Forensic Science International: Digital Investigation 49 (2024) 301763

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

Decrypting IndexedDB in private mode of Gecko-based browsers

Dohun Kim, Sangjin Lee, Jungheum Park *

School of Cybersecurity, Korea University, Seoul, South Korea

ARTICLE INFO

Keywords:
Digital forensics
Memory forensics
Private browsing
IndexedDB
Firefox
Tor browser

ABSTRACT

Various technical and legal issues hinder direct investigation on cloud services, which facilitates alternative approach to investigate services through artifacts left by web browsers. Among diverse web browser artifacts, client-side storages such as IndexedDB have been focused to retrieve contextual information about user behavior. However, analyzing such client-side storages has been difficult in private mode environments, as they were only kept in memory or not supported at all, depending on the browser. Recently, Firefox has started to support IndexedDB storage in private mode by storing encrypted files on disk during private sessions since July 2023. Since then, Gecko-based browsers' effort to support client-side storages through encrypted files on disk has been continued with Tor Browser also began supporting IndexedDB in the same way since October 2023. Meanwhile, the research to utilize those encrypted files on investigation has not progressed much yet. This paper shows how to decrypt client-side storages generated on Gecko-based browsers' private mode by extracting cipherkeys in memory. Experimental results indicate that when private session is running, our proof-of-concept tool successfully decrypts all encrypted files. Additionally, there is a possibility of recovering data even in an inactive state by utilizing hibernation file on disk.

1. Introduction

The usage of Software-as-a-Service (SaaS) in cloud-based environments has been consistently increasing, becoming an integral part of our lives that we can no longer separate from. Now, instead of installing document editing programs on our PCs, we access the Office365 site online to edit and save documents stored in remote data center. This brought us various changes from a digital forensics perspective when investigating user behavior. Data related to user behavior is no longer solely stored on local PCs; rather stored on remote cloud servers. The most reliable method to investigate those data would be seizing the remote server and directly examining information related to the user. However, there are many legal and technical obstacles, such as the cooperation of the cloud operating company. As a result, directly accessing user data stored on remote servers is difficult unless the user provides the account information of the SaaS.

In this environment, research has been conducted to extract user behavior data from the user's PC instead of remote cloud servers. Particularly, as most cloud services are accessed through web browsers, studies have focused on extracting user behavior data through web browser artifacts stored on the PC, including client-side storages. Among these artifacts, there has been ongoing research specifically targeting the recently emerging storage, IndexedDB, which contains relatively more contextual information (Palıgu et al., 2019).

However, a significant obstacle hindering the analysis of these web browser artifacts is the various private browsing technologies, such as private modes in different browsers and anonymous browsing technologies like Tor Browser. Private mode, adopted as various names in lots of browsers such as Chrome's Incognito mode and Firefox's Private browsing mode (*This paper writes those private browsing modes as* ***Private Mode***), typically limits or entirely eliminates traces including client-side storages on disk and memory, depending on the browser. Meanwhile, recently, Gecko-based browsers including Firefox have begun supporting client-side storages such as IndexedDB in private mode by the form of encrypted on-disk storage. If it is possible to decrypt such encrypted client-side storage data, it could be highly beneficial from a digital forensics perspective, as it would allow obtaining contextual information related to a user's behavior in private mode.

Therefore, this paper aims to extract and decrypt encrypted client-side storage data left on the disk during the private mode. This will be achieved through analyzing process memory of Gecko-based browsers, and using information obtained through the analysis of the Gecko engine's source code (Mozilla, 2024).

This study has the following contributions.

* Corresponding author.
E-mail addresses: solmontea98@gmail.com (D. Kim), sangjin@korea.ac.kr (S. Lee), jungheumpark@korea.ac.kr (J. Park).

https://doi.org/10.1016/j.fsidi.2024.301763

Available online 5 July 2024

- Presenting a method to decrypt encrypted IndexedDB and Cache API data stored on the disk during the private mode in Gecko-based browsers (Firefox and Tor Browser).
- Confirming the possibility of extracting and decrypting encrypted data in various real-world cases through experiments.
- Providing a proof-of-concept tool implemented based on the proposed methodology.

This paper is structured as follows: **Section 2** reviews previous researches related to this paper, and **Section 3** provides relevant knowledges to this paper. **Section 4** provides a detailed examination of how Gecko-based browsers manage IndexedDB-related data in both disk and memory during private mode. **Section 5** introduces a decryption method for encrypted IndexedDB files stored on the disk, and explores how to apply this method in various cases. **Section 6** verifies the decryption of encrypted IndexedDB data through actual experiments in diverse cases. **Section 7** discusses the meaning of the experiment results and the limitation of this work. Finally in **Section 8**, the findings of this study are summarized, and future research topics to be pursued are proposed.

2. Related work

2.1. Forensic analysis on IndexedDB artifacts of SaaS

Palıgu and Varol (2022a,b) conducted a study on Microsoft Teams desktop application and Instagram web service to explore whether it is possible to extract various user behavior from IndexedDB storage stored on the user's PC. The research revealed that, without acquiring separate user accounts, it was possible to extract diverse information related to user behavior, such as messages sent, team creation details, and list of participants on scheduled meetings. Additionally, for the Instagram service, they found not only basic information like the number of followers and basic permission-related settings but also additional user behavior information including the content of comments and the location where a post was uploaded. By listing the obtained information alongside the recorded time information, they were able to analyze user's behavior related to the respective services in detail.

Likewise, various studies have attempted to track user behavior using IndexedDB. However, solutions for cases where analysis become challenging due to the utilization of private mode have not been discussed yet.

2.2. Forensic analysis on private mode in modern browsers

Horsman et al. (2019) conducted a comprehensive analysis of 30 web browsers to verify whether the private mode offered by browsers is truly private. They used each of the 30 browsers for searching specific keywords, and checked whether those keywords were detected on the disk image. As a result, among the 22 browsers that provided private mode, it was found that 5 browsers (Avant, Comodo Dragon, Edge, Epic, Internet Explorer) left keyword traces in local storage, session storage, and unallocated space on the disk.

Hughes et al. (2021) aimed to analyze artifacts on disk and memory targeting 4 widely used browsers (Chrome, Firefox, Edge, Brave) to identify traces related to user's private mode browsing. As a result, most of the browsing data were not saved on the disk in accordance with the claims of the manufacturers. However, in the memory, including the pagefile on the disk, various information such as visited URLs and email accounts were obtained by string-search.

Hariharan et al. (2022) analyzed the traces left in memory during the utilization of private mode in 4 portable web browsers: Brave, Tor, Vivaldi, and Maxthon which works without installation and does not leave traces on the disk. They were able to extract keywords or domains that were searched or accessed by searching known keywords.

Alfosail and Norris (2021) conducted a focused analysis on artifacts left in the Windows memory during the use of Tor Browser. They presented a comprehensive methodology by combining the information left on the disk. The authors also utilized Volatility's Yarascan plugin and Bulk Extractor tool to identify URLs that were likely accessed by the user through sorting the match counts.

Fernández-Fuentes et al. (2022, 2023) conducted study on the browser's private modes in Linux and Android operating system environments, expanding the scope from the widely-studied Windows operating system. They concluded that there were no meaningful information found on the file system in both Linux and Android. However, through searching for strings in memory, various data such as entered keywords and login-related information were obtained. Particularly within Linux in virtual machine and Android, relevant keywords could still be obtained even after rebooting the device.

Various studies have focused on finding methods to analyze browser's private modes with disk and memory artifacts. As a result, they were able to identify diverse browsing traces especially from memory. However, most existing studies are string-search based, which provides only fragmented information and leaving ambiguity about whether the user actually performed certain action. Client-side storage, such as IndexedDB, contains more diverse set of information, including the context of user behavior. Therefore, if the analysis of client-side storage becomes possible in private mode, it could provide deeper insights to the forensic analysts.

2.3. Forensic analysis on user applications by object-layout based memory forensics

Choi et al. (2023) utilized object layout to extract meaningful information from the virtual memory of Chrome browser. The authors identified various classes related to web browsing activities, such as browsers, tabs, and visited URLs from the source code of the Chromium engine. They automatically constructed the object layout for these classes, allowing them to capture these objects in the virtual memory of Chrome browser. Through this approach, the authors were able to extract diverse forensically meaningful information, including various browsing activities and URLs visited in private mode.

Fernández-Álvarez and Rodríguez (2022) similarly identified object layout of classes related to chat messages in the memory of the Telegram application. They extracted and reconstructed these classes, enabling the extraction of various information such as Telegram conversation histories, exchanged file details, and deleted messages.

3. Background

3.1. Client-side storage

Client-side storage is a technology that allows browsers to store and retrieve data on the disk of user's PC. Various client-side storage's tradeoffs are as follows. Cookies and Web Storage have been the most fundamental storage options existing since the past. Both Cookies and Web Storage are used to store small-sized string data, around few KBs ~ MBs.

IndexedDB was introduced around 2012 as a storage option that differs from traditional client-side storages. Unlike its predecessors, IndexedDB allows storing large amount of data in the range of several MBs to GBs, using NoSQL DB structure. The implementation of IndexedDB varies between browsers; Chromium-based browsers, for instance, support IndexedDB with LevelDB as the underlying storage, while Gecko-based browsers support it using SQLite DB as the underlying storage. In case of Gecko-based browsers, such as Firefox, IndexedDB data is managed by creating folders for each web service origin at *%UserProfile%\AppData\Roaming\Mozilla\Firefox\[Profile ID]\storage\default* path with DB files and binary large object (BLOB) files in it. In Firefox, when storing values that are too large for the SQLite DB, they are separately managed as BLOB files.

Cache API is a client-side storage designed to be utilized in the browser's service worker, and it has been supported since around 2015. Similar to IndexedDB, Cache API supports large storage capacity and typically allows web services to operate offline by storing various assets of a website. In Gecko-based browsers, the storage data for Cache API is created inside the same path as IndexedDB storage, with actually cached data within the *MORGUE* folder and the *caches.sqlite* file that manages it. In addition to the mentioned storage technologies, other various client-side storages like WebSQL exists. However, none of them are used widely nowadays due to the deprecation or compatibility issue.

Despite the existence of various client-side storage options, issues such as storage capacity and supported data types have led to IndexedDB and Cache API being the most widely used ones recently. Generally, for loading network resources required for application and file-based content, the use of Cache API is recommended. For other types of data, the usage of IndexedDB is recommended (web.dev).

3.2. Client-side storage implementations for private mode in modern web browsers

Client-side storage typically stores data on the disk in a regular browsing environment. However, support for client-side storage in private mode varies significantly from one browser to another. When comparing Chrome and Firefox, which use different browser engines, in Chrome's private mode, all client-side storages are managed in memory, and no separate files are stored on the disk (chromium). In Firefox's private mode, small-sized data like cookies or web storages are managed in memory. However, for larger data storage using IndexedDB and Cache API, support for these features in private mode was introduced for the first time in Firefox v115.0, released on July 4, 2023, as shown in Fig. 1. According to the release notes, the cipherkey used to encrypt IndexedDB data is stored only in memory, and when the private session ends, both the cipherkey and IndexedDB data are completely deleted (Firefox, a). Cache API also started to be supported in private mode of Firefox similar to IndexedDB, from version v122.0 released on January 23, 2024 (Firefox, b).

Meanwhile, other Gecko-based browsers like Tor Browser that always operates in the form of Firefox's private mode, did not initially support IndexedDB and Cache API. However, starting from Tor v13.0 released on October 12, 2023, it began to operate based on the source code of Firefox v115.0 and started supporting IndexedDB (Tor Project). However, Cache API is not yet supported in Tor Browser, and will be supported as soon as Tor Browser operates based on the source code of Firefox v122.0.

The support for client-side storage in the private mode of major browsers can be broadly categorized into two approaches: in-memory storage and encrypted on-disk storage. Each approach has its own tradeoffs. In-memory implementation is relatively simple since most databases support it. However, the storage capacity is affected by the amount of available memory. Additionally, faster access and manipulation time of in-memory storage than on-disk storage enables potential detection of private mode usage by measuring access times (fingerprint). On the other hand, encrypted on-disk storage approach, while more complex to implement, is balanced by the benefits of maintaining storage capacity and avoiding privacy concerns associated with potential detection of private mode usage.

3.3. Physical memory related files in Windows

Generally, physical memory is only accessible while the computer is running, and the data is wiped after the computer is shut down. However, in the Windows operating system, there is possibility of acquiring volatile memory information even after the computer has been shut down through the volatile memory stored on disk for virtual memory management. Typically, the files containing volatile memory on the disk are as follows. *Pagefile.sys* and *Swapfile.sys* are used to store portions of physical memory that have not been used for a long time when there is insufficient capacity in physical memory. When the stored data is needed again, these files operate by being swapped with another set of data existing in physical memory.

Unlike previous two files, *Hiberfil.sys* supports for various power and sleep options during the Windows operating system's shutdown. When using the "Fast Startup" option introduced after Windows 10, the memory area used by user processes just before shutdown does not persist, but the area used by Windows kernel and drivers, which was loaded into volatile memory, is stored in *Hiberfil.sys*. Furthermore, when using the "Hibernate" option, volatile memory data just before shutdown, including the memory areas of user processes, is stored directly in *Hiberfil.sys* file. This allows for the retrieval of whole volatile memory data even after shutdown (Bang et al., 2021).

4. Understanding encrypted IndexedDB data management in private mode of Gecko-based browsers

In this section, we will examine the detailed operation of the IndexedDB storage to uncover the decryption method for encrypted client-side storage files generated during the private mode in Gecko-based browsers. Additionally, we will inspect the structure of encrypted files.

4.1. Lifecycle of IndexedDB data in private mode

When accessing a web service that uses IndexedDB in Firefox's private mode, a new "private" folder is initially created in the Firefox storage path. Inside this folder, a unique 32 bytes UUID is assigned for each web service and used as the folder name. The IndexedDB data stored by the respective web services on the client are then saved within this folder. During this process, various classes of Gecko engine are

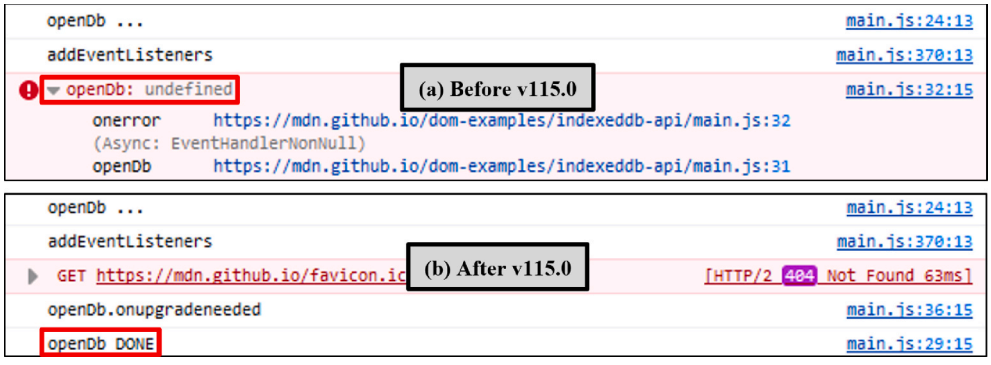

Fig. 1. Firefox supports IndexedDB in private mode after v115.0. (a) Before v115.0, openDb request fails since an IndexedDB related variable is undefined in private mode. (b) After v115.0, IndexedDB is supported in private mode and request for openDb succeeds.

related, and key fields of those classes are listed in Table 1.

Mainly, *IndexedDatabaseManager* class manages overall operations of the IndexedDB storage, particularly referencing objects of the *DatabaseFileManager* class that are created for each web service origin. The *DatabaseFileManager* class contains various information related to the origin, such as DB name, DB path, and whether it was created during private mode. Furthermore, each time IndexedDB SQLite DB files and BLOB files are stored on the disk during private mode, the *CipherKeyManager* class object within the *DatabaseFileManager* class generates a unique 32 bytes cipherkey for each file with random values. These cipherkeys are then added and managed in the hashtable inside *CipherKeyManager* class. The AEAD algorithm (ChaCha20-Poly1305) is applied to encrypt files on the disk using the randomly generated cipherkey and initial vector (IV). Among the cipher parameters used in AEAD algorithm, the additional associated data (aad) value is not utilized, and set to a null value.

When accessing the same origin's IndexedDB data during the same private session after the initial encryption and storage on disk, Firefox utilizes the cipherkey of the corresponding file managed in memory to decrypt the IndexedDB data on disk. When private mode ends, the "private" folder path where the IndexedDB data were stored is deleted. Simultaneously, cipherkeys for the encrypted files are erased as the entire cipherkey hashtable in *CipherKeyManager* object is initialized to value 0xE5 with deletion of *DatabaseFileManager* Object corresponding to each origin, as shown in Fig. 3.

4.2. CipherKey-related object's layout within memory space of browser process

To investigate how cipherkeys used to encrypt IndexedDB data are managed in memory, the hashtable area within the *CipherKeyManager* object, where the actual cipherkeys are stored, was examined. As shown in Fig. 2, the default hashtable can store up to 7 Entries, and when the number of entries exceed this limit, larger hashtable is created at a new address. The cipherkey entries contain the actual 32 bytes cipherkey following the header data related to the cipherkey. In particular, other fields than the randomly generated cipherkeys are fixed based on whether they were used for encrypting DB files or BLOB files in IndexedDB. For IndexedDB, the DB filename literal is fixed to "Default", resulting in a fixed *Length* field value of 0x07. The *DataFlags* field and *ClassFlags* field are fixed at value of 0x21 and 0x02, respectively. In contrast, for BLOB files stored separately from the DB files, as the filenames increase sequentially from "1", *Length* field value is fixed at 0x01 unless a specific DB file stores more than 10 BLOB files. *DataFlags* field and *ClassFlags* field are fixed at value of 0x05 and 0x02, respectively. Those flags' specific definitions are described in Mozilla's GitHub repository (Mozilla, 2021).

Table 1
Gecko-based browser's classes and their fields related to IndexedDB operations.

Class name	Field name	Description
IndexedDatabaseManager	FileManagerInfos	Maintains a list of all DatabaseFileManager objects per origin.
DatabaseFileManager	PersistenceType	Type of Database *(Default, Temporary, Private)*
	OriginMetadata	Metadata of Database *(Origin Name, whether it is created in Private Mode, etc.)*
	DatabaseName	Name of Database
	DatabaseID	ID of Database
	DatabaseFilePath	DB file path
	DirectoryPath	BLOB file directory path
	CipherKeyManager	Pointer to CipherKeyManager Object
	IsInPrivateBrowsingMode	Browsing Mode *(Private or not)*
CipherKeyManager	EntryStore	Pointer to hashtable that stores cipherkeys

4.3. Structure of encrypted DB and BLOB files

Next, the structure of encrypted DB files and BLOB files were examined. The results showed that all elements of the cipher parameters used for AEAD encryption were stored in the files except for the cipherkey.

Firstly, the encrypted DB file has a page size of 0x2000 bytes, and decryption is performed on a page-by-page basis. However, the SQLite header part (32 bytes) is not encrypted, so the SQLite header can be identified at the first 32 bytes of the first page in every encrypted DB file. Additionally, as shown in Fig. 4, the last 32 bytes of each page contain information about the cipher parameters used in AEAD encryption, where the first 12 bytes store the *Nonce* value, and the last 16 bytes store the *Initial Vector*. A noteworthy observation in the metadata area is that, the *Nonce* value is always set as a value less than the number of pages in the file.

For BLOB files, they were also encrypted using the AEAD encryption method, same as DB files. However, unlike DB files, BLOB files have a page size of 0x1000 bytes, and the cipher parameters used in encryption are stored in the first 48 bytes of each page as shown in Fig. 5. Firstly, since BLOB files do not end at a certain page unit, the first 2 bytes of each page store the actual length of the encrypted payload on that page. Additionally, 16 bytes from the beginning store the *Nonce*, which is 12 bytes. 32 bytes from the beginning store the *Initial Vector*, which is 16 bytes. The actual encrypted payload length is rounded up to the nearest multiples of 16, given that Firefox uses a basic encryption block size of 16 bytes. After that, for the last page, the remaining space from the end of the payload to the end of the page is filled with randomly generated data.

In the metadata area, a noteworthy point is that the 14 bytes area located after the *Ciphertext Length* is always initialized with the value 0x00 or 0xE5. As for the *Nonce* value, it equals the page's sequence number which starts from 0x00 on the first page and increments by 1 sequentially. The latter 4 bytes have same value throughout all pages within the same file.

5. Methodology: decryption of IndexedDB data encrypted by Gecko-based browsers

In this section, we will explore the algorithm for decrypting IndexedDB data using the encrypted file structures and cipherkey management method identified in **Section 4**. We will also consider how to apply this algorithm to decrypt data in various real-life cases.

5.1. Key ideas and procedures

Fig. 6 shows the decryption algorithm for encrypted IndexedDB data. Each phase is as follows.

5.1.1. Identification of candidate CipherKeys inside physical memory-related data

In **Phase 1**, according to the cipherkey related structure observed in **Section 4**, the physical memory and related data contains fixed pattern in the previous 8 bytes of the cipherkey. By searching for this pattern using regular expressions in memory, you can consider the subsequent 32 bytes of the detected portion as candidate cipherkeys for encrypted files. Detailed process is shown in **Algorithm 1**.

5.1.2. Extraction of a valid CipherKey by bruteforcing

In **Phase 2**, after extracting the candidate cipherkeys, bruteforcing is performed on encrypted file to find a key that does not result in a MAC

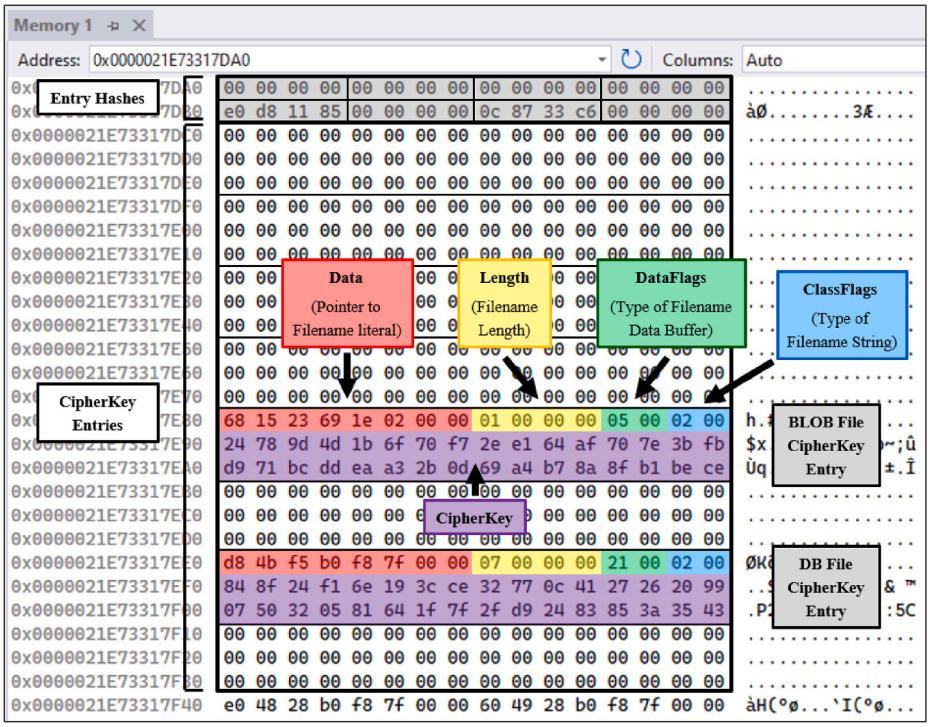

Fig. 2. Structure of the *CipherKeyManager* object in memory. Each cipherkey entry's header fields (first 16 bytes) except for 'Data' field are fixed to certain values, since filenames of IndexedDB data are always same (DB files are set to "Default") or predictable (BLOB files are incrementing by 1 starting from "1").

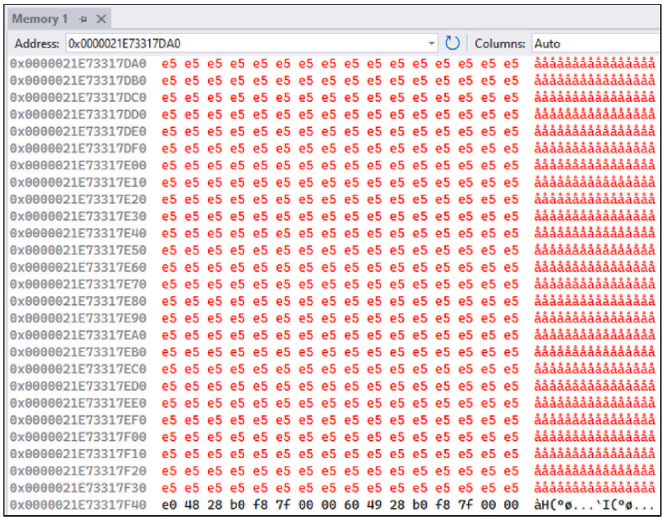

Fig. 3. Initialization of *CipherKeyManager* object after private session ends. The hashtable area is initialized with 0xE5.

authentication error during AEAD decryption. This is for identifying correct cipherkey for the target encrypted file. Cipher parameters used for AEAD decryption process are extracted from encrypted file based on structure identified in Section 4. Detailed process is shown in Algorithm 2 which is designed as an example for the IndexedDB DB file decryption process.

5.1.3. *Decryption of IndexedDB data*

In **Phase 3**, if a key is found that doesn not result in a MAC authentication error, that key is considered the correct cipherkey for the file. Subsequently, the AEAD algorithm is applied page-by-page to decrypt the encrypted file using the identified cipherkey.

5.2. *Considerations for practical applications*

The basic decryption algorithm has been made, but in real-world scenarios, there are often limitations on obtaining the necessary physical memory and encrypted IndexedDB data for decryption. Therefore, the methods for collecting physical memory information and encrypted IndexedDB data in various cases are explored as follows.

5.2.1. *When private session(s) are running*

In this case, physical memory data can be directly extracted, and since the encrypted file also exists on the disk and not deleted while private session continues, encrypted files can be acquired directly from the disk.

5.2.2. *When private session(s) are not running (= executed but terminated)*

When the browser is inactive, there are various cases to be

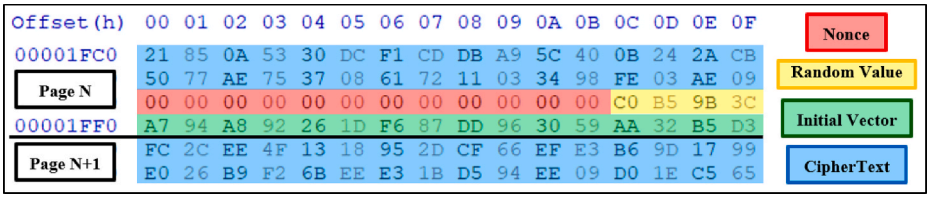

Fig. 4. Structure of pages inside an encrypted SQLite DB file. Cipher parameters are stored in the last 32 bytes of each page.

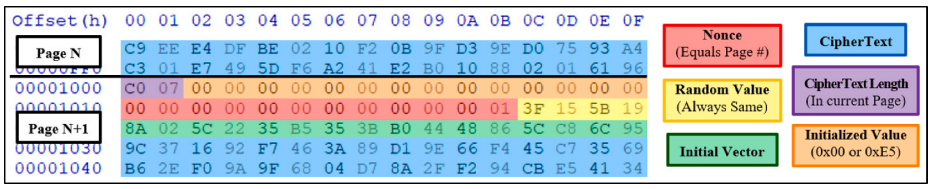

Fig. 5. Structure of pages inside an encrypted BLOB file. Cipher parameters are stored in the first 48 bytes of each page. *Initialized Value* field and *Random Value* field are not used during decryption process, but its values are fixed for all pages within the same file. *Nonce* field's value of each page equals the page's sequence number that starts from "0".

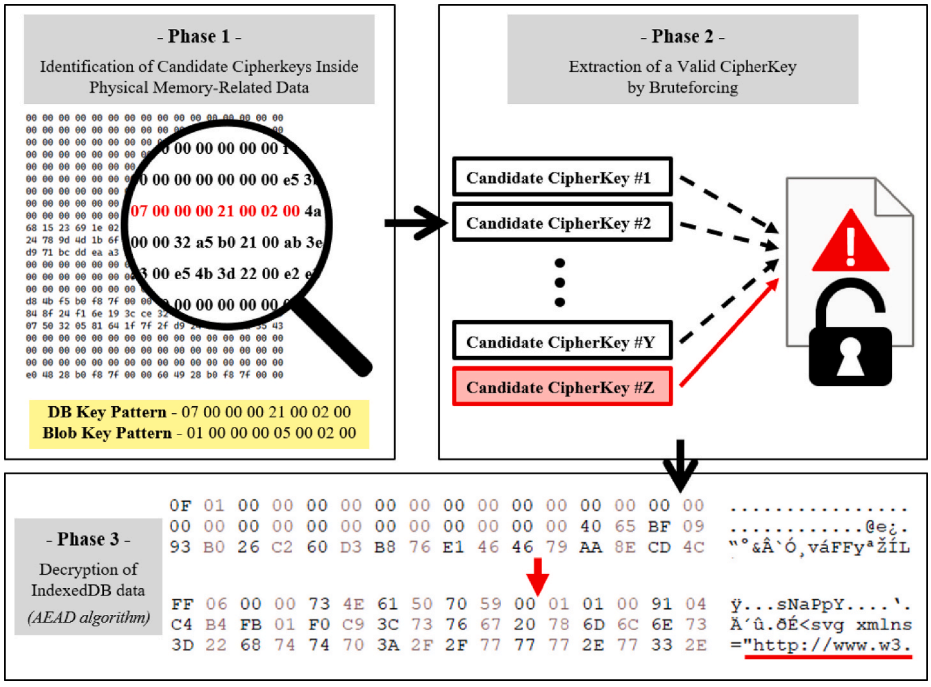

Fig. 6. Operational workflow of the proposed IndexedDB data decryption algorithm.

considered. To analyze the *Hiberfil.sys* file, we can consider 4 cases: when only the browser is closed (PC is still on), when the PC is shut down without any additional option settings, when the PC is shut down with the Fast Startup mode, and when the PC is shut down with the Hibernate mode.

Among the above 4 cases, except for the case when only browser is closed, the remaining cases require disk imaging, since it is not possible to acquire physical memory directly. Therefore, we must explore candidate cipherkeys in memory-related data such as *Hiberfil.sys*, *Pagefile.sys*, *Swapfile.sys* or in disk's unallocated area. In case when only browser is closed which means the PC is still on, then direct acquisition of physical memory is possible.

However, in case of encrypted files, when the browser is inactive, related paths are deleted, making direct acquisition impossible. Therefore, we need to perform carving on the disk image based on the encrypted file's structure shown in Section 4. When examining the DB and BLOB files separately, for BLOB files, as shown in Fig. 5, we can identify encrypted pages and its sequence using *Nonce* field within the first 48 bytes. Also, by using *Random Value* field, we can determine whether a page belongs to the same BLOB file. This allows us to reconstruct the original encrypted BLOB file by aligning the sequence, even if the file is fragmented.

Similarly, for DB files, we can identify encrypted pages using the *Nonce* field within the metadata 32 bytes as shown in Fig. 4. However, it is challenging to determine if identified encrypted pages belong to the same DB file. Also, identifying the page sequence is limited which makes it difficult to reconstruct the DB file if fragmented. Currently, attempting decryption at the extracted page level is possible, but to reconstruct the original encrypted DB file, one would need to explore various combinations. Therefore, the carving algorithm for entire DB and BLOB files are not completed, and under work.

6. Implementation and experiment

To test the decryption methodology discussed in Section 5, in various cases, the following experiment setup and procedure were implemented. The goal of this experiment is to investigate whether it is possible to extract and decrypt IndexedDB data left behind by Gecko-based browsers' private mode, specifically Firefox and Tor Browser, in various cases.

6.1. Implementation

During the experiment, a proof-of-concept (PoC) tool developed based on decryption algorithm in Section 5 was used. The tool takes, as input, a file to search for cipherkeys (typically physical memory file) and the encrypted file. The decrypted files are provided as output, and the valid cipherkey about the file found during bruteforce are printed as shown in Fig. 7. The source code and explanation of the tool is available on our GitHub repository (Esrever, 2024).

Particularly noteworthy point was that encrypted data related to Cache API (*caches.sqlite*, morgue cache files) was encrypted in the same

```
DB file - 629c24e1-5c7a-47b8-9467-ae5d276e7935.sqlite, memory file - Ubuntu 64-bit-98a4b745.vmem
DECRYPT SUCCESS!! - key is c54a0b62134d31f698eb1c36976d7326be257de184b5be5e42d2b59e0108ffb7

DB file - caches.sqlite, memory file - Ubuntu 64-bit-98a4b745.vmem
DECRYPT SUCCESS!! - key is 9bf814053e35574a27e384f4f622728719b2a7426632ba8f33867a6cd1a32041

DB file - {2351a800-2140-4af9-a2ee-df4e57b30b06}.final, memory file - Ubuntu 64-bit-98a4b745.vmem
DECRYPT SUCCESS!! - key is 84bd00e51908b9df39d2c2ee6576203d11efa7600c17235d62053ab6efbc5d77
```

Fig. 7. Output of the proof-of-concept tool developed based on the proposed methods. Target encrypted data (DB file) and memory-related data (memory file) used for searching candidate cipherkeys are displayed. If decryption is successful, the hexadecimal cipherkey value (32 bytes) is printed as underlined. Otherwise, if there is no valid cipherkey in the input memory file, no cipherkey values are printed.

way as IndexedDB. Each *caches.sqlite* and morgue caches' file structure was identical to DB and BLOB files, respectively. Only difference was that for morgue cache files, the *Length* field in cipherkey search pattern was changed from 0x01 to 0x26, since the morgue cache file names were made with unique 32 bytes UUID values. PoC tool was also successfully applied to decrypt Cache API's encrypted data.

6.2. Experimental setup

The operating system used for experiment was Windows 10 22H2 (version 19045.3930), with 16 GB RAM size. Test activities were performed on two different browsers, Firefox (v122.0) and Tor Browser (v13.0.9), both being the latest versions as of February 2024. FTK Imager (v4.7.0.1) was employed for physical memory and disk image acquisition. For preprocessing memory-related files, Hibernation-Recon (v1.2.2.86) was used for decompressing the *Hiberfil.sys* file and winmem-decompress (Msuhanov, 2018) was utilized to extract compressed pages in memory-related files.

As discussed in **Section 5**, 5 different scenarios were set up for testing against the target browsers, Firefox and Tor Browser.

- **Case 1**: When browser is active
- **Case 2**: When Private session is deactivated
- **Case 3**: When PC is shut down (No option)
- **Case 4**: When PC is shut down (Fast Startup mode)
- **Case 5**: When PC is shut down (Hibernate mode)

For each case, 20 IndexedDB files (10 DB files, 10 BLOB files) were generated by interacting with 5 widely used websites that actively utilize IndexedDB: YouTube, Google Drive, Office 365, Microsoft Teams, and Instagram. Various actions were performed on those sites to generate IndexedDB data. Subsequently, the extraction methods discussed in **Section 5** were applied to obtain the encrypted IndexedDB files and memory-related files. The PoC tool was then utilized to evaluate how many files were successfully decrypted.

On the other hand, in **Case 2** ∼ **4**, since a complete method to obtain encrypted data is not yet established when PC is shut down, the DB files were extracted before end of private session. It is assumed that the encrypted files were successfully carved from disk image.

6.3. Result

6.3.1. Test case 1

The results of the decryption tests conducted on the extracted data from Firefox and Tor Browser with private session activated are shown in Table 4. All DB and BLOB files were decrypted successfully, which means the cipherkey is maintained in the memory when the private session is active.

6.3.2. Test case 2 ∼ 5 with firefox

Table 2 shows the result of decryption tests on Firefox in **Case 2, 3, 4** and **5**. The result shows that when the private session is terminated, whether the browser is closed or the PC is shut down, the cipherkey is not retained in the memory at all. However, in the case of shutting down with the Hibernate mode, all DB and BLOB files were decrypted since entire user process area, including Firefox, is stored in the *Hiberfil.sys*.

6.3.3. Test case 2 ∼ 5 with Tor Browser

Table 3 shows the result of decryption tests on Tor Browser in **Case 2, 3, 4** and **5**. The result is consistent with Firefox, and there were no retained cipherkeys from memory-related files, except for the *Hiberfil.sys*

Table 4
Experimental result of case 1 for Firefox and Tor Browser.

Browser	DB file	BLOB file
Firefox	10/10	10/10
Tor Browser	10/10	10/10

Table 2
Experimental result of case 2, 3, 4, and 5 for Firefox.

Experiment result	RAM	Hiberfil.sys	Pagefile.sys	Swapfile.sys	Unallocated area
Case 2 (When Private Session is deactivated)	0/20	–	0/20	0/20	0/20
Case 3 (When PC is shut down with no options)	–	–	0/20	0/20	0/20
Case 4 (When PC is shut down with Fast Startup Mode)	–	0/20	0/20	0/20	0/20
Case 5 (When PC is shut down with Hibernate Mode)	–	**20/20**	0/20	0/20	0/20

Table 3
Experimental result of Case 2, 3, 4, and 5 for Tor Browser.

Experiment result	RAM	Hiberfil.sys	Pagefile.sys	Swapfile.sys	Unallocated area
Case 2 (When Private Session is deactivated)	0/20	–	0/20	0/20	0/20
Case 3 (When PC is shut down with no options)	–	–	0/20	0/20	0/20
Case 4 (When PC is shut down with Fast Startup Mode)	–	0/20	0/20	0/20	0/20
Case 5 (When PC is shut down with Hibernate Mode)	–	**20/20**	0/20	0/20	0/20

acquired from Hibernate mode shut down (**Case 5**). Only in **Case 5**, all DB and BLOB files were successfully decrypted.

7. Discussion

Test results show that if a PC with an active private session of Gecko-based browsers can be acquired, it is possible to decrypt 100% of the encrypted data related to IndexedDB. Even if the PC is shut down, using the Hibernate mode also allowed for the complete recovery of data. This means that new possibilities have opened up in the analysis of browser private mode, which was previously limited to methods such as string-search on disk or memory. However, according to the experimental results, once the browser is closed, the cipherkey could no longer be found in physical memory, aligning with the code initializing the relevant area of the browser process' memory, as shown in Fig. 3. Furthermore, none of the cipherkeys were able to be obtained from memory-related files on the disk. This suggests that hashtable area containing cipherkeys may not be affected by paging, since paging generally occurs at least-used memory areas, making it unlikely for cipherkeys used in an active browser to be paged out to disk.

This becomes our limitation of the current study. If there is no way to obtain the cipherkey after the private session ends, the potential of this study would significantly decrease since in most of the PCs we encounter in real-life, private sessions are likely to be disabled. Meanwhile, since it is not certain yet that cipherkeys are never paged on disk, further verification is needed with searching for possibilities of cipherkeys being cached on disk in various situations such as adjusting RAM size or intentionally excessing memory usage. Additionally, confirmation is required regarding the possibility of obtaining the cipherkey indirectly through the extraction of cryptographic secrets generated during the encryption process, beyond direct attempts to obtain the cipherkey.

8. Conclusion and future directions

In this paper, we aimed to comprehensively analyze user behavior in private mode by decrypting context data in client-side storages. The study involved code analysis and dynamic debugging of Gecko-based browsers, which recently started supporting IndexedDB storage on private modes as encrypted files on the disk. The research successfully found the operation mechanism and file structure of IndexedDB storage in private mode, extracted the cipherkey in memory, and decrypted IndexedDB data. Additional research revealed that Cache API related encrypted data can also be decrypted in a similar manner to IndexedDB. The novel findings above have been incorporated into a PoC tool, and revealed on GitHub. Additionally, the research proposed and experimented with analysis methodologies for various cases that can be encountered during real-life PC investigations, and explored how data can be extracted and decrypted in each case.

Through this study, various future works can be proposed. Above all, the possibility of cipherkeys being cached on disk should be confirmed to broaden this study's influence on cases when private session is ended. Besides, since methods to support client-side storage in private mode vary among browsers and operating system environments, future work can aim to explore how client-side storages are supported in private mode across different browsers and environments, including Safari based on the Webkit engine, and other environments such as Linux and mobile operating systems. Afterwards, it would be possible to test whether the decryption methodology proposed in this study with Gecko-based browsers on Windows environment can be applied in a similar manner.

On the other hand, for the Tor Browser, now new possibilities have opened up to obtain more diverse information based on client-side storage. In the experiments of this study, data was acquired using popular sites like Youtube or Google Drive which are commonly accessed in regular browsers. However, since Tor Browser is frequently used for accessing dark web sites, it may be possible to analyze user behavior within the dark web by examining how client-side storages are utilized in various dark web sites.

Algorithm 1. Extract Key Candidates From Memory

```
MEMORY ← Any types of data to search for cipherkeys (RAM, Memory-related file, Disk unallocated area and etc)
PATTERN ← Binary patterns for DB and BLOB files

function EXTRACT_KEY_CANDIDATES(MEMORY, PATTERN)
    regex ← re.compile(PATTERN)
    match_offset ← []
    key_candidates ← []

    f ← open(MEMORY, "rb")
    data ← f.read()

    for match_obj in regex.finditer(data) do
        offset ← match_obj.start()
        match_offset.append(offset)
    end for

    for idx in match_offset do
        f.seek(idx + 8)
        res ← f.read(32)
        key_candidates.append(res)
    end for

    f.close()
    sorted_key_candidates ← list(set(key_candidates))
    return sorted_key_candidates
end function
```

Algorithm 2. Bruteforce Against Encrypted Target Data

```
TARGET ← Encrypted IndexedDB file to be decrypted
KEYS ← Sorted key candidates extracted from Algorithm 1

function BRUTEFORCE_KEY_CANDIDATES(TARGET, KEYS)
    data ← TARGET.read(8192)

    ciphertext ← data[0:8160]
    nonce ← data[8160:8172]
    tag ← data[8176:8192]
    true_key ← ""

    for key in KEYS do
        decipher ← ChaCha20_Poly1305.new(key, nonce)
        decipher.update(b"")
        res ← decipher.decrypt_and_verify(ciphertext, tag)
        if ValueError occurs then
            continue
        end if
        true_key ← key
    end for

    if true_key == "" then
        break
    end if

    return true_key
end function
```

Acknowledgements

This work was supported by the Korea Copyright Protection Agency (KCOPA) grant funded by the Korea government (Ministry of Culture, Sports and Tourism) (No. 2024).

References

Alfosail, M., Norris, P.M., 2021. Tor forensics: proposed workflow for client memory artefacts. Comput. Secur. 106, 102311 https://doi.org/10.1016/j.cose.2021.102311.

Bang, S., Jin, P., Kim, D.H., Park, J., Lee, S., Park, A., Cho, B., Jung, I., 2021. A study on utilization of windows 10 hiberfil.sys file for digital forensics. Journal of Digital Forensics 15, 125–137. https://doi.org/10.22798/kdfs.2021.15.1.125.

Choi, G., Bang, J., Lee, S., Park, J., 2023. Chracer: memory analysis of chromium-based browsers. Forensic Sci. Int.: Digit. Invest. 46, 301613 https://doi.org/10.1016/j.fsidi.2023.301613.

chromium, . IndexedDB.URL: https://chromium.googlesource.com/chromium/src/+/master/content/browser/indexed_db/docs/README.md..

Esrever, 2024. Gecko IndexedDB CacheAPI decryptor. URL: https://github.com/esrever98/Gecko-IndexedDB-CacheAPI-Decryptor.

Fernández-Fuentes, X., Pena, T.F., Dominguez, J.A., 2022. Digital forensic analysis methodology for private browsing: Firefox and chrome on linux as a case study. Comput. Secur. 115, 102626 https://doi.org/10.1016/j.cose.2022.102626.

Fernández-Fuentes, X., Pena, T.F., Dominguez, J.A., 2023. Digital forensic analysis of the private mode of browsers on android. Comput. Secur. 134, 103425 https://doi.org/10.1016/j.cose.2023.103425.

Fernández-Álvarez, P., Rodríguez, R.J., 2022. Extraction and analysis of retrievable memory artifacts from windows telegram desktop application. Forensic Sci. Int.: Digit. Invest. 40, 301342 https://doi.org/10.1016/j.fsidi.2022.301342.

fingerprint, .Incognito mode detection: Detecting visitors who browse in private mode. URL: https://fingerprint.com/blog/incognito-mode-detection..

Firefox, 2023. Firefox 115.0 - release notes. URL: https://www.mozilla.org/en-US/firefox/115.0/releasenotes/.

Firefox, 2024. Firefox 122.0 - release notes. URL: https://www.mozilla.org/en-US/firefox/122.0/releasenotes/.

Hariharan, M., Thakar, A., Sharma, P., 2022. Forensic analysis of private mode browsing artifacts in portable web browsers using memory forensics. In: 2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS). https://doi.org/10.1109/ic3sis54991.2022.9885379.

Horsman, G., Findlay, B., Edwick, J., Asquith, A., Swannell, K., Fisher, D., Grieves, A., Guthrie, J., Stobbs, D., McKain, P., 2019. A forensic examination of web browser privacy-modes. Forensic Sci. Int.: Report 1, 100036. https://doi.org/10.1016/j.fsir.2019.100036.

Hughes, K., Papadopoulos, P., Pitropakis, N., Smales, A., Ahmad, J., Buchanan, W.J., 2021. Browsers' private mode: is it what we were promised? Computers 10, 165. https://doi.org/10.3390/computers10120165.

Mozilla, 2021. Gecko's StringDataFlags class. URL: https://github.com/mozilla/gecko-dev/blob/47e291435c52512f0c664a89abaf1dcab01c2f69/xpcom/string/nsStringFlags.h#L20.

Mozilla, 2024. Gecko-dev: mercurial Gecko repositories. URL: https://github.com/mozilla/gecko-dev.

Msuhanov, 2018. winmem_decompress: extract compressed memory pages from page-aligned data. URL: https://github.com/msuhanov/winmem_decompress.

Palıgu, F., Kumar, A., Cho, H., Varol, C., 2019. Browstexplus: a tool to aggregate indexeddb artifacts for forensic analysis. J. Forensic Sci. 64, 1370–1378. https://doi.org/10.1111/1556-4029.14043.

Palıgu, F., Varol, C., 2022a. Browser forensic investigations of instagram utilizing indexeddb persistent storage. Future Internet 14, 188. https://doi.org/10.3390/fi14060188.

Palıgu, F., Varol, C., 2022b. Microsoft teams desktop application forensic investigations utilizing indexeddb storage. J. Forensic Sci. 67, 1513–1533. https://doi.org/10.1111/1556-4029.15014.

Tor Project, 2023. Tor browser 13.0. URL: https://blog.torproject.org/new-release-tor-browser-130/.

web.dev, 2020. Storage for the web. URL: https://web.dev/articles/storage-for-the-web?hl=en.