Detecting Large Integer Arithmetic for Defense Against Crypto Ransomware

Mehmet Sabır Kiraz[1], Ziya Alper Genç[2], and Erdinç Öztürk[3]

[1] TÜBİTAK BİLGEM
mehmet.kiraz@tubitak.gov.tr
[2] University of Luxembourg
ziya.genc@uni.lu
[3] İstanbul Commerce University
erdinco@gmail.com

Abstract. The evolution of crypto ransomware has increasingly influenced real-life systems and lead to fatal threats to data security of individuals and enterprises. A crypto ransomware basically encrypts files of victims using either standard or their own customized crypto functions and request ransom from users to retrieve them again. In this paper, we propose a new detection and analyzing approach, called ExpMonitor, which basically targets ransomware's public key cryptographic algorithms carried out on victim's computer. ExpMonitor is based on observing public key encryption running on the CPU. Monitoring integer multiplication instructions can detect large integer arithmetic operations, which constitute the backbone of public key encryption. While existing detection mechanisms can only targets particular cryptographic functions our technique complements the state-of-the-art.

Keywords: Crypto Ransomware, Malware Analysis, Public Key Encryption, Modular Exponentiation

1 Introduction

Nearly two decades ago, Young and Yung [1] first drafted the frames of a malicious software which encrypts victim's files and as a result, prevents access. In that time, the threat was not considered important, due to the lack of completeness in the infrastructure that would allow a successful and meaningful attack. However, with the appearance of last pieces of the puzzle, TOR network [2] and cryptocurrency, cybercriminals enlarged the attack surface by preventing access to user's files and asking a ransom. Community of information security gave a special name to that kind of malware as *ransomware*.

Ransomware is basically classified into two sets: 1) Locker ransomware, 2) Crypto ransomware. In brief, locker ransomware locks the victim device (or computer) in such a way that it only locks the device not to be usable anymore [3–5]. The good thing about this scenario is that the plain data is never touched.

Namely, the data can be recoverable once the malware is removed or the storage hardware device is moved to a clean device.

On the other hand, due to its easy development and high earning profit, cryptographic ransomware has been very popular amongst cybercriminals. Secure communication over the internet is achieved via a key exchange protocol by generating a fresh session key which is only known to participating parties. Without loss of generality, a fresh session key (between the two participating party Alice and Bob) is either generated through Diffie-Hellman key agreement protocol or the session key is first generated by Alice, then encrypted with the public key of Bob, and finally it is sent to Bob. Once the session key is obtained, data is then securely transmitted through the network. We note that once the key is encrypted using a public key, it can only be decrypted by the owner of the private key corresponding to that particular public key. Therefore, even though entire information flows through the network openly, only the owner of the key that was used to encrypt it can decipher the actual information. Until recently, this protocol worked wonders for the security of data over the internet. Cybercriminals realized the power of this protocol and started to use it for encrypting data in a way that only themselves can decrypt it [6–9]. Via vulnerabilities in the operating systems and phishing attacks, they are able to get their public key and their encryption software into personal computers. Once infected, the crypto ransomware encrypts files and alerts user with a ransom notice. At this point, only the cybercriminals can decrypt any data that has been encrypted. However, paying ransom does not guarantee decryption, as there were many cases where users could not decrypt their files although they successfully paid ransoms.

CryptoLocker, distributed through infected email attachments, was the first well-known cryptographic ransomware which appeared at the end of 2013. According to ZDNet [6], the adversaries received about $27 million by tracing its four bitcoin addresses used by the ransomware. Unfortunately, we have now quite sophisticated threats affecting many users in many regions worldwide, especially developed countries which have high-tech economies [9]. Because new currency technologies like bitcoins are easily transferable between countries and are more difficult to track than conventional payments (due to bypassing banking systems), the criminal rates of ransomware has recently been evolving very rapidly by releasing new versions frequently. Although most of their targets are Windows, they appear recently on Linux and macOS machines. For example, in 2016 KillDisk affects Linux and KeRanger affects OSX. In fact, according to Symantec [7], while there is a decline in traditional ransomware in 2015, cryptographic ransomware now becomes the majority of all ransomware types. However, the existing mechanisms are still behind the expected security guarantees, and preventive and reactive defenses are unfortunately not also sufficient to mitigate the risks of data loss caused by sophisticated ransomware attacks.

1.1 Our Contributions

In this paper, we propose a new detection and analyzing mechanism, called ExpMonitor, based on monitoring asymmetric cryptographic operations which must be carried out by modern ransomware. Our technique leverages the features that exist on a large set of hardwares which yields a cross platform defense mechanism.

The contributions of this paper can be summarized as follows:

– We propose a novel technique that allows a robust detection of ransomware which employs public key encryption in order to maintain its functionality. Our proposal leverages the nature of large integer arithmetics which is an essential part of public key encryption algorithms. All types of cryptographic ransomware that utilize RSA or ECC is detected by ExpMonitor through monitoring hardware instructions for multiplication of large integers.
– Existing ransomware defense systems are implemented as a software which runs on an operating system (OS). These systems commonly assumes that the host OSs do not have any zero-day vulnerability. Unfortunately, previous experiences have shown that this assumption is not realistic. Namely, software based mitigations can be bypassed by advanced ransomware exploiting a zero-day vulnerability in OS kernel. On the contrary, ExpMonitor can be implemented in hardware level and do not need such security assumptions. To the best of our knowledge, ExpMonitor is the first ransomware defense mechanism that is implementable on hardware level, providing a complete mitigation to all ransomware that utilize public key operation. The ability to run on hardware yields a generic solution *i.e.,* OS/platform independence, unlike existing systems.
– Key escrow like mitigation systems are based on monitoring and logging crypto APIs [10]. These systems logs generated keys and random numbers (of all running applications) to later recover the encrypted files. However, if the asymmetric key pair is not generated on victim's computer, *i.e.,* generated on and downloaded from C&C center, then secret key will be able to monitored and hence encrypted files cannot be recovered. Unlike those systems, ExpMonitor can also mitigate ransomware families that immediately encrypts files under the public key of ransomware authors'.

1.2 Organization

In Section 2, we give the necessary background of cryptographic ransomware, present existing defense mechanisms (behavioral analysis, backup, monitoring session keys and RNG, and automated identification of cryptographic primitives in binary programming), and explain their drawbacks. In Section 3, we present our main mechanism ExpMonitor and provide the inner details including dissecting public key algorithms, multi-precision arithmetic. Section 4 discusses the symmetric and asymmetric encryption based ransomware families and limitations of our mechanism ExpMonitor. Finally, Section 5 concludes the paper.

2 Related Work

In this section, we will focus on the most promising defense mechanisms of crypto ransomware.

We note that existing ransomware families use both customized and standard cryptosystems. While cybercriminals utilize standard crypto libraries they can also implement their own cryptosystems. For example, Cryptolocker, CryptoWall, and WannaCry utilize standard Windows crypto libraries to encryption the files [11–13]. The ransomware families has been encreased very significantly because cybercriminals get money from their victims and it is being difficult to trace the payment processes and being easy to exchange to preferred currency.

2.1 Background of Existing Defense Mechanisms of Crypto Ransomware

Behavioral Analysis Based Remediation. Behavioral analysis detection mechanisms (except sandbox-based) generally require the live monitoring of all processes, in order to detect whether any of them behaves with suspicious activities [14–17] Any untrusted (with possibly maliciously behaving) process will be flagged as dangerous and terminated. More concretely, once a device or system is infected, it typically first looks for local files and removable devices such as USB sticks, and then looks network shares. A typical solution to slow down the ransomware is to utilize a sacrificial network share, therefore it will delay the ransomware to reach to the critical data. We note that the network shares should be setup on old slow disks which contain many small random files so that monitoring the networks can alert the suspicious activities (*e.g.,* with Windows Defender Advanced Threat Protection). One of the method for detecting suspicious activity is to utilize file activity monitoring (*i.e.,* the changes in Master File Table and the types of I/O Request Packets to the file system), hence in this way, it is possible to get a real time and historical record of all files and folder activities to detect abnormal file system activity. File renames are not a common action when it comes to activity on network file shares. Usually, in a day, only a few renames are performed even if there are hundreds of users on the network. Whenever there is a ransomware, it will result in a massive increase in file renames due to the encryption of the data. Therefore, it is possible to trigger an alert in the case of this unexpected behavior. Namely, if the number of renames are more than a certain threshold, then there is a potential Ransomware issue with very high probability. We highlight here that crypto ransomware can behave like a user (in order to bypass the file activity monitoring) where it encrypts files slowly. To overcome such a scenario, one can just insert decoy files and actively monitor the activity of these files [18–20]. We now give a brief overview of the following most recently proposed and popular ransomware defense systems which falls into this category.

In [5], Kharraz *et al.* proposed a dynamic analysis system, called Unveil, which executes applications in an artificial environment and monitors files system activities and desktop interactions to detect anomalies which could indicate presence of a ransomware. UNVEIL is the first defense system for ransomware which achieved a high detection rate of 96.3%. However, UNVEIL can also falsely identify high file system activity as a presence of ransomware.

Scaife *et al.* in [21] designed a detection system, called CRYPTODROP, which monitors common indicators of ransomware and terminates if a process performs suspicious activity. In the experiments, the authors show that CRYPTODROP could detect all of the ransomware families, though, the detection happened after a number of files are encrypted, the median loss is about 10 files (out of nearly 5100 available files).

In [22], Continella *et al.* proposed a file system driver, called SHIELDFS, which detects malicious activity at runtime and allows rollback of modifications made by ransomware. SHIELDFS detects an abnormal activity based on monitoring I/O operations of huge number of benign applications and analyzing the collected data to characterize I/O activity of ransomware. While being able to recover 100% of encrypted files, SHIELDFS brings a runtime overhead by 26%.

Backup Systems. Regular backups are usually the first step to mitigate both all zero-day attacks and ransomware. Unfortunately, there is a misbelief about the reliability of backup systems: according to Barkly [23], only 42% of users could fully recover their files from backups after a ransomware attack. On the other hand, assuming that all users are expected to be able to follow the complicated rules is also not realistic. In any case, we still believe that backups are a good approach to provide for zero-day attacks.

Monitoring Session Keys and RNG for Remediation. To the best our knowledge, Kolodenker *et al.*'s mechanism, called PAYBREAK, is the first and only ransomware defense mechanism which monitors calls to crypto APIs and collects the parameters such as encryption keys and type of algorithm [10]. Collected information is then stored in a key vault and retrieved to recover the encrypted files after a ransomware attack. Only the legitimate user is authorized to access key vault. In cryptographic literature, storing session keys is called key escrow and that concept has been debated since decades. The authors of PAYBREAK claims that their proposal is essentially different from government mandated key escrow systems by arguing that there is exactly one entity who can access PAYBREAK's key vault.

One of the major drawback of PAYBREAK is that it logs session keys of all running applications in a key vault acting as a key escrow (key recovery) agent. It would be reasonable to expect that key vault would be a valuable target for adversaries and may lead to fatal security or privacy issues once it is disclosed. We believe that such a single point of failure would constitute a big risk for enterprises. For instance, TLS, SSH and VPN protocols (which run in Application Level of OSI Model) promise *forward secrecy* which is contributed by *ephemeral keys*. Such a key escrow system would diminish that feature.

Another shortcoming for PAYBREAK will arise when a ransomware downloads a public key from its C&C center and encrypts files directly under that public key instead of a hybrid encryption scheme. Although this approach requires heavy computations and there is no known ransomware that uses this technique, nothing can stop cybercriminals from creating such a malware. Finally, PayBreak needs to know the signature of the third party crpyto libraries to detect and log the parameters, otherwise PayBreak cannot recover any encrypted files. As the authors of [10] acknowledges, signature based mitigation can be bypassed by using advanced packers and obfuscation methods. In brief, the authors experimented PayBreak for twenty real-world ransomware and could recover the files from twelve of them. Also, Paybreak cannot hook if ransomware families utilize their own cryptosystems as it only hooks standard API functions like CryptoAPI of Windows.

We would like to stress that we do not share the same opinion with Paybreak about the assumptions on the key vault. We believe that zero-day attacks can also be used by ransomware authors as in the Wannacry case [13]. For example, a ransomware can encrypt the files and then destroy the key vault in order to make recovery impossible. Also, the authors also mention that the size of the key vault could be very large (like 1 TB) with privileged access to prevent being filled with garbage data and alert the user in case of an attack. To make their scheme more practical they also discuss to minimize the size the key vault by, for example, having a secure rotation for the key vault. We highlight that the practicality of minimizing the key fault is not an easy task, for example in such a secure rotation, a ransomware can still encrypt files and then fill garbage data into the key vault to run over the keys again.

Automated Identification of Cryptographic Primitives in Binary Programs. This approach aims to detect a particular cryptographic algorithm for a given binary software in order to learn which algorithms and keys are used [24–26]. In [24], the authors propose a mechanism which can precisely identify cryptographic code in a given binary. However, the authors deal with only symmetric encryption algorithms by monitoring bitwise arithmetic instructions and claim that they can successfully extract the cryptographic parameters from a given malware binary. One of the drawback of this system is that it cannot detect a ransomware having its own proprietary cryptographic algorithm. For each instance of an executed function, the authors compute the ratio of bitwise arithmetic instructions. If the functions is executed for at least 20 times and the ratio is higher than 55%, then the function is flagged as an encryption/decryption function. Their experiments detected 94.6% of AES encryptions including the corresponding parameters where the missed 5.4% of AES instances were due to the memory reconstruction method. One other drawback of this mechanism also cannot detect if a ransomware implements his own cryptographic algorithm.

Similarly, in [25], the authors propose a mechanism to automatically target only symmetric encryption algorithms and their parameters inside binary code. Their system uses Data Flow Graph (DFG) isomorphisms of to identify symmetric cryptographic algorithms. However, the proposed system does not handle code-obfuscation techniques which is commonly used by malware to hide malicious functions, hence cannot mitigate ransomware. Very recently, [26] proposed another technique called *bit-precise symbolic loop mapping* to identify cryptographic functions in obfuscated binary code. Their approach basically is based on the comparison with known reference implementations. Namely, their scheme first captures the semantics of possible cryptographic algorithms with bit-precise symbolic execution in a loop. Then they perform guided fuzzing to efficiently match Boolean formulas with the reference implementations. While their solution can detect obfuscated malwares, it is again limited to referenced implementations because their mechanism cannot detect if cybercriminals do not utilize referenced implementations. Also, their technique could easily be circumvented by an attacker via replacement of Boolean functions and even simple instruction reordering or via function stitching [27].

3 Our Proposed Mechanism: ExpMonitor

3.1 Overview

While symmetric encryption requires high percentage of bitwise arithmetic instructions, asymmetric encryption requires high percentage of large integer arithmetic instructions. ExpMonitor basically aims to detect a ransomware which utilize public key operations (see Figure 1). It is a complement of existing malware defense and forensics including automated identification schemes for cryptographic primitives in binary programs. Before we proceed, we fist note that public key cryptographic operations have various mathematical constructs. As it is very crucial for us to be able to detect if a CPU is running a public key operation, it is imperative for us to determine common properties of these various constructs. We can summarize these common properties and provide an overview of the identification methodology as follows:

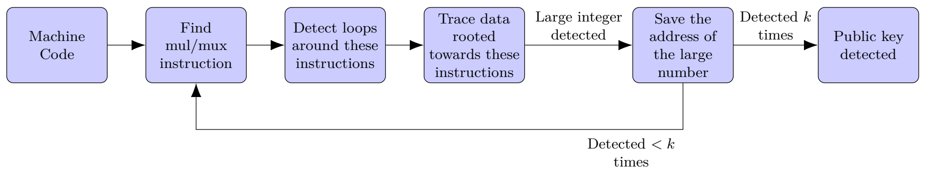
Machine Code
Find mul/mux instruction
Detect loops around these instructions
Trace data rooted towards these instructions
Large integer detected
Save the address of the large number
Detected k times
Public key detected
Detected $< k$ times

Fig. 1. Overview of ExpMonitor

1. Public key cryptographic operations make excessive use of arithmetic of large integers (which is widely used in practice such as in SSL/TLS, web servers, Certificate Authorities). The most common public key operations utilize very large parameters and cryptographic keys. These mathematically complex operations utilize algorithms that work on very large integers. Thus, we can state that large-integer arithmetic is one of the common criteria to be examined in a system that is trying to detect ransomware.

2. Code of public key cryptography contains loops. Modern CPU architectures have various sizes of core multipliers. A 64-bit CPU usually has a 64-bit multiplier and a 32-bit CPU has a 32-bit multiplier. Since the ratio of operand size to core multiplier is too high, public key cryptographic operations need to perform excessive integer multiplication operations. In order to be able to have scalable code, these operations are looped over small pieces of multiplication codes. For example, a 512x64 bit multiplication code could be written as a function and a 512x512 bit integer multiplication will be calling this function 8 times.
3. Input and output to code of public key cryptography have a predefined, verifiable relation. We can also determine what cryptographic parameters have been used. Since the encrypted value for ransomware applications is usually the session key, the input operands for large-integer arithmetic have very high entropy.

3.2 Dissecting Public Key Algorithms

For most of the ransomware attacks, RSA is being used for public key encryption. Therefore, we will here focus on the RSA setting but we highlight the procedures described below can also be applicable to elliptic curve setting. Before we proceed, we present below some background information on the RSA Encryption and Decryption Algorithms [28]. Let p and q be prime numbers, $N = pq$ and $1 \equiv ed \pmod{(p-1)(q-1)}$. RSA encryption algorithm is explained in Algorithm 1 and decryption algorithm is explained in Algorithm 2.

Algorithm 1 RSA Encryption

```
1: procedure Enc($M, (e, N)$)
2:     $C \leftarrow M^e \pmod{N}$
3:     return $C$
4: end procedure
```

Algorithm 2 RSA Decryption

```
1: procedure Dec($C, (d, N)$)
2:     $M \leftarrow C^d \pmod{N}$
3:     return $C$
4: end procedure
```

Fig. 2. RSA Algorithm.

As can be seen from Algorithm 1, the most computationally heavy part of RSA encryption operation is modular exponentiation. As it is our goal to figure out when public key operations are queued to be executed on a CPU, we decided to examine the modular exponentiation operations.

Modular exponentiation operation can be realized in many ways. Algorithm 3 shows left–to–right binary exponentiation algorithm, which is also known as square–and–multiply algorithm. Algorithm 4 shows right–to–left binary exponentiation algorithm. Algorithm 5 shows left–to–right binary exponentiation algorithm with a fixed–window length of 2 bits.

Algorithm 3 Left-to-right binary exponentiation with square-and-multiply method

```
procedure LeftToRight($g, n, e = (e_m e_{m-1} \dots e_1 e_0)_2$)
    $A \leftarrow 1$
    for $i \leftarrow mto0$ do
        $A \leftarrow A^2 \mod n$
        If $e_i = 1$ then $A \leftarrow Ag \mod n$
    end for
    return $A$                                   ▷ $A := g^e \mod n$
end procedure
```

Algorithm 4 Right-to-left binary exponentiation

```
procedure RightToLeft($g, n, e = (e_m e_{m-1} \dots e_1 e_0)_2$)
    $A \leftarrow 1$
    for $i \leftarrow 0, m$ do
        If $e_i = 1$ then $A \leftarrow Ag \mod n$
        $A \leftarrow A^2 \mod n$
    end for
    return $A$                                   ▷ $A := g^e \mod n$
end procedure
```

Algorithm 5 Left-to-right k-ary modular exponentiation with k-bit fixed windowing

```
procedure Precomputation($g, n, k$)
    $g_0 = 1$
    for $i \leftarrow 1, (2^k - 1)$ do
        $g_i \leftarrow g_{i-1} g \mod n$
    end for
end procedure
procedure LeftToRightWindowing($g, n, e = (e_m e_{m-1} \dots e_1 e_0)_2$, where $b = 2^k$
for some $k \geq 1$)
    $A \leftarrow 1$
    for $i \leftarrow n, 0$ do
        for $j \leftarrow 0, k - 1$ do
            $A \leftarrow A^2 \mod n$
        end for
        $A \leftarrow A g_{e_i} \mod n$
    end for
    return $A$                                   ▷ $A := g^e \mod n$
end procedure
```

Although there are many other binary exponentiation algorithms, we can easily state with information at hand that the essential building blocks of a binary exponentiation operation are large–integer arithmetic operations. Although there are several methods for implementing modular exponentiation, these methods have a very important operation in common: modular multiplication of large integers.

We highlight that elliptic curve cryptography (ECC) is also another commonly used approach for public key operations. There are several protocols involving ECC which can be considered as public key cryptography. However, the most compute-intensive part of ECC operations is common for all these protocols, i.e., scalar point multiplication. As for modular exponentiation, there are various methods for realizing scalar point multiplication. And as for modular exponentiation, these methods consist mostly of one operation: modular multiplication of large integers. We can conclude that modular multiplication of large integers is the most time–consuming component of public key cryptography.

As above statement is true for prime field ECC operations, it can be modified slightly to be valid for binary field ECC applications. Binary field ECC operations involve polynomial arithmetic rather than integer arithmetic. High level operations look very similar to prime field operations. However, low-level building blocks are very different. Multiplication of two large polynomials could be utilized using bitwise arithmetic operations, which can be detected using methods stated in [24]. Also, in 2011, Intel added a Carry-Less Multiplication Instruction into their Westmere architecture. This instruction is called PCLMULDQ and it multiplies two 63-degree polynomials (which are represented as 64-bit binary numbers) and results in a 127-degree polynomial (which is represented as a 128-bit binary number). This instruction could be utilized to realize ECC operations and characteristics of this instruction could be utilized to detect ECC operations.

Multi-Precision Arithmetic. Multiplication of two integers that are smaller than the machine word size is trivial for any machine that includes a core multiplier. Usually, the core multiplier that is included in the machine can multiply two word-sized integers with a 1 cycle/multiply throughput. However, multiplication of integers that are larger than the machine word size becomes a challenge for any programmer, as the size of the operands grow. Multi-precision arithmetic involves operations on integers that are larger than the machine word size. Since the most time-consuming and challenging arithmetic operation is multiplication, we only deal with multi-precision multiplication operation in this section. For the remainder of the paper, we use the term *large integer* for any integer that is larger than the machine word size.

Multi-Precision Multiplication. There are multiple methods of efficiently multiplying two large integers. Karatsuba-Ofman Algorithm [29], FFT-Based Multiplication Algorithm [30] are examples that can be listed among the classical school-book multiplication algorithm. Even though these complicated algorithms have a potential for more efficient implementations, we will examine and utilize the classical school-book multiplication algorithm for the scope and purpose of this paper.

We assume that the integers to be multiplied are of the same size, which is a more realistic case for public key applications. However, the methods that we discuss here can be applied to multiplication of different-sized integers. For simplicity, we pick our word size as 64 bits, as it is recently the most common case for processors. It should also be noted that our methods can be applied to multiplications with different word sizes. Assume we multiply two n-word integers A and B where $A = (A_{n-1}A_{n-2} \dots A_1A_0)_b$ and $B = (B_{n-1}B_{n-2} \dots B_1B_0)_b$ and $b = 2^{64}$. Algorithm 6 is detailing how to realize the multiplication of these two integers.

Algorithm 6 Classical school-book multiplication algorithm

```
1: procedure MontMul($(A, B, N)$)    ▷ $n$-bit integers $A = (A_{n-1}...A_1A_0)_b$,
   $B = (B_{n-1}...B_1B_0)_b$, $b = 2^{64}$
2:     for $t \leftarrow 0, n-1$ do    ▷ $t = (t_{2n-1}...t_1t_0)_b$
3:         $(t_{i+n}t_{i+n-1}...t_{i+1}t_i)_b \leftarrow (0t_{i+n-1}...t_{i+1}t_i)_b + A_iBb^i$
4:     end for
5:     return $t_{2n-1}...t_1t_0$    ▷ $AB = t_{2n-1}...t_1t_0$
6: end procedure
```

Algorithm 6 is illustrated in Figure 3 for the multiplication of two 512-bit integers. In this figure, we call each A_iB multiplication a diagonal, as the detailed multiplication figure looks like a diagonal as can be seen in Figure 4.

Software Implementations. In this section, we will present detailed information about implementations of modular multiplication on software.

Figure 3 shows only a breakdown of multiplication of large integers. Each A_iB multiplication needs to be examined more closely, as they are multiplication of a large integer with a single word. It should be noted that first diagonal is treated differently than the rest of the diagonals, as the result of the first diagonal does not need to be accumulated but the results of the rest of the diagonals need to be accumulated with the previous results. Figure 4 gives detailed explanation of multiplication of the first diagonal of a 512x64 bit multiplication operation.

Here, each multiplication is a 64x64 core multiplication and each result is a 128-bit integer. Low 64 bits of each result is stored in rax ad high 64 bits of each result is stored in rdx. A code snippet for the first diagonal is given below.

```
mov rbp, [A + 8*0]
mov rax, [B + 8*0]                  mov rax, [B + 8*1]
mul rbp                             mul rbp
                                    add R0, rax
mov [C + 8*0], rax                  adc rdx, 0
mov R0, rdx                         mov R1, rdx
```

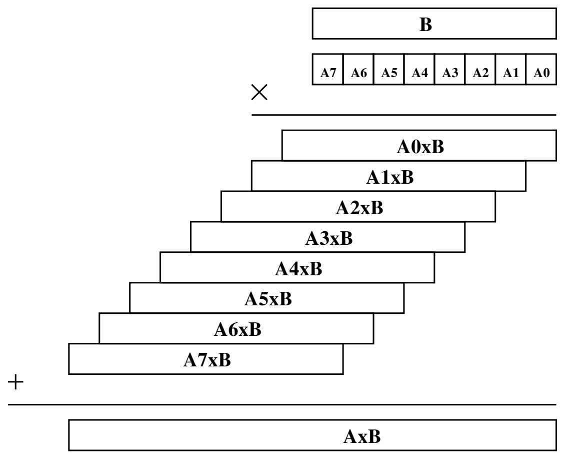

Fig. 3. Classical Multiplication of two 512-bit integers.

```
...
...
mov rax, [B + 8*6]
mul rbp
add R5, rax
adc rdx, 0
mov R6, rdx
```

```
mov rax, [B + 8*7]
mul rbp
add R6, rax
adc rdx, 0
mov R7, rdx
```

In Figure 4, it can be seen that the result C0 is stored in memory, as it will not change after this diagonal is finished. Results C1 through C8 are stored in registers R0 through R7, to avoid load-store operations. *mul* instruction operates on two 64-bit integers. One of these integers are implicitly stored in register *rax* and the other is stored in any register that will be given as input to the mul instruction. The output of mul instruction is stored in registers rdx and rax.

```
mul src
{rdx,rax} = src*rax
```

defines the multiply operation. This instruction has two main flaws:

- Requires the outputs to be moved to other registers once the multiplication is completed, as the next multiplication will destroy the current result
- It destroys all of the flags.

Intel introduced a new instruction mulx [31] (formerly known as Haswell), which operates as follows:

```
mulx dest_hi, dest_lo, src1
dest_hi:dest_lo = src1 * rdx.
```

This mulx instruction was introduced to 4^{th} generation Intel cores and in addition to being able to define the destination registers, it does not destroy the carry

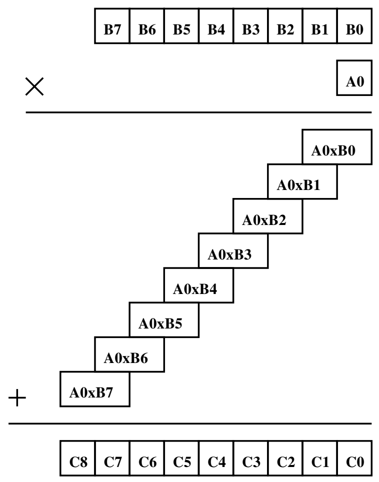

Fig. 4. First diagonal of a 512x64 bit multiplication operation

flag, which allows a better code sequence as shown below for the first diagonal of the multiplication.

```
mov rdx, [A + 8*0]

mulx R0, rax, [B + 8*0]
mov [C + 8*0], rax

mulx R1, rax, [B + 8*1]
add R0, rax

mulx R2, rax, [B + 8*2]
adc R1, rax
...
...
mulx R6, rax, [B + 8*6]
adc R5, rax

mulx R7, rax, [B + 8*7]
adc R6, rax
adc R7, 0
```

After the first diagonal, we have an intermediate result and next results of diagonal multiplication need to be accumulated onto this intermediate result. Figure 5 gives detailed explanation of multiplication of the second diagonal of a 512x64 bit multiplication operation. The rest of the diagonals behave exactly the same way. It should be noted that after each multiplication operation, there are three 64-bit integers to be added: high part of the previous result, low part of the current result and corresponding intermediate result. This is realized with code sequence below.

```
mov rbp, [A + 8*0]
mov rax, [B + 8*0]
mul rbp
add     R0, rax
adc rdx, 0
mov [C + 8*0], R0
mov rbx, rdx

mov rax, [B + 8*1]
mul rbp
```

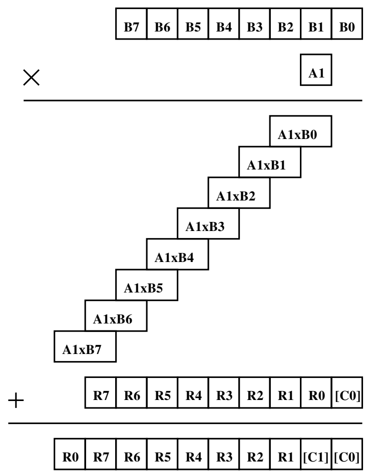

Fig. 5. Second diagonal of a 512x64 bit multiplication operation

```
mov R0, rdx
add R1, rax
adc R0, 0
add R1, rbx
adc R0, 0
...
...
```

```
mov rax, [B + 8*7]
mul rbp
mov R0, rdx
add R7, rax
adc R0, 0
add R7, rbx
adc R0, 0
```

It can be seen from Figure 5 is that there are actually two seperate carry chains. In order to be able to run these two carry chains in parallel, Intel introduced two more instructions: adcx and adox. These instructions are described as follows:

```
adcx dest/src1, src2
adox dest/src1, src2
```

Here, adcx instruction operates exactly same way as adc instruction, but does not alter the overflow flag (OF). On the otherhand, adox instruction operates exactly same way as adc instruction, but uses the OF as its carry flag and does not change the carry flag (CF). This allows parallel execution of two seperate carry chains. This is realized with code sequence below.

```
xor rax, rax

mov rdx, [A + 8*0]

mulx rbx, rbp, [B + 8*0]
adox R0, rbp
adcx R1, rbx
```

```
mov [C + 8*0], R0

mulx rbx, R0, [B + 8*1]
adox R0, R1
adcx R2, rbx
...
...
```

```
mulx rbx, R5, [B + 8*6]
adox R5, R6
adcx R7, rbx

mulx rbx, R6, [B + 8*7]
```

```
adox R6, R7
adcx rbx, rax
adox rbx, rax
mov R7, rbx
```

Algorithm Analysis for Achieving ExpMonitor. As stated above, the most compute–intensive operation for RSA is modular exponentiaton and analysis of modular exponentiation algoritm shows that the most time–consuming operation is multi–precision multiplication. Therefore, if we are able to detect if code for multiplication of large integers is included in some piece of software that will be run on the CPU, we will be able to detect if the CPU will be running RSA operations and take necessary measures.

There are multiple ways of handling large integer multiplications and various software implementations are shown in Section 3.2. It is also possible to produce different implementations. However, we can state that entire multiplication operation is best and most commonly realized using the core multipliers in the ALU of the processor core. Since unsigned multiplication is required for large–integer arithmetic, number of possible instructions required to be examined reduces down to two: mul and mulx.

If school–book multiplication algorithm is used, regardless of the implementation, to multiply two n–bit integers, we need to perform $(n/64)^2$ core multiplication, which means that $(n/64)^2$ mul or mulx instructions need to be issued. With this data in hand, it needs to be noted that these multiply instructions require consequtive data from the memory. Therefore, we can state that if too many mul or mulx instructions are utilized in a piece of code, and if these instructions are accessing consequtive memory locations, there is a strong chance that the piece of code is realizing public key operations.

One family of algorithm that has a similar property is signal processing. Signal processing algorithms also utilize many multiply and add operations, which could cause a confusion. However, there is a major difference between signal processing operations and large–integer multiplication operations. Signal processing operations accumulate the results of the multiplications and large–integer multiplication operations add the results of the multiplication operations in consequtive addresses in memory or onto different registers. Therefore, in addition to examining the number of mul or mulx instructions that are utilized by a program, one needs to also examine where the results of these operations are being sent to. Another property that could be examined is the entropy of the data that is utilized for mul or mulx instructions. For public key operations, the integers to be multiplied will be random integers and will have a large Hamming distance. Therefore, if the analysis could be done in real time and if the memory could be examined, this information could also be used to detect if public key operations are performed on a CPU.

Knowing this countermeasure, an attacker can try to insert dummy mul/mulx instructions and confuse the software protecting the CPU. However, if the countermeasure is well-constructed and is able to follow the pointers and data that is used by these instructions, it will be able to detect these dummy instructions.

Another important aspect of our countermeasure is that it has to incorporate physical input from the user. Any countermeasure will run by the operating system and if the operating system is compromised, attacker can bypass the outcome of our countermeasure. In order to prevent this from happening, whenever the system detects a public key operation, it will ask the user for keyboard input, either to allow or block the public key operation. If this could be realized, the attacker will be completely blocked.

4 Discussion & Limitations

ExpMonitor does not provide any detection mechanisms for ransomware that utilize only symmetric algorithms. However, it would not be easy for ransomware authors to manage the keys while utilizing only symmetric algorithms. Let's assume that a ransomware is utilizing only symmetric primitives. In this case, once infected, ransomware needs to either obtain encryption keys from the C&C server or generate the keys on the fly at the victim's computer.

– In the former case, it would be required to exist an active connection between the victims and the C&C server, and to maintain a table of victim ids and corresponding encryption keys on the C&C server (in order to deliver the keys to the corresponding victims). Obviously, as the ransomware gets spread, this would complicate the key management and increase the burden of network connections.
– In the latter case, ransomware needs to store encryption keys on the victim's computer using a master secret. However, this secret can be retrieved via reverse engineering techniques.

Therefore, by the nature of public key cryptosystems, we highlight that modern ransomware families need to employ hybrid encryption schemes that utilize both symmetric and asymmetric primitives.

It is also interesting to consider multi-core systems since monitoring computations require to trace which cores to be computed sequentially. We stress that this requirement not only apply to ExpMonitor but also to the existing mechanisms in [24–26]. Multi-core systems could be traced via monitoring high-level common cache structures. For example, modern Intel processors utilize L1 and L2 cache structures for each core and a common L3 cache that is shared by all the cores. Thus, any run-time monitoring could be realized via monitoring the L3 cache structure. Furthermore, we would like to note that large integer arithmetic could be realized without core multiply operations. Instead of a word-level multiplication, one can choose to utilize a bit-level multiplication structure, which requires a significant amount of and instructions. This approach is also likely to be used, albeit very slow and unfeasible. Since and instructions replace multiply instructions for this approach, large integer multiplication consists of bitwise arithmetic instead of integer arithmetic. This could be detected via the detection scheme utilized in [24].

5 Conclusion

In this paper, we present a new enhanced approach, called ExpMonitor, which helps to mitigate vast majority of crypto ransomware including several previously undefeated ones such as [32, 33]. ExpMonitor only deals with detection and analyzing rather than prevention. Therefore, it is interesting to enhance our system by adding the prevention mechanism. In the future, ransomware will be believed to still remain a major and rapidly growing threat by the help by anonymizing networks like TOR and payment methods like bitcoin.

References

1. Young, A., Yung, M.: Cryptovirology: Extortion-based security threats and countermeasures. In: Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on. pp. 129–140. IEEE (1996)
2. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13. pp. 21–21. SSYM'04 (2004)
3. Kevin Savage, Peter Coogan, H.L.: Symantec: Security response: The evloution of ransomware (August 2015), http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-evolution-of-ransomware.pdf (accessed on 2017-05-24)
4. James Scott, D.S.: The icit ransomware report (March 2016), http://icitech.org/wp-content/uploads/2016/03/ICIT-Brief-The-Ransomware-Report2.pdf (accessed on 2017-05-23)
5. Kharraz, A., Arshad, S., Mulliner, C., Robertson, W.K., Kirda, E.: UNVEIL: A large-scale, automated approach to detecting ransomware. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 757–772 (2016)
6. James Scott, D.S.: Cryptolocker's crimewave: A trail of millions in laundered bitcoin (December 2013), http://www.zdnet.com/article/cryptolockers-crimewave-a-trail-of-millions-in-laundered-bitcoin/ (accessed on 2017-05-23)
7. Symantec: Istr-internet security threat report (April 2016), https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf/ (accessed on 2017-05-23)
8. McAfee: Mcafee labs threat advisory teslacrypt ransomware (July 2016), https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/25000/PD25854/en_US/McAfee_Labs_Threat_Advisory-TeslaCrypt.pdf (accessed on 2017-06-01)
9. Symantec: Internet security threat report-istr, volume 22 (April 2017), https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf (accessed on 2017-05-23)
10. Kolodenker, E., Koch, W., Stringhini, G., Egele, M.: Paybreak: Defense against cryptographic ransomware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 599–611. ASIA CCS '17, ACM (2017)
11. Rajpal, V.K.M.S.: Bromium: Understanding Crypto-Ransomware-In-Depth Analysis of the Most Popular Malware Families (2015), https://www.bromium.com/sites/default/files/rpt-bromium-crypto-ransomware-us-en.pdf

12. Cabaj, K., Mazurczyk, W.: Using software-defined networking for ransomware mitigation: The case of cryptowall. IEEE Network 30(6), 14–20 (November 2016)
13. CERT-EU: Cert-eu security advisory 2017-012, wannacry ransomware campaign exploiting smb vulnerability (May 2017), https://cert.europa.eu/static/SecurityAdvisories/2017/CERT-EU-SA2017-012.pdf (accessed on 2017-06-01)
14. Bayer, U., Kruegel, C., Kirda, E.: Ttanalyze: A tool for analyzing malware p. 180–192 (2006)
15. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy 5(2), 32–39 (Mar 2007), http://dx.doi.org/10.1109/MSP.2007.45
16. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A layered architecture for detecting malicious behaviors. In: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection. pp. 78–97. RAID '08, Springer-Verlag, Berlin, Heidelberg (2008)
17. Martignoni, L., Paleari, R., Bruschi, D.: A framework for behavior-based malware analysis in the cloud. In: Information Systems Security: 5th International Conference, ICISS 2009 Kolkata, India, December 14-18, 2009 Proceedings. pp. 178–192. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
18. Yuill, J., Zappe, M., Denning, D., Feer, F.: Honeyfiles: deceptive files for intrusion detection. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004. pp. 116–122 (June 2004)
19. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Security and Privacy in Communication Networks: 5th International ICST Conference, SecureComm 2009, Athens, Greece, September 14-18, 2009, Revised Selected Papers. pp. 51–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
20. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: A look under the hood of ransomware attacks. In: Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148. pp. 3–24. DIMVA 2015, Springer-Verlag New York, Inc., New York, NY, USA (2015)
21. Scaife, N., Carter, H., Traynor, P., Butler, K.R.B.: Cryptolock (and drop it): Stopping ransomware attacks on user data. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 303–312 (June 2016)
22. Continella, A., Guagnelli, A., Zingaro, G., De Pasquale, G., Barenghi, A., Zanero, S., Maggi, F.: Shieldfs: A self-healing, ransomware-aware filesystem. In: Proceedings of the 32Nd Annual Conference on Computer Security Applications. pp. 336–347. ACSAC '16, ACM (2016)
23. Kevin Savage, Peter Coogan, H.L.: Symantec: Ransomware by the numbers: Must-know ransomware statistics 2016 (August 2016), https://blog.barkly.com/ransomware-statistics-2016 (accessed on 2017-05-26)
24. Gröbert, F., Willems, C., Holz, T.: Automated Identification of Cryptographic Primitives in Binary Programs, pp. 41–60. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
25. Lestringant, P., Guihéry, F., Fouque, P.A.: Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. pp. 203–214. ASIA CCS '15, ACM (2015)
26. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: 2017 IEEE Symposium on Security and Privacy (SP) (May 2017)

27. Gopal, V., Feghali, W., Guilford, J., Ozturk, E., Wolrich, G., Dixon, M., Locktyukhin, M., Perminov, M.: Fast Cryptographic Computation on Intel Architecture Processors Via Function Stitching (April 2010), http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-cryptographic-paper.pdf
28. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (Feb 1978)
29. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Proceedings of USSR Academy of Sciences 145(7), 293–294 (1962)
30. Schönhage, A., Strassen, V.: Schnelle multiplikation großer zahlen. Computing 7(3), 281–292 (1971)
31. Intel: 4th generation intel core processor based on mobile u-processor and y-processor lines datasheet, volume 1 of 2 (December 2013), http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-mobile-u-y-processor-lines-vol-1-datasheet.pdf (accessed on 2017-06-01)
32. Varsanov, E.: Remove Gerkaman@aol.com Ransomware (September 2016), http://www.virusresearch.org/remove-gerkamanaol-com-ransomware/ (accessed on 2017-05-26)
33. Group, T.: Threat Spotlight: TeslaCrypt - Decrypt It Yourself (April 2015), http://blogs.cisco.com/security/talos/teslacrypt (accessed on 2017-05-26)