CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution

Xin Li, *Member, IEEE,* Xinyuan Wang, *Member, IEEE,* Wentao Chang, *Member, IEEE*

***Abstract*—Malwares are becoming increasingly stealthy, more and more malwares are using cryptographic algorithms (e.g., packing, encrypting C&C communication) to protect themselves from being analyzed. The use of cryptographic algorithms and truly transient cryptographic secrets inside the malware binary imposes a key obstacle to effective malware analysis and defense.**

To enable more effective malware analysis, forensics and reverse engineering, we have developed *CipherXRay* – a novel binary analysis framework that can automatically identify and recover the cryptographic operations and transient secrets from the execution of potentially obfuscated binary executables. Based on the *avalanche effect* of cryptographic functions, CipherXRay is able to accurately pinpoint the boundary of cryptographic operation and recover truly transient cryptographic secrets that only exist in memory for one instant in between multiple nested cryptographic operations. CipherXRay can further identify certain operation modes (e.g., ECB, CBC, CFB) of the identified block cipher and tell whether the identified block cipher operation is encryption or decryption in certain cases.

We have empirically validated CipherXRay with OpenSSL, popular password safe KeePassX, the ciphers used by malware Stuxnet, Kraken and Agobot, and a number of third party softwares with built-in compression and checksum. CipherXRay is able to identify various cryptographic operations and recover cryptographic secrets that exist in memory for only a few microseconds. Our results demonstrate that current software implementations of cryptographic algorithms hardly achieve any secrecy if their execution can be monitored.

***Index Terms*—Binary analysis, avalanche effect, key recovery attack on cryptosystem, transient cryptographic secret recovery, secrecy of monitored execution, reverse engineering.**

I. Introduction

Malware analysis, forensics and reverse engineering seek to understand the inner workings of malware, which are invaluable to defending against malware. To prevent themselves from being analyzed and reverse engineered, more and more malwares (e.g., Agobot, MegaD, Kraken, Conficker) are using cryptographic algorithms (e.g., packing [51], encrypting C&C communication) to protect the malicious code and communication [40]. To prevent the in-memory cryptographic secrets (e.g., key, IV) from being recovered by key searching tools (e.g., rsakeyfind), sophisticated malware can make the cryptographic secrets truly transient in memory by encrypting or destroying the secrets right after using them at run-time. The use of cryptographic algorithms and truly transient cryptographic secrets inside the malware binary executable imposes a key obstacle to effective malware analysis and defense. In order to recover the true logic of packed malware and the plaintext of the encrypted malware C&C communication, we have to be able to effectively analyze the cryptographic operations and recover their secrets from the malware binary executable.

Xin Li, Xinyuan Wang and Wentao Change are with the Department of Computer Science, George Mason University, Fairfax, VA 22030, USA. email: {xlih, xwangc, wchang7}@gmu.edu

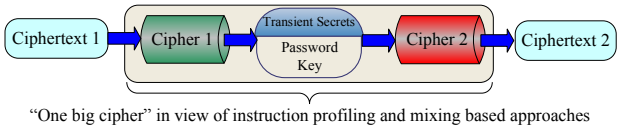


Fig. 1. Transient Cryptographic Secrets with Multiple Rounds of Cryptographic Operations

A number of methods [45], [43], [29] have been proposed to automatically recover the cryptographic keys from process memory or file. However, these methods require the cryptographic keys to be statically stored in plaintext form and they are not effective when the cryptographic keys are stored encrypted or transient.

Recently proposed binary analysis approaches (Reformat [49], Dispatcher [17], [27]) are able to automatically detect the existence of cryptographic operations from a given binary execution based on instruction profiling (i.e., calculate the percentage of bitwise and arithmetic instruction) and signature (e.g., specific constants and sequence of mnemonics) of particular cryptographic implementations. However, they are not effective in the presence of multiple rounds of cryptographic operations.

For example, sophisticated security software such as KeePass [6] (and KeePassX [7]) re-encrypts (with a random key generated at run-time) all the sensitive data (e.g., passwords, master keys) right after they have been decrypted and used at run-time. As illustrated in Figure 1, the plaintext form cryptographic secrets are transient in that they exist in memory for only one instant between cipher 1 and cipher 2 operations. KeePass has claimed [4] "even if you would dump the KeePass process memory to disk, you couldn't find the passwords."

In order to recover such transient cryptographic secrets, one needs to reliably identify not only exactly *where* but also exactly *when* those transient cryptographic secrets will be in memory. This requires one to accurately pinpoint the boundary of each of the multiple rounds of cryptographic operations.

Unfortunately, existing instruction profiling and signature based binary analysis approaches ([49], [17], [27]) tend to think multiple rounds of cryptographic operations (e.g., Cipher 1 and Cipher 2 in Figure 1) as one big cryptographic operation thus they are not able to recover the transient secrets in between multiple rounds of cryptographic operations. To the best of our knowledge, no existing binary analysis could accurately pinpoint the boundary between multiple rounds of cryptographic operations and recover truly transient cryptographic

secrets from the execution of a given binary executable.

In this paper, we present *CipherXRay* – a novel binary analysis framework that can accurately pinpoint the boundary of individual cryptographic operation from multiple rounds of cryptographic operations and recover truly transient secrets from the execution of a potentially obfuscated binary executable. Instead of using instruction profiling, we build CipherXRay upon one of the defining characteristics of all (good) cryptographic algorithms – the *avalanche effect*, which refers to the desired property of cryptographic function such that one bit change in the input or key would cause significant change in the output.

CipherXRay has the following nice features:

- It is able to reliably detect, pinpoint and distinguish the operations of public key cryptographic algorithms (e.g., RSA), block cipher (e.g., AES), and hash (e.g., SHA-1), even if they are mingled with each other or non-cryptographic operations, from the execution of a given binary executable.
- It can accurately pinpoint the location, size and boundary of the input, the output and the key buffers of each of the multiple rounds of cryptographic operations and determine the exact time when the input, the output, the IV and the key of each identified cryptographic operation will be ready. This allows us to recover the data input, the data output, the secret key (e.g., 256-bit AES key) or the private key (e.g., 1024-bit RSA key) used in every identified cryptographic operation even if multiple cryptographic operations are nested (e.g., encryption first and hash second) and the cryptographic secrets are transient.
- CipherXRay can further identify certain operation modes (e.g., ECB, CBC, CFB) of the identified block cipher and tell whether the identified cipher operation is encryption or decryption in certain cases.
- Since the binary code obfuscation (e.g., self-modifying code) does not change or remove the avalanche effect of any cryptographic operations in the original binary executable, CipherXRay could be effective on obfuscated binary executable as long as the avalanche effect can be detected.
- It is quite generic in that it is not dependent on specific implementation.

We have empirically evaluated the effectiveness of CipherXRay with OpenSSL, popular KeepassX password safe, malware Stuxnet, Kraken and Agobot, and a number of third party softwares with built-in chechsum and compression. Despite that KeePassX re-encrypts all the sensitive data within 21 microseconds after they have been decrypted and used at run-time, CipherXRay is able to recover not only all the protected entry passwords but also the 256-bit master key and the 128-bit IV that enable one to directly decrypt the KeePassX password file using OpenSSL; CipherXRay is also able to recover all block cipher secret keys from the binary executables obfuscated by a number of packers (e.g., UPX [10], ASPack, PECompact). CipherXRay has successfully recovered the secret key used by Agobot and the secrets of proprietary ciphers used by Kraken and Stuxnet malware.

To the best of our knowledge, CipherXRay is the first binary analysis framework that can accurately 1) pinpoint the boundary between multiple rounds of cryptographic operations; 2) recover truly transient cryptographic secrets and keys that exist in run-time memory for only a few micro seconds; and 3) recover the type of cryptographic operations and certain modes of operation of block ciphers. Our results demonstrate that the current software implementation of existing cryptographic algorithms achieves virtually no secrecy if their execution can be monitored.

The rest of this paper is organized as follows. Section II overviews CipherXRay. Section III describes how to detect the avalanche effect. Section IV discusses how to recover cryptographic secrets based on the avalanche effect. Section V presents how to recover the type, mode of operation of the identified cryptographic operation. Section VI presents the empirical evaluation of CipherXRay. Section VII discusses CipherXRay's implication, limitation and potential countermeasures. Section VIII overviews related works. Section IX concludes the paper.

II. Overview of CipherXRay

A. Goals and Assumptions

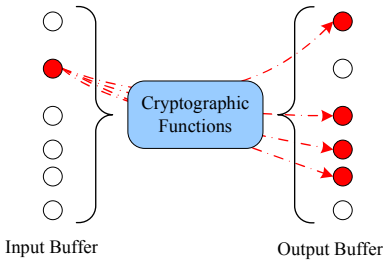
Given a potentially obfuscated binary executable, we want to uncover the cryptographic operations (e.g., encryption, decryption, hash) and their secrets from the execution. Specifically, we want to

- Determine if there is any cryptographic operation (e.g., encryption, decryption, hash) in its execution. If yes, we would like to pinpoint the location of all the cryptographic functions, their respective mode and the order of execution.
- Pinpoint the location, size and boundary of the input and the output buffers used by each cryptographic function identified.
- Determine exactly when the input and the output of each cryptographic function will be at which buffers. This enables us to recover those truly transient input and output of each cryptographic operation that will be immediately destroyed or re-encrypted after run-time use.
- Determine if there is any key used in each cryptographic operation. If yes, we would like to recover the key even if it will be destroyed right after run-time use.

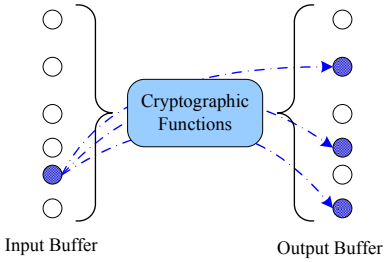
We assume that we can monitor the execution of the binary executable we are interested in. While the binary executable could be obfuscated, we assume the input and the output of any called cryptographic functions reside in some continuous memory buffer at run-time. The key, especially the private key, used in the cryptographic function could be stored in a transformed form, and it will be derived at run-time.

B. The Principle of CipherXRay

CipherXRay is designed upon the *avalanche effect*, which refers to the desirable property of all cryptographic algorithms (e.g., public key cryptographic algorithms, symmetric cryptographic algorithms, hash functions) such that a slight change (e.g., flipping a single bit) in the input would cause



(a) One bit change in the input



(b) Another bit change in the input

Fig. 2. The Avalanche Effect of Cryptographic Function

significant changes (e.g., half the output bits flip) in the output. Specifically, cryptographic functions are designed to exhibit the avalanche effect despite any obfuscation might be used in the implementation. On the other hand, non-cryptographic code almost never has the avalanche effect. Therefore, the avalanche effect is an fairly unique and defining characteristic of all good cryptographic functions. This enables us to reliably identify the cryptographic operations from potentially obfuscated executables.

Another nice feature of the avalanche effect is that it allows us to accurately pinpoint the location, size and boundary of both the input and output buffers. Figures 2(a) and 2(b) show the avalanche effect on the output buffer by changing two different bits in the input buffer. While changing different bits in the input buffer results different bits changed in the output buffer, the changed bits are cohesive within the fixed output buffer.

Let $m \geq 1$ and $n \geq 1$ be the number of bytes of the input and the output of the cryptographic function respectively. Because of the avalanche effect, any single bit change in the input would cause $4n$ bits changed in the output. Here we call those changed $4n$ bits in the output "touched" by the bit change in the input. Assume the $4n$ bits in the output touched by any bit in the input are random and independent from each other, then no more than $\frac{8n}{2^x}$ bits in the output would remain untouched after $x > 1$ different bit changes in the input. In other words, changing 8 different bits in the input of a good cryptographic function would leave no more than half bit untouched in a 128-bit output.

Given the location of buffers a and b, one can measure the impact of buffer a to buffer b by repeatedly changing different bits in buffer a and comparing the corresponding results in buffer b. This method is however not practical when the location of the input and the output buffers is unknown. Due to potential inherent randomness (e.g., using real-time clock value as seed), different runs of one executable may generate different internal states and outputs with exact the same input. Therefore, it is desirable to be able to measure the impact between any two chosen buffers with only one run of a given binary executable.

If any bit in memory impacts any other bit in memory during the binary execution, then there must exist information flow between those two bits. If we can track the information flow from a given source, we could measure the impact between memory buffers and detect the avalanche effect with a single run of the binary executable.

C. Overall CipherXRay Architecture

To handle potential dynamic binary transformations (e.g., packing) inside a binary executable, we build our CipherXRay framework upon *dynamic binary analysis* (DBA). On the other hand, we leverage results from static binary analysis to help dynamic binary analysis whenever possible. Since the cryptographic operations inside the binary executable is generally independent from the underlying operating system, we focus on analyzing the user space binary executables in this proof-of-concept research.[1].

Figure 3 shows the overall architecture of CipherXRay. CipherXRay dynamically intercepts and instruments the run-time instructions of the binary executable and collect valuable run-time information about the binary's execution. Specifically, CipherXRay tracks and records the taint propagation, the address and value of the bytes involved in the taint propagation. CipherXRay further analyzes the recorded run-time information and checks the patterns of instruction execution and memory access for any avalanche effect. Based on the detected avalanche effect patterns, CipherXRay identifies cryptographic operations and determines the exact location, size and boundary of the input buffer, the output buffer and any key buffer involved in each of the identified cryptographic operations and the exact time when the input, the output and the key (if any) will be in their corresponding buffers.

III. Identification of the Avalanche Effect

Assume we are able to effectively track the information flow across cryptographic functions, and we know the taint source (e.g., information received from the network), we want to identify if any part of the information flow from the taint source exhibits any avalanche effect. To the best of our knowledge, all existing cryptographic function implementations store the input, the output and the key of the cryptographic function in continuous memory buffers due to efficiency considerations. Therefore, we need ways to effectively and efficiently identify the avalanche effect between any two given continuous memory buffers of certain sizes.

[1]However, our binary analysis framework could be applied to kernel space binary executable if needed

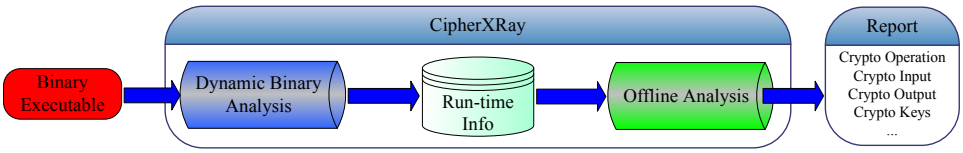


Fig. 3. Overall CipherXRay Architecture

A. Metrics for Detecting the Avalanche Effect

Given two continuous buffers $a[0,\dots m-1]$ of m bytes and $b[0,\dots n-1]$ of n bytes, if the information flow from any byte $a[i]$ ($i \in [0, m-1]$) touches one or more bytes in buffer $b[0,\dots n-1]$, we say $a[i]$ taints bytes in buffer $b[0,\dots n-1]$. If there exists the avalanche effect from buffer $a[0,\dots m-1]$ to buffer $b[0,\dots n-1]$, then any bit in buffer $a[0,\dots m-1]$ should taint about $4n$ random bits in buffer $b[0,\dots n-1]$. In addition, the bits in buffer $b[0,\dots n-1]$ tainted by each bit in buffer $a[0,\dots m-1]$ should be cohesive within buffer $b[0,\dots n-1]$. In other words, any byte $a[i]$ would leave no more than $\frac{8n}{2^8} = \frac{n}{32}$ bit untainted in buffer $b[0,\dots n-1]$. The probability that any byte $a[i]$ would leave one byte in buffer $b[0,\dots n-1]$ untainted is

$$\left(\frac{\binom{8n-8}{4n}}{\binom{8n}{4n}} \right)^8 < (\tfrac{1}{2})^{64} \quad (\text{for } n \geq 2)$$

Therefore, if there exists the avalanche effect from the m-byte buffer a to the n-byte buffer b, then *every byte* in buffer a would taint virtually *every byte* in buffer b.

To quantitatively measure the avalanche effect between two buffers, we use $C(a, m, b, n)$ to denote the *buffer a's m-th contribution rate to the n-byte buffer b*, which is defined as the portion of the first n bytes of buffer b that would be tainted by *every byte* from the first m bytes of buffer a. $C(a, m, b, n) \approx 100\%$ if and only if there exists the avalanche effect from the first m bytes of buffer a to the first n bytes of buffer b.

For example, Figure 4 shows that the m-byte buffer pointed by pointer $a2$ has avalanche effect over the n-byte buffer pointed by pointer $b2$. Therefore, $C(a2, m, b2, n)$ will be about 100%. Because the bytes between pointers $a1$ and $a2$ do not have avalanche effect to buffers $b1[0,\dots,n-1]$ and $b2[0,\dots,n-1]$, both $C(a1, m, b1, n)$ and $C(a1, m, b2, n)$ will be close to zero. On the other hand, $C(a2, m, b1, n) \approx \frac{n-(b2-b1)}{n}$ $(n \geq b2-b1)$ as buffer $a2[0,\dots m-1]$ does not have avalanche effect over memory region $b1[0,\dots b2-b1+1]$ but over memory region $b2[0,\dots n-(b2-b1)+1]$.

Therefore, if there exists avalanche effect from buffer $a[0,\dots m-1]$ to buffer $b[0,\dots n-1]$, we can eventually find the right values of a, m, b, and n that make $C(a, m, b, n) \approx 100\%$ by trying different values of a, m, b, and n. This allows us to accurately pinpoint the location, size and boundary of both the input and the output buffers of cryptographic functions.

Note, when searching for avalanche effect, we only need to consider those bytes that have taint relationship and aggregate them together. This allows CipherXRay to complete the search efficiently in polynomial time.

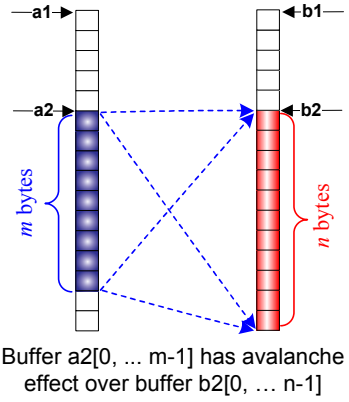


Fig. 4. Taint Contribution Rate between Buffers

B. How to Identify the Avalanche Effect When the Input is Unknown?

One challenge in identifying the avalanche effect is that the location and the size of the input buffer of the cryptographic operation is unknown. To pinpoint the input buffer, we taint any memory region and the input sources (e.g., file input and network input) which could potentially contain the input buffer of the cryptographic operation and monitor the execution of the binary. We need to periodically check the monitored execution for the avalanche effect. Since most software implementations of cryptographic algorithms consist of multiple meaningful routines, it is natural to check at the granularity of routine. Alternatively, we can check after every $x > 0$ instructions have been executed or a buffer has been updated. This approach would be more effective on those software implementations that have very few big routines or have lot of function inlinings. Due to space consideration, we only present how we periodically check at the granularity of routines.

We view the execution history as a sequence of routine invocations ordered by their completion time. Let $f_1, f_2, ..., f_n$ be the sequence of routine invocations where f_{i+1} completes right after f_i for any $i \geq 1$. For each routine invocation f_i, we construct a set B_i which contains those tainted buffers that have been updated in f_i or are still alive when f_i completes. Note, the run-time information collected by CipherXRay contains the snapshot of the value of tainted buffers, therefore, CipherXRay works even if the cryptographic input or key have been destroyed right after using inside any f_i. For each B_i, we take every continuous buffer a in set B_i as the potential input buffer, and every continuous buffer b in sets $B_{i+1}, B_{i+2}, ..., B_n$ as the potential output buffer

```
1: for each f_i do
2:    j ← 0
3:    k ← 8
4:    if j + k ≤ m then
5:       calculate S
6:       if S is empty then
7:          continue
8:       else
9:          for each k-byte buffer x in S do
10:            if a_{j+k} doesn't taint every byte in x then
11:               mark buffer a[j, j + k) as a block
12:               j ← j + k
13:               goto 4
14:            end if
15:         end for
16:         k ← k + 1
17:         goto 4
18:      end if
19:   end if
20:   if at least one block is identified then
21:      break
22:   end if
23: end for
```

Fig. 5. Recovering Block Ciphers

respectively. By applying the metrics defined above, we can detect the avalanche effect between any two continuous buffers a and b. Due to the nature of the avalanche effect, we are able to correlate the input buffer and output buffer of the cryptographic operation if there is any, and discover the exact range of the input buffer and output buffer even if part of buffers a and b is not involved in the cryptographic operation. The algorithm to pinpoint the input and output buffers is further detailed in section IV-A.

IV. Recovering the Secrets of Identified Cryptographic Operations

A. Recovering the Input and the Output of the Identified Cryptographic Operation

Given a continuous buffer a of m bytes, we use the following algorithms to detect whether buffer a contains the input of a cryptographic operation and if yes, further discover boundary of the input and output buffers.

Figure 5 shows the high level algorithm for identifying and recovering the input and the output of block ciphers. Specifically, we examine the avalanche effect at each routine completion f_i. Every time we examine a small portion of buffer a from offset j with a length of k. We further define the set S as the intersection of buffers which are 1) no less than k bytes, 2) tainted by bytes $a_j, a_{j+1}, ..., a_{j+k-1}$ at f_i. The algorithm stops either when there is a block cipher cryptographic operation identified and all of its corresponding input and output blocks are recovered, or when all routine invocations are checked and there is no block cipher identified.

Since hash function exhibits different dataflow patterns than block cipher, the algorithm for hash function, shown in Figure 6, is slightly different from that for block ciphers. Recognizing that hash functions usually calculate the hash of a long buffer piece by piece, we repeatedly check all routine invocations. If there are more bytes which have 100% contribution to the hash buffer already identified, we aggregate them into the input buffer. Once the potential hash buffer is no longer alive or we have checked all routine invocations, we consider this buffer as the final hash buffer.

```
1: j ← 0
2: k ← 8
3: if j + k ≤ m then
4:    calculate S
5:    if S is empty then
6:       j ← j + 1
7:       goto 3
8:    else
9:       for every buffer b in S do
10:         for l = j + k → m − 1 do
11:            if NOT a_l taints every byte in b then
12:               break
13:            end if
14:         end for
15:         mark buffer b as the hash of buffer a[j, l)
16:      end for
17:   end if
18: end if
```

Fig. 6. Recovering Hash

Public key cryptographic operations can be handled by the same algorithm for hash function detection.

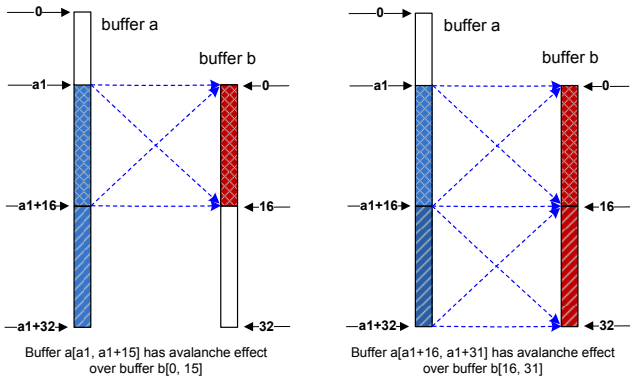
Once we have pinpointed the location, size and the boundary of the input and output buffers of the identified cryptographic operation, we can easily recover the content of the input from the snapshot of the identified input buffer right before it is consumed by the cryptographic operation and the content of the output from the snapshot of the identified output buffer right after the cryptographic operation has finalized the output. Note, our algorithms can recover the input and the output even if the generated output overwrites the input buffer.

As a concrete example, suppose we have a hypothetical general block cipher `XYZ` whose block size is 16 bytes. Figure 7 shows the sketch implementation of `XYZ`'s encryption in ECB chaining mode. The blockwise encryption is done in function `XYZ_Encrypt_Block()`. Assume we use the `XYZ` block cipher to encrypt a 32-byte plaintext buffer which is part of a larger buffer a in which the first several bytes are not the input to `XYZ_Enc_ECB()`, and the corresponding ciphertext is stored in buffer b. For the sake of simplicity, the plaintext has exactly two blocks, so `XYZ_Encrypt_Block()` is invoked twice. Let I_1 and I_2 be the two invocations' completion points in the execution history.

The avalanche effect observed during the execution of `XYZ` block cipher encryption is shown in Figure 8. At I_1, as depicted in Figure 8(a), when we apply algorithm in Figure 5, no taint propagation to any part of buffer b is detected until the byte at offset $a1$ is being examined. Since at I_1, the encryption of the first block $a[a1, a1 + 15]$ completes,

```
void XYZ_Enc_ECB(char *input, char *output, KEY *key) {
  initialization;
  for each block input_block in input
  { pre-block handling;
    XYZ_Encrypt_Block(input_block, output_block, key);
    post-block handling;
  }
}
```

Fig. 7. Sketch Implementation of *XYZ*'s Encryption in ECB Mode



(a) Taint Propagation at I_1 (b) Taint Propagation at I_2

Fig. 8. Avalanche Effect of *XYZ* Block Cipher at I_1 and I_2

the avalanche effect from $a[a1, a1 + 15]$ to $b[0, 15]$ can be detected. Further more, $a[a1{+}16, a1{+}31]$ doesn't taint any part of buffer b, so the algorithm marks buffer $a[a1, a1 + 15]$ as a block.

The algorithm then continues at I_2. In Figure 8(b), after the second block of the plaintext is encrypted, a new instance of avalanche effect from the plaintext to the ciphertext is added. When we apply algorithm in Figure 5, in addition to the previously detected block $a[a1, a1 + 15]$, the new avalanche effect from buffer $a[a1{+}16, a1{+}31]$ to buffer $b[16, 31]$ can be detected and buffer $a[a1{+}16, a1{+}31]$ and buffer $b[16, 31]$ are added as the new input block and output block respectively.

As a result, the algorithm successfully identifies the two-block plaintext buffer and the corresponding ciphertext buffer. The information regarding the two buffers is preserved for further processing to identify the mode and type of the block cipher operation.

B. Recovering the Secret Key and the Private Key

The security of cryptography is based on the secrecy of the key used in the cryptographic operation. In any pure software implementation of cryptographic algorithms, the key to be used must be somewhere in the memory at certain time no matter how the key is derived or obtained. Once we have identified some cryptographic operation, we want to know if there is any key involved. If yes, we want to identify when and where the key will be and recover it.

The key of any cryptographic algorithm must exhibit the avalanche effect on the data output. However, there are other sources in the memory that could have the avalanche effect on the data output:

- The data input to some cryptographic algorithms (e.g., block cipher, hash).
- The initialization vector used in the cryptographic operation.
- Internal buffers used in the cryptographic operation.
- Some static (or global) data (e.g., S-Box in AES) used in the cryptographic operation.
- Intermediate results in key derivation.

In order to reliably recover the key, we need to distinguish the key from above listed "noise" sources. Fortunately, we are able to do so based on the avalanche effect pattern and data liveness analysis:

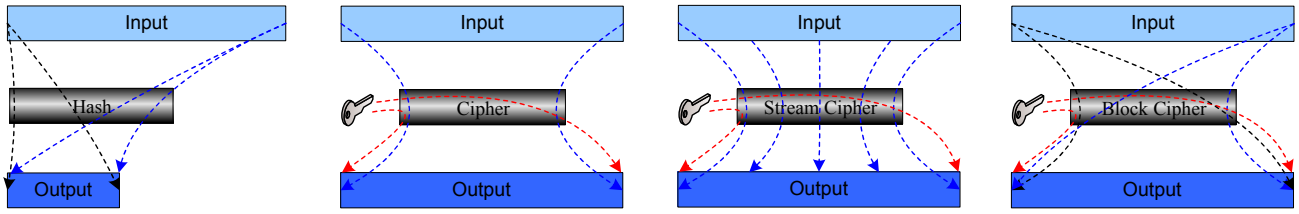
- Distinguish the key from the data input: For a block cipher with multiple blocks of data input, the data input to the block cipher will be different for different block. On the other hand, the same key will be used for each block.
- Distinguish the key from the initialization vector: For a block cipher with multiple blocks of data input, the initialization vector will be used for the first block only, while the key will be used for all blocks.
- Distinguish the key from the internal buffers used in the cryptographic operation: While the life span of the internal buffers in a cryptographic operation is no more than the life span of the cryptographic operation, the life span of the key buffer should be more than the life span of the whose cryptographic operation.
- Distinguish the key from the static (or global) data used in the cryptographic operation: Every byte of the key would impact the whole cryptographic output. On the other hand, any byte in the static data (e.g., S-Box in AES) used in the cryptographic operation usually impact only small portion of the cryptographic output.
- Distinguish the key from intermediate results in key derivation: For efficiency consideration, usually the key will be derived once even if it will be used many time. Even if the key will be derived (repeatedly) every time before the key will be used, we can always choose the buffer that has been updated most recently and yet its life span is longer than the cipher operation. In other words, we choose the candidate key buffer that is "closest" in time to the start of the cipher operation.

V. Identification of the Type of Cryptographic Operations

Based on the avalanche effect, we can detect the existence of cryptographic operations in the binary executable. We want to further determine the type of the cryptographic operations and their internals. For example, is the detected cryptographic operation a hash or a cipher? If it is a cipher, is it a block cipher or a stream cipher? If it is a block cipher, what operation mode it is using? Is the cipher operation encryption or decryption?

A. Distinguishing Different Types of Cryptographic Operation

While all cryptographic operations exhibit the avalanche effect one way or the other, the patterns of the exhibited avalanche effect are different for different types of cryptographic operations. This gives us a way to differentiate different types of cryptographic operations.



(a) Hash Function (b) Cipher Function (c) Stream Cipher (d) Block Cipher

Fig. 9. Different Patterns of the Avalanche Effect between a Cryptographic Hash, a Stream Cipher and a Block Cipher Functions

1) Distinguishing a Hash from a Cipher: One of the fundamental difference between a hash function and a cipher is that the output size of a Cryptographic hash function is fixed while the output size of a cipher corresponds to the input size. [2] Therefore, we can tell the identified cryptographic operation is not a cipher if the output is smaller than the input involved in the avalanche effect. If the identified cryptographic operation generates the output of fixed size with different input of different sizes, the cryptographic operation is likely a hash.

Due to the avalanche effect, any byte in the hash input would impact almost all bytes in the hash output. If the identified cryptographic operation does not exhibit this pattern, it is not a hash. Figures 9(a) and 9(b) illustrate the different patterns of the avalanche effect between a hash function and a cipher.

2) Distinguishing a Stream Cipher from a Block Cipher: A stream cipher generates the ciphertext by combining (e.g., XOR) the cleartext with the pseudorandom bitstream generated from using the secret key. Therefore, any byte of the stream cipher input would impact only one byte of the stream cipher output of corresponding offset, and any byte in the key would impact the whole pseudorandom bitstream. In other words, the data input (i.e., cleartext) of a stream cipher has no avalanche effect on its data output (i.e., ciphertext) while its key input does. On the other hand, both the data input and the key input of a block cipher have the avalanche effect on its data output. This enables us to distinguish a stream cipher from a block cipher by analyzing the information flow patterns. Note, certain operation mode (e.g., Integer Counter Mode) could turn a block cipher into a stream cipher. The block cipher in such operation mode will be considered a stream cipher. Figures 9(c) and 9(d) show the different patterns of the avalanche effect of a stream cipher and a block cipher. Due to space limitation, we are not able to discuss the details about how to detect stream ciphers in this paper.

B. Detect Block Cipher Modes of Operation

When the input to a block cipher is longer than its block size, the input needs to be partitioned into multiple blocks before it can be processed by the block cipher. The block cipher mode of operation determines how the block cipher operation will be applied to multiple blocks. Different mode of operation exhibits different pattern of the avalanche effect between the input blocks and the output blocks. This gives us a way to detect and differentiate the block cipher modes of operation.

[2]The output size of a cipher may be slightly bigger than the input size due to padding

Mode of Operation	Encryption	Decryption
ECB	$\langle 1:n \rangle$	$\langle 1:n \rangle$
CBC	$\langle 1:n,\ 1:n,\ 1:n,\ldots \rangle$	$\langle 1:n,\ 1:1 \rangle$
CFB	$\langle 1:1,\ 1:n,\ 1:n,\ldots \rangle$	$\langle 1:1,\ 1:n \rangle$
OFB	$\langle 1:1 \rangle$	$\langle 1:1 \rangle$

TABLE I
PATTERNS OF THE AVALANCHE EFFECT OF DIFFERENT MODES OF OPERATION

Given input block $X[0,\ldots,n-1]$ and output block $Y[0,\ldots,n-1]$, the pattern of the avalanche effect between them could be:

- $1:1$ every byte $X[i]$ $(i \in [0,n-1])$ from the input block $X[0,\ldots,n-1]$ has 100% contribution rate to only byte $Y[i]$ in the output block $Y[0,\ldots,n-1]$.
- $1:n$ every byte $X[i]$ $(i \in [0,n-1])$ from the input block $X[0,\ldots,n-1]$ has 100% contribution rate to every byte $Y[j]$ $(j \in [0,n-1])$ in the output block $Y[0,\ldots,n-1]$.

Table I shows the patterns of the avalanche effect between the input blocks and output blocks under different modes of operation.

The encryption and decryption in ECB (Electronic Codebook) mode have exact the same pattern of the avalanche effect between the input blocks and output blocks: $\langle 1:n \rangle$. Specifically, every byte in a given input block impacts every byte in the corresponding output block and it has no impact on any other output block.

The pattern of the avalanche effect in CBC (Cipher Block Chaining) mode encryption is $\langle 1:n,\ 1:n,\ 1:n,\ldots \rangle$ in that every byte in a given input block impacts every byte in the corresponding output block and all the subsequent output blocks. The pattern of the avalanche effect in CBC (Cipher Block Chaining) mode decryption is $\langle 1:n,\ 1:1 \rangle$ in that every byte in a given input block impacts every byte in the corresponding output block and only one byte in the subsequent block.

The pattern of the avalanche effect in CFB (Cipher Feedback) mode encryption is $\langle 1:1,\ 1:n,\ 1:n,\ldots \rangle$ in that every byte in a given input block impacts only one byte in the corresponding output block and every byte in all subsequent output blocks. The pattern of the avalanche effect in CFB (Cipher Feedback) mode decryption is $\langle 1:1,\ 1:n \rangle$ in that every byte in a given input block impacts only one byte in the corresponding output block and every byte in the subsequent

output block.

The encryption and decryption in OFB (Output Feedback) mode have exact the same pattern of the avalanche effect between the input blocks and output blocks: $\langle 1 : 1 \rangle$. Specifically, every byte in a given input block impacts only one byte in the corresponding output block and it has no impact on any other output block.

Therefore, we can reliably distinguish between ECB, CBC, CFB and OFB modes of operation based on the patterns of the avalanche effect as long as the block cipher input (and output) is no less than three blocks. Specifically, we make no distinction between block cipher in OFB (and CTR) mode and stream cipher since OFB (and CTR) mode does turn a block cipher into a stream cipher.

C. Distinguishing between Encryption and Decryption of Block Cipher

As shown in Table I, the encryption and the decryption in CBC and CFB modes have different patterns of the avalanche effect. Therefore, we can determine whether a block cipher is encryption or decryption if it is in CBC or CFB mode.

For block cipher in ECB mode, the encryption and the decryption have exactly the same pattern of the avalanche effect: $\langle 1 : n \rangle$. Nevertheless, we can distinguish the encryption from the decryption in ECB mode as long as there exists any padding in the last block of the plaintext. Because the padding is generated from the encryption code, it has different source of information flow than the "real" plaintext. The last block of the input to the block cipher encryption is the padded plaintext, and the last block of the input to the block cipher decryption is the ciphertext. Therefore, if we taint the source of the input to the block cipher, only part of the last block of the input (the plaintext) to the block cipher encryption will be tainted (the padding part will not be tainted), and the whole last block of the input (the ciphertext) to the block cipher decryption will be tainted. This allows us to distinguish the encryption from the decryption for block ciphers in ECB mode.

VI. Empirical Evaluation

We have implemented a proof-of-concept prototype of CipherXRay upon the dynamic instrumentation framework Valgrind [41] in Linux[3]. Specifically, we have built our dynamic information analysis upon Flayer tool [22]. Besides standard data flow tracking, we have implemented preliminary support of tracking conditional control dependency by analyzing the control flow and data flow together. Due to inherent limitation of pure dynamic analysis, our current CipherXRay prototype does not support analyzing implicit data flow.

We have evaluated CipherXRay with OpenSSL [9] crypto library, Common-Off-The-Shelf KeePassX password manager, encrypted C&C communication of Agobot, proprietary ciphers used by malware Kraken and Stuxnet, a number of third party programs that use various checksum and compression algorithms. We have also conducted preliminary experiments with several packers. Note, although the source code of many of the programs (e.g., OpenSSL, KeePassX) and malware (e.g., Agobot) used in our experiments were available, CipherXRay used the binary executables only in all the experiments. The source code was only used as the "ground truth" for verifying the results by CipherXRay. The Windows packers we experimented were in the binary executable form only.

All the experiments have been conducted on a PC with an 3.07GHz Intel Core i7 CPU and 4GB RAM running Linux kernel 2.6.32.

[3]CipherXRay is not bound to any operating system, and it can be ported to any operating system and even VMM or hypervisor

A. Combination of Multiple Rounds of Block Cipher Operations and Hash

To evaluate CipherXRay's capability in detecting cryptographic operations and revealing the internals of the cryptographic operations, we designed `test program 1` which reads the plaintext from a disk file and processes the plaintext with multiple rounds of block cipher encryption and decryption followed by a final round of hash in the following order:

1) Blowfish Encryption in CBC mode
2) AES-256 Encryption in CFB mode
3) AES-256 Decryption in CFB mode
4) Blowfish Decryption in CBC mode
5) SHA-1 Hash

The two block ciphers, Blowfish and AES-256, use different secret keys. In this experiment, CipherXRay should be able to distinguish a block cipher from a hash function. For block ciphers, CipherXRay should further identify the block size, the operation mode, the chaining mode, the secret key, the size and the location of the corresponding input buffer and output buffer; for hash functions, CipherXRay should be able to tell the size and the location of the input buffer and output hash buffer.

To demonstrate CipherXRay's accuracy in pinpointing the boundary of the input buffer of a block cipher, we intentionally skip the first 100-byte of the plaintext read from a file and feed the remaining as the input to the Blowfish block cipher in step 1). The subsequent steps take the output of the previous step as the input.

Table II shows the information (e.g., cryptographic function , the address and size of the input buffer, the output buffer and the secret key if there is one) about each block cipher and the SHA-1 hash collected by `test program 1`, which will be used as the "ground truth" in validating the CipherXRay's analysis results.

Table III shows the analysis output by CipherXRay. It has similar structure to Table II with two differences: Table III doesn't show the block cipher algorithm because CipherXRay is unable to identify it; Table III has also two extra columns regarding the runtime execution information of the block cipher operation:

1) The entry point address of the routine who is the first to see the complete output buffer of the cryptographic operation at its execution end.
2) The return EIP address.

By comparing the two tables, we can observe that CipherXRay successfully identified the chaining mode, the operation type, the address and size of the input buffer, the

Crypto Operation	Input Buffer		Output Buffer		Block Size	Key	
	Address	Size	Address	Size		Address	Size
Blowfish CBC Encryption	804b780	867	804afa0	872	8	804a740	16
AES-256 CFB Encryption	804afa0	872	804a7c0	872	16	804a760	32
AES-256 CFB Decryption	804a7c0	872	804c740	872	16	804a760	32
Blowfish CBC Decryption	804c740	872	804bf60	867	8	804a740	16
SHA-1	804bf60	867	be9236bc	20	N/A	N/A	N/A

TABLE II
BLOCK CIPHER INFORMATION FROM TEST PROGRAM 1

Crypto Operation	Input Buffer		Output Buffer		Block Size	Key		Runtime Environment	
	Address	Size	Address	Size		Address	Size	Routine	Return EIP
CBC Encryption	804b780	867	804afa0	872	8	804a740	16	4099bc0	40e4463
CFB Encryption	804afa0	872	804a7c0	872	16	804a760	32	4094c60	40e533a
CFB Decryption	804a7c0	872	804c740	872	16	804a760	32	4094c60	40e533a
CBC Decryption	804c740	872	804bf60	872	8	804a740	16	4099bc0	40e4463
Hash	804bf60	867	be9236bc	20	N/A	N/A	N/A	407f470	407fb2a

TABLE III
BLOCK CIPHER OPERATIONS IDENTIFIED BY CIPHERXRAY

output buffer and the secret key of the first three block cipher operations. CipherXRay also successfully identified the hash operation which follows the last round of block cipher operation. For the last block cipher operation, i.e., Blowfish CBC decryption, the size of the output buffer identified by CipherXRay is 872 bytes while the actual size reported by `test program 1` is 867 bytes. The size difference is caused by the padding of CBC mode. In CBC mode decryption, the padding bytes will be stripped off by the block cipher and the application only sees the stripped buffer. However, CipherXRay tracks the decryption process according to the avalanche effect and stops once the complete output buffer is seen. As a result, the output buffer of a CBC mode block cipher decryption operation identified by CipherXRay will be the unstripped buffer. Thus, the size difference is reasonable.

To further prove CipherXRay's accuracy, we examined the runtime environment information for each step. By checking OpenSSL's crypto library's symbol table, we found the routine symbol names which correspond to the runtime environment addresses listed in Table III. The routine symbol names are shown in the table below.

Address	Routine Name
4099bc0	BF_cbc_encrypt()
40e4463	bf_cbc_cipher()
4094c60	AES_cfb128_encrypt()
40e533a	aes_256_cfb128_cipher()
407f470	SHA1_Final()
407fb2a	SHA1()

From the two tables, we can learn that for step 1 and step 4, the complete output buffer of the block cipher operation is first seen when an invocation of `BF_cbc_encrypt()` finishes. Additionally, this instance of routine `BF_cbc_encrypt()` is invoked by `bf_cbc_cipher()`. For step 2 and step 3, the complete output buffer of the block cipher operation is first seen when an invocation of `AES_cfb128_encrypt()` finishes. Additionally, this instance of routine `AES_cfb128_encrypt()` is invoked by `aes_256_cfb128_cipher()`. For step 5, the complete output buffer of the SHA-1 hash is first seen when an invocation of `SHA1_Final()` finished and this instance of routine `SHA1_Final()` is invoked by `SHA1()`. The runtime environment information reported by CipherXRay matches the ground trueth, which validates the accuracy of CipherXRay's analysis result.

It took CipherXRay about 8 minutes to complete the analysis in this experiment.

B. RSA Encryption and Decryption

We have chosen to use RSA in our experiment on the public key cryptography analysis. We have used two programs in the experiment: one to encrypt a short message using RSA public key and the other to decrypt the ciphertext using the corresponding RSA private key. We have used 1024-bit RSA key in the experiment, and the short message is less than 1024 bits. Our primary goal is to recover the RSA private key, whose format is defined in PKCS#1 [30] as shown in Figure 10.

By analyzing the execution log of the RSA public key encryption program, CipherXRay has been able to identify both the modulus n and the public exponent e of the public key.

Table IV shows the RSA private key fields identified by CipherXRay from analyzing the execution log of the RSA private key decryption program. It shows that CipherXRay has successfully identified the length field and value field of modulus n, public exponent e, prime1 p, prime2 q, exponent1 $d \mod (p-1)$, exponent2 $d \mod (q-1)$ and coefficient $q^{-1} \mod p$. Besides, the first byte of the key, a type field, the length field of the version and the length field of the private exponent are also identified. However, the value field of the private exponent d has not been identified.

Further investigation shows that the RSA implementation in OpenSSL uses the Chinese Remainder Theorem to calculate the modulo exponentiation which does not use the private exponent d at all. Therefore, CipherXRay does not see any avalanche effect from the private exponent d. This further proves the accuracy of CipherXRay. Having two prime numbers p and q identified, it is trivial to derive the private

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Fig. 10. RSAPrivateKey Format

Address Range	Offset Range	Size	Field
04186028 - 04186028	0 - 0	1	
0418602d - 0418602d	5 - 5	1	
04186030 - 04186031	8 - 9	2	
04186033 - 041860b2	11 - 138	128	n
041860b4 - 041860b7	140 - 143	4	e
041860b9 - 041860bc	145 - 146	2	
0418613d - 0418613d	277 - 277	1	
0418613f - 0418617e	279 - 342	64	p
04186180 - 04186180	344 - 344	1	
04186182 - 041861c1	346 - 409	64	q
041861c3 - 04186203	411 - 475	65	$d \bmod (p-1)$
04186205 - 04186205	477 - 477	1	
04186207 - 04186246	479 - 542	64	$d \bmod (q-1)$
04186248 - 04186248	544 - 544	1	
0418624a - 04186289	546 - 609	64	$q^{-1} \bmod p$

TABLE IV
RSA PRIVATE KEY FIELDS IDENTIFIED BY CIPHERXRAY

exponent d from p, q, e, n. Therefore, CipherXRay has successfully recovered the RSA private key.

In the public key recovery experiment, it took CipherXRay about 10 minutes to recover RSA public key with a 17-bit e, about 40 minutes to to recover the 1024-bit RSA private key.

C. KeePassX

To validate CipherXRay's capability in recovering the cryptographic secrets and keys from Common-Off-The-Shelf security applications, we chose KeePassX 0.4.3 [7], a cross-platform clone of KeePass Password Safe [6]. Besides using strong encryption (e.g., AES-256) to protect the password database, KeePass and KeePassX keep the hash of the master key and all the entry passwords encrypted in the memory at run-time. Specifically, within 21 microseconds after the entry passwords have been read and decrypted from the password database, KeePassX re-encrypts all entry passwords in the memory with a random key generated at run-time. By keeping the cryptographic secrets encrypted in the memory, KeePass [4] claimed that "even if you would dump the KeePass process memory to disk, you couldn't find the passwords."

In order to recover the entry passwords and the master key from KeePass or KeePassX, we have to determine exactly when and where the entry passwords and the master key will be briefly unencrypted in the memory.

We used KeePassX to create a 1468-byte encrypted password database of four entries. After marking the encrypted password database as tainted, we used CipherXRay to monitor the execution of KeePassX. The execution log of KeePassX shows that the tainted data was used by a lot of non-cryptographic functions in addition to cryptographic functions. Nevertheless, CipherXRay successfully identified that there is a 128-bit block cipher decryption in CBC mode of operation on the data read from the database file. CipherXRay further identified the 256-bit secret key that was derived from the master password. Based on these information, CipherXRay determined that the IV should be 16 bytes and have $\langle 1:1 \rangle$ pattern on the first block of the plaintext. It further identified the 16-byte IV that corresponds to the data read from the encrypted password database from offset 32 to 47. CipherXRay also successfully recovered the four entry passwords in plaintext form even though KeePassX re-encrypts them immediately after they have been derived in the memory.

To verify the correctness of the recovered master key and IV, we used OpenSSL command line tool to open the KeePassX password file with the recovered 256-bit master key and 128-bit IV. We were able to obtain all the four plaintext password entries.

It took CipherXRay about 35 minutes to recover the 256-bit master key and all the entry passwords protected by KeePassX.

D. Malware Agobot, Kraken and Stuxnet

To validate CipherXRay's capability in front of real world malware, we have experimented with Agobot, Kraken and Stuxnet. Agobot has built-in support of SSL 3.0 to encrypt its C&C traffic. We set up one Agobot node that communicates with the Agobot server, and used CipherXRay to monitor and analyze the execution of Agobot node. The Agobot negotiated to use AES-256 CBC mode as its cipher in the TLS session with the Agotbot server. It took CipherXRay about 15 minutes to detect the first AES-256 CBC decryption and the 240-byte Rijndael key schedule that has the avalanche effect on the decrypted plaintext output inside Agobot. The first 32 bytes of the key schedule are the dynamically generated 256-bit AES key used by Agobot. Therefore, CipherXRay automatically recovered not only the decrypted plaintext C&C communication message of Agonot, but also the secret key dynamically generated by Agobot at run-time.

Both Kraken and Stuxnet used proprietary ciphers to protect its C&C communication and files. It is desirable to see if CipherXRay could detect such proprietary ciphers that are not cryptographically strong.

Since the C&C server our Kraken sample tried to connect to was no longer alive, we could not monitor the live C&C traffic from the botmaster. Instead we used the C-reimplementation of the reverse-engineered Kraken encryption/decryption algorithms [34] to encrypt a short message. CipherXRay was able to to detect the avalanche effect from the first three 4-byte units (key1, key2 and seed) of the 76-byte input to the 64-byte output buffer. Specifically, every byte in key1, key2 and seed taints all bytes of the 64-byte output. When the Kraken encryption runs in block-wise mode, each 8-byte block in the input has the avalanche effect on the corresponding 8-byte block in the output, which is the typical pattern of 64-bit block cipher in ECB mode.

Stuxnet uses a proprietary stream cipher to protect its 1860-byte configuration file mdmcpq3.pnf. We experimented with

the reverse engineered stuxnet decryption algorithm [39] to decrypt a 1860-byte input. CipherXRay successfully detected the avalanche effect from the key to the 1860-byte output. This allows us to automatically recover the key and the output of Stuxnet's proprietary stream cipher.

In summary, CipherXRay can detect the avalanche effect from not only standard cryptographic algorithms (e.g., AES) but also the proprietary ciphers of Kraken and Stuxnet, and automatically figure out the key, the output and the block size from real world malwares.

E. Checksum and Compression

Checksum and compression are closely related to cryptographic operations. We have experimented with a few third party software that have built-in checksum and a popular compression utility.

Open-vcdiff [1] is a encoder and decoder program for the VCDIFF generic differencing and compression data format, and it uses Adler32 checksum [48]. cksfv [3] uses CRC32 to create simple file verification listings and test existing sfv files.

We used CipherXRay to analyze the executions of Open-vcdiff and cksfv. Interestingly, CipherXRay detected the avalanche effect from the execution of cksfv. No avalanche effect was detected from Open-vcdiff. After further investigation, we have confirmed that CRC32 checksum does have the avalanche effect while Alder32 checksum does not. Note, although CRC32 is not a cryptographic hash function, it is indeed a hash function that maps a large input to a fixed size output. Therefore, CipherXRay did perform correctly with programs that have built-checksum. It also confirmed that VCDIFF compression does not have the avalanche effect.

We further experimented with popular bzip2 [2] compression program. CipherXRay detected one instance of hash-like avalanche effect to a buffer of 8 bytes. No cipher-like avalanche effect was found in the compression algorithms used in bzip2. We manually checked the source code of bzip2, and we found out that bzip2 stores CSC32 checksums in two adjacent 32-bit fields (blockCRC and combinedCRC) in the "structure holding all the compression-side stuff." The two adjacent CRC32 fields together exhibit the hash-like avalanche effect to a buffer of 8 bytes.

In summary, CipherXRay did not detect any avalanche effect from any program tested that does not have cryptographic or CRC32 operations. Since CRC32 is indeed a hash function (not cryptographically strong though) that exhibits the avalanche effect, CipherXRay did perform correctly with programs using CRC32 checksum.

F. Obfuscated Cryptographic Binary

It is desirable to see how CipherXRay performs in the presence of binary obfuscation. Since Valgrind does not support Windows application directly, we used Wine [8] to run our obfuscated Windows programs on Linux. This allows CipherXRay to monitor the instruction execution of Windows binary executables.

We used the Windows binary packers ASPack, PECompact and a cross-platform packer UPX [10] to pack a binary test program that encrypts the plaintext read from a disk file with AES-256 block cipher in CBC mode. ASPack and PECompact were applied to the Windows version of the test program, and UPX was applied to the Linux version of the test program.

In all cases, CipherXRay was able to recover the same cryptographic secrets from the packed test program as that of the unpacked version. CipherXRay has successfully identified the block cipher's operation mode, operation type (whether it is encryption or decryption), the location, size and boundary of the input buffer, output buffer and secret key of the AES-256 CBC mode encryption operation. The success with the Windows version of the packed test program confirms that CipherXRay is independent from the underlying operating systems.

We further packed the Stuxnet decryption test program with UPX. CipherXRay was able to detect the same avalanche effect and recover the key and the cryptographic output as the unpacked version.

Note, binary obfuscation such as packing would defeat static binary analysis. Fortunately, most analysis of our CipherXRay prototype is dynamic binary analysis, which enables CipherXRay to be effective on certain obfuscated binary executables.

In this experiment, it took CipherXRay less than two minutes recover all the block cipher secrets from the packed binary executables without using any knowledge of the packers.

VII. Discussion

Current implementation of CipherXRay assumes all the input, output and cryptographic keys are kept in continuous memory buffers. To the best of our knowledge, all widely used software implementations (e.g., OpenSSL) of cryptographic algorithms follow this convention due to performance consideration. In addition, all the malwares we have experimented have kept the cryptographic secrets in continuous buffer as well.

In principle, deliberate obfuscation (e.g., putting the cryptographic input, output and key in discontinuous buffers) does not remove the avalanche effect among the cryptographic input, key and output, but makes it harder to detect the avalanche effect.

Since CipherXRay is based on information flow analysis, it is subject to any attack on information flow analysis [19] such as obfuscation [46], [44], [37]. White box cryptography [50], [11] aims to protect the software implementations of cryptographic primitives against attacks from the execution environment. It uses all kinds of obfuscation techniques to hide the secret or private key in cryptographic software implementation. Theoretically program obfuscation has been shown to have some fundamental limitation [13], [47]. How much program obfuscation could negatively impact CipherXRay is an open problem and needs further investigation.

Note, the hash function (e.g., CRC32) and the cipher (e.g., proprietary cipher of Stuxnet) detected by CipherXRay are not necessarily cryptographically strong.

As shown in the Stuxnet experiment in section VI-D, current CipherXRay prototype is able to detect and recover the key and

the output of stream ciphers. Automatic detection of stream cipher input was partially implemented. It is a future work to complete the implementation of stream cipher support in CipherXRay.

Intel AES instruction set [5] aims to eliminate the major timing and cache-based attacks that threaten table-based software implementations of AES. While we do not expect the AES instruction set would eliminate the avalanche effect between memory buffers, we are not able to empirically validate this due to the fact that Valgrind does not support AES instruction set.

VIII. Related Works

Key recovery attack against cryptosystem. Traditional key recovery attacks assume physical access to the cryptosystem (e.g., smart card), and they can use timing (e.g., timing attack [31]) and power (e.g., differential power analysis [32]) to recover the cryptographic secrets. Furthermore, electromagnetic radiation has been used to recover the secret key from a nearby cryptographic device [25]. Brumley and Boneh [14] extended the timing attack from local to network-based. Shamir and Someren [45] investigated how to efficiently identify stored secret keys from large amount of data (such as file). Recently researchers have investigated how to recover secret keys from memory offline [29] and live applications [28], [23]. Most of these key recovery attacks depend on specific implementations, and they are not able to pinpoint the cryptographic operations. Therefore, they are unable to recover transient keys involved in multiple rounds of nested cryptographic operations.

Binary analysis. Binary analysis has been widely used in malware detection, analysis [42], [24], [12], [52], [33], [20], [16] and protocol reverse engineering [18], [35], [49], [17], [21], [36], [15]. Specifically, Reformat [49], Dispatcher [17] used instruction profiling (i.e., ratio of bitwise and arithmetic instructions) to detect the cryptographic operations from the binary execution. Based on instruction profiling, BCR [15] further extracts the interface of identified encryption/decryption functions from binary executables. Gröbert [26], [27] used combinations of instruction profiling and implementation specific signature (e.g., constants, sequence of mnemonics) to identify cryptographic functions from binary executable. One fundamental limitation of instruction profiling based approaches is that they can not distinguish multiple rounds of nested cryptographic operations, which would appear as one big cryptographic operation to them. Therefore, they are not able to recover the transient cryptographic secrets as shown in Figure 1.

Based on the observation that decrypted data is likely to have lower entropy than encrypted data, Lutz [38] used entropy to detect the decryption operation (only) from the binary execution. Apparently, such method does not work in the presence of multiple rounds of nested cryptographic operations.

By using checksum specific heuristics, TaintScope [48] is able to identify all kinds of checksums (e.g., CRC32, Adler32, MD5) from a given binary execution. BitFuzz [16] used taint degree - the number of the bytes in taint source that would taint the current value, to detect various encoding functions (e.g., encryption, checksum). The key technique used in TaintScope and BitFuzz is based on detecting the mixing of multiple input bytes (rather than the avalanche effect), which will detect those checksum algorithms (e.g., Adler32) that do not exhibit the avalanche effect. Because multiple rounds of nested cryptographic operations as a whole would exhibit the pattern of mixing of multiple input bytes, they would appears as "one big operation" in view of mixing based detection. Therefore, mixing based approaches are not able to pinpoint the boundary between multiple rounds of cryptographic operations thus they can not recover the transient cryptographic secrets in between nested cryptographic operations as shown in Figure 1.

In contrast, CipherXRay can accurately pinpoint the boundary of individual cryptographic operation from multiple nested cryptographic operations. This enables it to recover transient cryptographic secrets that only exist in between nested cryptographic operations.

IX. Conclusions

We have presented a novel binary analysis framework CipherXRay. Based on the defining characteristic of all (good) cryptographic algorithms – the avalanche effect, CipherXRay has been shown to be able to detect public key cryptography, block cipher and hash operations and pinpoint exactly when and where the cryptographic input, output, IV and keys will be in the memory even if they exist for only a few microseconds.

We have empirically validated the effectiveness of CipherXRay with OpenSSL, popular password safe KeePassX, encrypted C&C communication of Agobot, proprietary ciphers used by malware Kraken and Stuxnet, and a number of third party programs. Despite KeePass re-encrypts all the entry passwords in memory within 21 microseconds after they have been read and decrypted from the password file, CipherXRay is able to recover not only all the entry passwords protected by KeePassX but also the 256-bit master key and the 128-bit IV that enable one to directly decrypt the KeePassX password file offline using OpenSSL library.

Our experiments demonstrate that current software implementations of cryptographic algorithms achieve hardly any secrecy if their execution can be monitored. While this new capability helps better analyze sophisticated malwares protected by strong cryptographic algorithms, it raises the question on whether and to what extent a monitored execution could keep its secrecy.

Acknowledgment

We would like to thank the anonymous reviewers for their comments and suggestions that helped us improve the quality of our paper.

References

[1] An encoder/decoder for the VCDIFF (RFC3284) format. http://code.google.com/p/open-vcdiff/.
[2] bzip2 high-quality data compressor. http://www.bzip.org/.
[3] Creates simple file verification (.sfv) listings and tests existing sfv files. http://linux.softpedia.com/get/Utilities/cksfv-6520.shtml.

[4] Detailed Information about the Security of KeePass. http://keepass.info/help/base/security.html.

[5] Intel Advanced Encryption Standard (AES) Instructions Set - Rev 3. http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/.

[6] KeePass Password Safe. http://keepass.info.

[7] KeePassX, Cross Platform Password Manager. http://www.keepassx.org.

[8] Run Windows applications on Linux, BSD, Solaris and Mac OS X. http://www.winehq.org.

[9] The OpenSSL Project. http://www.openssl.org/.

[10] Ultimate Packer for eXecutables. http://upx.sourceforge.net/.

[11] White-Box Cryptography. http://whiteboxcrypto.com.

[12] C. K. Andreas Moser and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007)*, pages 231–245. IEEE, May 2007.

[13] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs (Extended Abstract). In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 2001)*, pages 1–18, 2001.

[14] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, San Diego, CA, USA, August 2003. USENIX.

[15] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, February 2010.

[16] J. Caballero, P. Poosakam, S. McCamant, D. Babić, and D. Song. Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 413–425, October 2010.

[17] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 621–634. ACM, October 2009.

[18] J. Caballero and D. Song. Polyglot: Automatic Extraction of Protocol Format using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 317–329. ACM, October 2007.

[19] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)*, pages 143 – 163, July 2008.

[20] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P 2010)*, pages 61–76. IEEE, May 2010.

[21] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P 2009)*, pages 110–125. IEEE, May 2009.

[22] W. Drewry and T. Ormandy. Flayer: Exposing Application Internals. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT '07)*, August 2007.

[23] T. Duong and J. Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In *Proceedings of the 2011 IEEE Symposium on Security & Privacy (S&P 2011)*, pages 481–489, Oakland, CA, May 2011.

[24] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC 2007)*, page 233 246, June 2007.

[25] K. Gandolfi, C. Mourtel, and F. Olivier. Results of Electromagnetic Analysis. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, May 2001.

[26] F. Gröbert. Automatic Identification of Cryptographic Primitives in Software. Deplima Thesis, Ruhr-University Bochum, Germany, February 2010.

[27] F. Gröbert, C. Willems, and T. Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011)*, September 2011.

[28] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security & Privacy (S&P 2011)*, pages 490–505, Oakland, CA, May 2011.

[29] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60. USENIX, August 2008.

[30] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. *RFC 3447*, IETF, February 2003.

[31] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 1996)*, pages 104–113, 1996.

[32] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 1999)*, pages 388–397, 1999.

[33] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P 2010)*, pages 29–44. IEEE, May 2010.

[34] F. S. Leder and P. Martini. NGBPA Next Generation BotNet Protocol Analysis. In *Proceedings of the 24th Internatinal Conference on Information Security (IFIP/Sec 2009)*, May 2009.

[35] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS 2008)*, February 2008.

[36] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, February 2010.

[37] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 272–280. ACM, October 2003.

[38] N. Lutz. Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master Thesis MA-2008-08, Swiss Federal Institute of Technology Zurich, 2008.

[39] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet Under the Microscope (Revision 1.31). http://www.eset.com/resources/white-papers/Stuxnet_Under_the_Microscope.pdf.

[40] J. Mulroy. Attackers Get Sneakier With Encrypted Malware. http://www.pcworld.com/businesscenter/article/243721/attackers_get_sneakier_with_encrypted_malware.html?

[41] N. Nethercote and J. Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, 2007. ACM Press.

[42] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS 2005)*, February 2005.

[43] T. Pettersson. Cryptographic Key Recoveryfrom Linux Memory Dumps. In *Presentation, Chaos Communication Camp*, August 2007.

[44] I. Popov, S. Debray, and G. Andrews. Binary Obfuscation Using Signals. In *Proceedings of the 16th USENIX Security Symposium*, pages 275–290. USENIX, Auguest 2007.

[45] A. Shamir and N. van Someren. Playing Hide and Seek with Stored Keys. In *Proceedings of the Third International Conference on Financial Cryptography (FC 1999)*, pages 118 – 124, February 1999.

[46] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS 2008)*, February 2008.

[47] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, November 2005.

[48] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P 2010)*, pages 497–512. IEEE, May 2010.

[49] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 2009)*, pages 200–215, September 2009.

[50] B. Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.
[51] W. Yan, Z. Zhang, and N. Ansari. Revealing Packed Malware. *IEEE Security and Privacy*, 6(5):65–69, October 2008.
[52] H. Yin, D. Song, M. Egele, E. Kirda, and C. Kruegel. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 497–512. ACM, October 2007.

Xin Li received the B.S. and M.S. degrees in electrical engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2003 and 2005, respectively. After that, he worked as a software engineer in Intel Asia-Pacific Research and Development Ltd for 3 years. Since September 2008, he has been pursuing the Ph.D. degree at George Mason University. His current research interests include dynamic binary instrumentation, dynamic binary analysis and malware analysis.

Xinyuan Wang is an Associate Professor of Computer Science at George Mason University. He received his Ph.D. in Computer Science from North Carolina State University in 2004 after years professional experience in networking industry. His main research interests are around computer network and system security including malware analysis and defense, attack attribution, anonymity and privacy, VoIP security, digital forensics. Xinyuan Wang is a recipient of the NSF Faculty Early Career Development (CAREER) Award.

Wentao Chang received the B.S. degree (Hons.) in Computer Science and Technology from Nanjing University, Nanjing, China in 2006, M.S. degree in Computer Science from George Mason University in 2010. He is currently working towards the Ph.D. degree in Computer Security at George Mason University. His research interests include malware analysis, web and mobile security.