Detecting Encryption Functions via Process Emulation and IL-Based Program Analysis*

Ruoxu Zhao, Dawu Gu, Juanru Li, and Hui Liu

Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
dwgu@sjtu.edu.cn

Abstract. Malware often encrypts its malicious code and sensitive data to avoid static pattern detection, thus detecting encryption functions and extracting the encryption keys in a malware can be very useful in security analysis. However, it's a complicated process to automatically detect encryption functions among huge amount of binary code, and the main challenge is to keep high efficiency and accuracy at the same time. In this paper we propose an enhanced detection approach. First we designed a novel process level emulation technique to efficiently analyze binary code, which is less resource-consuming compared with full system emulation. Further, we conduct program partitioning and assembly-to-IL(intermediate language) translation on binary code to simplify the analysis. We applied our approach to sample programs using cryptographic libraries and custom implemented version of typical encryption algorithms, and showed that these routines can be detected efficiently. It is convenient for analysts to use our approach to deal with the encrypted data within malware automatically. Our approach also provides an extensible interface for analysts to add extra templates to detect other forms of functions besides encryption routines.

Keywords: Encryption detection, Process emulation, Intermediate language, Binary code analysis.

1 Introduction

Recent years have witnessed a dramatic rise in the growth of work on automatically detecting certain algorithms in programs especially in malware. In order to solve the problem of algorithm detection, a number of approaches were proposed, and most of them are mainly heuristic[10][9][6]. However, despite an increasing interest in algorithm identification in binary programs, in particular in detecting cryptographic primitives, there still lacks systematic and convenient approaches that facilitate researchers to perform efficient detection.

* Supported by the National Science and Technology Major Projects 2012ZX03002011-002.

T.W. Chim and T.H. Yuen (Eds.): ICICS 2012, LNCS 7618, pp. 252–263, 2012.

We present a generic encryption function detecting approach using *Process Emulation* and *IL(intermediate language)-based Program Analysis*, which is targeted at achieving fast, convenient and extensible detection. The basic principles behind our technique are stripping unnecessary runtime information, simplifying analysis process and providing interface for new extensions. First, we designed and implemented our own process emulator to reduce the overhead brought by emulating full system environment. Then we adopted a custom defined IL to simplify analyzed program. Based on this IL, not only we designers but also other analysts could easily write a template to match certain algorithms. And finally, we combined IL-based template matching and dynamic data verification to improve the accuracy and efficiency of encryption routines identification.

Some of the contributions of this work are listed below.

- *Lightweight process emulation.* We designed process emulation, a novel emulation technique, to run a program within its host operating system, and only emulate the necessary components of a system for the program to be analyzed. This technique provides a lightweight emulation environment with fast speed while keeping fine-grained analyzing capability.
- *IL-based program transformation.* To address the issues of dynamic program pattern matching and analysis, we further extended detection method by introducing intermediate language as analyzing medium, increasing its efficiency and accuracy, and acquiring platform compatibility at the same time.
- *Flexible template matching.* We provided an open interface for analysts to write template of different algorithms in IL form. Our emulator dynamically loads templates during the detection phase and uses template to construct heuristics.
- *Template based data filtering and verification.* Traditional matching approaches may verify all runtime data, and meanwhile test huge amount of unrelated data. Our IL based analyzer first matches code fragments with templates and filters out those data of mismatching code fragments. Then, a data verifier is designed to check matched data and deploy refined input-output verification. The process not only improves verification efficiency significantly, but also reduces false positive rate to negligible level.

The rest of the paper is structured as following. Section 2 gives an overview of algorithm detection problem and related work. Section 3 describes our approach in detail. Section 4 gives concrete instance of template based encryption function detection and evaluation results. Some countermeasures to our approach are discussed in section 5 and an overview about future work is given. And section 6 concludes this paper.

2 Problem Statement

Encryption function detection is a problem of searching certain algorithms in programs especially in binary code. This work is based on the following assumptions: (1) The knowledge of the algorithm is obtained before detection; (2)

The implementation of the algorithm is not aimed at failing the detection deliberately. These assumptions are reasonable in the real world for the following reasons. First, it is always prudent to adopt mature encryption algorithms for the consideration of security, and these mature encryption algorithms are generally public and are tested for a long term. So we suppose that the precondition of detecting an encryption algorithm is knowing its details. Second, In most cases, the purpose of the encryption algorithms in malware is to protect malicious code and to hide sensitive data. Thus these encryption algorithms are often implemented without being obfuscated or packed in order to provide accuracy and efficiency.

Previous detection methods generally take advantage of certain properties of an algorithm as the signature. Caballero et al.[3] took advantage of the fact that encryption routines use a high percentage of bitwise arithmetic instructions. The approach of Groebert el al.[4] was based on both generic characteristics of cryptographic code and signatures for specific instances of cryptographic algorithms. Zhang et al.[12] proposed an algorithm plagiarism detection approach using critical runtime values. And Zhao et al.[13] used input-output correlation of certain ciphers to detect cryptographic data.

There are several reasons why proposing new detection techniques is necessary to current security analysis.

- Existing approaches usually use tools such as QEMU[2] and PIN[8] to trace data and instructions. And these tools don't have satisfactory performance. Actually, Groebert el al.[4] reported that for a malware analysis process the tracing took 14 hours and the analysis phase 8 hours.
- Existing approaches are not extensible. That is to say, analysts can't easily adjust these specific approaches to either adapt different implementations of algorithms or to detect new ones.
- Taking traced instructions alone as input is not enough to acquire effective heuristics. For dynamic data based detection, the main problem is how to filter out useless data according to heuristics.

In contrast to previous work in this area, the goal of our work is to design extensive, convenient and efficient detection approach. We argue that a new approach for efficient tracing is necessary. And because the data feature related to algorithm is very important for heuristics, it is suggested to combine instructions and data together to acquire powerful heuristics. What's more, a simple form of program is able to improve analyzing efficiency and help an analyst deploy her own detection. We improved the detection approach in two aspects: one is to perform a high speed program tracing using process emulation, and the other one is to translate program into IL to simplify construction of heuristics and third-party matching extension design. In addition, our approach verifies the matching result with input-output data correlation to reduce the chance of false positive, and to extract the input and output parameter(e.g., the secret key) at the same time.

3 Our Approach

Our approach adopts a hybrid methodology combining code characteristic matching and data input-output verification. To make dynamic analysis possible, the program we're trying to analyze is first executed in an emulation environment, and low-level runtime data is acquired in this step. Then, traced instructions are partitioned to fragments, and translated to our IL representations. For each block of program, fuzzy matching techniques which are inaccurate but fast are used with existing algorithm templates implemented in IL. And finally, dynamic data verification is conducted to identify the correct algorithm and extract parameters.

3.1 Process Emulation

A full-system emulator, such as Bochs[7], often emulates a set of fully functional hardware, and runs an operating system on the emulated hardware. It usually runs as a user-space process in the host operating system. A program to be analyzed runs in the emulated operating system, where non-privileged and privileged instructions are all executed in a software emulated environment.

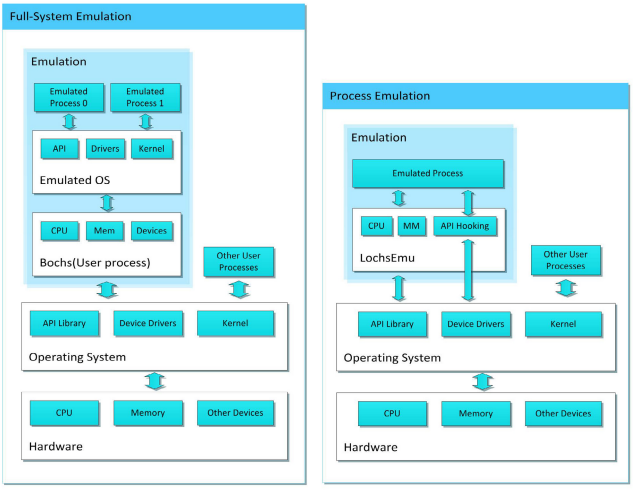

Fig. 1. Comparison of Full System Emulation and Process Emulation

Because of the nature of instruction emulation, full-system emulators often have a poor performance. Through actual tests, we found that Bochs emulator runs 10^2 slower than non-emulated environment. To emulate a single instruction, we often need tens even hundreds of actual instructions, which considerably impacts the runtime performance of a full-system emulator.

In program analysis using full-system emulation, we see that the guest (emulated) operating system and the host operating system are usually the same,

and the OS specific operations, such as process context switch, are trivial to our analysis. Therefore, we came up a program emulation proposal that directly emulates the target program on host operating system, which we called process emulation.

Being different from full-system emulation, process emulation directly uses the host operating system to provide OS-specific features, such as handling system API calls. This assumption requires the guest OS and the host OS to be the same. The process emulator is a user-space application that can emulate other user-space applications, where CPU instruction execution, memory management and some OS features are emulated by the process emulator, and system calls/APIs are executed by the host operating system. The comparison of full system emulation and process emulation is shown in figure 1.

One advantage of process emulation is that all system API calls are hooked by the emulator. Hence, sandboxing can be easily achieved and malware can be run directly on the emulator, preventing the malware from interfering with the real OS.

3.2 Program Partitioning

The first step of analysis after program tracing is program partitioning, where sequential instructions traced from process emulation are partitioned into basic blocks or program segments. The goal of this stage is to make partitioned segments the same scale as an algorithm implementation. In static analysis, it is possible to reconstruct the whole control flow graph or function call hierarchy, but in dynamic analysis however, it's usually impossible to obtain the complete image of a program, because we cannot get through all execution paths in one time of execution. Whenever a conditional branch is met, only the determined path is executed, so we cannot build a complete control flow graph through limited execution traces. Hence, partitioning the program into appropriate scale at appropriate point is crucial to the follow-up steps. We develop some partitioning algorithms with different granularity, including basic blocks, inter-procedure, procedure call, etc.

3.3 Intermediate Language

Dynamic program tracing usually produces low-level, fine-grained program data, including processor register values, memory access values, etc. The fact that our IL is designed to be close to machine language makes translation from tracing result to IL can be done with the lowest cost. The instruction set of our IL is highly reduced as well, which helps to increase template matching performance, and grasp primary runtime information at the same time. In this way, we build up an IL that is light-weighted, platform-compatible and easy to analyze, and is used in each step of analysis, including dynamic translation, template algorithm implementation, program matching and dynamic data verification. The structure of an IL template is shown in figure 2.

```
VAR_DESC     ::= 'var'    VAR_NAME ':' VAR_TYPE
                 (',' VAR_NAME ':' VAR_TYPE)* ;
INPUT_DESC   ::= 'input'   VAR_NAME (',' VAR_NAME)* ;
OUTPUT_DESC  ::= 'output'  VAR_NAME (',' VAR_NAME)* ;

PROGRAM      ::= INSTRUCTION+ ;
INSTRUCTION  ::= UNARY_INSTRUCTION | BINARY_INSTRUCTION |
                 ASSIGNMENT_INSTRUCTION | BRANCH_INSTRUCTION |
                 CONDITIONAL_BRANCH_INSTRUCTION |
                 ARRAY_INSTRUCTION ;

UNARY_INSTRUCTION      ::= LVALUE '=' UNARY_OP RVALUE ;
BINARY_INSTRUCTION     ::= LVALUE '=' RVALUE BINARY_OP RVALUE ;
ASSIGNMENT_INSTRUCTION ::= LVALUE '=' RVALUE ;
BRANCH_INSTRUCTION     ::= 'jmp' LABEL ;
CONDITIONAL_BRANCH_INSTRUCTION   ::=
                 'jmp' LABEL '[' RVALUE BRANCH_OP RVALUE ']' ;
ARRAY_INSTRUCTION      ::= ARRAY_GET | ARRAY_SET | ARRAY_COPY ;

VAR_TYPE     ::= int32 | int64 | array | string ;
LVALUE       ::= VAR_NAME ;
RVALUE       ::= VAR_NAME | CONSTANT ;
UNARY_OP     ::= 'length' | 'append' | 'remove' | '~' ;
BINARY_OP    ::= '+' | '-' | '^' | '&' | '|' | '*' | '/' |
                 '%' | '>>' | '<<' | '>>>' | '<<<' ;
BRANCH_OP    ::= '>' | '>=' | '<' | '<=' | '==' | '!=' ;
```

Fig. 2. IL Program Template

3.4 Assembly-to-IL Translation

In the translation step, binary instructions are translated into IL instructions. This is usually done after program partitioning, because the translation may lose information about original program context. The translation is not accurate, which means that some irrelevant information is discarded. For example, the zero flag in x86 architecture indicates if an arithmetic operation produces a zero value[5], and most of the time, we don't care if the value is zero, so the value of zero flag is discarded in the translation.

Selection of Instructions. Not all traced instructions are translated into IL instructions. Normally, translations are limited to these categories: arithmetic, logical, bitwise, data transferring, control-flow transferring, etc.

Memory Access. Data used in program execution is usually contained in memory. For almost every algorithm implementation, its input and output parameters are first stored in memory, then displayed on the screen or stored in a file. By identifying memory reading and writing, we can generate dynamic inputs and outputs of a program, and perform data verification in later stage. We treat the memory as a global array object, and memory reading and writing are translated into array getting and setting at the index specified by the address of memory access.

Data Preservation. The advantage of dynamic program analysis is that we have direct access to runtime data which is unavailable in static analysis. Each instruction in IL program segment has an optional field that stores its original context, including memory access values, instruction pointer, etc.

3.5 Template Matching

In template matching step, IL segments are matched to template IL programs using fuzzy matching techniques, and the matched segments are further verified in data verification step.

A template program is an implementation of a certain algorithm written in IL code, which can be executed in IL interpreter and has explicit input and output format. A dynamic translated IL program segment, on the other hand, contains an incomplete translation of traced instructions, and usually cannot be executed in the interpreter. Also, it contains runtime data of the original program, which is different from IL templates. Template matching is done in IL-instruction level or IL control-flow level(control flow information is contained in dynamic translated program), and is controlled by a posteriori threshold, which defines the matching accuracy.

In template matching step, efficiency is usually more important than accuracy. Previous research of data pattern matching showed that analyzing large amount of irrelevant data is the bottleneck of dynamic data analysis. Hence, the main purpose of template matching is that we can filter out most of the impossible traced result with little cost. To keep a low false-negative rate, we should only filter out the "obviously impossible" segments. Fortunately in most cases, most of the dynamic translated segments satisfy such a condition. We designed some fine-tuned template matching algorithms, including direct mapping, instruction frequency, CFG matching and scale predicting.

3.6 Dynamic Data Verification

In this step, all input and output data is first extracted from the program segment. We assume that all the data needed for analysis is stored in memory, and we define the input data as the memory values first referenced by memory reading, and the output data as the memory values last set by memory writing. This data is then further processed into memory chunks, according to its memory offset(address).

Next, we try to construct possible algorithm parameters from the memory chunks. We use some heuristic techniques to eliminate low-priority data, such as pointer values, all-zero (initializing) values, etc. Each possible set of parameters is in turn injected into the IL interpreter.

Then, after injection of parameters, the IL interpreter executes the template program to produce output results. Each output result is then verified in the program segment's output data, and if a matching is found, we confirm that the implementation of a certain algorithm exists in a program segment. The workflow of data verification is shown in figure 3.

We can see that we don't have to know the exact implementation of the algorithm we're trying to analyze. We just have to provide one template implementation, and the data verification will test if they are the same.

False positives are highly unlikely to happen when the input and output parameters reach a certain length, say 128-bit. We may take the AES encryption

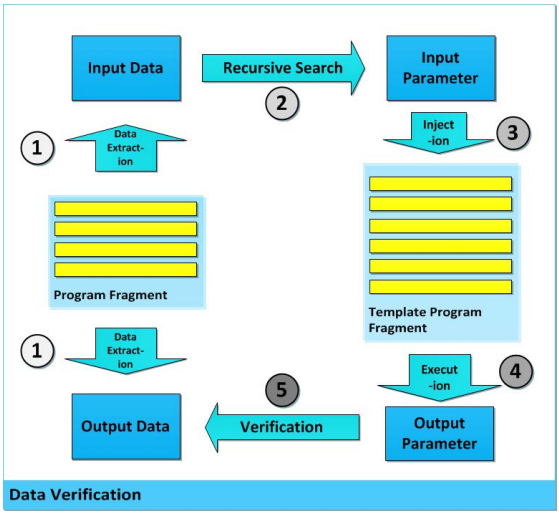

Fig. 3. Dynamic Data Verification

as an example. Each 128-bit input block is encrypted into an 128-bit output block, and whenever the correct 128-bit data shows up in a program segment's runtime data, we may safely say that it contains an AES encryption, because the implementation is similar to the template, which is verified in the template matching step, and the corresponding data is correct, verified in this step.

The data verification step tells us if an algorithm implementation truly exists in the original program, and extracts its corresponding parameter, completing the analysis.

4 Experiment and Evaluation

We choose several custom programs as well as common cryptography libraries such as OpenSSL[11] to implement cryptographic algorithms, and use them as testing programs for proof-of-concept evaluation of accuracy, effectiveness and performance. The cryptographic algorithms we use include AES(128-bit and 256-bit), RC4, MD5, SHA1 and SHA2, and the implementations of the same algorithm are different and independent. The AES implementations are from the original Rijndael implementation, OpenSSL library and Nettle[1] crypto library, the RC4 implementation is custom, and the hash functions are from OpenSSL library. A primary result of all testing programs is shown in table 1.

Partitioning Strategy. In experiments, we use the inter-procedure partitioning as the main program partitioning method. The basic-block partitioning can hardly satisfy the structure of template algorithms, since template algorithms often have many basic blocks and a complicated control-flow. The

Table 1. The Test Results

Binary	Algorithm	Algorithm Detected	Description
aes_std.exe	AES 128-bit	aes_subkey_be	Original AES implementation
aes_nettle.exe	AES 128-bit	aes_subkey_le	Nettle crypto library
aes_ssl.exe	AES 128-bit	aes_key	OpenSSL library
aes256_ssl.exe	AES 256-bit	aes_key_256	OpenSSL library
rc4_custom.exe	RC4	rc4_key	Custom implementation
md5_ssl.exe	MD5	md5_core	OpenSSL library
sha1_ssl.exe	SHA1	sha1_core	OpenSSL library
sha2_ssl.exe	SHA2	sha2_core	OpenSSL library

procedure partitioning tracks a whole function call in one partition, which has a huge memory consumption, and is difficult to achieve an acceptable performance because of the vast amount of data. The inter-procedure partitioning satisfies all our needs, as it can get the appropriate partition scale, and the fact that it has no memory need makes the analysis can be done simultaneously with tracing.

Matching Algorithms. We find that complicated matching algorithms are not necessary in our analysis, and we primarily combine the instruction frequency and scale predicting as the matching algorithm. Direct mapping algorithm has complexity of $O(n^2)$,which is too slow for fast but inaccurate matching. Instruction frequency has complexity of $O(n)$, and in actual experiments it can distinguish matching program segments from other segments quite well. CFG matching algorithm is unavailable in most circumstances, since the CFG of a segment is not always available in one dynamic execution. And finally, the scale predicting turns out to be very effective. It has the complexity of $O(1)$ and can efficiently identify those segments which are too large or too small for a template program. The matching similarity is combined from all matching algorithms, and mapped to [0, 1]. We use a threshold of 0.95 in all experiments, and then the non-matching segments usually have similarity of less than 0.90. We list each of the matching similarity in table 2.

Table 2. Matching Results

Binary	Algorithm	Algorithm Detected	Similarity
aes_std.exe	AES 128-bit	aes_subkey_be	0.9650
aes_nettle.exe	AES 128-bit	aes_subkey_le	0.9713
aes_ssl.exe	AES 128-bit	aes_key	0.9610
aes256_ssl.exe	AES 256-bit	aes_key_256	0.9574
rc4_custom.exe	RC4	rc4_key	0.9585
md5_ssl.exe	MD5	md5_core	0.9527
sha1_ssl.exe	SHA1	sha1_core	0.9577
sha2_ssl.exe	SHA2	sha2_core	0.9652

Performance. The performance evaluation includes both program emulation (tracing) and analysis. As tracing and analysis are done at the same time, we run each testing program twice, one with analysis and one without analysis. The performance result is shown in table 3. The tracing time is usually trivial in each program execution comparing to the analysis. We see that program tracing takes less than 1 second, which is much faster than whole system emulation(booting Bochs alone will take about 5 minutes, and tracing is also slower). During analysis, the dynamic data verification step is the most time-consuming one, because there's a lot of data to be verified by IL template, which is run by the IL interpreter. Improper program partitioning and template arguments can severely slow down this step, as a large segment can produce much irrelevant data, and a small length of template input argument will heavily increase the number of times in searching, thus burdening the data verification. We also tested the effectiveness of template matching, and found that analysis took 10 to 50 times longer without template matching. Besides, these is no acceleration in the IL interpreter, which also downgrade the analysis. Despite all this, the average analysis(including tracing) speed is 167 kIPS(instructions per second), which is more than 10 times faster than the previous result of 15 kIPS(excluding tracing). Such performance result is quite acceptable considering there's no optimizations in this proof-of-concept evaluation.

Table 3. Performance Evaluation

Binary	Time(trace)	Time(analysis)	Time(total)	Insts	kIPS
aes_std.exe	0.013(s)	3.712	3.725	103,757	27.854
aes_nettle.exe	0.068	21.395	21.463	808,828	37.684
aes_ssl.exe	0.025	0.822	0.847	230,241	271.831
aes256_ssl.exe	0.025	0.847	0.872	234,680	269.128
rc4_custom.exe	0.051	1.775	1.826	459,642	251.720
md5_ssl.exe	0.016	0.422	0.438	147,256	336.200
sha1_ssl.exe	0.065	0.512	0.577	36,018	62.422
sha2_ssl.exe	0.011	0.745	0.756	57,507	76.067

5 Discussion

5.1 Countermeasures

Our method may produce false negatives when used against protected code or custom implementations of an algorithm. In these conditions, the original structure of an algorithm is sabotaged, and then failing our analysis. We discuss these conditions in details, and possible counterattacks against these conditions.

Anti-emulation. Malware may use anti-emulation techniques to protect from being analyzed. These techniques are usually small hacks or tricks which

utilize bugs or incompleteness of the emulator. By fixing bugs and improving the completeness of emulation, we can overcome most of the anti-emulation techniques.

Code Obfuscation. Many malware authors use code obfuscation techniques to protect their program from being detected. Obfuscation usually transforms the instruction flow and control flow of a program, which compromise the ability of matching template algorithms in our analysis. Hence, our analysis method cannot be used against strong code obfuscation(such as VM obfuscation). However, with a few changes, we may make our analysis method invulnerable to code obfuscation. We see that data integrity can be preserved even in obfuscation, we just have to modify the partitioning and matching algorithms. The first and simplest modification is to lower the threshold of template matching. Many simple obfuscators use instruction transforms to confuse analysts, but the fundamental meaning of a program remains unchanged. As our matching is not 100% accurate, we just have to enlarge the tolerance of the similarities between a program segment and an algorithm template. To deal with strong obfuscation which usually uses virtual machine protection, we may try carefully select the representing set of instructions to be translated into IL code. For example, a virtual machine obfuscator may translate a single DIV instruction into its own VM representation. During interpretation of the VM representation, such DIV instruction will eventually be executed by the same or a similar instruction. We may select a set of instructions that are rarely used by internal logic part of a VM obfuscator, and in this way we can still use instruction frequency as a valid matching algorithm.

Custom Implementations. A malware author may use a non-standard version of standard algorithm. For example, one may change the constants in a cryptographic algorithm, producing a similar but different algorithm. Such modifications will bypass the data verification part of our analysis, as the detected algorithm produced a different result. This issue may be addressed by considering the constants in a algorithm as input arguments, and keeping only the computations in the algorithm templates. Some developers may break an algorithm into small parts, and such an implementation cannot be detected using a whole algorithm. We may also try split a template algorithm into small blocks, but doing so will certainly increase the running time of analysis.

6 Conclusion

In this paper, we presented a novel program analysis technique using process emulation and IL-based analysis which is fast and extensible. We tested the effectiveness and accuracy using custom programs implemented with common cryptographic libraries. The result showed that we could identify encryption or hashing functions within these programs, and extract the corresponding input and output data of these functions. The performance evaluation shows that

program tracing and analysis could be done within acceptable time, usually less than one minute for small-scale programs, which is superior to most existing analysis techniques.

We further studied possible countermeasures against our technique, and future improvements of our technique. We showed that these countermeasures could be solved by strengthening our system and refining algorithms. We plan to develop new program matching algorithms which may concern data characteristics, and further improve the performance of our technique.

References

1. Nettle: a low-level crypto library (last visited, 2012), `http://www.lysator.liu.se/~nisse/nettle/`
2. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
3. Caballero, J.: Binary code extraction and interface identification for security applications. Technical report, DTIC Document (2009)
4. Gröbert, F., Willems, C., Holz, T.: Automated Identification of Cryptographic Primitives in Binary Programs. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 41–60. Springer, Heidelberg (2011)
5. Intel. Intel 64 and ia-32 architectures software developers manual. intel, `http://www.intel.com/products/processor/manuals` 64
6. Jhi, Y.C., Wang, X., Jia, X., Zhu, S., Liu, P., Wu, D.: Value-based program characterization and its application to software plagiarism detection. In: Proceeding of the 33rd International Conference on Software Engineering, pp. 756–765. ACM (2011)
7. Lawton, K.P.: Bochs: A portable pc emulator for unix/x. Linux Journal 29es, 7 (1996)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Notices, vol. 40, pp. 190–200. ACM (2005)
9. Oksanen, K.: Detecting algorithms using dynamic analysis. In: Proceedings of the Ninth International Workshop on Dynamic Analysis, pp. 1–6. ACM (2011)
10. Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., Su, Z.: Detecting code clones in binary executables. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 117–128. ACM (2009)
11. OpenSSL: The Open Source toolkit for SSL/TLS (last visited, 2012), `http://www.openssl.org/`
12. Zhang, F., Jhi, Y.C., Wu, D., Liu, P., Zhu, S.: A first step towards algorithm plagiarism detection (2011)
13. Zhao, R., Gu, D., Li, J., Yu, R.: Detection and analysis of cryptographic data inside software. Information Security, 182–196 (2011)