defective cryptography | 6 | 3 | 0 |
| XSS | 4 | 0 | 0 |
| Total | 116 | 36 | 10 |

examples in February 2023, we reproduced the vulnerabilities to confirm their existence and attempted to get in touch with the developers to report these issues. Finally, we got confirmations of 10 vulnerabilities from the developers of 8 wallets with over $2000 bounties. By study time, we revisit the wallets with vulnerabilities and check whether these vulnerabilities have been fixed. The outcomes of these checks are detailed in Table 6, presenting an overview of the vulnerabilities' current status.

**Active Wallets without Fixing the Issues.** We refer to the wallets that have been updated in the last six months as "active wallets". For 29 active wallets, our manual inspection suggests that their developers are mainly involved in updating features or fixing bugs in functionality, such as adding support for DApps, launching promotion campaigns for the new tokens, or integrating with cryptocurrency exchanges, etc. Besides, Some developers believe it is the users' responsibility to keep the wallet safe and refuse to fix weaknesses related to password policy and cryptography. Although wallet users control their keys and are primarily responsible for keeping them safe, expecting all users to have good password management habits is unrealistic. Thus, wallet developers should also pay as much attention as possible to the security of the wallets in order to set the stage for wallet users and mitigate the potential risk to users.

We also find that some wallets proclaimed to have fixed demonic vulnerabilities but only fixed those mentioned in Metamask blog Security Notice (2022), overlooking other functions with demonic vulnerability. Figure 6 presented in §5.3.1 shows the example of P* Wallet, which leaves the private key display page unfixed. This suggests that some developers lack comprehensive knowledge of vulnerability-related information, resulting in incomplete vulnerability fixes.

**Active Wallets with Issues Fully/Partly Fixed.** Among 26 active wallets, developers of 8 wallets have confirmed 10 vulnerabilities of their wallets to us and all of them have fixed the issues by study time. 5 wallets fully fixed 5 issues before our reports and 10 wallets addressed 14 vulnerabilities post-report silently, with 4 of the fixes occurring before our contact. Among the remaining 3 wallets, none have fully addressed their vulnerabilities. Two wallets (including the wallet with the most vulnerabilities mentioned in §5.2.2) have only resolved the demonic vulnerability silently after our reports, leaving other vulnerabilities unfixed. The developers' choice to fix severe vulnerabilities like the demonic vulnerability first, both in terms

of proportion and sequence, may suggest they prioritize the most critical issues, intending to address less severe vulnerabilities in future updates.

Besides, one wallet fixes the clickjacking vulnerabilities before our report while still leaving its 6-digit password policy unchanged after the report. We speculate this is because the developers may have been concerned that fixing the password policy would create backward compatibility issues, or simply think that the current password strength is sufficient.

**Inactive Wallets.** For the remaining 15 wallets that are not updated or removed from the Chrome web store, we find some of these wallet developers have moved to new projects after constructing a basic feature of their wallet extensions while some just stopped the maintenance. However, as non-custodial wallets can operate without central services, the users who still use these wallets may suffer from unsuspecting attacks. Considering at least 314K users that have ever used these wallets, there may be a number of users who are at risk of being attacked.

In summary, our vulnerability disclosures result in updates to 20 wallets by their developers, which include developers of 8 wallets who confirmed and fixed the issues and developers of 12 wallets who silently patched the vulnerabilities, fixing a total of 26 vulnerabilities. This accounts for 22.4% of the vulnerabilities we identified, protecting their thousands of users from potential attacks. However, it is worth noting that many developers still do not place enough emphasis on the security development process of their wallets, thereby potentially compromising the security of their user base.

**RQ2 Answer:** The case studies of 116 real-world vulnerabilities reveal a widespread prevalence of security issues in browser-based wallets. Notably, the demonic vulnerability becomes most critical and it is often overlooked by developers regarding its potential occurrence. Other vulnerabilities also shed light on a mix of wallet-specific and general web vulnerabilities within these browser extensions. This underscores a potential deficiency in secure development practices among their developers. The conducted impact analysis reveals that these vulnerabilities pose a significant risk to approximately 9.2 million users, with threats ranging from information leakage to financial loss. In response to our disclosure of these vulnerabilities, there has been a rectification of 26 vulnerabilities across 20 different wallets, thereby mitigating their possible adverse effects.

## 6 Discussion

### 6.1 Implications

The work on characterizing and detecting security issues in browser-based cryptocurrency wallets is essential for stakeholders in the community.

For developers and wallet users, the taxonomy of security issues in browser-based cryptocurrency wallets assists them in understanding potential vulnerabilities, offering a valuable reference throughout the application's security lifecycle. Besides, despite the diversity in programming languages and platforms, other kinds of

blockchain wallets and blockchain applications often share common operational workflows or mechanisms. For example, many desktop wallets also use the same key management techniques (e.g., using PBKDF2 for key generation) as browser-based wallets. Thus, certain components of WalletRadar, including automated testing modules and identified vulnerability patterns, might be adapted for use in other applications with some modifications.

The evaluation of our proposed detection framework demonstrates the effectiveness of our proposed approach while suggesting that a large number of current browser-based wallets suffer from various security vulnerabilities. Besides, although WalletRadar is primed to detect vulnerabilities identified in this paper, it also retains high extensibility. For instance, the vulnerable function database and the rules for the vulnerability detector can be added to or optimized, facilitating targeted detection of subsequent vulnerabilities related to browser-based wallets. Combining this approach with web application audit methods, as described in this paper, can help reduce vulnerabilities during wallet development and lower the risk of financial loss for users.

Further, our impact analysis of these vulnerabilities suggests that millions of users may be vulnerable to attacks related to these vulnerabilities, which could result in substantial financial losses. Moreover, our subsequent analysis of these wallets over time after reporting these vulnerabilities shows that only a small number of developers fully fix these vulnerabilities, while some serious vulnerabilities such as the demonic vulnerability are even mistakenly believed to be fixed due to developers' superficial knowledge of them. Developers should take wallet security issues more seriously and gain a better understanding of these vulnerabilities to effectively fix them and fully mitigate their impact.

We recognize the ethical aspects of detecting vulnerabilities in browser-based wallets and maintain confidentiality by anonymizing application names. Our responsible Conflict of interest ensures developers have time to fix issues before we make them public, and we provide repair strategies for all vulnerabilities. Our goal is to improve the security of the blockchain ecosystem, making it safer for users.

### 6.2 Limitations

Firstly, we initially established a taxonomy of browser-based cryptocurrency wallet security issues based on information collection from various channels (as described in §3.1). While we have done our best to refine the taxonomy, it may still be incomplete. Nevertheless, the utility and effectiveness of our proposed taxonomy were confirmed through evaluation, revealing numerous security issues faced by current browser-based cryptocurrency wallets.

The current keyword-based semantic approach for identifying key wallet pages in dynamic analysis might fall short in more complex semantic contexts. More advanced techniques such as optical character recognition (OCR) and large language models (LLMs) could enhance page traversal capabilities by understanding and interpreting complex page content more effectively. These techniques promise to refine the automation process, potentially increasing the coverage beyond the current 90% and reducing the need for manual intervention.

Besides, as most browser-based wallets are non-custodial wallets and their users' addresses are not publicly known, it is hard for us to track whether these wallets are actually under attack or not on the blockchain. Nevertheless, our experiments and the following investigations indicate that these vulnerabilities are still not fixed and could have the potential to cause financial loss to a large number of users.

## 7 Related work

### 7.1 Web application analysis

Browser extensions for cryptocurrency wallets are essentially web applications, leading us into a broader discussion on web application analysis in software engineering. Since it has been in development for many years, there has been a lot of work dedicated to studying web applications and developing related analysis tools. Some efforts focus on automatic web application testing Nguyen et al. (2019); Zheng et al. (2021); Lin et al. (2023). For example, Zheng et al. Zheng et al. (2021) presents an end-to-end automated web testing framework. Using curiosity-driven reinforcement learning, it efficiently generates high-quality action sequences for web application testing. Another major and more relevant research direction to this work is developing automated detection frameworks to detect security issues in web applications like XSS Pan et al. (2017); Eriksson et al. (2022); Pazos et al. (2023) or privacy breach related vulnerabilities Zhao et al. (2015); Chen and Kapravelos (2018); Kariryaa et al. (2021). For example, Pan et al. Pan et al. (2017) propose a detecting framework employing hybrid analysis combined with lightweight static analysis consisting of a text filter and an abstract syntax tree parser and dynamic symbolic execution to detect DOM-based XSS vulnerabilities. To detect privacy leaks, Chen et al. Chen and Kapravelos (2018) propose a hybrid taint analysis technique that leverages both dynamic taint tracking and static analysis by using information gathered from static data flow and control-flow dependency analysis to propagate taint at runtime. Besides, Kariryaa et al. Kariryaa et al. (2021) find that users have limited technical knowledge about browser extensions' capabilities and they are only focused on the features these extensions provide.

The above work provides many insights into this paper for the vulnerability taxonomy and the detection methods of browser extension vulnerabilities. Based on these static and dynamic analysis methods, this paper customizes and optimizes the WalletRadar to suit the specific needs of browser-based cryptocurrency wallets, addressing their distinct vulnerabilities and operational processes.

### 7.2 Cryptocurrency wallet analysis

Current research on cryptocurrency wallet mainly focuses on mobile application wallets Sai et al. (2019); He et al. (2020); Li et al. (2020); Praitheeshan et al. (2020); Hu et al. (2021); Uddin et al. (2021). For example, Sai et al. Sai et al. (2019) first used static code analysis and network data analysis to evaluate the security issues

of Android-based cryptocurrency wallets, and found that the security of popular cryptocurrency wallet apps is not significantly worse than banking apps, but they lack privacy protections. Li et al. Li et al. (2020) assessed the security issues of cryptocurrency wallet apps and presented a comprehensive attack surface on them. He et al. He et al. (2020) carried out related attack experiments on the premise that the attacker can access the user's mobile device with high privileges.

In addition to researchers focusing on mobile wallets, Praitheeshan et al. Praitheeshan et al. (2020) conducted a security assessment of smart contract-based on-chain Ethereum wallets. With the help of automated scanning tools, they conducted a security analysis on the wallets on the chain using smart contracts and gave a classification of security issues. Guri M et al. Guri (2018) conducted a security analysis on air-gapped cryptocurrency wallets (i.e., wallets isolated from the Internet) and proved that attackers may still attack isolated offline wallets to steal private keys through various exfiltration techniques.

There is a lack of systematic security analysis of browser-based cryptocurrency wallets. Considering the large user base of browser-based wallets, it is necessary to get an understanding of them and develop an automated security assessment tool to help developers deal with potential vulnerabilities during the development process. In this work, we take the first step to characterize and detect vulnerabilities in browser-based wallets.

## 8 Conclusion

This paper presents the first systematic assessment of vulnerabilities in browser-based cryptocurrency wallets. We propose WalletRadar, an automated detection framework leveraging a hybrid of static and dynamic analysis to efficiently identify vulnerabilities. The experiments on popular browser-based wallets demonstrate that WalletRadar operates automatically on the majority of these wallets with high accuracy. The evaluation results have also uncovered widespread security issues, highlighting a concerning lack of awareness among many developers regarding these vulnerabilities. Unfortunately, only a few developers have thoroughly addressed these security flaws.

**Funding** There is no external funding received for this work.

**Data Availability** For data availability, due to the sensitive nature of the research, detailed vulnerability data in browser-based wallets cannot be openly shared. Basic wallet information used in experiments will be available at https://github.com/WebWalletVul/WebWallet, with detailed vulnerability data provided upon reasonable request for confidentiality and security reasons.

### Declarations

**Conflict of interest** The authors have no Conflict of interest to declare that are relevant to the content of this article.

**Ethical approval** Not applicable since there are no human and/or animal studies included in this paper.

## References


A Guide to Wallet Security and Best Practices. https://www.algorand.foundation/wallet-security-best-practices (2022)

beautify-web/js-beautify: Beautifier for javascript - GitHub. https://github.com/beautify-web/js-beautify (2022)

Bitcoin - Open source P2P money. https://bitcoin.org/en/ (2022)

Calzavara, S., Roth, S., Rabitti, A., Backes, M., Stock, B.: A tale of two headers: a formal analysis of inconsistent {Click-Jacking} protection on the web. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 683–697 (2020)

Chrome Web Store. https://chrome.google.com/webstore/category/extensions?hl=en (2023)

Check Point Research Detects Vulnerability in the Everscale Blockchain Wallet, Preventing Cryptocurrency Theft. https://research.checkpoint.com/2022/check-point-research-detects-vulnerability-in-the-everscale-blockchain-wallet-preventing-cryptocurrency-theft/ (2022)

Chen, Q., Kapravelos, A.: Mystique: Uncovering information leakage from browser extensions. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1687–1700 (2018)

CoinMarketCap: Cryptocurrency Prices, Charts And Market. https://coinmarketcap.com/ (2023)

Coinbase Wallet - Your key to the world of crypto. Coinbase Wallet - Your key to the world of crypto (2023)

CPR Alerts Crypto Wallet Users of Massive Search Engine Phishing Campaign That Has Resulted in at Least Half a Million Dollars Being Stolen. https://research.checkpoint.com/2021/cpr-alerts-crypto-wallet-users-of-massive-search-engine-phishing-campaign-that-has-resulted-in-at-least-half-a-million-dollars-being-stolen/ (2022)

Crypto Market Sizing Report H1 2023. https://crypto.com/research/crypto-market-sizing-report-h1-2023 (2023)

Crypto Wallet Security – A Complete Guide. https://www.appsealing.com/crypto-wallet-security/ (2022)

Double trouble: crypto-stealing DoubleFinger. https://www.kaspersky.com/blog/doublefinger-crypto-stealer/48418/ (2023)

Dom-based XSS. https://portswigger.net/web-security/cross-site-scripting/dom-based (2022)

Eriksson, B., Picazo-Sanchez, P., Sabelfeld, A.: Hardening the security analysis of browser extensions. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pp. 1694–1703 (2022)

Esprima. https://esprima.org/ (2022)

From \$900 to \$20,000: Bitcoin's Historic 2017 Price Run Revisited. https://www.coindesk.com/markets/2017/12/29/from-900-to-20000-bitcoins-historic-2017-price-run-revisited/ (2017)

FTX to start U.S. bankruptcy proceedings, CEO to exit. https://www.reuters.com/business/ftx-scrambles-funds-regulators-take-action-2022-11-11/ (2022)

Gupta, S., Gupta, B.B.: Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. Int. J. Syst. Assur. Eng. Manag. **8**, 512–530 (2017)

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 (2018). IEEE

He, D., Li, S., Li, C., Zhu, S., Chan, S., Min, W., Guizani, N.: Security analysis of cryptocurrency wallets in android-based applications. IEEE Netw. **34**(6), 114–119 (2020)

Hu, Y., Wang, S., Tu, G.-H., Xiao, L., Xie, T., Lei, X., Li, C.-Y.: Security threats from bitcoin wallet smartphone applications: Vulnerabilities, attacks, and countermeasures. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy, pp. 89–100 (2021)

Huang, L.-S., Moshchuk, A., Wang, H.J., Schecter, S., Jackson, C.: Clickjacking: attacks and defenses. In: 21st USENIX Security Symposium (USENIX Security 12), pp. 413–428 (2012)

Hodlers beware! New malware targets MetaMask and 40 other crypto wallets. https://cointelegraph.com/news/hodlers-beware-new-malware-targets-metamask-and-40-other-crypto-wallets (2022)

Kariryaa, A., Savino, G.-L., Stellmacher, C., Schöning, J.: Understanding users' knowledge about the privacy and security of browser extensions. In: Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), pp. 99–118 (2021)

Kumar, S., Pathak, S., Singh, J.: A comprehensive study of XSS attack and the digital forensic models to gather the evidence. ECS Transact. **107**(1), 7153 (2022)

Kumar, S., Pathak, S., Singh, J.: An enhanced digital forensic investigation framework for XSS attack. J. Discr. Math. Sci. Cryptogr. **25**(4), 1009–1018 (2022)
LummaC2 Stealer: A Potent Threat to Crypto Users. https://blog.cyble.com/2023/01/06/lummac2-stealer-a-potent-threat-to-crypto-users/ (2023)
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 (2020). IEEE
Lin, Y., Wen, G., Gao, X.: Automated fixing of web UI tests via iterative element matching. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1188–1199 (2023). IEEE
MetaMask Security Monthly: December 2022. https://metamask.io/news/security/metamask-security-monthly-december-2022/ (2022)
MetaMask: The crypto wallet for Defi, Web3 Dapps and NFTs. https://metamask.io/ (2023)
Nguyen, H.V., Phan, H.D., Kästner, C., Nguyen, T.N.: Exploring output-based coverage for testing PHP web applications. Autom. Softw. Eng. **26**, 59–85 (2019)
OWASP Top 10. https://owasp.org/Top10/ (2022)
Pazos, J.C., Légaré, J.-S., Beschastnikh, I.: XSNARE: application-specific client-side cross-site scripting protection. Empir. Softw. Eng. **28**(5), 110 (2023)
Pan, J., Mao, X.: Detecting DOM-sourced cross-site scripting in browser extensions. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 24–34 (2017). IEEE
Praitheeshan, P., Pan, L., Doss, R.: Security evaluation of smart contract-based on-chain ethereum wallets. In: Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14, pp. 22–41 (2020). Springer
Phantom – Crypto & NFT Wallet – Solana | Ethereum | Polygon. https://phantom.app/ (2023)
Qaiser, S., Ali, R.: Text mining: use of TF-IDF to examine the relevance of words to documents. Int. J. Comput. Appl. **181**(1), 25–29 (2018)
Sai, A.R., Buckley, J., Le Gear, A.: Privacy and security analysis of cryptocurrency mobile applications. In: 2019 Fifth Conference on Mobile and Secure Services (MobiSecServ), pp. 1–6 (2019). IEEE
Selenium. https://www.selenium.dev/ (2022)
Security Threat Exposed for Browser-based Crypto Wallets. https://blockworks.co/news/security-threat-exposed-for-browser-based-crypto-wallets (2022)
Security Notice: Extension Disk Encryption Issue. https://medium.com/metamask/security-notice-extension-disk-encryption-issue-d437d4250863 (2022)
Slope Wallet Incident Update. 8/2/2022 https://solana.com/news/8-2-2022-application-wallet-incident/ (2022)
The Wild Crypto World in 2022: Fraud, Security Breaches & Resilient Builders. https://www.ledger.com/blog/the-wild-crypto-world-in-2022-fraud-security-breaches-resilient-builders (2022)
Trinity Attack Incident Part 1: Summary and next steps. https://blog.iota.org/trinity-attack-incident-part-1-summary-and-next-steps-8c7ccc4d81e8/ (2022)
Turan, M.S., Barker, E., Burr, W., Chen, L.: Recommendation for password-based key derivation. NIST Spec. Publ. **800**, 132 (2010)
Uddin, M.S., Mannan, M., Youssef, A.: Horus: a security assessment framework for android crypto wallets. In: Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17, pp. 120–139 (2021). Springer
Visconti, A., Mosnáček, O., Brož, M., Matyáš, V.: Examining pbkdf2 security margin-case study of luks. J. Inf. Secur. Appl. **46**, 296–306 (2019)
Wu, L., Brandt, B., Du, X., Ji, B.: Analysis of clickjacking attacks and an effective defense scheme for android devices. In: 2016 IEEE Conference on Communications and Network Security (CNS), pp. 55–63 (2016). IEEE
Year 2022 in Review - Crypto Wallet Security Incidents. https://www.certik.com/zh-CN/resources/blog/01iz10lvnaAIcuNZ2zNJqA-2022-year-in-review-crypto-wallet-security-incidents (2023)
Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., Liu, Y.: Automatic web testing using curiosity-driven reinforcement learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 423–435 (2021). IEEE
Zhao, R., Yue, C., Yi, Q.: Automatic detection of information leakage vulnerabilities in browser extensions. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1384–1394 (2015)

## Authors and Affiliations

**Pengcheng Xia[1] · Yanhui Guo[1] · Zhaowen Lin[1] · Jun Wu[1] · Pengbo Duan[1] · Ningyu He[2] · Kailong Wang[3] · Tianming Liu[4] · Yinliang Yue[5] · Guoai Xu[6] · Haoyu Wang[3]**

✉ Jun Wu
wujun@bupt.edu.cn

Pengcheng Xia
xpc357@bupt.edu.cn

Yanhui Guo
yhguo@bupt.edu.cn

[1] Beijing University of Posts and Telecommunications, Beijing, China

[2] Peking University, Beijing, China

[3] Huazhong University of Science and Technology, Wuhan, China

[4] Monash University, Melbourne, Australia

[5] Zhongguancun Laboratory, Beijing, China

[6] Harbin Institute of Technology, Shenzhen, China