JOURNAL OF CYBERSECURITY

Journal of Cybersecurity, 2024, tyae008
https://doi.org/10.1093/cybsec/tyae008

Research paper

CipherTrace: automatic detection of ciphers from execution traces to neutralize ransomware

Mostafa AbdelMoez Hassanin[*] and Ivan Martinovic

Department of Computer Science, University of Oxford, 7 Parks Rd, OX1 3QG, United Kingdom

[*]Corresponding author. Department of Computer Science, University of Oxford, 7 Parks Rd, OX1 3QG, United Kingdom. E-mail: mostafa@hassanin.ch

Received 5 November 2023; revised 25 March 2024; accepted 22 April 2024

Abstract

In 2021, the largest US pipeline system for refined oil products suffered a 6-day shutdown due to a ransomware attack [1]. In 2023, the sensitive systems of the US Marshals Service were attacked by a ransomware [2]. One of the most effective ways to fight ransomware is to extract the secret keys. The challenge of detecting and identifying cryptographic primitives has been around for over a decade. Many tools have been proposed, but the vast majority of them use templates or signatures, and their support for different operating systems and processor architectures is rather limited; neither have there been enough tools capable of extracting the secret keys. In this paper, we present CipherTrace, a generic and automated system to detect and identify the class of cipher algorithms in binary programs, and additionally, locate and extract the secret keys and cryptographic states accessed by the cipher. We focus on product ciphers, and evaluate CipherTrace using four standard cipher algorithms, four different hashing algorithms, and five of the most recent and popular ransomware specimens. Our results show that CipherTrace is capable of fully dissecting Fixed S-Box block ciphers (e.g. AES and Serpent) and can extract the secret keys and other cryptographic artefacts, regardless of the operating system, implementation, or input- or key-size, and without using signatures or templates. We show a significant improvement in performance and functionality compared to the closely related works. CipherTrace helps in fighting ransomware, and aids analysts in their malware analysis and reverse engineering efforts.

Keywords: binary analysis; dynamic analysis; reverse engineering; virtual machine introspection; cipher identification; cryptographic key identification and extraction

Introduction

New malware samples are discovered daily, and malware is rapidly evolving and becoming more sophisticated and evasive. Cryptographic algorithms are often proprietary in malware samples, and only executables are available. These executables are always obfuscated and updated regularly, therefore an analysis tool needs to account for potential obfuscation and/or morphism.

The potential of such a tool is very high, as 23% of malware incidents involve ransomware [3], and the average cost of a ransomware incident has been doubling yearly, reaching a staggering $1.54 million in 2023 [4, 5].

A recent survey on the detection techniques for ransomware has shown that, fully dissecting Fixed S-Box block ciphers (e.g. AES) takes care of at least 75% of ransomware specimens out in the wild [6]. That is due to the fact that a ransomware becomes obsolete if the secret keys can be extracted at runtime.

The objective of this paper is to assess the feasibility of automatically detecting generic cryptographic primitives and identifying ciphers to neutralize ransomware. It focuses on product ciphers, which are types of ciphers that employ a combination of substitution, permutation, and modular arithmetic operations [7]. There are two main use cases for our system:

- Analyze the activity of a binary, i.e. software.
- Record a live-system and analyze network or process activity in a replay snippet.

Since our system operates on a whole-system level, so an activity can be binary (e.g. extract ransomware secret keys), or network (e.g. extract SSL keys). Not to mention since we run the analysis on a replay, the analysis is repeatable in different configurations and settings. In a real world scenario, such a system could help in situations where recovering the key could unlock the systems affected by ransomware [8].

We review state-of-the-art prior works, and design and implement CipherTrace. We focus on product ciphers, and evaluate CipherTrace using four standard cipher algorithms known to be used in ransomware, with different implementations on different operating systems. Additionally, we use four well-known hashing algorithms, and five of the most recent and popular ransomware in the wild to test its effectiveness in real-world scenarios. The sample we tested are the obfuscated specimens used to infect targets since 2018.

General idea of CipherTrace

We employ a generic analysis technique to automatically detect and identify proprietary cipher algorithms, and extract their secret keys and cryptographic states (crypto states for short). A crypto state is essentially the cipher- or plaintext(s). In addition to fighting ransomware, our tool is meant to aid researchers in the automated reverse engineering of malicious binaries or system activities. In this case, extracting any cryptographic artefact (crypto artefact for short) will help dissect the cipher employed.

To address the limitations of the related work, we list our design goals as follows:

(1) Achieve isolation and have an overarching view on the binary's execution as well as its environment.
(2) Account for the runtime properties of the cipher, such as routines and memory management.
(3) Account for the cipher's operational components, such as internal state, substitution, or permutation.
(4) Enable automation to facilitate automated and scalable analysis and reverse engineering.
(5) Locate and extract the cryptographic artefacts, such as secret keys and internal cryptographic states.

Approach and design

Static analysis approaches have been challenged for decades by obfuscation, and the accuracy of the results is heavily challenged by potential morphism [9]. Dynamic analysis approaches lack higher-level semantics, which are present in source code, e.g. functions, memory buffers, and data types [10]. We follow a Virtual Machine Introspection (VMI) approach, which is a whole-system dynamic analysis approach applied by the underlying dynamic analysis platform (i.e. PANDA). A VMI approach achieves isolation [11], as the specimen is incapable of interacting with the analysis code. VMI enables employing higher-level semantics in the analysis (via our PANDA plugin), and since it runs on a whole-system level, it accounts for processes that may spawn off the specimen. CipherTrace has also proved to be resilient to a few classes of code obfuscation and many anti-debugging and anti-sandboxing checks and tricks (see the Evaluation section). This is due to applying our core concept of identifying cryptographic elements (crypto elements for short) in the Analysis engine.

To address the lack of higher-level semantics in dynamic analysis, we extract such semantics via our PANDA plugin and cross-check them on two Intermediate Representations, i.e. TCG (Emulator) and LLVM (Platform). After which we initiate our analysis by looking for the operational components of cryptography, henceforth, cryptographic elements, such as key-scheduling, round routines, and substitution or permutation, then we end up deriving a cipher class heuristically following a decision tree as per the previously mentioned definition of a product cipher. This novel technique allows us to be systematically closer to the features of the cipher as well as to its runtime properties, which makes it harder to evade and allows for generalization.

Another challenge of whole-system analysis is the increased rate of false positives compared to in-process analysis, since all cascading processes (and function calls) are also analyzed. To address that, we mainly focus on the intrinsic properties of the cipher algorithm (e.g. routines, number of executions, and entropy/randomness between inputs and outputs), and we aggregate results in the user-space per process and per function (the root basic block), and analyze multiple Control Flow Graphs (CFGs) in-parallel to increase the confidence in results.

Scope

As per our threat model, the adversary attempts to hinder the analysis by changing the implementation details, e.g. key-size, function names, and so on. Evaluating the security of the dynamic analysis platform and a few classes of code and data flow obfuscation (e.g. flattening the control flow, unrolled loops, and obfuscating input/output buffers) are out of scope. We focus on product ciphers, targeting mostly block ciphers (e.g. AES).

Obfuscation

We categorize obfuscation schemes into binary, code, and data obfuscation. Binary obfuscation aims to obfuscate the binary itself, mainly to evade static analysis techniques, e.g. packing [9]. Code obfuscation aims to obfuscate the control flow or code blocks in a program to decrease the readability, e.g. flattening the control flow, or obfuscating the loops [12]. Data obfuscation aims to obfuscate the data flow or memory management in a program [12]. Dynamic analysis approaches—including that of CipherTrace—are resilient to code packing by default, as only the executed code will be instrumented or introspected. And by tracing crypto elements, even if the code is obfuscated, the overall runtime features cannot be hidden. So, the number of executions of the core cryptographic routine(s) have to be reflected, arithmetic instructions have to be replaced by other equivalent ones, and high randomness/entropy cannot be removed.

CipherTrace is resilient to a few classes of control flow obfuscation. For example, a block cipher's substitution step (i.e. S-Box) is quite hard to hide while CipherTrace operates on a whole-system level and addresses the cipher's features and runtime properties simultaneously—analyzing multiple CFGs. As far as code obfuscation is concerned, due to employing introspection, CipherTrace analyzes the executed code regardless.

Contributions

As a result of our work, our contributions can be summarized as follows:

- We systematically explore detecting and identifying cryptographic primitives (crypto primitives for short) from introspection traces. This is to achieve isolation, increase anti-debugging resilience, and run the analysis on a whole-system level having an overarching view of an activity.
- We design generic crypto primitive detection based on the concept of crypto elements, to account for the design features of a given cipher.
- CipherTrace can also locate and extract secret keys, as well as crypto states. It is the only automated tool with such a level of granularity, cross-platform support and performance.

Table 1. Overview of prior works: CipherTrace is the only one that employs VMI to identify a cipher class, extract the secret keys, achieves isolation, and addresses obfuscation.

System	Platform	Technique	Tasks
Lutz (2008) [13]	Valgrind	DBI-Heuristics	T1
Re-Format (2009) [13]	AutoFormat [25]	DBI-Heuristics	T1
kerckhoffs (2010) [15]	Intel PIN	DBI-Heuristics	T2, T4
CipherXRay (2010) [16]	Valgrind	DBI-Heuristics	T2, T4, T5
Aligot (2012) [17]	Intel PIN	DBI-Heuristics	T2, T4
MovieStealer (2013) [24]	PANDA	VMI-Heuristics	T3
Hosfelt (2015) [18]	Intel PIN	DBI-ML	T2
CryptoHunt (2017) [19]	Intel PIN	DBI-Heuristics	T2
K-Hunt (2018) [23]	Intel PIN	DBI-Heuristics	T2, T4
CryptoKnight (2018) [20]	Intel PIN	DBI-ML	T2
bacs (2018) [21]	Intel PIN	DBI-Heuristics	T2
CipherTrace	PANDA	VMI-Heuristics	T3, T4, T5

T1: Analyzes network traffic. **T2**: Analyzes binary programs.
T3: Analyzes whole system activity. **T4**: Locates cryptographic keys.
T5: Identifies a cipher class.

- We implement and open-source CipherTrace, as well as the tested benchmark.
- Our PANDA plugin (func_stats) enhanced PANDA's dynamic analysis capabilities, assisting researchers in malware analysis, forensics, and reverse engineering.

Related work

In the last two decades, a lot of work has addressed the problem of detecting and identifying crypto primitives in network traffic and binary programs via dynamic analysis [13–21]. All these tools are only compatible with Intel processors because they rely on Intel PIN [22], except for a few which are not compatible with Windows [13, 16]. The aforementioned work applies heuristics, except for a few which applied Machine Learning (ML) [18, 20].

Design

The prior works discussed in this paper focus primarily on detecting implementations of certain crypto algorithms (e.g. AES and DES), rather than detecting a cipher (e.g. a block cipher), except for CipherXRay.

CipherXRay (2010) [16], kerckhoffs (2010) [15], Aligot (2012) [17], and K-Hunt (2018) [23] are able to locate the secret keys, but they lack isolation and dismiss higher-level semantics [11]. It is noteworthy that only MovieStealer follows a VMI approach, but it does not aim to identify crypto primitives or their secret keys [24]. Refer to Table 1 for an overview.

Functionality

Aligot and kerckhoffs were unable to identify a serpent256 block cipher when tested, even whilst having AES in their reference implementations, and K-Hunt addresses only insecure keys. CipherXRay needs to recover all input and output parameters of the cipher algorithm, and therefore it experiences various parameter reconstruction challenges [23], unlike CipherTrace, which adopts an offline analysis strategy running lightweight heuristics on the functions and the memory buffers they access. Also, CipherXRay's dynamic taint analysis may affect the execution [23], unlike CipherTrace's VMI approach in dynamic analysis. Another limitation for CipherXRay is that it does not check the intrinsic properties of the avalanche effect and may suffer from false positives. The avalanche effect is a desirable property in cryptography, wherein flipping a single bit in the key or the plaintext changes the ciphertext drastically. More information can be found in the Evaluation with closely related works section.

Cryptographic primitive detection

Crypto primitive detection relies mostly on basic block detection, loop detection, instructions profiling, and memory access patterns. Refer to Table 2 for an overview of the different stages in the prior works. We add CipherTrace for comparison.

The task of crypto primitive detection and identification involves going through some (or all) of the following stages: inspection, traits extraction, information measurement, instructions profiling, and identification. Inspection can be either instrumentation or introspection—the objective is to obtain an execution trace (refer to Table 1 for more information).

The purpose of traits extraction is to extract the most fundamental features in a binary program, i.e. basic blocks and loops. Information measurement determines whether the information is rich in content (entropy) and/or exhibits a high distribution (randomness). Finally, instructions profiling enquires as to what the instructions exhibit in terms of execution context–from 'operands' and 'opcodes'.

CipherXRay, Re-Format, and bacs are data flow oriented, as they do not depend on identifying basic blocks or loops. Re-Format employs data lifetime analysis with dynamic taint analysis to identify the runtime memory buffers in network traffic. CipherXRay employs data flow tracking to detect bit-level flips in input and output buffers, looking for the avalanche effect, while bacs is based on Data Flow Graph (DFG) look-ups, but without locating the secret keys. On one hand, most of the prior works mentioned employ data flow analysis (or input/output relations) in heuristics. Moreover, kerckhoffs and Aligot employ constant detection in their search, but not in a strong dependency such as in bacs. On the other hand, CipherXRay, MovieStealer, K-Hunt, and CipherTrace do not employ constant detection at all, making them more generic, but MovieStealer does not aim to identify crypto primitives or their secret keys. Noteworthy that, CipherXRay, Aligot, CryptoHunt, K-Hunt, and CipherTrace look for correlations between the number of executions (or execution length) and the input-size, making it easier to spot cryptographic rounds and

Table 2. Technical overview of prior works: CipherTrace employs a VMI dynamic analysis approach (PANDA) to achieve its task.

	Basic blocks	Loops	Information measurement	Instructions profiling
Lutz (2008) [13]	Conditional branching 'jump' instrs	Backward edges in CFG	Shannon entropy test	Ratio of arith and bit-wise instrs
Re-Format (2009) [14]	None	None	None	Ratio of arith and bit-wise instrs
kerckhoffs (2010) [15]	'jump' or 'return' instrs	Repeated execution of code addr	Shannon entropy test	Percentage of bit-wise arith instrs
CipherXRay (2010) [16]	None	None	Contribution rate [16]	None
Aligot (2012) [17]	Intel PIN's statistics	Repeated instruction sequence	Shannon entropy test	High ratio of bit-wise arith instrs
MovieStealer (2013) [24]	PANDA's callbacks	Repeated execution of code addr	Chi-square randomness test	High counts of arith instrs
Hosfelt (2015) [18]	Branching	Using instruction counter	N/A	High counts of arith instrs
CryptoHunt (2017) [19]	Branching and 'return' instrs	Bit-precise symbolic mapping	Shannon entropy test	Number of bit-wise instrs
K-Hunt (2018) [23]	Intel PIN's statistics	Repeated execution of code addr	Shannon entropy test Chi-square randomness test Monte Carlo simulation	Ratio of arith and bit-wise instrs
CryptoKnight (2018) [20]	Branching and 'return' instrs	Repeated execution of code addr	Shannon entropy test	Ratio of bit-wise instrs
bacs (2018) [21]	None	Signature-based lookup via DFG	None	DFG lookup
CipherTrace	PANDA's callbacks	Repeated execution of code addr	Shannon entropy test Chi-square randomness test	High counts of bit-wise and arith

block ciphers, but CryptoHunt does not aim to even locate the secret keys.

Obfuscation

Kerckhoffs, CipherXRay, Aligot, CryptoHunt, and K-Hunt depend on the semantics of cryptography in their task of detecting and identifying cryptographic operations [23].

K-Hunt claims that this fact is enough to make a tool obfuscation-resilient, but we argue that such semantics can still be obfuscated. Therefore, we adopt a generic VMI-based technique to extract such semantics on a whole-system level (using our PANDA plugin), and via multiple Intermediate Representations (TCG and LLVM). CipherTrace does also aggregate results, and analyzes multiple CFGs in-parallel to increase the confidence level. K-Hunt (2018) [23] and bacs (2018) [21] consider Virtual Machine (VM) obfuscation, antivirtualization, and various anti-debugging tricks. Apart from CipherXRay, Aligot (2012) [17], CryptoHunt (2017) [19], and K-Hunt, all the other tools discussed in this paper do not consider code or data obfuscation. Aligot authors claim that their tool is able to detect unrolled loops [17]. CryptoHunt is able to deal with data obfuscation by combining loop I/O relations with bit-precise symbolic execution [19]. K-Hunt is able to deal with nonstandard key buffers [23]. Finally, CipherXRay is able to deal with intrinsic memory buffers. In comparison, CipherTrace is able to deal with a few classes of code obfuscation, anti-debugging, and anti-VM checks and tricks. All the ransomware specimens we tested employ obfuscation (ranging from low to very high), and we caught the elements of product ciphers in all of the specimens. See Cryptovirological artefact analysis section for more information.

Cryptovirology

Kerckhoffs, CipherXRay, Aligot, CryptoHunt, and K-Hunt tested their tool with malicious binaries. Kerckhoffs tested only one binary, and K-Hunt tested two binaries. CipherXRay and Aligot were mostly focused on botnets, the former tested three binaries, and the latter tested four binaries. In comparison, CipherTrace tested five binaries of the most recent and popular ransomware families, so its results are more dependable, provided the context, and since the sample size (and variance) is higher.

Henceforth we define the closely related works as those tools that aim to identify crypto primitives, and at least locate secret keys, and they are: Kerckhoffs, CipherXRay, Aligot, and K-Hunt.

Design and implementation

Cryptographic instructions involve a high number of bit-wise and arithmetic instructions, and they correlate with memory access patterns, usually in ratio to the input-size. Moreover, routines translate into loops in application code, and loops are mostly exhibited as 'call' or 'jump' instructions in machine code. On an instruction level, a given set of instructions constitutes a basic block of code. A basic block of code is the set of instructions that has a single entry and a single exit. In CipherTrace, we associate the basic block(s) of code with a higher-level semantic (e.g. a function), whereas a function is defined via function headers in the executed binary.

In CipherTrace, the two main components are the Inspection Engine and the Analysis Engine. The Inspection Engine collects the synthetic information of function calls and their accessed memory buffers (via our PANDA plugin func_stats), where it outputs a 'func_stats' file. In the Analysis Engine, multiple stages of analysis take place, and they are; STACK ANALYSIS, FILTERING, and CRYPTO ELEMENTS IDENTIFICATION. CipherTrace outputs an execution report, graphs, and the interesting points in the execution (i.e. tap points), whereby we dump the memory buffers (i.e. secret keys, plaintexts, and ciphertexts). A tap point is essentially a machine's state at a certain program counter (i.e. instruction count). Refer to Fig. 1 for a system overview.

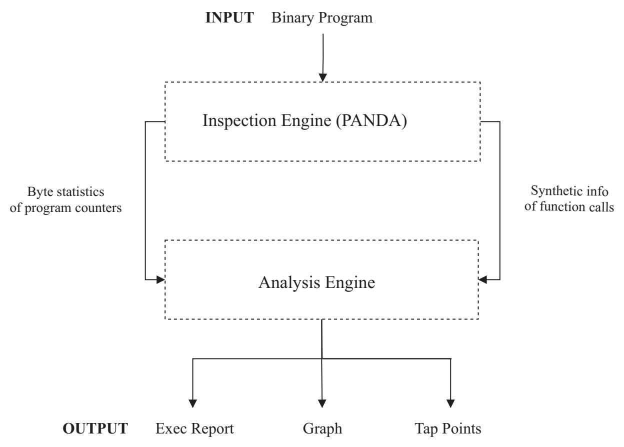

INPUT
Binary Program
Inspection Engine (PANDA)
Byte statistics of program counters
Synthetic info of function calls
Analysis Engine
OUTPUT
Exec Report
Graph
Tap Points

Figure 1. Architecture of CipherTrace.

Table 3. Synthetic information extracted by func_stats.

Description	From
The guest instruction count (unique identifier across the replay)	PANDA
The address space identifier	PANDA
The called function address; equivalent to the initial program counter	PANDA
The caller address; equivalent to the return address of the current stack entry	PANDA
The callers of the current function; features the blocks that have led to the function call	PANDA
The function calls themselves; not what has led to them (i.e. not the callstack)	PANDA
The number of all distinct basic blocks executed	PANDA
The highest number a basic block got executed	PANDA
The address of the most executed basic block	PANDA
The sum of all executions of all blocks; comparable with ‘llvm_bb’ field	PANDA
The total bytes read from memory	PANDA
The memory “reads” array (refer to Table 4 for more info)	PANDA
The total bytes written to memory	PANDA
The memory “writes” array (refer to Table 4 for more info)	PANDA
The number of times arithmetic operations were executed	Assembly
The number of times memory operations were executed	Assembly
The total number of instructions executed	Assembly
The number of all visited basic blocks	LLVM
The number of all visited functions	LLVM
The number of all visited modules	LLVM
The number of times memory allocation instructions were visited	LLVM
The number of times binary (or logical) operators were visited	LLVM
The number of times ‘call’ instructions were visited	LLVM
The number of times intrinsic instructions (e.g. ‘memcpy’ and ‘memmove’) were visited	LLVM
The number of times memory ‘load’ instructions were visited	LLVM
The number of times memory ‘store’ instructions were visited	LLVM
The total number of all visited instructions	LLVM

Inspection engine

Execution cycle

PANDA accomplishes emulation via basic block translation. PANDA’s Emulator—QEMU—generates a corresponding basic block of binary code via an Intermediate Language (IL), i.e. Tiny Code Generator (TCG), i.e. directly executable on the host to emulate the guest behavior. PANDA also supports LLVM IL, which is used to construct, optimize, and produce intermediate machine code, allowing the analysis to take place in a simplified but semantically equivalent domain, however at the expense of performance [26].

Memory tainting

Memory tainting is the process of tracking the propagation of flagged data in memory to determine the data flow. Dynamic binary analysis requires a taint-checking technique to instrument/inspect the data flow. PANDA leverages whole-system tainting similar to DECAF [27, 28] and DRAKVUF [29], in which it labels a memory buffer and tracks it along the exclusion.

Data collection

We categorize the collected information as platform-based (PANDA), assembly-based (from instructions profiling), or LLVM-based (from the lifted LLVM code). Unlike the related work, we also adopt LLVM-based binary (and logical) operators, which are essential for cross-platform support (see the results of our OPENSSLAES256 sample). LLVM-based information is quite dynamic and verbose, since it relies on the lifted code in the LLVM IL. In comparison, the platform-based information also relies on the assembled TCG IL. An IL is essential in dynamic analysis for emulating the guest instructions on the host. The output of func_stats is what we call a LOG. It is basically a file featuring lines of function calls (also known as records), each line represents a function call’s statistics and memory accesses in a JSON format (refer to Tables 3 and 4).

Table 4. Description of memory fields extracted by func_stats.

Field	Description
Base	The memory address where the memory buffer starts
Len	The length of the memory buffer
Entropy	Shannon’s entropy test for the memory buffer
Nulls	The number of nulls in the memory buffer
pc	The program counter of the memory buffer
PrintableChars	The number of printable characters in the memory buffer

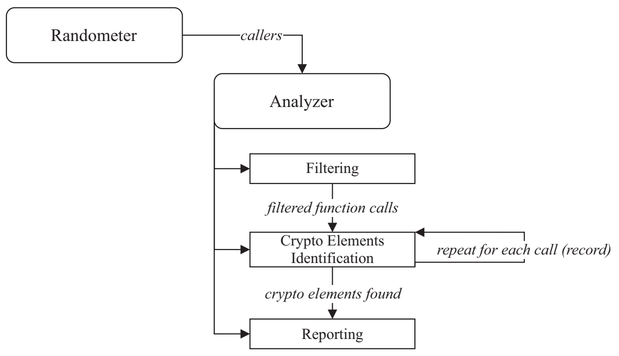


Figure 2. Overview of the analysis engine.

Analysis engine

An overview of the analysis engine (Fig. 2).

Measuring the callers

In the Randometer module, we apply the Chi-square randomness test to look for callers that have high-randomness (low Chi-square) write buffers and low randomness read buffers, indicating a cryptographic operation on an input. We start from callers of functions to account for control flow obfuscation, nested function calls, and to start the analysis from the highest level possible in the callstack. On top of reporting positive results, an empirical study analyzing the memory buffers in a PANDA replay using the Chi-square randomness test showed that, the test returns extremely low values (very close to 1.0) for encrypted data, and very high values (in the thousands) for encoded data [24]. We tested different values, and we found that 1000/10 000 for output/input buffers is the least likely to miss important data, and it also includes edge cases. If the look-up pair is reversed (i.e. 10 000/1000), one will be looking for decryption functions.

Stack analysis and filtering

In the Analyzer module, in the FILTERING stage of analysis, we aim to recover the most interesting part of the CFG. Therefore, after we filter by callers fed from the Randometer under which we have our candidates, we filter the main function calls that exhibit certain criteria. Provided a CFG, the main function calls are the closest nodes (with the highest number of executions and basic blocks) to the root node in the callstack—which is where the story begins. We take the ones whose write buffers exhibit high entropy (indicating compression, encoding, or cryptography). We use Shannon's entropy for measuring memory buffers. We use the filter >1, which indicates a slightly higher than normal entropy in a buffer of 256-bytes [24]. Our tests support such a choice as it avoids unnecessary noise. We found that 99% of the data sample has entropy of 0–3.9, and 63% has entropy of >1, where an entropy of 0–0.9 represents 24% of the sample. These 24% are either noise, or a nested function to the main one found in the >1 range. In addition to considering entropy, we also dismiss unnecessary function calls by looking for at least the following criteria:

- 1 loop (a repeated execution of a basic block's address).
- 1 arithmetic/bit-wise instruction.
- High count of basic blocks, and high number of executions of basic block(s).

And henceforth, a CFG is referred to as stack (its nodes are the function calls), and its levels is the stacksize. We use a stacksize of 3 (our default), which is thought to be the lowest it could be (as 2 produced a lot of false positives in our tests). Finally, we aggregate and group the function calls by function. The stack analysis and filtering algorithm is as follows:

Algorithm 1: Filter LOG records (R)

```
FILTER ($R$)
    inputs : All LOG records R
    output: The filteredStackRecs denoted by $F$
    distinctStackRecs is $N^*$
    foreach distinctStackRec $d_i \in N^*$ do
        f_write_entropy:= 1.0 ;
        foreach record $r_i \in R$ do
            if $r_{i_{functionstack}} = d_{i_{functionstack}}$ then
                tmpRec:= $r_i$;
                mem_writes:= $tmpRec_{writes}$;
                write_entropy_count:= 0;
                foreach mem_write $w \in$ mem_writes do
                    if $w_{entropy} >$ f_write_entropy then
                        write_entropy_count++;
                if write_entropy_count $> 1$ & $tmpRec_{insn\_arith} > 1$ &
                  $tmpRec_{maxexecs} > 1$ then
                    $F \overset{+}{\leftarrow} tmpRec$
    return $F$;
```

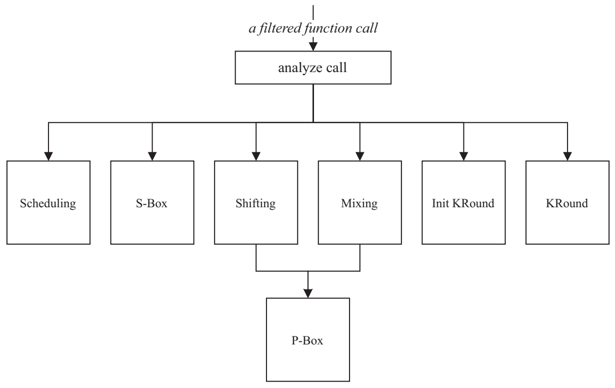


Figure 3. Overview of the crypto elements finder.

Identification of crypto elements

In this stage, we look for the operational components of a crypto algorithm in the stacks/CFGs we filtered, so that we can heuristically identify the class of the cipher algorithm. The main component in the crypto algorithm identification stage is finding the crypto elements. An overview of this component is shown in Fig. 3.

In the interest of space, we omit the details in the main algorithm for detecting and identifying crypto elements. However, we describe the crypto elements in-detail as follows:

Description 1: State .

A crypto state *is represented by the expression $s_{initial}..s_{final}$; starting from the initial state and ending with the final state, whereby $s_{initial}$ is the* plaintext *and s_{final} is the* ciphertext. *It mainly refers to the state of a memory buffer which manifests the internal state of a cipher [7]. It is expected that a function would read and write from/to the same buffer of the same size to perform a crypto operation, e.g. a substitution. Therefore, in a given function call, the* state *is the intersection of memory addresses in memory reads and writes of the same buffer length, wherein the entropy is higher or equal on write.*

Description 2: Key scheduling .

In a function call, one of the accessed memory buffers of a key scheduling element is the crypto key to be scheduled or expanded. So for the memory addresses that only exist in memory writes (i.e. a potential expanded key), if the length of memory writes is greater than the length of memory reads, wherein the entropy is higher on write, and there is a ratio between the write buffers and the maximum number of times a basic block got executed, then we have a candidate. Add to that, from our observations, the function call in question usually has a high number of arithmetic instructions–on a scale of 10 compared to other calls.

Description 3: Substitution box .

When a function call exhibits reading single bytes from a memory buffer to substitute a state *in place byte-by-byte. For a state of n bytes, each byte is accessed only once for the number of rounds that the cipher requires (x, e.g. 9 in* AES128 *excluding initial key round), in which we would find x number of function calls exhibiting the same features.*

Description 4: Mixing .

When a function call features the following criteria: it accesses the same state *memory address as other elements, and shares the same* caller *with any of them. It would differ in exhibiting a lower frequency in access patterns, as it does not need to substitute multiple*

Table 5. Crypto and hashing algorithms sample.

Design principle	*Algorithm*	*Operating system*
Permutation–substitution network	*AES-128*[1]	*Win7_x86*
Fiestal network	*Twofish-128*[2]	*Win7_x86*
Permutation–substitution network	*Serpent-256*[3]	*Win7_x86*
Permutation–substitution network	*Openssl-AES256*	*Win7_x86*[4]
		Debian_x64[5]
KGA via permutation-index pointers	*RC4*[6]	*Win7_x86*
Keyed Merkle–Damgård construction	*SHA256*[7]	*Win7_x86*
Noncryptographic arithmetic sequence	*DJB2*[8]	*Win7_x86*
Merkle–Damgård construction	*MD5*[9]	*Win7_x86*
Merkle–Damgård construction	*SHA256*[10]	*Win7_x86*

[1] https://github.com/ceceww/aes.git [2] http://www.cartotype.com/downloads/twofish
[3] https://github.com/JasonQSY/serpent.git [4] openssl 1.1.1d 10th Sep 2019
[5] OpenSSL 0.9.8o 01 Jun 2010 [6] https://github.com/ogay/rc4
[7] https://github.com/h5p9sl/hmac_sha256 [8] http://www.cse.yorku.ca/~oz/hash.html
[9] https://github.com/JackieTseng/md5 [10] https://github.com/okdshin/PicoSHA

times in place. And secondly, the bit-wise and arithmetic instructions are higher in Mixing *as it performs substitution (less than an S-Box) but with bit-wise or arithmetic instructions. In AES, it is manifested as matrix multiplication of columns, after permuting* (Shifting), *and substituting* (S-Box).

Description 5: Shifting .

Mixing *and* Shifting *crypto elements access the same memory addresses even with the same length, but* Mixing *has more "insn_arith" and "llvm_insn_store" statistics compared to* Shifting, *due to the associated processing that it carries out. Also, note the tandem in* Mixing *and* Shifting *in the AES algorithm, forming an AES flavoured Permutation-step [7].*

Description 6: Initial KRound .

This is a routine exhibiting a high number of bit-wise and arithmetic instructions, as it performs such operations on the state *and the expanded secret key. This occurs for x number of rounds, the same number of function calls. Also, if there is* Key Scheduling *, we mostly expect this element to exist or vice versa.*

Description 7: KRound .

This crypto element depends on the Initial KRound *crypto element. In fact, it is almost identical to* KRound *except for the former's dependency on the* Key Scheduling *element, as this element operates directly on the* state, *wherein its length must not be 0.*

In essence, our key-stone element is the state, which has to exist, as this is the buffer that the cipher performs its operations on, and that eventually becomes the ciphertext. Product ciphers perform a few core operations, i.e. a substitution step (S-Box), and a permutation step (P-Box). A substitution step is when we substitute an input's buffer byte-by-byte from a fixed table (Fixed S-Box). And when we substitute with logic (e.g. shift to the left or to the right), then we have a permutation step. There are two types of permutations we classified during our evaluation: one that involves heavy bit-wise and arithmetic operations (i.e. Mixing) and one that performs less of the same (i.e. Shifting). Finally, we survey the identified crypto elements conjointly to determine the class of the cipher. For example, if we found P-Box (Shifting), and S-Box (Substitution), then we have a candidate product cipher applying a substitution–permutation network, e.g. AES, Serpent, or TwoFish. We evaluate our design in the System design section.

Memory reconstruction

We use this technique to dump the secret key, plaintext, or ciphertext by the Analyzer module. We refer to them as 'key' and 'state' objects, respectively. In this technique, we reconstruct the memory buffers at different points in the execution and for different memory locations for a certain function caller and address space (e.g. a process). As soon as the Analyzer module identifies key and state elements, we could easily report the 'key' and 'state' objects relative to the crypto element, i.e. accessed by the function. In the Verifier module, we search in the tap point's memory buffers for some 'searchterms', in other words, what we expect to find. The Verifier module is a quick way to verify the results of our tool. See Discussion and future work section for more information on potential enhancements.

Evaluation

In the course of our experimental evaluation, we aim to demonstrate that we can detect and identify block ciphers and extract the secret keys using CipherTrace. We executed our evaluation on WSL v1, using a machine with 'Core i7-8550U, 1.8GHz, 16.0GB RAM' specification.

In the Crypto algorithms identification section , we test CipherTrace using four standard crypto algorithms known to be used in ransomware as well as in widely used software such as KeePass ([30]) and OpenSSL library ([31]). The crypto algorithms are three block ciphers and one stream cipher (RC4). One of the block ciphers (AES) was tested using two different key sizes and on two different operating systems and processor architectures. Additionally, we test four widely used hashing algorithms: one keyed (HMAC) and three unkeyed algorithms. All of them are cryptographic except for the unkeyed one (DJB2). In the Cryptovirological artefact analysis section, we test our tool against cryptovirological programs (i.e. ransomware) obtained from Threat Intelligence reports. More information on the sample is found in their respective sections. Finally, in the Evaluation with closely related works section, we cross-evaluate our results with the closely related works.

Crypto algorithms identification

We test hashing algorithms to evaluate the resilience of CipherTrace to noise and false positives. We also test the RC4 (stream cipher) to cross-check the findings with the rest of the sample, as we mostly target block ciphers. All the algorithms we use for evaluation are standard algorithms, or different implementations of them, and they are not obfuscated. In Table 5, we list the algorithms we use in our evaluation.

Table 6. Results of testing CipherTrace using different algorithms: CipherTrace could fully dissect Fixed S-Box block ciphers, additionally detect other ciphers and even extract a few of their crypto artefacts.

Program	Info	Elements	Buffers found
AES128	func_stats size: 115.3MB Duration: 10 min High-Arith Callers: 1	All	Secret Key (auto) Plaintext (auto) Ciphertext (auto)
SERPENT256	func_stats size: 112.4 MB Duration: 11–25 min/caller High-Arith Callers: 7	All	Secret Key (auto) Plaintext (auto) Ciphertext (auto)
TWOFISH128	func_stats size: 133.6 MB Duration: 7–46 min/caller High-Arith Callers: 9	No S-BOX	Secret Key (semiauto) Plaintext (semiauto) Ciphertext (n/a)
OPENSSLAES256	func_stats size: 633.8 MB Duration: 30–761 min/caller High-Arith Callers: 17	All	Secret Key (auto) Plaintext (n/a) Ciphertext (n/a)
RC4	func_stats size: 20.9 MB Duration: 7 min High-Arith Callers: 1	No S-Box	Secret Key (none) Plaintext (none) Ciphertext (none)
HMACSHA256	func_stats size: 106 MB Duration: 3 min High-Arith Callers: 4	Shifting Mixing Scheduling	Secret Key (none) Plaintext (none)
DJB2	func_stats size: 103 MB Duration: 3 min gh-Arith Callers: 2	Mixing	Secret Key (n/a) Plaintext (none)
MD5	func_stats size: 103 MB Duration: 3 min High-Arith Callers: 2	None	Secret Key (n/a) Plaintext (none)
SHA256	func_stats size: 108 MB Duration: 1 min High-Arith Callers: 2	None	Secret Key (n/a) Plaintext (none)

Serpent is well-known and identical to AES [32]. Hence, it should be easy to detect for those tools which could already detect AES. The tools that are unable to detect it are not tracing a cipher *per se*, limiting them by-design to certain algorithms or implementations. We test CipherTrace utilizing the PANDA replays we recorded, and using our PANDA plugin with CipherTrace's default configuration mentioned previously. We used a stack_limit of 200 (the default) to ensure that we start the analysis from the highest process possible in the stack (or process tree). From our tests, we realized that 32 was enough to dissect aes128, and 200 was more than enough for a complete overview of the system. In Table 6, we test CipherTrace against the crypto algorithms.

AES on Win7 and deb squeeze

CipherTrace has successfully classified AES as a block cipher and extracted and verified all the memory buffers (including the secret key) for aes128, as well as the secret key for opensslaes256. It only verified the secret key for opensslaes256 as it was the only buffer PANDA could locate in the replay. The replay of opensslaes256 was obtained by recording the use of OpenSSL in an extended browser session, hence the high replay size (and duration). OpenSSL's implementation of the AES algorithm is different from aes128. We also achieved same results by executing the AES algorithms on a different OS with different processor architecture (x64).

Serpent

serpent256 is the control algorithm in our experiment. It was successfully classified as a block cipher, and all its buffers were automatically extracted and verified.

TwoFish

In the twofish128 replay, the S-Box crypto element failed to be identified, because it is precomputed [33]. The current version of CipherTrace only identifies Fixed S-Box elements, and that will be further discussed in the Discussion and future work section. However, by using the synthetic information we extracted and the Randometer module of CipherTrace, we could extract the memory buffers.

RC4

In the RC4 replay, the S-Box crypto element failed to be identified, indicating the absence of—at least—a Fixed S-Box. However, a permutation step could be identified, which is something RC4 does in addition to key scheduling [34], which was also identified.

Hashing algorithms

All hashing algorithms tested were unkeyed except for hmacsha256, where potential secret keys could be explored. As per to the Design principles of our algorithms sample mentioned in Table 5, none employed a permutation–substitution network, indicating a product cipher [7]. Thus, S-Box, Shifting, or Mixing elements are not expected. After running CipherTrace, the absence of such elements in md5 and SHA256 was confirmed. Notably, a Mixing element was identified in DJB2, indicating a function accessing a state to perform bit-wise arithmetic operations and has lower frequency in access patterns compared to other functions that substitute or permute, aligning with our tested implementation.

HMACSHA256

We identified Shifting, Mixing, and scheduling elements. We expected to find a Key Scheduling element, even if we failed to extract a secret

Table 7. Recent and popular ransomware sample tested.

Name	Obfuscation	Encryption model	Data encryption	File encryption
Maze[11] [37–39]	Very high	3-tier	XORing, RC4	Cha-Cha
REvil[12] [40, 41]	High	4-tier	RC4	AES-128-CBC
Ryuk[13] [42, 43]	Low	3-tier	RC4	AES-256-CBC
Conti[14] [44, 45]	Medium	3-tier	Encoding	AES-256-CBC
Netwalker[15] [46]	High	3-tier	XORing, RC4	AES

[11]SHA256:dee863ffa251717b8e56a96e2f9f0b41b09897d3c7cb2e8159fcb0ac0783611b
[12]SHA256:3795a2228558a1b136746ea70125bc53cf05e2a6ce078d39667af4e3adee3a02
[13]SHA256:23f8aa94ffb3c08a62735fe7fee5799880a8f322ce1d55ec49a13a3f85312db2
[14]SHA256:eae876886f19ba384f55778634a35a1d975414e83f22f6111e3e792f706301fe
[15]SHA256:9c6d7dbe229d4257bc12df969637e773472892d80129416239d7a11edc7c7e82

Table 8. Analysis of the ransomware sample: CipherTrace could detect and classify cipher algorithms and extract secret keys and crypto states from all specimens in the sample.

Ransomware	E1	E2	E3	E4	E5	E6	E7	G	S	K
Maze	✓	✓	✓	✓	✓	✓	✓	36	1	33
REvil	✓	✓	✓	✓	✓	✓	✓	41	12	42
Ryuk	✓	✓	✓	✓	✓	✓	✓	28	1	36
Conti	✓	✓	✓	✓	✓	✓	✓	18	8	19
Netwalker	✓	✓	✓	✓	✓	✓	✓	15	9	23

E1: State. **E2**: Key Scheduling. **E3**: S-Box.
E4: Shifting. **E5**: Mixing. **E6**: Init KRound.
E7: KRound. ✓: Found at a different caller.
G: CFGs. **S**: State tap files. **K**: Key tap files.

key despite finding x10 more key artefacts compared to other algorithms. The reason is our implementations' dependency on finding a Fixed S-Box block cipher to extract the full key—which is intentional. However, as far as identifying Shifting and Mixing is concerned, we analyzed our sample and results, and found that this is anticipated, as a keyed Merkle–Damgård construction may perform permutation [35,36].

Summary

By using CipherTrace, we managed to identify all Fixed S-Box block ciphers we tested and extract their secret keys. When we tested stream ciphers (i.e. RC4), no S-Box element was identified, indicating that it is not a block cipher, and with other elements present, we deduced that it is a stream cipher. When testing hashing algorithms, no elements were identified for unkeyed hashing algorithms, and the Key Scheduling element was identified when testing a keyed hashing algorithm, indicating the presence of a secret key. The use of hashing algorithms might count as a proof-of-resilience to noise or false positives.

Performance

On average it takes around 8 min (per caller) to run the analysis and to locate and extract the memory buffers (including the secret keys)—the specifications of the machine used can be found above. The duration depends heavily on the replay size and the level of callstack, i.e. from which process the analysis starts, whether from the binary, from a process higher up in the tree, or even higher (from an OS process). In our evaluation we start the analysis from a given Address Space (i.e. a specific process), but the stack can go as far as our stack_limit can reach. All in all, our numbers demonstrate an improvement in performance compared to the closely related works assessed in this paper. More information can be found in the Evaluation with closely related works section.

Cryptovirological artefact analysis

To further demonstrate the efficacy of our system for the automation of malware analysis and reverse engineering, we apply it against the most recent and popular ransomware families listed in Table 7.

Obfuscation in the sample

All of the specimens perform binary, code, and data obfuscation. They also—at least—encrypt strings, Windows API names, and encode the ransomware note. We notate Maze with very high obfuscation as it tries to evade static and dynamic analyses by employing control flow obfuscation and a few techniques to evade detection when launching malicious code, starting processes, and calling the Windows APIs. It also utilizes anti-debugging, anti-VM, and anti-sandboxing checks and tricks. REvil encrypts the exploit code, in addition to its antianalysis features. Ryuk, Conti, and Netwalker encrypt the API names/calls as far as data encryption is concerned. However, Netwalker also encrypts the config in addition to the multi layers of obfuscation to evade an Anti Virus. It does also employ RC4 and XORing in data encryption compared to only encoding for Ryuk and RC4 for Conti—where the latter has less antianalysis features. We learned that Ryuk's code obfuscation (e.g. code flow obfuscation) resulted in a scattered CFG, which we had to account for at the expense of performance—i.e. 40% more analysis time.

CipherTrace ransomware results

We show the results of our experiment in Table 8. We were able to detect and classify the ciphers, and extract the secret keys and other forensic artefacts from the malicious binaries.

We identified crypto elements and artifacts in all tested ransomware binaries, producing numerous detailed CFG graphs for each. Additionally, we generated state and key tap files to extract potential keys and crypto states, containing various tap points based on replay length, data size, and configuration parameters, as outlined in Crypto algorithms identification section. Cryptographic elements in all samples were linked to the same function caller, indicating the cipher's main function location, except for Ryuk, where key operations were managed by a different caller.

Maze

We identified that the binary we used in our evaluation employs Cha-Cha [37], even if there are other variants that employ AES [47, 48]. We believe that we caught block cipher crypto elements such as S-Box and a permutation step because Maze encrypts the Cha-Cha key via RSA [37]—a block cipher [49]. Also, the definition of crypto elements we established is quite generic, and fits Cha-Cha's permutation [50]. We mostly differentiate between Shifting and Mixing from the bit-wise and arithmetic instructions ratio and dependency on other elements, facilitating the identification of Cha-Cha.

Other specimens

We were able to identify the crypto elements when block ciphers were employed in encrypting files or data. Netwalker showed a high number of Key Scheduling and KRound Init across the execution report, exhibiting very high key-related activity (i.e. expansion or reuse), and that was slightly different from the rest of the ransomware sample.

Verifying the sample's artefacts

This section discusses our tool's usage for analysts and victims of ransomware. Through in-depth reverse engineering of the Conti specimen—selected for its easily found hard-coded master public key in the binary—we proved our tool's efficacy and verified its results. Analysis of the specimen showed each encrypted file includes about 512 bytes of metadata: file size, a 32-byte file key, and a 16-byte salt, all encrypted with this public key.

Since CipherTrace does not yet fully support asymmetric block ciphers (e.g. RSA), and the specimens employ asymmetric cryptography to encrypt each file's key, we cannot automatically extract file keys. Therefore, after employing manual analysis, we extracted and verified a few crypto states (i.e. plain- and ciphertexts), as well as the RSA-4096 key, and the encryption key of a tainted file.

By using the Verifier module with the content of some honeypot files, we confirmed extracting plaintext content of several files. Additionally, the Verifier module highlighted interesting function callers and program counters, pinpointing intersections between key and state tap points for further analysis. This is important because the AES-256 key, which encrypts a file, is subsequently encrypted by the RSA-4096 key and stored in the same file, transforming the AES key into a crypto state (e.g. a ciphertext). The program counters reported by the Verifier module confirmed the RSA-4096 hard-coded public key of Conti, which was helpful to trace. This is promising for fully supporting asymmetric block ciphers as well. This public key was the most read 512 bytes by the conti.bin.eae8. process, which is expected as it reads the key for each file encryption operation to encrypt the file's metadata.

We further employed the file_taint plugin from PANDA, alongside the tainted_instr plugin, to trace writing metadata and the encryption operation for a honeypot file, aligning with our tool's findings. Moreover, through combining known information and our func_stats plugin's output, we identified several potential file encryption keys (32-byte memory buffers). However, attributing keys to specific files without file tainting or brute forcing remains challenging. In future versions, we will aim to group reporting potential secret keys with corresponding plaintext and ciphertext for easier verification.

System design

By adopting a modular design in the implementation of CipherTrace, we managed to achieve separation of concerns and allow for independent execution, development, enhancement, and maintenance.

This led to very well-designed configuration management. In each module, conceptually, we followed a divide and conquer approach in pursuing the module's objective. For instance, in the Analyzer module, we split the analysis into multiple stages to facilitate implementing, executing, and evaluating the concept, and make room for spot-on improvements. STACK ANALYSIS enabled us to reach our main function calls easier, and FILTERING allowed for performing a Control Flow analysis on their associated (or nested) function calls to start the CRYPTO ELEMENTS IDENTIFICATION stage with what we need.

The idea of looking for a state crypto element was very successful, as we could identify it in all of our samples, because hiding an internal state of a crypto algorithm is highly unlikely to be successful. In other words, we could always identify a memory buffer that increases in entropy and is affected by high bit-wise and/or arithmetic operations as the execution goes. However, as discussed in the Discussion and future work section, we may need to implement more advanced parameter reconstruction techniques to account for potential obfuscation of memory buffers accessed by the function call we analyze.

One may argue the naming of crypto elements is tied to how AES works. However, we could rename Shifting to Low P-Box, and Mixing to High P-Box, but at the expense of readability and not clearly showcasing our tool on an algorithm, i.e. used by more than 75% of ransomware samples out in the wild. Noteworthy that, in our implementation of these two elements, we made Mixing a prerequisite for Shifting to facilitate finding AES algorithms to easily fight the ransomware specimens propagating on the internet, but that neither affects the results of classifying a cipher (as both are considered permutations), nor extracting secret keys or crypto states at all. It can also be easily changed. The rationale is that, if a High P-box element was identified then, i.e. permutation, and if there is an additional weaker one, then i.e. AES's permutation.

Evaluation with closely related works

We compiled and tested the published tools from the closely related works against our crypto algorithms sample mentioned in Table 5 above. CipherXRay's authors did not publish their implementation, and therefore we could not test it against our algorithms sample, however we theoretically examined it against our tool. In Table 9, we compare CipherTrace against the tools that could identify crypto primitives and could also locate their secret keys, i.e. kerckhoffs, CipherXRay, Aligot, and K-Hunt. CipherTrace ties with kerckhoffs, which has the best performance for the task, while having a bigger scope and more functionality.

Due to following a DBI-based approach in the closely related works, at least Maze, REvil, and Conti might not have even launched due to their antianalysis capabilities. CipherTrace is a fully automated system, which differentiates it from the prior works. For instance, we executed the system on all test samples via a single shell script in one go, and CipherTrace was fully executed against each replay as per to the default configuration.

Table 9. Comparison with the closely related works: CipherTrace is the only one that employs a VMI dynamic analysis approach, is semantics-independent, and can detect and identify ciphers without using signatures or templates.

System	Approach	C1	C2	C3	C4	C5	Implementation tested	Test result	Avg. performance
Kerckhoffs	DBI-Heuristics	✗	✗	✗	✗	✗	✓	AES ✓ Serpent ✗	8 min
CipherXRay	DBI-Heuristics	✓	✓	✗	✗	✓	Not available	N/A	15 min [16]
Aligot	DBI-Heuristics	✗	✗	✗	✗	✗	✓	Nothing identified	Analysis unending (term. at 7 h)
K-Hunt	DBI-Heuristics	✓	✗	✗	✗	✓	Not applicable	N/A	N/A
CipherTrace	VMI-Heuristics	✓	✓	✓	✓	✓	✓	AES ✓ Serpent ✓ TwoFish ✓	8 min

C1: detects a proprietary cipher. **C2**: identifies a cipher class.
C3: extracts all crypto artefacts. **C4**: visualizes the CFG.
C5: no signatures or templates.

Discussion and future work

Core concept and approach

Our core concept is to identify the so-called crypto elements by utilizing the intrinsic properties of cryptography and the design differences between different ciphers. In the analysis, we rely on the internal state of crypto primitives to identify the crypto elements, because hiding the internal state is highly unlikely to succeed. We followed a VMI approach in dynamic analysis, and this helped to address antianalysis features of different binaries, and enabled stack analysis and filtering of function calls from a whole-system perspective employing higher-level semantics using the plugin we authored. Additionally, we take into account the higher-level semantics of a binary, i.e. memory buffers and functions in a controlled way that does not affect the analysis. The VMI approach, our PANDA plugin, the core concept, and our algorithms, made CipherTrace generic and capable of identifying proprietary cryptographic implementations, and extract their potential secret keys and crypto states. Our concept can be implemented differently, and we consider our algorithms the absolute minimum to catch crypto elements.

Notable limitations

Despite being capable of identifying the S-Box crypto element in Serpent, which is slightly different from AES in design and very different in the implementation we tested, the current version of CipherTrace is only compatible with Fixed S-Box elements. And this explains the failure to identify the S-Box crypto element in the twofish128 replay. Additionally, the current version mainly supports block ciphers, but with slight modifications it could also fully support stream ciphers as the evaluation demonstrated, since stream ciphers are also product ciphers [7]. Moreover, CipherTrace inherits PANDA's limitations, e.g. levels of callstack information. In our evaluation, we used a limit of 200 and we faced no issues. The limit is also customizable in CipherTrace.

False positives

We look for specific cipher design features [7], and i.e. why it is hard to see a high number of false positives in the results—which is our conceptual contribution. For instance, in DJB2 replay, we identified only a Mixing element, but not a cipher. With that said, even if the address space (i.e. process) is specified, the false positives produced are essentially the cascading function calls from/to the target. To address this, we use the appropriate stacksize and stack_limit criteria as discussed in the stack analysis and filtering in the Analysis engine section. One may argue an adversary may misdirect the analysis by creating a decoy block of code in the specimen (i.e. executed), which has a ultra-high loop count, or high number of bit-wise/arithmetic instructions. It is true that this will highly likely produce false positives for the crypto elements that filter for such criteria. However, we argue that this will not affect the overall results, as we are not only looking for a few elements in some specific functions or basic blocks. We produce a report for all function callers in the specimen with the called functions labelled with the crypto elements identified. And to increase confidence in the results, we run the analysis on multiple CFGs in parallel, collect the information from two intermediate representations, and aggregate the statistics on a basic block and a function level.

Information measurement

We could either apply different tests to measure information content (entropy) and distribution (randomness), and/or consider them collectively. We observed how sensitive the Chi-square test is, similar to the observations that Wang *et al*. made [24], which makes the Chi-square test an addition to be implemented as a randomness test in func_stats, in addition to Shannon's entropy test.

Parameter reconstruction

Since we depend on the memory buffers accessed by the filtered function calls in our analysis, as well as in extracting crypto artefacts, then we need to account for obfuscated memory buffers. Moreover, we need to be able to identify tables in memory (i.e. arrays and similar data structures), which would help in addressing the Pre-computed S-Box ciphers, e.g. in the TwoFish algorithm. This would also help in extracting the secret key in the RC4 replay. Also, since our core analysis algorithm does not depend on memory buffer sizes but rather only ratios, splitting the buffers or forcing the allocation of decoy buffer sizes is unlikely to affect the analysis results, but it would affect the functionality of extracting secret keys and crypto states, and that would be addressed in future versions.

More automation

In the task of validating the extracted potential secret keys, we could automatically test the extracted secret keys on the extracted state buffers. In spite of this being easier for some algorithms than for others, it still requires a lot of work and could be a potential upgrade for our tool in future versions. The rationale for such a feature is that the tool may extract dozens of memory buffers and the analyst does not know which of them are interesting—

unless they are reported as a group, certain information is tracked, or tainting is applied. In fact, to address that, we could even enhance our PANDA plugin (func_stats) to flag the potential secret key buffers, which end up in certain syscalls—i.e. to highlight that the keys being exfiltrated or communicated over a network interface, or stored on the file system or the system's registry. This will flag the potential keys used in the course of a ransomware activity.

Conclusion

We presented CipherTrace, an offline dynamic analysis system to identify the class of cipher algorithms and extract their secret keys and crypto states (plain- and ciphertexts). Our experimental evaluation demonstrated that, our novel technique of defining, identifying, and surveying crypto elements to classify a cipher—via VMI dynamic analysis approach—could fully dissect Fixed S-Box block ciphers. Additionally, it exhibited resilience to noise, a few obfuscation schemes, and different implementations of crypto algorithms. With the underlying dynamic analysis platform, CipherTrace inherently supports different operating systems and processor architectures. We compared our tool with the most effective and state-of-the-art tools to-date, using four standard cryptographic algorithms, which are often used in ransomware. Moreover, it detected and identified block ciphers in five malicious binaries (i.e. ransomware), and extracted various crypto artefacts including secret keys and crypto states (plain- and ciphertexts)—despite the very high binary, code, and data obfuscation employed by the specimens. It outperformed the closely related works such as kerchhoffs and Aligot in functionality and performance, and it is capable of identifying proprietary ciphers same as with CipherXRay and K-Hunt, but outperformed the former, and could identify a cipher class and extract all secret keys unlike the latter. Our system is fully automated, and proved very useful in malware analysis and reverse engineering. As shown in our evaluation, the core idea is applicable to different classes of ciphers, asymmetric cryptography, or crypto systems at large. Consequently, this paper highlights the substantial promise of automatically detecting and identifying cipher from execution traces and its vital role in neutralizing ransomware threats.

Author contributions

Mostafa AbdelMoez Hassanin (Conceptualization [Lead], Data curation [Lead], Formal analysis [Lead], Funding acquisition [Lead], Investigation [Lead], Methodology [Lead], Project administration [Lead], Resources [Lead], Software [Lead], Validation [Lead], Visualization [Lead], Writing – original draft [Lead], Writing – review & editing [Lead]), and Ivan Martinovic (Supervision [Supporting], Writing – review & editing [Supporting])

Conflict of interest: No competing interest is declared.

Funding

None declared.

Data availability

Our PANDA plugin (func_stats) has been merged with PANDA (master) [51], i.e. the official release. The source code of CipherTrace and the tested benchmark (including full execution logs) are publicly available in our repository [52].

Appendix

Ransomware sample

PANDA execution details for the ransomware sample can be found in Table A10.

Table A10. Analysis details of the ransomware sample.

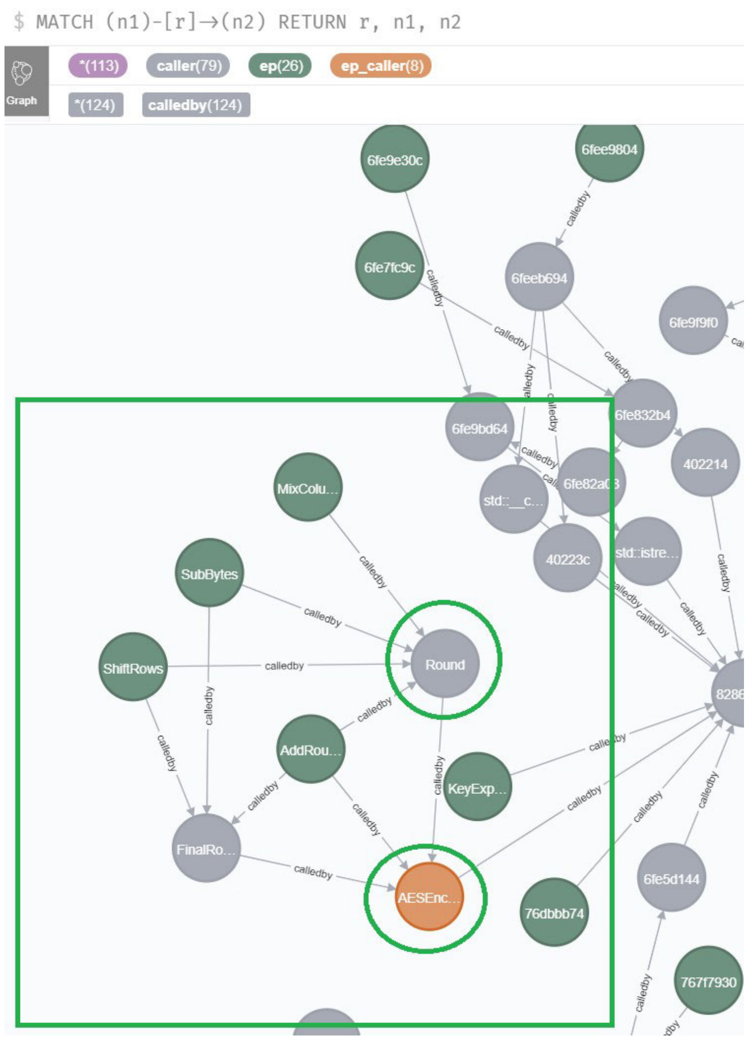
Name	Replay	unigram	func_stats	Randometer	Analyzer
Maze	561 s – 1.6 GB	11 h – 17GB	8 h – 5.2 GB (12% – Err)	106 callers (3 min/c)	1 min/caller (max 1 h)
REvil	277 s – 400 MB	17 h – 17GB	5.5 h – 3.8 GB (17% – Term)	163 callers (4 min)	0-3 min/caller (max 1.5 h)
Ryuk	240 s (reduced) – 1 GB	15 h – 25GB	18 h – 36 GB (3 asids, limit 32)	75 callers (<1 min/c)	1-2 min/caller (max few h)
Conti	1101 s – 800 MB	15 h – 17GB	100 h – 242 GB (Term)	43 callers (3 min)	3-15 min/caller (force timeout in 20)
Netwalker	230 s – 1.3 GB	3 h – 19GB	68 h – 145 GB (72% – Term)	150 callers (10 s)	2-15 min/caller (max 2 h)

Analysis CFG: AES256

The Control Flow Graph (CFG) for AES256 algorithm is illustrated in Fig. A4.

Analysis Log: AES256

Running CipherTrace's Randometer with unigrams output is in the following page.



```
$ MATCH (n1)-[r]→(n2) RETURN r, n1, n2
```

Figure A4. AES128 CFG. The rectangle marks the finding. The circle is the main function which calls the round routine.

```
Running CipherTrace's Randometer with unigrams output
randometer.py:main@45 [INFO] Starting Randometer for information measurement at <timestamp>
randometer.py:main@46 [INFO] Reading unigram read file unigram_mem_read_report.bin
randometer.py:main@52 [INFO] Reading unigram write file unigram_mem_write_report.bin
randometer.py:main@60 [INFO] Computing randomness of read buffers using Chi-Squared test...
randometer.py:main@63 [INFO] Computing randomness of write buffers using Chi-Squared test...
randometer.py:main@68 [INFO] Computing read buffer entropy...
randometer.py:main@71 [INFO] Computing write buffer entropy...
randometer.py:main@74 [INFO] Entropy reads: 20141 writes: 14683
randometer.py:main@77 [INFO] Applying read entropy mask > 0:
randometer.py:main@81 [INFO] Applying write entropy mask > 0:
randometer.py:main@87 [INFO] Applying read rand mask > 10000:
randometer.py:main@91 [INFO] Applying write rand mask < 1000:
randometer.py:main@105 [INFO] Results: reads: 191, writes: 9
randometer.py:main@140 [INFO] Callers for ASID 30249000 are set(['7740a5f0'])
randometer.py:main@142 [INFO] Randometer finished measuring information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at <timestamp>
Running CipherTrace's Analyzer for CALLER 7740a5f0 and REPLAY aes128 and ASID 30249000 with func_stats output
analyzer.py:main@15 [INFO] Starting Analyzer for a 115276032 bytes trace func_stats at <timestamp>
analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs, 3, 1.0
analyzer.py:main@34 [INFO] Reading the symbols file func_db
analyzer.py:main@40 [INFO] Reading the stats file func_stats
analyzer.py:main@50 [INFO] Filtering by caller
analyzer.py:main@52 [INFO] filteredByCaller count: 3
analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs, llvm_bb) that have maximum values
analyzer.py:main@57 [INFO] Find the main stack records (where it all begins, with max values of the maxexecs)
analyzer.py:main@60 [INFO] mainRecs count: 1
analyzer.py:main@66 [INFO] Exclude the main records from the filtered by caller ones be able to apply stack filtering.
analyzer.py:main@73 [INFO] Do the analysis for each comprehended caller stack
analyzer.py:main@98 [INFO] Stack filter: [u'828a9420', u'828a90ec', u'82897534']
analyzer.py:main@101 [INFO] Filter the records: those which share the stack filter, and certain stats (arith>1, loop>1, compc_write_entropy>1.0)
analyzer.py:main@107 [INFO] Create the graph file (for CFG)
analyzer.py:main@116 [INFO] Aggregate by maxexecs and groupby the entrypoint (function name)
analyzer.py:main@120 [INFO] Find the crypto elements, as per to certain traits
analyzer.py:main@148 [INFO] entrypoint 4014dc
analyzer.py:main@149 [INFO] maxexecs_addr 40152d
analyzer.py:main@153 [INFO] <----------Scheduling------------->
analyzer.py:main@148 [INFO] entrypoint 4015c1
analyzer.py:main@149 [INFO] maxexecs_addr 4015cf
analyzer.py:main@159 [INFO] <----------KRoundInit------------->
analyzer.py:main@165 [INFO] <----------KRound------------->
analyzer.py:main@171 [INFO] <----------S-Box------------->
analyzer.py:main@183 [INFO] <----------Shifting------------->
analyzer.py:main@148 [INFO] entrypoint 401606
analyzer.py:main@149 [INFO] maxexecs_addr 401613
analyzer.py:main@153 [INFO] <----------Scheduling------------->
[omitted]
analyzer.py:main@190 [INFO] Collecting state bases
analyzer.py:main@211 [INFO] Report overall stats
analyzer.py:main@212 [INFO] entrypoint of comprehended caller 76351fb7
analyzer.py:main@223 [INFO] Collect & report routines/functions, max 3 if there are duplicates
analyzer.py:main@264 [INFO] Overall routine reporting
analyzer.py:main@269 [INFO] Round routine is: Round(uchar ..) - 404d58
analyzer.py:main@274 [INFO] Main function is: AESEncrypt(uchar ..) - 402094
analyzer.py:main@280 [INFO] Collecting tap points data: states (ciphertexts or plaintexts)
analyzer.py:main@295 [INFO] Collecting tap points data: key(s)
analyzer.py:main@304 [INFO] Writing tap files (if any)
analyzer.py:main@319 [INFO] Analyzer finished analyzing trace of 115276032 bytes at <timestamp>
Duration (excl. verifier): 8 minute(s) for REPLAY aes128 and ASID 30249000 and CALLER 7740a5f0
Running CipherTrace's Verifier Light for REPLAY aes128 and ASID 30249000 with CipherTrace Analyzer's output and stringsearch
verifierlight.py:main@9 [INFO] Starting Verifier light at <timestamp>
verifierlight.py:main@12 [INFO] Reading matches file enc_string_matches.txt
verifierlight.py:main@17 [INFO] Reading .tap files in directory file .
verifierlight.py:main@23 [INFO] Reading tap file ./aes128_76351fb7_states.tap
verifierlight.py:main@23 [INFO] Reading tap file ./aes128_76351fb7_keys.tap
verifierlight.py:main@36 [DEBUG] Callers found: defaultdict(<type 'int'>, {'401f1a': 3, '401f4f': 21})
verifierlight.py:main@37 [DEBUG] PCs found: defaultdict(<type 'int'>, {'401500': 1, '401c65': 1, '401bce': 2})
verifierlight.py:main@39 [INFO] The verifier should find data series
verifierlight.py:main@42 [INFO] Verifier light finished searching PCs and Callers in 15 matches at <timestamp>
Running CipherTrace's Verifier with textprinter output
Running textprinter for TAP aes128_76351fb7_keys.tap and REPLAY aes128 and ASID 30249000 with CipherTrace Analyzer's output
verifier.py:main@63 [INFO] Data Series 1 was found in the read tap buffers, starting from line 15778 till 15794
verifier.py:main@63 [INFO] Data Series 2 was found in the read tap buffers, starting from line 15762 till 15778
verifier.py:main@68 [INFO] Verifier finished searching 3 lines in buffers at <timestamp>
Running textprinter for TAP aes128_76351fb7_states.tap and REPLAY aes128 and ASID 30249000 with CipherTrace Analyzer's output
verifier.py:main@63 [INFO] Data Series 1 was found in the read tap buffers, starting from line 0 till 16
verifier.py:main@67 [INFO] Data Series 3 was found in the write tap buffers, starting from line 0 with till 16
verifier.py:main@68 [INFO] Verifier finished searching 3 lines in buffers at <timestamp>
Duration: 10 minute(s) for REPLAY aes128 and ASID 30249000 and Caller 7740a5f0
```

References

1. Turton W, Mehrotra K. Hackers breached colonial pipeline using compromised password. Bloomberg, 2021.
2. Lyngaas S. US Marshals service still recovering from february ransomware attack affecting system used by fugitive hunters. CNN, 2023.
3. Langlois P. Data Breach Investigations Report. Verizon Business, 2020.
4. PurpleSec. 2021 ransomware statistics, data and trends. Washington, 2021.
5. Sophos. The State of Ransomware 2023. Abingdon: Sophos, 2023.
6. Berrueta E, Morato D, Magaña E,. *et al*. A survey on detection techniques for cryptographic ransomware. *IEEE Access* 2019;7:144925–44. https://doi.org/10.1109/ACCESS.2019.2945839.
7. Ferguson N, Schneier B, Kohno T. *Cryptography Engineering: Design Principles and Practical Applications*. Hoboken: Wiley Publishing, 2010.
8. Nakashima E, Lerman R. FBI held back ransomware decryption key from businesses to run operation targeting hackers. Washington: The Washington Post, 2021.
9. Or-Meir O, Nissim N, Elovici Y,. *et al*. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput Surv* 2019;**52**:1–48.

10. Song DX, Brumley D, Yin H,. *et al.* BitBlaze: a new approach to computer security via binary analysis. In: R Sekar, AK Pujari (eds), *Proceedings of the Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Vol. 5352 of Lecture Notes in Computer Science*. Berlin: Springer, 2008, 1–25
11. D'Elia DC, Coppa E, Nicchi S,. *et al.* SoK: using dynamic binary instrumentation for security (and how you may get caught red handed). In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, Asia CCS'19*. New York: Association for Computing Machinery, 2019, 15–27.
12. Hosseinzadeh S, Rauti S, Laurén S,. *et al.* A survey on aims and environments of diversification and obfuscation in software security. In: *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016, CompSysTech'16*, New York: Association for Computing Machinery, 2016, 113–20.
13. Lutz N. Lutz towards revealing attackers – intent by automatically decrypting network traffic. Master's Thesis, ETH Zürich, 2008.
14. Wang Z, Jiang X, Cui W,. *et al.* ReFormat: automatic reverse engineering of encrypted messages. In: *Proceedings of the 14th European Conference on Research in Computer Security, ESORICS'09*. Berlin, Heidelberg: Springer-Verlag, 2009, 200–15.
15. Gröbert F, Willems C, Holz T. Automated identification of cryptographic primitives in binary programs. In: R Sommer, D Balzarotti, G Maier (eds), *Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer, 2011, 41–60.
16. Li X, Wang X, Chang W. CipherXRay: exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE Trans Depend Secure Comput* 2014;**11**:1.
17. Calvet J, Fernandez J, Marion JY. Aligot: cryptographic function identification in obfuscated binary programs. In: *Proceedings of the ACM Conference on Computer and Communications Security*. New York: ACM, 2012, 169–82.
18. Hosfelt DD. Automated detection and classification of cryptographic algorithms in binary programs through machine learning. Master's Thesis, Johns Hopkins University, 2015.
19. Xu D, Ming J, Wu D. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos: IEEE Computer Society, 2017, 921–37.
20. Hill G, Bellekens X. CryptoKnight: generating and modelling compiled cryptographic primitives. *Information* 2018;**9**. https://doi.org/10.3390/info9090231.
21. Lestringant P. Identification of cryptographic algorithms in binary programs. (Identification d'algorithmes cryptographiques dans du code natif). Ph.D. Thesis, University of Rennes 1, 2017.
22. Luk CK, Cohn R, Muth R,. *et al.* Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not* 2005;**40**:190. https://doi.org/10.1145/1064978.1065034.
23. Li J, Lin Z, Caballero J,. *et al.* K-Hunt: pinpointing insecure cryptographic keys from execution traces. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, New York: Association for Computing Machinery, 2018, 412–25.
24. Wang R, Shoshitaishvili Y, Kruegel C,. *et al.* Steal this movie: automatically bypassing DRM protection in streaming media services. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington: USENIX, 2013,. 687–702.
25. Lin Z, Jiang X, Xu D,. *et al.* Automatic protocol format reverse engineering through context-aware monitored execution. In: *Proceedings of the 15th Symposium on Network And Distributed System Security (NDSS)*. Reston: The Internet Society, 2008.
26. Dolan-Gavitt B, Hodosh J, Hulin P,. *et al.* Repeatable reverse engineering with PANDA. In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW-5*. New York: Association for Computing Machinery, 2015.
27. Henderson A, Prakash A, Yan LK,. *et al.* Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*. New York: Association for Computing Machinery, 2014, 248–58.
28. Davanian A, Qi Z, Qu Y,. *et al.* DECAF++: elastic whole-system dynamic taint analysis. In: *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Beijing: USENIX Association, 2019, 31–45.
29. Lengyel TK, Maresca S, Payne BD,. *et al.* Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC'14*. New York: Association for Computing Machinery, 2014, 386–95.
30. KeePass Password Safe. Dominik Reichl, 2003.
31. The OpenSSL Project. OpenSSL Project, 1998.
32. Biham E, Anderson R, Knudsen L. Serpent: a new block cipher proposal. In: S Vaudenay (ed.) *Fast Software Encryption*. Berlin, Heidelberg: Springer, 1998, 222–38.
33. Schneier B, Kelsey J, Whiting D,. *et al. The Twofish Encryption Algorithm*. Hoboken: John Wiley & Sons, 2000.
34. Paul G, Maitra S. *RC4 Stream Cipher and Its Variants*. Boca Raton: CRC Press, 2011.
35. Hirose S, Park J, Yun A. A simple variant of the Merkle–Damgård scheme with a permutation. *J Cryptol* 2007;**25**:113–29.
36. Coron JS, Dodis Y, Malinaud C,. *et al.* Merkle-Damgård revisited: how to construct a hash function. *Adv Cryptol* 2005;**3621**:430–48.
37. Mundo A. Ransomware Maze. San Jose: McAfee, 2020.
38. DSCI. Maze Ransomware. Technical Report. Noida, 2020.
39. SentinelLABS. Case study: catching a human-operated maze ransomware attack in action. Mountain View, 2020.
40. Intel471. REvil ransomware-as-a-service: an analysis of a ransomware affiliate operation. Wilmington, 2020.
41. Fakterman T. REvil/Sodinokibi: the crown prince of ransomware. La Jolla: Cybereason, 2019.
42. ANSSI. RYUK RANSOMWARE. Paris: CERT-FR, 2021.
43. Cohen I, Herzog B. A targeted campaign break-down – Ryuk Ransomware – check point research. San Carlos: Check Point Software Technologies LTD, 2018.
44. VMWare Security Blog. TAU Threat Discovery: Conti Ransomware. Palo Alto, 2020. https://blogs.vmware.com/security/2020/07/tau-threat-discovery-conti-ransomware.html (4 March 2021, date last accessed).
45. BleepingComputer.com. Conti ransomware shows signs of being Ryuk's successor. New York, 2020.
46. NAKIVO. All you should know about netwalker ransomware. Sparks, 2020.
47. Keijzer N. The new generation of ransomware - an in depth study of ransomware-as-a-service. Student Thesis, University of Twente, 2020.
48. NHS. 2020. https://digital.nhs.uk/cyber-alerts/2020/cc-3681 (4 November 2021, date last accessed).
49. Rivest RL, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems. *Commun ACM* 1978;**21**:120. https://doi.org/10.1145/359340.359342.
50. Yadav P, Gupta I, Murthy SK. Study and analysis of eSTREAM cipher Salsa and ChaCha. In: *Proceedings of the 2016 IEEE International Conference on Engineering and Technology (ICETECH)*. IEEE: Piscataway, 2016, 90–94.
51. mabdelmoez. Integrate func_stats plugin: collect synthetic information of called functions by mabdelmoez. Pull Request #801 – panda-re/panda. GitHub, 2020.
52. mabdelmoez. mabdelmoez/ciphertrace: CipherTrace: automatic detection of ciphers from execution traces to neutralize ransomware. Los Gatos: CipherTrace, 2021.