RøB: Ransomware over Modern Web Browsers

Harun Oz, Ahmet Aris, and Abbas Acar, *Cyber-Physical Systems Security Lab, Florida International University;* Güliz Seray Tuncay, *Google;* Leonardo Babun and Selcuk Uluagac, *Cyber-Physical Systems Security Lab, Florida International University*

https://www.usenix.org/conference/usenixsecurity23/presentation/oz

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

RøB: Ransomware over Modern Web Browsers

Harun Oz[1], Ahmet Aris[1], Abbas Acar[1], Güliz Seray Tuncay[2], Leonardo Babun[1], and Selcuk Uluagac[1]

[1]Cyber-Physical Systems Security Lab, Florida International University, Miami, Florida, USA
[2]Google, Mountain View, CA, USA
{hoz001, aaris, aacar001, lbabu002, suluagac}@fiu.edu, gulizseray@google.com

Abstract

File System Access (FSA) API enables web applications to interact with files on the users' local devices. Even though it can be used to develop rich web applications, it greatly extends the attack surface, which can be abused by adversaries to cause significant harm. In this paper, for the first time in the literature, we extensively study this new attack vector that can be used to develop a powerful new ransomware strain over a browser. Using the FSA API and WebAssembly technology, we demonstrate this novel browser-based ransomware called RØB as a malicious web application that encrypts the user's files from the browser. We use RØB to perform impact analysis with different OSs, local directories, and antivirus solutions as well as to develop mitigation techniques against it. Our evaluations show that RØB can encrypt the victim's local files including cloud-integrated directories, external storage devices, and network-shared folders regardless of the access limitations imposed by the API. Moreover, we evaluate and show how the existing defense solutions fall short against RØB in terms of their feasibility. We propose three potential defense solutions to mitigate this new attack vector. These solutions operate at different levels (i.e., browser-level, file-system-level, and user-level) and are orthogonal to each other. Our work strives to raise awareness of the dangers of RØB-like browser-based ransomware strains and shows that the emerging API documentation (i.e., the popular FSA) can be equivocal in terms of reflecting the extent of the threat.

1 Introduction

The developers of web browsers spend significant effort on enhancing browsers by continuously adding new technologies. Web application developers take advantage of these technologies by offering new functionalities that previously could be performed only by native applications. One such technology is the File System Access (FSA) API[1], which has been developed by the Web Platform Incubator Community Group [59]. It enables web applications to interact with the users' local file systems [60]. Although not a web standard at the moment, the FSA API is embedded in and is fully supported by the most popular browsers like Chrome and Edge and is partially supported by Opera and Safari [4], which, combined, share the 91.29% of the desktop browser market as of May 2023 [7]. The FSA API has been steadily gaining popularity and is already being used by some popular web applications such as the online development platform Microsoft Visual Studio Code (i.e., vscode.dev [11]) and social media platform Snapchat [9].

Even though the FSA API can be used to develop powerful web applications, it can also be abused by adversaries to develop a novel ransomware strain as a web application that encrypts the user's files from the browser. Such an attack would effortlessly be performed by an adversary who designs a seemingly benign web application and uses malicious tactics (i.e., phishing, malvertisement) to trick the user to grant access to their sensitive portions of the local file system. Despite the briefly mentioned risks of ransomware in the FSA API documentation [61], the deployed countermeasure in its current form (i.e., hard-coded blocking system-sensitive directories) is not effective to protect sensitive user files on non-system directories, subdirectories of the systems-sensitive directories, or any other directories such as cloud-integrated directories, external directories or network-shared folders. More importantly, no prior works investigated the detailed impact analysis of this new threat vector.

In this work, we implemented a novel browser-based ransomware, namely RØB - Ransomware over Browser, that performs its malicious actions via the emerging web technologies, the FSA API and WebAssembly (Wasm). Although the security model of the FSA API suggests restricting access to some of the system directories (e.g., file system root, user's home, operating system), our experiments reveal that RØB can still encrypt files in user directories, data partitions, external storage devices (e.g., flash drives), shared network volumes, and cloud-integrated directories, making the sug-

[1]Please see for a live demo of the FSA API: `https://googlechromelabs.github.io/text-editor/`.

gested defense mechanism by the FSA API developers futile.

Antivirus (AV) software often detects ransomware by monitoring sensitive folders and identifying suspicious behaviors on the victim's computer. We performed an extensive analysis with commercial antivirus (AV) solutions such as AVG, Kaspersky, Avast, Malware Bytes, and TrendMicro. We found RØB can evade all these AVs. In addition to AVs, many highly accurate ransomware defense studies exist in the literature such as detection systems that employ static analysis or dynamic analysis features [52]. We examine their effectiveness against RØB-like ransomware; however, they, unfortunately, fail to detect RØB due to the distinct features such as not requiring any installation, running within the browser, and using Wasm-based encryption libraries. Hence, there is a need for a new solution that can effectively tackle browser-based ransomware attacks.

We propose three potential defense solutions at different levels (i.e., browser-level, file-system-level, and user-level). Our first solution, namely *malicious modification identification*, monitors the FSA API to detect malicious modifications of RØB-like attacks before they overwrite the victim's local files. Our second approach, namely *local activity monitoring*, monitors the browser's local activity (e.g., read and write function/API calls, file system activities) to detect the potential patterns of ransomware. Our third solution aims to increase the (in)security awareness of users via a *new UI design* for the FSA's permission dialog boxes. Unlike the existing dialog boxes of FSA API, the new dialog boxes we present inform users about the risks and implications of allowing web applications that utilize FSA API to interact with local files. These three proposed approaches are crucial to providing solutions to mitigate this new attack vector at different levels; however, neither of them is a panacea on its own due to the distinct features of this new attack vector. More research effort is needed to enable web applications to interact with local files in a secure manner.

Contributions: The contributions of this work are as follows:

- For the first time, we thoroughly analyzed a novel attack vector for ransomware that has not been explored before. Particularly, we show how the FSA API can be exploited to launch ransomware attacks over modern browsers.
- We conducted a comprehensive impact analysis on three different OSs, 29 distinct directories together with their subdirectories, five cloud providers, and five antivirus solutions. Our results demonstrate the limitations of traditional antivirus solutions and the ineffectiveness of the access limitation currently deployed in the FSA API.
- We evaluated the effectiveness of state-of-the-art existing ransomware detection solutions and found that they, unfortunately, fall short in detecting RØB-like ransomware due to its distinct features (e.g., no payload, no crypto library access).
- We proposed three potential defense solutions to mitigate the risks posed by browser-based ransomware: 1) Malicious Modification Identification, 2) Local Activity Monitoring, and 3) New UI Design. We implemented and evaluated the effectiveness of the first two solutions as well as provided new modified UIs to address the issues in the old UIs. To support open source and further research, we released the source code of the defense solutions [2].

Responsible Disclosure. The ransomware risk through the FSA API had been very briefly mentioned in the documentation [61]. However, we argue that the documentation, in its current form, significantly downplays the extent of the ransomware threat and gives misleading explanations regarding the efficacy of the countermeasures provided by the API. To share our findings, improve the security documentation, and contribute to the production of a working countermeasure against this threat, we contacted the FSA developers, which is Google, through several channels. We submitted a security bug to Chromium outlining the above points in detail. We also had a video meeting and email exchanges with the developers of the FSA API who gave us positive feedback strongly supporting our work and expressed interest in collaboration to implement the practical defenses we outlined in our paper. They have also agreed to make changes in the security documentation to better explain the extent of the ransomware threat based on the findings of our work. In addition, we also responsibly disclosed the issue to cloud providers (e.g., Apple, Box) whose products we identified as being at risk and AV vendors. Apple reviewed the bug report and did not take responsibility as their product was indirectly impacted by this issue. We have yet to receive any responses from the other parties at the time of this writing. Further details of the disclosure process are given in Section A in Appendix.

Ethics. Due to ethical considerations, we did not make the RØB (i.e., ransomware) implementation publicly available. And, we performed all the analysis on local servers; so no human subjects have been involved in this research.

Organization. Section 2 gives the background information. Section 3 presents the threat model. Section 4 introduces the system model and impact analysis of RØB. Section 5 investigates the effectiveness of existing ransomware defense solutions against RØB. Section 6 articulates three defense approaches we proposed. Section 7 gives the related work and Section 8 concludes the paper.

2 Background

2.1 The File System Access API

Overview. The Web Platform Incubator Community Group [59] created the File System Access (FSA) API to

[2] https://github.com/cslfiu/RoB_Ransomware_over_Modern_Web_Browsers

enable the development of powerful web applications that interact with files that are located in users' local file systems [60]. For instance, developers can build online document editors, business tools, and integrated development environments (IDE) that can directly interact with a user's local file system from the browser without any installation. Although not a web standard now, the FSA API is embedded in and fully supported by the most popular browsers like Chrome and Edge and is partially supported by Opera and Safari as of now [8]. Moreover, it is available for all browser engine teams such as Gecko of Mozilla to implement this new feature.

Internals. The FSA API has a single-entry point, named `chooseFileSystemEntries()`. This entry point opens a picker dialog allowing a user to select multiple files or directories. After the selection, the browser asks for the user's permission to read the contents of the files or directories via opening a read permission dialog. Depending on whether the user selects a file or a directory, the API returns either a `FileSystemFileHandle` or `FileSystemDirectoryHandle`. These handles provide methods to interact with the files or directories. When a web application calls the `createWritable()` method on a `FileSystemFileHandle` to modify a file, the browser prompts the write permission dialog for the user. Subsequent calls to `createWritable()` for the same file within the same session do not prompt the user for permissions again. Similarly, if the user grants both read and write permissions for a directory, the returned `FileSystemDirectoryHandle` gives access to all files and subdirectories within it. Analogous to the file handle, any modifications on the files in the same directory within the same session do not require repeated permission prompts.

Security Model. The security model of the FSA API considers a few attacks including the possibility of attacks that encrypt files (i.e., ransomware), malware storing, and execution [24]. There are two main strategies adopted by the API to tackle these attacks. First, it utilizes a permission model which requires web applications to obtain access from users via permission dialogues. This permission model is simple; there is one permission dialog for read access and another one for write access. However, the effectiveness of this model is limited as adversaries can hide their real intents and use social engineering to get a user's explicit permission [6, 28]. Second, the FSA API utilizes an access limitation strategy by blocking web applications' access to the critical parts of the file system such as the root directory as well as the user's home, operating system (OS), and browser profile directories. However, this strategy also falls short in preventing the attack we present in this work as the sub-directories of these restricted directories and any directory that are not explicitly blocked by this approach (e.g., cloud directory) can still be encrypted. We analyze the effectiveness of this approach further in Section 4.3.

Other Browsers' Positions and Concerns. Browsers such as Mozilla and Brave [19, 48] did not integrate this API into their browsers due to some concerns. Particularly, Mozilla tagged this API as harmful [48] because they do not think "...meaningful end-user consent is possible to obtain..." [48] and Brave considers this API as "non-standard privacy-risking" [19]. Additionally, Safari's WebKit recently partially implemented the FSA API in its engine [4]. However, they restricted the capabilities of FSA API by allowing only access to the Origin Private File System, which is mapped to a database outside of the user's OS. Our findings in this paper align with the concerns of other major browsers.

3 Threat Model

In our threat model, we consider a scenario where an attacker creates a malicious web application or hijacks an existing one, gaining access to the user's local file system via the FSA API. The attacker then uses phishing and malvertisement to lure victims to the web application and trick them into granting read and write access via seemingly benign web applications. Then, the adversary can use any encryption algorithm on the victims' files and overwrite the local files. For the encryption, the adversary can use an encryption library of Wasm or JavaScript or can implement the encryption algorithms by themselves. Lastly, the adversary can use various extortion methods (e.g., Bitcoin) that have been used by the classical ransomware families to obtain payment. It is worth noting that in this threat model, the FSA API works as intended, but only is abused by the attacker to get access to the user's local files and overwrite them.

Attack Practicality. As described above, the user needs to navigate to the malicious site and grant read/write access for browser-based ransomware to be effective. This threat model can be considered practical for several reasons. First, social-engineering techniques like phishing is still one of the top cybercrime used by the attacks in the wild [28] affecting major enterprises and end-users [57]. Second, the threat model of a user accessing a malicious URL [23] and a tricked user granting permission [12] have been used by other state-of-the-art studies in the literature. With the advancements in browsers and the trend of moving applications to the web, attackers may come up with more compelling strategies to successfully lure users into using their ransomware. Additionally, the average layman user can find the browser more trustworthy and grant read/write requests on file systems more inconsiderately than downloading, installing, and executing unknown software [58]. More importantly, the current UI lacks any indicator to warn users of potential ransomware attacks, increasing the likelihood of users falling for this type of attack. Last but not least, while traditional ransomware must bypass many built-in checkpoints (e.g., email attachment scanners, download scanners, local antivirus programs) throughout the attack, RØB-like ransomware can reach the victim directly without bypassing these steps, which would also increase the

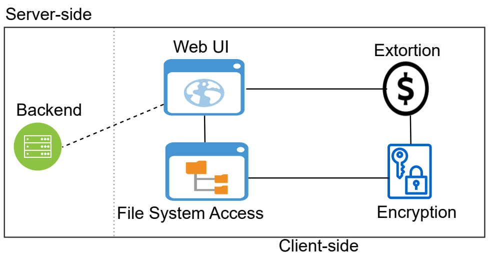


Figure 1: System model of RØB ransomware.

practicality of the threat model used in this paper.

4 RØB - Ransomware over Browser

In this section, we give the details regarding the design, implementation, and impact evaluation of our proof-of-concept ransomware implementation RØB.

4.1 System Model

Overview. Figure 1 shows the system model of the RØB that includes five modules: *Backend*, *Web User Interface (UI)*, *File System Access*, *Encryption* and, *Extortion*.

Backend Module. This module receives HTTP requests from clients (victims), it creates a public-private key pair and a unique ID for each victim. Keys and victim IDs are stored in its database. Afterward, it sends an HTTP response to the victim, which includes the other components of RØB, client ID, and the generated encryption keys for the client. The keys stored by the Backend module for each client are shared with the ones who make payments for the recovery of their files.

Web User Interface Module. This module includes the contents regarding the look of the website that aims to trick victims to enable RØB to access their local file system. The attacker can design the Web UI component differently depending on the malicious scenario. For example, this module can be designed by the adversary as a media (e.g., picture, video) editor.

File System Access Module. This module contains the necessary logic to interact with the victim's files from the web application using the FSA API. RØB works in a read-encrypt-overwrite loop for every file in the selected directory of the user.

Encryption Module. This module includes the functions/modules to encrypt the victim's files. RØB performs hybrid encryption on the victim files to make recovery attempts impossible for users. In our implementation, this module first generates a symmetric key and encrypts the victim's files with AES-256. After the encryption of all of the files, it encrypts the AES key with RSA-2048 using the public key that is generated by the Backend module.

Extortion Module. This module redirects the user to the ransom note link that informs the victim about the ransomware attack and gives details regarding the ransom payment method. RØB can employ Bitcoin as the payment method. In our implementation, this module redirects the victim to another web page that displays the victim ID assigned by the Backend module for the victim as well as the ransomware note and the associated payment details.

Algorithm 1: Algorithm of RØB.

```
1 dirHandle = window.showDirectoryPicker();
2 for entry ∈ dirHandle.values() do
3     file = entry.getFile();
4     encryptedContent = encrypt(file);
5     writable = file.createWritable();
6     writable.write(encryptedContent);
7     writable.close();
8 end
```

4.2 Implementation of RØB

File System Access Module Implementation. This module uses the FSA API to access and modify the victim's files as outlined in Algorithm 1. In Line 1, `showDirectoryPicker()` function is invoked which opens a directory select dialogue for the user. When the user selects a directory and grants the read permission, the API returns a `FileSystemDirectoryHandle` that contains the methods needed to interact with the files of the user. The statements in Lines 2-8 iterate through files in the directory selected by the user. `getFile()` method called in Line 3 returns a `FileSystemEntry` object of a file and forwards it to `encrypt()`, which returns the encrypted file contents. After, `createWritable()` in Line 5 obtains a writable stream to a file and asks for the user's permission to modify the file. Once the user has granted write permission, the `write()` method overwrites the file. Subsequent calls to `createWritable()` on the same file handle within the same session do not require additional user permission. Therefore, RØB can continue performing its malicious actions without prompting the users as long as the session continues.

Encryption Module Implementation. This module first generates a symmetric key and encrypts the victim's files with the AES-256 algorithm. It uses AES-GCM mode as it provides balanced performance and confidentiality. After the encryption of all of the files, it encrypts the AES key with RSA-2048 using the public key that is generated by the Backend module when the victim initially made a request to the malicious web application. To prevent key exposure, the module overwrites the portion of the memory where the key is stored with random values by leveraging an additional function named `clear_memory`. We implemented the Encryption module using the Enigma library which utilizes a Wasm-compiled version of OpenSSL to increase the performance [2]. We performed an investigation on the encryption speed of RØB with varying file sizes. Our results show that the encryption speed of RØB is 0.62 MB/s for a 1MB file, 3.85 MB/s for a 10MB file, and 33.2 MB/s for a 100MB file with Chrome 89 installed on a computer with 2,7 GHz Dual-Core Intel Core i5 processor and 8 GB of RAM. Although the loading process of the Wasm binaries to the browser presents a small overhead, the results show that the encryption speed of RØB scales with the size of the data.

4.3 Attack Surface Investigation

In this subsection, we perform experiments to study the impact of RØB on victim files and directories.

Evaluation Setup. We performed our first analysis on a computer (*Test-PC:1*) running Windows 10 Pro with a 2,6 GHz Intel Core i7 processor and 16 GB of available RAM. We also employed computers powered by Linux (*Test-PC:2* - Ubuntu 20.04 LTS with a 2,6 GHz Intel Core i7 processor and 16GB of RAM), and macOS (*Test-PC:3* - macOS Big Sur 11.0.1 with a 2,7 GHz Dual-Core Intel Core i5 processor and 8 GB of RAM) for our evaluations. This enabled us to understand the type of directories affected in multiple platforms used by a diverse set of users. Moreover, the analysis with network-shared folders required two Windows computers to share a folder over the network. Therefore, in addition to the *Test-PC:1*, we employed a second Windows-based computer (*Test-PC:4)* with the same specs of *Test-PC:1*.

Types of Directories Affected. In this analysis, we investigate and identify the affected directories in various operating systems (e.g., Windows, Linux, and macOS) that RØB can encrypt. We analyze the access limitation of the FSA API on a set of directories such as local directories, cloud-integrated directories, external storage devices, and network shared folders and determine if RØB can access and encrypt full directory contents or only contents of subdirectories.

• *Local Directories.* To determine the directories that are affected by RØB, we created test folders that include 50 files with different file types (`docx, xlsx, pdf, txt, jpeg`) on test computers *Test-PC:1, Test-PC:2,* and *Test-PC:3*. Subsequently, we placed the test directories in different parts of the file systems (in total 21 different directories on 3 OSs) of the test computers. On each test computer, we ran RØB as a web application using a Node.js server, browsed the RØB web application, and tried to select the directory under test that has our folder with the files.

As explained earlier, the security model of the FSA API prevents the web application from accessing specific directories. We tested those directories (i.e., the root directory of the file system, user's home, OS, and browser profile directories) on *Test-PC:1* and verified that the FSA API does not allow a user to select those directories. This implies that the security model of the FSA API prevents RØB from accessing these directories. Nevertheless, we continued our tests by placing our test directories in other parts of the local file system to choose such directories via RØB. We realized that RØB can access and encrypt the full contents (including subdirectories)

Table 1: List of local directories in different operating systems that RØB affects. FDA: Full Directory Access. SDA: Subdirectory Access. ✔ indicates that RØB has FDA/SDA and encrypts files in that directory. ✖ indicates that access is denied by the API and RØB cannot encrypt files in that directory.

	Windows		**Linux**		**macOS**	
Directory	***FDA***	***SDA***	***FDA***	***SDA***	***FDA***	***SDA***
Documents	✖	✔	✖	✔	✖	✔
Desktop	✖	✔	✖	✔	✖	✔
Pictures	✔	✔	✔	✔	✔	✔
Videos	✔	✔	✔	✔	✔	✔
Music	✔	✔	✔	✔	✔	✔
Downloads	✖	✔	✖	✔	✖	✔
Data Partition	✖	✔	✔	✔	✔	✔

of the Windows directories of Pictures, Videos, and Music. In addition, we found out that although the security model of the FSA API does not allow a user to choose the Documents, Desktop, Downloads, and the data partition directories (e.g., D:/), there is no access limitation dictated by the API on their subdirectories.

To summarize, our evaluation showed that RØB can arbitrarily encrypt the full contents of Pictures, Videos, and Music, and the subdirectories of Documents, Desktop, Downloads, and data partition directories on a Windows computer. Considering the other test computers (i.e., *Test-PC:2* and *Test-PC:3* running Linux and macOS respectively), our evaluation showed that the FSA API has very similar access limitations for web applications (hence for RØB) on computers powered by these operating systems. Table 1 provides a summary of our findings. Once RØB can have full directory access or subdirectory access, it encrypts all of the files inside. Although we see similar full and subdirectory access patterns of RØB for most of the local directories of Windows, Linux, and macOS, we see an interesting pattern for data partitions. While the FSA API prevents RØB from fully accessing the data partition and allows only subdirectories to be accessed for Windows, it allows full access to data partition and subdirectories in both Linux and macOS platforms.

• *Cloud-Integrated Directories.* To analyze the effects of RØB on the files that are stored in cloud-integrated directories, we first created dummy accounts on popular cloud platforms such as Google Drive, DropBox, Box, iCloud, and Microsoft One Drive. These platforms have desktop applications that allow users to work on the files locally and sync them to the cloud using file system integration. We downloaded their desktop applications to *Test-PC:1* to enable that feature and create their integrated directories. Subsequently, we placed 50 files with different file types in each cloud-integrated directories on *Test-PC:1* and chose these directories as the target while running RØB.

Our experiments show that when RØB accessed these directories, it successfully encrypted the files in those cloud-integrated directories. After that, the sync engine of the cloud provider reflected the modifications to the cloud storage and the changes made to the files in the cloud became persistent. We summarized our findings in Table 2.

We note that even though these platforms have their built-in ransomware detection mechanisms, none of the cloud platforms were able to detect RØB during or after the attack. On the other hand, as a ransomware protection method, some cloud platforms use versioning techniques [10]. If the files that are stored in the cloud are attacked by ransomware, the cloud user can retrieve the earlier (unencrypted, original) version of the files. However, this feature is not standard among all of the cloud providers and every cloud vendor has its own versioning scheme. While cloud providers such as Dropbox keep track of versions of files for basic, plus, and family users for 30 days and professional and business users for 180 days [3], OneDrive of Microsoft keeps 25 versions of the stored files [50]. Differently, Google Drive can keep versions of files for 30 days or the first 100 versions of them [10]. Box has different solutions with various versioning features. Specifically, while Box Individual keeps only one version of a file, Personal Pro and Business Starter solutions keep 10 and 25 file versions, respectively. On the other hand, iCloud does not utilize the versioning feature. We further investigated the versioning scheme of the cloud providers and realized that Dropbox, Google Drive, and Microsoft OneDrive can store more than 100 versions of the files and RØB cannot have a permanent effect on the files stored by them. In addition, Box Individual and Apple iCloud do not have a versioning scheme and we verified that the files encrypted by RØB cannot be recovered by such solutions.

We conclude that if victims are using iCloud or Box Individual, RØB can cause them to lose their significant files unless a ransom is paid. RØB-like ransomware attacks can create significant damage to iCloud users. In addition, although the versioning strategy employed by Dropbox, Google Drive, and One Drive seems to be resilient against RØB, it should not be considered as a silver bullet. This is due to the fact that the backup files in the version history do not always reflect the most recent state of a file. If RØB is launched before a backup is done, then critical changes on user files can still be lost.

• *External Storage Devices.* To test the impact of RØB on external disks, we placed a test directory that included 50 files with different file types in a test external disk (e.g., Western Digital 4TB) and a test flash drive (e.g., Toshiba 16GB) connected to *Test-PC:1*. After, we ran RØB and chose the directory in each external storage device as the target.

Our experiments showed that RØB is able to encrypt all of the files located in the selected directories in each external storage device. As external storage devices are used by ordinary users and enterprise users to backup important data, RØB can have detrimental effects on the files stored in such devices. Unfortunately, such devices are outside the scope of the security model of the FSA API which leaves them prone to RØB-like ransomware attacks.

Table 2: Cloud providers, their versioning schemes, and the impact of RØB where ✔ signifies that files are not recoverable after the encryption of RØB and ✖ signifies that files are recoverable after the encryption of RØB due to versioning feature of the cloud provider. However, if RØB is launched before a backup is done, then critical changes on user files can still be lost for cloud providers with versioning (i.e., Google Drive, Dropbox, and Microsoft OneDrive).

Cloud Provider	Versioning Scheme	Affected by RØB?
Google Drive	30 days or 100 versions	✖
Microsoft OneDrive	25 versions	✖
Dropbox	30 days (personal), 180 days (business)	✖
Apple iCloud	No versioning	✔
Box Individual	No versioning	✔

• *Network Shared Folders.* To test the effect of RØB on network shared folders, we created a test directory that included 50 files with different file types in our test computer *Test-PC:1* and we shared it over the network with *Test-PC:4*. After that, we run RØB on *Test-PC:4* and selected the test directory that is shared over the network as a target.

We observed that RØB is able to encrypt the files that reside in the shared folder which shows that running RØB in one computer can affect the folders/files that are shared by multiple computers over the network. Shared folders are frequently used by both individuals and enterprises and if sensitive/important files are stored in the shared folder, the effects of RØB on these folders can be very serious.

4.4 Desktop vs. Browser-based Ransomware

In this section, we discuss the fundamental differences between browser-based ransomware and desktop ransomware.

Initial User Access. Desktop ransomware typically spreads via phishing, advertisements, or emails, which are used to trigger downloading the malicious payload. In comparison, browser-based ransomware, such as RØB, is a malicious web application that needs to attract its victims to its domain in some way. Similar methods such as phishing, malvertisements, or emails (as a link but not as an attachment) can be used by RoB to distribute the link and to gain initial user access. To attract more users, it can also be designed as a benign-looking web application (e.g., a free media editor).

Infection and Execution. For desktop ransomware to be effective, the user (victim) must download (i.e., infect) and execute the binary on its system. Unlike desktop ransomware, RØB is fileless, i.e., no download or execution is required. However, after luring victims to its domain, browser-based ransomware still needs to trick its victims into granting read and write access. In this manner, browser-based ransomware has the advantage of executing its actions via browsers, which are used by millions of users and perceived as trustworthy while desktop ransomware requires users to download and execute unknown binaries.

Encryption. RØB uses Wasm to encrypt the victim files. As it employs all of the encryption logic in the Wasm, unlike desktop ransomware, it does not utilize the platform's encryption libraries or OS system calls. Hence, it can hide its file encryption process from the defense solutions that monitor the system calls made to the platform's encryption APIs.

Extortion. Desktop ransomware delivers ransom payment information to the victim via leaving a ransom note on the desktop, changing the desktop background, or using lock screens. However, RØB cannot employ those techniques and has to find other ways. Particularly, RØB can: A) redirect the victim to the extortion page upon finishing encryption, B) add ransom note files to the parent folders of the encrypted files, C) change the names of encrypted files and add ransom note to file names by adhering to maximum path length constraints of platforms. If Option A is employed, then the ransom note may never reach the victim if the victim closes the page before the encryption is completed. Options B and C are better ones, ensuring that the ransom note will be reachable to the victim even if the page is closed. However, those options may give clues to defense solutions in detecting RØB as discussed in Section 6.

5 Effectiveness of Current Defenses

In this section, we investigate the effectiveness of existing ransomware defense solutions against RØB.

5.1 Antivirus Solutions

In this section, we tested the effectiveness of the full versions of five different antivirus solutions, namely Malware Bytes, AVG Antivirus, Kaspersky, Trend Micro, and Avast against RØB. We chose these antivirus solutions because they explicitly promise ransomware defense via malicious behavior monitoring for users. To perform our experiments, we downloaded and installed each antivirus solution to our test computer *Test-PC:1*, put test directories with 10, 50, and 100 files with various file types, and checked if they can detect RØB when it is instructed to run on the test folder.

• *Malware Bytes.* Malware Bytes Premium [43] promises to protect users' documents and financial files against ransomware. We installed Malware Bytes Premium and selected the test directory when running RØB. We verified that Malware Bytes Premium could not detect RØB. Moreover, we also tried the browser extension of Malware Bytes, namely Browser Guard [42] which blocks malicious web pages and web applications that include ransomware. We performed another test while Malware Bytes's Browser Guard extension is activated and it could not detect RØB.

• *AVG Antivirus.* We installed the full version of the AVG Internet Security [16], which promises ransomware protection. AVG Internet Security monitors sensitive folders such as Documents, Pictures, etc., and allows a user to add a folder to the sensitive folders list for monitoring. We added the path of the test directory to the sensitive folders list. Following that, we ran RØB and chose the test directory as the target. AVG Internet Security was not able to detect RØB.

• *Kaspersky.* We installed the full version of Kaspersky Total Security [5]. Kaspersky Total Security has an anti-ransomware tool that monitors the personal computer for ransomware-like behavior in real time. Nevertheless, we ran RØB and could successfully encrypt the files within the test folder. Kaspersky Total Security was not able to detect RØB in any of the test cases.

• *Trend Micro.* We installed the full version of Trend Micro Antivirus+ Security [56], which promises to utilize prevent various threats such as ransomware and online attacks via its Advanced AI Learning utility. We performed three tests with 10, 50, and 100 files in the test folder. We confirmed that Trend Micro Antivirus+ Security was not able to detect RØB in any of our test experiments.

• *Avast.* We installed the full version of Avast One Essential [15], which promises to provide ransomware protection via monitoring important folders. We added the path of the folder among the important files/folders to be monitored. We verified that Avast One Essential was not able to detect RØB in any of our test cases.

5.2 State-of-the-art Ransomware Defense

The ransomware defense approaches for PCs can be grouped into three categories: 1) Static analysis-based detection methods, 2) Dynamic analysis-based detection methods, and 3) Key extraction-based recovery solutions.

Static Analysis-based Solutions. Many researchers proposed static analysis-based solutions [45, 64] that utilize structural features such as strings and opcodes to detect ransomware. Although those solutions can detect well-known ransomware strains, they are vulnerable to common evasion attempts such as obfuscation [17,53]. In the concept of browser-based ransomware attacks, the adversaries are free to use any tool available and employ obfuscation techniques to evade all types of static analysis-based tools. Therefore, such solutions are not suitable for browser-based ransomware attacks.

Dynamic Analysis-based Solutions. The dynamic analysis-based solutions use behavioral features such as network activity, API/system calls, I/O access patterns, and file system activity to detect ransomware [52]. First, RØB does not need frequent C&C server communication. In fact, only one HTTP request made to the Backend module of RØB is sufficient for it to be sent in an HTTP response packet and perform its malicious actions. In addition, RØB's communication is based on HTTP over TCP which is used by almost every benign website and web application. Therefore, the solutions that use network traffic features [21, 22, 47] would struggle to detect RØB. Second, unlike conventional ransomware, RØB can perform malicious actions without being installed on the system. Therefore, it can evade the registry-based solutions [35, 37]. Due to the high computation cost of the malware analysis environments [51], the ransomware analysis environments such as [25, 38] (albeit was very useful) has become impractical against RØB-like attacks as it is not practical to analyze every website before the visit of a user in such analysis environments. Additionally, ransomware defense solutions such as [25,39,54] utilize the features retrieved from the file system activities such as folder listing, files written, read, renamed, and deleted. These defense solutions have been designed by monitoring the file system activities performed by the process of the ransomware executable. Nevertheless, the file system activities of RØB are different from the traditional ransomware defense solution (see Section 6 for a detailed explanation). So, these solutions will not be effective to detect RØB. Furthermore, defense solutions such as [39, 54] include the browser as a benign web application, so that they will introduce a false positive in detecting RØB with their current implementation.

Moreover, RØB uses browsers to perform its malicious actions. Running on the browser without being installed on a system can create additional challenges for API/system call monitoring solutions [13, 40, 44, 65]. Considering RØB, such solutions face two difficulties: 1) In RØB, the adversary is free to embed his own encryption code to the Encryption module which will not use the crypto APIs of the OS. 2) RØB runs in the browser (a benign program) and monitoring the system calls made by the browser will introduce an overhead, as each website visit creates many browser processes and users can have multiple tabs open. Therefore, monitoring API/system calls of browser processes would incur high overhead. Additionally, monitoring the system calls can considerably slow down the process [63] which can impact the browsing experience. For these reasons, API/system calls monitoring solutions will not be practical and effective in detecting RØB.

Key Extraction-based Solutions. Some ransomware defense systems use memory forensic techniques to retrieve the keys of the attacker to recover the files. The study in [40] presents a ransomware recovery mechanism that stores encryption keys by hooking the crypto functions of the OS. Similarly, the study in [46] combines process monitoring, and file change monitoring to detect ransomware and hook crypto API functions to retrieve the key. While these approaches are effective on ransomware families that use Crypto APIs of the OS, they will be ineffective against RØB-like attacks since it does not use crypto APIs of the OS. Differently, in [33], the authors aimed to restore the files encrypted by ransomware utilizing behavioral features such as encryption time and backup damaging behavior. The considered ransomware needs privileges to perform its malicious actions; however, RØB does not need any privileges to perform the attack.

To test the feasibility of key extraction from RØB, we created a Node.js script utilizing puppeteer [1] to periodically

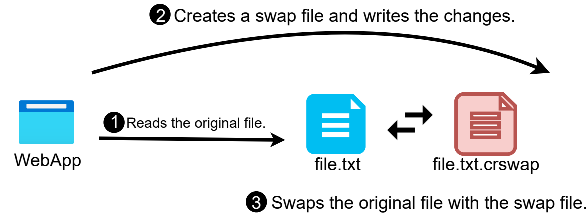


Figure 2: Behavior of RØB using the FSA API.

capture heap snapshots of the web application. We performed two experiments. First, we ran RØB on a test directory and retrieved two heap snapshots: one during the attack and another afterward. Second, we adjusted our script to continuously capture heap snapshots of RØB every 5 seconds, retrieving a total of 4 different snapshot files. We inspected all output files to search for our predefined key. While we did not encounter the key in the files from our first experiment, we detected the key in a single file from our second experiment. The focus of these experiments was the potential extraction of a raw key during a browser-based ransomware attack. Nevertheless, if the intermediate key representations (e.g., AES T-tables) are detected in the snapshot, it would also be sufficient to enable the recovery of the key as well. Our experiments reveal that extracting the key during a browser-based ransomware attack is feasible, but it is not practical. Firstly, taking heap snapshots (each snapshot is $\simeq$4.8MB) of every website the user is visiting and storing them for further analysis requires a huge memory and may potentially affect the user experience. Additionally, RØB can solely utilize the RSA public key encryption to encrypt each file, potentially evading this type of defense solution.

6 Potential Defense Solutions

In this study, we propose three different defense solutions that are based on the above-mentioned approaches to mitigate this new attack vector at different levels and we implement a proof-of-concept design for each proposed defense solution. In the next subsection, we first explain the details of these approaches and present proof of concept implementations.

6.1 Approach 1: Malicious Modification Identification via API Hooking

In this approach, we aim to find indicators that would effectively *identify malicious modification*, hence signal the presence of RØB-like attacks.

Stopping the attack. We show the file system activities of RØB in Figure 2. In a read-encrypt-write loop, it reads the files in the accessed directory one by one ❶. After that, it creates a swap file for each file in the directory and writes the modifications back to the related swap file. We found that the FSA API names the swap files by appending the `.crswap` extension to the original files ❷. We realized that creating swap files with `.crswap` extension for every modified file was not documented by the API and it is one *unique* and *obscure* behavior of the API while working on the local file system. Lastly, as represented in ❸, it swaps the content of the original file with the swap file and the change made by the web application becomes permanent. Note that the modifications made by RØB are not permanent until step ❸. Therefore, intercepting the activity of a web application in this step prevents RØB from making permanent changes to the original files and stops the attack.

Implementation. Based on our analysis, we found that hooking the specific functions of the FSA can stop the activity of the RØB before it makes a permanent change to the user files. For this, we implemented a hooking script in JavaScript. While the user is using the browser, the hooking script continuously runs in the background and simultaneously checks if critical functions of the FSA API are called by a web application. When the FSA API functions such as `ShowDirectoryPicker`, and `write` are called by a web application, the hooks of these functions are activated and the activity of the web application is monitored and stopped by this module before permanent malicious changes are made on the user files. Specifically, hooking into the `ShowDirectoryPicker` function allows us to get the directory path that a web application is accessing via FSA API.

Our implementation considers two patterns of ransomware: 1) read-encrypt-overwrite, and 2) read-encrypt-delete-write. Considering the first pattern, hooking into the `write` function stops the web application that is overwriting a file in the local file system, thus allowing to check the created swap file. In terms of the second pattern, hooking into `removeEntry` function of the API allows us to prevent a web application from deleting a file. Since some ransomware families do not work in a read-encrypt-overwrite pattern and instead delete the original file and write the encrypted version of the original file in a new file. For this type of ransomware, the hooking module detects a delete activity, pauses the deletion event, and continues to monitor if the web application attempts to create a new file. In this special case, the `write` function hooking is activated and enables us to check the created swap file.

Identifying Malicious Modification. To detect a RØB-like ransomware attack, the intent of the modification made by a web application to a user's files must be identified. To achieve that, we identify two indicators, *entropy change and file size change*. Both the original version and modified version of the file can be obtained at the same time by pausing the activity of the web application in step ❸ as depicted in Figure 2.

• *Entropy Change.* Since files with high entropy can indicate the file is encrypted, many ransomware defense solutions used entropy to identify encrypted files [52]. Particularly, such studies mark a file as encrypted if the entropy of the file is above a threshold. Different from those ransomware solutions in the literature, we take advantage of having both versions of the file and use the entropy change after modification as a feature instead of threshold comparison. Compared to any benign file modifications, encryption operation triggers a bigger entropy change on files. We tested and verified this hypothesis by performing in total 500K benign and 500K malicious (encryption) modification operations on 5000 files with various file types (`txt, jpeg, docx, pdf, xlsx` in a dataset of files obtained from [31]. We provided more detailed explanation of dataset collection in Section C in Appendix.

Our analysis with 1 million modified files shows that benign modifications result in very small changes in entropy. For instance, the entropy of `txt, xlsx, jpeg, docx` files increases by 0.05 on average after benign modifications. However, malicious modifications (encryption) on files can result in large changes in entropy. For example, after encryption, the entropy of files increase by 3.5 for `txt`, 0.10 for `xlsx`, 0.60 for `jpeg`, 0.60 for `docx`, and 0.10 for `pdf` files on average. For this reason, we utilize entropy change as a feature to detect RØB-like attacks in this solution.

• *File Size Change.* We also observed that the file size change between the original file and its modified version is another indicator to detect the malicious/benign intent of the modification. Since the encryption operation does not expand the data included in the file, the size of the file remains relatively similar after the encryption. On the other hand, benign modifications change the size of the file relatively more than the encryption operation.

Our analysis with 1 million modified files shows that benign modifications on files result in significant changes in size. For instance, the size of `txt`, `xlsx`, `jpeg`, and `docx` files increases by 15% ($\simeq 300Kb$) on average after benign modifications. However, malicious modifications (encryption) on files can result in smaller changes in size compared to benign modifications. Particularly, our analysis shows that size of file size changes 0.002% for `txt`, 0.06% for `xlsx`, 0.14% for `jpeg`, 0.012% for `docx`, and 0.006% for `pdf` files on average after the encryption. These results show that the file size chance is another effective feature to identify malicious modifications.

Classifier Evaluation. To test the effectiveness of this approach, we created a machine learning classifier that takes the entropy change and the file size change of the files as features and identifies the malicious modifications.We implemented the classifier using Python's scikit-learn library. We trained the classifier with a dataset that includes the features of original files and their artificially modified versions and encrypted for various file types. To prevent overfitting, we utilized a 10-fold cross-validation. We measured the performance metrics for various classifiers that were previously used in the various malware detection systems [52] such as Random Forest (RF), K-nearest Neighbor (KNN), Decision Tree (DT), and XGBoost. Table 3 presents the results for the evaluation of the efficacy of our first approach in identifying encryption on the user files. We observed that the RF classifier outperforms other classifiers in the context of identifying encrypted `txt` and `docx` files by introducing only one false positive (FP)

Table 3: Performance evaluation of different ML algorithms.

Model		Acc.	Recall	Prec.	F1	TP	TN	FN	FP
RF	TXT	0.99	0.99	0.99	0.99	99997	99999	3	1
	PDF	0.99	0.99	0.99	0.99	100000	99981	0	19
	JPEG	0.99	0.99	0.99	0.99	99996	99996	4	4
	DOCX	0.99	0.99	0.99	0.99	99997	99999	3	1
	XLSX	0.99	0.99	0.99	0.99	99938	99710	62	290
KNN	TXT	0.99	1	0.99	0.99	100000	99982	0	18
	PDF	0.99	0.99	0.99	0.99	99988	99990	12	10
	JPEG	0.99	0.99	0.99	0.99	99995	99989	11	5
	DOCX	0.99	1	0.99	0.99	100000	99982	0	18
	XLSX	0.99	0.99	0.99	0.99	99692	99888	308	112
DT	TXT	0.99	0.99	0.99	0.99	99998	99995	2	5
	PDF	0.98	0.99	0.99	0.99	99951	99981	49	19
	JPEG	0.98	0.99	0.99	0.99	99992	99992	8	8
	DOCX	0.99	0.99	1	0.99	99996	100000	4	0
	XLSX	0.99	0.99	0.99	0.99	99700	99942	300	58
XGB	TXT	0.98	0.99	0.99	0.99	99997	99991	3	9
	PDF	0.99	1	0.99	0.99	100000	99983	0	17
	JPEG	0.99	0.99	0.99	0.99	99995	99991	5	9
	DOCX	0.99	0.99	0.99	0.99	99998	99995	2	5
	XLSX	0.99	0.99	0.99	0.99	99710	99935	290	65

case. In the case of `jpeg` files, the DT classifier presents the best performance by presenting 99.5% accuracy without introducing any FP. Finally, KNN achieves the best accuracy performance on `xlsx` files introducing only two FP cases.

Evaluation Against Adaptive Attackers. In this approach, we showed that entropy and file size changes are simple, yet effective features in encryption detection. However, To analyze the impact of these evasion techniques on the different file types, we created a new dataset, which includes 500 distinct files (100 per file format) using these evasion techniques. We then retrained our classifier with this new dataset, by considering each evasion technique. A detailed description of the dataset generation procedure and classifier selection is given in Section D in Appendix.

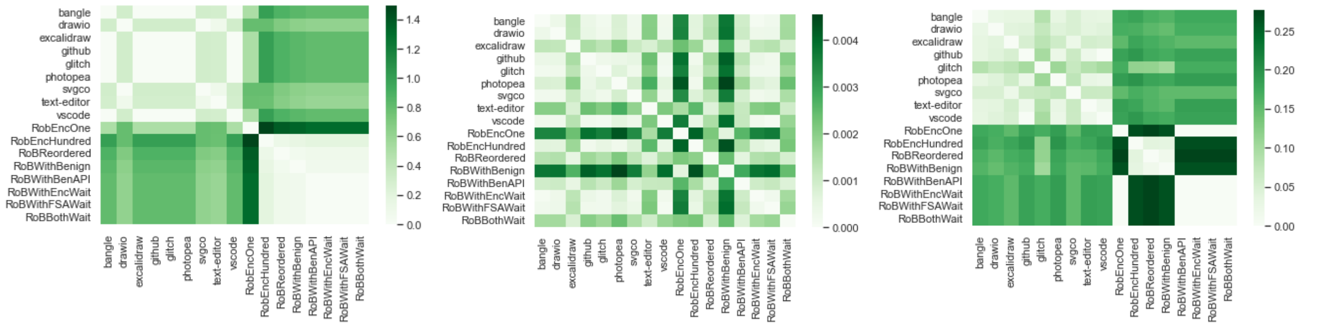
Our results showed that entropy is closely proportional to the encryption ratio in the partial encryption technique. In terms of detection, our classifier introduced 8 FN cases in total for 500 files with 25% partial encryption. Furthermore, injecting low-entropy data led to a significant size increase and decrease in entropy in all file types. We observed that injecting low-entropy data padding introduced 8 FN cases in total. Finally, encoding over encryption techniques had a deterministic impact on the entropy by setting a fixed entropy value across all file types. Specifically, Base64, Base32, and hexadecimal encoding set the entropy to 5.99, 5.0, and 3.99 respectively, and increased file sizes by 33%, 60%, and 100%. Hexadecimal encoding was most effective against our classifier, introducing 12 FN cases in total. Our experiments revealed that while these techniques alter both file size and entropy in ways that can make detection more challenging, they fall short of fully mimicking the characteristics of benign file modifications to completely evade our classifier. With this insight, we utilized a custom evasion technique that combines both data padding and partial encryption to perfectly mimic benign modifications made by the user. As the entropy and size changes in the resulting files closely resemble those in the modified files, this combined technique successfully evaded our classifier, introducing 454 FN cases in total.

Usability & Discussion: Hooking the web applications that use the FSA API prevents permanent malicious modifications before they overwrite the user files. Our current implementation shows entropy change and file size change are effective in identifying malicious modifications but introduce false negative and false positive cases in case of evasion attempts. Although these attempts are theoretically possible, they come with a cost and increase the complexity of the attack. Regarding the usability of our first approach, false positive cases might be introduced by benign web applications performing heavy compression/encryption as part of legitimate operations. To mitigate this, an alerting module can be implemented to warn the user about potential malicious file changes performed by the browser. This alerting module can launch a third dialog box that explicitly mentions detected potential malicious actions (similar to AVs detecting malicious files) and the risks of permanent data loss. This new dialog box would prompt the users to verify the accessed website and offer the option to proceed or cancel the modification. Although this might introduce an additional inconvenience due to the frequent permission dialog prompt, it would help reduce false positives. Finally, the extortion method of the RØB can also be considered to increase the effectiveness of our second approach. For example, the created files by web applications and their names can be monitored for the creation of ransom notes.

6.2 Approach 2: Local Activity Monitoring

Our second approach to prevent RØB-like attacks employs local activity monitoring of web applications that use the FSA API. Such an approach can be implemented to monitor the following local activities: 1) the FSA API function calls, 2) browser process system calls, and 3) file system activities.

Data Collection. For the benign dataset, we created test folders and accessed them via benign web applications. We performed benign operations such as editing the files, removing/adding the content from the file. For malicious dataset, we implemented RØB with different configurations. To start with, we created two non-adaptive configurations. The first one encrypts a single file (i.e., RØBEncOne) while the second one encrypts 100 files in a single directory (i.e., RØBEncHundred). We also created six more adaptive attacker configurations, which will be explained later in this section. To implement this approach, we first created a script in Node.js that hooks every available function of the FSA API and logs the called functions. We used those FSA API function calls as the first feature. Second, while interacting with the file system, we



(a) The FSA API Function Calls (2-gram)

(b) System Calls (4-gram)

(c) File System Activities (1-gram)

Figure 3: The heatmap plots for the similarity matrices of the features used in the local activity monitoring. The darker marking means the pair is more differentiable.

retrieved the `PID` with `lsof` and monitored the system calls made by the process by `strace`. We used the system calls as the second feature type. Finally, we collected the file system activities via instrumenting `inotifywait`.

FSA API Function Calls. The FSA API implements several functions (`getFile`, `write`) that can be used by web applications to interact with the local files of users. We hypothesize that the sequence of function calls of RØB-like ransomware attacks and benign web applications display distinguishable patterns, which can be used to detect the attacks. API call monitoring has been successfully used by many malware detection studies [26]. For example, Windows API call sequences are considered one of the representative characteristics in behavior-based malware detection [27]. Our method differs from those studies because the FSA API function calls are specific to RØB-like ransomware; thereby never have been analyzed.

Our initial analysis on the dataset showed that `getFile()` → `Write()` → `Write.close()` patterns occur for both benign web applications and RØB due to the natural usage of the FSA API. However, we observed that while this pattern repeats once for every file in the test folder for RØB, the pattern is repeated multiple times for the benign web application as the user performs multiple changes on the files. This is expected as ransomware is incentivized to encrypt as many files as possible while users of benign web applications are expected to work and modify a single file multiple times.

System Calls. Although RØB uses Wasm for encryption and does not employ the crypto APIs of the platform, it would still be possible to monitor the other system calls made by that browser process while interacting with the file system. We hypothesize that the system calls made by web applications through the browser can be used to differentiate benign and malicious RØB-like web applications. The system call monitoring has been successfully utilized by numerous different types of malware detection methods in the literature [26]. Our study differs from those studies as we only monitor the browser's system calls.

We first manually inspected the system calls made by RØB and vscode as an example. We observed that vscode uses a significantly higher number of write-related system calls than RØB. We found that while RØB's system calls are uniformly distributed, vscode's system calls are randomly distributed for each file in terms of the file size change. That is, every time a location in memory is accessed by a benign application, varying sizes of changes are performed while the malicious applications apply almost the same amount of change every time a location in memory is accessed.

File System Activities. File system activities are utilized by ransomware detection mechanisms in the literature [52]. However, none of these approaches focus on the file system of activity of the browser. To minimize the overhead, this approach can benefit from API hooking and identification of the portion of the file system accessed by the web application. Hence, only the corresponding portion of the file system that the web application accesses can be monitored for low-level file system activities. We first analyzed the file system activities of RØB and vscode manually. Our analysis showed that patterns of file system activities are generated only once for each file on the test folder for RØB, whereas we see the occurrence of the pattern multiple times for individual files for vscode. This observation can be used to detect RØB-like ransomware attacks.

Evaluation Against Adaptive Attackers. To evaluate Approach 2 against adaptive attackers, we created six different versions of RØB: 1) RØBReordered: changes the order of the FSA API calls randomly, 2) RØBWithBenign: adds benign modifications (e.g. writing) between the encryption operations, 3) RØBWithBenAPI: makes benign API calls (e.g., battery status), 4) RØBWithEncWait: waits a random amount of time during the encryption process, 5) RØBWithFSAWait: adds random time intervals between the FSA API calls, and 6) RØBWithBothWait: adds random time intervals both during encryption and between the FSA calls.

N-gram Analysis. In this part, we analyze nine benign and eight (two non-adaptive and six) malicious web applications using n-gram analysis. We calculated the features using the 10% quantile ranges and used Euclidean for the distance calculation. We presented our results as heatmaps in Figure 3. We found that web applications generate 33 FSA API function calls on average while they generate 15k system calls and 8k file system activities on average. We experimentally adjusted the value of n for each feature. We observed the best results for 2-gram in FSA API function calls, 4-gram for system calls, and 1-gram for file system activities.

Overall results show that while ransomware samples are clearly differentiable using the FSA API function calls and file system activities, they are less differentiable with the system calls. For example, a threshold-based detection system based on system calls would likely miss the ransomware sample encrypting 100 files in a directory (i.e., RØBEncHundred). The reason for this is that the impact of encrypting different files cannot be observed in the system calls; therefore, encrypting multiple files is creating a benign-looking behavior, which is similar to a user modifying a file multiple times. While the FSA API function calls and file system activities features contain the modified file, which makes the multiple files encrypting ransomware even easier to detect. On the other hand, our results show that re-ordering the API calls, adding benign API calls, or random waiting strategies do not have any impact on FSA API function calls and file system activities since other benign API calls and timestamps of the API calls are not considered during the feature extraction. We also observed that out of all adaptive strategies, the strategies involving additional waiting time affect the system calls feature significantly, resulting in many false positives. Consequently, a threshold-based detection system using system calls could fail to detect ransomware samples such as RØBWithEncWait, RØBWithFSAWait, and RØBWithBothWait. Similarly, such a detection system would also misclassify some benign web applications like photopea, text-editor, or excalidraw.

Usability & Discussion. In this approach, we showed the feasibility of local activity monitoring to detect RØB-like ransomware. While monitoring the local activity of the web applications benefits from hooking to minimize the overhead, as we observed in our evaluation against multiple types of adaptive attackers, adversaries can cause false results by changing the implementation of RØB to call the other functions of the API, make redundant system calls, or make a few small changes on the files before encryption. On the other hand, benign applications such as cloud storage services (Google Drive, Dropbox, OneDrive), online code editors (GitHub, VS Code), data processing tools (e.g., machine learning applications), and batch file conversion tools may perform mass modification on multiple files, similar to the patterns of RØB. In such cases, browser vendors might consider integrating security alerts after a certain amount of modification with clear information about the nature of the threat. So, users can make informed decisions. This security alert may include information about the application requesting access, the types and quantity of modifications being made by the web applications. Additionally, browsers might define a threshold for the number of allowed modifications before additional user intervention is required. This threshold could be determined based on typical usage patterns to minimize false positives.

6.3 Approach 3: New UI Design

In this approach, instead of detecting the malicious activity of RØB, we aim to raise the security awareness of the users and better inform them about the risks of allowing web applications to interact with local files.

Current Permission Boxes. The current dialog boxes implemented in Chromium are shown in Figure 4 and 5. In the permission dialog box demonstrated in Figure 4, the web application asks for a permission to read the contents of all of the files inside the directory selected by the user. In the permission dialog box presented in Figure 5, the web application asks for a permission to be able to (over)write all of the files inside the directory picked by the user.

Issues. We found the following issues in the current permission dialog boxes. First, they do not clearly state the risks of approving the permissions. For example, the current read access permission box does not have any indicator for the potential information disclosure of user-sensitive files. Similarly, the current write access permission box does not have warn for the risks of permanent data loss. Second, despite their different capabilities read and write permission dialog boxes look very similar. The users may mistakenly click one another and give access to a web application. Third, the changes made by the web application are not explicitly given in the write dialog box to help the user while accepting the permanent changes. Fourth, since it is not stated in the current permission dialog boxes, the user may not be aware of the fact that the web application will able to access the subdirectories inside the selected directory too.

Design Decisions. Designing user interfaces for permissions to ease security decisions is crucial. We define our design decisions by applying guidelines from the state-of-the-art studies [12, 20, 29, 49]. We note the following:

- The proposed interface must explicitly show the risks. In [20], it has been shown that the users are likely to grant dangerously excessive permissions when previous instances of them have not contained reason for concern. For this, the user interface can include words like "sensitive information disclosure", "permanent loss". To further mitigate the ransomware threat in the write permission box, the keyword can also be like "encryption" or "ransomware".
- The proposed user interface must be designed with colors stated in the previous studies [49] that would effectively capture the attention of the user. The warning icons have been shown to be effective in the context of connection security [30] to attract the user's attention.
- Permission dialog boxes must not be identical so that the user would know the difference for each box.
- The proposed new user interface must show all of the accessed/changed files for a meaningful user decision [48].

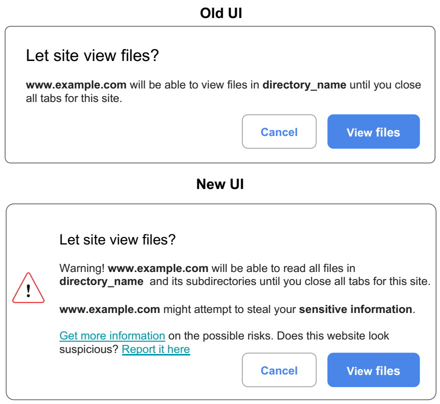


Figure 4: Read access permission box (old and new).

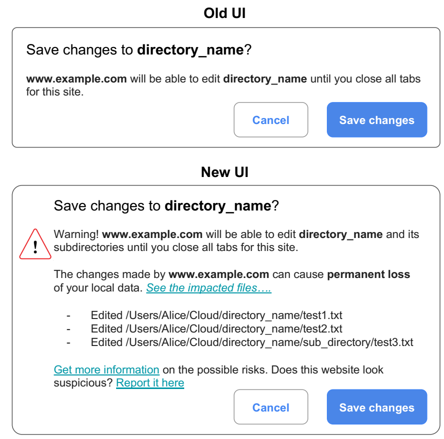


Figure 5: Write access permission box (old and new).

• The proposed new user interface must include a link that redirects the user to a web page for more information about the API and its risks.

Improvements Over Old UI. With these design decisions, we proposed two new UIs as shown in Figure 4 and Figure 5 for the read and write access, respectively. In the new UIs, we added the warning icon, the keyword "subdirectory" in the explanation, and a hyperlink to get more information about this API and its risks. Also, in the write box, we added the keywords such as "permanent loss" and an option allowing users to see which files were modified by the web application.

Usability & Discussion: The key benefit of the new UI design approach is its seamless integration with the existing permission dialog boxes, without incurring any additional overhead. The new UI clearly outlines the capabilities of web applications and any potential malicious intent they may possess. Furthermore, to enhance both the effectiveness and usability of this approach, the new UI could incorporate animations and avatars to boost user engagement and understanding [36], incorporate multilingual support, and integrate with accessibility technologies [18]. The new UI design can be integrated into the API's source code, this approach directly impacts every web application utilizing the FSA API, i.e., no installation required. Although redesigning user interfaces can help in protecting users from various attacks, an attacker can still gain the user's trust via malicious tactics [34, 41].

7 Related Work

Ransomware Defense. The ransomware defense approaches can be grouped into three categories: static analysis-based, dynamic analysis-based, and key extraction-based. Static analysis-based solutions [64] use structural features such as strings and opcodes to detect ransomware. Dynamic analysis-based solutions use behavioral features such as network and registry activity, API/system call usage [13, 40, 65], I/O access patterns, and file system activity [52], network traffic features [14, 21, 22, 47], registry changes [35, 37]. The works [33, 46] used memory forensics and behavioral analysis to extract keys and recover the files.

Web API Security. Several works in the literature have analyzed the security and privacy of emerging web APIs. In [32], the authors identified a vulnerability in the Geolocation API, analyzed its impact, and discussed potential countermeasures. In [55], the authors explored a new attack vector through the screen-sharing API and discussed the effectiveness of the existing web defense systems. In [62], Weeks discussed the possible adverse effects of exploiting the FSA API, which includes data exfiltration and the potential code execution.

8 Conclusion

In this work, we designed and implemented the first browser-based ransomware - RØB and showed the inefficacy of the underlying FSA API documentation. Our extensive evaluations with 3 different OSs, 29 distinct directories and 5 cloud providers showed that RØB is capable of encrypting numerous types of files in various local directories, cloud-integrated directories, external storage devices, and network-shared folders. As existing ransomware detection systems including commercial antivirus solutions face several issues against RØB due to its distinct features, there was a need to propose a new defense solution against RØB-style attacks. Therefore, we proposed three different defense approaches to mitigate this new attack vector at different levels.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their helpful feedback and time. We also express our sincere gratitude to the developers of the FSA API at Google for their support and cooperation. Moreover, this work was partially supported by the US National Science Foundation (Awards: 1663051, 2039606, 2219920), Cyber Florida, Google ASPIRE Program, and Microsoft. The views expressed are those of the authors only, not of the funding agencies.

References

[1] "Puppeteer," https://pptr.dev/, 2023.

[2] "Enigma," https://github.com/cubbit/enigma, 2019.

[3] "Version history overview," https://help.dropbox.com/files-folders/restore-delete/version-history-overview, 2020.

[4] "The file system access api with origin private file system," https://webkit.org/blog/12257/the-file-system-access-api-with-origin-private-file-system/, 2022.

[5] "Kaspersky total security," https://usa.kaspersky.com/total-security, 2022.

[6] "Protecting against consent phishing," https://docs.microsoft.com/en-us/azure/active-directory/manage-apps/protect-against-consent-phishing, 2022.

[7] "Browser market share worldwide," https://gs.statcounter.com/browser-market-share, 2023.

[8] "Can i use file system access api?" https://caniuse.com/native-filesystem-api, 2023.

[9] "Project fugu api showcase," https://developer.chrome.com/blog/fugu-showcase/?api=file-system-access, 2023.

[10] "View activity & file versions," https://support.google.com/drive/answer/2409045, 2023.

[11] "Visual studio code," https://vscode.dev/, 2023.

[12] D. Akhawe and A. P. Felt, "Alice in warningland: A Large-Scale field study of browser security warning effectiveness," in *22nd USENIX Security Symposium*, 2013.

[13] B. Al-rimy, M. Maarof, Y. Prasetyo, M. M. S. Z. Syed, Shaid, and A. F. M. Ariffin, "Zero-day aware decision fusion-based model for crypto-ransomware early detection," *International Journal of Integrated Engineering*, 2018.

[14] O. M. K. Alhawi, J. Baldwin, and A. Dehghantanha, "Leveraging machine learning techniques for windows ransomware network traffic detection," in *Cyber Threat Intelligence*, 2018.

[15] Avast, "Avast one essential protection for your life today," https://www.avast.com/en-us/avast-one#pc, 2022.

[16] AVG, "Get free antivirus that's trusted by experts," https://www.avg.com/en-us/homepage#pc, 2022.

[17] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A first look at code obfuscation for webassembly," in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2022.

[18] J. P. Bigham, I. Lin, and S. Savage, "The effects of "not knowing what you don't know" on web accessibility for blind web users," in *Proceedings of the 19th International Conference on Computers and Accessibility*, 2017.

[19] Brave, "Remove support for native file system api," https://github.com/brave/brave-browser/issues/11407#issuecomment-851742821, 2019.

[20] C. Bravo-Lillo, S. Komanduri, L. F. Cranor, R. W. Reeder, M. Sleeper, J. Downs, and S. Schechter, "Your attention please: Designing security-decision uis to make genuine risks harder to ignore," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, 2013.

[21] K. Cabaj, M. Gregorczyk, and W. Mazurczyk, "Software-defined networking-based crypto ransomware detection using http traffic characteristics," *Computers and Electrical Engineering*, 2018.

[22] K. Cabaj and W. Mazurczyk, "Using software-defined networking for ransomware mitigation: The case of cryptowall," *IEEE Network*, 2016.

[23] Y. Chen, Y. Gao, N. Ceccio, R. Chatterjee, K. Fawaz, and E. Fernandes, "Experimental security analysis of the app model in business collaboration platforms," in *31st USENIX Security Symposium*, 2022.

[24] Chromium, "File system access web api - chromium security model - google docs," https://docs.google.com/document/d/1NJFd-EWdUlQ7wVzjqcgXewqC5nzv_qII4OvlDtK6SE8/edit, 2023.

[25] A. Continella, A. Guagnelli, G. Zingaro, G. Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: A self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[26] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, 2008.

[27] A. A. E. Elhadi, M. A. Maarof, and B. I. Barry, "Improving the detection of malware behaviour using simplified data dependent api call graph," *International Journal of Security and Its Applications*, 2013.

[28] FBI, "Spoofing and phishing," https://www.fbi.gov/scams-and-safety/common-scams-and-crimes/spoofing-and-phishing, 2023.

[29] A. P. Felt, S. Egelman, D. A. Matthew Finifter, and D. Wagner, "How to ask for permission," in *7th USENIX Workshop on Hot Topics in Security*, 2012.

[30] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo, "Rethinking connection security indicators," in *Twelfth Symposium on Usable Privacy and Security*, 2016.

[31] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, 2009.

[32] X. Han, J. Xiong, W. Shen, Z. Lu, and Y. Liu, "Location heartbleeding: The rise of wi-fi spoofing attack via geolocation api," in *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[33] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017.

[34] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *21st USENIX Security Symposium*, 2012.

[35] J. Hwang, J. Kim, S. Lee, and K. Kim, "Two-stage ransomware detection using dynamic analysis and machine learning techniques," *Wireless Personal Communications*, 2020.

[36] Y. Javed and M. Shehab, "Investigating the animation of application permission dialogs: A case study of facebook," in *Data Privacy Management and Security Assurance*, G. Livraga, V. Torra, A. Aldini, F. Martinelli, and N. Suri, Eds., 2016.

[37] B. Jethva, I. Traoré, A. Ghaleb, K. Ganame, and S. Ahmed, "Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring," *Journal of Computer Security*, 2019.

[38] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium*, 2016.

[39] A. Kharraz and E. Kirda, "Redemption: Real-time protection against ransomware at end-hosts," in *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing, 2017.

[40] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[41] N. Kshetri, "The economics of click fraud," *IEEE Security Privacy*, 2010.

[42] Malwarebytes, "Malwarebytes browser guard," https://www.malwarebytes.com/browserguard, 2023.

[43] ——, "Malwarebytes premium for windows," https://www.malwarebytes.com/premium, 2023.

[44] F. Martinelli, F. Mercaldo, C. Michailidou, and A. Saracino, "Phylogenetic analysis for ransomware detection and classification into families," in *ICETE*, 2018.

[45] M. Medhat, S. Gaber, and N. Abdelbaki, "A new static-based framework for ransomware detection," *2018 IEEE 16th Intl Conf. Congress*, 2018.

[46] S. Mehnaz, A. Mudgerikar, and E. Bertino, "Rwguard: A real-time detection system against cryptographic ransomware," in *Research in Attacks, Intrusions, and Defenses*, 2018.

[47] J. Modi, I. Traore, A. Ghaleb, K. Ganame, and S. Ahmed, "Detecting ransomware in encrypted web traffic," in *Foundations and Practice of Security*, 2020.

[48] Mozilla, "Mozilla specification positions," https://mozilla.github.io/standards-positions/#file-system-access, 2022.

[49] J. Muñoz-Arteaga, R. M. González, M. V. Martin, J. Vanderdonckt, and F. Álvarez Rodríguez, "A methodology for designing information security feedback based on user interface patterns," *Advances in Engineering Software*, 2009.

[50] OneDrive, "Restore a previous version of a file stored in onedrive," https://support.microsoft.com/en-us/office/restore-a-previous-version-of-a-file-stored-in-onedrive-159cad6d-d76e-4981-88ef-de6e96c93893, 2021.

[51] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Comput. Surv.*, 2019.

[52] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A survey on ransomware: Evolution, taxonomy, and defense solutions," *ACM Comput. Surv.*, 2022.

[53] H. Oz, F. Naseem, A. Aris, A. Acar, G. S. Tuncay, and A. S. Uluagac, "Poster: Feasibility of malware visualization techniques against adversarial machine learning attacks," in *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022.

[54] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "Cryptolock (and drop it): Stopping ransomware attacks on user data," in *IEEE 36th International Conference on Distributed Computing Systems)*, 2016.

[55] Y. Tian, Y. C. Liu, A. Bhosale, L. S. Huang, P. Tague, and C. Jackson, “All your screens are belong to us: Attacks exploiting the html5 screen sharing api,” in *IEEE Symposium on Security and Privacy*, 2014.

[56] Trendmicro, “Antivirus+ security,” https://www.trendmicro.com/en_me/forHome/products/antivirus-plus.html, 2022.

[57] A. van der Heijden and L. Allodi, “Cognitive triaging of phishing attacks,” in *28th USENIX Security Symposium*, 2019.

[58] E. von Zezschwitz, S. Chen, and E. Stark, “"it builds trust with the customers" - exploring user perceptions of the padlock icon in browser ui,” ser. IEEE Security and Privacy Workshops. IEEE Computer Society, 2022.

[59] W3C, “File system access,” https://github.com/WICG/file-system-access, 2023.

[60] ——, “File system access,” https://wicg.github.io/file-system-access/, 2023.

[61] ——, “File system access,” https://wicg.github.io/file-system-access/#security-ransomware, 2023.

[62] M. Weeks, “Internal affairs: Hacking file system access from the web,” https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Internal-Affairs-Hacking-File-System-Access-From-The-Web.pdf, 2021.

[63] L. Zeng, Y. Xiao, and H. Chen, “Linux auditing: Overhead and adaptation,” in *IEEE International Conference on Communications (ICC)*, 2015.

[64] B. Zhang, W. Xiao, X. Xiao, A. K. Sangaiah, W. Zhang, and J. Zhang, “Ransomware classification using patch-based cnn and self-attention network on embedded n-grams of opcodes,” *Future Generation Computer Systems*, 2020.

[65] J. Zhou, M. Hirose, Y. Kakizaki, and A. Inomata, “Evaluation to classify ransomware variants based on correlations between apis,” in *6th International Conference on Information Systems Security and Privacy*, 2020.

Appendix

A Responsible Disclosure Communication Process Details

In this section, we provide the details of our communication and disclosure process with the respective developers and editors of the FSA API, which is Google.
Email Contact: On February 17, 2022 we sent an email to one of the main developers of the FSA API about our findings. They acknowledged our findings and redirected us to the editor of the FSA documentation for possible documentation [61] improvements as they agreed it was a weak explanation. Although we had several exchanges with them, they, unfortunately, did not swiftly act on it to provide any improvements (although we offered our help). Note that at that point, we have been already conducting an extensive analysis on this problem (i.e., different OSes, anti-virus products, cloud providers, etc.).
Security Bug Report and Opening GitHub Issue: Afterwards, we submitted a security bug report to Chromium on November 7, 2022, which is not a public process, and explained our findings and possible documentation improvements. They recommended us to open a public GitHub issue in order to be able to swiftly update the documentation based on our findings. However, due to the anonymity requirements of the conference and to prevent the further publicization of the issue at that moment, we have not opened an issue on GitHub then.
Video Conference with the FSA API Developers/Editors: Then, we contacted and met with the FSA API developers/editors via video conference on November 10, 2022. In the meeting, we further explained the impacts of the ransomware through the FSA API, possible documentation improvements, and our defense solutions to the developers. They acknowledged that the ransomware risks in the documentation were downplayed and agreed to update the documentation based on our findings. In the meeting, we also mentioned the potential publication of the paper to the developers and editors of the API. In turn, they asked us to provide them with our suggestions on how to improve the documentation and we have been working with them on this front to better reflect the ransomware risks in the documentation.

B Further UI Improvements

Further UI improvements are also possible. We found that some cloud platforms (e.g., OneDrive) have similar permission boxes for file-sharing, which include other details that can be adapted by the FSA API. First, there is an icon of the accessed website to prevent spoofing attacks. Second, there is an explanation that the website is not endorsed by the API. Moreover, there is a link to report suspicious websites, which could help the developer in building a blocklist to help users in the long term. Finally, there is a link to the website's privacy statement to learn more about how the data will be used by the website. We have not integrated these into our designs in Figure 4 and 5 to maintain the simplicity of our design.

C Approach 1 - Dataset Creation

To calculate the size and entropy change between an original file and its modified versions, we need a comprehensive dataset that covers different types of files with various versions where users performed a diverse set of modifications.

Table 4: The list of web applications used for the local activity monitoring experiments.

Web Application	Type	Link
RobEncOne	malicious	N/A
RobEncHundred	malicious	N/A
RoBReordered	malicious	N/A
RoBwithBenign	malicious	N/A
RoBwithBenAPI	malicious	N/A
RoBwithEncWait	malicious	N/A
RoBwithFSAWait	malicious	N/A
RoBBothWait	malicious	N/A
bangle	benign	https://bangle.io/
drawio	benign	https://app.diagrams.net/
excalidraw	benign	https://excalidraw.com/
GitHub	benign	https://github.dev/github/dev
glitch	benign	https://googlechromelabs.github.io/browser-fs-access/demo/
photopea	benign	https://www.photopea.com/
svgco	benign	https://svgco.de/
text-editor	benign	https://googlechromelabs.github.io/text-editor/
vscode	benign	https://vscode.dev/

We searched for such databases in various resources including IEEE DataPort, Google Dataset Search, and Kaggle. However, we could not find a suitable dataset for our needs. Hence, we decided to create our own dataset that includes different file types with various realistically-generated versions.

We selected five file types, namely `pdf, docx, xlsx, txt, jpeg`, that may contain sensitive data of users and enterprises. The Digital Corpora consists of almost 1 million real files from government websites and is distributed freely [31]. Using the Digital Corpora, we collected 1000 files from each file type. The average file size of each type of a file is 0.67MB for `pdf`, 0.24MB for `docx`, 0.29MB for `xlsx`, 0.58MB for `txt`, and 0.15MB for `jpeg` formats.

To mimic user behavior on the files, we considered adding and removing contents from the files to generate different versions. While doing that, we paid attention to preserving the file formats. To have a comprehensive dataset, we created 100 different versions of each file. The first 50 versions are content removed versions of files are created to mimic deletion operations made by users. On the other hand, the other second 50 versions are content-added versions of the files are created to reflect data appending operation made by the user. We developed Python scripts that perform these procedures on the files. The low-level details of how we created 100 different versions of each file with respect to each file type are summarized as follows:

Modification of `txt` files For the content removal operation from a text file, we define a range between 1 and n, where n is the number of words in the text file. Afterward, we generate a random number r in the range of $[1,n]$ using Python's built-in random function that generates random numbers with respect to Uniform Random Distribution. Then, starting from the end of the text file, we remove r words from the original version of the text file and created the new version of the file. For the content insertion operation, we generate another random number r in the same way and we randomly choose r words from the word database that includes the contents of all the text files in our dataset and append these randomly chosen words to the end of the text file.

Modification of `docx` files For the content removal operation from a `docx` file, we used a similar methodology with `text` files. Specifically, we first generate a random number r between 1 and n where n is the number of words in the `docx` file. Then, starting from the end of the `docx` file, we remove r words from the `docx` file to create the content removed version. In the context of creating content-added versions of the `docx` file, we utilized our `docx` content database. Such that, we retrieve r words randomly from the `docx` content database and also randomly choose a `jpeg` file from our `jpeg` file database and append them to the `docx` file.

Modification of `pdf` files For the content removal operation from a `pdf` file, we first generate a random number r that is in the range of $[1,n]$ where n represents the number of pages of the `pdf` file. Afterward, we remove the content that resides on the rth page of the `pdf` file to generate the content-removed version of the file.To generate the content-added version of the file, we randomly choose a `pdf` file from our dataset and add its content on the rth page to the end of the `pdf` file.

Modification of `jpeg` files We perform the content-removal operation on `jpeg` file by cropping the file randomly. To achieve this, we define two random variables, namely r_1 that is between 1 and n, and r_2 that is between 1 and m, where n represents the width and m represents the height of the `jpeg` file. By using these randomly generated width and height values, we crop the `jpeg` file starting from the left top corner ((0,0) coordinates). After this operation, the `jpeg` file is cropped to become an $r_1 x r_2$ image. We create the other 50 different versions of a `jpeg` file by merging it with another `jpeg` file randomly selected from our database, which includes all `jpeg` files in our dataset.

Modification of `xlsx` files To perform the removal operation on an `xlsx` file, we first calculate the number of rows in a `xlsx` file. Then, we define a range between 1 and n, where n is the number of rows in the `xlsx` file. Afterward, we create a random number r in this range and remove r rows from the end of the `xlsx` file. To perform adding operation, we add a random number (r) of rows to the end of the `xlsx` file that is retrieved from `xlsx` file database.

Reflecting Malicious Changes. To reflect malicious changes (i.e, encryption with RØB) on the files, we encrypted each file including modified versions in our dataset with RØB that uses the AES-256 encryption algorithm.

D Approach 1 - Evaluation Against Adaptive Attackers

To evaluate our first approach against more adaptive attackers, we randomly selected 500 files (100 per file type) from our original benign dataset. We created corresponding 500 malicious files for each evasion technique. We repeated this process for different techniques. Then, we evaluated the impact of the technique and its success rate in evading our classifier.

Table 5: Evaluation against more adapt attackers.

Technique		Acc.	Recall	Prec.	F1	TP	TN	FN	FP
Partial Encryption	TXT	1.0	1.0	1.0	1.0	100	0	0	0
	PDF	0.98	0.98	1.0	0.98	98	0	2	0
	JPEG	0.99	0.99	1.0	0.96	99	0	1	0
	DOCX	0.98	0.98	1.0	0.98	98	0	2	0
	XLSX	0.97	0.97	1.0	0.98	97	0	3	0
Low-entropy Data Padding	TXT	1.0	1.0	1.0	1.0	100	0	0	0
	PDF	0.97	0.97	1.0	0.98	97	0	3	0
	JPEG	0.99	0.99	1.0	0.96	99	0	1	0
	DOCX	0.99	0.99	1.0	0.96	99	0	1	0
	XLSX	0.97	0.97	1.0	0.98	97	0	3	0
Post-encryption Encoding (Base64)	TXT	1.0	1.0	1.0	1.0	100	0	0	0
	PDF	0.99	0.99	1.0	0.99	99	0	1	0
	JPEG	0.99	0.99	1.0	0.96	99	0	1	0
	DOCX	0.99	0.99	1.0	0.99	99	0	1	0
	XLSX	0.96	0.96	1.0	0.97	96	0	4	0
Post-encryption Encoding (Base32)	TXT	1.0	1.0	1.0	1.0	100	0	0	0
	PDF	1.0	1.0	1.0	1.0	100	0	0	0
	JPEG	1.0	1.0	1.0	1.0	100	0	0	0
	DOCX	1.0	1.0	1.0	1.0	100	0	0	0
	XLSX	0.99	0.99	1.0	0.99	99	0	1	0
Post-encryption Encoding (Hexadecimal)	TXT	1.0	1.0	1.0	1.0	100	0	0	0
	PDF	0.94	0.94	1.0	0.96	94	0	6	0
	JPEG	0.95	0.95	1.0	0.97	95	0	5	0
	DOCX	1.0	1.0	1.0	1.0	100	0	0	0
	XLSX	0.99	0.99	1.0	0.99	99	0	1	0
Custom Evasion	TXT	0.1	0.1	1.0	0.18	10	0	90	0
	PDF	0.12	0.12	1.0	0.21	12	0	88	0
	JPEG	0.06	0.06	1.0	0.11	6	0	94	0
	DOCX	0.09	0.09	1.0	0.16	9	0	91	0
	XLSX	0.09	0.09	1.0	0.16	9	0	91	0

During the classifier evaluation, we used 10-fold to ensure a clean split between training and test data. The average size of each file type in our new dataset is 0.72MB for `pdf`, 0.23MB for `xlsx`, 0.33MB for `docx`, 0.12MB for `jpeg` and 0.40MB for `txt` files. Also, the average entropy values of each type in our dataset are 7.54 for `pdf`, 7.68 for `xlsx`, 7.37 for `docx`, 7.80 for `jpeg` and 4.32 for `txt` files. To mimic the adaptive attacker behavior on the files, we used partial encryption, low-entropy data padding (e.g., injecting low-entropy data), encoding post-encryption, and custom evasion. We algorithmically chose the best-performing classifier in an automated fashion. Specifically, the KNN classifier yielded the best results for `xlsx` files, the Decision Tree classifier for `pdf`, `jpeg`, and `txt` files, and the XGBoost classifier for `docx` files. We presented the results of our experiments in Table 5. The details of our dataset creation procedure for evasion techniques are as follows:

Partial-Encryption: To mimic partial-encryption behavior on the files, we encrypted 25% of the file content using the AES-256 algorithm.

Low-entropy data padding: To mimic the low-entropy data padding behavior on the files, we initially encrypted the files using the AES-256 algorithm. Subsequently, we injected a random amount of low-entropy data consisting of null characters (e.g., `\x00`, with a randomly defined length varying from 10,000 to 20,000) to the file content.

Encoding Post-Encryption: To perform the encoding post-encryption technique, we initially encrypted each file type using the AES-256 algorithm. Then, we applied various encoding techniques to the files, including Base64, Base32, and hexadecimal encoding.

Custom Evasion: In this scenario, we combined both data padding and partial encryption techniques to mimic the benign modification of the user. To achieve that, we continuously encrypt the 25% of the file and injected a amount of data until it achieves the $\pm 10\%$ of the average entropy and size of the benignly modified file.