Exploitation of the Vulnerabilities of Hive Ransomware for Finding the Private Key

Nunzio Amato, Riccardo Di Pietro and Stefano Zanero

May 12, 2023

Exploitation of the vulnerabilities of Hive ransomware for finding the private key [*]

Amato Nunzio[†], Di Pietro Riccardo[2‡], and Zanero Stefano[2§]

Deloitte Risk Advisory S.r.l S.B

Abstract

The spread of ransomware has become one of the major sources of cyber risk in recent years. Once installed on a machine, this type of malware encrypts victim's files and demands a ransom for the decryption key needed to regain access to the locked assets. The cost required for data recovery is very high and many companies do not have the funds to pay it. In this paper, we analyze the Hive Ransomware (version v5, v5.1, v5.2) and study its vulnerabilities during the generation of the private key used for encrypting the master key. By using these weaknesses, we provide a tool for all companies infected with this type of malware so that they are able to recover their data without the need to pay the ransom.

1 Introduction

Ransomware is a type of malicious software that encrypts victim's files and demands payment in exchange for the decryption key. It has been a threat for decades, but has become more prevalent and sophisticated in recent years. One of the most significant developments in the evolution of ransomware has been the use of cryptocurrencies, which makes it harder for law enforcement to track payments. In 2019, a criminal organization called TA2102 used the Maze ransomware to carry out the first high-profile *double extortion* attack [9], in which they not only encrypted the victim's data, but also exfiltrated it and threatened to publish it unless a ransom was paid. Since then, *double extortion* attacks have become more common and complex. In addition, criminals can now buy *ransomware as a service* (RaaS) [10] on the dark web and employ sophisticated strategies to maximize their profits. A type of ransomware that is known for its use of *double extortion* tactics, as well as its distribution through a *RaaS* model, is Hive. It is an affiliate-based ransomware family that first appeared in June 2021 and has been known to attack a diverse range of industries, with a particular focus on healthcare and public health organizations, as well as government facilities, communications companies, critical manufacturing industries, and information technology companies. Hive ransomware actors have attacked over 1,300 companies worldwide and received around $100 million in ransom payments [3]. The goal of this work is the same of Kim et al. [4], i.e. to provide all victimized companies a method to find the private key used during the encryption by the Hive Ransomware. This process can be applied to versions v5, v5.1, v5.2 of the malware. In summary, we provide the following contributions:

- We identify a vulnerability during the generation phase of the private key used for encrypting the master key

[*]

[†]Designed and implemented the work
[‡]Did numerous tests and provided a lot of suggestions
[§]Provided a lot of suggestions

- We exploit this weakness to reduce the cardinality of the key space needed to find the private key
- We define a process for finding the private key that explores the possible combinations in an intelligent way

2 Hive ransomware analysis

This section presents an analysis of several samples of the Hive Ransomware. The hash values of these executables are listed in Table 1. They are designed to be used on a Windows operating system, and we specifically analyzed versions v5, v5.1, and v5.2. The main differences between these versions and earlier ones include the change in the base programming language (from Go to Rust) and the improvement of the encryption algorithm. We conducted both a static and dynamic analysis of these samples using Ghidra and x64dbg, respectively.

SHA-256	version	architecture
4b62c93fbf0b964c4de93a0ce456bccdaee2908b3c0135b3f62912068a728d3e	5	32-bit
a464ae4b0a75d8673cc95ea93c56f0ee11120f71726cc891f9c7e8d4bec53625	5.1	32-bit
f4a39820dbff47fa1b68f83f575bc98ed33858b02341c5c0464a49be4e6c76d3	5	64-bit
f5d1acc98d62b3a3bfc640bfafd3144fa66112470512e26197cc9b643e438a0e	5.2	64-bit

Table 1: Hash value of Hive Ransomware sample files

2.1 Encryption process

Since Microsoft [7] has already conducted a comprehensive analysis of the malware, we only summarize the key steps carried out. Then we explain the vulnerabilities detected.

Master key generation The malware allocates a buffer of size 0xCFFF00 bytes and uses an algorithm to generate random bytes (we will discover in Section 3 that the bytes are not truly random) and fill the buffer. The first 0xA00000 bytes of the buffer are filled by the algorithm and they are then reused for the remaining 0x2FFF00 bytes. This process is repeated twice, resulting in the creation of two master keys used by the malware to encrypt the files.

Creation of threads The malware creates 1256 threads that are used to speed up the encryption process.

Terminating specific processes and services The Hive ransomware shuts down certain processes, like *excel*, *sql*, *wordpad*, and services, such as *windefend*, *oracle*, *backup*. This is done to prevent processes from interfering with the encryption phase.

Encryption of the master key Hive encrypts the two master keys in order to protect them from being discovered or accessed by unauthorized parties. In particular, it uses two rounds of encryption for each key. The steps in each round are the following:

1. A 32-byte private key is generated with the same algorithm used for the generation of the master key.

2. Using the Curve25519 elliptic curve algorithm for the Diffie-Hellman key exchange (ECDH) [2], the victim public key is derived from the private key and the Basepoint (it is 9 followed by 31 zeros).

3. A 24-byte nonce is generated using the same algorithm used for the private key and the master key.

4. A shared key is generated by using ECDH with Curve25519. The inputs are the private key generated in the step 1 and the Hive public key (embedded in the sample).

5. The key used for the encryption of the master key is generated by using HChaCha20 [1]. The inputs are the HCHACHA nonce (it is composed of 16 bytes, all of which are equal to zero) and the shared key.

6. The master key is encrypted using Poly1305-XChaCha20 [8]. The inputs are the master key and the nonce generated in step 3. In this step a 16-byte Message Authentication Code (MAC) is created for ensuring the integrity of the encryption process.

The first round involves encrypting the master key. The second round involves encrypting the result of the first round. Figure 9 describes the structure of the key at the end of the first and the second rounds, respectively.

Storage of the encrypted master key The outcome of the second round is stored in the root of the drive that is being encrypted, with a .key extension.

Encryption of the disk Once the malware has written both master keys to the disk, it begins the multi-threaded file encryption process. For each file, it randomly selects a byte sequence from one of the master keys and uses it to encrypt the file by XORing the byte sequence with the file's content. Specifically, the malware selects two distinct offsets that are used in sequence to XORs the file's content.

3 Vulnerability in Hive's algorithm

The analysis in Section 2 is used to understand the context in which we find ourselves. Within this context, we discovered a vulnerability. It arises from the algorithm with which the nonce, the private key, and the master key are generated. Therefore, our purpose is not to question the cryptographic power of Curve25519 and HChaCha20. Instead, we want to emphasize the weakness of the algorithm used to generate a sequence of bytes given as input to these robust cryptographic algorithms.

3.1 PRNG algorithm

The ransomware uses the same pseudo-random number generator (PRNG) algorithm to generate the nonce, the private key and the master key. The PRNG employs two Windows APIs to generate a stream of bytes: QueryPerformanceCounter (QPC) [5] and QueryPerformanceFrequency (QPF) [6]. The QPC method retrieves the value of the high-resolution performance counter that measures the time elapsed since the system was started, while the QPF method returns the frequency of the performance counter, fixed at system boot.

The PRNG uses the time difference between the generation of the seed (QPC_0) and the current system time value (QPC_i) to compute the elapsed time, and subsequently, generate

Algorithm 1 PRNG algorithm

```
Input: length, $QPC_0$, QPF
Output: key

// Initialization keys
$key[length] \leftarrow 0$

for $i \leftarrow 0$ to $length$ do

    $elapsed\_time \leftarrow QPC_i - QPC_0$
    $byte_i \leftarrow elapsed\_time \bmod QPF$
    $key[i] \leftarrow byte_i$

end for


return $key$
```

each byte. This computation involves dividing the time difference by the frequency of the performance counter to obtain the elapsed time in seconds. Additionally, the remainder of this division operation is computed and utilized in the generation of the pseudo-random bytes. Algorithm 1 provides a representation of the byte generation process. Listing 1 describes the PRNG algorithm implemented in C++.

3.2 Vulnerability detection

The algorithm's utilization of a temporal component for byte generation may appear to produce unpredictable bytes at first glance. However, upon closer examination, it has been found that if the value of QPF is set to 0x989680 (10 MHz), that is a common value for the performance counter frequency on many systems, the number of possible generated bytes is limited to 64. As a result, a brute-force attack on the 32-byte private key would require searching through 2^{192} combinations instead of 2^{256}, a significant reduction in the search space. Although this number is still far too large to be practical, it can serve as a starting point for implementing a method that further narrows down the range of possible keys (Section 4).

Table 2 shows the periodic sequence of bytes that is obtained when the difference between successive QPC values ($\Delta(QPC_i - QPC_{i-1})$) is equal to 1 and the QPF is equal to 10 MHz. This sequence highlights the algorithm's predictability, as it generates a fixed sequence of bytes that repeats every 64 iterations. This predictability is a significant vulnerability that can be exploited to recover the private key.

3.3 Relationship between QueryPerformanceCounter and the PRNG

According to the Listing 1, a seed s is created using the value of QueryPerformanceCounter, QPC_0. This seed is then used to generate a sequence of n bytes. By analyzing this sequence and the periodic sequence in Table 2, we discover a relationship between QPC_i and QPC_0 (see Figure 1). The difference between QPC_i and QPC_0 determines the position in the periodic sequence (Table 2) of the byte that is generated at the i-th iteration. Therefore, if we know

00	64	C8	2C	90	F4	58	BC
20	84	E8	4C	B0	14	78	DC
40	A4	08	6C	D0	34	98	FC
60	C4	28	8C	F0	54	B8	1C
80	E4	48	AC	10	74	D8	3C
A0	04	68	CC	30	94	F8	5C
C0	24	88	EC	50	B4	18	7C
E0	44	A8	0C	70	D4	38	9C

Table 2: Periodic sequence of bytes

the $\Delta(QPC_i - QPC_0)$ $\forall i \in n$, we could reconstruct the entire sequence of generated bytes. Moreover, we can generate the bytes even without the knowledge of QPC_0. We just need the knowledge of $\Delta(QPC_i - QPC_{i-1})$ $\forall i \in n$ and, in at most 64 attempts, we can reconstruct the private key.

$$\begin{cases} \quad \forall i \in n, \\ \text{if } \Delta(QPC_i - QPC_0) \equiv 0 \pmod{64}, \\ \qquad \text{then } byte_i = 0x00 \\ \text{if } \Delta(QPC_i - QPC_0) \equiv 1 \pmod{64}, \\ \qquad \text{then } byte_i = 0x64 \\ \text{if } \Delta(QPC_i - QPC_0) \equiv 2 \pmod{64}, \\ \qquad \text{then } byte_i = 0xC8 \\ \vdots \\ \text{if } \Delta(QPC_i - QPC_0) \equiv 64 \pmod{64}, \\ \qquad \text{then } byte_i = 0x9C \end{cases} \tag{1}$$

3.4 Anomalies during the generation of bytes

When the value of the register that manages the QPC is increased by 1 at each iteration of the algorithm, the private key retrieval would require at most 64 combinations to be tried. However, since the frequency at which the CPU executes instructions is different with respect to the frequency of the high-resolution performance counter, it is not so trivial to recover the key necessary to decrypt the master key. A possible scenario is presented in Figure 1.

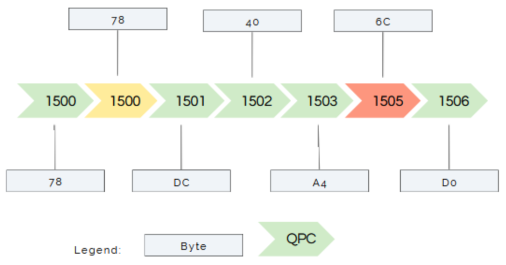


Figure 1: Anomalies in the generation of bytes

Figure 1 shows possible values that can be retrieved by invoking QueryPerformanceCounter and the bytes generated by using those results. As we expected, if the CPU is able to compute multiple iterations of the algorithm before the value of the high-resolution performance counter is updated, we have the same QPC and thus the same byte during consecutive iterations (see yellow arrow). On the other hand, if the high-resolution counter is updated before the method QueryPerformanceCounter is invoked, the next QPC will be different to the previous one. In case $\Delta(QPC_i - QPC_{i-1}) = 1$ the $byte_i$ will be the one that follows the $byte_{i-1}$ in the periodic sequence (see green arrow). Otherwise, if $\Delta(QPC_i - QPC_{i-1}) > 1$ we have a hop in the periodic sequence (see red arrow). In fact, in correspondence to QPC = 1505, the byte generated is not `0x8` but `0x6C`. In order to describe these anomalies and to define a method for reducing the key set needed for retrieving the private key, we modelled a set of features.

3.5 Modelling of features

These anomalies allow us to define some features that describe the behaviour of the CPU during the generation of the bytes. In particular, these features are used for describing the 24-byte nonce and the 32-byte private key. Consider the following nonce in Table 3:

2C	F4	58	BC	20	84	84	E8	4C	78	DC	40
A4	08	6C	D0	98	FC	60	C4	28	8C	F0	F0

Table 3: Example of 24-byte nonce

The red cell represents the position in which we have a *hop*, i.e. where $\Delta(QPC_i - QPC_{i-1}) > 1$. In this case *the length of hop* represents the value of $\Delta(QPC_i - QPC_{i-1})$. The green cell represents the position in which there is a byte equal to the previous one (*consecutive bytes*), i.e where $\Delta(QPC_i - QPC_{i-1}) = 0$. The features are: *number of hops*, *position of hops*, *difference between position of hops*, *length of hops*, *number of consecutive bytes*, *position of consecutive bytes*, *difference between position of consecutive bytes*. The features of the nonce in Table 3 are:

- number of hops = 3
- position of hops = $[1, 9, 16]$
- difference between position of hops = $[8, 7]$
- length of hops = $[2, 3, 2]$
- number of consecutive bytes = 2
- position of consecutive bytes = $[6, 23]$
- difference between position of consecutive bytes = $[17]$

3.6 Preliminary works

These features are used to perform a smart research for finding the private key. To describe this process, it is important to note that when the master key is encrypted, the only information in plaintext are the 24-byte nonce and the victim's public key. Moreover, there are few instructions (see Figure 10) between the generation of the private key and the nonce. Assuming that the private key and the nonce are generated with a similar CPU frequency, the knowledge of the nonce and of its features can be a means to perform an optimized research of the private key. To understand the relationship between the nonce and the private key, we patched two of the analyzed samples (the 32-bit sample v5 and the 64-bit sample v5, see Table 1). We created precisely two patches for each of them. The first patch gives the possibility to retrieve the couple nonce-private key and to write it into a file. The second patch (that is used in Section 4.1) allows us to write the master key into a file. We also disabled the harmful aspect of the executable so that it could be used only for generating keys. The purpose of patching two samples of Hive is to study the characteristics of the bytes generated by the PRNG by running a 32-bit and a 64-bit executable respectively. All experiments were performed on two machines, one equipped with an Intel Core i5-4590 processor that runs at a clock speed of 3.30 GHz, and the other with an Intel Core i7-8665U processor that runs at a speed of 1.90 GHz (Table 4). Both machines had Windows 10 installed.

Version	Release	RAM	CPU	QPF
Win10PRO	22h2	16GB	Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz	10000000
Win10HOME	22h2	8GB	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz	10000000

Table 4: Tested Laptops

3.7 Relationship between the 24-bytes nonce and 32-bytes private key

After modifying the malware, we run the first patch many times to retrieve data useful for establishing the correlation of the features between the nonce and the private key. We perform parallel benchmark experiments (Atto, Cinebench, Novabench) while executing the patched malware to examine the impact of system load on the features extracted from private key-nonce pairs. Specifically, we stressed the RAM, disk, and graphics card, and sought to determine whether the features differed when the malware was executed with and without the benchmark. Our results showed that there was no discernible variation in the features, indicating that the benchmark had no significant effect on the malware's behaviour. Moreover, we discover empirically that the features are influenced mainly by the frequency of the CPU. Thus we vary the values of the CPU frequency during the collection of the data to figure out how the features evolve with respect to the variation of the frequency. More precisely, for each frequency, we run the patched malware 5000 times and we gather the features for each nonce-private key pair (see Figure 11). Then, we analyze what are the major occurrences of *number of hops nonce*, *number of consecutive bytes nonce*, *number of hops pk*, *number of consecutive bytes pk* for each frequency.

Figures 2 and 3 show the trend of the features as the frequency increases. As we expected, the *number of hops* and the *number of consecutive bytes* of the nonce are less than that of the private key, since the former consists of 24 bytes and the latter of 32 bytes. We can also notice that the variation of the features retrieved from the patched 32-bit malware is different from the 64-bit malware. In both cases, as the frequency of the CPU increases, the *number of hops* feature decreases. However, for the 32-bit malware, when a frequency of 3.80 GHz is reached the *number of hops* feature vanishes whereas the *consecutive bytes* feature appears. Instead, on the 64-bit sample, the *number of hops* feature disappears at around 2.38 GHz and the *consecutive bytes* feature appears at around 2.18 GHz. Moreover, for the 32-bit malware,

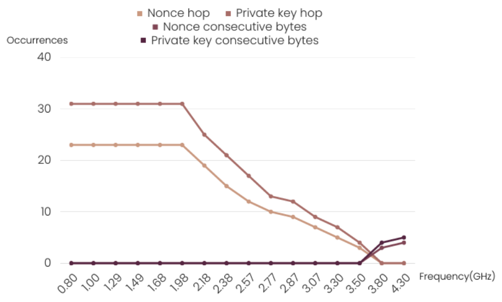


Figure 2: features 32-bit

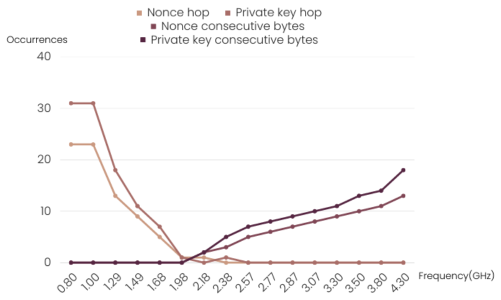


Figure 3: features 64-bit

from 0.8 GHz to 1.98 GHz the *number of hops* are 23 for the nonce and 31 for the private key. For the 64-bit malware, the previous features are found only at 0.8 GHz and at 1 GHz. The feature that changes when the *number of hops* of the nonce is equal to 23 and the *number of hops* of the private key is equal to 31 is the *length of hops*. Tables 5 and 6 show the different *length of hops* and the average values. Regarding the 32-bit sample, the *length of hops* that characterizes the frequency 0.8 GHz is 5 (in average, 28.38 for the private key and 20.98 for the nonce). As the frequency increases, the *length of hops* decreases until it reaches 2 at 1.98 GHz. From that value on, the *length of hops* mainly takes on the value of 2. Regarding the 64-bit sample, the *length of hops* starts to assume the value of 2 from 1 GHz (in average 29.28 for the private key and 21.42 for the nonce). The analysis also highlights that the features of the nonce are only slightly different from those of the private key. Therefore, the nonce can be useful for detecting the possible features of the private key. Thanks to the collected data, we defined a range of features that the private key can take based on the features assumed by the nonce (Figure 12). The knowledge of the nonce thus allows us to narrow down our area of interest during the search for the private key.

	length of hops private key			
Freq(GHz)	2	3	4	5
0.8	0.0002	0.0002	1.41	28.38
1	0.0002	1.11	28.72	0.72
1.29	0.018	28.39	2.33	0.23
1.49	10.58	19.93	0.46	0.009
1.69	19.97	10.78	0.20	0.01
1.98	30.41	0.508	0.04	0.01

	length of hops of nonce			
Freq(GHz)	2	3	4	5
0.8	0.0001	0.0002	1.05	20.98
1	0.0001	0.83	21.29	0.69
1.29	0.019	20.88	1.79	0.22
1.49	7.79	14.88	0.31	0.005
1.68	14.75	8.11	0.11	0.006
1.98	22.74	0.24	0.002	0.002

Table 5: length of hops nonce and private key (32-bit)

	length of hops private key			
Freq(GHz)	2	3	4	5
0.8	14.15	15.89	0.52	0.36
1	29.28	1.30	0.35	0.0004

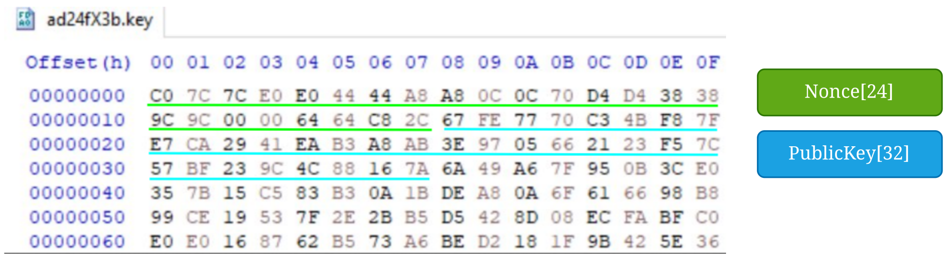
	length of hops nonce			
Freq(GHz)	2	3	4	5
0.8	10.32	11.76	0.62	0.17
1	21.42	1.03	0.30	0.10

Table 6: length of hops nonce and private key (64-bit)

4 Implementation of the search of the private key

After designing the features, our objective is to compute an intelligent search of the private key, trying to reduce the number of possible combinations. The idea is to:

- extract the nonce and the public key from the file .key (Figure 4);
- compute the features of the nonce and, in particular, take into account of the *number of hops* and the *number of consecutive bytes* of the nonce;
- use the two features previously retrieved to search inside the table in Figure 12 the possible range of features of the private key. After that, search inside the collected data all the entries that have *min number of hops pk* $\leq$ *number of hops pk* $\leq$ *max number of hops pk* and *min consecutive bytes pk* $\leq$ *consecutive bytes pk* $\leq$ *max consecutive bytes pk.*
- once the occurrences are retrieved, start the search of the private key (Figure 5).



```
ad24fX3b.key

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  C0 7C 7C E0 E0 44 44 A8 A8 0C 0C 70 D4 D4 38 38
00000010  9C 9C 00 00 64 64 C8 2C 67 FE 77 70 C3 4B F8 7F
00000020  E7 CA 29 41 EA B3 A8 AB 3E 97 05 66 21 23 F5 7C
00000030  57 BF 23 9C 4C 88 16 7A 6A 49 A6 7F 95 0B 3C E0
00000040  35 7B 15 C5 83 B3 0A 1B DE A8 0A 6F 61 66 98 B8
00000050  99 CE 19 53 7F 2E 2B B5 D5 42 8D 08 EC FA BF C0
00000060  E0 E0 16 87 62 B5 73 A6 BE D2 18 1F 9B 42 5E 36
```

Figure 4: Extraction of the nonce and the public key

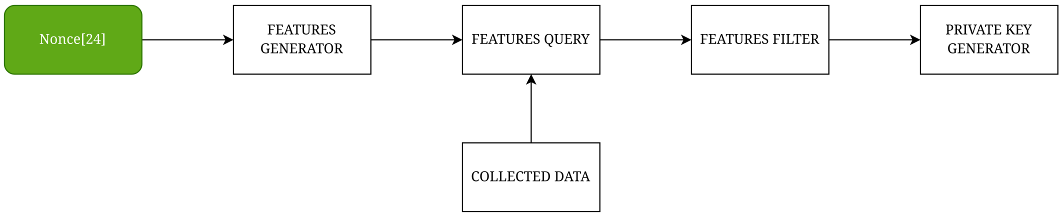


Figure 5: Steps for retrieving the useful private key features

Before generating the possible private keys, we first extract the *difference between position of consecutive bytes*, the *difference between position of hops* and *the length of hops* of the private key from the rows obtained by the previous query (Figure 11). Starting from the *difference between position of consecutive bytes pk* and the *difference between position of hops pk*, we compute all the possible positions in which the *hops* and *consecutive bytes* can be found. For example, in case the *difference between position of hops pk* $= [7, 7, 8, 7]$ and *difference between position of consecutive bytes pk* $= []$, the *position of hops pk* are $[1, 8, 15, 23, 30]$ and $[2, 9, 16, 24, 31]$ and there is no *position of consecutive bytes.* These positions are used in combination with the *length of hops* for generating possible strings of 32 bytes (remember that each byte can take only 64 values). Each generated key together with Basepoint 9 is given as input to Curve25519 and a public key is generated (Algorithm 2). The last key is compared with the public key present in the file .key. In the event that the two keys are equal, the private key for decrypting the file .key is found (Figure 6).

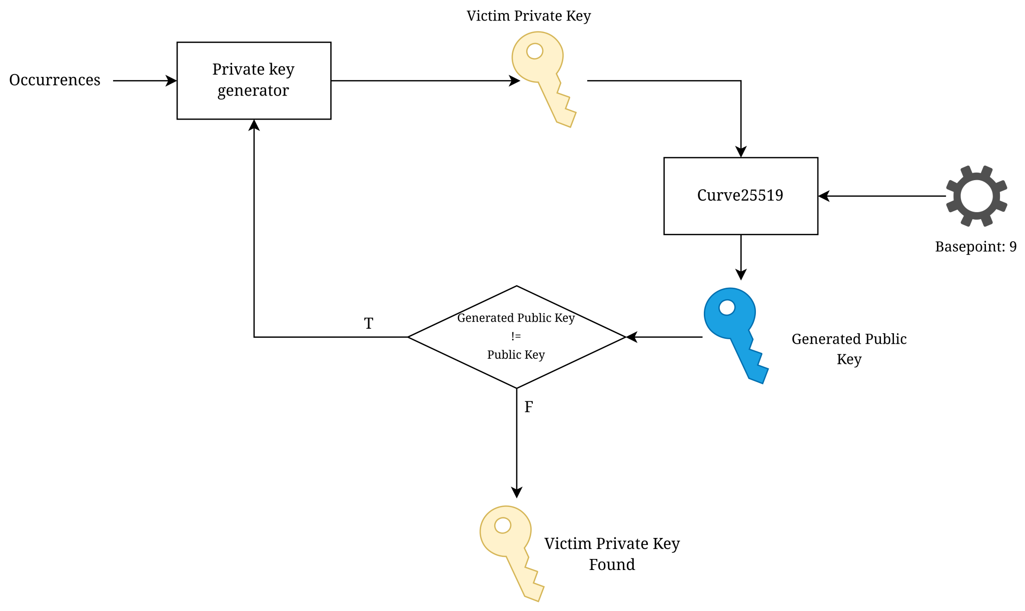


Figure 6: Private key search

4.1 Alternative way for the search of the private key

The problem with the previous method is that we are not able to explore all the possible combinations from the collected data since there can be some features needed for finding the private key that are not covered. In fact, given a *number of hops* feature and a *number of consecutive bytes* feature, we are not able to explore neither all the possible positions in which there are *hops* and *consecutive bytes* nor the possible *length of hops*. If we do not find the key from the collected data, we can take advantage of the analysis described in Figures 2 and 3. In fact, by looking at the features of the nonce, we can determine the frequency used by the CPU during nonce generation. Once the right frequency is found, we can run the second patch of the malware for generating a master key. Since the master key is created by the same private key and nonce algorithm, we can use the master key to search for the private key: scan the master key, take 32 bytes at a time, compute the features and generate private keys. We compute precisely the *difference of position between consecutive bytes*, the *difference of position between hops* and the *length of hops*. After that we generate strings of 32 bytes and we use the same procedure described in Section 4.

5 Results and discussion

The aforementioned methods were tested by executing four samples of Hive (Table 1) on different Windows 10 machines. In total, 100 malware executions were performed, resulting in the generation of 200 master keys, given that each sample generates two master keys. Additionally, as each master key is encrypted twice during two encryption rounds, there were 400 private keys to be recovered during the testing phase.

In particular, the method described in Section 4 was first employed, and if it produced negative results, the method in Section 4.1 was used. The value returned by QueryPerformanceFrequency on the tested machines is 10 MHz, as expected. We removed the malicious component from the executables and made them generate only the two master keys, which were then encrypted. During the execution of the malware, we intentionally did not set a fixed CPU frequency, as done during the data collection phase. This decision was made to test the plausibility of our assumptions made during the data collection phase. Specifically, we wanted to verify whether the private key and nonce were generated at a similar CPU frequency, and whether the nonce could be used as useful information for the recovery of the private key. By using the methods described in the previous sections, we were able to recover various private keys. Table 9 describes the number of private keys recovered from both the second encryption round and the first encryption round. From the second encryption round, 172 private keys were recovered. They were then used to decrypt the second encryption round and retrieve the content of the first encryption round. From the 172 content of the first encryption round, 161 private keys were recovered. In total, therefore, we were able to decrypt 161 master keys out of 200. Furthermore, we were able to recover 333 private keys out of the 400 available. Tables 7 and 8 report some of the files *.key of which we were able to find the private key.

Tables 10 and 11 show the number of recovered private keys and the number of attempts on average needed to find the private key according to the features of the nonce. The first table depicts the scenario where the nonce has a *number of hops* greater than zero and *consecutive bytes* equal to zero. The second table shows the case where the nonce has a *number of consecutive bytes* greater than zero and the *number of hops* equal to zero. Figures 7a and 8a describe the percentage of recovered private keys in the second encryption round and the first encryption round, respectively. From Table 10, we can observe that as the *number of hops* increases, the recovery of the private key becomes more challenging and the number of attempts increases (Figures 7b). For instance, when the nonce had a *number of hops* between 0 and 10 (second round of encryption), we managed to recover 95% of the private keys (52892 attempts on average), whereas when the nonce had a *number of hops* between 18 and 23, we could only recover 37% of the private keys (10389328 attempts on average). Table 11 shows that the difficulty of recovering the private key increases along with the number of attempts (Figure 8b), as the *number of consecutive bytes* in the nonce grows. Specifically, we were able to recover 84% of the private keys related to the second round of encryption when the nonce had between 0 and 5 *consecutive bytes* (167458 attempts on average), while we could recover 68% of the private keys when the nonce had between 11 and 15 *consecutive bytes* (2249004 attempts on average).

These results highlight the power of the methods described previously. We not only lowered the number of potential combinations needed to find the private key from 2^{256} to 2^{192}, but by utilizing the nonce features, we were able to decrease the attempts even further. This also confirms that the nonces and private keys were likely generated using a CPU with a similar frequency, and that by examining the properties of the nonce, we can limit the set of possible keys that we need to search through. The reason why we may sometimes fail to recover the private key is primarily due to a mismatch in the frequency at which the nonce was generated and that at which the private key was generated. Thus, in cases where there is a sudden variation in frequency between the nonce and the private key, it becomes challenging to recover the private key.

file key	hop nonce	hop pk	tested keys
zIGJJr0E.key	1	3	13694
8zploAPL.key	1	5	48979
66h8PpTM.key	2	2	1214
zIGnJr0E-firstRound.key	2	2	19114
-MQBxMJf.key	2	3	13820
OIoQvSNO-firstRound.key	3	3	31400
OIoQvSNO.key	3	4	30900
dvIWE2Df.key	3	5	123334
avP-W2aO.key	12	13	1086634
P4t9hF-D.key	23	31	10000890

Table 7: Examples of decrypted files *.key (case only hops in the nonce)

file key	consecutive bytes nonce	consecutive bytes pk	tested keys
6cPdfbPJ.key	3	3	128714
OrrJLrwE.key	4	5	136990
OrrJLrwE-firstRound.key	4	5	191814
lamMMMpo.key	5	5	482796
-pQMxNXf.key	5	6	133440
I9oMbRNO.key	6	6	1307654
I9oMbRNO-firstRound.key	7	9	1136400
PkwoA-Wq.key	10	12	1700890
LmaPO1-s.key	11	11	2673334
LmaPO1-s.key	13	14	2097394

Table 8: Examples of decrypted files *.key (case only consecutive bytes in the nonce)

round of encryption	found keys	total keys
2	172	200
1	161	172

Table 9: Number of recovered private keys

range hops	round of encryption	found keys	total keys	attempts
0-10	2	61	64	52892
0-10	1	45	49	68022
11-17	2	36	38	1132827
11-17	1	26	31	1172224
18-23	2	6	16	10389328
18-23	1	6	10	11384356

Table 10: Number of retrieved private keys and attempts given the range of nonce's features (case only hops in the nonce)

range consecutive bytes	round of encryption	found keys	total keys	attempts
0-5	2	40	44	167458
0-5	1	33	39	177468
6-10	2	18	22	1123483
6-10	1	26	31	1163443
11-15	2	11	16	2249004
11-15	1	6	10	2329643

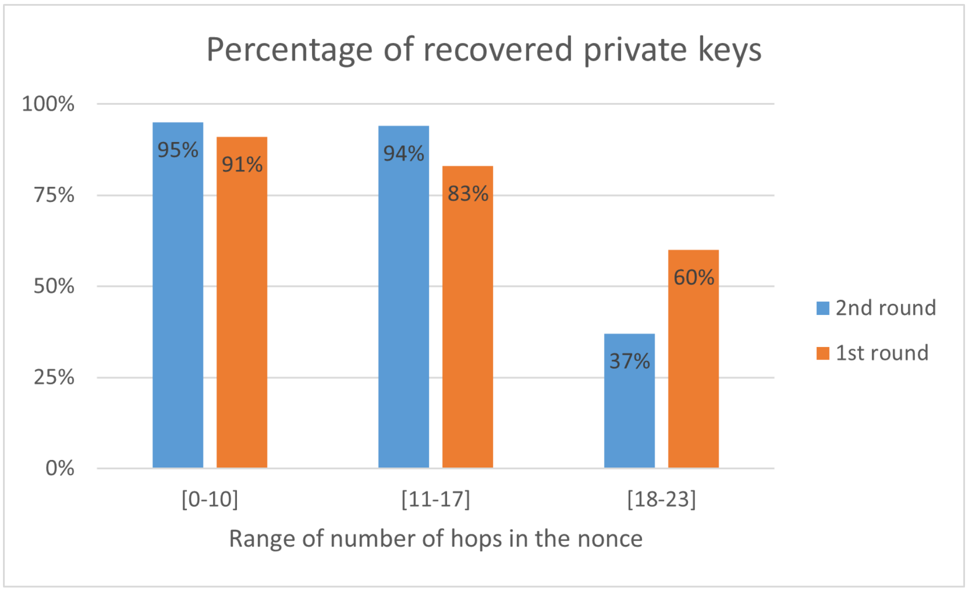
Table 11: Number of retrieved private keys and attempts given the range of nonce's features (case only consecutive bytes in the nonce)

6 Conclusions

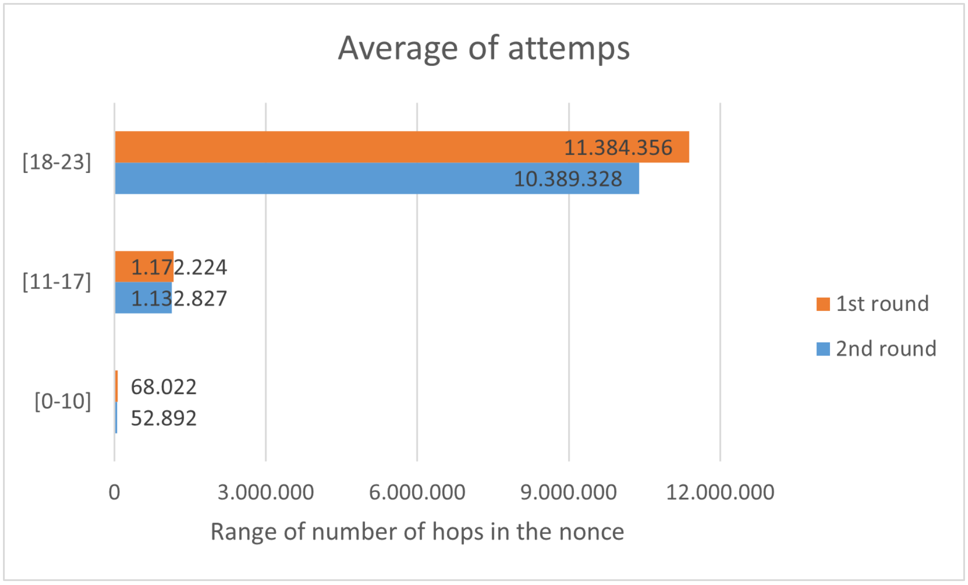
In summary, the aim of this study was to analyze versions v5, v5.1, and v5.2 of the Hive ransomware and exploit its vulnerabilities within it. Its weakness lies in the fact that the PRNG algorithm used to generate the master key, nonce, and private key generates a periodic sequence of 64 bytes. Our objective was to define a model to exploit these vulnerabilities by creating features to synthesize the bytes present in the nonce, private key, and master key. We hypothesized that since there are few assembly instructions between the generation of the nonce and private key, they are generated at a similar frequency to the CPU. This led us to find a relationship between the features of the nonce and the private key, which we recovered after patching two Hive samples. We collected the data by executing the patched malwares 5000 times at each frequency we were able to extract from the machines used in the experiment. After collecting the data, we created features for each private key-nonce pair and compared the trends of the nonce and private key features. We found that by starting with the information from the nonce, it is possible to recover useful information for private key recovery. We then implemented a method to search for private keys based on the nonce features and tested it by running the malware in different Windows 10 machines. Our results confirm that the nonce can be used useful for private key recovery. These results have important implications for companies that have been infected with Hive ransomware. Affected companies can take the nonce in the *.key file and compute its features. Then, they can leverage the information in Figure 12 to understand possible private key features and implement a search for it. However, this study only covers versions v5, v5.1, and v5.2. From version v5.3, the function BCryptGenRandom, instead of QueryPerformaceCounter, is used to generate random bytes. Further research could study how BCryptGenRandom is used and determine if a vulnerability is present there, as well. In conclusion, these results show that even criminal groups make mistakes during the writing of their malicious code. By reversing and analyzing the malware, weaknesses in the code can be identified and exploited. Therefore, it is important for companies to invest in skilled reverse engineering and malware analysis professionals. This can help them avoid having to pay a ransom in the event of an attack.

7 Acknowledgments

This work has been supported by Deloitte Risk Advisory S.r.l S.B.

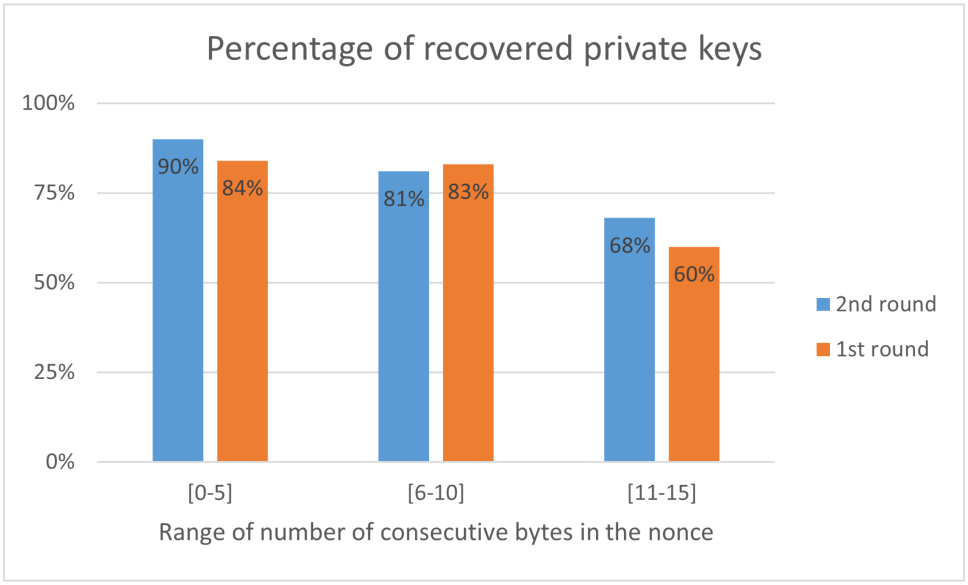


(a) Percentage of recovered private keys (case only hops in the nonce)

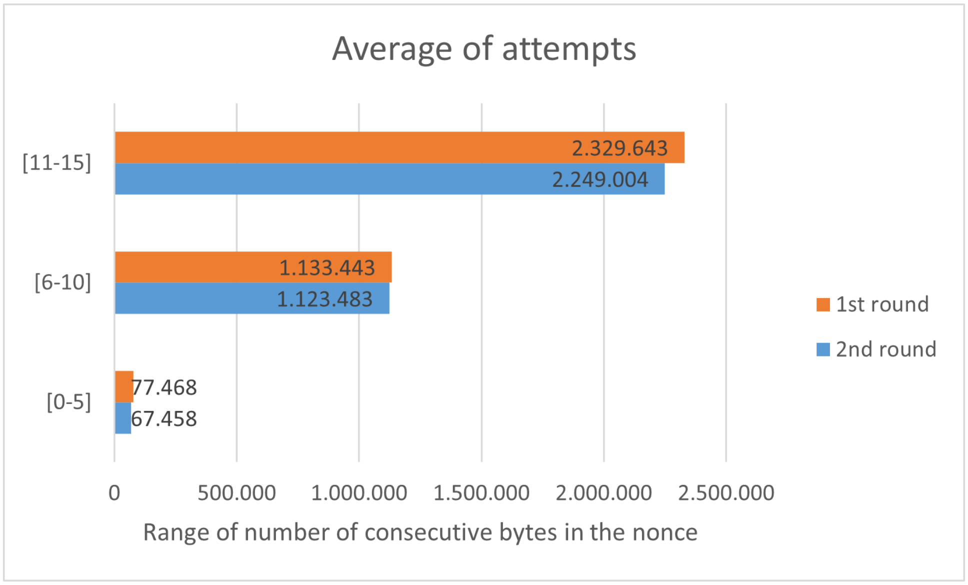


(b) Number of attempts for retrieving the private keys (case only hops in the nonce)

Figure 7



(a) Percentage of recovered private keys (case only consecutive bytes in the nonce)



(b) Number of attempts for retrieving the private keys (case only consecutive bytes in the nonce)

Figure 8

References

[1] Daniel Bernstein. Chacha, a variant of salsa20. 01 2008.

[2] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] CISA. #stopransomware: Hive ransomware. https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-321a.

[4] Giyoon Kim, Soram Kim, Soojin Kang, and Jongsung Kim. A method for decrypting data infected with hive ransomware, 2022.

[5] Microsoft. Queryperformancecounter function (profileapi.h). https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter.

[6] Microsoft. Queryperformancefrequency function (profileapi.h). https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancefrequency.

[7] Microsoft. Hive ransomware gets upgrades in rust. https://www.microsoft.com/en-us/security/blog/2022/07/05/hive-ransomware-gets-upgrades-in-rust/, 2006.

[8] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[9] Check Point Research. Ransomware evolved: Double extortion. https://research.checkpoint.com/2020/ransomware-evolved-double-extortion/.

[10] Windows. Ransomware as a service: Understanding the cybercrime gig economy and how to protect yourself. https://www.microsoft.com/en-us/security/blog/2022/05/09/ransomware-as-a-service-understanding-the-cybercrime-gig-economy-and-how-to-protect-yourself/.

A Appendix

In this additional section, we show several figures and the algorithm used to search for the private key that we believe can support the comprehension of the work.

Algorithm of the private key generator Algorithm 2 defines the way the private keys are generated starting from the *length of hops pk*, *difference between hops pk*, *difference between consecutive bytes pk*. It precisely describes the scenario in which we have no *consecutive bytes* but only *hops*.

Tested laptops As part of our investigation into the predictability of the PRNG, we also tested the laptops that were available to us to determine which ones had a QPF value of 10 MHz. In most of the tested machines, we found a QPF value of 10 MHz. However, we observed that there can be additional values based on the operating system and architecture of the machine (Table 12).

Phase of encryption of the master key Figure 9 shows the two rounds of encryption of the master key. In the first round the master key is encrypted and put inside a structure that contains the nonce, the public key and the MAC generated during that round. In the second round this structure is further encrypted and then stored in a file .key.

Generation of the private key and the nonce Figure 10 emphasizes the small code distance between the generation of the private key and the nonce. This allows us to make the assumption that they are both generated with a similar CPU frequency.

Example of the features of the nonce-private key pair Figure 11 shows the features of the nonce-private key couple retrieved by executing the patched 32-bit malware at a frequency of 3.5 GHz.

Possible range of features Figure 12 shows what is the possible range of features of the private key given the knowledge of the nonce features. Starting from the feature nonce, we can know what are the *min number of hops pk*, *max number of hops pk*, *min number of consecutive bytes pk* and *max number of consecutive byte pk*. This knowledge allows us to restrict the key set in which we search for the private key.

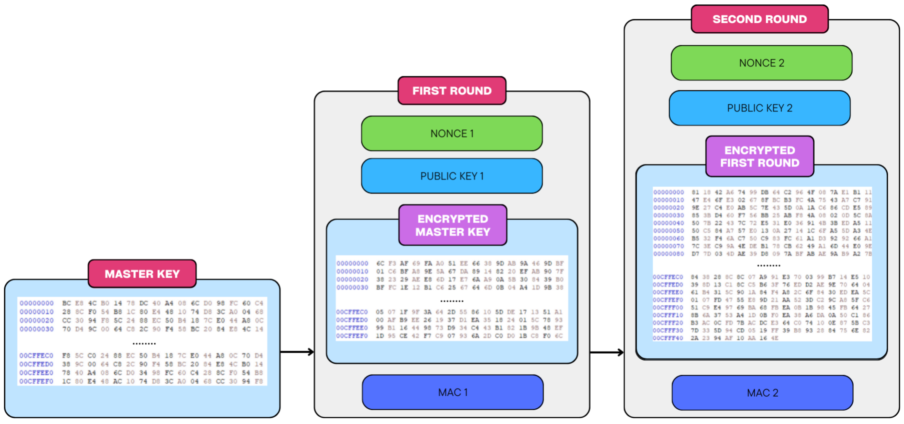
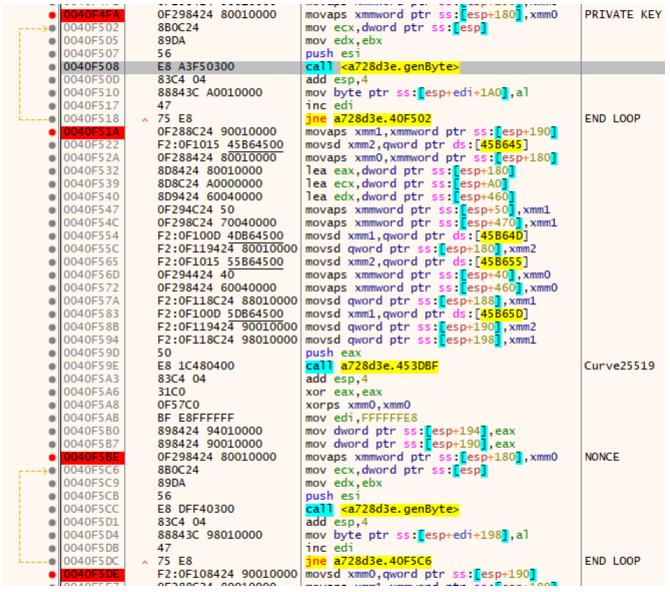
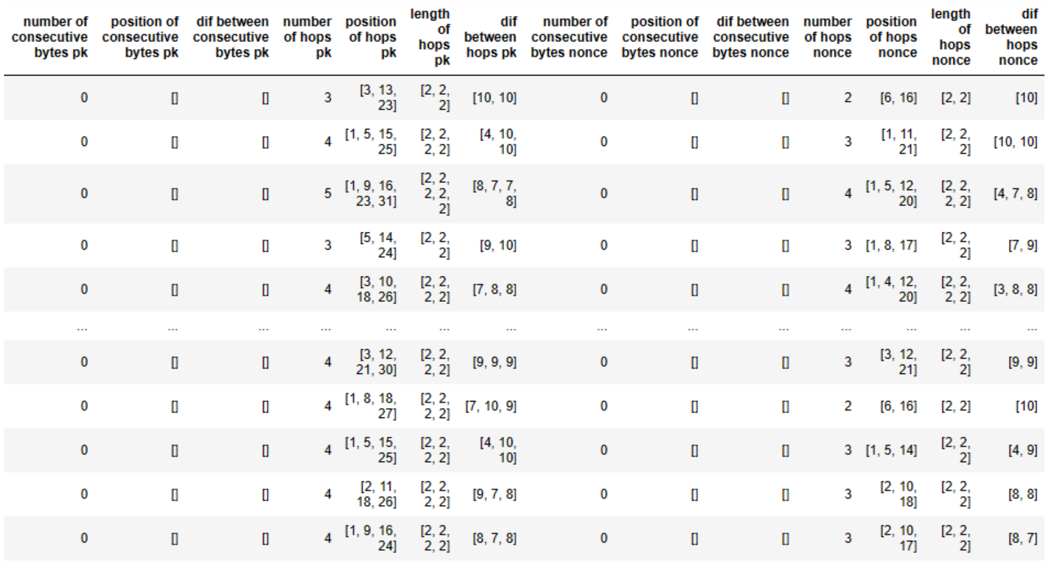


Figure 9: Keystream



```
0040F4FA  0F298424 80010000   movaps xmmword ptr ss:[esp+180],xmm0     PRIVATE KEY
0040F502  8B0C24              mov ecx,dword ptr ss:[esp]
0040F505  89DA                mov edx,ebx
0040F507  56                  push esi
0040F508  E8 A3F50300         call <a728d3e.genByte>
0040F50D  83C4 04             add esp,4
0040F510  88843C A0010000     mov byte ptr ss:[esp+edi+1A0],al
0040F517  47                  inc edi
0040F518  75 E8               jne a728d3e.40F502                       END LOOP
0040F51A  0F288C24 90010000   movaps xmm1,xmmword ptr ss:[esp+190]
0040F522  F2:0F1015 45B64500  movsd xmm2,qword ptr ds:[45B645]
0040F52A  0F288424 80010000   movaps xmm0,xmmword ptr ss:[esp+180]
0040F532  8D8424 80010000     lea eax,dword ptr ss:[esp+180]
0040F539  8D8C24 A0000000     lea ecx,dword ptr ss:[esp+A0]
0040F540  8D9424 60040000     lea edx,dword ptr ss:[esp+460]
0040F547  0F294C24 50         movaps xmmword ptr ss:[esp+50],xmm1
0040F54C  0F298C24 70040000   movaps xmmword ptr ss:[esp+470],xmm1
0040F554  F2:0F100D 4DB64500  movsd xmm1,qword ptr ds:[45B64D]
0040F55C  F2:0F119424 80010000 movsd qword ptr ss:[esp+180],xmm2
0040F565  F2:0F1015 55B64500  movsd xmm2,qword ptr ds:[45B655]
0040F56D  0F294424 40         movaps xmmword ptr ss:[esp+40],xmm0
0040F572  0F298424 60040000   movaps xmmword ptr ss:[esp+460],xmm0
0040F57A  F2:0F118C24 88010000 movsd qword ptr ss:[esp+188],xmm1
0040F583  F2:0F100D 5DB64500  movsd xmm1,qword ptr ds:[45B65D]
0040F58B  F2:0F119424 90010000 movsd qword ptr ss:[esp+190],xmm2
0040F594  F2:0F118C24 98010000 movsd qword ptr ss:[esp+198],xmm1
0040F59D  50                  push eax
0040F59E  E8 1C480400         call a728d3e.453DBF                      Curve25519
0040F5A3  83C4 04             add esp,4
0040F5A6  31C0                xor eax,eax
0040F5A8  0F57C0              xorps xmm0,xmm0
0040F5AB  BF E8FFFFFF         mov edi,FFFFFFE8
0040F5B0  898424 94010000     mov dword ptr ss:[esp+194],eax
0040F5B7  898424 90010000     mov dword ptr ss:[esp+190],eax
0040F5BE  0F298424 80010000   movaps xmmword ptr ss:[esp+180],xmm0     NONCE
0040F5C6  8B0C24              mov ecx,dword ptr ss:[esp]
0040F5C9  89DA                mov edx,ebx
0040F5CB  56                  push esi
0040F5CC  E8 DFF40300         call <a728d3e.genByte>
0040F5D1  83C4 04             add esp,4
0040F5D4  88843C 98010000     mov byte ptr ss:[esp+edi+198],al
0040F5DB  47                  inc edi
0040F5DC  75 E8               jne a728d3e.40F5C6                       END LOOP
0040F5DE  F2:0F108424 90010000 movsd xmm0,qword ptr ss:[esp+190]
```

Figure 10: Generation of private key and nonce



number of consecutive bytes pk	position of consecutive bytes pk	dif between consecutive bytes pk	number of hops pk	position of hops pk	length of hops pk	dif between hops pk	number of consecutive bytes nonce	position of consecutive bytes nonce	dif between consecutive bytes nonce	number of hops nonce	position of hops nonce	length of hops nonce	dif between hops nonce
0	[]	[]	3	[3, 13, 23]	[2, 2, 2]	[10, 10]	0	[]	[]	2	[6, 16]	[2, 2]	[10]
0	[]	[]	4	[1, 5, 15, 25]	[2, 2, 2, 2]	[4, 10, 10]	0	[]	[]	3	[1, 11, 21]	[2, 2, 2]	[10, 10]
0	[]	[]	5	[1, 9, 16, 23, 31]	[2, 2, 2, 2, 2]	[8, 7, 7, 8]	0	[]	[]	4	[1, 5, 12, 20]	[2, 2, 2, 2]	[4, 7, 8]
0	[]	[]	3	[5, 14, 24]	[2, 2, 2]	[9, 10]	0	[]	[]	3	[1, 8, 17]	[2, 2, 2]	[7, 9]
0	[]	[]	4	[3, 10, 18, 26]	[2, 2, 2, 2]	[7, 8, 8]	0	[]	[]	4	[1, 4, 12, 20]	[2, 2, 2, 2]	[3, 8, 8]
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	[]	[]	4	[3, 12, 21, 30]	[2, 2, 2, 2]	[9, 9, 9]	0	[]	[]	3	[3, 12, 21]	[2, 2, 2]	[9, 9]
0	[]	[]	4	[1, 8, 18, 27]	[2, 2, 2, 2]	[7, 10, 9]	0	[]	[]	2	[6, 16]	[2, 2]	[10]
0	[]	[]	4	[1, 5, 15, 25]	[2, 2, 2, 2]	[4, 10, 10]	0	[]	[]	3	[1, 5, 14]	[2, 2, 2]	[4, 9]
0	[]	[]	4	[2, 11, 18, 26]	[2, 2, 2, 2]	[9, 7, 8]	0	[]	[]	3	[2, 10, 18]	[2, 2, 2]	[8, 8]
0	[]	[]	4	[1, 9, 16, 24]	[2, 2, 2, 2]	[8, 7, 8]	0	[]	[]	3	[2, 10, 17]	[2, 2, 2]	[8, 7]

Figure 11: Examples of features retrieved by using 3.5 GHz as frequency

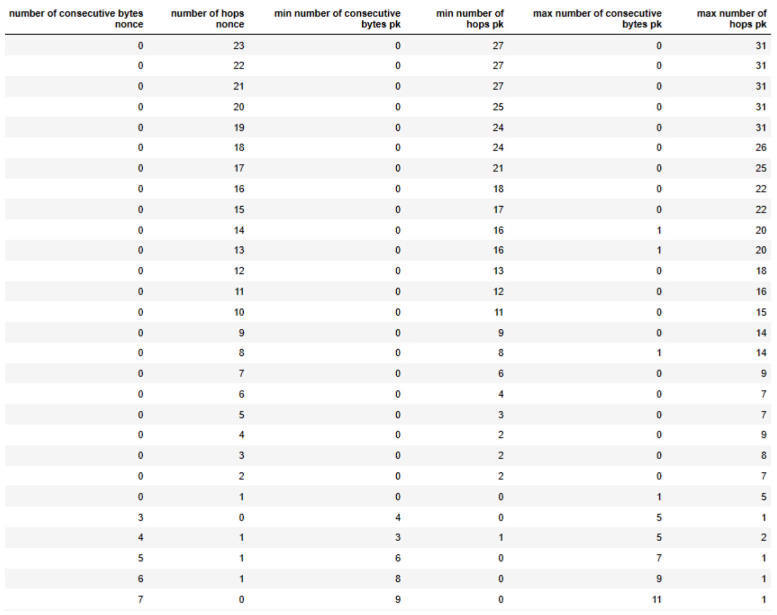
Algorithm 2 Private key generator (case only hops)

```
Input: dif_between_hops_pk, length_hops_pk, public_key, sequence[64]
Output: private_key

Initialization position of hops
$position\_hop \leftarrow []$
$shift\_hop \leftarrow 0$
$position\_hop.append(1)$
$prev \leftarrow position\_hop[0]$

if $len(dif\_between\_hops\_pk) \geq 1$ then
    for $i \leftarrow 0$ to $len(dif\_between\_hops\_pk)$ do
        $position\_hop.append(prev + dif\_between\_hops\_pk[i])$
        $prev \leftarrow position\_hop[i+1]$
    end for
end if
$shift\_hop \leftarrow 32 - position\_hop[-1]$

Search private key
for $k \leftarrow 0$ to $shift\_hop$ do
    for $w \leftarrow 0$ to 64 do
        $private\_key \leftarrow sequence[w]$
        $prec\_position \leftarrow w$
        $count\_length \leftarrow 0$
        for $y \leftarrow 1$ to 32 do
            if $y \in position\_hop$ then
                $cur\_hop \leftarrow length\_hops\_pk[count\_length]$
                $count\_length \leftarrow count\_length + 1$
                $cur\_position \leftarrow (prec\_position + cur\_hop) \bmod 64$
                $cur\_byte \leftarrow sequence[cur\_position]$
                $private\_key \leftarrow private\_key + cur\_byte$
                $prec\_position \leftarrow cur\_position$
            else
                $cur\_position \leftarrow (prec\_position + 1) \bmod 64$
                $cur\_byte \leftarrow sequence[cur\_position]$
                $private\_key \leftarrow private\_key + cur\_byte$
                $prec\_position \leftarrow cur\_position$
            end if
        end for
        $generated\_public\_key \leftarrow curve25519(private\_key, BASE\_POINT)$
        if $generated\_public\_key == public\_key$ then
            return $private\_key$
        end if
    end for
    $position\_hop \leftarrow [x + 1$ for $x \in position\_hop]$
end for
return 0
```



number of consecutive bytes nonce	number of hops nonce	min number of consecutive bytes pk	min number of hops pk	max number of consecutive bytes pk	max number of hops pk
0	23	0	27	0	31
0	22	0	27	0	31
0	21	0	27	0	31
0	20	0	25	0	31
0	19	0	24	0	31
0	18	0	24	0	26
0	17	0	21	0	25
0	16	0	18	0	22
0	15	0	17	0	22
0	14	0	16	1	20
0	13	0	16	1	20
0	12	0	13	0	18
0	11	0	12	0	16
0	10	0	11	0	15
0	9	0	9	0	14
0	8	0	8	1	14
0	7	0	6	0	9
0	6	0	4	0	7
0	5	0	3	0	7
0	4	0	2	0	9
0	3	0	2	0	8
0	2	0	2	0	7
0	1	0	0	1	5
3	0	4	0	5	1
4	1	3	1	5	2
5	1	6	0	7	1
6	1	8	0	9	1
7	0	9	0	11	1

Figure 12: Possible range of features

Version	Release	RAM	CPU	QPF
Win10PRO	22h2	16GB	Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz	10000000
Win10HOME	22h2	8GB	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz	10000000
Win10PRO	20h2	8GB	Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz	10000000
Win10PRO	2004	12GB	Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz	10000000
Win10PRO	2004	8GB	Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz	10000000
Win7PRO	2004	12GB	Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz	2240976
Win10PRO	22H2	8GB	Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz	10000000
Win10PRO	1607	16GB	Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz	2531253
Win10enterprise	21H2	16GB	Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz	10000000
Win10enterprise	21H2	16GB	AMD Ryzen 7 PRO 5850U with Radeon Graphics 1901 MHz	10000000

Table 12: QPF Values of tested Laptops

Listing 1: vulnerable PRNG algorithm

```
//global
const int LENGTH = 0xCFFF00;  // 0xCFFF00 or 0x20 or 0x18

int64_t genByte(int64_t elapsedTimeSeed, int64_t remainderSeed);
void computeElapsedTime(int64_t *elapsedTime, int64_t *remainder);

int main() {
        uint8_t *key; // master key or private key or nonce
        key = (uint8_t*)malloc(LENGTH * sizeof(uint8_t));;

        int64_t elapsedTimeSeed, remainderSeed;
        computeElapsedTime(&elapsedTimeSeed, &remainderSeed);  // seed

        for (int i = 0; i < LENGTH; i++) {
                int64_t difRemainder = genByte(elapsedTimeSeed, remainderSeed);
                uint8_t nextByte = difRemainder & 0xFF;
                key[i] = nextByte;
        }
}

void computeElapsedTime(int64_t *elapsedTime, int64_t *remainder) {

        LARGE_INTEGER numberOfTicks;
        LARGE_INTEGER frequency;
        QueryPerformanceCounter(&numberOfTicks);
        QueryPerformanceFrequency(&frequency);
        *elapsedTime = numberOfTicks.QuadPart / frequency.QuadPart;
        *remainder = numberOfTicks.QuadPart % frequency.QuadPart;
        *remainder = *remainder * (0X3B9ACA00 / frequency.QuadPart);
}

int64_t genByte(int64_t elapsedTimeSeed, int64_t remainderSeed) {
        int64_t elapsedTime, remainder;
        computeElapsedTime(&elapsedTime, &remainder);
        int64_t difRemainder;
        if (remainder > remainderSeed) {
                difRemainder = remainder - remainderSeed;
        }
        else {
                remainder = remainder + 0x3B9ACA00;
                difRemainder = remainder - remainderSeed;
        }
        return difRemainder;
}
```