K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces

Juanru Li
Shanghai Jiao Tong University
Shanghai, China
jarod@sjtu.edu.cn

Zhiqiang Lin
The Ohio State University
Columbus, Ohio, USA
zlin@cse.ohio-state.edu

Juan Caballero
IMDEA Software Institute
Madrid, Spain
juan.caballero@imdea.org

Yuanyuan Zhang
Shanghai Jiao Tong University
Shanghai, China
yyjess@sjtu.edu.cn

Dawu Gu
Shanghai Jiao Tong University
Shanghai, China
dwgu@sjtu.edu.cn

ABSTRACT

The only secrets in modern cryptography (crypto for short) are the crypto keys. Understanding how crypto keys are used in a program and discovering insecure keys is paramount for crypto security. This paper presents K-Hunt, a system for identifying insecure keys in binary executables. K-Hunt leverages the properties of crypto operations for identifying the memory buffers where crypto keys are stored. And, it tracks their origin and propagation to identify insecure keys such as deterministically generated keys, insecurely negotiated keys, and recoverable keys. K-Hunt does not use signatures to identify crypto operations, and thus can be used to identify insecure keys in unknown crypto algorithms and proprietary crypto implementations. We have implemented K-Hunt and evaluated it with 10 cryptographic libraries and 15 applications that contain crypto operations. Our evaluation results demonstrate that K-Hunt locates the keys in symmetric ciphers, asymmetric ciphers, stream ciphers, and digital signatures, regardless if those algorithms are standard or proprietary. More importantly, K-Hunt discovers insecure keys in 22 out of 25 evaluated programs including well-developed crypto libraries such as Libsodium, Nettle, TomCrypt, and WolfSSL.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Cryptanalysis and other attacks*;

KEYWORDS

Dynamic binary code analysis, cryptographic key identification

ACM Reference Format:
Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243783

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243783

1 INTRODUCTION

Many applications today contain cryptographic operations. Without them, basic security mechanisms such as secure communication and authentication can hardly be achieved. In modern cryptography (crypto for short), there is no need to hide the crypto algorithms, i.e., their constructions are open. The only secret in modern crypto are the crypto keys. The security of a crypto key depends on the size of the key, the process that generates the key, and how the key is used. Unfortunately, developers often make mistakes in key generation, derivation, and sanitization that may result in keys being guessed or leaked.

Over the past few years, we have witnessed numerous cases of insecure crypto keys in software implementations. For instance, some keys are generated without sufficient randomness (e.g., the not-so-randomly-generated numbers in virtualized environments [41]), some keys can be easily leaked (e.g., due to software vulnerabilities such as Heartbleed [38]), some keys can be forged (e.g., using unauthenticated encryption [37]), and some developers may just simply misuse the keys (e.g., using a constant symmetric key that is never changed [39] or the same initialization vector to encrypt different versions of a document [67]). As such, there is a strong need to systematically inspect crypto implementations to identify insecure keys.

Unfortunately, crypto software is difficult to analyze for a number of reasons. First, there is a large body of crypto algorithms (e.g., symmetric ciphers, asymmetric ciphers, stream ciphers, digital signatures) that developers can use. Second, crypto software is complex, e.g., it may contain multiple crypto algorithms such as using an asymmetric cipher to exchange a symmetric key as in TLS. Third, crypto software is often proprietary, and thus only executables are available.

There exist prior works that use binary code analysis to analyze crypto software. For example, ReFormat [66] and Dispatcher [32] detect crypto operations based on the execution statistics of bitwise and arithmetic instructions. Gröbert *et al.* [43] propose to identify specific crypto primitives (e.g., RC4, AES) and their parameters (e.g., plaintext or crypto keys) using crypto function signatures and heuristics. Most recently, CryptoHunt [68] proposes a technique called bit-precise symbolic loop mapping to identify commonly

used crypto functions (e.g., AES, RSA). However, none of these prior works detects insecure crypto keys.

In this paper, we present K-Hunt, a tool to identify insecure cryptographic keys in an executable, without source code or debugging symbols. K-Hunt does not use signatures to identify crypto algorithms. Instead, it directly identifies crypto keys and analyzes them to detect insecure keys. In a nutshell, K-Hunt identifies insecure crypto keys by analyzing how keys are generated, propagated, and used. It utilizes the runtime information to locate the code blocks that operate on the crypto keys and then pinpoint the memory buffers storing the keys. Meanwhile, it also tracks the origin and propagation of keys during program execution.

We have implemented K-Hunt atop dynamic binary instrumentation and applied it to analyze the x86/64 binaries of 10 cryptographic libraries and 15 applications that contain crypto operations. K-Hunt identifies 25 insecure crypto keys including deterministically generated keys, insecurely negotiated keys, and recoverable keys. Our results show that insecure crypto keys are a common problem, as the 25 insecure keys K-Hunt identifies are spread across 22 programs. Only three of the 25 programs evaluated do not contain insecure keys. Surprisingly, K-Hunt found insecure keys in some well-established crypto libraries such as Libsodium, Nettle, TomCrypt, and WolfSSL. We have made responsible disclosure to the vulnerable software vendors, and patches are under development.

In short, we make the following contributions:

- We propose a novel binary analysis approach to identify insecure crypto keys in program executables such as deterministically generated keys, insecurely negotiated keys, and recoverable keys. Our approach does not rely on signatures and can be applied to proprietary and standard algorithms.
- We have designed and implemented K-Hunt, a scalable tool that implements our approach. K-Hunt implements various techniques to significantly optimize the performance of the binary code analysis.
- The evaluation results on real world software show that K-Hunt can analyze real world crypto libraries and COTS binaries to identify insecure keys used by symmetric ciphers, asymmetric ciphers, stream ciphers, and digital signatures.

2 BACKGROUND

Since crypto algorithms today are quite standard, developers are mostly concerned about their implementation correctness and runtime robustness. In contrast, the secure use of crypto keys has attracted less attention. This is a problem because in many popular crypto libraries the responsibility of key management is left to the developers, who may not be crypto experts. Therefore, we have witnessed numerous mistakes regarding insecure crypto keys. We summarize below three common mistakes that lead to insecure crypto keys, and that can be detected using K-Hunt.

Deterministically Generated Keys (DGK). NIST has pointed out that "*all keys shall be based directly or indirectly on the output of an approved Random Bit Generator (RBG)*" [31]. However a common mistake is deterministic key generation, i.e., deriving key material from data sources without enough entropy. A hard-coded key in the program is a case of deterministic key generation. Another case is when the key generation process does not provide strong randomness, which enables brute-force attacks against such keys.

Insecurely Negotiated Keys (INK). A key agreement protocol (or key exchange protocol) defines the series of steps needed to establish a crypto key for secure communication among two or more parties. Such protocols allow the participants to securely establish shared keys over an insecure medium, without the need of a previously-established shared secret. An important requirement for a key agreement protocol is that two or more parties should agree on a key in a way that *they all should influence the outcome of the key*. This precludes any undesired third parties from influencing the key choice and is essential to implement perfect forward secrecy. An insecurely negotiated key happens when the key agreement protocol allows a single peer to generate the shared secret without involving the other peers. In particular, many proprietary key agreement protocols directly designate one party to generate the key and then send that key to other parties. In these cases a malicious peer can surreptitiously weaken the protocol's security [30].

Recoverable Keys (RK). Keeping crypto keys unnecessarily long in memory is a vulnerability due to lack of key sanitization. It creates an attack window for attackers to recover the key that can be exploited through code injection or side channel attacks [46, 48]. One root cause of missing crypto key sanitization is that key buffers are usually allocated on the stack or the heap managed by the operating system (OS). However, the OS seldom sanitizes such memory regions. For instance, if a key buffer is allocated on the heap and is freed after the crypto operation, popular OSes such as Windows and Linux will not immediately wipe it. Instead, the buffer is only labeled as "unused" and will be wiped only when re-allocated. Furthermore, in popular crypto libraries the key sanitization responsability is left to the applications, whose developers may not be crypto experts.

3 OVERVIEW

We use a simple but representative program, illustrated in Figure 1, to demonstrate how our insecure crypto key detection works. This simple program encrypts `Data` through masking a `Key` generated by the `keygen` function. The program captures a crypto operation (a simple cipher that mixes `Key` and `Data`) and a crypto key management (a home-made key derivation that generates a random key). It has an insecure crypto key because 1) the key only contains four bytes of randomness and 2) the key is not sanitized after the encryption.

To detect the insecure crypto key in this running example, a security analyst would need to (1) find which code blocks are crypto related; (2) identify the crypto key used by those blocks; (3) check how the identified crypto key is generated, i.e., which data sources affect it and how it is derived from those key materials; and (4) monitor key propagation (i.e., the memory buffers that store the key) to check whether it is still available after the crypto operation. K-Hunt is designed to automate these steps with a principled approach.

Challenges. To detect insecure keys using the above steps, our approach needs to address the following challenges:

```
1  uint8_t Key[16];
2  uint8_t Data[256] = {0};
3
4  void keygen(uint8_t * key, size_t len)
5  {
6        uint8_t seed[4];
7        for ( size_t i = 0; i < 4; ++i )
8              seed[i] = rand() & 0xff;
9        for ( size_t i = 0; i < len; ++i )
10             key[i] = seed[i % 4];
11 }
12
13 void encrypt( uint8_t * buf, size_t len )
14 {
15       for ( size_t i = 0; i < len; ++i )
16             buf[i] ^= Key[i % 16];
17 }
18
19 int main()
20 {
21       keygen(Key, 16);
22       encrypt(Data, 256);
23 }
```

(a) A simple crypto scheme

```
0040101E call rand                                  keygen
00401023 and eax, 0FFh
00401028 mov ecx, [ebp+var_8]
0040102B mov [ebp+ecx+var_C], al
...
00401057 mov  eax, [ebp+arg_0]
0040105A add  eax, [ebp+var_4]
0040105D mov  cl, [ebp+edx+var_C]
00401061 mov  [eax], cl ;                  key buffer writing
```

```
0040108E mov   eax, [ebp+var_4]                     encrypt
00401091 xor   edx, edx
00401093 mov   ecx, 10h
00401098 div   ecx
0040109A movzx edx, byte_413780[edx] ; key buffer reading
004010A1 mov   eax, [ebp+arg_0]
004010A4 add   eax, [ebp+arg_4]
004010A7 movzx ecx, byte ptr [eax]
004010AA xor   ecx, edx
004010AC mov   edx, [ebp+arg_0]
004010AF add   edx, [ebp+var_4]
004010B2 mov   [edx], al
```

(b) Partial corresponding disassembly code

Figure 1: An example illustrating typically how a crypto key is used in a binary executable.

- **How to identify crypto operations without signatures**. Previous works that analyze crypto software often rely on signatures for specific crypto algorithms. Thus, if the algorithm is proprietary, the identification would fail, as no signatures would typically be available. In Figure 1a, the simple, homemade crypto operation cannot be identified with signatures.
- **How to identify the crypto keys**. Even when the crypto operation has been identified, how to accurately locate the memory buffer that contains the crypto key is still non-trivial. For instance, the `encrypt` function in Figure 1a accesses two buffers: `Data` and `Key`. Unfortunately, when analyzing binary executables, there is no semantic information available on the buffers. Thus, our approach must identify which buffer is the crypto key buffer.
- **How to detect insecure crypto keys in complex programs**. Having identified the buffer holding a key, we still need to determine if the key is correctly derived and managed. Unfortunately, programs that contain crypto algorithms are often complex and these algorithms usually only occupy a very small percentage of the entire program. It is infeasible and ineffective to analyze the whole program executable. Thus, we have to design an efficient way for detecting insecure crypto keys.

Insights. Fortunately, all of the challenges listed above can be solved with the following key insights:

- **Identifying crypto operations independent of their implementation**. Oftentimes, crypto operations are identified by scanning the implementation with signatures of well-known crypto algorithms. However, such approach cannot detect propietary algorithms. Instead, our approach identifies the crypto basic blocks at the core of the crypto operation. For this, it uses a dynamic analysis technique that leverages the insight that these core basic blocks usually mingle crypto keys and data, and thus have distinct properties. For instance, as shown in Figure 1a, the key masking operation (at line 16) reads from two data buffers and produces the ciphertext. Such crypto basic blocks have distinguishable properties such as high use of arithmetic instructions, producing data streams with high randomness, and having execution length proportional to the input size. If a basic block meets these three constraints, it is very likely that it is a crypto basic block.
- **Locating the crypto keys**. Once the core crypto basic blocks are identified, our approach then examines the data accessed by those blocks. Typically, a crypto basic block will process two inputs. For encryption, the plaintext and the key. For decryption, the ciphertext and the key. And, for digital signatures, the input message and the key. Note that while the verify function of a digital signature takes three inputs (message, key, signature), only the message and the key are used in the crypto operations. Therefore, in all three cases we need to separate the crypto key from the other input. Interestingly, we notice that the size of the crypto key is usually very small (e.g., 128-bit) compared to the plaintext, ciphertext, or message, which could be of arbitrary length. We also observe that the crypto key and the other input are usually stored at different memory buffers. And, those buffers are usually filled with content derived from different data sources, e.g., a pseudo-random generator for keys and the network or the filesystem for the plaintext/ciphertext/message.
- **Detecting the insecure crypto keys**. Since it is very complex to analyze the entire program to understand the handling of crypto keys, we instead propose a key-centric strategy. Having identified the crypto operations and located the crypto keys, we use the identified keys as an index to further check the origin of each key and its propagation. For instance, through checking the origin of the key in Figure 1 we can find that it is generated from the `keygen` function at line 10, and through checking the input of this function we can discover the crypto key buffer contains inadequate information (i.e., only 32 bits) . Moreover,

by monitoring the key buffer we can observe that its content is preserved until the program terminates, and thus it is an insecure crypto key. This backward and forward key tracking hence provides a simple way to detect insecure crypto keys.

Problem Scope. The objective of this work is to identify insecure crypto keys in binary executables. In particular, we focus on detecting (1) whether the key is generated from deterministic inputs, (2) whether a shared key is generated using key materials from a single party, and (3) whether the key is not sanitized immediately after the cryptographic operations. We focus on analyzing x86/64 stripped executables without source code or debugging symbols. In addition, we focus on ciphers (symmetric, asymmetric, stream) and digital signatures. For these classes of cryptographic primitives, it does not matter which specific crypto algorithm the program uses. Thus, our approach handles both standard and proprietary algorithms. Other cryptographic primitives such as hash functions do not use keys, or in the case of keyed hashes they do not apply the crypto operations on the input and the key simultaneously.

4 DESIGN

K-Hunt uses dynamic analysis to identify insecure crypto keys. It assumes that there exist test cases to execute the program so that it uses the crypto operations. Our approach uses dynamic analysis because it needs statistics about the program execution. Furthermore, static analysis faces many limitations for analyzing memory buffers, which store the crypto keys. At a high level, K-Hunt comprises of two phases:

- **Pinpointing the Key**. In the first phase, detailed in §4.1, the target program is executed with a lightweight coarse-grained binary code instrumentation to firstly identify the crypto basic blocks and then identify the crypto keys they use.
- **Detecting the Insecure Key**. In the second phase, detailed in §4.2, the target program is executed again with a heavyweight fine-grained instrumentation, which tracks the memory reads and writes and conducts a function level taint analysis. Through taint analysis of the pinpointed keys, K-Hunt then detects the insecure crypto keys.

4.1 Pinpointing Crypto Keys

The first phase of K-Hunt is to pinpoint the crypto keys. An overview of how K-Hunt performs this analysis is presented in Figure 2. It first identifies the crypto basic blocks by running the executable with multiple test inputs, and then analyzes the data those basic blocks operate on to locate the crypto keys.

Step-I: Crypto Basic Block Identification. One observation of modern crypto algorithms (e.g., AES, RSA, DSA) is that they are typically built with only a few compact transformations, which correspond to just a few basic blocks in a program. Therefore, if we can identify these basic blocks, we would identify the crypto operations that use them.

Definition 1. *A crypto basic block is defined as a basic block that satisfies the following constraints: (i) the basic block uses arithmetic calculations to implement a cryptographic operation; (ii) the basic block is executed multiple times to mix a data stream with a key stream; (iii) the produced or consumed data have high randomness.*

Our approach first computes, for each basic block, the ratio of x86/64 arithmetic and bitwise instructions (e.g., `mul`, and `xor`) [27, 32]. A basic block is considered a candidate crypto basic block if it has a ratio larger than a pre-defined threshold. This threshold has been experimentally selected as 15% in our current design after analyzing common crypto libraries and utilities. A special case is that a basic block is directly considered a candidate if it contains instructions from the Advanced Encryption Standard instruction set (AES-NI), an extension to the x86 instruction set for microprocessors from Intel and AMD.

Next, it checks whether the candidate basic blocks are *data sensitive*. A basic block is data sensitive if the total amount of execution for the basic block is proportional to the size of its input data. We prepare four test suites with inputs of different size magnitude to test the program, and calculate for each candidate basic block, the total number of executions of the basic block across all inputs, the total basic block's input data size, and the total basic block's output data size. Candidate basic blocks for which the number of executions increases approximately linearly with the ratio of input/output data size are kept as candidate crypto basic blocks. Other candidates are discarded.

At this point, our approach has checked the first two conditions in Definition 1. The last condition checks if the data operated on by the candidate basic block has high randomness. However, each time a basic block is executed it only operates on part of the input data. Thus, our approach accumulates all the data that each candidate basic block operates on during the entire program execution into data bundles.

Definition 2. *A data bundle is defined as the sequence of all the data that an operand of an instruction operates on during the entire execution of the program. The size of a data bundle is the number of data items it contains.*

For example, in Figure 1b the instruction at `0x0040109A` has one memory read operation and is executed 256 times. Therefore, there will be a data bundle with 256 data items: the sequence of value read that is from `byte_413780[edx]`. Our approach focuses on data bundles that are generated by instructions with memory operands, since registers have a limited size that does not typically hold a crypto key. Once the data bundles are built, they are examined to identify those that contain highly random data. For this, our approach leverages the ent [40] utility to measure the randomness of the collected data bundle with both *Chi-Square distribution* and *Monte Carlo π approximation* tests. If ent judges the data bundle as random, the candidate basic block is considered a crypto basic block.

In our running example in Figure 1b, the basic block (`0x0040108E-0x004010B2`) is the crypto basic block to be identified. It satisfies the above constraints for a crypto basic block: it utilizes arithmetic and bitwise instructions to implement the crypto operation, the number of executions of the basic block is proportional to the size of the input data, it accesses several memory buffers, and its produced data has high randomness.

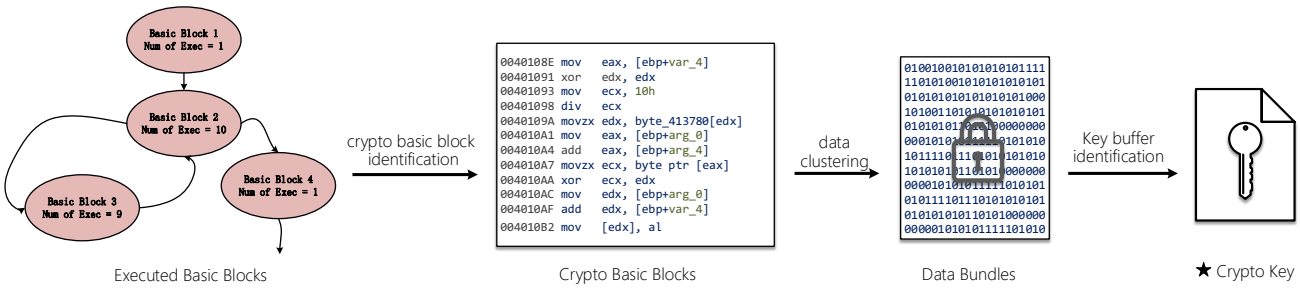

Figure 2: The process of pinpointing crypto keys in binary executables

Step-II: Crypto Key Buffer Identification. Having identified the crypto basic blocks in Step-I, our approach next identifies the crypto keys used by those basic blocks. Different types of data may be handled by a crypto basic block: crypto key, plaintext, ciphertext, and message to be signed. More concretely, a crypto operation will take two inputs: crypto key and plaintext for encryption, crypto key and ciphertext for decryption, and crypto key and message for digital signature. Thus, the crypto key should always be an input to the crypto basic block. Therefore, we can exclude the data output by the crypto basic block and just focus on the input data. Still, our approach has to separate the crypto key from the other input. And, it can no longer use randomness for this since both the crypto key and the ciphertext will have high randomness.

Definition 3. *A crypto buffer is defined as all operated memory addresses in one data bundle of a crypto basic block, and the size of a crypto buffer is the number of unique memory addresses it contains.*

For instance, the memory read data bundle of the instruction at `0x0040109A` in Figure 1b contains 256 items. This bundle, however, only contains 16 unique memory addresses, thus the size of corresponding crypto buffer is 16 instead of 256. When we test the randomness of the data, we use the concept of bundle because a buffer may be accessed randomly and we should concern about the access sequence. If the high randomness is discovered, we then only concern about the range of accessed memory, and thus we use the concept of buffer to help distinguish key from data.

To identify which input is the crypto key buffer, K-Hunt leverages the following two complementary insights.

- **Using the buffer size**. Typically, the crypto key buffer is small compared to the ciphertext/plaintext/message buffer: a key can be stored in a relatively small buffer but the crypto input often needs a larger memory buffer. This feature becomes even more obvious when executing the crypto basic blocks with multiple inputs: either the key is repeatedly used or it is updated between different iterations (e.g., the state update of a stream cipher), and the key is generally stored in a fixed-length memory buffer whereas the length of the ciphertext/plaintext/message varies according to the size of the program input.
- **Using the execution context**. The other insight is that crypto keys and other input data are typically initialized by different functions. If we track the execution context of how the data is initialized, we can easily differentiate them as well. For instance, the used crypto key buffer is usually initialized by a key derivation function or from a pseudo-random number generator, while the plaintext/ciphertext/message is generally directly read from a file or network socket.

4.2 Detecting Insecure Keys

The second phase of K-Hunt detects the insecure crypto keys. Unlike the first phase where we perform a lightweight dynamic binary analysis to collect execution statistics (e.g., the number of executions for a basic block and the randomness of data bundles), the second phase requires a heavyweight dynamic binary analysis to trace how keys are generated and propagated. At a high level, our analysis is a function-level variant of dynamic taint analysis (e.g,. [59, 63]) with the following taint policies.

Taint Sources. K-Hunt uses three different taint tags to capture whether a value has been derived from a local input (i.e., filesystem or return value from the `rand` function), a remote input (i.e., the network), or none of those two (i.e., deterministic value). Thus, each memory location (i.e., byte) in the shadow memory [63] has a two-bit taint tag with values 00 for no input, 01 for local input, and 10 for remote input. At initialization, all memory locations will be assigned the no-input tag. During program execution, if K-Hunt observes that a memory location is assigned from local input (remote input), K-Hunt assigns the local (remote) taint tag to the memory location. For instance, `Key` at line 13 in our running example in Figure 1a will be assigned with a local input tag.

In addition, to quantitatively measure how many of bytes of information generated for crypto keys, we introduce the concept of the length of input for the buffer involved for key generation.

Definition 4. *The input length (IL) of a buffer is the number of bytes derived from program inputs.*

For instance, in our running example the `keygen` function initializes four bytes of the `seed` buffer using function `rand`. Then K-Hunt keeps a 4-byte IL for this buffer.

Taint Propagation. K-Hunt could use a fine-grained taint analysis (e.g., [59]) to trace each instruction and propagate the taint tags correspondingly. However, we found such an approach often incurs high performance overhead to the analyzed program, e.g.,

causing a remote peer to close the network socket due to connection time out, or the GUI freezing for non-networking programs, e.g., WinRAR. Therefore, we have develop a lightweight, function-level, taint propagation policy, which propagates taint tags based only on the memory read and memory write operations inside the execution context of a function. This propagation is based on the fact that K-Hunt only requires knowing whether a buffer contains data from either deterministic, local input, or remote input (i.e., the three taint tags). Therefore, it does not need to propagate the taint tags precisely for each instruction. Instead, it can propagate taint tags at the higher function level, significantly improving efficiency.

More specifically, K-Hunt taints all the data definitions (i.e., memory writes) inside a function using the taint tag of the input data. As such, it only needs to instrument the memory read and write instructions. The memory read instructions define the taint source for the function, and the memory write instructions define the taint tags for the memory data based on the current function's taint . If a new taint source to the function is observed, the new taint tags are unioned with the current function's taint tag. All the defined data inside this function will from that point on have a unioned taint tag (e.g., tag with bits 11 to represent both local and remote input). In our running example, the `seed` buffer in the `keygen` function will be tainted with local input tag, and thus the global `Key` buffer will also be assigned the local input tag.

While propagating the taint tags, we also propagate the IL for the buffers. Such information is particularly useful for determining whether a key has sufficient randomness. In particular, for each function, K-Hunt tracks the total number of IL. Then whenever there is a memory write information, it propagates the current IL to this buffer. When the function returns, it recalculates the IL for each buffer based on both the number of bytes accessed and also the current IL for this function. Assume that one function accesses two buffers during its execution. K-Hunt records how many bytes in each buffer are accessed, respectively. If the number of bytes does not exceed the IL of the host buffer, K-Hunt adds this number to the IL of the newly initialized buffer. Otherwise it adds the IL of the host buffer to the IL of the new buffer. Finally, we check whether the IL of the new buffer is larger than the size of the buffer. If so, the IL is adjusted to the size of the buffer. Through performing such IL propagation, K-Hunt maps the information from input buffer to the output buffer of one function.

Taint Sinks. With the aforementioned taint sources and taint propagation policy, all the memory locations will have a taint tag showing whether the data comes from local or remote input, or is deterministic. To identify insecure crypto keys, K-Hunt checks the taint tag of the buffer at the crypto basic block (to check whether the key is improperly generated) or when the program exits (to check for key residue). In particular, we use the following policies to detect the insecure crypto keys.

- **Detecting DGK**. If a key is not derived from any input, i.e., is deterministic, then K-Hunt considers that the generation of the key is flawed. In addition, K-Hunt checks whether the key receives enough information from non-deterministic inputs. This is done through an analysis of the key buffer IL. For example, in our running example the key is initialized in the `keygen` function. At the beginning of the function a `seed` buffer is initialized with a 4-byte IL. Then the `key` buffer in the same function is initialized. At that time, the IL of the `key` buffer is also assigned as 4 because the function only accesses one buffer with a 4-byte IL. As a result, even if the size of the `key` buffer is 16, it has a smaller IL of 4-byte. Eventually, if the IL of a key buffer has the size less than a threshold, 16 bytes (128 bits) in our current design, we consider the key is insecure. In this case the used key buffer has a 4-byte IL and is obviously insecure.
- **Detecting INK**. An insecurely negotiated key is a crypto key shared between two parties (e.g., a session key between a client and a server) where the key value is only influenced by one party [30]. A negotiated key is the symmetric key used for encrypting or decrypting network data. If we know there is a negotiated key but the taint tag for this key includes only local input tag or remote input tag (not both), K-Hunt considers this is an insecurely negotiated key.
- **Detecting RK**. When a crypto operation terminates, all memory buffers holding involved crypto keys should be cleared [69]. To detect any recoverable keys, K-Hunt searches the memory to check any partial existences of the keys in memory when the process terminates. A key is considered recoverable if one third of its content still exists in memory [49]. Therefore, we cross check the content of the data bundles, and if one third of the content still matches with the original key buffer, K-Hunt considers it a recoverable key. In our running example, the `Key` buffer is allocated in the global memory region, and is not cleared after process termination. K-Hunt therefore detects it is an RK case.

5 IMPLEMENTATION

We have implemented K-Hunt using Intel PIN [56], a popular dynamic binary instrumentation (DBI) framework. We use dynamic analysis instead of static analysis for different reasons. First, we need to measure execution statistics such as the randomness of the runtime data and the number of executions of a basic block. Second, static analysis faces limitations analyzing memory buffers, e.g., due to indirect memory addressing. Third, the static dentification of function boundaries, needed for the function-level taint propagation, is challenging, especially for C++ libraries [28]. Using DBI, K-Hunt can leverage the runtime information to identify function boundaries. Finally, DBI is able to handle executables with some protections (e.g., code packing or VM obfuscation).

We have implemented three Pintools for *code profiling*, *randomness testing*, and *key tracking*. In Phase I, the executable is first run with the code profiling Pintool to find which candidate basic blocks should be tested for randomness. Then, the program is executed again with the randomness testing Pintool to collect the runtime information needed for the randomness test. In Phase II, the key tracking Pintool is used to check how program inputs affect the key derivation and how crypto keys are propagated.

To detect the key residue, we should rigorously have implemented a kernel module to monitor all of the process pages belonging to the target process right after the process terminates. Unfortunately, PIN does not provide such kernel-level APIs. Instead, we instrument the callback function `PIN_AddFiniFunction`

in our key tracking pintool to trigger the memory check. This callback is invoked right after the execution of all user defined cleanup functions and before the process terminates.

Labeling Program Inputs. K-Hunt needs to set the input taint tag when the program receives local or remote input. To this end, it hooks the system APIs that deal with such inputs (e.g., `read`, `fread`, `recv`) , as well as APIs related to random number generation (e.g., `rand()`). The local tag is set if the input comes from the filesystem or a random number generation API, and the remote tag if the input comes from a network socket.

Differential Testing. K-Hunt can optionally use a differential analysis step to identify candidate basic blocks in Phase I that are unrelated to crypto operations, and thus be removed from the candidate set. To this end, it compares two traces, obtained by running the program executable with and without triggering the crypto operations (e.g., executing 7-zip with or without file encryption). Then, it identifies candidate basic blocks that do not appear in the execution with crypto operations, as well as candidate blocks that apper in both executions with and without crypto operations. In both cases, those candidate basic blocks cannot be crypto basic blocks. Note that differential analysis is just an optional optimization to reduce the number of candidate basic blocks to be considered.

On-Demand Tracing. Crypto operations are often CPU-intensive and a dynamic analysis with large performance overhead could significantly interfere with the normal program execution. To address this, K-Hunt uses on-demand tracing , applying heavyweight program analysis only on necessary code blocks. For instance, in the first phase, both the number of executions of candidate basic blocks and the data randomness are analyzed to determine the crypto basic blocks. However, the testing of randomness requires a time-consuming analysis. To reduce this overhead, K-Hunt first uses code profiling to count the number of candidate basic block executions and excludes irrelevant candidate basic blocks. In this manner, it only needs to apply the more expensive randomness testing on the remaining candidate basic blocks. In the second phase of analysis, K-Hunt only instruments the memory read and memory write instructions to propagate the taint tag at function level, as described in §4.2. This significantly reduces the overhead of our taint analysis.

Entropy Test. A time-consuming step in K-Hunt is the randomness test. To speed this step, we conduct a more lightweight entropy test before, so that the randomness test is only applied to those bundles with high entropy. Note that a bundle with high randomness must also possess high entropy, while a bundle with high entropy may not have high randomness [65].

Online Analysis. K-Hunt uses an online analysis approach. It could also operate on execution traces to reduce the runtime overhead. Nonetheless, we found an online approach is more suitable to our goal because offline analysis leads to extremely large execution traces (often >100GB), which create an I/O bottleneck slowdown.

Library	Version	Category	Protection
Botan [3]	1.10.13	Crypto Libraries	-
Crypto++ [6]	5.6.4	Crypto Libraries	-
Libgcrypt [11]	1.6.6	Crypto Libraries	-
LibSodium [12]	1.0.12	Crypto Libraries	-
LibTomcrypt [13]	1.17	Crypto Libraries	-
Nettle [16]	3.3	Crypto Libraries	-
GnuTLS [7]	3.5.13	SSL/TLS Libraries	-
mbedTLS [21]	2.3.0	SSL/TLS Libraries	-
OpenSSL [17]	1.1.0f	SSL/TLS Libraries	-
WolfSSL [26]	3.9.10	SSL/TLS Libraries	-
Application	**Version**	**Category**	**Protection**
7-Zip [1]	9.20	File Compressor	-
Ccrypt [4]	1.10	File Encryptor	UPX
Cryptcat [5]	1.2.1	Messenger	-
Cryptochief [20]	1.337	File Encryptor	-
Enpass [2]	5.6.0	Password Manager	-
Imagine [8]	1.1.0	Picture Browser	UPX
IpMsg [9]	4.60	Messenger	-
KeePass [10]	1.34	Password Manager	-
MuPDF [15]	1.11	PDF Parser	-
PSCP [18]	0.62	SSH client	-
Sage [19]	2.0	Ransomware	-
Ultrasurf [22]	15.04	Proxy client	Themida
WannaCry [23]	1.0	Ransomware	Dynamic DLL
Wget [24]	1.11.4	Downloader	-
WinRAR [25]	5.40	File Compressor	-

Table 1: The collected binary executables in our benchmark.

6 EVALUATION

We have tested K-Hunt with 10 popular crypto libraries and 15 real-world programs, identifying many insecure keys among these programs. In this section, we present our evaluation results. We first describe our experimental setup in §6.1, then present the detection results for crypto keys in §6.2, and finally detail the identified insecure keys in §6.3.

6.1 Experiment Setup

As far we know, there are no standard benchmarks that cover widely used crypto algorithms. As such, we create a benchmark, detailed in Table 1, that contains recent versions of 10 crypto libraries and 15 real-world programs that use crypto operations. Our collected benchmark suite includes: (a) popular standard symmetric ciphers (AES, Twofish), asymmetric ciphers (RSA), stream ciphers (RC4, ChaCha20), and digital signatures (DSA, ECDSA, Ed25519); (b) different key sizes for the same algorithm (e.g., AES-128, AES-256); (c) one proprietary cipher (Cryptochief); (d) proprietary programs for which we do not have access to the source code (Cryptochief, Imagine, UltraSurf, WinRAR); (e) benign programs that use binary code protection techniques such as code packing (Ccrypt, Imagine, UltraSurf); and (f) two samples from recent ransomware families (Sage, WannaCry).

For each crypto library, we developed test programs to encrypt/decrypt using AES, RSA, and to sign/verify using ECDSA. Since Libsodium does not support RSA and ECDSA, we included the Ed25519 digital signature instead.

For each program, we have manually built the ground truth regarding their key management using the program's source code (when available), or by manually reverse-engineering the executables (for proprietary programs and malware). Note that for programs for which source code is available, we only use the source code to build the ground truth. K-Hunt operates on executables

Target	Algorithm	B1	B2	B3	N	S	IL
Botan	AES-256	53	13	7	1	240	32
	RSA-2048	1180	569	162	6	1024	256
	ECDSA	958	921	300	2	224	128
Crypto++	AES-256	1281	26	5	1	240	32
	RSA-2048	1949	924	214	6	896	256
	ECDSA	1916	1425	305	8	288	64
Libgcrypt	AES-256	126	25	3	1	240	32
	RSA-2048	565	463	153	6	896	896
	ECDSA	340	322	49	10	320	96
LibSodium	AES NI-256	7	4	4	1	240	32
	Ed25519	690	686	171	8	288	256
LibTomcrypt	AES-256	60	43	4	1	240	32
	RSA-2048	404	385	69	7	1152	1152
	ECDSA	330	274	72	4	128	97
Nettle	AES-256	38	13	3	1	240	32
	RSA-2048	411	87	61	6	1152	896
	ECDSA	186	92	39	8	288	32
mbedTLS	AES-256	44	40	13	1	240	32
	RSA-2048	154	138	39	12	1664	256
	ECDSA	255	245	47	9	384	64
OpenSSL	AES-256	58	10	4	1	240	32
	RSA-2048	210	175	41	10	1552	640
	ECDSA	188	143	17	6	192	50
WolfSSL	AES-256	50	36	4	1	240	32
	RSA-2048	295	235	36	7	1152	1152
	ECDSA	277	202	27	5	160	32
7-zip	AES NI-256	2	2	2	1	240	32
Ccrypt	AES-256	44	5	1	1	240	32
Cryptcat	Twofish	54	14	7	1	160	varied
Cryptochief	Proprietary ∗	23	12	1	1	8	3
Enpass	AES NI-256	8	3	3	1	240	32
Imagine	DSA-1024 ∗	241	72	12	5	464	928
IpMsg	AES-256	168	12	4	1	240	32
Keepass	AES-256	481	118	19	1	240	32
MuPDF	AES-128	262	46	4	1	176	16
PSCP	AES-256	195	9	5	1	240	32
Sage	ChaCha20 ∗	31	17	2	1	256	32
UltraSurf	RC4 ∗	191	79	6	1	1024	16
WannaCry	AES-128 ∗	26	12	3	1	352	16
Wget	AES-256	268	22	3	1	240	32
WinRAR	AES-128 ∗	181	58	3	1	176	32
	AES-256 ∗	214	51	3	1	240	48

Table 2: Results of the key pinpointing and key identification. ∗ denotes that the implementation of this algorithm is proprietary.

and does not require access to the program's source or debugging symbols.

Input Preparation. We have prepared test cases of different sizes for each program, so that each test case triggers a cryptographic operation. The test cases can be easily produced using the high-level descriptions of the software available on their webpages, e.g., that it encrypts a file with password, there is no need to know any low level details about the program. Furthermore, most of the payload of the test cases can be random, as long as the crypto operation is triggered. For the differential analysis, we also prepared test cases of different sizes that do not trigger a crypto operation.

Executions. For each test case, K-Hunt executes the program three times with different instrumentations. In the first execution, K-Hunt obtains basic block statistics to identify candidate crypto basic operation blocks. In the second execution, K-Hunt tests the data randomness to identify the actual crypto basic blocks and analyzes them to locate the crypto keys. In the third execution, K-Hunt performs the taint analysis of the crypto keys to detect insecure crypto keys.

Host. K-Hunt can run on both Windows and Linux thanks to Pin's support for both platforms. We use a Dell workstation installed with both Windows 7 and Fedora 25 operating systems as the testing platform. The workstation has an Intel Core i7-6700 CPU (3.4GHz), with 16GB physical memory and a 2TB disk.

6.2 Effectivenesses of Key Identification

Table 2 details the results of K-Hunt's key buffer pinpointing for each of the programs[1]. The table shows in column **B1** the number of candidate basic blocks that contain a high arithmetic instruction ratio; in **B2** the subset of B1 candidate basic blocks with a linear relation with the input size; in **B3** the number of identified crypto basic blocks, i.e., B2 candidate basic blocks that produce data bundles with high randomness; in **N** the number of identified key buffers; in **S** the total size of the identified key buffers; and in **IL** the input length of the identified key buffers.

We can observe from columns **B1**–**B3** that all three constraints to identify crypto basic blocks are needed. If we only use the first constraint, i.e., arithmetic instruction ratio, many irrelevant basic blocks are included especially for data transformation programs (e.g., WinRAR, MuPDF). If we only use the first two constraints, asymmetric ciphers still have a large number of candidate basic blocks. Thus, we must also the randomness of the operated data to locate the crypto basic blocks that actually use the crypto key.

We evaluate the correctness of the crypto key detection using the manually-generated ground truth. We do not find any false positives among these programs, but false positives are possible in some cases discussed in §7, e.g., if keys are stored in registers. We find two false negatives, one in Wget, the other in PSCP. Both are due to key exchange protocols that combine an asymmetric cipher with a symmetric cipher. In both cases, K-Hunt only discovers the secret key of the symmetric cipher because there is a linear relation between the number of crypto basic block executions and the input size. Since the input for the asymmetric cipher is a single block when protecting the symmetric key, K-Hunt is not able to pinpoint the public keys. However, if the public key is used to protect the data rather than the key our analysis is able to identify these public keys (e.g., the case of Imagine discussed in §6.3).

To better understand how our analysis performs, we have conducted an in-depth investigation and obtained a number of interesting findings, which are summarized below:

Key buffers of block ciphers. In our benchmark the most frequently used cipher is the AES block cipher. We found that all of the AES implementations store the round keys (11 or 15 rounds) in one key buffer (176 or 240 bytes). Nonetheless, the format of those key

[1]Note that GnuTLS is missing because it relies on Nettle or Libgcrypt as its crypto backend, and the results of both Nettle and Libgcrypt are already listed.

buffers are often diversified in terms of byte orders and buffer sizes. For instance, AES key buffers of WinRAR and Putty-SCP use different byte orders. Meanwhile, malware authors can also deliberately obfuscate the key format to evade the detection. We found that the recent ransomware, WannaCry, allocates a non-standard AES-128 round key buffer with double the normal size (i.e., 352 instead of 176 bytes). As a result, signature-based key searching techniques (e.g., [14, 57, 64]) are unlikely to identify the key. K-Hunt instead is not affected by such an implementation variation. Additionally, K-Hunt handled Intel's hardware AES encryption in the case of Libsodium, Enpass, and 7-zip through directly labeling AES NI instructions and identifying the relevant key buffers.

Key buffers of stream ciphers. In our benchmark three programs (Cryptochief, Ultrasurf, and Sage) use stream ciphers. A stream cipher often maintains a "state" in a fixed-length memory buffer to update the key stream. This state is continuously updated during the encryption. Therefore we could not distinguish it by simply assuming the key is immutable. K-Hunt pinpoints the key buffer from crypto blocks that operate on it, and accurately detects its range. The key buffers of stream ciphers RC4 and ChaCha20 are both 256 bytes and the home-made stream cipher in Cryptochief [20] uses a 8-byte key buffer, as shown in the **S**-column.

Key buffers of public-key ciphers. Unlike block ciphers and stream ciphers where K-Hunt identifies a single buffer (**N** column in Table 2), K-Hunt identified multiple key buffers for asymmetric ciphers. This happens because public keys used by asymmetric ciphers usually consists of several components that may be stored in different buffers, e.g., d and n in RSA. Interestingly, we found the length of key buffers are often much larger than the length of the required public key. We then checked the source code of each crypto library and found that certain public key encryption are actually implemented in a very optimized way, which often uses a varied public key with larger integers. For instance, we found all tested crypto libraries implement the RSA algorithm with the Chinese Remainder Theorem (CRT) optimization, which requires a number of extra parameters (e.g., d_p, d_q, q_{inv}) besides the well-known parameters (e.g., n, d, p, q). In this situation the detected RSA key buffers covers all used large integers.

6.3 Effectiveness of Insecure Key Detection

After pinpointing the crypto keys, K-Hunt uses the key tracking Pintool to identify insecure keys: deterministically generated keys (DGK), insecurely negotiated keys (INK), and recoverable keys (RK). Table 3 summarizes the detection results. Among the 25 tested programs, 22 contain at least one insecure key. Only three (Botan, Crypto++, PSCP) do not have any insecure crypto keys. Overall, K-Hunt finds 25 insecure keys in 8 libraries and 14 applications. Thus, insecure keys occur not only in applications using crypto operations, but also in well-established crypto libraries.

The most common class of identified insecure keys are recoverable keys. K-Hunt found 21 recoverable keys in 8 libraries and 13 applications. K-Hunt also found two deterministic keys (each in a separate application) and two insecurely negotiated keys (each in a separate application as well). In the following, we detail the detection of each class of insecure keys.

Target	DGK	INK	RK			
			NMZ	MMZ	RKPS	RKPH
Botan	-	-	-	-	-	-
Crypto++	-	-	-	-	-	-
Libgcrypt	-	-	-	✓	-	-
LibSodium	-	-	✓	-	-	-
LibTomcrypt	-	-	✓	-	-	-
Nettle	-	-	✓	-	-	-
GnuTLS	-	-	-	✓	-	-
mbedTLS	-	-	-	✓	-	-
OpenSSL	-	-	-	✓	-	-
WolfSSL	-	-	✓	-	-	-
7-zip	-	-	-	-	✓	-
Ccrypt	-	-	-	-	-	✓
Cryptcat	-	-	-	-	-	✓
Cryptochief	✓	-	-	-	-	✓
Enpass	-	-	-	-	✓	-
Imagine	✓	-	-	-	-	-
IpMsg	-	✓	-	-	✓	-
Keepass	-	-	-	-	✓	-
MuPDF	-	-	-	-	✓	-
PSCP	-	-	-	-	-	-
Sage	-	-	-	-	✓	-
UltraSurf	-	✓	-	-	✓	-
WannaCry	-	-	-	-	✓	-
Wget	-	-	-	-	✓	-
WinRAR	-	-	-	-	-	✓

Table 3: Results of the detected insecure keys in the tested benchmarks.

(I) Detecting DGK. Since the key generation process is not managed by the crypto libraries, we excluded the libraries from the detection of deterministically generated keys, and only evaluated the 15 applications. Among the 15 applications, K-Hunt identified deterministically generated keys in two: Cryptochief that uses a proprietary stream cipher and Imagine that uses a close-source implementation of DSA-1024.

The first case of DGK is in the Imagine picture browser software, which uses a digital signature to verify whether a user provided license code is valid. K-Hunt identified five crypto key buffers. Those buffers had only deterministic taint tag, which indicated a deterministic key is used. We then conducted a manual analysis of the application and found a very complex crypto flaw: the program uses an incorrect DSA signature verification to check the license code, which leads to the exposure of private key. Such an insecure key management has actually been leveraged to break the ECDSA digital signature of Sony Play Station 3 [42].

In particular, a DSA cipher contains five large integers: p, q, g as fixed parameters (which are public known), x as the private key (which should never be leaked), and k as a random factor (which should be a random secret). It generates a public key $y = g^x \mod p$. To use DSA for digital signature, it has to go through the following two processes.

- **Signature Generation**. To sign a message m, the DSA cipher computes the signature as a pair (r, s) with a cryptographic hash function H (e.g., SHA-256) through equation (1) and (2), and then distribute r and s to the receiver.

$$r = g^k \mod p \mod q \tag{1}$$

$$s = k^{-1}(H(m) + x \cdot r) \mod q \tag{2}$$

- **Signature Verification**. The receiver then verifies the signature (r, s) with the public knowledge of p, q, g, y and $H(m)$ to

verify whether the signature is valid if v, calculated using the following equation from (3) to (6), equals to r.

$$w = s^{-1} \ mod \ q \tag{3}$$

$$u_1 = H(m) \cdot w \ mod \ q \tag{4}$$

$$u_2 = r \cdot w \ mod \ q \tag{5}$$

$$v = (g^{u_1} \cdot y^{u_2} \ mod \ p) \ mod \ q \tag{6}$$

Through manual reverse-engineering, we noticed that the five detected key buffers correspond to the publicly known parameters p, q, g, y, and the random secret factor k. Thus, Imagine actually leaks the random secret factor k by hard-coding it in the software. A further reverse engineering of the license registration process of Imagine reveals that this software only tells the user s without passing r, when a user provides a message m to generate its digital signature. However, the signature verification requires r in equation (5). Therefore, the software then locally computes r by using equation (1) through a hard-coded k. Unfortunately, the exposure of k is a severe mistake for DSA digital signature. Specifically, if an attacker has obtained a legal pair of (r, s), with the leaked k, it can then compute the private key x with the following equation:

$$x = r^{-1}(k \cdot s - H(m)) \ mod \ q \tag{7}$$

As such, an attacker is able to forge any other legitimate digital signatures with the private key x. We have informed the developer of Imagine, and this vulnerability has been confirmed and the patch is still under development as the time of this writing.

The second case of deterministically generated key was found in Cryptochief, a file encryption tool[2]. This program accepts two files as inputs: one is a key file and the other is a plaintext file. It then outputs an encrypted file. The lengths of the key and plaintext files are both arbitrary, while the length of the encrypted file is equal to that of the plaintext file.

We used K-Hunt to monitor the encryption process of Cryptochief. Interestingly, K-Hunt pinpointed that the key buffer has only 8 bytes, and this key buffer is composed from three local input sources. The IL for each input source is only one byte. Therefore, this key buffer has only three bytes IL. This indicates that the key buffer only acquires at most 24-bits of information from the local input. As such, an attacker can easily brute force all the possible keys with just 2^{24} possibilities to decrypt any ciphertext without the key file. Our further analysis with program source code reveals that the key derivation function of Cryptochief only uses *the head byte, the tail byte and the count of all bytes* of the key file data to generate an 8-byte key buffer.

(II) Detecting INK. In secure key agreement protocols, the shared key should be influenced by all participants. As a result, when K-Hunt reveals that a shared key is generated with non-deterministic inputs, it further checks whether the non-deterministic inputs come from different parties. In our experiments we only consider the client-server model, which involves two participants. Of the five crypto protocols in Cryptcat, Wget, UltraSurf, IpMsg, and PSCP, two protocols in Ultrasurf and IpMsg use insecurely negotiated keys. Both programs generate session keys locally and the server just accepts the key from the client.

More specifically, in the case of IpMsg that adopts a message encryption with RSA-2048 and AES-256, although K-Hunt does not identify the used RSA key, it does discovers that the AES key is selected locally (the key is generated using the Windows `CryptGenRandom` API). Actually, it is a common mistake to conduct a session key exchange if the client encrypts the session key with a deterministic server RSA public key (and then sends it to the server). This not only brings the issue that the session can be controlled by a malicious or tampered client, but also hinders the server to authenticate the client since every attacker could forge the identity through using the public key.

(III) Detecting RK. We investigated the root cause of recoverable keys in both libraries and applications by analyzing the source code, or reverse-engineering the program's binary when the source was not available. The recoverable keys are split into four subclasses in Table 3, two that affect crypto libraries and two that affect applications. These subclasses are explained below.

RK in Crypto Libraries. Recoverable keys are due to the lack of crypto key buffer sanitization. In particular, we found four libraries (Libsodium, LibTomcrypt, Nettle, WolfSSL) that do not zero out the crypto key buffer, which we term No-Memory-Zeroing (NMZ). For instance, in Nettle and WolfSSL, the library developers do not provide any scrubbing functions. For LibTomcrypt, the scrubbing function provided by the library developers is implemented as an empty function. An unintentional case happens in Libsodium. Although it provides a `sodium_memzero` scrubbing function to clean most of its key buffer, Libsodium ignores the round key extension in its AES implementation (utilizing AES NI hardware feature): the round key buffer on the stack is not cleaned and thus could be leaked. We have reported this recoverable key to the developers, and they have patched this issue.

We also found four libraries that use what we term Manual-Memory-Zeroing (MMZ), in which a scrubbing function is provided to clean the key buffer. For instance, mbedTLS uses `mbedtls_aes_free` to zero the key; GnuTLS uses `zeroize_temp_key` in `wrap_nettle_cipher_close` to zero the key; and Libgcrypt uses `gcry_cipher_close` to zero the key. However, a major problem for MMZ is that it does not clean the sensitive data automatically and it may leave the crypto key in memory if the scrubbing function is not invoked.

RK in Crypto Applications. We identify two common root causes for recoverable keys in crypto applications. We term the first RK in Program Heap (RKPH). In this category crypto key buffers are freed without having been sanitized after the crypto operation completes. There are 4 applications with insecure crypto keys in this category. Among those applications, Ccrypt, Cryptcat, and WinRAR, all have been developed for more than 10 years, but they are still vulnerable to the crypto key buffer sanitization issue.

We term the second RK in Program Stack (RKPS). A crypto key buffer can also be placed in the stack and developers often ignore the sanitization of stack variables. This recoverable key category

[2]Cryptochief was first mentioned by Bruce Schneier in his blog article [62] in 2006. The source code of this program is released [20] after the *Hack.lu* 2014 CTF contest.

affects 9 applications: 7-zip, Enpass, IpMsg, KeePass, MuPDF, Sage, UltraSurf, Wannacry, and Wget. Interestingly, this issue helps the forensic analysis of the ransomware families Sage and Wannacry. In particular, since they place the encryption keys in the stack and do not clean them, it is possible for an analyst to retrieve the key from the stack memory, similarly to the heap crypto key identification case in the RSA private key extraction of Wannacry (which affects a Windows XP crypto library that we do not analyze) [45].

6.4 Performance Overhead

As described in §5, K-Hunt includes three Pintools: *code profiling*, *randomness testing*, and *key tracking*. To evaluate the performance overhead of these Pintools, we selected four command-line crypto applications and two representative crypto libraries. We report the performance overhead of the three Pintools compared to null PIN by running the 6 selected programs 10 times each. As shown in Figure 3, on average the performance overhead of code profiling is 2.1 times, 5.7 times for randomness testing, and 7.6 times for key tracking. We observe that the overhead is larger for programs with complex data transformation (e.g., 7-zip), asymmetric ciphers (e.g., RSA), and digital signatures (e.g., ECDSA). Nonetheless, the performance overhead is reasonable for most programs tested by K-Hunt.

7 LIMITATIONS AND FUTURE WORK

K-Hunt has a number of limitations. First, while K-Hunt is able to pinpoint the insecure keys, it does not report any specific crypto algorithms (e.g,. AES, DSA) to which the insecure key belongs. In fact, K-Hunt has all the building blocks to support the identification of each specific crypto algorithm used by an binary executable. More specifically, since K-Hunt has identified the data bundles of key buffer K, ciphertext C, and plaintext P, we can actually perform a brute force search of the encryption algorithm E by computing whether $C = E(K, P)$, where E are those well-known crypto algorithms. Certainly, this is only possible when software uses standard crypto algorithms (e.g., if they follow the never-implement-your-own-crypto practice [29, 61]). We leave the identification of specific crypto algorithms in a binary executable as one of our future efforts.

Second, K-Hunt performs the taint propagation at the function level. That is, if a function uses a tainted tag, all the data defined in that function will have that tainted tag. Such taint propagation may overly propagate the tainted tag, making insecure keys appear secure. For instance, it might be possible that a function uses a random function, but the return value of the random function is never assigned to the crypto key. While we have not encountered such a case, we plan to address this issue by implementing a fine-grained taint propagation policy, and meanwhile address the performance issues caused from this policy.

Finally, K-Hunt will not be able to detect the crypto keys if they are stored in CPU registers. A particular case is the secure in-cache execution [34, 44] technique against the cold-boot attack [46]. In this case the crypto key is never evicted to memory and thus our approach is not able to detect it.

There are also other possible extensions of K-Hunt. For instance, crypto operations are increasingly used by mobile apps to protect their sensitive data. Thus, extending K-Hunt to detect insecure keys in mobile apps is a logical next step.

8 RELATED WORK

Crypto Key Identification. There has been significant interests of identifying crypto keys. For instance, Shamir *et al.* presented an efficient algebraic attack which can locate the secret RSA keys in long bit strings, and more general statistical attacks which can find arbitrary crypto keys embedded in large programs [64]. Halderman *et al.* proposed the cold-boot attack [46] to retrieve the crypto keys from physical memory of the device. Hargreaves *et al.* presented a linear scan method to recover encryption keys from memory [47]. Maartmann *et al.* discussed the forensic identification and extraction of crypto keys [57]. However, those approaches focus on identifying a key with its mathematic structure and do not consider utilizing dynamic program analysis to discover the used key, whereas K-Hunt fully utilizes dynamic program execution information such as the number of basic block execution and data entropy/randomness to identify the crypto key. Moreover, K-Hunt makes a step even further by using the dynamic taint analysis to detect the insecure crypto keys, which is less concerned by previous key identification studies.

Crypto Primitive Identification. A number of efforts have focused on identifying the crypto primitives from various aspects (e.g., [33, 43, 54, 58, 70]). However, archiving efficient and accurate crypto primitive identification is still a non-trivial task. There are still many open challenges needed to be addressed. Recent results indicate that data flow analysis [52, 53] is a promising technique to help identify crypto algorithms. One major problem of state-of-the-art crypto primitive identification techniques is that they are sensitive to function boundary and parameter recognition. Existing techniques (e.g., CryptoHunt [68]) require the boundary of crypto function to be identified accurately to recognize crypto function.

Crypto Misuse Detection. Public awareness of crypto flaw is growing and the increased awareness has resulted in an increase of efforts to detect crypto misuses [37, 50, 55]. Over the past a few years, many crypto misuse cases in mobile apps and firmwares (e.g., [35, 39]) have been discovered. For commodity software, some crypto misuses for popular software products are also discovered [36, 37, 41, 67]. Recently, TaintCrypt [60] proposed the concept of cryptographic program analysis to help developers detect the crypto misuse using LLVM-based static source code analysis. However, it requires the source code to conduct the analysis. K-Hunt complements the existing crypto misuse detection approach by exclusively focusing on identifying the insecure crypto keys from binary executables.

Comparison. Clearly we are not the first to look into the security issues of crypto code, and there are a number of closely related works that focus on identifying the crypto primitives, as shown in Table 4. In particular, among the compared systems, Kerckhoffs, Aligot, Crypto-DFG, and Cryptohunt require the pre-defined templates to identify crypto algorithms. Therefore, they cannot detect proprietary ciphers. ReFormat, Dispatcher, and MovieStealer are not specifically designed for crypto primitive identification, and thus they

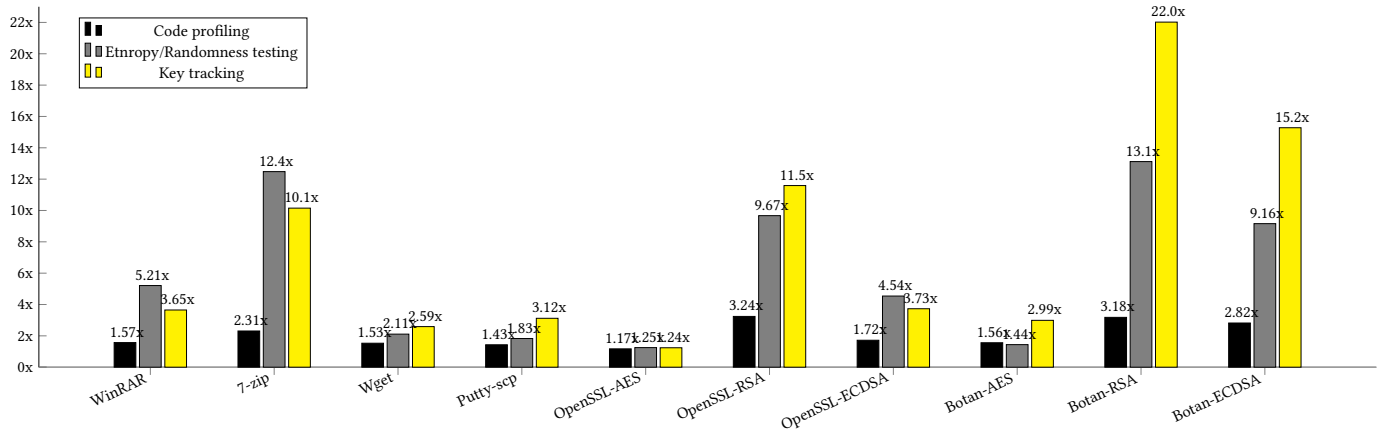

Figure 3: Runtime overhead (times) of three pintools of K-Hunt compared to null PIN.

Systems	C1	C2	C3	C4	C5	C6	C7	C8
Aligot [33]	✗	✓	✓	✓	✓	✗	✓	✗
CipherXRay [54]	✓	✓	✓	✓	✓	✓	✓	✗
Crypto-DFG [52]	✗	✗	✓	✓	✓	✓	✗	✗
Cryptohunt [68]	✗	✓	✓	✓	✓	✓	✗	✗
Dispatcher [32]	✓	✗	✓	✓	✓	✓	✗	✗
Kerckhoffs [43]	✗	✓	✓	✓	✓	✗	✓	✗
MovieStealer [65]	✓	✓	✓	✓	✗	✓	✗	✗
ReFormat [66]	✓	✗	✓	✓	✓	✓	✗	✗
K-Hunt	✓	✓	✓	✓	✓	✓	✓	✓

C1: No need of crypto template
C2: Obfuscation resilient
C3: Detecting block cipher
C4: Detecting stream key cipher
C5: Detecting public-key cipher
C6: Detecting proprietary cipher
C7: Identifying crypto key
C8: Detecting insecure key

Table 4: Comparison with the closely related works.

cannot identify the crypto keys. Only CipherXRay and K-Hunt can identify both proprietary ciphers and crypto keys, but CipherXRay did not make any attempt to identify the insecure keys. Moreover, a substantial difference between K-Hunt and CipherXRay is that K-Hunt focuses on the core part of a crypto algorithm and identifies keys from only several crypto blocks. In contrast, CipherXRay needs to recover both input and output parameters of the entire crypto algorithm. Thus it still suffers from the issue of how to accurately identify the boundary of parameter buffers and faces both false positives and false negatives [51].

An important requirement for the crypto identification is that the analysis should not affect the normal execution of the program. ReFormat, Dispatcher, MovieStealer, and our K-Hunt utilize lightweight heuristics, which do not impose much overhead to the normal execution. Kerckhoffs, Cryptohunt and Aligot use an offline analysis strategy. Crypto-DFG performs a purely static Data Flow Graph (DFG) isomorphism based detection and thus does not affect the execution either. Only CipherXRay adopts a heavyweight dynamic taint analysis and may affect the execution. For instance, it takes CipherXRay about 40 minutes to recover a 1024-bit RSA private key, which is unacceptable for establishing normal network connection.

We also compared the accuracy of each system. We found that if the approach requires a very precise criteria to judge the crypto function, it yields false negative. For instance, Kerckhoffs uses I/O comparison with known cryptographic functions to identify specific ciphers. However, this comparison is very sensitive to the implementation variation. Moreover, we also found that only using one heuristic feature to detect crypto algorithm is often not accurate. Dispatcher, for example, has both false positives and false negatives [43, 58]. Another case is CipherXRay, which only checks whether all bits of the output buffer are affected by each bit of the input buffer. For the cryptographic avalanche effect, however, the criteria becomes if one bit of the input buffer is flipped, the output buffer changes significantly (e.g., half the output bits flip). As a result, CipherXRay does not check the intrinsic properties of the avalanche effect and may suffer from false positives. In contrast, K-Hunt focuses on the intrinsic properties of crypto operations, does not require any templates or signatures, and is thus crypto implementation agnostic.

Finally, since binary executables can be obfuscated, the identification of crypto primitives must also consider the code obfuscations. Among the compared systems, Dispatcher, ReFormat, and Crypto-DFG can be easily cheated by changing the instructions with alternatives and thus are not obfuscation-resilient. For obfuscation-resilient systems such as Kerckhoffs, Aligot, CipherXRay, and Cryptohunt, they are based on semantics of crypto. For K-Hunt, it utilizes the fact that even if the crypto basic blocks are obfuscated, e.g., certain arithmetic instructions are replaced by other equivalent arithmetic instructions, the runtime features of execution number and high entropy/randomness cannot be removed. Therefore, K-Hunt can still work against obfuscated crypto code.

9 CONCLUSION

We have presented K-Hunt, a dynamic analysis system to identify insecure keys in an input executable. K-Hunt first pinpoints the crypto keys by leveraging general properties of crypto operations. Then, it identifies insecure keys, namely, deterministic generated keys, insecurely negotiated keys, and recoverable keys by tracking how the crypto keys are generated and propagated. We have implemented K-Hunt and tested it with 10 cryptographic libraries and 15

applications that contain crypto operations. Our evaluation results show that K-Hunt pinpoints the crypto keys used by symmetric ciphers, asymmetric ciphers, stream ciphers, and digital signatures. More importantly, K-Hunt discovers insecure keys in 22 out of 25 evaluated programs, including in well-established crypto libraries such as LibSodium, Nettle, TomCrypt, and WolfSSL. We have responsibly disclosed the vulnerabilities to the affected software vendors and patches are under development.

10 AVAILABILITY

The source code of K-Hunt and also the tested benchmark will be made public available at https://github.com/gossip-sjtu/k-hunt/.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their valuable comments and helpful suggestions. The work was partially supported by the Key Program of National Natural Science Foundation of China under Grant No.:U1636217, the National Key Research and Development Program of China under Grant No.: 2016YFB0801200. This research was also partially supported by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731, the Spanish Government through the DEDETIS grant TIN2015-7013-R, and the European Union through the ElasTest project ICT-10-2016-731535. We specially thank the Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial joint Institution of FinTech Security*.

REFERENCES

[1] 7-Zip. http://www.7-zip.org/.
[2] Best password manager for iOS, Android, Windows, Linux, Mac | Enpass. https://www.enpass.io/.
[3] Botan: Crypto and TLS for C++11. https://botan.randombit.net/.
[4] ccrypt. http://ccrypt.sourceforge.net/.
[5] Cryptcat Project. http://cryptcat.sourceforge.net/.
[6] Crypto++ Library 5.6.5 | Free C++ Class Library of Cryptographic Schemes. https://www.cryptopp.com/.
[7] GnuTLS. http://www.gnutls.org/.
[8] Imagine: Freeware Image & Animation Viewer for Windows. http://www.nyam.pe.kr/dev/imagine/.
[9] IP Messenger. https://ipmsg.org/index.html.en.
[10] KeePass Password Safe. https://keepass.info/index.html.
[11] Libgcrypt. https://gnupg.org/software/libgcrypt/index.html.
[12] libsodium. https://download.libsodium.org/doc/.
[13] libtomcrypt. http://www.libtom.net/LibTomCrypt/.
[14] Manpages of aeskeyfind in Debian jessie. https://manpages.debian.org/jessie/aeskeyfind/index.html.
[15] MuPDF. https://mupdf.com/.
[16] Nettle - a low-level crypto library. https://www.lysator.liu.se/~nisse/nettle/.
[17] OpenSSL. https://www.openssl.org/.
[18] PuTTY: a free SSH and Telnet client. https://www.chiark.greenend.org.uk/~sgtatham/putty/.
[19] Sage ransomware - Malwarebytes Labs. https://blog.malwarebytes.com/threat-analysis/2017/03/explained-sage-ransomware/.
[20] Source Code of CryptoChief. https://github.com/ctfs/write-ups-2014/tree/master/hack-lu-ctf-2014/cryptochief.
[21] SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/.
[22] Ultrasurf and Ultrareach - Internet Freedom, Privacy, and Security. https://ultrasurf.us/.
[23] WannaCry ransomware attack - Wikipedia. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.
[24] Wget - GNU Project. https://www.gnu.org/software/wget/.
[25] WinRAR archiver, a powerful tool to process RAR and ZIP files. https://www.rarlab.com/.
[26] wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/.
[27] Intel® 64 and ia-32 architectures software developer's manual. *Combined Volumes*, 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, 2018.
[28] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proc. 25th Usenix Security Symposium*, 2016.
[29] A. Apvrille and M. Pourzandi. Secure Software Development by Example. *IEEE Security & Privacy*, 3(4):10–17, 2005.
[30] J.-P. Aumasson. Should Curve25519 keys be validated? https://research.kudelskisecurity.com/2017/04/25/should-ecdh-keys-be-validated/, 2013.
[31] E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication*, 800(133), 2012.
[32] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.
[33] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2012.
[34] Y. Chen, M. Khandaker, and Z. Wang. Secure In-cache Execution. In *Proc. 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.
[35] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A Large Scale Analysis of the Security of Embedded Firmwares. In *Proc. USENIX Security Symposium*, 2014.
[36] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs. Security Analysis of Pseudo-random Number Generators with Input:/dev/random is not Robust. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2013.
[37] T. Duong and J. Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2011.
[38] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
[39] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2013.
[40] ENT: A Pseudorandom Number Sequence Test Program. http://www.fourmilab.ch/random/.
[41] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift. Not-so-random numbers in virtualized linux and the whirlwind rng. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 559–574. IEEE, 2014.
[42] fail0verflow. PS3 Epic Fail. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf, 2010.
[43] F. Gröbert, C. Willems, and T. Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *Proc. International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.
[44] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with Private Keys without RAM. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
[45] A. Guinet. Wannacry in-memory key recovery for WinXP. https://github.com/aguinet/wannakey, 2017.
[46] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.
[47] C. Hargreaves and H. Chivers. Recovery of Encryption Keys from Memory Using a Linear Scan. In *Proc. International Conference on Availability, Reliability and Security (ARES)*, 2008.
[48] K. Harrison and S. Xu. Protecting Cryptographic Keys from Memory Disclosure Attacks. In *Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.
[49] N. Heninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Halevi S. (eds) Advances in Cryptology - CRYPTO 2009. Lecture Notes in Computer Science, vol 5677*. Springer, Berlin, Heidelberg, 2009.
[50] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Proc. Asia-Pacific Workshop on Systems (APSys)*, 2014.
[51] P. Lestringant. *Identification of Cryptographic Algorithms in Binary Programs*. PhD thesis, Université Rennes, 2017.
[52] P. Lestringant, F. Guihéry, and P.-A. Fouque. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *Proc. ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2015.
[53] P. Lestringant, F. Guihéry, and P.-A. Fouque. Assisted Identification of Mode of Operation in Binary Code with Dynamic Data Flow Slicing. In *Proc. International Conference on Applied Cryptography and Network Security (ACNS)*, 2016.
[54] X. Li, X. Wang, and W. Chang. CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution. *IEEE Transactions on Dependable and Secure Computing*, 11(2):101–114, 2012.

[55] Y. Li, Y. Zhang, J. Li, and D. Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Proc. International Conference on Network and System Security (NSS)*, 2014.

[56] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[57] C. Maartmann-Moe, S. E. Thorkildsen, and A. Årnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6:132–140, 2009.

[58] F. Matenaar, A. Wichmann, F. Leder, and E. Gerhards-Padilla. CIS: The Crypto Intelligence System for Automatic Detection and Localization of Cryptographic Functions in Current Malware. In *Proc. International Conference on Malicious and Unwanted Software (Malware)*, 2012.

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

[60] S. Rahaman and D. Yao. Program analysis of cryptographic implementations for security. In *Proc. IEEE Secure Development Conference (SecDev)*, 2017.

[61] B. Schneier. Cryptography: The Importance of Not Being Different. *Computer*, 32(3):108–109, 1999.

[62] B. Schneier. Schneier on Security: The Doghouse: KRYPTO 2.0. https://www.schneier.com/blog/archives/2006/06/the_doghouse_kr.html, 2006.

[63] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proc. 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[64] A. Shamir and N. Van Someren. Playing "hide and seek" with stored keys. In *Proc. International conference on Financial Cryptography (FC)*, 1999.

[65] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services. In *Proc. USENIX Security Symposium*, 2013.

[66] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. 14th European Symposium on Research in Computer Security*. 2009.

[67] H. Wu. The Misuse of RC4 in Microsoft Word and Excel. *IACR Cryptology ePrint Archive*, 2005.

[68] D. Xu, J. Ming, and D. Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 921–937. IEEE, 2017.

[69] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko. Dead Store Elimination (Still) Considered Harmful. In *Proc. 26th Usenix Security Symposium*, 2017.

[70] R. Zhao, D. Gu, and J. Li. Detection and Analysis of Cryptographic Data Inside Software. In *Proc. Information Security Conference (ISC)*, 2011.