Advanced Monero wallet forensics: Demystifying off-chain artifacts to trace privacy-preserving cryptocurrency transactions

By:
Jeongin Lee, Geunyeong Choi, Jihyo Han, Jungheum Park

From the proceedings of
The Digital Forensic Research Conference
DFRWS APAC 2025
Nov 10-12, 2025

Forensic Science International: Digital Investigation 54 (2025) 301988

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS APAC 2025 - Selected Papers from the 5th Annual Digital Forensics Research Conference APAC

Advanced Monero wallet forensics: Demystifying off-chain artifacts to trace privacy-preserving cryptocurrency transactions

Jeongin Lee, Geunyeong Choi, Jihyo Han, Jungheum Park *

School of Cybersecurity, Korea University, Seoul, South Korea

ARTICLE INFO

Keywords:
Digital forensics
Live forensics
Memory forensics
Cryptocurrency
Transaction tracing
Monero
Forensic tool development

ABSTRACT

Monero, a privacy-preserving cryptocurrency, employs advanced cryptographic techniques to obfuscate transaction participants and amounts, thereby achieving strong untraceability. However, digital forensic approach can still reveal sensitive information by examining off-chain artifacts such as memory and wallet files. In this work, we conduct an in-depth forensic analysis of Monero's wallet application, focusing on the handling of public and private keys and the wallet's data storage formats. We reveal how these keys are managed in memory and develop a memory scanning algorithm capable of identifying key-related data structures. Furthermore, we analyze the wallet keys and cache files, presenting a method for decrypting and interpreting serialized keys and transaction data encrypted with a user-specified passphrase. Our approach is implemented as an open-source Volatility3 plugin and a set of decryption scripts. Finally, we discuss the applicability of our methodology to multi-cryptocurrency wallets that incorporate Monero components, thereby validating the generalizability of our techniques.

1. Introduction

Cryptocurrencies do not reveal the connection between wallet owner entity and wallet address. Despite all transactions being recorded on a public blockchain ledger, this principle ensures cryptocurrency untraceability. The entity can prove ownership of his/her wallet at any time by utilizing their private key. However, entity recognition is possible by leveraging transaction properties, behavioral patterns, and off-chain data (Wu et al., 2021).

Dark coins maximize anonymity by encrypting transaction data before storing it on the public ledger. In particular, Monero, a type of dark coin, encrypts the sender, receiver, and transaction amount before storing them on the Monero blockchain. Monero's untraceability ensures that only the parties involved in a transaction can read its data. Attempts to spoil Monero's untraceability (Kumar et al., 2017; Hinteregger and Haslhofer, 2019; Möser et al., 2018) discovered the possibility of identifying sender entities, but could not fully track the Monero transaction.

Digital forensic approaches are employed to seize cryptocurrencies, including Monero, and to trace the transaction flow. Koerhuis et al. (2020) conducted a forensic analysis of artifacts generated by Monero and Verge cryptocurrency wallets, identifying and analyzing residual data critical for transaction tracing. Because cryptocurrency wallet applications store not only transaction data but also private keys on the user's device, a suspect's device serves as a valuable source of evidence in illegal transaction investigations. An investigator can use the private keys obtained from a suspect's device to prevent further movement or laundering of illicit proceeds.

The Monero project maintains two wallet applications: Command Line Interface (CLI) (Monero Project, 2025a) and Graphical User Interface (GUI) (Monero Project, 2025b). The source code of Monero CLI contains the core components necessary for Monero transactions, while Monero GUI provides enhanced usability for users. It is noteworthy that Monero GUI imports parts of the Monero CLI source code. This indicates that digital forensic approaches applicable to Monero CLI can also be applied to Monero GUI.

In this work, we analyze the source code of the Monero CLI to investigate its mechanisms for managing public and private keys. While the user-specified passphrase resides in memory as a printable string, our forensic analysis reveals that Monero CLI stores public and private keys as raw byte streams. Therefore, effective forensic investigation requires complementary strategies beyond string-based search approaches. As outlined earlier, this complementary strategy is also applied to the forensic investigation of Monero GUI.

* Corresponding author.
E-mail addresses: jjeongin@korea.ac.kr (J. Lee), geunyeong@korea.ac.kr (G. Choi), hanjihyo@korea.ac.kr (J. Han), jungheumpark@korea.ac.kr (J. Park).

https://doi.org/10.1016/j.fsidi.2025.301988

Table 1
Comparison with previous studies on Monero wallet forensics.

Work	Memory Forensics	Disk Forensics	Decrypt wallet files	Types of artifacts covered in research						Version
				PublicKeys	PrivateKeys	BlockHeight	TxID	TxKey	TxTime	
Ali et al. (2018)	✓	×	×	×	×	✓	✓	×	×	v0.11.0.0
Koerhuis et al. (2020)	✓	✓	×	✓	×	×	✓	×	×	v0.12.0.0
Our work	✓	✓	✓	✓	✓	✓	✓	✓	✓	v0.18.3.4

Additionally, the wallet files of both Monero CLI and Monero GUI are analyzed. A single wallet comprises a pair of files: a keys file (*wallet keys file*, named ⟨*walletname*⟩.keys) and a cache file (*wallet cache file*, named ⟨*walletname*⟩). While Monero CLI and a user-specified passphrase enable interpretation of the two files, the specific methodologies for decrypting and parsing these artifacts remain unaddressed. Monero uses spend keys to send funds and view keys to receive funds. The *wallet keys file* stores not only the $SpendKey_{prv}$ but also the $ViewKey_{prv}$, $SpendKey_{pub}$, and $ViewKey_{pub}$. Transaction data sent and received by a wallet is cached in that wallet's *wallet cache file*. Therefore, a proper understanding of these files is necessary.

Our contributions can be summarized as follows.

- By analyzing the source code of Monero CLI, we investigate the mechanism for managing spend and view keys in a wallet. Because Monero CLI loads all the keys into memory as raw byte streams, we developed a scanning algorithm to identify memory-resident instances of the key management class. This algorithm can also successfully scan for keys within the memory processes of Monero GUI.
- We examine the wallet files used by Monero CLI and Monero GUI to store wallet data. Both files are serialized with key data and transaction data, and then encrypted using a user-specified passphrase. We provide a method for decrypting and properly interpreting wallet files.
- We open-source a Volatility3 plugin, that scans memory dumps to identify instances managing Monero's spend and view keys, and decryption scripts, that decrypt and interpret *wallet keys file* and *wallet cache file* (MoeyEx, 2025).
- By applying our scanning algorithm to other multi-wallet applications that incorporate the Monero CLI source code, we demonstrate the coverage and versatility of our proposed approach.

This paper is organized as follows: Section 2 reviews existing work which aims to analyze cryptocurrency forensics. Section 3 provides an overview of the fundamental elements that constitute the Monero wallet, and clearly delineates the boundaries and objectives of this research. Section 4 details methods for (1) scanning memory to identify the C struct `account_base`, used by Monero to manage $SpendKey_{pub}$, $ViewKey_{pub}$, $SpendKey_{prv}$, and $ViewKey_{prv}$, and (2) decrypting and interpreting *wallet keys file* and *wallet cache file*. Section 5 demonstrates the successful detection of `account_base` instances using a proof-of-concept plugin for the Volatility Framework and validates the accuracy of the *wallet keys file*, *wallet cache file* decryption methodology. Moreover, Section 6, Section 7 present a detailed discussion of our results and outline potential directions for future research, respectively.

2. Background and related work

2.1. Privacy cryptocurrency and transaction tracing

Monero uses a CryptoNote protocol to enhance a user's privacy. For this reason, existing studies to address a Monero aim to analyze traceability on their blockchain (Möser et al., 2018; Kumar et al., 2017; Borggren et al., 2020; Hinteregger and Haslhofer, 2019). Monero records a large number of decoy inputs on the blockchain to conceal the identity of the sender. This makes it impossible to know what is a real entity in a single transaction. However, Möser et al. (2018) described the possibility of identifying the real entity among Monero transaction inputs prior to February 2017. Additionally, Kumar et al. (2017) evaluated the effectiveness of attacks on Monero's untraceability and proposed mitigation strategies. Borggren et al. (2020) demonstrated the possibility of identifying individuals and groups using machine learning (ML) models trained on simulated blockchain datasets. On the other hand, Hinteregger and Haslhofer (2019) empirically demonstrated that while a significant portion of Monero transactions remained traceable up to 2017, over 95 % of recent transactions became untraceable following protocol enhancements (RingCT, increased ring size).

This demonstrates that Monero's security policy upgrades have effectively enhanced untraceability, rendering practical transaction tracing nearly impossible when relying solely on on-chain data. Consequently, as the limitations of on-chain analysis grow increasingly apparent, a comprehensive forensic strategy integrating off-chain data, such as wallet logs, coin service usage records, and cached artifacts, has become imperative. By leveraging off-chain data that captures real-world user behavior and system interactions, this work establishes a novel framework for advancing privacy coin tracing methodologies.

2.2. Existing studies on Monero wallet forensics

Recent digital forensics studies on cryptocurrency wallets demonstrate practical artifact extraction by systematically analyzing residual data in memory and storage across software/hardware implementations.

Compared to existing research focused on Bitcoin and Ethereum, there are relatively few studies targeting the Monero cryptocurrency. Koerhuis et al. (2020) conducted a comprehensive forensic analysis of artifacts generated by Monero wallet software across memory, disk storage, and network traffic. Experimental results revealed that memory analysis exposes critical plaintext artifacts essential for wallet recovery and fund tracing. While it is stated that private keys can be obtained from encrypted wallet files, no specific decryption method is described. Ali et al. (2018) proposed a methodology for directly extracting diverse protocol structures from the memory of Monero wallet processes implementing the CryptoNote protocol. This study empirically demonstrates that memory forensics can trace actual transaction inputs and wallet activities despite Monero's on-chain anonymity design. However, the target of analysis is the old version of Monero (v0.11.0), and the latest RingCT update needs to be reflected.

2.3. Research gap in the existing literature

Table 1 compares the scope of analysis, addressed forensic artifacts, and contributions of prior studies by Ali et al. (2018), Koerhuis et al. (2020), and our work in the context of Monero wallet forensics. Our approach combines two forensic vectors: analyzing process memory and decrypting wallet files retained on disk storage. This dual methodology enables the extraction of public/private keys and transactional details, artifacts undetectable in prior studies, providing new pathways for cryptocurrency forensic analysis. This enables the tracing of cryptocurrency transactions and the acquisition of critical evidence necessary for the criminal investigations.

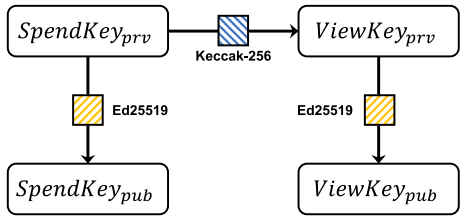

Fig. 1. Derivation of spend and view keys in Monero

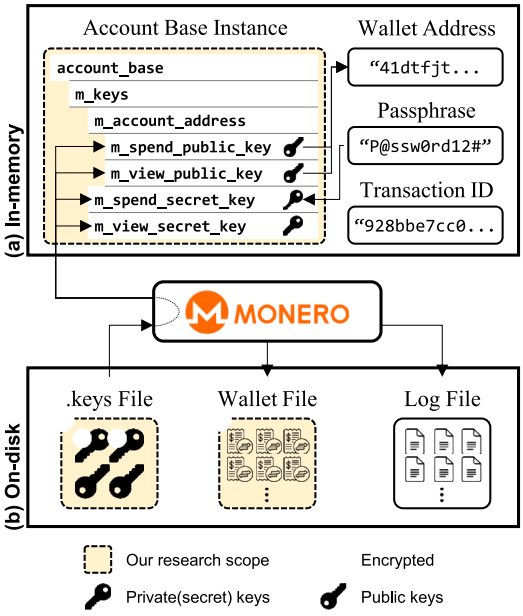

Fig. 2. Components of Monero CLI/GUI in a digital forensics aspect.

3. Components of Monero wallet and research scope

3.1. Components of Monero CLI and GUI wallets

Many cryptocurrency wallet applications store a mnemonic, which generates a seed to create private key, on the user's device. Monero's mnemonic consists of 25 words and is directly converted as a $SpendKey_{prv}$. In other words, Monero creates a mnemonic word list from $SpendKey_{prv}$ and shows it to the user. Because all keys used in Monero are derived from $SpendKey_{prv}$, it is possible to recover a Monero wallet, including $ViewKey_{prv}$, $SpendKey_{pub}$, and $ViewKey_{pub}$, if the $SpendKey_{prv}$ is found. Fig. 1 illustrates a process to derive $ViewKey_{prv}$, $SpendKey_{pub}$, $ViewKey_{pub}$ from $SpendKey_{prv}$.

3.1.1. From a memory forensics perspective

The Monero CLI handles four keys using the `account_base` class. When a user inputs the passphrase associated with the wallet they are trying to open, the Monero CLI attempts to decrypt the *wallet keys file* and copies all keys into memory buffers corresponding to the fields of the `account_base` instance (see the in-memory components illustrated in Fig. 2 (a)). These key streams are then loaded into the virtual address space of the Monero wallet process. Section 4.3 presents a technique for scanning `account_base` instances by analyzing the memory dump of a Monero CLI process. Note that the Monero GUI and CLI share the same core codebase. For example, since the `account_base` class used in the CLI is also used in the GUI, analysts can scan `account_base` instances from memory dumps of both Monero CLI and GUI processes. This finding suggests that other wallet applications, even those not based directly on the Monero CLI or GUI, may also utilize the Monero core code.

3.1.2. From a disk forensics perspective

As shown in Fig. 2 (b), when a wallet is created using the Monero CLI, two files are generated: a *wallet cache file* and a *wallet keys file*. The *wallet cache file* caches data generated during wallet usage, including transaction-related information such as incoming and outgoing amounts, fees, change, and the block height at which each transaction was confirmed. The *wallet keys file* stores all four types of keys used by Monero. Both files are encrypted with a user-specified passphrase and stored on the user's device. Section 4.4 and 4.5 describe the procedure for properly interpreting these files.

3.2. Research scope and questions

Fig. 2 shows the components of a Monero wallet application and the target data addressed in our work. On disk, the *wallet keys file* and *wallet cache file* are encrypted using a user-specified passphrase. In particular, the $SpendKey_{prv}$ and $ViewKey_{prv}$ are doubly encrypted: these private keys are first obfuscated using an XOR operation before the *wallet keys file* is encrypted and stored. In contrast, in memory, analysts can scan not only wallet addresses but also the passphrase, transaction IDs, and more. However, the $SpendKey_{prv}$ stored in memory is also encrypted and represented as a raw byte stream.

The Monero CLI and Monero GUI generate precious artifacts in both memory and on disk. However, the public and private keys stored in their memory cannot be obtained using string searching-based analysis techniques. Additionally, methods for decrypting and interpreting the *wallet keys file* and *wallet cache file* have not been addressed. Private keys are essential for seizing illicit funds from suspects.

For this reason, we first analyze how the Monero CLI handles public and private keys. As previously explained, the `account_base` class contains these keys; therefore, we propose a novel approach to scan for `account_base` instances in memory. Furthermore, by analyzing the Monero CLI source code, we try to decrypt both the *wallet keys file* and *wallet cache file*.

To demonstrate the contributions of our work, we present the following research questions.

- **RQ1:** How can raw byte stream components, such as an `account_base` instance of both Monero CLI and Monero GUI, be identified and decrypted from a memory dump?
- **RQ2:** How can the *wallet keys file* and *wallet cache file* be decrypted and meaningfully interpreted?
- **RQ3:** Can our proposed approach be applied to extract public and private keys from other cryptocurrency wallet applications that support Monero?

4. Off-chain artifacts of Monero wallet

4.1. Experimental setup

Our experiment was conducted on virtual machines configured with a 2-core CPU and varying amounts of RAM (4 GB, 8 GB, and 16 GB). The guest operating system was Windows 11 24H2 Pro (Build 26100.3775). Virtual machines were created using VMware Workstation 17 Pro (v17.5.2 build-23775571). The host system was equipped with an Intel Core i5-14600K CPU and 64 GB of RAM, running Windows 11 Pro 24H2 (Build 26100.3915). We used Monero CLI and GUI version 0.18.3.4 to create cryptocurrency wallets.

4.2. Dataset creation

We attempted to simulate as many user activities as possible using Monero CLI to replicate real-world forensic scenarios. These simulated activities include wallet creation, sending and receiving Monero transactions, wallet backup/restoration, passphrase modification, and wallet locking/unlocking. The in-memory Monero component

Table 2
Transaction list for creating dataset.

No.	Transaction ID	Sender	Receiver	Amount(XMR)
1	bb957ca1aa4a548fcb09f1ba70abc5cc90a4f85d15922bb13d40bab96cb66c6b	Test_2	Test_1	0.00996928
2	ad49c70b4e05b9f956a99523068ab9b08229168befe0e2091c97dd83489b3a39	Test_1	Test_3	0.00900000
3	ee0f5701dafee5649eeadc4f2c1a7eaa85cdb73828010d5c3ff313882a987bb3	Test_1	Test_4	0.00030000
4	af7934453a430895b2a5b6d8cbcbf683c13893e8650f01f80304ccc3a391b9c2	Test_3	Test_1	0.00200000
5	f799a90e7c8518ef60aac07846064c40c8dc6d80f024f1e4d61758cf8883cba3	Test_4	Test_1	0.00010000

Table 3
Object layout of the `account_base` class (based on 64-bits).

Offset	Offset(h)	Size	Field
0	0x0	32	m_spend_public_key
32	0x20	32	m_view_public_key
64	0x40	32	m_spend_secret_key
96	0x60	32	m_view_secret_key
128	0x80	24	m_multisig_keys
152	0x98	8	m_device
160	0xA0	8	m_encryption_iv
168	0xA8	8	m_creation_timestamp

(`account_base`) was consistently found regardless of the user activity performed. The *wallet cache file*, which caches transaction data, had new data appended whenever Monero was sent or received. Table 2 lists the Monero transactions performed for dataset generation.

```
namespace cryptonote {
 class account_base {
 private:
  struct account_keys m_keys {
   struct account_public_address
    m_account_address {
    crypto::public_key m_spend_public_key;
    crypto::public_key m_view_public_key;
   }
   crypt::secret_key m_spend_secret_key;
   crypto::secret_key m_view_secret_key;
   std::vector<crypto::secret_key>
    m_multisig_keys;
   hw::device *m_device;
   crypto::chacha_iv m_encryption_iv;
  } // end of account_keys
  uint64_t m_creation_timestamp;
 } // end of account_base
} // end of cryptonote
```

Listing 1: account_base class

4.3. Volatile data: live memory instances

4.3.1. Understanding of account_base class used for managing spend and view keys

Monero manages the $SpendKey_{pub}$, $ViewKey_{pub}$, $SpendKey_{prv}$, and $ViewKey_{prv}$ in memory through the `account_base` class, which belongs to the `cryptonote` namespace. The `account_base` class stores these four keys in its `m_keys` field. Listing 1 shows the prototype of the `account_base` class.

The `m_keys` field stores the $SpendKey_{pub}$ in `m_spend_public_key` and the $ViewKey_{pub}$ in `m_view_public_key`, respectively. In addition, the $SpendKey_{prv}$ and $ViewKey_{prv}$ are stored in `m_spend_secret_key` and `m_view_secret_key`, respectively. The `crypto::public_key` and `crypto::secret_key` types are arrays consisting of 32 one-byte elements. The `m_encryption_iv` is used for decrypting the $SpendKey_{prv}$ and is an array consisting of eight one-byte elements. Table 3 summarizes the object layout of `account_base`.

Fig. 3 shows an instance of `account_base` found in memory along with the actual key pairs. We identified discrepancies between the key pairs generated by Monero CLI and their in-memory instances. The $SpendKey_{prv}$ stored in memory as a raw byte stream did not match the actual key value, unlike the $SpendKey_{pub}$, $ViewKey_{pub}$, $ViewKey_{prv}$. We identified a critical security mechanism in Monero CLI: only the $SpendKey_{prv}$ is encrypted before being stored in memory, while other keys remain unencrypted. Notably, Monero GUI employs the same `account_base` class as its CLI counterpart but stores the $SpendKey_{prv}$ in memory without encryption, unlike the CLI's encrypted implementation.

4.3.2. Decryption of $SpendKey_{prv}$ value of an active account_base instance

The source code of the Monero CLI explains how `account_base` obfuscates $SpendKey_{prv}$. As shown in Fig. 1, the $SpendKey_{prv}$ serves as the

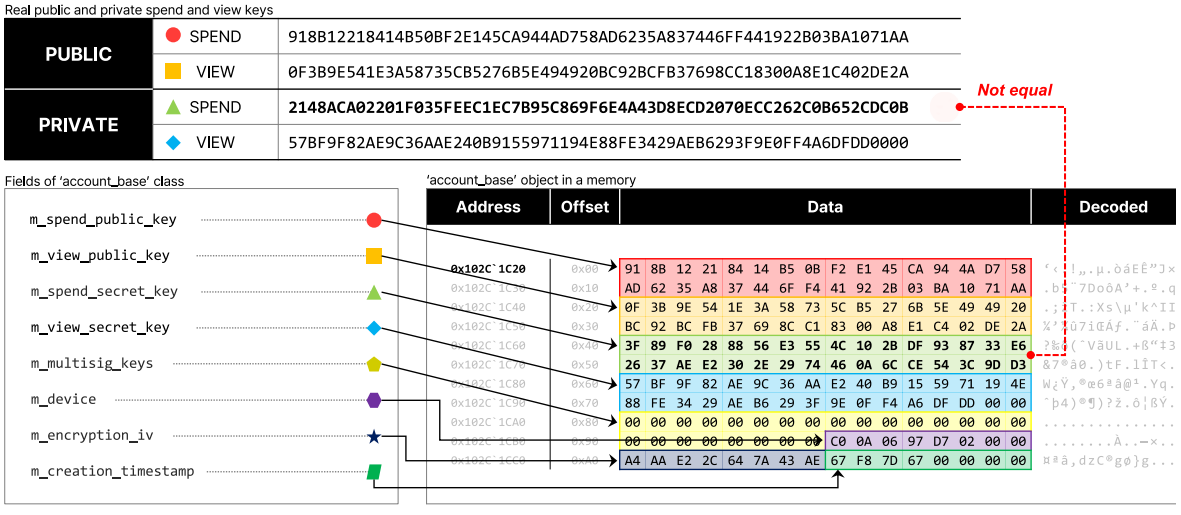

Fig. 3. Real keys and in-memory stored keys created and used by Monero; this figure shows that real $SpendKey_{prv}$ and in-memory stored $SpendKey_{prv}$ are different (they are marked by yellow-green triangle).

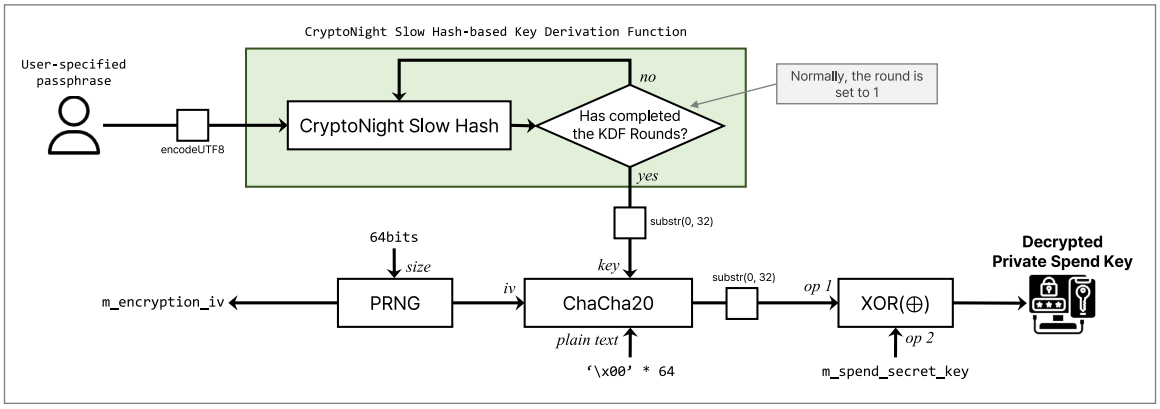

Fig. 4. Decryption process of $SpendKey_{prv}$ stored in the `account_base` instance.

root for deriving the remaining three keys. Monero CLI encrypts this sensitive data using a user-specified passphrase to ensure its protection. Fig. 4 illustrates the decryption process for the $SpendKey_{prv}$ residing in memory. The passphrase is transformed into a cipher key using the CryptoNight Slow Hash-based Key Derivation Function (KDF). Then, a 64-byte null-byte array is encrypted using ChaCha20 (Bernstein et al., 2008). The Initialization Vector (IV) is generated by a Pseudo-Random Number Generator (PRNG) and stored in the `m_encryption_iv` field of the `account_base` class. The first 32 bytes of the encrypted null-byte array are utilized as the stream cipher key for XOR encryption. Finally, the $SpendKey_{prv}$ is encrypted using the user-specified passphrase.

4.3.3. Method for efficient scanning of account_base instances

Algorithm 1. Algorithm to scan `account_base` instance

```
1:  MEMORY_DUMP ← memory dump file stream
2:  CPU_ARCH ∈ {4, 8}
3:
4:  procedure discover_account_base_instances(passphrase)
5:      results ← []
6:      for offset ← 0
          to sizeof(MEMORY_DUMP)-sizeof(account_base)
          step CPU_ARCH do
7:          data ← read(MEMORY_DUMP, offset,
              sizeof(account_base))
8:          obj ← convert_as(data, typeof(account_base))
9:
10:         if passphrase ≠ NULL then
11:             decSSK ← decrypt_spend_secret_key(
                  passphrase, obj.m_encryption_iv,
                  obj.m_spend_secret_key)
12:             if derive_public_key(decSSK)
                  ≠ obj.m_spend_public_key then
13:                 continue
14:             end if
15:         end if
16:         if derive_public_key(obj.m_view_secret_key)
              ≠ obj.m_view_public_key then
17:             continue
18:         end if
19:         if obj.m_creation_timestamp
              < unix_timestamp('2014-04-18') then
20:             continue
21:         end if
22:         add(results, obj)
23:     end for
24:     return results
25: end procedure
```

The `account_base` class is aligned to multiples of 4 or 8 bytes in memory due to alignment rules. This alignment reduces the time required to locate `account_base` instances during memory scans. Algorithm 1 presents the pseudo-code for scanning memory-resident instances of the `account_base`. This algorithm was inspired by prior work that identifies in-memory instances based on the possibility to be the given objects (Choi et al., 2023; Qi et al., 2022).

Algorithm 1 takes a memory dump as input and iteratively shifts the offset by 4 or 8 bytes (depending on CPU architecture), reading chunks of data equivalent to the size of the `account_base`(line 7 of Algorithm 1). The read data stream is reconstructed into an `account_base` object (line 8 of Algorithm 1).

Monero uses Ed25519 to generate wallets (Monero Community, 2025). We note the characteristic of Ed25519 whereby the public key is deterministically derived from the private key. For a given private key candidate stream, the correct private key can be identified by deriving a candidate public key using Ed25519 and comparing it with the actual public key. For example, as shown in Fig. 3, to verify whether a $ViewKey_{prv}$ candidate found at offset 0x102C'1C80 is the actual $ViewKey_{prv}$ managed by Monero's `account_base`, we derive a $ViewKey_{pub}$ candidate from the $ViewKey_{prv}$ candidate and compare it with the $ViewKey_{pub}$ located at offset 0x102C'1C40. If the two values match, the `account_base` instance can be successfully identified. This process is outlined in lines 10–18 of Algorithm 1. Since the $SpendKey_{prv}$ is encrypted, it must first be decrypted (line 11 of Algorithm 1). However, by validating the $ViewKey_{pub}$ derived from the $ViewKey_{prv}$ alone, our algorithm can detect `account_base` instances even without requiring a passphrase.

The detection accuracy improves as more fields are validated. We additionally verify whether the `m_creation_timestamp` is later than Monero's release date (lines 19–21 of Algorithm 1). Since Monero's first block was generated on 2014-08-14, we confirm that the `m_creation_timestamp` value is newer than this date. The `m_creation_timestamp` stores the wallet creation time as a UNIX time.

4.4. Non-volatile data: wallet keys file internals

4.4.1. Decryption with a valid passphrase

The *wallet keys file* stores the four keys that make up a Monero wallet: $SpendKey_{pub}$, $ViewKey_{pub}$, $SpendKey_{prv}$, and $ViewKey_{prv}$. If the Monero CLI or GUI is not running, these keys must be obtained from the *wallet keys file*. This file is encrypted using a user-specified passphrase, and notably, both the $SpendKey_{prv}$ and $ViewKey_{prv}$ are encrypted twice. The *wallet keys file* is organized as follows: the first 8 bytes from offset 0 represent the IV, and the remaining bytes from offset 16 contain the encrypted content. A CryptoNight Slow Hash-based KDF is used to derive the encryption key

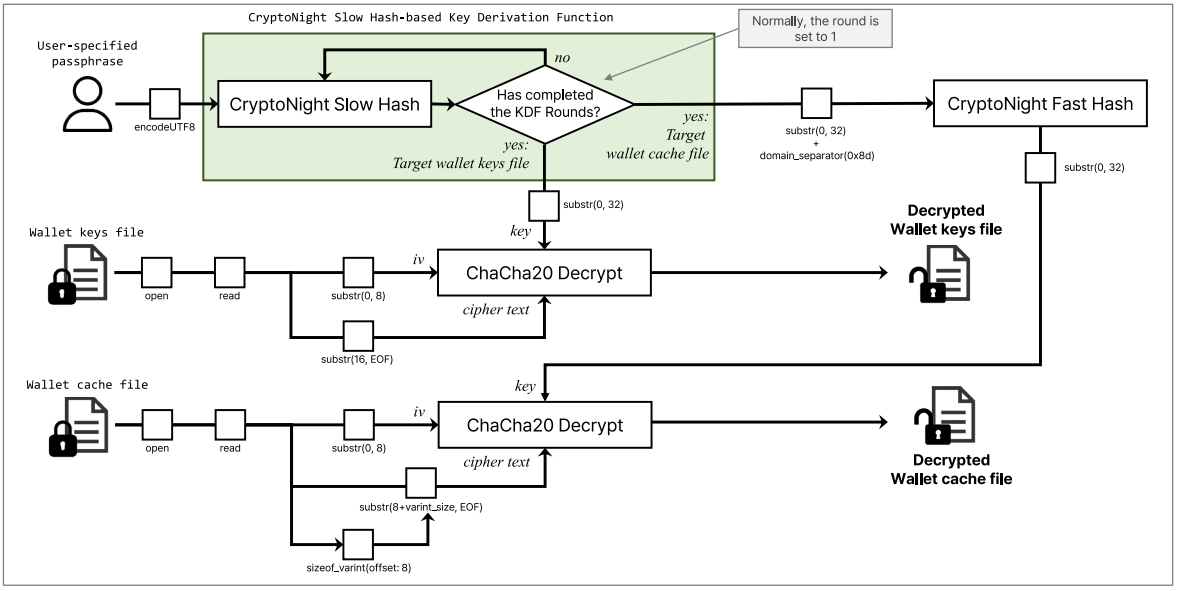

Fig. 5. Diagram of *wallet keys file* and *wallet cache file* decryption process.

from the user-specified passphrase, and the content is then encrypted using the ChaCha20 cipher. The entire process for decrypting the *wallet keys file* is illustrated in Fig. 5.

4.4.2. Deserialization of key file format

If the decryption of the *wallet keys file* is successful, the resulting output is in JavaScript Object Notation (JSON) format. Among the many JSON properties, we focus on the 'key_data' property because it contains the serialized `account_base` instance. The value of 'key_data' is serialized using the Portable Storage (PS) format (Monero Project, 2025c). The PS format is composed of a header, a section, and multiple entries. The header includes Signature Part A (0x01011101), Signature Part B (0x01020101), and a version number (0x01). The section stores the number of entries represented as key–value pairs. Each entry consists of a key, which corresponds to a field name of a C struct/class, and a value, which represents the data stored in that field.

The 'key_data' contains the same data as the `account_base` instance. The name of each entries correspond to the name of the sub-fields of the `account_base` class: `creation_timestamp`, `spend_public_key`, `view_public_key`, `encryption_iv`, `spend_secret_key`, and `view_secret_key`. It is important to note that even after decrypting the *wallet keys file*, the $SpendKey_{prv}$ and $ViewKey_{prv}$ remain encrypted. These private keys are obfuscated with the same encryption process illustrated in Fig. 4. Similar to how the in-memory $SpendKey_{prv}$ is encrypted, the $SpendKey_{prv}$ in the *wallet keys file* is XORed with a null-byte array encrypted using the user-specified passphrase. In contrast, the $ViewKey_{prv}$ in the *wallet keys file* is XORed with a 32-byte stream starting at offset 32 of the encrypted array. By re-applying the XOR operation, the plaintext private keys can be recovered.

4.5. Non-volatile data: wallet cache file internals

4.5.1. Decryption with a valid passphrase

The *wallet cache file* is a cache file that stores Monero transaction data generated by the wallet. Similar to the *wallet keys file*, the first 8 bytes at offset 0 serve as the Initialization Vector (IV) used to encrypt the *wallet cache file*. Next, the size of the encrypted *wallet cache file* contents is represented as a variable-length integer (Varint) type, followed by the encrypted data itself. The file is encrypted using ChaCha20, with the cipher key generated by combining a CryptoNight Slow Hash-based KDF and Fast Hash, using the user-specified passphrase as input. The decryption process for the *wallet cache file* is also illustrated in Fig. 5.

4.5.2. Deserialization of wallet file format

The decrypted *wallet cache file* contents consist of serialized binary data. "monero wallet cache" is a signature of the *wallet cache file*, followed by a one-byte version information field. The *wallet cache file* contains various fields, including those that store detailed transaction data. A full list of its fields is documented in the Appendix (Table. 8). Among these, we focus on the `m_payments` and `m_confirmed_txs` fields, which store incoming and outgoing transaction data, respectively.

The `m_payments` field provides the timestamp, block height, transaction ID, amount, and fee for incoming transactions. Table 4 summarizes the name and description of each field. By combining the `m_amount` and `m_timestamp` sub-fields, it is possible to determine when and how much Monero the user received.

Outgoing transaction data is provided by the `m_confirmed_txs` field. The data includes the timestamp, block height, transaction ID, amount, fee, change, and destination address. The names and descriptions of each field are listed in Table 5. To prove an arbitrary transaction in Monero, the transaction ID and the recipient's Monero wallet address are required. By combining the `m_dests` sub-field of `m_confirmed_txs` with the `m_tx_keys` field, the transaction can be verified on the Monero blockchain explorer.

5. Implementation and demonstration

5.1. MoeyEx: A Volatility3 plugin for extracting Monero's account_base instances

To demonstrate the scanning approach proposed in Section 4.3, we developed a plugin for Volatility 3, a well-known memory forensic framework (Volatility Foundation, 2025). This proof-of-concept plugin scans a given memory dump for `account_base` instances using Algorithm 1. The source code and datasets developed for this study have been uploaded to GitHub and are publicly available (MoeyEx, 2025).

5.2. Usage and results

Our proof-of-concept Volatility3 plugin attempts to decrypt the $SpendKey_{prv}$ using the inputed passphrase. When a valid `account_base`

Table 4
Incoming transaction-related sub-fields in `m_payments`

Field Name	Size(bytes)	Description
m_tx_hash	32	Transaction hash(tx id)
m_amount	varint(1–10)	Received amount(in piconero)
m_fee	varint(1–10)	Transaction fee
m_block_height	varint(1–10)	Block height
m_timestamp	varint(1–10)	Block creation timestamp

Table 5
Outgoing transaction-related sub-fields in `m_confirmed_txs`

Field Name	Size(bytes)	Description
m_tx	variable	Transaction header
m_amount_in	varint(1–10)	Before sending amount
m_amount_out	varint(1–10)	Amount after subtracting fee
m_change	varint(1–10)	Change
m_block_height	varint(1–10)	Block height
m_dests	variable	Destination address etc.
m_timestamp	varint(1–10)	Block creation timestamp

instance is found, the plugin outputs the $SpendKey_{pub}$, $ViewKey_{pub}$, $SpendKey_{prv}$, $ViewKey_{prv}$, and the creation timestamp. Fig. 6 illustrates the output generated by our plugin, while Table 6 presents the execution times corresponding to different memory dump sizes. Our algorithm reconstructs the ED25519 public key from private key candidates and compares it with the public keys stored in memory. The probability that a random candidate matches is approximately one in 2^{256}. Hence, if an `account_base` instance exists in memory, our method can reliably detect it. In addition to the plugin from a memory forensics perspective, as described in Section 4.4 and 4.5, we also developed proof-of-concept scripts to decrypt and interpret the contents of the *wallet keys file* and *wallet cache file* from a disk forensics perspective. Both scripts take a user-specified passphrase and the corresponding wallet keys/cache file as input. As shown in Fig. 7, the *wallet keys file* decryption script decrypts the file using the given passphrase and presents the four keys to the investigator. The *wallet cache file* decryption script attempts to decrypt the file using the provided passphrase. If decryption is successful, it interprets the serialized binary data and outputs Monero transaction data. Fig. 8 shows the output of the *wallet cache file* decryption script, including the transaction type (incoming/outgoing), timestamp, block height, transaction ID, transaction key, amount, fee, change, destination address and public key.

6. Discussion

6.1. Implications of findings

Our findings not only provide precious artifacts that string-search approaches may overlook, but also help analysts in identifying a valid passphrase associated with a wallet. For example, if an analyst tries to recover a passphrase from a suspect's device memory dump, by attempting to decrypt the $SpendKey_{prv}$ using candidate passphrases extracted from the Monero CLI/GUI memory dump and verifying whether the derived public key matches, the correct user-specified passphrase can potentially be identified (see lines 10–15 of Algorithm 1). Moreover, since a Monero wallet address is constructed by combining the $SpendKey_{pub}$ and $ViewKey_{pub}$, a valid wallet address can be distinguished from among multiple candidate strings. By properly understanding how data is handled by the Monero CLI/GUI, our approach helps analysts avoid unnecessary or irrelevant data.

6.1.1. Answer to **RQ1**: memory forensics for Monero

During analysis of the Monero CLI source code, we discovered that the code refers to the `account_base` class whenever private keys are used. This observation suggests that an instance of `account_base` is always created in the process memory space while the Monero CLI or GUI is running. Inspired by the object layout-based instance search approach employed in prior works (Choi et al., 2023; Qi et al., 2022), we developed an algorithm (Algorithm 1) to locate volatile components of the Monero wallet in memory. While the Monero CLI encrypts the $SpendKey_{prv}$ and stores it within the `account_base` object, the Monero GUI does not. The $SpendKey_{prv}$ is encrypted using an XOR stream key derived from a user-specified passphrase. However, since the passphrase can often be easily discovered (Koerhuis et al., 2020), the correct passphrase may be identified by attempting to decrypt the $SpendKey_{prv}$ using candidate passphrases retrieved from memory.

6.1.2. Answer to **RQ2**: disk forensics for Monero

By analyzing the Monero CLI source code, we discovered a method to decrypt both the *wallet keys file* and the *wallet cache file*. Although these files are encrypted with a user-specified passphrase, analysts may infer the passphrase by leveraging various sources, such as user passwords stored in web browser login data. Both files are encrypted using ChaCha20 with a cipher key derived from the user-specified passphrase. Decrypting the *wallet keys file* yields data in JSON format; however, the `key_data` containing all key information is serialized in the PS format, so all keys can be recovered through deserialization. The *wallet cache file* has a very complex structure but contains the full history of Monero transactions sent or received by the user. This enables extraction of more detailed information than the transaction history displayed by the Monero CLI or GUI. For example, the `m_transfers` field in the *wallet cache file* includes the RingCT mask value, which conceals the amount in outgoing transactions. However, the Monero CLI does not provide an option to output this mask value.

Table 6
Runtime duration of our proof-of-concept implementation.

Memory Dump Size (GB)	Runtime (Second)
4	357
8	1,012
16	2,351

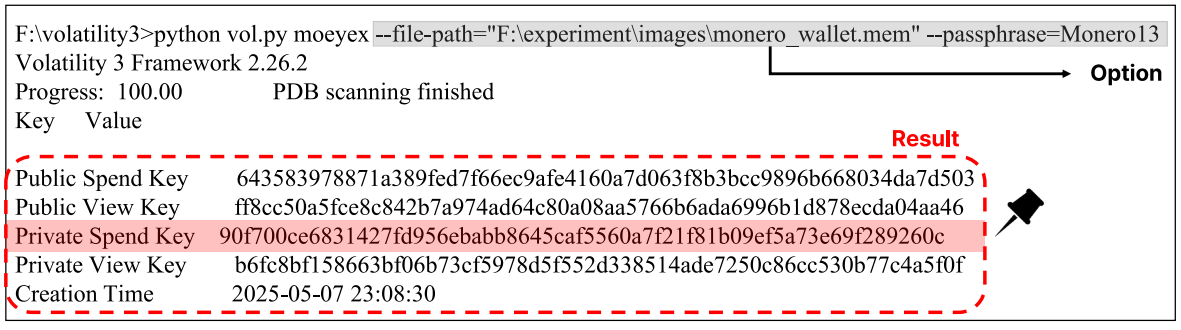

Fig. 6. An example output of MoeyEx, our proof-of-concept Volatility plugin.

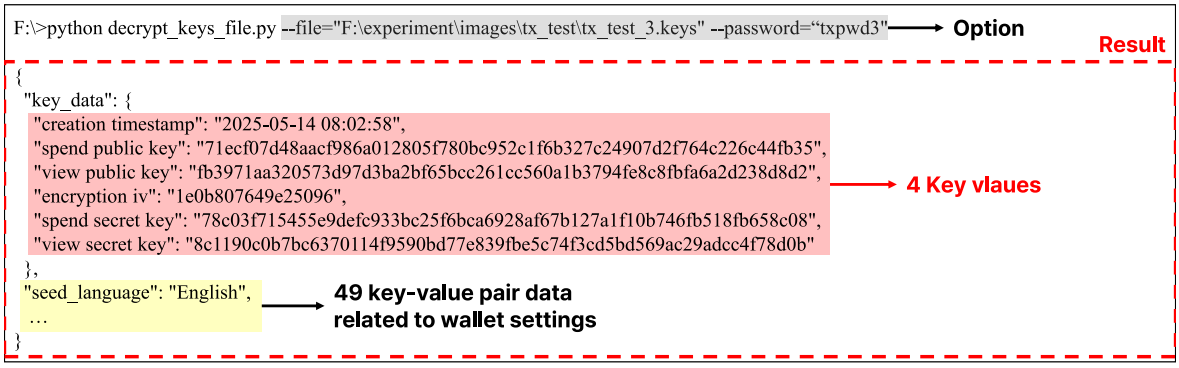

Fig. 7. An example output of *wallet keys file* decryption script included in our implementation.

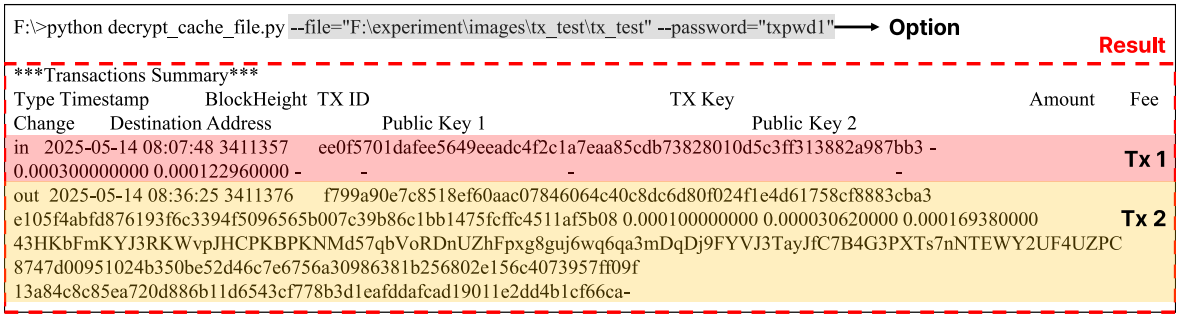

Fig. 8. An example output of *wallet cache file* decryption script included in our implementation.

6.1.3. *Answer to **RQ3**: coverage of our work*

Various multi-cryptocurrency wallet applications support the functionality for Monero transactions. Based on this, we hypothesized that some of these wallets may incorporate the Monero CLI source code to enable such functionality. To validate this hypothesis, we examined several Monero-supporting wallet applications, including Feather, MyMonero, Exodus, and Guarda, and successfully located `account_base` instances within Feather's memory space. For example, Fig. 9 illustrates an `account_base` instance identified in a memory dump of the experimental system where the Feather wallet application was executed. Notably, when examining a newly created Feather wallet, the timestamp field consistently appears as the fixed value '2014-06-07 15:00:00'. Consequently, these findings suggest that our proposed approach can be effectively applied to analyze other Monero-supporting wallets for identifying the public and private keys used in Monero transactions.

6.2. *Limitations*

First, our study revealed that the `account_base` instance could no longer be found immediately after the Monero CLI terminated. The `crypto::secret_key` used by the Monero CLI to store private keys is protected from being paged out via the `epee::mlocked` class and is zeroized through the `tools::scrubbed` class during the destruction of the `account_base` object. This indicates that our forensic approach is only applicable while the Monero CLI or GUI is still running and its memory remains resident in the system. However, the passphrase may still be retrievable even after the Monero GUI has been closed, meaning that the *wallet keys file* can still be decrypted despite the inability to scan the `account_base` instance. Therefore, investigators should consider appropriate strategies, such as collecting a physical memory image, when analyzing a suspect's device.

Second, when a wallet is restored, the *wallet cache file* only contains transaction data from after the specified restore height. As a result, information from before the restore height may not be included, which means this approach has a limitation.

Third, both volatile and non-volatile data are stored in encrypted form. Identifying the correct passphrase required for decryption remains a persistent challenge in digital forensic investigations. We consider the possibility of using a brute-force attack to discover a valid passphrase by decrypting the encrypted private key and verifying it through public key derivation. However, this approach falls outside the scope of our current study. The proposed verification algorithm could be extended to create an efficient brute-force attack system.

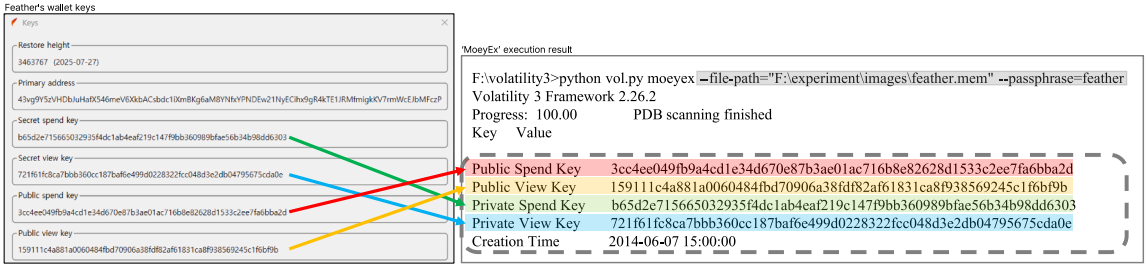

Fig. 9. An example output that shows an `account_base` instance of the Feather wallet application identified in a memory dump.

In addition, for reference, all experiments in this study were conducted within a single primary account, focusing on extracting public and private keys (rather than addresses). Since Monero subaddresses are derived from the primary account's private keys, investigators can generate them directly without needing to extract them explicitly from memory.

7. Conclusion and future directions

In this work, we identified the mechanisms by which the Monero CLI handles public and private keys through analysis of its source code. For volatile data analysis, we developed an algorithm and a proof-of-concept tool, named MoeyEx, to scan and decrypt `account_base` instances from memory dumps. Since both Monero GUI and Feather wallet applications import core modules from Monero CLI, the proposed instance-scanning approach can successfully detect the target `account_base` instance within the memory areas of both wallet applications. Additionally, our analysis of non-volatile storage demonstrates methods for decrypting and accurately interpreting the contents of *wallet keys file* and *wallet cache file*. Using these techniques, analysts can not only recover all relevant keys but also extract more detailed Monero transaction data.

For future work, we plan to conduct a more in-depth analysis of the wallet file structure, particularly in scenarios involving multisignature and subaddress features. Additionally, we aim to extend our implementation to support various operating systems and versions, such as Linux and macOS. This enhancement will enable broader data extraction and further improve the comprehensiveness of our study.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00460321, Development of Digital Asset Transaction Tracking Technology to Prevent Malicious Financial Conduct in the Digital Asset Market).

Appendix. Monero wallet's keys and cache files

Table. 7
Detailed storage format and example data of '`key_data`' value stored in a decrypted *wallet keys file*

Hex	Description
14	Length of next section key (20)
6D 5F 63 72 65 61 74 69 6F 6E 5F 74 69 6D 65 73 74 61 6D 70	Section key ("m_creation_timestamp")
05	Type code (UINT64)
29 F4 01 68 00 00 00 00	Little-Endian (2025-04-18 06:41:45.0000000 Z)
06	Length of next section key (6)
6D 5F 6B 65 79 73	Section key ("m_keys")
0C	Type code (OBJECT)
10	Number of inner section entries (4)
11	Length of first inner section key (6)
6D 5F 61 63 63 6F 75 6E 74 5F 61 64 64 72 65 73 73	Section key ("m_account_address")
0C	Type code (OBJECT)
08	Number of inner section entries (2)
12	Length of first inner section key (18)
6D 5F 73 70 65 6E 64 5F 70 75 62 6C 69 63 5F 6B 65 79	Section key ("m_spend_public_key")
0A	Type code (STRING)
80	Length of string (32)
32 46 B6 5A 87 F1 4E 4C 1E 08 AC 02 CD AC FB 02 B4 3D 86 85 58 A3 40 4C 4F 90 F4 26 13 8C 4E ED	Key of Public Spend Key
11	Length of second inner section key (17)
6D 5F 76 69 65 77 5F 70 75 62 6C 69 63 5F 6B 65 79	Section key ("m_view_public_key")
0A	Type code (STRING)
80	Length of string (32)
24 B7 14 6B E8 6D 34 7F 54 DF B0 10 AB 91 DB 7B B0 71 3C 3B EA 76 34 48 63 12 2E ED E4 2F B5 03	Key of Public View Key
0F	Length of second inner section key (15)
6D 5F 65 6E 63 72 79 70 74 69 6F 6E 5F 69 76	Section key ("m_encryption_iv")
0A	Type code (STRING)
20	Length of string (8)
CE 0A FC CD 8F AB 5A AA	Encryption of IV
12	Length of third inner section key (18)
6D 5F 73 70 65 6E 64 5F 73 65 63 72 65 74 5F 6B 65 79	Section key ("m_spend_secret_key")
0A	Type code (STRING)
80	Length of string (32)
2B 5C EC 6B 59 8C 60 80 66 08 D6 0A 12 47 E9 5B F7 D0 3A 6B E8 49 8F C6 8B 0F BF 6D E9 B1 36 B0	Key of Private Spend Key
11	Length of fourth inner section key (17)
6D 5F 76 69 65 77 5F 73 65 63 72 65 74 5F 6B 65 79	Section key ("m_view_secret_key")
0A	Type code (STRING)
80	Length of string (32)
D4 3E A4 BC 68 BB F1 B4 09 59 AF 7A 49 59 77 16 0D 3E 2F 4C A2 0C 86 73 CF 22 39C0 6D 52 30C8	Key of Private View Key

Table. 8
Top-level root fields of *wallet cache file*

Field Name	size(bytes)	Description
MAGIC_FIELD("monero wallet cache")	20	Magic string "monero wallet cache"
VERSION_FIELD(2)	1	Cache version (currently 2)
m_blockchain	variable	Blockchain hash list
m_transfers	variable	List of outputs owned
m_account_public_address	64	Public spend/view key
m_key_images	variable	Key image map
m_unconfirmed_txs	variable	Unconfirmed outgoing transaction map
m_payments	variable	Confirmed incoming transaction map
m_tx_keys	variable	Outgoing transaction key map
m_confirmed_txs	variable	Confirmed outcoming transaction map
m_tx_notes	variable	Transaction description map
m_unconfirmed_payments	variable	Unconfirmed incoming transaction map
m_pub_keys	variable	Public key map
m_address_book	variable	Address book map
m_scanned_pool_txs[0]	variable	Scanned mempool transaction
m_scanned_pool_txs[1]	variable	Scanned mempool transaction
m_subaddresses	variable	Subaddress index list(major index, minor index)
m_subaddress_labels	variable	Subaddress label map
m_additional_tx_keys	variable	Additional transaction key
m_attributes	variable	Wallet attributes
m_account_tags	variable	Account labels
m_ring_history_saved	variable	Ring signature history
m_last_block_reward	8	Last mining reward
m_tx_device	variable	Hardware wallet transaction data
m_device_last_keysync	8	Hardware wallet last key image synchronization time
m_cold_key_images	variable	Cold wallet key image map
m_has_ever_refreshed_from_node	1	Node synchronization status
m_background_sync_data	variable	Background synchronization data

References

Ali, S.S., ElAshmawy, A., Shosha, A.F., 2018. Memory forensics methodology for investigating cryptocurrency protocols. In: Proceedings of the International Conference on Security and Management (SAM), the Steering Committee of the World Congress in Computer Science, Computer, pp. 153–159.

Bernstein, D.J., et al., 2008. Chacha, a variant of salsa20. In: Workshop Record of SASC, Lausanne, Switzerland, pp. 3–5.

Borggren, N., Kim, H.y., Yao, L., Koplik, G., 2020. Simulated blockchains for machine learning traceability and transaction values in the monero network. arXiv preprint arXiv:2001.03937.

Choi, G., Bang, J., Lee, S., Park, J., 2023. Chracer: memory analysis of Chromium-based browsers. Forensic Sci. Int.: Digit. Invest. 46, 301613.

Hinteregger, A., Haslhofer, B., 2019. An empirical analysis of monero cross-chain traceability. arXiv:1812.02808.

Koerhuis, W., Kechadi, T., Le-Khac, N.A., 2020. Forensic analysis of privacy-oriented cryptocurrencies. Forensic Sci. Int.: Digit. Invest. 33, 200891.

Kumar, A., Fischer, C., Tople, S., Saxena, P., 2017. A traceability analysis of monero's blockchain. In: Foley, S.N., Gollmann, D., Snekkenes, E. (Eds.), Computer Security – ESORICS 2017. Springer International Publishing, Cham, pp. 153–173.

MoeyEx, 2025. Moeyex. URL: https://github.com/jeong0000/MoeyEx.

Monero Community, 2025. Edwards25519 Elliptic Curve. URL: https://docs.getmonero.org/cryptography/asymmetric/edwards25519/.

Monero Project, 2025a. monero. URL: https://github.com/monero-project/monero.

Monero Project, 2025b. monero-gui. URL: https://github.com/monero-project/monero-gui.

Monero Project, 2025c. Portable storage format. URL: https://github.com/monero-project/monero/blob/master/docs/PORTABLE_STORAGE.md.

Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., Christin, N., 2018. An empirical analysis of traceability in the monero blockchain. arXiv:1704.04299.

Qi, Z., Qu, Y., Yin, H., 2022. LogicMem: Automatic profile generation for binary-only memory forensics via logic inference. In: Proceedings of the Annual Network and Distributed System Security Symposium. NDSS'22).

Volatility Foundation, 2025. Volatility 3. URL: https://github.com/volatilityfoundation/volatility3.

Wu, J., Liu, J., Zhao, Y., Zheng, Z., 2021. Analysis of cryptocurrency transactions from a network perspective: an overview. J. Netw. Comput. Appl. 190, 103139.