Memory Forensics Against Ransomware

Pranshu Bajpai and Richard Enbody
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
{bajpaipr,enbody}@msu.edu

***Abstract*—Ransomware leverages the unique knowledge of the cryptographic secrets, such as an encryption key, for ransom extraction. Therefore, acquiring the decryption key via exploitation of weak cryptographic implementations or side-channel attacks allows data restoration without the requirement of ransom payment. In this paper, we examine the effectiveness of physical memory forensics against ransomware to recover raw symmetric and asymmetric keys and demonstrate file decryption against several real-world ransomware. Furthermore, we deploy our own virulent ransomware that are equipped with an effective hybrid cryptosystem to explore the limits of such memory-based side-channel attacks on ransomware. Our results indicate that cryptographic keys can be discovered during encryption in the ransomware process memory for durations long enough to facilitate complete data recovery.**

I. Introduction

Ransomware are malicious cryptographic software which perform unauthorized encryption of data on the host such that decryption is infeasible without the correct cryptographic material (keys). Strong cryptosystems [1] are deployed to acquire the leverage needed for ransom extraction. Key management is a crucial part of all variations of effective cryptosystems observed in modern ransomware [1].

Ransomware developers deploy standard encryption algorithms, such as AES and RSA, making decryption infeasible without ransom payment. As ransomware have risen to become the top security concern in the current IT space, several solutions have been proposed against this formidable threat. However, a vast majority of these solutions attend to the prevention and detection of ransomware. In contrast, this paper proposes a response and recovery technique to neutralize a ransomware attack *after* the ransomware is successfully executing on the host.

The primary insight of our work is that cryptographic keys deployed by the ransomware are exposed in physical memory during the process of encryption on the host. Since this physical memory is a *whitebox* to the host [2], it becomes feasible to extract decryption keys from the ransomware process memory during execution. The extracted keys can be used for data recovery and provide a viable alternative to ransom payment.

The main technical challenge encountered during this work is the volatility of physical memory. Contents of physical memory are highly ephemeral in nature which can complicate key extraction. We overcame the related challenges by writing an efficient key extraction utility that scans a suspicious process's memory space for the presence of symmetric and asymmetric keys. We validated the accuracy of the extracted keys by deploying them in subsequent data decryption and verifying the decrypted data against the original plaintext.

Our results demonstrate successful data recovery against known families of real-world ransomware. We conclude that the extraction of cryptographic keys can be a feasible recovery strategy against novel ransomware.

The rest of this paper is organized as follows. Related works and our contributions are discussed in Section II. The ransomware kill chain and relevant constraints are introduced in Section III. The theory of key exposure in memory is presented in Section IV. Our methodology is described in Section V, while the relevant results are presented in Section VI. Finally, we conclude the paper in Section VII.

II. Related work and our contributions

The NIST cybersecurity framework proposes five core functions that need to be concurrently performed to attain the desired state of security.[1] These core functions are: *identify*, *protect*, *detect*, *respond*, and *recover*. In the context of ransomware, the bulk of the defense efforts have focused on the identification, protection and detection functions [3]. For instance, anti-malware software and firewalls serve the protect and detect functions. The menace of ransomware is a critical issue that necessitates a *defense-in-depth* strategy incorporating *all* NIST functions. Particularly, research efforts are needed towards response and recovery against ransomware. Presently, complete and regular backups provide the only feasible solution facilitating recovery against ransomware. However, as the issue of ransomware persists, it is essential to develop alternative response and recovery solutions. Backups cannot be solely relied upon since in theory, backups offer the perfect solution, but in practice, backups are often unavailable, partial, and infrequent. Apart from backups, successful "piracy" of ransomware's encryption keys is the only viable recovery solution.

The intricacies of key management in the hybrid encryption model deployed by the most virulent Category 6 ransomware have been discussed by Bajpai *et al.* [1]. This hybrid cryptosystem, summarized in Algorithm 2, is a combination of symmetric and asymmetric encryption most commonly observed in ransomware. Here, symmetric algorithms, such as

[1] www.nist.gov/cyberframework

TABLE I: Comparison of proposed solutions against ransomware

Proposed solutions	NIST functions	Constraint
R-Locker [9]	Detect, Respond	C_4
PayBreak [13]	Detect, Recover	C_6
ShieldFS [11]	Detect, Respond	C_4
UNVEIL [12]	Detect, Respond	C_4
CryptoLock [10]	Detect, Respond	C_4
Redemption [8]	Detect, Respond	C_4

AES, are deployed in bulk data encryption and asymmetric algorithms, such as RSA, for symmetric key protection.

Extraction of cryptographic keys from system memory has been previously discussed. Shamir and Van Someren [4] proposed randomness tests to identify memory regions suspected of containing key material. Pettersson [5] presented techniques to recover cryptographic keys from Linux memory dumps. Kaplan [6] provided a comprehensive review of extracting keys from memory. Halderman *et al.* [7] provide a methodology to recover AES keys in the presence of bit errors from decaying physical memory. The authors demonstrated cold boot attacks on disk encryption software such as `TrueCrypt` and `FileVault`. However, to the best of our knowledge, there are no previous works on extracting keys from ransomware process memory.

Several solutions have been proposed against ransomware and a comparison of these existing solutions is summarized in Table I. Most solutions such as Redemption [8], R-Locker [9], CryptoLock [10], ShieldFS [11], and UNVEIL [12] focus on the behavioral patterns of ransomware. Such behavioral approaches risk overlooking novel ransomware that deviate from expected patterns that are observed in the training data (false negatives). In parallel, a high number of false positives resulting from legitimate file encryption and compression software causes *alert fatigue* in users. R-Locker is a honeyfile-based approach to ransomware that traps ransomware by placing dummy files on the host. Modification of these dummy files is then treated as suspicious activity. However, effective locations in the system for such honeyfiles are not clear. Choice of the directory where these honeyfiles are planted requires careful consideration since false assumptions will permit a ransomware to encrypt a significant portion of the disk before an alarm is triggered. While a solution could be to place these honeyfiles in all directories, this causes clutter that is intolerable in most practical environments.

Attacking ransomware's key management offers a promising, resource-efficient solution against ransomware [3]. PayBreak [13] attempts duplication of the keys generated on the host by the ransomware. However, this solution can be easily circumvented by generating keys outside the host's domain [1].

Briefly, our paper makes the following original contributions:

- Explicit identification and enumeration of the constraints that govern ransomware behavior.
- Detailed analysis of CryptoAPI calls frequently observed in ransomware.
- Empirical evaluation of expected key exposure durations in system memory.
- Demonstration of a successful side-channel attack towards file recovery against various real-world ransomware.

TABLE II: Kill chain observed in potent ransomware.

C_n	Condition	Description
C_1	Infiltration	Ransomware's initial entry into the host machine
C_2	Execution	Execution privileges to infect the host
C_3	Preparation	Process injection or hijacking; generation of cryptographic secrets
C_4	Enumeration	Identifying valuable files and resources on the host
C_5	Encryption	Modifying the host system to bring it to an infected state
C_6	Protection	Protecting cryptographic secrets to maintain leverage over the victim
C_7	Extraction	Maintain payment channel to acquire the ransom
C_8	Restoration	Reinstate the host to the state before infection

III. Identifying constraints on ransomware

The set S represents the ransomware *kill chain* such that all fundamental constraints, $C_n \in S$, bind ransomware operations as detailed in Table II. These constraints are *necessary and sufficient* conditions for all ransomware and facilitate building effective solutions since attacking one or more of these constraints severely debilitates our adversary's attack model. For instance, maintaining the secrecy of the symmetric keys used for encryption is a fundamental constraint since key leakage results in the disruption of the ransomware kill chain. In Table I, we evaluate the solutions proposed against ransomware in light of these constraints and the NIST Cybersecurity Framework. Note that $\{C_7, C_8\}$ serve the financial interests of ransomware operators and are not constraints on the cryptographic functionality of the ransomware. Ultimately, all ransomware abide by these constraints and all effective solutions *must* violate one or more of these constraints.

IV. Key exposure in memory

Encryption routines, similar to other programs, require data to be loaded in physical memory during execution for optimal performance. This data includes the symmetric and asymmetric keys deployed by the ransomware.

A. Symmetric keys

In the case of Feistel ciphers (e.g. Blowfish, Triple DES, Serpent, Twofish) and AES, this data includes *key schedules* – an array of *round keys* that are computed from the AES key and are used in different rounds of the encryption process. Figure 1 shows the round keys that are precomputed from the

AES key. These key schedules have specific patterns that can be located in memory. In the case of an AES-128 key, the first 16 bytes are the actual key and the 112 bytes that immediately follow this key are the round keys derived from the actual key. Hence, we scan blocks of memory while evaluating the blocks against expected properties of a key schedule. If the expected key schedule bytes match the actual bytes observed in memory, then we have located a key. Our approach towards locating symmetric keys in memory is further discussed in Section V.

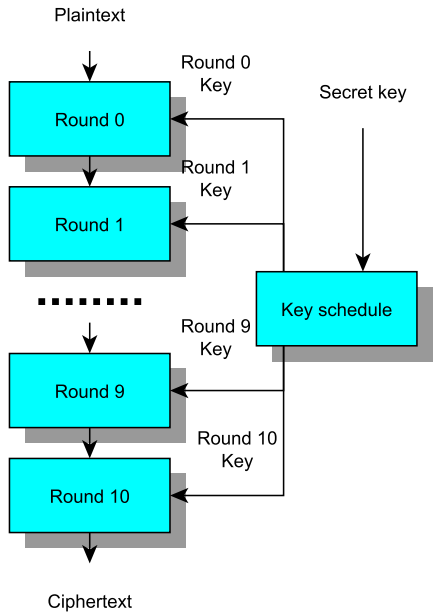
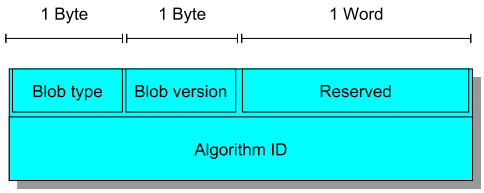


Fig. 1: AES-128 key schedule

B. Asymmetric keys

In the case of asymmetric keys, our search simplifies due to the presence of well-known structures encapsulating the actual keys. The infamous WannaCry ransomware deploys a pair of victim-specific RSA keys that are generated on the host post-infection. A *set* of symmetric encryption keys, $S_k = \{k_0, k_1, \ldots k_n\}$, is created. Each of these keys pertain to a file encrypted on the host. Following file encryption, WannaCry encrypts this set S_k with an RSA public key, P_k, generated on the host such that only the corresponding private key, P_s, can decrypt this set of encryption keys. This private key, P_s, is then encrypted with the attacker's second RSA public key, P_{Ak}, that ships with the ransomware. In the absence of implementation flaws, the cryptosystem is unbreakable. Note that the second layer of RSA key pair was likely introduced so that the operators can decrypt and release the single, unique key, P_s, to the victim upon ransom payment since this victim-specific RSA key pair was generated on the host and is unique to the host.

Clearly, possession of P_s allows us to decrypt S_k, which is then used to decrypt the data. Locating the binary key blob



(a) Asymmetric key blob structure in memory

```
0x030A0828  07 02 00 00 00 a4 00 00 52 53 41 32  .....¤..RSA2
0x030A0834  00 08 00 00 01 00 01 00 61 35 63 f3  ........a5có
0x030A0840  e8 7f fc b5 81 34 ab 72 53 d0 0d c7  è.üµ.4«rSÐ.Ç
0x030A084C  11 65 75 00 ef 1f 97 d8 ab 0b 08 dc  .eu.ï.—Ø«..Ü
0x030A0858  97 c1 1a 35 ae 48 fd 77 13 52 26 43  —Á.5®Hýw.R&C
0x030A0864  30 51 10 14 81 f9 ab 4c 3f fb 63 01  0Q...ù«L?ûc.
0x030A0870  f1 ee df 25 cf 1c ad 57 25 7b dd bd  ñîß%Ï..W%{Ý.
0x030A087C  1c 5f 9e 4f 4c 8b f3 0b 57 4c 27 df  ._žOL.ó.WL'ß
0x030A0888  23 59 f2 8a ec 1c df 97 5a bd b0 48  #Yòšì.ß—Z.°H
0x030A0894  84 56 ea 7e 42 1e 10 44 99 74 9e f9  .Vê~B..D™tžù
0x030A08A0  06 71 d3 98 e0 75 94 87 3b f5 0f c7  .qÓ˜àu”.;õ.Ç
0x030A08AC  2b 53 c4 ee ee f6 44 15 64 b4 4b 77  +SÄîîöD.d´Kw
0x030A08B8  15 bc ac c7 f2 4a 35 07 a9 73 52 53  ..¬ÇòJ5.©sRS
0x030A08C4  7d f3 13 c6 33 da 09 8b d1 af fe 6a  }ó.Æ3Ú..Ñ¯þj
```

(b) RSA private key in memory

Fig. 2: RSA private key observed in ransomware process memory

pertaining to P_s in memory hence becomes a lucrative target. Once exported, this key blob is held in a PRIVATEKEYBLOB structure in memory and the unique header bytes, illustrated in Figure 2a, surrounding the key become a signature for our search algorithm. Figure 2b shows the victim-specific RSA private key blob held in the ransomware process memory. The public key blob has a similar signature and we are able to extract both from memory.

V. Methodology

Conventional implementations of cryptographic algorithms are highly susceptible to side-channel attacks when the encryption is performed on an adversarial system. Cryptographic keys must be held in memory during encryption and a hostile entity controlling the host can implement a *white box attack* [14] for key recovery. As previously stated, an encryption routine loads data into memory for efficient cryptographic operations. For the calculation of $A \oplus B$, A and B must be in memory during the calculation for optimal performance. Thus, our methodology targets key extraction during the process of malicious encryption. Note that the knowledge of the symmetric key is sufficient for decryption (Kerckhoff's principle [15]). Therefore, it is assumed that the Initialization Vector (IV) is known since the IV is not meant to be a secret [16]. Specific to ransomware, IV may come embedded within the ransomware binary or can be a default value.

A. Static versus dynamic memory analysis

Locating the AES keys in memory can be done in two ways: 1) obtaining a memory dump (a static snapshot of the physical memory area pertaining to the suspected process)

and saving it for later analysis, or 2) searching through a live process's memory space dynamically while the process is running. Both approaches require a *trigger* condition to initiate the memory capture and have advantages and disadvantages. Acquiring a memory dump for later analysis requires storage space since memory dumps can grow to become several megabytes. Alternatively, searching for an AES key in memory dynamically eliminates the need for storage but presents a potential race condition where due to the volatility of memory, AES keys appear and disappear in a memory region while the algorithm is searching other regions. During this work, we chose 2) over 1) and performed a live memory scan made feasible with efficient memory scanning.

B. Establishing a trigger condition

Constant and impartial memory search for all processes is clearly inefficient and undesirable. Therefore, memory is scanned or dumped only when certain *trigger conditions* become true. A trigger condition in essence identifies signs of ransomware-like activity on the system. We used suspicious API call graphs as a trigger condition during experimentation. API call graphs were derived from real-world ransomware. For instance, ransomware frequently use calls such as `CryptGenKey`, `CryptImportKey`, `CryptEncrypt`, and `FindNextFile` for file enumeration and encryption (as shown in Algorithm 2). While this trigger can generate false positives during legitimate encryption, our proposed solution is tolerant of these false positives due to the transparent key extraction detailed in Section VI. Ultimately, this work focuses on the viability of key extraction from ransomware's process memory and the trigger condition is replaceable as a modular component.

C. Filling the gap with key extraction from memory

Since existing solutions cannot classify a previously unseen process as ransomware with certainty, there exists a gap between detection of a potential ransomware process and effectively neutralizing that threat with minimal system disruption. In order to fill this gap, we propose extracting keys from memory such that in the case of a false positive (incorrect classification as a ransomware process), we have only transparently extracted key(s) from the process's memory space without being disruptive on the host. Such key extraction is a measured response and is less intrusive and aggressive than killing the process and deleting associated files. Killing a suspicious process is an aggressive, potentially disruptive, action that is undesirable in favor of system stability and reliability. Trigger conditions are not perfect. Heuristical identification of ransomware tends to generate a sizable amount of false positives due to which killing the suspected process is ill-advised. Ultimately, the decision to kill a process should lie with the system user and be more *manual* rather than based on automated heuristics unless a ransomware threat can be identified with a high degree of certainty (e.g. with signature-based detection). There is a delicate balance between system security and user experience and aggressive security responses, such as killing suspicious processes, should be avoided. Similarly, generating alerts for the user continually based on false ransomware identification will lead to alert fatigue.

D. Experimental setup

Our experimental setup comprised of a Windows 10 host sandboxed in a virtual machine (2GB RAM + i7, 4 cores) infected with real-world ransomware acquired from online malware repositories [17] [18] [19]. To prove the efficacy of our approach, we performed tests pertaining to AES and RSA key extraction from memory, since these are the most commonly observed algorithms deployed in modern ransomware [1]. Mainly, we tested the following uncertainties around key extraction from memory: 1) consistency with which we can identify symmetric keys, 2) duration of key exposure (varies based on ransomware's implementation, language, and the size of the file being encrypted), 3) size of the scanned memory space, and 4) speed of memory search.

E. Identifying the keys in memory

Contrary to asymmetric keys, symmetric keys by themselves have no structure or header that can be identified in memory. AES keys in memory are simply a collection of raw bytes with no structural signatures. Hence, locating symmetric keys in memory poses a challenge. Tests of randomness as performed by Shamir and van Someren [4] are prone to false positives since certain data such as compressed files in memory contain blocks of seemingly-random data.

We identified symmetric keys in memory using their relevant key schedules. The key schedule (Figure 1) is the actual block used for encryption and is computed from the key. In the case of AES, this key schedule is 176, 208, or 240 bytes in memory for AES keys of size 128, 192, or 256 bits respectively [20]. We perform a search (`ReadProcessMemory`) for this key schedule as shown in Algorithm 1.

Algorithm 1: Search for AES key schedule

```
Data: Process memory bytes
Result: Symmetric key
initialization;
while not end of process memory do
    read currentBytes(offset);
    compute keyScheduleBytes;
    if currentBytes + nextBytes = keyScheduleBytes
     then
        foundKey = True;
        store(currentBytes);
    else
        offset = offset + keySize ;
```

For brevity, we have shown experiments pertaining to the extraction of AES and RSA keys. However, similar techniques can be applied to discover keys for other symmetric ciphers such as *Serpent*, *TwoFish*, *Triple DES* and *Blowfish*. This is

because these ciphers all contain pre-computed round keys as part of the key schedule that is held in memory to facilitate encryption. For the following discussion, we have chosen the symmetric ciphers that competed in the NIST AES selection process [16].

The Serpent algorithm performs a well-known key expansion that is deterministic. The relevant round keys are 132 words that are 32 bits each and are derived from a 256 bit Serpent encryption key [21]. Hence, the user supplies the initial encryption key, K, which is padded to 256 bits if needed. Therefore, we can assume that the Serpent keys are 256 bits in memory. Next, this user-supplied key is expanded to 33 (128-bit each) subkeys $K_0, K_1, ...K_{32}$ as shown in Figure 3. The Serpent key schedule is similar to that of AES in that the subkeys, or round keys, are derived from the user-supplied master key and its S-boxes. Thus, in order to extract the Serpent encryption key, we consider all 560 byte arrays in memory. The first 256 bits of this array are treated as the user-supplied master key and 33 round keys are derived from this master key. If the remaining bytes in the memory chunk under consideration match the expected round keys, then we have found the Serpent encryption key. The error correcting properties of the AES key schedule are also applicable for Serpent, consequently key extraction becomes similar.

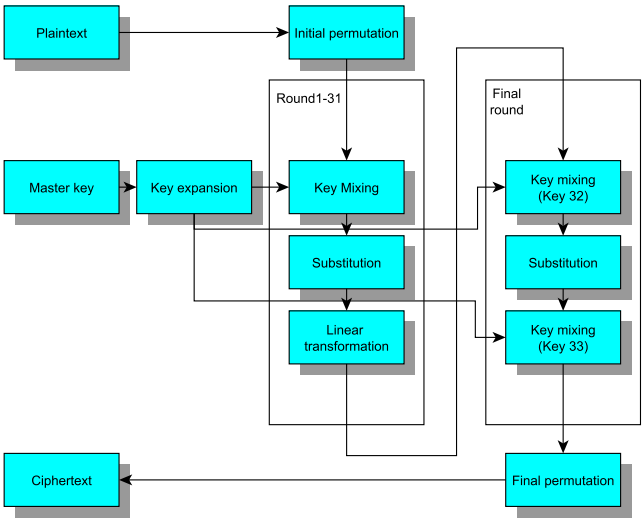


Fig. 3: Serpent encryption routine

Similar to Rijndael, Twofish master keys can be 128, 192, or 256 bits. Twofish formulates key-dependent S-boxes that have 40 (32 bit each) round keys $K_0, K_1, ...K_{39}$. The keying material hence produced may span multiple pages in the physical memory due to the large size. This makes identification of Twofish keys convoluted in memory. However, other approaches [22] have been proposed to successfully extract Twofish keys from memory.

RSA keys are extracted from memory based on the blob header structure as shown in Figure 2a and described in Section IV.

F. Protection of extracted symmetric keys

The symmetric keys that are extracted from memory require proper key hygiene such that they are not exposed (*confidentiality*) or modified (*integrity*) or deleted (*availability*) at any time. In the case of a true positive, we have extracted symmetric keys pertaining to a ransomware process and need to protect these keys such that they are available when required for decryption. Consequently, we need to ensure that they are preserved in an offsite storage away from the ransomware-infected host's domain. In the case of a false positive, we have extracted a benign process's keys from memory. This may include, for example, disk encryption keys. Naturally, protecting the confidentiality of these keys is paramount to the practical application of our solution. We propose encrypting all extracted keys with the user's pre-generated master RSA public key such that in the event of a ransomware incident *only* the user is able to decrypt and retrieve the keys from the offsite location using the corresponding private key. Therefore, the proposed system eliminates any privacy and confidentiality concerns of the user. This system to secure the extracted keys is shown in Figure 4.

Algorithm 2: Hybrid encryption scheme in ransomware (API calls shown in blue).

```
Result: Data encryption using standard algorithms
hProv = CryptAcquireContext();
publicKey = CryptImportKey();
symmetricKey = CryptGenKey();
while FindNextFile() do
    if fileType in F then
        encryptFile(hProv, symmetricKey);
        encryptedsymKey =
         encryptKey(symmetricKey,publicKey);
        DeleteFile();
    end
end
CryptDestroyKey(symmetricKey);
LocalFree(publicKey);
ransomNote();
```

VI. Results

Key exposure durations in memory were first tested to determine feasibility of dynamic extraction. Accordingly, we simulated Category 6 ransomware attacks for ransomware written in different languages such as C++, Python, Java, and Go, and displayed timestamps pertaining to key generation and key destruction in memory. Since real-world ransomware will not allow such time-stamping, we tested against our own variants of effective, Category 6, hybrid ransomware. These ransomware all followed a *multi-key* approach. A multi-key approach is one where ransomware uses different AES keys to encrypt different files such that no two files are encrypted with the same AES key. Multi-key approach reduces key exposure time if the ransomware follows proper key sanitation and

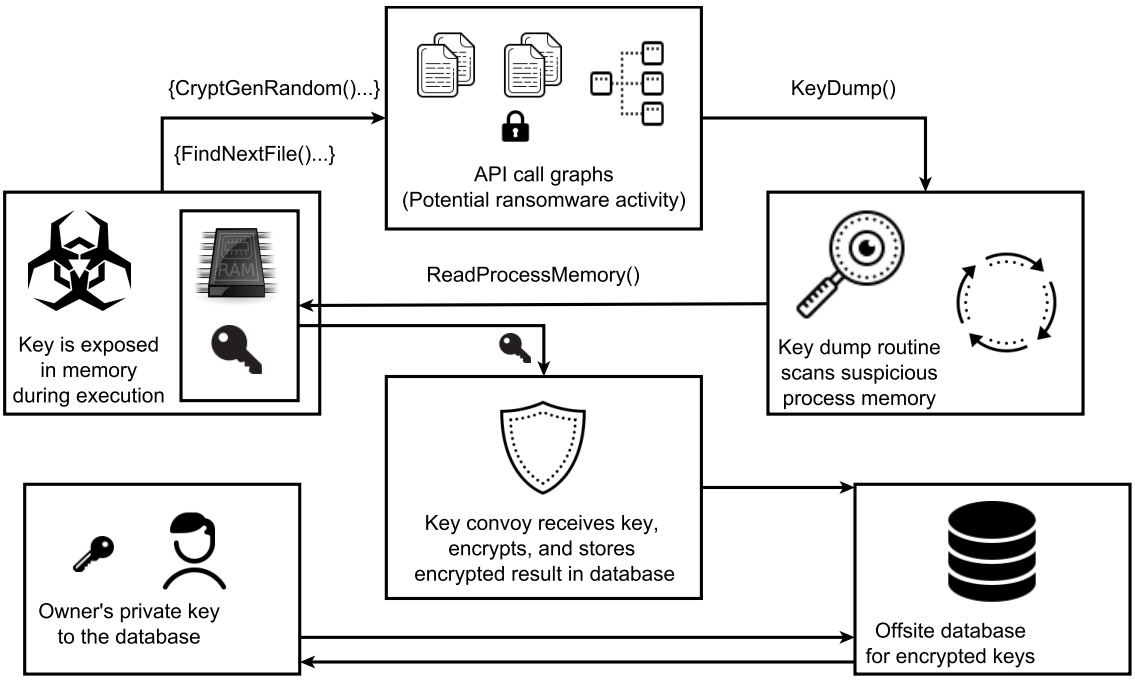


Fig. 4: Protecting extracted symmetric keys

wipes key from memory immediately following each file's encryption. Encryption time is directly proportional to file size and therefore AES keys are exposed in memory longer for larger files. Figure 5 shows the minimum amount of time that an AES encryption key is exposed in memory pertaining to varying file sizes during encryption. This includes time taken towards: 1) generation of a fresh AES key, 2) file encryption, and 3) encryption of the key with an RSA public key that has been previously imported by the ransomware. As expected, compiled languages, such as C++, performed encryption faster than interpreted languages, such as Python, and thus exposed keys for shorter durations during encryption.

A. *Testing on real-world ransomware*

We tested the efficacy of our approach against the real-world ransomware shown in Table III. Testing posed challenges when a ransomware variant depended on communication with the Command-and-Control (C&C) server since unavailability of the C&C server prevents access to the encryption key which results in a dysfunctional malware. Table III lists the ransomware variants for which we were able to locate the encryption key in memory and decrypt all files. Testing for a wider set of ransomware will yield the same results while the test parameters are unchanged. Since all ransomware will expose encryption secrets in memory during encryption, our results can be scaled to other variants.

The `LockCrypt2.0` ransomware follows the cryptosystem summarized in Algorithm 2. This ransomware reuses the same symmetric key for all file encryption, thus exposing

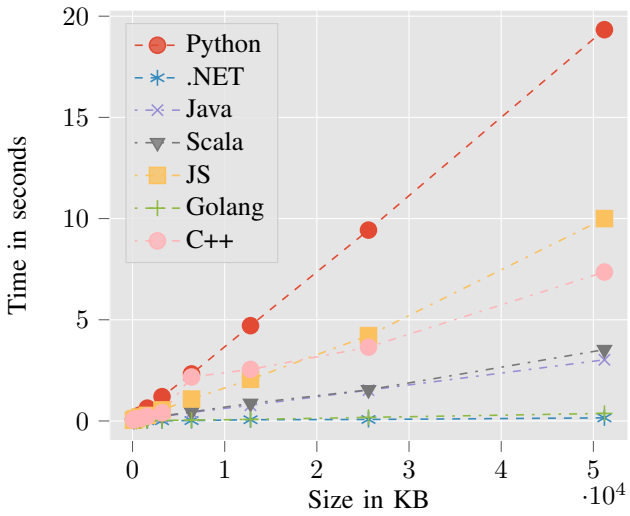


Fig. 5: Key exposure durations in memory.

the key for a significantly long duration. For several runs of the malware, we were able to identify the symmetric key in memory every time with no false positives or false negatives.

Upon conclusion of the file encryption routine, the symmetric key must be explicitly destroyed from memory using `CryptDestroyKey` to prevent key leakage. In cases where ransomware developers overlook this critical step, the symmetric key(s) linger in memory, facilitating easier key extraction. We performed extraction of the `AES-256` key in

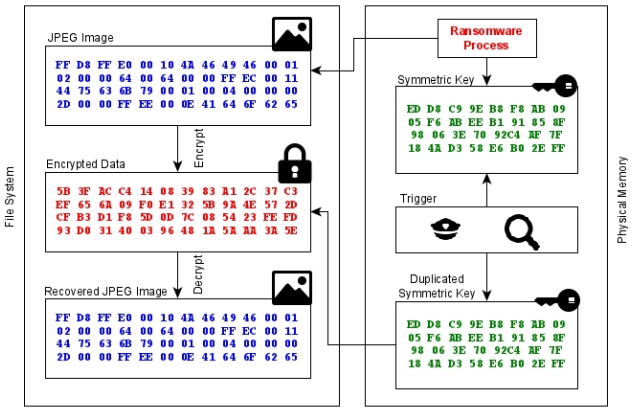


Fig. 6: Extracting key from memory to decrypt files

memory *during the encryption* using Algorithm 1 as shown in Figure 6. Finally, we tested the validity of the raw hexadecimal key that was extracted from memory by decrypting a JPEG image (magic bytes: `FF D8 FF E0`) that was encrypted by the ransomware as shown in Figure 6.

The size of the memory space being searched varied according to the language that the ransomware binary was written in. We measured an average search speed of 0.5 MB/s. This implies that keys are discovered in memory within seconds for ransomware written in, for instance, C or C++ since the process memory size for ransomware written in these languages averaged at 15 MB. With most ransomware written in such compiled languages, this is a promising result.

For ransomware that do not explicitly call `CryptDestroyKey` or similar function within the file encryption loop, we are able to dump multiple keys for a multi-key ransomware with one scan instance. This is because a memory scan conducted during the later stages of encryption now discovers keys that were used to encrypt previous files in addition to the key that is presently being used to encrypt a file. Ransomware are known to make such cryptographic implementation errors [23] and to be oblivious to basic key hygiene. Proper key hygiene in this case demands that a symmetric session key be wiped from memory immediately after it fulfills its purpose. Even in the absence of cryptographic flaws, a perfectly implemented Category 6 ransomware will reveal the key to decrypt all files in the case of a single-key ransomware and at least a subset of the keys in the case of a multi-key ransomware.

In the case of `WannaCry`, we were able to discover the victim-specific private key, P_s, in memory by sampling its process memory at closely-spaced intervals while running the malware in a debugger. The `PRIVATEKEYBLOB` structure, as discussed in Section IV-B, was then located and used for decrypting the set S_k of `AES-256` keys. We also observed `WannaDecryptor` – `WannaCry`'s decryption component – decrypted all files successfully when the 1172 bytes of P_s was made available on the host in the `PRIVATEKEYBLOB` format with the expected file name of `00000000.dky`. However, further testing is needed to determine the accuracy and consistency of key extraction from multi-key ransomware and we plan to expand the sample set (Table III) to include more variants in the future. Table III shows the ransomware variants for which we were able to locate the encryption key in memory and decrypt files encrypted by the ransomware using the extracted key.

VII. Conclusion and Future Work

We realize that using memory forensics to discover ephemeral keys is a convoluted response to ransomware when compared with other approaches. Therefore, this approach is not meant to substitute preventative measures against ransomware since prevention is preferred over cure. These memory forensics strategies are presented as *defense-in-depth* against ransomware's key management *after* the ransomware infection is already on the host and is successfully executing. When all else has failed (including backups), few recovery options exist to regain files without paying the ransom. In this paper, we explore another potential solution in the *layered* defense against ransomware. Antivirus solutions take the same defense-in-depth stance. Since the adversary performs the encryption on an untrusted host (victim's system), we can implement an attack on the ransomware's key management.

Conventional implementations of encryption routines are highly insecure when a hostile entity controls the execution environment. Cryptography is meant to protect the confidentiality of data *after* encryption. It is assumed that data will be encrypted on a trusted host and hence key exposure *during* the encryption process is not considered a weakness. For ransomware, however, this assumption becomes a vulnerability since key(s) are exposed on the victim's machine, and this machine is a *whitebox* to the victim. Note the whitebox cryptography [24] allows obfuscation of keys in memory during the process of encryption. Moreover, using techniques such as TRESOR [25], it is possible to beat this methodology by storing keys in CPU registers instead of RAM. However, such implementations are complex and can be assumed to be beyond the skillset of most ransomware developers [23] [1].

For our future work, we wish to test the effectiveness of the proposed methodology against a larger set of multi-key ransomware strains, thus evaluating the presence of any residual edge cases where the proposed methodology fails to deliver the required decryption keys. Additionally, we wish to perform more focused experiments with improving the accuracy of the trigger condition (e.g. heuristics-based ransomware detection techniques) such that the false positives are minimized, limiting the required memory dumps or scans. Note that our approach is tolerant of false positives since these scans and dumps happen transparently and are not disruptive. However, minimization of false positives is still desired to control any unnecessary strain on system resources such as disk space and CPU load.

We also wish to examine ways of mapping the extracted keys to encrypted files for the multi-key ransomware. Currently, we are using a "bruteforce" methodology to attempt

TABLE III: Memory-based key extraction against ransomware

Name	Algorithm	MD5 hash signature	Decryption %
Lockcrypt2.0	$AES-256+RSA$	3CF87E475A67977AB96DFF95230F8146	100
eCh0raix	$AES-256+RSA$	DA34C9A18D9693ACCC477B12695BCF37	100
CryptoRoger	$AES-256$	6B98FD062FBF0984DD3589EDB092FA80	100
WannaCry	$AES-256+RSA$	DB349B97C37D22F5EA1D1841E3C89EB4	100
AdamLocker	$AES-256$	D4452ADFC41A7075F5E5796172775898	100
Alphabet	$AES-256+RSA$	DBE78231174B03239EB262CC2D2D0900	100
Alphalocker	$AES-256+RSA$	C8EF7849A40DBC220B6B3CB5C9FAE496	100
CryptoRansomware	$AES-256$	84C44DF77EFB8A55ABD217A379C2589A	100
BlackRuby	$AES-256+RSA$	4958DDE3003BD4A89A6E82DC9ABD16CB	100

decryption of a file with every extracted key in the database until a match is found. This approach does not scale well for environments with thousands of files and could be made more efficient if we stored mappings of keys to files based on the `Windows-1252` sequential file enumeration that most ransomware deploy. Moreover, we wish to test key extraction against ransomware deploying other encryption algorithms (besides AES and RSA shown in this paper).

There are several advantages of using memory-based key extraction against ransomware. This approach is ransomware-language-independent, that is, it does not depend on the language that was used to write the ransomware. Moreover, this approach is platform-independent and can be easily scaled to be applied to other operating systems such as MacOS or Linux. Furthermore, such key extraction works even against ransomware that do not use the host's CryptoAPI for key generation since all implementations will expose keys in memory.

In conclusion, memory attacks against modern ransomware show potential for practical deployment and can be used for file recovery following a ransomware infection. However, it must be noted that such key extraction will serve best as an *additional* layer in a defense-in-depth solution against ransomware.

REFERENCES

[1] P. Bajpai, A. K. Sood, and R. Enbody, "A key-management-based taxonomy for ransomware," in *2018 APWG Symposium on Electronic Crime Research (eCrime)*. IEEE, 2018, pp. 1–12.

[2] P. Bajpai, "Extracting ransomware's keys by utilizing memory forensics," Ph.D. dissertation, Michigan State University, 2020.

[3] P. Bajpai and R. Enbody, "Attacking key management in ransomware," *IT Professional*, vol. 22, no. 2, pp. 21–27, 2020.

[4] A. Shamir and N. Van Someren, "Playing 'hide and seek' with stored keys," in *International conference on financial cryptography*. Springer, 1999, pp. 118–124.

[5] T. Pettersson, "Cryptographic key recovery from linux memory dumps," *Chaos Communication Camp*, vol. 2007, 2007.

[6] B. Kaplan *et al.*, "Ram is key extracting disk encryption keys from volatile memory," 2007.

[7] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[8] A. Kharraz and E. Kirda, "Redemption: Real-time protection against ransomware at end-hosts," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 98–119.

[9] J. Gómez-Hernández, L. Álvarez-González, and P. García-Teodoro, "R-locker: Thwarting ransomware action through a honeyfile-based approach," *Computers & Security*, vol. 73, pp. 389–398, 2018.

[10] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 303–312.

[11] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 336–347.

[12] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "{UNVEIL}: A large-scale, automated approach to detecting ransomware," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 757–772.

[13] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 599–611.

[14] O. Billet, H. Gilbert, and C. Ech-Chatbi, "Cryptanalysis of a white box aes implementation," in *International Workshop on Selected Areas in Cryptography*. Springer, 2004, pp. 227–240.

[15] C. E. Shannon, "Communication theory of secrecy systems," *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[16] W. E. Burr, "Selecting the advanced encryption standard," *IEEE Security & Privacy*, vol. 99, no. 2, pp. 43–52, 2003.

[17] D. Plohmann, M. Clauß, S. Enders, and E. Padilla, "Malpedia: a collaborative effort to inventorize the malware landscape," *Proceedings of the Botconf*, 2017.

[18] [Online]. Available: https://www.hybrid-analysis.com/

[19] [Online]. Available: https://app.any.run/submissions/

[20] N.-F. Standard, "Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, no. 1-51, pp. 3–3, 2001.

[21] R. A. E. B. L. Knudsen, "Serpent: A proposal for the advanced encryption standard," in *First Advanced Encryption Standard (AES) Conference, Ventura, CA*, 1998.

[22] C. Maartmann-Moe, S. E. Thorkildsen, and A. Årnes, "The persistence of memory: Forensic identification and extraction of cryptographic keys," *digital investigation*, vol. 6, pp. S132–S140, 2009.

[23] B. Herzog and Y. Balmas, "Great crypto failures," 2016.

[24] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "White-box cryptography and an aes implementation," in *International Workshop on Selected Areas in Cryptography*. Springer, 2002, pp. 250–270.

[25] T. Müller, F. C. Freiling, and A. Dewald, "Tresor runs encryption securely outside ram." in *USENIX Security Symposium*, vol. 17, 2011.