Automated Identification of Cryptographic Primitives in Binary Programs

Conference Paper · September 2011

DOI: 10.1007/978-3-642-23644-0_3 · Source: DBLP

Carsten Willems
VMRay GmbH
30 PUBLICATIONS **2,185** CITATIONS

Thorsten Holz
Ruhr-Universität Bochum
234 PUBLICATIONS **9,452** CITATIONS

Automated Identification of Cryptographic Primitives in Binary Programs

Felix Gröbert[1], Carsten Willems[2], and Thorsten Holz[1]

[1] Horst Görtz Institute for IT-Security, Ruhr-University Bochum
[2] Laboratory for Dependable Distributed Systems, University of Mannheim

Abstract. Identifying that a given binary program implements a specific cryptographic algorithm and finding out more information about the cryptographic code is an important problem. Proprietary programs and especially malicious software (so called *malware*) often use cryptography and we want to learn more about the context, e.g., which algorithms and keys are used by the program. This helps an analyst to quickly understand what a given binary program does and eases analysis.
In this paper, we present several methods to identify cryptographic primitives (e.g., entire algorithms or only keys) within a given binary program in an automated way. We perform fine-grained dynamic binary analysis and use the collected information as input for several heuristics that characterize specific, unique aspects of cryptographic code. Our evaluation shows that these methods improve the state-of-the-art approaches in this area and that we can successfully extract cryptographic keys from a given malware binary.

Keywords: Binary Analysis, Malware Analysis, Cryptography

1 Introduction

Analyzing a given binary program is a difficult and cumbersome task: an analyst typically needs to understand the assembly code and interpret it to draw meaningful conclusions from it. An attacker can hamper the analysis attempts in many ways and take advantage of different code obfuscation techniques [10, 15]. A powerful way to complicate analysis is *cryptovirology* [23], i.e., using cryptography in a program such that specific activities are disguised. The following list provides a few recent examples of real-world malware which use cryptography in one form or another:

- Wang et al. analyzed a sample of *Agobot*, which uses SSL to establish an encrypted IRC connection to a specific server [20].
- Caballero et al. showed that *MegaD*, a malware family communicating over TCP port 443, uses a custom encryption protocol to evade network-level analysis [2].
- Werner and Leder analyzed *Conficker* and found that this malware uses the OpenSSL implementation of SHA-1 and a reference implementation of MD6.

Interestingly, the attackers also later patched the MD6 implementation in a malware update to fix a buffer overflow in the MD6 reference implementation [7]. Furthermore, Porras et al. found that the malware authors use RSA with 1024 bits for signature verification [16], in newer versions the attackers even use RSA with a 4096 bit key.

- Werner and Leder also analyzed the Waledac malware [21]. About 1,000 of the 4,000 functions used by Waledac have been borrowed from OpenSSL. Furthermore, AES in CBC Mode with an IV of zero is used. The self-signed RSA client certificates are used in a key exchange protocol.
- Stewart analyzed the algorithms used by the Storm Worm malware [17]. For the peer-to-peer and fast-flux communication, the malware uses a static XOR algorithm for subnode authentication and a RSA key with 56 bits [5].

An analyst needs to manually identify the cryptographic algorithms and their usage to understand the malicious actions, which is typically time-consuming. If this task can be automated, a faster analysis of malware is possible, thus enabling security teams to respond quickly to emerging Internet threats. In this paper, we study the problem of identifying the type of cryptographic primitives used by a given binary program. If a standardized cryptographic primitive such as AES, DES, or RC4 is used, we want to identify the algorithm, verify the instance of the primitive, and extract the parameters used during this invocation.

This problem has been studied in the past, for example by Wang et al., who introduced a heuristic based on changes in the code structure when cryptographic code is executed [20]. This heuristic has been improved by Caballero et al., who noted that encryption routines use a high percentage of bitwise arithmetic instructions [2]. While these approaches are useful to detect cryptographic code, we found that they sometimes miss code instances and also lead to false positives. In this paper, we thus introduce improved heuristics based on both generic characteristics of cryptographic code and on signatures for specific instances of cryptographic algorithms. In contrast to previous work in this area, we improve the heuristics to perform a more precise analysis and also extract the parameters used by the algorithm, which significantly reduces the manual overhead necessary to perform binary analysis.

In summary, this paper makes the following primary contributions:

- We introduce novel identification techniques for cryptographic primitives in binary programs that help to reduce the time a software analyst needs to spend on determining the underlying security design.
- We have implemented a system that allows the automated application of our technique by utilizing a dynamic binary instrumentation framework to generate an execution trace. The system then identifies the cryptographic primitives via several heuristics and summarizes the results of the different identification methods.
- We demonstrate that our system can be used to uncover cryptographic primitives and their usage in off-the-shelf and packed applications, and that it is able to extract cryptographic keys from a real-world malware sample.

2 Related Work

2.1 Static Approaches

All static tools we tested use signatures to determine whether a particular, compiled implementation of a cryptographic primitive is present in a given binary program. A signature can match a x86 assembly code snippet, some "magic" constants of the algorithm, structures like S-boxes, or the string for an import of a cryptographic function call. If a signature is found, the tools print the name of the primitive (e.g., DES or RSA), and optionally the implementation (e.g., OpenSSL or the name of the reference implementation).

We evaluated six publicly available tools using a set of 11 testing applications for different cryptographic primitives. All analysis tools claim to be able to detect the listed algorithms. In Table 1, we summarize the performance of the tools. A + sign denotes that the tool has found the applications's algorithm, while a - sign denotes that the tool has not found the specific algorithm. Overall, none of the tools was able to detect all cryptographic primitives and further tests showed that most tools also generate a significant number of false positives. Furthermore, it is in general hard to statically analyze malware [13] and an attacker can easily obfuscate his program such to thwart static approaches.

Table 1. Detection performance for six publicly available static tools.

	KANAL plugin for PEiD	Findcrypt Plugin for IDA Pro	SnD Crypto Scanner	x3chun Crypto Searcher	Hash & Crypto Detector	DRACA
Gladman AES	+	-	+	-	+	-
Cryptopp AES	+	-	+	+	+	-
OpenSSL AES	+	+	+	+	-	-
Cryptopp DES	+	+	+	+	-	+
OpenSSL DES	+	-	+	+	-	-
Cryptopp RC4	-	-	+	-	-	-
OpenSSL RC4	-	-	-	-	-	-
Cryptopp MD5	+	+	+	+	+	+
OpenSSL MD5	+	+	+	+	+	+
OpenSSL RSA	-	-	-	-	-	-
Cryptopp RSA	-	-	-	-	-	-

2.2 Dynamic Approaches

One of the first papers which addresses the problem of revealing the cryptographic algorithms in a program during runtime was presented by Wang et al. [20]. The authors utilize data lifetime analysis, including data tainting, and dynamic binary instrumentation to determine the turning point between ciphertext and plaintext, i.e., message decryption and message processing phase. Then, they are able to pinpoint the memory locations that contain the decrypted message. Wang et al. evaluate their work with an evaluation of their implementation against four standard protocols (HTTPs, IRC, MIME, and an unknown one used by *Agobot*). In their tests, they are able to decipher all encrypted messages using their implementation. The main drawback of this approach is that only a single turning point between decryption and message processing can be handled: if a program decrypts a block from a message, processes it, and continues with the next block, this behavior will not be identified correctly.

As a followup paper, Caballero et al. [2] refined the methods of Wang et al. [20]. For the automated protocol reverse engineering [4, 9, 22] of the MegaD malware, the authors rely on the intuition that the encryption routines use a high percentage of bitwise arithmetic instructions. For each instance of an executed function, they compute the ratio of bitwise arithmetic instructions. If the functions is executed for at least 20 times and the ratio is higher than 55%, the function is flagged as an encryption/decryption function. In an evaluation, this method reveals all relevant cryptographic routines. To identify the parameters of the routine (e.g., the unencrypted data before it gets encrypted) the authors evaluate the read set of the flagged function. To distinguish the plaintext from the key and other data used by the encryption function, they compare the read set to the read sets of other instances of the same function. As only the plaintext varies, the authors are able to identify the plaintext data.

Caballero et al. also cite Lutz [12] on the intuition that cryptographic routines use a high ratio of bitwise arithmetic instructions. Lutz's approach is based on the following three observations: first, loops are a core component of cryptographic algorithms. Second, cryptographic algorithms heavily use integer arithmetic, and third, the decryption process decreases information entropy of tainted memory. A core method of the tool is to use taint analysis [14] and estimate if a buffer has been decrypted by measuring its entropy. The main problem of relying on entropy is the possibility of false positives depending on the mode of operation. If we consider for example cipher-block chaining (CBC) mode, we observe that the input to the encryption algorithm is the latest ciphertext XORed with the current plaintext. Thus, the input to the algorithm will have a similar entropy as its output, because the XOR operation composing the input will incorporate pseudo-random data from the latest output of the cipher.

In this paper, we refine the heuristics introduced by others and show that we can improve the detection accuracy. In comparison to previous work in this area, we also study the identification of a larger set of algorithms (hash algorithms and asymmetric cryptography) and the identification and verification of input, output, and key material. In a recent and concurrent work, Caballero et al. introduced a technique called *stitched dynamic symbolic execution* that can be used to locate cryptographic functions in a binary program [3]. We could combine this technique with the methods introduced in this paper to precisely identify cryptographic code in a given binary.

3 Finding Cryptographic Primitives

In this section, we present in detail our heuristics and the intuition behind them. Furthermore, we provide an overview of the system we have implemented to automatically pinpoint cryptographic primitives. We start with an overview of the system and then introduce the trace and analysis implementations separately.

3.1 System Overview

The system implementation is divided in two stages, which are performed for each analysis of a binary sample. In the first stage, we perform fine-grained binary instrumentation, and the second stage implements several heuristics to identify cryptographic code from the data gathered by the first stage.

During the controlled execution of the target binary program (first stage of Figure 1), we use the technique of *dynamic binary instrumentation* (DBI) to gain insight on the program flow. We perform DBI to collect an execution trace, which also includes the memory areas accessed and modified by the program. We use the DBI framework *Pin* [11], which supports fine-grained instruction-level tracing of a single process. Our implementation creates a run trace of a software sample to gather the relevant data for the second stage.

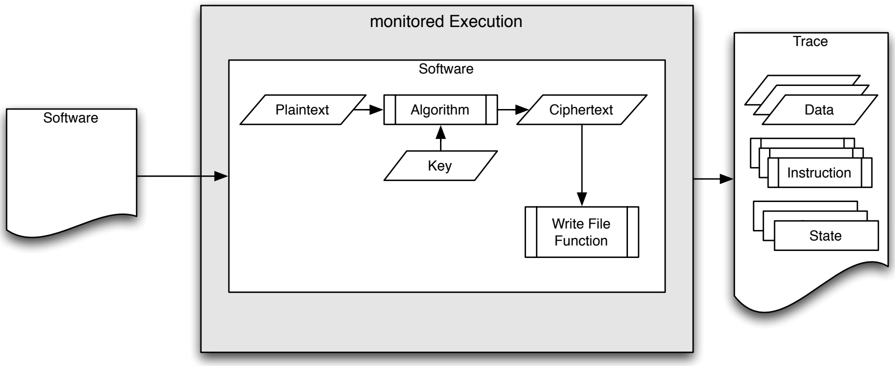

Fig. 1. Schematic overview of implementation for stage 1.

In the second stage, the instruction and data trace is used to detect cryptographic algorithms, e.g., RC4, MD5, or AES, and their parameters, e.g., keys or plaintext. An overview of the second stage is shown in Figure 2. To detect the algorithms and their parameters, we first elevate the trace to high-level structured representations, i.e., loops, basic blocks, and graphs. Then, we employ different identification methods and utilize the high-level representation of the trace to inspect the execution for cryptographic primitives. Based on the findings, the tool generates a report that displays the results, especially the identified algorithms and their parameters.

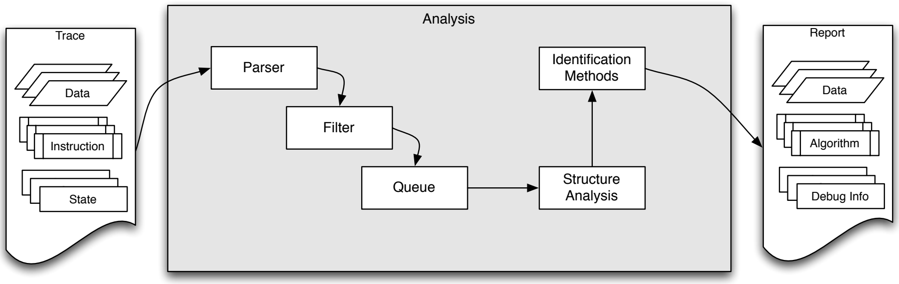

Fig. 2. Schematic overview of implementation for stage 2.

3.2 Fine-Grained Dynamic Binary Instrumentation

Execution tracing, or simply *tracing*, is the process of analyzing a binary executable during runtime to generate a protocol that describes the instructions executed and the data accessed by the executable. *Dynamic instrumentation* is a technique that performs code transformations to insert analysis code into an arbitrary program.

As mentioned before, the software sample is traced using the Pin dynamic binary instrumentation framework [11]. Since Pin uses dynamic instrumentation, there is no need to recompile or relink the program. Pin discovers code at runtime, including dynamically-generated code, and instruments it as specified by the user-supplied *Pintool.* Using the Pin API, a Pintool has access to context information such as register contents or debug symbols. The Pin framework deals with the dynamic code injection, code integrity, and restoring of registers which were modified by the Pintool. Pin differentiates between two modifications to program code: *instrumentation routines* and *analysis routines.* Instrumentation routines detect where instrumentation should be inserted, e.g., before each instruction, and then modify the code accordingly. Thus, the instrumentation routines are only executed the first time a new instruction is executed. On the other hand, analysis routines define what actions are performed when the instrumentation is activated, e.g., writing to the trace file. They occur every time an instrumented instruction is executed.

Data Reduction. In order to minimize the size of the trace file, we utilize two filter methods. On the one hand, we exclude instructions from libraries of which we have a priori knowledge that they do not contain cryptographic code. Using a DLL whitelist, we are able to circumvent large code portions. This is especially useful to reduce the trace time and file size. On the other hand, we can filter by thread ID and are also able to start the trace after a certain number of instructions already occurred, for example to skip an unpacker (if we have a priori knowledge that a specific packer is used by the given sample).

Collected Data. For the analysis we need to record the following information on an instruction-level granularity:

- Current thread ID
- Current instruction pointer together with involved registers and their data
- Instruction disassembly
- Accessed memory values, before and after the instruction, including mode (read or write), size, and address
- (Optional:) Debug information of the current instruction location, e.g., DLL module, function symbol, offset to function symbol

Using this information, we are able to conduct the next step: the analysis of the trace. The analysis, which is performed after or in parallel to the trace, is divided into two kinds of procedures. First, high-level information, e.g., the control flow graph, is generated from the trace. Next, the cryptographic code identification methods are executed upon the high-level representation.

Basic Block Detection. A basic block is defined as a sequence of instructions which are always executed in the given order. Each basic block has a single entry and single exit point. Since the basic blocks are generated dynamically from a trace, the result of the basic block detection algorithm may differ from a static detection algorithm [19]. The basic blocks are generated from the dynamic trace, thus non-executed code will not be considered by the detection algorithm, because it is not incorporated in the trace. Nevertheless, an advantage of dynamic tracing is the ability to monitor indirect branches and thus we are able to incorporate their result into the basic block detection algorithm. If a basic block is changed by self-modifying code, the change is noticed when the new code is first executed. A modified basic block is therefore registered as a new basic block, because the new block's instructions are different from the old ones.

Loop Detection and Control Flow Graph Generation. Loops are defined as the repeated execution of the same instructions, commonly with different data. To perform the detection of loops, we follow the approach from Tubella and González [18]. We could use the dominator relationship in the flow graphs (e.g., via the Lengauer and Tarjan algorithm [8]), but this would not recover the same amount of information: these algorithms operate on a control flow graph, and therefore do not convey in which order control edges are taken during execution. However, using the Tubella and González algorithm, we are able to determine the hierarchy of loops and the exact amount of executions and iterations of each loop body. The algorithm detects a loop by multiple executions of the same code addresses. A loop execution is completed if there is no jump back to the beginning of the loop body, a jump outside of the loop body, or a return instruction executed inside the loop body. Loop detection, with the fine granularity presented here, is a clear advantage of dynamic analysis. For example, with static analysis, the number of iterations or executions of a particular loop cannot always be easily determined.

A common optimization technique for cryptographic code is the *unrolling* of loops to save the instructions needed for the loop control, e.g., counters, compare, and jump instructions, and to mitigate the risk of clearing the instruction pipeline by a falsely-predicted jump. While many implementations discussed here partially unroll loops, no implementation unrolls every loop. Therefore, we find a lot of looped cryptographic code and can still rely on this heuristic.

We also build a control flow graph based upon the basic block detection algorithm. Given the list of executed basic blocks, we detect the control flow changes, i.e., which basic block jumps to which block, and use this information to reconstruct the control flow graph.

Memory Reconstruction. To further analyze the data incorporated in a trace, we need to reconstruct the memory contents, i.e., generate memory dumps from the trace at different points in time. This is especially important because cryptographic keys are larger (e.g., 128 or 256 bit) than the word size of the architecture (e.g., 32 bit). Thus, a cryptographic primitive can extend over several words in memory and has to be accessed by multiple operations. To reconstruct such a primitive, we need to consider and combine multiple operations. As we do not conduct fine-grained taint analysis [1, 6, 14], we need to reassemble the memory based on its addresses, which can serve as a rough approximation.

If an instruction involves a memory access, we record the following information in the trace:

- Memory address and size of access (8, 16, or 32 bit)
- Actual data read or written
- Mode of operation (read or write)

From this information we reconstruct the memory content. Since data at an address can change during the trace, we may have several values for the same address. Thus, instead of dumping the memory for a particular point in time, we instead reconstruct blocks of memory that have a semantic relationship. For example, a read of 128 bit cryptographic key material may occur in four consecutive 32 bit reads. Then, later a 8 bit write operations to the same memory region may destroy the key in a memory reconstruction. Therefore, we try to separate the 8 bit writes from the read 128 bit key block.

For this method, we rely on a few characteristics of the memory block, i.e., the interconnected memory composed of several words. First, we distinguish between read or write blocks and thus separate the traced memory accesses based on the access mode. Second, we assume that a block is accessed in an ascending or descending sequential order. Thus, we save the last n memory accesses, which occurred before the current memory access. In our experiments $n = 6$ turned out to be a reliable threshold. As a third characteristic, we use the size of the access to distinguish between multiple accesses at the same address.

3.3 Heuristics for Detecting Cryptographic Primitives

In this section, we discuss the different properties of cryptographic code and elaborate on the implemented methods to detect the cryptographic code and its primitives. First, we provide an overview of the identification methodology and then, based on code observations we make, we explain the developed identification methods. In order to successfully identify the cryptographic primitives we have to algorithmically solve the following questions: *which* cryptographic primitives are used, *where* are they implemented in code, *what* are their parameters, and *when* are they used?

We distinguish between two classes of identification algorithms: *signature-based* and *generic*. The main differentiation is the knowledge needed for the identification algorithm. For signature-based identification, we need a priori knowledge about the specific cryptographic algorithm or implementation. On the other hand, for generic identification we use characteristics common to all cryptographic algorithms and therefore do not need any specific knowledge.

Observations. We now point out three important features of cryptographic code, which we found and confirmed during the course of this work.

1) Cryptographic code makes excessive use of bitwise arithmetic instructions. Due to the computations inherent in cryptographic algorithms many arithmetic instructions occur. Especially for substitutions and permutations, the compiled implementations make extensive use of bitwise arithmetic instructions. Also, many cryptographic algorithms are optimized for modern computing architectures: for example, contemporary algorithms like AES are speed-optimized for the Intel 32 bit architecture and use the available bitwise instructions.

2) Cryptographic code contains loops. While substitutions and permutations modify the internal data representation, they are applied multiple times commonly with modifications to the data, e.g., the round key. We can recognize, even in unrolled code, that the basic blocks of cryptographic code are executed multiple times.

Solely for an identification method the presence of loops is insufficient. The observation rather has to be combined with other methods to provide a sound identification, because loops are inherent in all modern software. Although the number of encryption rounds is unique to each algorithm and may be used for an identification, this is not the case for unrolled algorithms, where the original number of rounds cannot be found in the majority of unrolled testing applications which we investigated.

3) Input and output to cryptographic code have a predefined, verifiable relation. The cryptographic algorithms which we consider in this paper are deterministic. Therefore, for any input the corresponding output will be constant over multiple executions. Given a cryptographic primitive was executed during the trace, the input and output parameters contained in the trace will conform to the deterministic relation of the cryptographic algorithm. Thus, if we can extract possible input and output candidates for a cryptographic algorithm, we can verify whether a reference algorithm generates the same output for the given input.

Thereby, we cannot only verify which cryptographic algorithm has been traced, but we can also determine what cryptographic parameters have been used. Of course, this observation can only be utilized with a reference implementation: if the software program contains a proprietary algorithm, we cannot verify it.

Identification Methods. Based on our observations detailed before, we developed and implemented several identification methods.

Chains heuristic. The first heuristic is based on the sequence of instructions, i.e., the ordered concatenation of all mnemonics in a basic block. For identification, an unknown sample's sequence is created and compared to the set of existing, known sequences in the pattern database. If the sequence can be found, a cryptographic implementation has been detected. We prepared the pattern database with different open-source cryptographic implementations. To differentiate between sequences defining an algorithm and sequences defining an implementation, we generated multiple datasets for each algorithm and each implementation. Thereby, we can identify implementations and algorithms in different levels of granularity and compare the effectiveness of the different patterns. Then, we form different datasets using union, intersection, and subtraction as follows:

- For each implementation of an algorithm
- For each algorithm, based on the intersection of all implementations of the particular algorithm
- An unique dataset for each algorithm, based on the subtraction with other algorithms

Mnemonic-Const Heuristic. The second identification method extends the first one and is based on the combination of instructions and constants. The intuition is that each implementation of a cryptographic algorithm contains unique (mnemonic, constant)-tuples that are characteristic for this algorithm, e.g., every MD5 implementation we studied contains `ROL 0x7` and `ROL 0xC` instructions. We also studied whether constants alone are characteristic enough (e.g., `0xc66363a5` is the first value of the unrolled lookup table for AES implementations), but found that such an approach leads to many false positives in practice. However, a combination of instructions and constants leads to a more robust approach, and thus we developed an identification method which employs a dataset based on (mnemonic, constant)-tuples. For every implementation we generate a set of instructions and their corresponding constants, e.g., `ROL 0x7`. Then, we again form the different data sets as described in the first heuristic.

An example for the datasets is given in Figure 3. For a given set of (mnemonic, constant)-tuples from a trace, we can measure to which percentage the tuples from a signature dataset are included in the trace. We observed that the unique and intersection datasets have a stronger relation to the algorithm. Implementation datasets have a looser connection to the traced implementation and pose a higher risk of generating false-positives. The number of tuples per testing application varies between 40 and 454 and the mean value is 165 tuples.

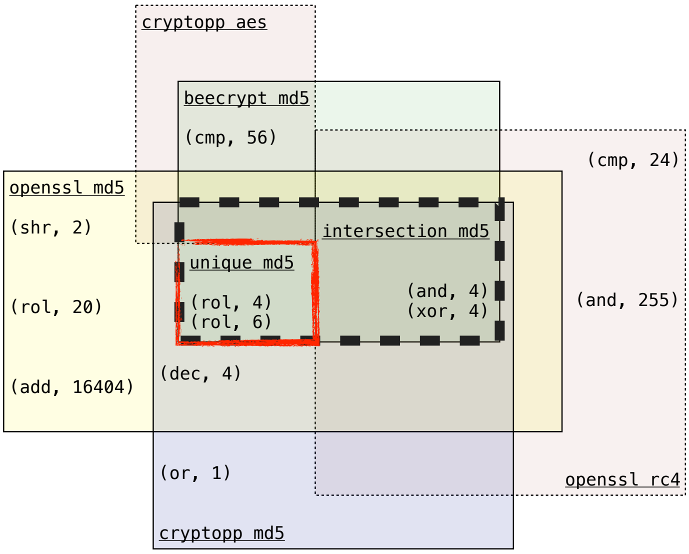

Fig. 3. Composition of (mnemonic, constant)-tuple datasets

The comparison is implemented as follows: First, we generate the set of (mnemonic, constant)-tuples found during the tracing. Using this trace-set, we check for each of the known pattern datasets to which percentage the trace-set intersects the signature-set. If the percentage is above the threshold of 70%, we report a positive identification. The threshold was empirically determined during the development process and in preliminary tests of our tool.

Verifier Heuristic. The third identification method is focused on memory data. We use *verifiers* to confirm a relationship between the input and output of a permutation box. Using the memory reconstruction method described in Section 3.2, we are able to verify complete instances of a cryptographic algorithm using plaintext, key, and ciphertext residing in memory.

As the memory reconstruction method reassembles cryptographic data of any length, we are able to reconstruct a set of possible key, plaintext, and ciphertext candidates. These candidates are then passed to a reference implementation of the particular algorithm, the *verifier*. If the output of the algorithm matches the output in memory, we have successfully identified an instance of the algorithm including its parameters. The main limitation of this approach is the premise that the algorithm is public and our system contains a reference implementation to verify the input-output relation.

We do not specifically have to consider and distinguish between encryption or decryption, because the encryption and decryption are commonly the same algorithms for stream and block ciphers. The efficiency of this approach is bound to the amount of candidates: if we can identify specific cryptographic code using other identification methods before, the efficiency is highly increased, since less candidates need to be checked. Optionally, we can reduce the set of candidates using previous identification results (e.g., if a signature has detected AES code, we can reduce the memory reconstruction to this code section, instead of the complete trace). Further, we only need to check for 128, 192, and 256 bit keys and 128 bit input/output blocks, based on the previous identification of AES.

Interestingly, this method isolates the cryptographic values from further modifications. Since we only verify and test using the reference implementation, further modifications, i.e., padding, encoding, or compression, can be separated and we detect the exact parameters to the cryptographic algorithm. Because of this soundness of the method, we already can note that we do not encounter false-positives using this method, as shown in the evaluation. In our proof-of-concept implementation, we only focussed on symmetric cryptographic algorithms.

Other Approaches. For comparison, we also implemented the approaches by Caballero et al. [2] and Wang et al. [20] to evaluate their method. A simple, yet effective, generic identification method is built upon the first observation: we evaluate basic blocks and determine whether the percentage of bitwise instructions is above a certain threshold. If the percentage is above the empirically-determined threshold of 55%, then we have identified cryptographic code. To eradicate false-positives, we use a minimum instructions per basic block threshold of 20: this threshold was determined by Caballero et al. and proved to be successful in our experiments, too.

Following the work from Wang et al. [20], we also implemented a cumulative measurement of the bitwise arithmetic instructions. Instead of measuring the bitwise percentage for basic blocks or function names, we update the percentage of bitwise instructions as we traverse the trace.

4 Experimental Evaluation

We have implemented the heuristics introduced in the previous section and now evaluate our approach and compare it to related work in this area. First, we provide an overview of the testing environment and then describe the system's performance for the testing applications, an off-the-shelf application, a packed testing application, and a real-world malware sample.

4.1 Evaluation Environment

The tracing is performed in a Sun VirtualBox 3.1.2 running Windows XP SP3 which is hosted on Mac OS X 10.6.2. The Pin version is 2.7-31933. The virtual machine is configured to have 1024 MB of RAM and operates with a single core of the host computer. The trace is written to the disk of the host computer through a VirtualBox shared folder. The host computer, on which the analysis VM is running, is equipped with a 2.4 GHz Intel Core 2 Duo with 4 GB of RAM. The FIFO queue size of the analysis is by default 500,000 instructions. With a fully loaded queue, the analysis process uses about 1.9 GB of RAM.

Table 2. Overview of testing applications

Implementation	Algorithm	Version	Compiler	Mode
Beecrypt	AES	4.1.2	VC dynamic	ECB encryption
Brian Gladman	AES	07-10-08	VC static	CBC encryption
Cryptopp	AES	5.6.0	VC static	CFB encryption
OpenSSL	AES	0.9.8g	MinGW static	CFB encryption
Cryptopp	DES	5.6.0	VC static	CFB encryption
OpenSSL	DES	0.9.8g	MinGW static	ECB encryption
Cryptopp	RC4	5.6.0	VC static	encryption
OpenSSL	RC4	0.9.8g	MinGW static	encryption
Beecrypt	MD5	4.1.2	VC dynamic	
Cryptopp	MD5	5.6.0	VC static	
OpenSSL	MD5	0.9.8g	MinGW static	
Cryptopp	RSA	5.6.0	VC static	OAEP SHA1
OpenSSL	RSA	1.0.0-beta3	VC dynamic	PKCS1.5

For the evaluation, we developed 13 testing applications and Table 2 provides an overview. The applications take input (e.g., two files holding plaintext and key), initialize the cryptographic library including the algorithm, perform the operation (i.e., encryption or decryption), and then output the result to a file. An overview of the cryptographic libraries' versions, used compilers and mode of operation is also given in Table 2. The compilers used were the Microsoft C/C++ Compiler version 15.00.21022.08 and the MinGW port of GCC version 3.4.2. We used different optimization levels when compiling the test applications to study the effect of compiler settings. Furthermore, some cryptographic libraries were linked statically, others dynamically, to test the Pintool's handling of dynamically loaded libraries.

4.2 Results

The performance of the analysis is rated by the successful identification of the cryptographic algorithm and parameters. Therefore, we analyze each trace of a testing application and review which identification method has identified the correct cryptographic algorithm.

Previous Approaches. First, we evaluate existing identification methods which attempt to identify the cryptographic algorithm only and not the parameters, and Table 3 shows the results. Note that we did not fully implement Lutz's identification method due to the lack of a taint-tracking functionality, thus the actual performance of this approach might be better in practice. False positives are abbreviated as FP and basic blocks as BBL. The results of the tools were compared with the source code and control flow graphs of the testing application in order to rate the performance of the methods.

The method of Caballero et al. [2] has a good success rate despite its simplicity. It always identifies the cryptographic basic blocks of the cipher and the hash

Table 3. Analysis results for heuristics published in previous work.

Implementation	Algorithm	Caballero et al.	Lutz	Wang et al.
Beecrypt	AES	success	found BBL	no result
Brian Gladman	AES	success	only FP	no result
Cryptopp	AES	partial	found BBL	error
OpenSSL	AES	success	found BBL	success `OPENSSL_cleanse`
Cryptopp	DES	success	found BBL	error
OpenSSL	DES	success	key schedule	success `DES_ecb_encrypt`
Cryptopp	RC4	partial	only FP	error
OpenSSL	RC4	success	no results	no result
Beecrypt	MD5	success	found BBL	success `md5Process`
Cryptopp	MD5	success	found BBL	error
OpenSSL	MD5	success	partial	success `MD5_Final`
Cryptopp	RSA	success & FP	only FP	error
OpenSSL	RSA	no success & FP	only FP	no result

implementations. It also identifies the key scheduling basic blocks and we rate this as a successful identification, because key scheduling is a core part of cryptographic algorithms. However, for two Cryptopp applications, the method only partially identifies the set of basic blocks: it misses parts of the key scheduling and the encryption phase. In case of the Cryptopp RSA testing application, the method successfully identifies the asymmetric encryption, but also lists several false-positive basic blocks. For the OpenSSL RSA implementations, the method only identifies false-positive basic blocks.

The method of Lutz [12] cannot be completely evaluated, because we did not implement the taint-tracking needed for it. However, we can note, that using data comparison without taint-tracking, the method is still able to identify cryptographic code. For the AES and DES testing applications, it identifies encryption basic blocks or key schedule blocks, due to entropy changes in the data. Also for the MD5 applications, it identifies the core MD5 functions. Although, with each successful identification, there is also a high rate of false-positives. For testing applications with few loops (e.g., OpenSSL RC4) the method shows no results, because the loop bodies or number of loop iterations are to small. The identification of plaintext or ciphertext is not successful in all tests.

The cumulative bitwise percentage method by Wang et al. [20] shows a good success rate for the testing applications with debug symbols. The method is based on the identification of functions by their debug symbols, therefore it yields false-positive or no results for the testing applications without debug symbols (the Cryptopp applications and Gladman's AES implementation do not contain debug symbols). Nevertheless, for the Beecrypt and OpenSSL applications the success rate is 57%.

Improved Heuristics. Next, we evaluate our methods and the results of the evaluation are shown in Table 4. The performance of the `chains` method is over-

	beecrypt	beecrypt	cryptopp	cryptopp	cryptopp	cryptopp	cryptopp	gladman	openssl	openssl	openssl	openssl	openssl
	aes	md5	aes	des	md5	rc4	rsa	aes	aes	des	md5	rc4	rsa
rc4 unique	0 %	0 %	100 %	100 %	100 %	100 %	100 %	0 %	50 %	0 %	0 %	100 %	0 %
des unique	0 %	0 %	44 %	100 %	44 %	44 %	44 %	22 %	33 %	100 %	11 %	0 %	0 %
rsa unique	22 %	8 %	58 %	61 %	50 %	46 %	89 %	34 %	18 %	1 %	7 %	1 %	89 %
md5 unique	0 %	100 %	6 %	29 %	100 %	6 %	12 %	0 %	0 %	0 %	100 %	0 %	0 %
rc4 intersect	68 %	68 %	100 %	100 %	100 %	100 %	95 %	64 %	77 %	77 %	68 %	100 %	68 %
aes intersect	100 %	82 %	100 %	100 %	82 %	82 %	94 %	100 %	100 %	88 %	88 %	71 %	88 %
des intersect	56 %	51 %	87 %	100 %	77 %	77 %	82 %	51 %	74 %	100 %	64 %	46 %	64 %
rsa intersect	34 %	28 %	71 %	71 %	63 %	57 %	93 %	41 %	35 %	24 %	29 %	16 %	92 %
md5 intersect	40 %	100 %	60 %	67 %	100 %	52 %	62 %	26 %	45 %	43 %	100 %	38 %	52 %
rc4 cryptopp	13 %	14 %	83 %	82 %	82 %	100 %	57 %	16 %	17 %	16 %	15 %	11 %	31 %
rc4 openssl	60 %	58 %	68 %	63 %	58 %	55 %	65 %	38 %	55 %	53 %	50 %	100 %	45 %
aes beecrypt	100 %	33 %	35 %	34 %	27 %	27 %	58 %	62 %	41 %	29 %	27 %	26 %	40 %
aes gladman	41 %	12 %	27 %	28 %	23 %	22 %	45 %	100 %	21 %	17 %	13 %	11 %	32 %
aes cryptopp	12 %	13 %	100 %	73 %	64 %	62 %	59 %	15 %	16 %	14 %	14 %	10 %	29 %
aes openssl	52 %	34 %	56 %	55 %	47 %	47 %	62 %	40 %	100 %	45 %	37 %	30 %	52 %
des cryptopp	12 %	14 %	74 %	100 %	65 %	62 %	53 %	15 %	15 %	15 %	15 %	10 %	29 %
des openssl	26 %	22 %	36 %	38 %	29 %	30 %	36 %	22 %	32 %	100 %	29 %	20 %	27 %
rsa cryptopp	12 %	9 %	48 %	43 %	39 %	36 %	72 %	14 %	11 %	9 %	9 %	6 %	23 %
rsa openssl	22 %	19 %	47 %	47 %	42 %	38 %	62 %	28 %	23 %	17 %	20 %	11 %	91 %
md5 beecrypt	45 %	100 %	50 %	56 %	74 %	41 %	58 %	26 %	38 %	35 %	73 %	35 %	47 %
md5 cryptopp	11 %	22 %	74 %	76 %	100 %	72 %	57 %	14 %	15 %	13 %	23 %	10 %	30 %
md5 openssl	34 %	66 %	49 %	53 %	71 %	41 %	55 %	25 %	37 %	41 %	100 %	27 %	45 %

Fig. 4. Results of the signature matching using (mnemonic, constant)-tuples

all good if we consider the unique-signatures. Since the signatures are partially generated from the testing applications, their matching performance seems successful in the evaluation. But if we evaluate against slightly different code, we expect that the detection rate might decrease. Therefore, a fuzzy matching algorithm for the mnemonic sequence comparison could mitigate the problem and we will investigate such a method as part of our future work.

The performance of the (mnemonic, constant)-tuple matching method is the most successful of the signature identification methods. The details of the results are presented in Figure 4: the signatures are displayed on the y-axis and the testing applications are shown on the x-axis. Each highlighted field links the testing application to the respective signature. If we apply the threshold of 70%, all implementations are correctly identified.

Third, the `verifier` heuristic, which also verifies the existence and the parameters of a symmetric encryption, is also capable of detection the cryptographic primitives within a given program. Table 4 shows that the method is able to detect nearly every instance of the symmetric encryption algorithms (RSA and MD5 are thus marked as `n/a`). The only undetected trace is Gladman's AES implementation. By design, the method does not yield false-positive results. The success of this method is closely bound to the memory reconstruction method described in Section 3.2. In case of the Gladman AES implementation, the memory reconstruction method is unable to reconstruct the cryptographic parameters. Thus, the method has no success. Although the memory reconstruction often leads up to 2000 candidates for encryption key, plaintext, and ciphertext each, the time for the candidate check is feasible. For AES, our non-optimized AES candidate check function is able to conduct 400,000 checks per second. If 2000 candidates for each parameter exist, the verification of all the candidates would only need $\frac{2000^2}{400000} = 10s$.

Table 4. Analysis results for our improved identification methods.

Implementation	Algorithm	chains	mnemonic-const	verifier
Beecrypt	AES	success	success	success
Brian Gladman	AES	success	success	no success
Cryptopp	AES	success	success	success
OpenSSL	AES	success	success	success
Cryptopp	DES	success	success	success
OpenSSL	DES	success	success	success
Cryptopp	RC4	success	success	success
OpenSSL	RC4	success	success	success
Beecrypt	MD5	success	success	n/a
Cryptopp	MD5	success	success	n/a
OpenSSL	MD5	success	success	n/a
Cryptopp	RSA	success	success	n/a
OpenSSL	RSA	success & FP	success	n/a

4.3 Off-the-Shelf Application

To show the generic usage of our approaches, we tested our system implementation against off-the-shelf software. We traced and analyzed a SSL session of the Curl HTTP client. Curl itself utilizes the OpenSSL library for establishing a SSL connection. In the testing environment, we used Curl version 7.19.7 with OpenSSL version 0.9.8l. We downloaded a HTML file from a webserver using HTTPs and traced the execution as explained in Section 3.2. We observed that the remote SSL server and the Curl client negotiated the following SLL cipher suite setting: `TLS_DHE_RSA_WITH_AES_256_CBC_SHA`. This means that the cipher suite specifies Diffie-Hellman Key Exchange, with RSA certificates, symmetric encrypted by AES in CBC mode with 256 bit keys, and integrity checked by SHA1. Thus, we knew that the analysis should at least detect the RSA and AES invocation. The selected cipher was used to encrypt three packets of SSL application data. Obviously, the first packet was the client HTTP request of 160 encrypted bytes, and then followed the server response with 272 bytes for the HTTP header and 5168 bytes of content.

The results are summarized in Table 5. The method by Caballero et al. successfully detected 19 basic blocks in the encryption and key scheduling functions `AES_set_decrypt_key`, `AES_set_encrypt_key`, `AES_decrypt`, and `AES_encrypt`. Lutz's method revealed 2,121 entropy changes in 26 loop bodies corresponding to 22 functions, for example in the AES encryption and decryption functions, but also in false-positive functions like `ASN1_OBJECT_it` or `OBJ_NAME_do_all_sorted`. The method by Wang et al. generated no results, probably due to the fact that the trace did not start at the beginning of the application.

The `chains` method, which compares mnemonic sequences, detected both AES and RSA without false-positives. An interesting result was revealed by the signature-based `mnemonic-const` identification method: since we were not able to generate an unique or intersecting set for the AES algorithm, we only had

Table 5. Analysis performance for the Curl trace

Method	Results
Caballero et al.	detected core AES basic blocks
Lutz	detected core AES loops, few FPs
Wang et al.	no results
`chains`	detected AES and RSA, including implementation
`mnemonic-const`	detected AES implementation, one false-positive
`verifier`	detected 94.6% of AES instances including parameters

the implementation signature for OpenSSL AES to match the trace. Among the implementation signatures, the OpenSSL AES signature had a relatively low match of 49%, compared to the results from the previous section. Nevertheless, other implementation signatures followed at about 20-30% and OpenSSL AES still stood out among them. The intersect and unique signatures (available only for DES, RSA, MD5) detected one high false-positive (intersecting DES with 56%) and some lower false-positives around 35%.

The `verifier` identification method outperformed all other methods. Of the 350 blocks of encrypted AES data, which we recorded using *tcpdump* for verification purposes, the identification method was able to find and verify the plaintext, key, and corresponding ciphertext of 331 blocks (success rate of 94.6%). Using the AES reference implementation, the method checked whether 3395 candidate keys and 4205 candidate plaintexts correspond to one of 8037 candidate ciphertexts. The missed 5.4% of AES primitives were caused by the memory reconstruction method, because the identification method only uses data from the reconstruction and verifies it using the reference implementation. Thus, the missing data has not been reconstructed and therefore could not be verified.

4.4 Distortion with Executable Packers

In order to test the identification performance against binary modification, e.g., binary packing and obfuscation, we packed a testing application and analyzed it using our system. The used packer was ASPack in version 2.12 and the testing application was a simple XOR application with an input/output of 4096 bytes. We chose ASPack since it is a common, widely used packer and the tool represent a large class of packing programs.

While the trace size increased by factor 17 and the analysis took longer, the analysis tool was still able to identify all blocks of XOR encrypted text. Interestingly, the packer introduced 24 new loops, but the loop analysis was still able to point out the original XOR encryption loop, which was also found in the original testing application. The packed loop still had 32 executions, with 128 iterations each, to encrypt a total of 4096 bytes. While this evaluation is only brief and we studied only a single packer, the result nevertheless indicates that the different heuristics are not perturbed by introducing executable packers and can thus also handle packed binaries.

4.5 Real-World Malware Sample: *GpCode*

We also tested the system against a real-world malware sample to demonstrate that we can indeed identify cryptographic primitives of a given binary sample in an automated way. The ransomware *GpCode* is a prime example for the application of cryptography in malware: after having infected a system, the malware's intend is not to hide its presence on the machine. Instead, *GpCode* encrypts the system's files with a key generated by the malware. Afterwards, the malware informs the victim of payment methods in order to obtain a decryption tool. The malware uses a custom executable packer and serves as another test case for distortion introduced via binary obfuscation.

In our tests, we found that only certain document formats are encrypted, e.g., `.doc`, `.pdf`, `.txt` files. For each file, the first three 16 byte blocks were encrypted and a marker (`0x03000000`) was appended to the file. Our tool determined that all encryption operations use the same 256-bit key to perform AES in ECB mode and the tool correctly extracted this key. Furthermore, the tool found that the (symmetric) AES key is encrypted using the malware author's RSA-1024 (asymmetric) public key in order to let the victim forward this information to the author. When executing the malware sample in our system, we were able to locate all instances of AES encryptions. Due to the malware's iteration over the complete filesystem, the tracing took 14 hours and the analysis phase 8 hours. Note that no manual intervention was necessary, the tool extracted the relevant information in an automated way. A victim could use our tool to discover the AES key and then decode all files accordingly.

5 Limitations

The heuristics presented in this paper also have several drawbacks and limitations which we discuss next. Obviously, dynamic analysis has the general constraint that if code is not executed, it cannot be analyzed. Thus, we rely on the fact that the binary executable unconditionally executes the cryptographic code that we want to analyze. Otherwise, the code would not be incorporated in the trace and thus cannot be used by the later identification methods. A drawback of our current implementation is the fact that the DBI framework Pin cannot handle all kinds of malicious software since the malware might detect the presence of the instrumentation code. However, we could implement the same heuristic based on other, more robust DBI or malware analysis frameworks.

The signature-based heuristics we introduced in this paper rely on the knowledge of the cryptographic algorithm such that we can generate the signatures. If the attacker implements his own cryptographic protocol, then these heuristics can not detect this fact. Several modifications to the internal functions of cryptographic algorithms can be performed, mostly to gain a space or time advantage. A very common form is a lookup table, which can be employed instead of bitwise addition and shifting. Another common programming technique is loop unrolling to avoid the flushing of the CPU's instruction pipeline and to save the loop's control instructions, e.g., JMP or INC. Since the correct and efficient implementation of cryptographic algorithms is a non-trivial task, many public code libraries exist to support application developers. Since the implementation is hard and even small changes can break the strength of the software, we expect that cryptographic code is often reused from cryptographic libraries such as OpenSSL or interfaces such as the Microsoft Cryptography API.

A compiler could generate code that has other characteristics not caught by our heuristics. To address this problem, our testing applications are created using two different compilers, because each compiler has a different approach towards optimizing the assembly code and thus produces different results. Furthermore, the results might depend on the compiler settings and optimizations used when creating the binary. Hence, we varied the compiler settings for the different evaluation programs. A related problem is interpreted code: during our analysis, we consider mainly C/C++ compiled code. However, an attacker could also use an interpreted language such as Python to implement his cryptographic routines which complicates analysis. Although an intermediate language can be well suited for heuristic identification, this is out of the scope of this work.

6 Conclusion

In this paper, we presented several methods to identify cryptographic code in binary programs. We pointed out the drawbacks of state-of-the art approaches in this area and evaluated available tools and techniques. Based on the insights and characteristics of cryptographic implementations, we developed three improved heuristics to enhance the detection accuracy. The implemented system was evaluated and we showed that our approach outperforms existing methods.

Availability. We believe that the interest in security analysis of cryptographic code will increase in the future. To foster research in this area, we publish our implementation of the different techniques and the data sets we used for the evaluation. All information is available at http://code.google.com/p/kerckhoffs.

Acknowledgements. This work has been supported by the the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009). We also thank the anonymous reviewers for their valuable insights and comments.

References

1. Beaucamps, P., Filiol, E.: On the Possibility of Practically Obfuscating Programs Towards a Unified Perspective of Code Protection. Journal in Computer Virology 3(1), 3–21 (2007)
2. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In: ACM Conference on Computer and Communications Security (CCS) (2009)

3. Caballero, J., Poosankam, P., McCamant, S., Babi ć, D., Song, D.: Input Generation via Decomposition and Re-stitching: Finding Bugs in Malware. In: ACM Conference on Computer and Communications Security (2010)
4. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In: ACM Conference on Computer and Communications Security (CCS) (2007)
5. Holz, T., Steiner, M., Dahl, F., Biersack, E., Freiling, F.: Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In: First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET) (2008)
6. Kruegel, C., Balzarotti, D., Robertson, W.K., Vigna, G.: Improving Signature Testing through Dynamic Data Flow Analysis. In: Annual Computer Security Applications Conference (ACSAC). pp. 53–63. IEEE Computer Society (2007)
7. Leder, F., Werner, T.: Know Your Enemy: Containing Conficker - To Tame A Malware. Know Your Enemy Series of the Honeynet Project (2009)
8. Lengauer, T., Tarjan, R.: A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Transactions on Programming Languages and Systems 1(1), 121–141 (1979)
9. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In: Network and Distributed System Security (NDSS). The Internet Society (2008)
10. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: ACM Conference on Computer and Communications Security (CCS) (2003)
11. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. ACM New York, NY, USA (2005)
12. Lutz, N.: Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's thesis, ETH Zürich (2008)
13. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: Annual Computer Security Applications Conference (ACSAC) (2007)
14. Newsome, J., Song, D.X.: Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In: Network and Distributed System Security (NDSS) (2005)
15. Popov, I.V., Debray, S.K., Andrews, G.R.: Binary Obfuscation Using Signals. In: USENIX Security Symposium (2007)
16. Porras, P., Saidi, H., Yegneswaran, V.: Conficker C P2P Protocol and Implementation. Tech. rep., SRI International (2009)
17. Stewart, J.: Inside the Storm: Protocols and Encryption of the Storm Botnet. Black Hat USA (2008)
18. Tubella, J., González, A.: Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In: 4th International Symposium on High-Performance Computer Architecture (1998)
19. Vigna, G.: Static Disassembly and Code Analysis. Malware Detection (2006)
20. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: ReFormat: Automatic Reverse Engineering of Encrypted Messages. In: European Symposium on Research in Computer Security (ESORICS) (2009)
21. Werner, T., Leder, F.: Waledac Isn't Good Either! InBot (2009)
22. Wondracek, G., Comparetti, P., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. In: Network and Distributed System Security (NDSS) (2008)
23. Young, A., Yung, M.: Cryptovirology: Extortion-Based Security Threats and Countermeasures. In: IEEE Symposium on Security and Privacy. pp. 129–141 (1996)