CIS: The Crypto Intelligence System for Automatic Detection and Localization of Cryptographic Functions in Current Malware

Felix Matenaar
RWTH Aachen, Germany
felix.matenaar@rwth-aachen.de

Andre Wichmann
Fraunhofer FKIE, Germany
andre.wichmann@fkie.fraunhofer.de

Felix Leder
Norman ASA, Norway
felix.leder@norman.com

Elmar Gerhards-Padilla
Fraunhofer FKIE, Germany
elmar.gerhards-padilla@fkie.fraunhofer.de

Abstract

Finding and extracting crypto algorithms in binary code is often a tedious reverse engineering task. A significant amount of manual work is required when unknown implementations are used. This is especially true for malware that contains variants of existing or even completely new algorithms. So far, no flexible and generic crypto detection framework exists that can support analysts in this task. The framework must be able to handle various heuristics that each are ideal to detect specific types of cryptographic algorithms. In addition, a suitable set of heuristics must be selected that can identify a wide range of crypto algorithms from various classes since the type of crypto implemented in a binary is not always known.

In this paper, we present the architecture of CIS, the Crypto Intelligence System, that fulfills the requirements for such a framework. Furthermore, we evaluate different heuristics for the real-world usage in the framework. The overall evaluation, using real programs, shows that CIS simplifies the job of an analysts significantly with a high detection and low false positive ratio.

1 Introduction

For the research of malware, and especially botnets, it is often essential to identify and reverse engineer the cryptographic algorithms that are used [3]. The continuous tracking of botnets, like the infamous banking trojans *ZeuS* [1] and *SpyEye* [2], is often impossible without extracting the communication crypto and keys. Update mechanisms and mitigation techniques can require the same steps [14].

Similar problems occur in the field of auditing the security implementations in commercial applications, especially if the source code is not available.

This is often a tedious reverse engineering task with a significant amount of manual effort. Analysts can spent several days or weeks just to find the relevant algorithms in programs that can easily exist of hundreds to thousands of functions with multiple thousands of instructions.

The existing attempts to automate parts of this process are rather immature. The search for known constants of crypto algorithms [6, 8] is most common but also limited to specific algorithms [7]. In addition, these approaches are not easily applicable to packed code or when the constants are calculated at run-time only. Other, more generic approaches are limited by the technology used [13], not extendable, too slow, or results are too wide to pinpoint the exact piece of code [4].

A framework for supporting analysts in pinpointing crypto algorithms needs to be generic with respect to types of cryptography, extendable to support multiple, complementing heuristics, has a low interference with the binary's execution, and needs to be as precise as possible. The latter requires a high detection rate with low false positives. In the context of malware it is essential that the framework also works with packed executables. To this end, the paper makes the following contributions:

- The architecture of CIS, the Crypto Intelligence System, an extensible crypto detection and localisation framework is developed
- The framework is complemented with the selection of suitable heuristics that can generically identify sets of cryptographic algorithms including variants.
- The most important goal is to support analysts by pinpointing the exact crypto algorithm. Therefore, the framework in combination with the selected heuristics is evaluated with respect to false positives and successful detections using known applications.

The rest of the paper is organized as follows. Section 2 discusses the problem of localising cryptographic routines in malware and its implications. Section 3 describes the architecture of the Crypto Intelligence System. In section 4, we evaluate different detection heuristics for their use in the framework. In section 5, we evaluate the benefit for analysts using real-world applications. Section 6 discusses related work, and section 7 concludes this paper.

2 Problem Statement

Our goal is to build a general, flexible framework for detecting cryptography in order to lower reverse engineering cost. We consider two challenges that need to be solved. First, malware analysis requires reliable and stealthy analysis techniques to cope with reverse engineering protection often found in malware. Second, the detection of cryptography is not a simple problem since it can be reduced to Rice's Theorem [23]. The right type of heuristics have to be used to lower the effort for analysts.

2.1 The Crypto Detection Problem

When we speak about detecting cryptographic functions in binary code, it is important to define cryptography first. A further question is if this problem can be divided into smaller parts for which there exist alternative ways for detection.

We distinguish between symmetric, asymmetric, and hash algorithms. Each class has their own set of properties that must be met in order to be secure. Due to the properties of each class, some heuristics work better or exclusively on one type. Other heuristics are more generic with the result of being less precise.

A common approach is to refine the detection problem to known crypto algorithms for each class. Previous work on this topic refers to algorithm-level detection by using pattern matching on well-known constants used in some cryptographic algorithms or by comparing function input and output with reference implementations [13]. On the one hand, this supports the analyst by also providing hints on the specific crypto. On the other hand, it limits the approach by not detecting new as well as custom tailored algorithms.

In summary, we classify approaches towards detection of cryptography using a two-fold approach:

- Methods for generically detecting arbitrary crypto algorithms

- Methods for detecting predefined sets of crypto algorithms.

2.2 Framework Requirements

The ability to be able to generically detect various types of crypto algorithms in combination with the specific usage scenario must be reflected in the architecture of CIS. The requirements for such a framework are being

- Able to handle packed and protected programs
- Least invasive to avoid interference with the original
- Provide a range of input for various detection approaches
- Extendable with respect to generic crypto detections

The majority of malware contains protective measures against reverse engineering. Most common in this context is the use of packers. Similar techniques can be part of licensing components in regular COTS software. They usually target either static analysis or specific dynamic analysis techniques. Therefore, it is usually recommended to use a hybrid analysis approach [15].

In the same context of evasion, it is important to be least invasive and thus, harder to detect. Especially certain types of dynamic analysis approaches are easy to detect, like debuggers and process modifications. A less invasive approach has to be used in in order to reduce the chances of being detected.

Invasiveness is also important in the context of run-time. Approaches that slow the analysis down by factors of hundreds to thousand can easily result in communication or watchdog timeouts and interfere with the binary in a way that it cannot be analyzed dynamically. An important aspect in this context is logging. The performance overhead by logging every details, like every executed instruction, quickly leads to this magnitude of overhead. In many cases, though, information can be processed faster on the fly without the this overhead. In other cases, the processing itself can pose such an overhead that it would have an impact on the analysis. A generic framework must be able to include real-time as well as logging and a-posteriori analysis.

Different heuristics require different pieces of information. The framework must be able to provide a range of useful intelligence. This needs to be available to multiple detection heuristics in order to be extendable. It needs to be avoided that extensions have to collect the same data twice. In the context of known crypto detection approaches six pieces of information are of relevance: instructions, basic blocks, functions, memory contents, crypto library hooks, system call tracing.

3 Design

In the following, the architecture of the Crypto Intelligence System (CIS) is described. Its design is directly derived from the requirements for crypto detection framework as discussed in the previous section. We chose QEMU [18] as the basis for our framework, as its dynamic binary translation architecture provides instruction-level analysis granularity while at the same time retaining a significantly

higher runtime performance compared to instruction emulation used by, for example, Bochs [29]. Another reason for choosing QEMU is that it is detected less often by malware than other virtualization environments [26].

The CIS consists of several independent parts. Introspection enables the system to control the instrumentation process with key events like thread creation, runtime loading of shared libraries or program termination. To gain further data about the internal behavior of the analysis target which is fed into the heuristics, the QEMU Tiny Code Generator (TCG) is used to hook into the binary translated code. Event treatment is a challenge in dynamic program analysis because most of the runtime overhead is produced during the data measurement.

3.1 Instrumentation Features

To fullfill the data requirements of our cryptography detection heuristics, it is necessary to implement more fine-granular instrumentation features than breakpoints, API- and Syscall tracing. For being able to identify locations of certain program behavior, the analysis run must be divided into code units. We chose both the execution of functions and single basic blocks as boundaries for our code units since there are heuristics for both cases. Checking the begin or end of basic blocks can be done reliably. In contrast, a reliable differentiation between functions assumes certain calling conventions which do not need to be satisfied by the analysed program. A combination of *call*/*ret* instruction instrumentation and stack-frame tracking was implemented to make this mechanism as reliable as possible.

Our core instrumentation features contain instruction and memory access tracing. The instruction tracing for example enables the measurement of percentage use of different instruction categories inside a basic block. This can be precalculated at translation time and stored together with the basic block start address as the access key. This technique makes measurements of instruction spreading very efficient. The implementation of the memory access tracing covers all instructions reading and/or writing to memory. The CIS is able to limit the instrumentation to certain memory pages only. This eliminates the instrumentation overhead of code whose behavior is already known as this is the case for system wide shared libraries for example.

3.2 Event Processing

Event processing is a crucial design part in our framework. The CIS includes an extensible event propagation architecture to cope with data processing as it is measured. A modular publish-subscribe like callback mechanism is used for analysis modules to register for certain program data. Events like memory accesses and basic block execution is measured by the low-level instrumentation modules directly hooking into TCG. Using the callback based API, further modules can subscribe to such events and aggregate them

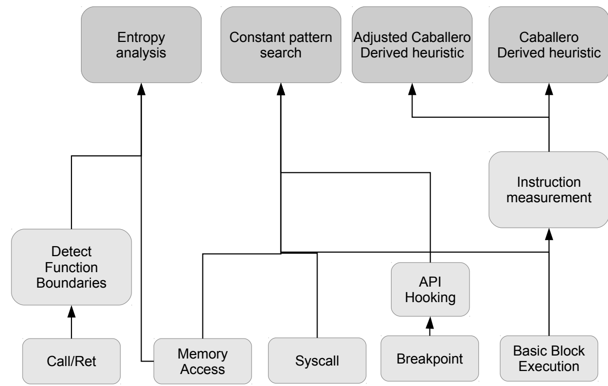

Figure 1. Data Processing from Bottom to Top

all the way up to the result logging for each heuristic. Figure 1 illustrates the event data flow in the CIS for some sample heuristics.

Many dynamic program analysis frameworks create trace data at runtime which is then stored persistently. The actual analysis is then executed in a post-processing step. On the one hand, this shifts analysis overhead from the critical runtime into an a-posteriori phase where calculation time is not a concern. In addition, this enables the application of many different analysis algorithms during one single program run. On the other hand, storing all required data during runtime can slow down the analysis due to the write speed of the storage medium, and might even be completely infeasible as the sheer amount of data and meta-data generated by instruction tracing for example can be too large to handle. The great advantage of an analysis during runtime is the possibility of event aggregation and filtering. Already analysed parts of the program could for example be conditionally excluded from instrumentation, which as a consequence reduces runtime overhead. Therefore, we use a two-stage processing approach. In the first stage, which is conducted during the analysis, fast heuristics can be applied. These need to be able to do real-time processing. The second stage, which is conducted a-posteriori after the analysis based on collected data, enables the usage of algorithms with higher complexity. Our a-posteriori analysis reads a trace file for each program thread and reconstructs the sequence of relevant runtime events. These are then used as input for the post processing heuristics.

4 Heuristics

The CIS framework provides the foundation for data collection from binary code. In order to be usable in a real-world scenario, it needs to be complemented by crypto de-

tection. For this purpose, 8 different detection heuristics are evaluated. In order to support analysts in real-world scenarios, not only a suitable frame-work has to be available, but also a suitable set of detection heuristics needs to be included in an overall solution.

For this purpose we have selected four existing heuristics. In addition, we developed four new heuristics. Since the goal of CIS is to generically detect a wide range of cryptographic algorithms, the quality of all heuristics is evaluated in order to select the best set. Our major criterion is that crypto algorithms can be detected generically. Thus, we evaluated the quality of the results for three different classes of algorithms: symmetric, asymmetric, and cryptographic hash algorithms.

In the following, we will first introduce the crypto heuristics and then present the result of our selection process.

4.1 Crypto Detection Heuristics

We have chosen eight heuristics for our evaluation. Caballero, constant search, absolute entropy, and crypto-API are well established approaches. Adjusted Caballero, entropy difference, asymmetric, and taint-graph are modifications of existing approaches developed towards our needs.

Caballero: The Caballero heuristic assumes that cryptographic functions rely on arithmetic and bitwise operations [24]. It considers the ratio of these operations in a block of code. If the ratio exceeds a certain threshold the algorithm or code block is assumed to be related to crypto. In a pre-study, we evaluated different parameters and found that the approach works best for code blocks of 20 or more instructions and a threshold of 70% for arithmetic and bitwise instructions when not taking ”mov” into account. This is in contrast to the original approach. Previous work [25] uses the original heuristic with a threshold of 55%.

Adjusted Caballero: The caballero heuristic includes a large set of instructions considered to be used in cryptography, like floating point operations. However manual inspection using reference implementations suggested that most of these instructions are shift operations, bitwise AND and OR and XOR. Therefore, we derived a second heuristic that only considers these instructions. This decreases the likelihood for false positives for graphics libraries and the like. Since we are reducing the amount of positive instructions, the threshold is reduced to 40%. This threshold was selected during another pre-study.

Asymmetric Caballero: The instructions named in the two Caballero based approaches are usually found in symmetric crypto. Asymmetric cryptography relies on different instructions. Therefore, a special variant was tuned for asymmetric crypto. In a pre-study, we found a good detection if at least 50% of the instructions are of type mul, div, or add, which is usually related to big number operations. The minimum block size is 10.

Abolute Entropy: Encrypted data is considered to have a high information entropy because it can ideally not be distinguished from random noise. Therefore, the idea to detect encrypted data using entropy measurements. We designed a heuristic that calculates the entropy of all memory regions that were read or written during the execution of a function. Note that this is slightly different from previous heuristics that do entropy analysis on files; we compute entropy on memory regions during program execution. When tracing the program counter it is possible to exactly pin-point the code that accesses high-entropy memory regions. We used the scaled entropy formula from [10] and set the threshold to 0.7.

Entropy Difference: Encrypted data that enters a program is not only accessed by crypto functions, but also during copy and other operations. The uniqueness of crypto functions is that they transform encrypted, high entropy input to lower entropy plain texts or vice versa if the plain text is of lower entropy. This can be captured when considering the entropy of memory regions that are read and compare it to that of memory regions that are written to in the same code block. This helps to differentiate crypto from regular copy operations. As for the absolute entropy we used the scaled entropy formula from [10].

Constant search: A range of crypto algorithms require specific constants for initialization, especially to hash algorithms. By monitoring for these it is possible to pin-point and name the algorithms. When linked against full crypto libraries, programs can contain several of these algorithms even if they are not used. This can lead to false positives. In order to avoid this situation, we only consider constants that used during the execution.

Taint-graph: Another heuristic exploits the fact that good symmetric algorithms have to fulfill the properties of confusion and diffusion. This essentially means that a small portion of the input has a significant influence on a large portion of the output. By tracking the data-flow in code blocks and determining the data input and output dependencies, we can capture this relationship. A detailed description would exceed the scope of this paper. We used an minimum input size of 4 bytes and assume confusion if these have an impact on at least 8 memory writes in the same block.

Crypto API: Cryptographic functions don't necessarily have to be implemented inside a program. A broad range of functions is available as part of today's operating systems. Therefore, we also monitor the use of such libraries.

4.2 Evaluation

In order to select the most generic heuristics, we evaluated the presented ones towards detection quality as well as false positives. In the end, four heuristics were found to be suitable for CIS.

Detection Quality The detection quality was studied using three different classes of crypto algorithms: symmetic, asymmetric, and hashes. 16 symmetric crypto algorithms were the ones implemented in the stegano-suite OpenPuff [16]. The 3 asymmetric algorithms and 9 hash algorithms that come as source code in Schneier's book [17] are used in addition.

All heuristics are evaluated towards true positives in the general case. Some of the heuristics have immanent weaknesses that makes it impossible for them to detect all algorithms. The constant search does not work on crypto that doesn't use any constants. The same applies to programs that don't make use of crypto APIs. Actually, none of the implementations used for the evaluation makes use of APIs. Thus, it is only included for completeness but didn't detect anything. This may be completely different for other applications.

Figure 2 summarizes the detection rates using reference implementations. Figure 3 is more detailed regarding the investigated algorithm classes. The Caballero variants have very good detection on symmetric and hash algorithms while failing mostly for the asymmetric class. The asymmetric variant of Caballero improves this a little bit. Constants are usally used in symmetric and hash algorithms and thus, perform ok on these classes while failing on the asymmetric class. The same applies for our taint-graph heuristic. It exploits the confusion and diffusion properties that are part of the symmetric class.

The absolute entropy heuristic has perfect detection. The entropy different achieves the same result for the three asymmetric algorithms but is less successful for the other classes.

All in all, the entropy approach outperforms the detection of all others. The two main Caballero variants have a good detection of more than 80% each. The constant search also proves to be powerful since a range of algorithms rely on initialization constants.

False Positives A good detection quality alone is not enough. A framework that should support an analyst also requires to limit the amount of false alerts. Thus, we evaluated the eight heuristics against seven real-world applications, displayed in table 4. Limiting the number of applications enabled us to verify the false positives manually.

We measured false positives for each heuristic using a collection of known programs in figure 4. The test set includes network usage and compression. Each program is analyzed and functions that trigger one of the heuristics but are non-cryptographic are considered as false positive. The results show that the constant search heuristic has the least number of false positives. The adjusted Caballero heuristic produced only half as much false positives than the original Caballero heuristic. The reason for this is that the original considers much more instructions to be cryptographic than the adjusted one. This results in larger false positive rates in the uharc test case in which arithmetic instructions are used for compression. The entropy measurement produced the highest number of false positives. While this is a bad result compared to the other three heuristics one has to consider the fact that no false negatives were produced in previous evaluation phase. The entropy difference performs much better in this context. The taing-graph heuristic has the highest false positive rate, which shows that the semantics behind data dependencies are hard to grasp. The asymmetric variant of Caballero is rather average and shares the same difficulties with the uharc test case with the others. No program was using crypto libraries, so that the API heuristic cannot detect anything.

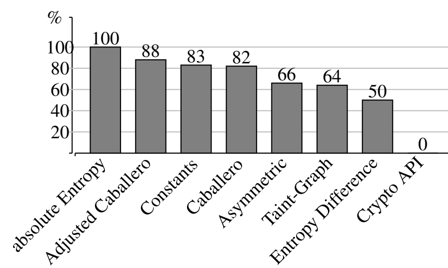

Figure 2. Detection of reference implementations

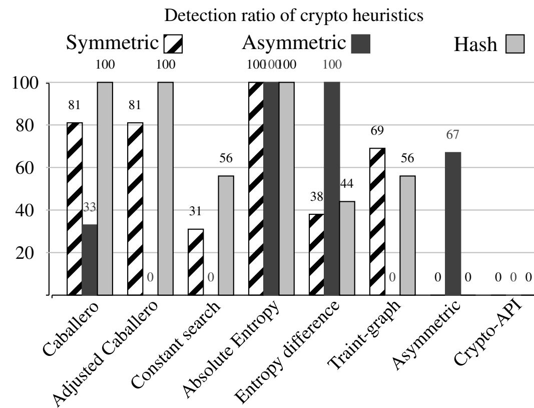

Figure 3. Class specific detection quality

4.3 Heuristic Selection

Based on the presented results, we select the original Caballero, the adjusted Caballero, the absolute, and the constant search for the CIS framework. The two Caballero based approaches are a good trade-off between detection

Program	Caballero	Adjusted Caballero	Constants	Absolute Entropy	Entropy Difference	Taint-Graph	Asymmetric	Crypto-API	Functions
curl	0	1	0	10	3	42	2	0	532
notepad	0	0	0	2	2	1	0	0	34
calc	0	1	0	5	2	6	0	0	91
wget	1	3	0	12	3	13	5	0	347
nslookup	1	1	0	5	3	0	0	0	49
uharc	27	10	3	17	9	12	17	0	301
telnet	0	1	0	9	3	4	0	0	64
Absolute	29	17	3	60	25	78	24	0	1418
Percent	2.0	1.2	0.21	4.2	1.76	5.5	1.69	0.0	100

Figure 4. False positives using regular programs

and false positives. The absolute entropy has the highest percentage of false positives but has also detected all crypto functions. The false positive ratio of only 4.2% is still acceptable since it still reduces the functions that an analyst has to investigate significantly. The constant search is the complement. It has a very low false positive rate and can pin-point the exact algorithm if known constants are used. The only limitation is that not every crypto algorithms uses constants.

5 Evaluation

The previous chapter presented how each heuristic performed during the true positive and false positive tests. For a final evaluation regarding the practical applicability of the CIS, the first four heuristics from figure 2 were chosen due to the previous results.

To perform the evaluation of the framework's practical applicability, we chose five off-the-shelf programs that are known to include cryptography: Curl HTTPS, Aescrypt, File Encrypter, Aphex Crypter, AES File Crypter. The goal was to figure out how well the CIS integrates into the overall reverse engineering of cryptography. Criteria for this test set included the program code size not to be overwhelming to limit the amount of verification time. In addition implementations of all three algorithm classes as mentioned in section 2 had to be used in the program set.

5.1 Practical Applicability

We first gathered all locations of cryptographic functions in the test set by hand. Then the CIS was used to carry out an automated analysis. Both results were manually compared. A comparison of the amount of true positives to the number of false positives for each heuristic is illustrated in figure 5.

As in the previous section, the absolute entropy measurement found every cryptographic function that was included. However, compared to other heuristics, a large amount of false positives were produced which considerably slowed down the manual verification. The Caballero heuristic and the adjusted one produced similar results as in the heuristic test phases, whereas the constant search resulted to be the most reliable. All three functions that were not detected did not include any constants to be found. However, one false positive was produced during execution. We found out that this was the only case where only one single constant pattern was found in a code region. Therefore adding a threshold for a minimum number of constant patterns to detect would have mitigated this problem regarding our evaluation.

5.2 Evaluation Conclusion

The evaluation shows that the CIS framework is a good tool for supporting analysts to finding cryptographic routines. In combination with the right heuristics it pin-points cryptographic algorithms with high precision while meeting all requirements set. A combination of heuristics can be used to fine-tune the trade-off between detection and false positives.

The evaluation confirmed our decision to create an adjusted Caballero heuristic. It created only half as much false positives while maintaining true positives. The constant search heuristic resulted to be reliable and applicable for algorithms for which constants can be found. We consider the applicability of the entropy heuristic to be subjective due to its extreme behavior regarding false and true positives. We suggest this heuristic to be used in cases where the verification cost of higher false positive rates is acceptable in exchange to a higher probability.

The evaluation shows that our framework is capable of meeting the required run-time performance overhead boundaries, which are not presented in detail in this paper.

The combination of runtime- and a-posteriori analysis enabled the application of all heuristics in a single run without the need to generate huge trace files which slow down execution or might be too large to store at all. Only the entropy measurement needed a-posteriori analysis. Using solely the other heuristics would eliminate the need to persistently store data besides the analysis results. Considering previous approaches which used up to several hours for the a-posteriori analysis of one single program run, this is a great performance achievement[25].

As a side note, we implemented an IDA Pro integration so that CIS can easily integrate into the existing reverse engineering workflow and to support the analyst as good as possible.

6 Related Work

A wide range exists in the area of binary code analysis in general, and malware in particular. Most of the work

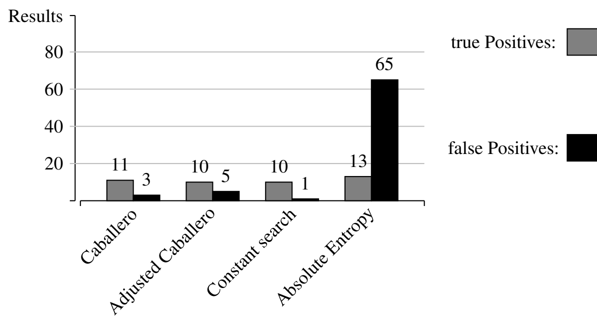

Figure 5. Evaluation using off-the-shelf programs

focuses on observing the semantic behavior of the malware related to API calls [22]. A focus on crypto exists in only few approaches.

PIN [19] and Valgrind [20] are user-space dynamic instrumentation framworks that change and monitor and the code during run-time. This impact reduces the analyzable sample and are easy to detect.

In order to avoid this, newer approaches employ virtualisation, which allows to control the operating system in addition to the program. The analysis framework that is closest to our own is *TEMU* of the *bitblaze* framework [5], which also extends QEMU. Further *VM introspection* approaches are based on hardware assisted virtualization[21].

The importance of detecting and finding cryptographic routines in malware is well recognized by the research community. Work in this area can roughly be categorized into static and dynamic analysis techniques.

One static analysis technique is to search for constants used in certain cryptographic algorithms. Several tools exist for this task [6, 7, 8]. Also work has been done to find the actual keys used for cryptography. This can be carried out e.g. for memory images [27].

In a work from Caballero et al., dynamic analysis is used to generate an instruction trace for offline analysis [9]. While the main focus is autoamtic reverse engineering of protocols, they also present a technique to identify what they call *encoding functions*, which also includes compression and obfuscation functions.

The automatic extraction of decrypted network traffic is studied in [10]. Different heuristics, based on function specific features, like loops and constants, is combined with memory taint tracking. This allows to identify candidates for crypto functions and extract the data that was touched during their execution. A similar idea is used in [11] to get the clear text from encrypted network data. Wang at al. make the assumption that the data will be decrypted shortly after it is received. This should be detectable by a high cumulative amount of arithmetic and bitwise instructions, which then should drop again when the decrypted data is processed afterwards. While their aim was not to find the decryption routine itself, the idea could be adapted to locate the decryption routine. This idea is supported by findings in [12].

In [13], several heuristics for detecting cryptographic routines are evaluated using a single analysis framework. While the topic is very similar to ours, they had a stronger focus on algorithm identification. [30] introduces further identification methods.

7 Conclusion

Finding and extracting cryptographic functions in binary code is often a hard reverse engineering task that requires a lot of manual effort. Still, it is an essentially important analysis step in the fight against malware. Likewise, it is relevant for the binary audit of regular applications.

While different detection heuristics have been proposed and evaluated in frameworks tailored specifically for the heuristics in question, there has not yet been made the effort to generally characterize the crypto detection problem and develop a framework capable of integrating arbitrary crypto detection heuristics. This paper tries to make a first step into this direction by deriving important requirements any crypto detection framework should fulfill, and presents the Crypto Intelligence System architecture. The CIS is extendable and can support a range of different heuristics in a single run. At the same time it retains a good run-time performance that minimizes the impact on the object under investigation.

The framework is complemented by a selection of crypto detection heuristics that are evaluated for their applicability to the general detection problem.

The evaluation using real-world programs shows that CIS is a good means to support the analysts with this tedious task by providing good precision in the results and by integrating well in the overall analysis process.

Building upon these foundations, the CIS will be used to evaluate and compare other crypto detection heuristics in future work. This includes fine-tuned versions of the existing heuristics. In addition to that, more FP tests and real-world use cases with malware help to strenghten the supportive character of CIS.

References

[1] Binsalleeh, H. and Ormerod, T. and Boukhtouta, A. and Sinha, P. and Youssef, A. and Debbabi, M. and Wang, L.: On the analysis of the Zeus botnet crimeware toolkit. In: IEEE Eighth Annual International Conference on Privacy Security and Trust (PST), pp. 31–38 (2010).

[2] Brand, M.: Forensic Recovery and Analysis of the Artefacts of Crimeware Toolkits. In: Proceedings of the 9th Australian Digital Forensics Conference (2011).

[3] Leder, F., Werner, T., Martini, P.: Proactive Botnet Countermeasures–An Offensive Approach. Cooperative Cyber Defence Centre of Excellence Tallinn, Estonia(2009).

[4] Leder, F., Martini, P., Wichmann, A.: Finding and Extracting Crypto Routines from Malware. In: Proceedings of the 2nd IEEE International Workshop on Information and Data Assurance (WIDA'09), Phoenix, Arizona, USA (2009).

[5] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Journal of Information Systems Security, pp. 1–25, Springer (2008).

[6] PEiD Krypto Analyzer (KANAL) Plugin. http://www.peid.info/plugins/, last visit: Apr. 2012

[7] Guilfanov, I.: FindCrypt. Hex Blog (2006). http://www.hexblog.com/?p=27, last visit: Apr. 2012

[8] Loki: SnD Crypto Scanner plugin for OllyDbg and Immunity Debugger.

[9] Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM conference on Computer and communications security, pp.621–634 (2009).

[10] Lutz N: Towards revealing attacker's intent by automatically decrypting network traffic. Master's thesis, ETH, Zurich, Switzerland, Jul. 2008.

[11] Wang Z., Jiang X., Cui W., Wang X.: ReFormat: Automatic reverse engineering of encrypted messages. In European Symposium on Research in Computer Security, Saint-Malo, France (2009).

[12] Leder F., Martini P. : Ngbpa next generation botnet protocol analysis. Emerging Challenges for Security, Privacy and Trust, 24th IFIP International Information Security Conference, Pafos, Cyprus, May 2009.

[13] Gröbert, F., Willems, C., Holz, T.: Automated Identification of Cryptographic Primitives in Binary Programs. In: Recent Advances in Intrusion Detection, pp. 41–60, Springer (2011).

[14] Leder, F., Werner, T.: Know Your Enemy: Containing Conficker. Honeynet Project KYE series, March 2009

[15] Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection, Proc. of the 23rd Annual Computer Security Applications Conference, 2007

[16] Oliboni, C.: OpenPuff - tool for steganography (2011), http://embeddedsw.net/doc/OpenPuff_Help_EN.pdf

[17] Schneier, B.: Applied cryptography (2nd ed.): protocols, algorithms, and source code in C, John Wiley & Sons, Inc., 1995

[18] Bellard F., *QEMU, a fast and portable dynamic translator*, Proc. of USENIX Annual Technical Conference, 2005

[19] 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, 2005

[20] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation, 2007

[21] Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions, 2008

[22] Pfoh, J., Schneider, C., Eckert, C.: Nitro: Hardware-based System Call Tracing for Virtual Machines, 2011

[23] Rice, H. G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Trans. Amer. Math. Soc. 74, 358-366, 1953.

[24] Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: Enabling Active Botnet Infltration using Automatic Protocol Reverse-Engineering, 2009

[25] Groebert, F., Willems C., Holz., T.: Automatic Identification of Cryptographic Primitives in Software, 2010

[26] Zhu, D. Y., and Chin, E. Detection of vm-aware malware, 2007

[27] Shamir, A., Van Someren, N.: Playing Hide and Seek With Stored Keys, Lecture Notes in Computer Science, 1998

[28] Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection

[29] Lawton, Kevin P., Bochs: A Portable PC Emulator for Unix/X, Sept. 1996

[30] Joan Calvet, Jos M. Fernandez and Jean-Yves Marion; Aligot: Cryptographic Function Identification in Obfuscated Binary Programs, 2012