Detection and Analysis of Cryptographic Data inside software *

Ruoxu Zhao, Dawu Gu, Juanru Li, and Ran Yu

Lab of Cryptology and Computer Security
Dept. of Computer Science, Shanghai Jiao Tong University, Shanghai, China

Abstract. Cryptographic algorithms are widely used inside software for data security and integrity. The search of cryptographic data (include algorithms, input-output data and intermediated states of operation) is important to security analysis. However, various implementations of cryptographic algorithms lead the automatic detection and analysis to be very hard. This paper proposes a novel automatic cryptographic data detection and analysis approach. This approach is based on execution tracing and data pattern extraction techniques, searching the data pattern of cryptographic algorithms, and automatically extracting detected Cryptographic algorithms and input-output data. We implement and evaluate our approach, and the result shows our approach can detect and extract common symmetric ciphers and hash functions in most kinds of programs with accuracy, effectiveness and universality.

Keywords: Cryptographic data, Symmetric Cipher, Hash Function, Data Pattern, reverse engineering

1 Introduction

The use of cryptographic algorithms to protect private information is common in software. Software dealing with huge amount of data such as Archive and compression tools, Disk encryption tools, Instant Messengers often use symmetric ciphers and hash functions to encrypt, decrypt and verify the data.

In practice, the complexity of binary program understanding makes analysts hard to identify which ciphers are inside the software, even only standard algorithms such as the AES, RC4 or SHA-1 are used. What's more, many programs achieve security through obfuscation. For instance, SkyPE uses RC4 algorithm while obfuscating it so that analysts spent years to understand[5]. It is important to develop automatic techniques for the analysts to detect specific cryptographic algorithms before security analysis.

Compared to the theoretical analysis of cryptographic algorithms, the analysis of the implementation is at most a craft rather than a science [12] [8] [9]. The main difficult is that the implementations of one algorithm might be various even if the mathematical abstraction is the same. For instance, the AES

* Supported by SafeNet Northeast Asia grant awards.

takes different implementations on 8-bit platform and 32-bit platform. Many cryptography libraries also use loop unwinding to optimize the algorithms and yet change the form of implementations. Malicious program even modifies or obfuscates the code, trying to fail the analysis. How to identify cryptographic algorithm inside program accurately and effectively is still an open problem that the existing tools cannot solve perfectly.

In this paper, we take the first leap toward cryptographic data detection and analysis based on the *data pattern*. To the best of our knowledge, all existing cryptographic algorithm identification techniques focus on program analysis or memory dump analysis [11] [6]. These techniques try to recover the abstract structure of algorithms inside programs or dumped memory and judge the existence of certain ciphers. Our approach, however, observes the data feature and dependency of specific ciphers during runtime information. We do program tracing first and conduct data analysis to extract the so called *data pattern*, which is the input-output of certain instruction collection. The pattern gives the analyst clues to quickly detect and understand the encryption process of the program. We implement an analysis system to achieve the goal of automatic identification, and the results show that our system can not only detect symmetric ciphers and hash functions in most kinds of programs with high accuracy, but could also extract cryptographic parameters such as expanded round key automatically.

The approach we proposed is able to reduce manual work significantly in debugging, forensic analysis and reverse engineering. Furthermore, the universal model we adopted is expected to be applied to different implementations of the same cipher regardless of the programming language.

2 Background and Related Work

Identification of cryptographic algorithm and data is an important yet seldom discussed topic in program analysis research. State of the art tools and techniques for cryptographic algorithm identification are divided into static based and dynamic based. In this section we summarize existing works and discuss their inadequacy.

2.1 Static Analysis based Approaches

Static analysis based cipher identification is the most widely used technique. Many tools have been developed to help analyzing such as Krypto Analyzer (KANAL) and SnD Crypto Scanner. The key step of static analysis is to parse binary or source code of the program and try to find unique pattern of specific ciphers. Static analysis based approaches strongly rely on signatures, which are often the constant values related to certain ciphers or specific instruction sequence related to certain version of cryptographic libraries. The static based approaches have many defects. First, they often rely on pre-build signature libraries, and detection fails when the signature changes with software updating or code re-compile. Second, they cannot deal with packed programs because the

normal code is compressed or encrypted before execution. Finally, static analysis based approaches only detect the existence of ciphers, but cannot analyze particular encryption and decryption data.

2.2 Dynamic Analysis based Approaches

Dynamic analysis of software is the hot topic of security analysis in recent years especially using emulation technique or program instrumentation. Although many approaches and tools have been developed to do universal analysis[2] [4], Felix Gröbert's work[7] is the first significant dynamic analysis focusing on cipher identification. His work uses PIN tools[10] to dynamic trace the program, and then mixes signature based searching with simple memory reconstruction and searching. However, the model adopted by [7] takes advantage of many observation. For instance, the proposed signature-based and generic bitwise-arithmetic/loop based identification methods are all based on signatures or unique tuples, which are not so dynamic and universal. The only general identification method in [7] is generic memory-based identification method. The method is focused on memory data and uses verifiers to confirm an XOR encryption or a relation- ship between the input and output of a permutation box. A set of possible key, plaintext, and ciphertext candidates are passed to a reference implementation of the particular algorithm. If the output of the algorithm matches the output in memory, the verifier has successfully identified an instance of the algorithm including its parameters. Although this method exploits relationship between plaintext and ciphertext, many potential information is ignored. On the other hand, in digital forensic research, novel methods for cryptographic key identification in RAM are proposed[11] which relies on the property that the keys in memory is far more structured than previously believed.

In this paper we combine the dynamic analysis with structured key data. First we define the input-output of certain instruction collection as *data pattern*, then using the concept of data pattern we can easily analyze the uniqueness of cryptography algorithms by exploit details about the algorithms. For instance, according to the key concept of Symmetric ciphers - *pseudorandomness*, we can search and find the pseudorandom data pattern and test if the data pattern correspond to certain Symmetric ciphers.

3 Cryptographic Data Pattern Analysis

Our goal is to automatically detect cryptographic data, which includes the executed instructions of cryptographic primitives, encrypted data and secret keys. The main idea is to analyze a program's runtime data[1] rather than instructions and generate some data patterns matching to certain cryptographic primitives.

Although the mathematical definition of cryptographic primitives are determinate, the implementations of the same cryptographic algorithm can be quite different. For example, real-world programs may use optimizations like table

lookup to speed up the cryptographic algorithms(e.g., AES fast implementation). Common cryptographic libraries such as OpenSSL and Crypto++ also take different approaches to implement the same algorithm. Programmers may even have their custom implementations. What's more, code obfuscation is often used to protect software, which makes the obfuscated code extremely difficult to analyze. However, we discovered that the input and output data must fulfill certain relations for deterministic algorithms. That is, if the input is given, there should be only one single possible output to a deterministic algorithm. By verifying if the input and output data match the pattern of a certain algorithm, we can say that a program execution contains the data characteristics of a certain algorithm with high credibility. Even if the program is obfuscated, the input and output data has to be present in the program execution data, which can be then analyzed regardless how the data is processed.

Because the size of traced data is usually very large, we have to determine the data sampling points. We found that modern computer programs are highly structured. The control flow of a traced program tells us how a program is executed. Although instructions are not important to data analysis, they help to build up high-level structures of traced programs. In our analysis, we have four levels of data representations during the analyzing process. The structure of these high-level representations are shown in Figure 1.

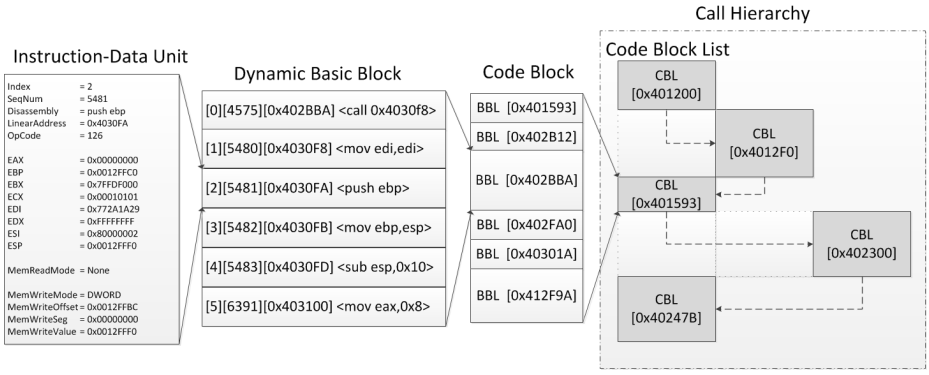

Fig. 1. Data Representations

Instruction-Data Unit An Instruction-Data Unit is the basic unit of program tracing. It contains the instruction binary data, register values, linear address, memory access information, etc. A traced file usually contains millions of Instruction-Data Units.

Dynamic Basic Block A Dynamic Basic Block(Dynamic BBL) contains a sequence of instructions to form a fixed group which has only one entering and one exiting instruction. The Dynamic Basic Block Generation algorithm is a

little different than static ones, because we have to determine the control flow according to the actual traced result.

Code Block A Code Block(CBL) consists of a sequence of continuous Dynamic BBLs that are executed without calling other functions. That is, a new CBL is generated when a call instruction is executed. Code Blocks are used to construct the Call Hierarchy of a function.

Call Hierarchy A Call Hierarchy of a traced program is a recursive structure of Code Blocks, represented by a Code Block List(CBL List). A CBL List contains a single function call, which may call other functions during its execution, thus a CBL List may contain other CBL Lists to build up the recursive structure of a function call.

3.1 Data Patterns

Based on the concept of Instruction-Data Flow, the analyst could extract Intermediate Data State at any arbitrary time of execution. The Intermediate Data State contains the virtual memory state after certain instructions are executed. As mentioned earlier, we made the assumption that the parameters of cryptographic primitives must appear in memory during its execution. Our goal is to verify the existence of cryptographic algorithms by examining if the input and output parameters which match a certain pattern are contained in memory, and therefore we can extract the parameters.

A data pattern are defined as the mathematical relationship between the input and output data of a particular cryptographic algorithm. We know that for deterministic algorithms, output data is determined once the input data is given. This pattern is the key feature that is used in our analysis. It is unnecessary to know specifically how an algorithm is implemented in the target program; all we have to do is to verify the relationships between input and output data using our own implementation. Basically, the data pattern for any cryptographic algorithm is unique and concrete, therefore it can be used as a signature of algorithms in our analysis.

Dynamic Data Patterns A dynamic data pattern is a group of data that matches one or more data templates at runtime. Dynamic data patterns must be verified at runtime, because the content of data cannot be pre-defined. Some examples of dynamic data patterns are:

- Feistel cipher
  Feistel cipher encryption takes plaintext and a group of keys as input, and ciphertext as output. Here the plaintext, ciphertext and keys are not pre-defined, but can be verified during runtime. We describe the Feistel cipher encryption calculation as $F(pt, k)$, which takes the plaintext(pt) and keys(k) as input parameters, and outputs the ciphertext(ct). By dynamically verifing if pt, k and ct satisfy $ct = F(pt, k)$, we can verify the existence of the data pattern of Feistel cipher encryption.

- Rijndael key expansion
  Rijndael key expansion is used to expand a short key into a group of separate round keys. Also, neither the input key nor the expanded keys can be pre-defined.
- RC4 key scheduling
  Similar to Rijndael key expansion, the input and output of RC4 key scheduling is unknown until runtime.

Static Data Patterns Unlike dynamic data patterns, static data patterns can be pre-defined. They can be simply a block of data with known content. A good example of static data patterns is the 256-byte S-box and inverse S-box for AES. Their content is pre-determined and usually directly appear in memory. Another example is the constants in hash functions such as SHA-1. In our analysis, we do not directly use constant signatures as direct evidence of existence of cryptographic algorithms, but they can be used to locate the cryptographic routines.

Data Element Formats The Intermediate Data State of a program trace at any arbitrary time consists of a group of memory chunks. A memory chunk is a block of memory that is continuous in its linear address, demonstrated in Figure 2. Data elements are extracted from these memory chunks. There are three kinds of data elements in our analysis:

- Fixed length. For example, a 128-bit memory chunk containing a block of AES plaintext.
- Variable length. For example, the expanded key in RC4 can be either 256 bits(8-bit each element) or 1024 bits (32-bit each element).
- Arbitrary length. For example, the input key in Blowfish can be from 1 bit to 448 bits.

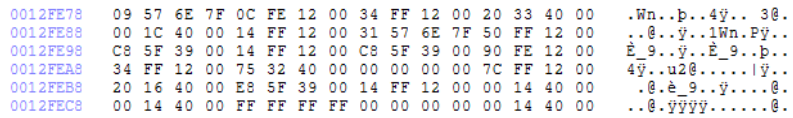

Fig. 2. Memory Chunk

Another thing we should pay attention to is that data elements are usually aligned. In x86 architecture, a block of memory usually has a 32-bit alignment to get maximum performance. So we treat alignment as another property for data elements.

4 Implementation

Our whole program analysis system consists of two parts: the front end is a tracing engine called Fochs, and the back end is a program analyzer called Lochs. A system architecture overview is shown in Figure 3.

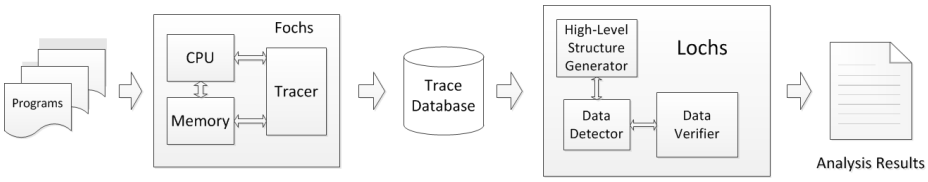

Fig. 3. System Architecture Overview

To conduct an analysis, the testing programs are first executed in Fochs program tracer. The trace is done manually, and its result is saved to trace database for analysis. Then traced data is analyzed in Lochs program analyzer, where possible cryptographic algorithms are examined. After the analysis, a report is generated with the analysis results.

4.1 Fochs: Data Tracing System

The data tracing system we use is the digital forensic analyzer called Fochs. Fochs is based on the open-source Bochs x86 PC emulator[3]. The reason why we choose Bochs is that Bochs performs full-system emulation, and we can conveniently access the CPU status and memory status. We modifies Bochs so that it can trace program execution including its context, and save the trace result for further analysis. The structure of Fochs is shown in Figure 4.

To get a valid and usable trace result, there are three major problems we should solve: what context data should be traced during program execution, which instructions should be traced, and how can we pick the process we want to trace in a multi-process environment. We analyzed our requirements and came up with solutions to these problems.

Execution Context To get a valid program trace result, we have to log the register values and memory accesses for each instruction. We figured that only the register values before each instruction execution are necessary for our analysis, so only the values of general purpose registers before each instruction execution are traced. We also found that in common cases, each instruction has at most one memory reading and one memory writing in the current x86 architecture, thus one memory reading and writing is traced for each instruction. Another important value is the linear address of each instruction, which is the Instruction Pointer register value. Also the instruction binary code is traced for disassembling.

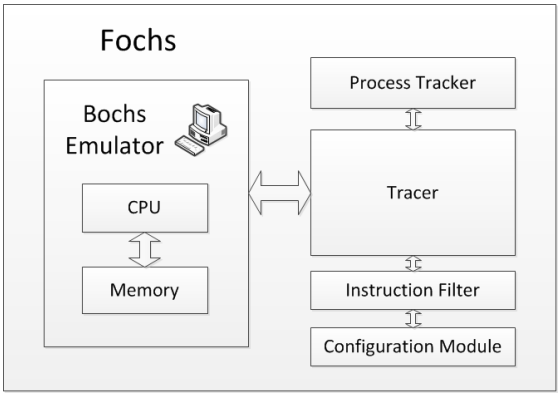

Fig. 4. Fochs program tracer

Some repeat speedup instructions may have multiple memory accesses in a single instruction execution. In our implementation of Fochs, we trace the instruction whenever it has more than one memory access of the same type (reading/writing), acting like a single instruction executed multiple times. We also disabled the MMX and SSE instructions so that no more than 32-bit size of memory can be access during one cycle of instruction execution.

Instruction Filtering If we trace every instruction that CPU executes, the result would be tremendous. We have to eliminate the number of instruction traced to focus on the instructions that contain our analyzing target. We do instruction filtering primarily based on the linear address. In Windows operating system, user space and kernel space are separated, where user space is in low address and kernel space is in high address. The address where an executable is loaded into memory can be easily found using any PE analysis tool or debugger. We limit the linear address that we trace to the bounds of the traced executables, and in this way we are able to ignore the unnecessary OS execution code such as process scheduling, and unnecessary user-space DLLs are also ignored. A configuration module is used to provide different configurations to trace different programs.

There are also times that only the instructions with memory accesses should be traced, because our analysis is based on memory data, and those instructions that have no memory access can be ignored. However, we still have to keep the branch instructions to build high-level representations such as Code Blocks and Call Hierarchy.

Process Tracking Another critical feature that should be provided by the tracer is that only one single process is traced in a multi-process environment. In Windows operating systems, each process has a unique CR3 register value. CR3 register is used to locate page directory address for the current process. We

track a process by filtering the CR3 register value: first, the entry address for each executable is manually obtained, and then whenever CPU runs to the entry point, the current CR3 register value is saved, which is the unique value for the desired process. In this way, we can successfully get rid of the interference of other unrelated processes.

By instruction filtering and process tracking, the trace can be focused on a single process. But still, the number of traced instructions can be quite large, usually 10^6 (100MB data) to 10^7 (1GB data). So the traced result has to be saved to disk for further analysis.

4.2 Lochs: Data Analysis System

The back end of our program analyzing system is called Lochs. Lochs analyzes cryptographic primitives of the traced results of Fochs automatically. There are three stages of data analysis. First, Lochs constructs high-level structures of the traces, including Dynamic Basic Blocks, Code Blocks and Call Hierarchies; and then, data reduction is performed to eliminate the unnecessary data to be analyzed; at last, Lochs does template verifications on the selected data to examine cryptographic algorithms and their parameters. The structure of Lochs is shown in Figure 5.

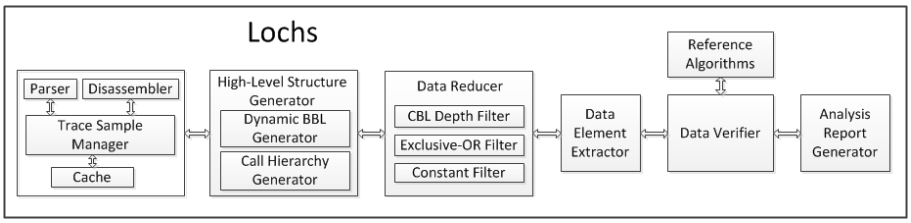

Fig. 5. Lochs program analyzer

High-Level Representations Before data analysis, we have to extract data from the traces first. The points where Intermediate Data States are sampled are critical to our analysis, because we have to select the points where cryptographic data is most likely to appear. We may sample memory data at each instruction trace, but this is obviously impossible to analyze for common traces that contain 10^7 instructions. To solve this problem, we first construct high-level representations for the traces. Three levels of data representation are constructed: Dynamic Basic Blocks, Code Blocks and Call Hierarchies(CBL Lists). The algorithms to build these high-levels are mentioned earlier. These representations are constructed only once and then serialized to or deserialized from disk for future uses.

In the first stage of analysis, trace results are converted from binary data to CBL Lists. These CBL Lists are passed on to the second stage for further processing.

Heuristic Data Reduction A complete trace of a program often contains huge amount of data unrelated to cryptography, even though these data is pre-filtered in the tracing process. These unrelated data may include program initialization, GUI operations, user input handling, error handling, etc. Therefore, a highly-optimized data reduction is performed in the second stage to reduce analysis time. After Call Hierarchies are constructed, heuristic data reductions are conducted on these Call Hierarchies which are represented by CBL Lists. Currently we have mainly three kinds of data reduction methods.

- CBL List Depth
  Functions that contain cryptographic primitives usually have a lower depth. That is, these functions usually have a single purpose, and they are less likely to call other functions because of performance issues. Therefore, cryptographic functions are most likely to appear in the inner CBL Lists, which have a low depth value.
  During the data reduction procedure, we first filter the CBL Lists according to their depth value. A threshold of depth 6 is reasonable to most of the analysis.
- Exclusive-OR Instructions
  Through observations to cryptographic algorithms, the exclusive-or operation is heavily used. The second data reduction method is based on the idea that cryptographic functions should contain a certain percentage of exclusive-or instructions. This heuristic method is applicable because we can safely judge that a function contains no cryptographic primitives if it has no exclusive-or operations.
- Characteristic Constants
  Many cryptographic algorithms and their implementations contain characteristic constants. For example, in the fast implementation of AES, a 1k-size lookup table is commonly used; in hash functions like MD5 and SHA-1, several pre-defined constants are quite unique and must be used. These characteristics are used to filter CBL Lists for a specific algorithm, and those CBL Lists where characteristic constants appear are analyzed first.

Through data reduction, usually more than 90% of total data can be reduced. The reduced data is then passed to algorithm detectors to test if it contains a specific data pattern.

Data Verification In the final stage of analysis, data is extracted from their high-level representations, and algorithm verifiers verify the extracted data to test if it satisfies a certain data pattern. Based on our earlier observation that program functionalities are implemented in the unit of functions, data analysis is conducted on Call Hierarchies. A Call Hierarchy is the representation of an

entire function call, including its data. We compute the input and output for each function call, and verify if these data matches any data pattern of cryptographic algorithms.

First, for each CBL List l, we compute its input data $IN(l)$ and output data $OUT(l)$. Data exists in format of continuous memory chunks, which may contain cryptographic parameters.

Then, for a specific cryptographic algorithm, its data format is pre-defined. Possible data element combinations are extracted recursively, and passed to verifiers to test cryptographic algorithm existence and extract parameters.

At last, the verifier receives data elements, and test if they satisfy a pre-defined pattern. Each verifier implements a reference algorithm. This algorithm can be quite simple (testing constant existence), or rather complicated (AES block encryption). If a group of data elements matches a pre-defined pattern, the parameters are extracted from the data elements, and the detection procedure is successful. We can expand the usage of our system by implementing more reference algorithms, and add them to the Reference Algorithms module in Lochs, as shown in Figure 5. The extensibility of our analysis system is guaranteed by its modular architecture.

5 Evaluation

We do our experiments using real-world applications as well as custom programs. There are five kinds of testing programs:

- *Compression tools*, including RAR 3.93 (AES encryption, SHA-1 hashing) and FreeArc 0.666 (Blowfish encryption)
- *File encryption tools*, including AES Crypt 3.08 (AES file encryption) and TrueCrypt 7.0a (disk formatting using AES encryption)
- *Cryptography softwares*, including Putty 0.60 (login sessions with AES/Blowfish encryption) and KeePass Password Safe 1.19b (password database saving)
- *Custom programs with different implementations*, including AES-OpenSSL (AES 128-bit and 256-bit block cipher), AES-OpenSSL-CBC (AES CBC mode cipher), MD5-OpenSSL (MD5 message digest), SHA1-OpenSSL (SHA-1 message digest), AES-Custom-Impl (a custom implementation of AES), RC4-OpenSSL (RC4 cipher), RC4-Custom-Impl1 (a custom implementation of RC4) and RC4-Custom-Impl2 (another different custom implementation of RC4).
- *Custom programs obfuscated by VMProtect and Themida*, including custom programs with AES, RC4, SHA-1 OpenSSL implementations that both the executable and OpenSSL libraries(libeay32.dll, ssleay32.dll) are obfuscated by VMProtect, and a custom program with AES OpenSSL implementation that only the executable is obfuscated by Themida and the OpenSSL libraries are original.

We implemented 8 reference algorithms, which are: AES 128-bit key expansion/block cipher, AES 256-bit key expansion/block cipher, Blowfish key

scheduling, RC4 key scheduling, MD5 message digest and SHA-1 message digest. The block ciphers take a block of data and a group of expanded keys as input, and a block of data as output. The key expansions and key schedulings take a short key as input, and an expanded key as output. And the message digests take a block of data and an input message digest as input, and an updated message digest as output.

We run all of the reference algorithms on each of the traces of test programs. The testing results and performance analysis are shown in the following sections.

5.1 Accuracy

We successfully discovered the existing pre-known algorithms in all of the testing programs, and extracted the parameters including AES keys, plaintexts and ciphertexts, Blowfish keys, RC4 keys, MD5 input data and SHA-1 input data. There are some flaws that we failed to discover AES block cipher in Putty AES encrypted login session, and the AES block cipher in our custom implementation of AES. We also found a previously unknown SHA-1 algorithm in the first custom implementation of RC4. We also successfully discovered the underlying algorithms as well as the plaintexts, ciphertexts and secret keys in programs obfuscated by VMProtect[14] and Themida[13], and the analysis results are the same as the results without code obfuscation. The analysis results of testing programs are shown in Table 1.

It's shown that our approach can successfully identify the same algorithm with different implementations. For example, one of the two custom implemented versions of RC4 uses a 32-bit memory to store an 8-bit value, while the other uses an 8-bit memory. Another example is that we use a regular implementation of AES as well as a fast implementation, which has optimizations such as table lookup. Our analysis doesn't rely on a specific implementation of a certain algorithm, so both implementations are successfully identified.

Real-world softwares may have countermeasures against this analysis method. For example, continuous memory chunks can be broken into smaller chunks to avoid matching a certain data pattern. However, these countermeasures require specific programming, and we did not find any software that uses such a countermeasure. To cope with these countermeasures, we can use non-perfect matching such as fuzzy matching, which gives a high possibility of the existence of certain algorithms.

False negatives may occur if the software contain countermeasures against this method. In our evaluation, 2 cases only identified AES key expansion process but not the AES encryption. One of the possible reasons is that our analysis optimization ignored deeper function calls which contain the encryption process. We can refine the optimization stage to resolve this issue. Because of the uniqueness of cryptographic data, false positives can be very rare. Real-world softwares hardly contain cryptographic data in both the input and output of a function call, which actually has no cryptographic primitive. We found no false positives during our analysis.

Table 1. The Test Results

	algorithm(s)	key expansion/key scheduling
RAR 0	SHA-1	
RAR 1	AES(128-bit) encryption	AES-128 Key Expansion
FreeArc	-	Blowfish Key Scheduling
AES Crypt	AES(256-bit) encryption	AES-256 Key Expansion
TrueCrypt	-	AES-256 Key Expansion
Putty(AES)	-	AES-256 Key Expansion
Putty(Blowfish)	-	Blowfish Key Scheduling
KeePass AES	AES(256-bit) encryption	AES-256 Key Expansion
AES128-OpenSSL-ECB	AES(128-bit) encryption	AES-128 Key Expansion
AES256-OpenSSL-ECB	AES(256-bit) encryption	AES-256 Key Expansion
AES128-OpenSSL-CBC	AES(128-bit) encryption	AES-128 Key Expansion
AES-Custom-Impl	AES(128-bit) encryption	AES-128 Key Expansion
RC4-OpenSSL RC4	-	RC4 Key Scheduling
RC4-Custom-Impl1	-	RC4 Key Scheduling
RC4-Custom-Impl2	-	RC4 Key Scheduling
MD5-OpenSSL	MD5	-
SHA1-OpenSSL	SHA1	-
AES128-VMProtect	AES(128-bit) encryption	AES-128 Key Expansion
AES256-VMProtect	AES(256-bit) encryption	AES-256 Key Expansion
RC4-VMProtect	-	RC4 Key Scheduling
SHA1-VMProtect	SHA1	-
AES256-Themida	AES(256-bit) encryption	AES-256 Key Expansion

These test results demonstrated that our analysis is successful, in both real-world applications and custom implemented programs, and can be used to analyze obfuscated code.

5.2 Performance

The tracing process is usually manually operated, and the tracing time is trivial and can be ignored. The later analyzing process is fully automated, and the time of each analyzing stage is recorded and listed below, where Stage 1 represents the construction of high-level representations(Dynamic BBL, CBL, and CBL List), and the algorithm names represent time used to analyze each algorithm.

The performance evaluation is shown in Table 2, including the total number of instructions and file size of each trace, the analysis speed (in instructions per second), time used for each stage and algorithm, and the total time used. The results show that the average file size is about 500MB(5M instructions), the average analysis speed is about 15k instructions per second, and the analysis time is usually within or around 10 minutes.

The performance results also show that the most time consuming part is the constructions of high-level representations, and that analysis for RC4 is most time consuming among all these algorithms, because there is no constant heuris-

Table 2. Performance

	Instructions	Size	Speed (instrs/sec)	Total Time
RAR 0	919k	77MB	19.4k	47s
RAR 1	1,359k	114MB	9.6k	2m22s
FreeArc	7,786k	653MB	34.6k	3m45s
AES Crypt	2,396k	201MB	14.1k	2m50s
TrueCrypt	12,800k	1,074MB	15.0k	14m11s
Putty (AES)	3,651k	306MB	14.5k	4m11s
Putty (Blowfish)	7,297k	612MB	18.2k	6m42s
KeePass	9,005k	755MB	22.5k	6m40s
AES-OpenSSL-128	5k	467KB	9.2k	$<$ 1s
AES-OpenSSL-256	6k	500KB	7.9k	$<$ 1s
AES-OpenSSL-CBC	6k	510KB	20.3k	$<$ 1s
AES-Custom-Impl	11,294k	947MB	17.7k	10m36s
RC4-OpenSSL	10k	840KB	14.5k	$<$ 1s
RC4-Custom-Impl1	298k	25MB	36.3k	8s
RC4-Custom-Impl2	10,078k	845MB	16.5k	10m13s
MD5-OpenSSL	4k	383KB	16.8k	$<$ 1s
SHA1-OpenSSL	5k	465KB	23.7k	$<$ 1s
AES128-VMProtect	6k	549KB	20.9k	$<$ 1s
AES256-VMProtect	7k	581KB	6.6k	1s
RC4-VMProtect	10k	869KB	21.8k	$<$ 1s
SHA1-VMProtect	7k	619KB	31.2k	$<$ 1s
AES256-Themida	20k	1MB	14.4k	1s

tic data reduction for RC4. The time spent for other algorithms is almost the same.

6 Conclusion

In this paper we have presented a novel approach of analysis of cryptographic data. We use a two-stage method to trace and analyze program data. First, dynamic data tracing is conducted based on full system emulation. The trace results are saved for further analysis. Then, we use an automatic analyzer to perform cryptographic data analysis on the trace results. The target of our analysis is to identify cryptographic algorithms and to extract their parameters. We studied the data patterns for symmetric ciphers AES, Blowfish, stream cipher RC4, and cryptographic hash functions MD5 and SHA-1, and implemented their reference algorithms. In the analysis phase, the high-level representations of traces are first constructed, including Dynamic Basic Blocks, Code Blocks and Call Hierarchies. Then, heuristic data reductions are conducted to reduce the size of data to be analyzed. And at last, we use the reference algorithms to verify the existence of certain algorithms and extract their parameters.

It is possible to extend our analysis method to asymmetric cryptographic algorithms. For example, it's possible to identify RSA encryptions/decryptions

which comply with PKCS formats. However the analysis process can be much slower, because the asymmetric algorithms usually take much longer time than symmetric algorithms. It's quite difficult to identify custom asymmetric algorithms because of the irregularity of asymmetric cryptographic data.

We did our experiments on 22 Windows programs, including both real world applications and custom implemented programs, and some of them are obfuscated using VMProtect or Themida. We successfully identified the existing cryptographic algorithms in these programs, and extracted the keys or input data of these programs. For most of the programs that contain symmetric cipher we also extracted the plaintext and ciphertext. The programs with code obfuscation are also successfully analyzed. The analysis result showed the universality and effectivity of our analysis method.

References

1. S. Bhansali, W. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, page 163. ACM, 2006.
2. J. Caballero, H. Yin, Z. Liang, and D. X. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007, pages 317-329, 2007.
3. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, pages 321-336. USENIX, 2004.
4. W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391-402. ACM New York, NY, USA, 2008.
5. De-obfuscating the RC4 layer of Skype. *http://lukenotricks.blogspot.com/2010/08/de-obfuscating-rc4-layer-of-skype.html*
6. Findcrypt plugin. *http://www.hexblog.com/ida_pro/files/findcrypt2.zip*
7. Gröbert, F. Automatic Identification of Cryptographic Primitives in Software. Diploma Thesis, Ruhr-University Bochum, 2010.
8. D. Janssens. Heuristic methods for Locating Cryptographic Keys Inside Computer Systems. PhD thesis, Katholieke Universiteit Leuven, 1999.
9. D. Janssens, R. Bjones, and J. Claessens. KeyGrab TOO - The search for keys continues...,. Whitepaper, Utimaco Safeware AG, KU Leuven, 2000.
10. 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190-200. ACM New York, NY, USA, 2005.
11. C. Maartmann-Moe, S. Thorkildsen, and A. Årnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. Digital Investigation, 6:132-140, 2009.

12. A. Shamir and N. Van Someren. Playing Hide and Seek with Stored Keys. *Lecture Notes in Computer Science*, 1999.
13. Themida - Oreans Technology : Software Security Defined. *http://www.oreans.com/themida.php*
14. VMProtect Software Protection. *http://vmpsoft.com*