R+R: A Systematic Study of Cryptographic Function Identification Approaches in Binaries

Yongming Fan
Purdue University
fan322@purdue.edu

Priyam Biswas
Intel
priyam.biswas@intel.com

Christina Garman
Purdue University
clg@cs.purdue.edu

2024 Annual Computer Security Applications Conference (ACSAC) | 979-8-3315-2088-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACSAC63791.2024.00089

***Abstract*—Cryptographic functions are instrumental in securing our electronic communications and systems; yet time and time again they are mis-used, mis-implemented, or created in an ad-hoc manner. Additionally, while cryptography plays a fundamental role in securing systems, it is unfortunately also used for malicious purposes, such as hiding payloads in malware. Many such instances occur in closed-source code or binary applications, which inherently present a challenge for independent audit and analysis. Therefore, detecting the presence of cryptographic functions in a binary application can be both an indicator of malicious behavior as well as a point of interest for cryptographic analyses and vulnerability discovery.**

While general purpose binary analysis and function identification techniques are themselves broad and thriving areas that could help solve these problems, a variety of work across industry and academia has focused on a subset of this space: developing techniques and tools that are specifically tailored to identifying different cryptographic primitives in binary applications. Despite the relative popularity of this work and recent advances in the space, it already lacks consistent means of evaluation or comparisons across tools. As such, we set out to conduct comprehensive reproduction and replication studies on all existing work in the space, from multiple perspectives. We noticed there is a significant gap in comparing tools, as there is no standardized testing framework allowing one to easily compare and contrast their strengths and weaknesses on a level playing field. As such, to complement the traditional R+R studies, we developed a comprehensive testing and evaluation framework which includes a number of modern cryptographic algorithms and real world examples, that allows for the comparison of both existing and future work in a uniform manner. We then carried out reproduction and replication studies, using both their benchmarks and ours. Finally, based on our insights from the studies, we highlight major gaps in existing work, especially as they relate to modern cryptographic primitives and real-world use cases, and discuss a variety of important avenues for future work.

***Index Terms*—Software Security, Cryptography, Binary Analysis**

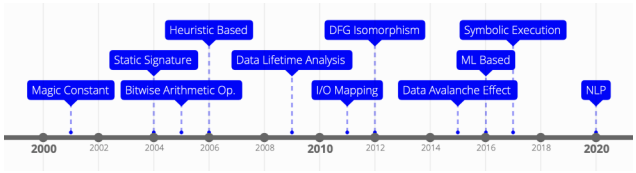

Figure 1. Evolution of cryptographic function identification techniques

1. Introduction

Cryptography has found itself at the core of secure modern technology and both plays a key role in securing communications and is imperative to many modern systems and applications. From data integrity to authentication and online banking to messaging friends, cryptography is everywhere. Well-known cryptographic libraries, such as OpenSSL [1], are widely used not only to generate TLS certificates and validate certificate information, but also to implement cryptography in general purpose applications. Despite their universal benign usage, cryptographic functions are also heavily used in malware and to evade security protocols. In recent ransomware attacks [2], [3], cryptographic functions have been used to encrypt a victim's information and later asked to pay ransom in exchange of recovering their files and information.

On the one hand, the usage of cryptographic functions makes it easier to carry out secure and private operations, and on the other hand, its malicious usage by bad actors makes it harder for cryptography and security experts to perform forensic analyses and reverse engineer code. Hence, the ability to automatically identify or detect the presence of cryptographic functions in binary applications can be a crucial part of the security experts' arsenal. It can assist in binary analysis to give a better depiction of how the functions work. Determining the type of cryptographic function in a binary can help to pinpoint the existence of a malicious payload [4] and also help cryptographic experts identify potentially insecure protocols or implementations (or those that require more manual analysis).

While general purpose binary analysis and function identification techniques are themselves broad and thriving areas that could potentially solve these problems, a variety of work across industry and academia has focused on solving these challenges more specifically: developing techniques and

2576-9103/24/$31.00 ©2024 IEEE
DOI 10.1109/ACSAC63791.2024.00089

tools that are uniquely tailored to identifying *cryptographic primitives* in binary applications. This allows researchers to often provide better results in different domains, leveraging cryptography-specific features. Cryptographic functions tend to have many mathematical computations, nested loops, and exclusive input-output mappings which are distinct from non-cryptographic functions and can thus be useful as a means of identification. Harvey et al. [5] first utilized magic constants to identify cryptographic primitives in 2001. Later works focused on signature based detection mechanisms. However, due to the limitations of signature based approaches, subsequent work placed a greater emphasis on heuristic based detection, such as identification of basic blocks, instruction patterns, etc. With the popularization of modern machine learning algorithms, newer works utilized deep learning and AI to extract cryptographic features from a binary.

Unfortunately, while there has been a wealth of work in the space, it has generally failed to keep up with modern cryptographic innovations and applications, as well as current software engineering and optimization techniques. This can be difficult to notice as there is no standardized benchmarking or comparison framework, and individual works tend to test on algorithms that they are best at identifying, while neglecting those they cannot. Additionally, as many techniques are tailored for specific cryptographic algorithms, architectures, or settings, which change rapidly over time, reproducing past results can be challenging, making it difficult to provide comparisons across the growing landscape as well.

Given how important these tools and techniques can be for both researchers and practitioners, we aim to reproduce and replicate all existing tools to date in this domain. In this endeavor, we also offer a comprehensive analysis that categorizes and discusses the existing body of work, highlighting the strengths, weaknesses, and techniques employed. We also seek to help drive future research in the area by identifying potential challenges and various factors that may influence performance. To help provide for a more even comparison of existing and future work, and to identify gaps that are in need of future research, we create a benchmarking framework featuring a number of modern cryptographic primitives and applications. We then use this framework to evaluate existing work. Finally, throughout the paper, we discuss and motivate a variety of open research problems in the space.

Our contributions. Our primary contributions are:

- We conduct a thorough examination of cryptographic function identification methods, exploring the range of techniques utilized and evaluating their respective strengths and limitations.
- We create a standardized suite of performance metrics and benchmarks to evaluate the effectiveness of current detection mechanisms and analyze existing tools based on this suite.
- We conduct comprehensive replication and reproduction studies on all existing work.

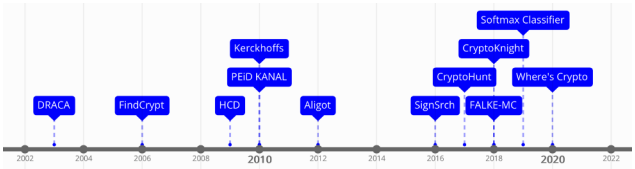

Figure 2. Timeline of the development of cryptographic algorithm detection tools

- We provide an in-depth analysis of our findings and categorize the tools according to their performance across various dimensions.
- Based off of this analysis and our study of existing work, we discuss the research gaps in this domain and propose directions for future work.

2. Background

In this section, we provide background on and a classification of detection approaches that have been employed by existing tools to date. Figure 1 shows the evolution timeline of various techniques and approaches used in cryptographic function identification research whereas Figure 2 shows the development timeline of all studied tools. We also provide an overview of some of the cryptographic features that existing tools focus on, to help better understand how they work and contextualize later discussion.

Cryptographic primitive vs. cryptographic function. A brief note on notation used throughout this paper. When we refer to a *cryptographic function* or *cryptographic algorithm*, we are generally referring to the entirety of a specific scheme or class of algorithms, such as RSA or hash functions. When we say *cryptographic primitive*, we generally mean a building block for a larger cryptographic scheme, such as a Feistel network.

2.1. Categorization of detection approaches

Precision, scalability and performance of an identification approach all rely on the underlying techniques used. Detection approaches can be divided into three categories: i) Dynamic, ii) Static, and iii) Machine learning based approaches.

Static Approaches. Static approaches perform various forms of static analysis to detect any static signatures, such as 'magic' constants, instruction sequences, and different code structures such as S-boxes, as a means to identify cryptographic algorithms. This type of approach is based on the assumption that cryptographic functions will perform a large number of arithmetic computations compared to non-cryptographic functions. Harvey et al. [5] first proposed the idea of identifying cryptographic algorithms in binary files. They focused on finding algorithms based on their constant characteristics. By taking advantage of feature matching, subsequent works [6], [7], [8] primarily focused on protocol reverse engineering. Chang et al. [9] created a library of

3000 plus signature characteristics of cryptographic algorithms and also explored recognizing cryptographic algorithms based on Minimum Perfect Hash Function (MPHF). Lestringant et al. [10] used data-flow isomorphism to find symmetric key algorithms.

Off-the-shelf tools such as Draft Crypto Analyzer (DRACA) [11], Kanal [12], Kerckhoffs [13], Hash & Crypto Detector (HCD) [14], Signsrch [15], and Findcrypt [16], utilize static signature patterns of different cryptographic functions. Static based approaches have low performance overhead. However, these detection mechanisms can be easily bypassed using simple obfuscation techniques. For example, the authors of Aligot [17] showed that simple data obfuscation techniques can bypass static analysis based detection mechanisms.

Dynamic Approaches. Dynamic approaches [18], [17], [19] primarily focus on identifying cryptographic primitives from execution traces. Lutz et al. [20] first applied dynamic analysis to identify cryptographic algorithms based on three indicators: i) presence of loops, ii) changes in entropy, and iii) high ratio of bitwise arithmetic instructions. Based on the data avalanche effect, CipherXRay [19] pinpoints the boundary of cryptographic operations and recovers transient cryptographic secrets. However, this approach does not work in the case of stream ciphers, for example, as they do not show any data avalanche effect.

Gröbert et al. [21] proposed heuristic based approaches on both generic characteristics of cryptographic code and on signatures for specific instances of cryptographic algorithms by mapping input-output (I/O) relations. Aligot [17] further extends this idea of I/O mapping. It retrieves I/O parameters in an implementation-independent fashion, and compares them with known cryptographic functions as well as performs an interloop data flow analysis. CryptoHunt [18] used bit-precise symbolic loop mapping to identify cryptographic functions and applies guided fuzzing to make the solution scalable. Park et al. [22] proposed a hardware-assisted tracing technique to detect symmetric-key cryptographic routines in anti-reverse engineered binaries via recording the change of flow instructions from the CPU at run-time. Dynamic approaches in general perform better than static based approaches in obfuscated binaries, but suffer from much greater performance overhead.

Machine Learning Based Approaches. With the growing popularity of machine learning, several recent works have explored utilizing such techniques. Shin et al. [23] discussed how recurrent neural networks can identify functions in binaries with greater accuracy and efficiency. However, their approach was generalized for any function identification. For the purpose of cryptographic function identification, Wright et al. [24] proposed an artificial neural network model to classify functional blocks as being either cryptographic or not by extracting the frequency of different logic instructions from a disassembled program. Benedetti et al. [25] used the 'grap' tool to detect cryptographic algorithms by creating patterns for AES and ChaCha20. Falke [26] proposed a neural network based approach by modeling classifiers for arbitrary cryptographic algorithms from sample files and then automatically extracting features to train the neural network. It offered a high detection rate in combination with a low false positive rate. Hill et al. [27] proposed a Dynamic Convolutional Neural Network based learning system (CryptoKnight) which learns from new cryptographic execution patterns to classify unknown software. Jia et al. [28] proposed an NLP-based approach which first extracts the semantic information of assembly instructions and then transfers them into 100-dimensional vectors and later uses K-Max-CNN-Attention to classify cryptographic functions.

2.2. Cryptographic Features

Cryptographic functions have their own (often quite distinct) set of characteristics, and work thus far has often leveraged these for identification, as they can lead to more tailored or performant techniques than general purpose binary analysis and code identification. Many of these features only pertain to certain algorithms and cannot be used to identify others. We now discuss some of the popular cryptographic features used for identification purposes.

Magic Constants. In this approach, algorithm specific "magic constants" are searched for in the binary. These are constant values that must occur in any implementation of a given algorithm. For instance, Signsrch [15] relies on the magic number '0x9e3779b9' to detect the TEA algorithm; however, it fails to identify the algorithm when data obfuscation is applied [18].

Presence of Loops. Most cryptographic functions use some form of loops for key generation, encryption, or other operations. CryptoHunt [18] utilized loops as a means for detection. Given that loops may present in regular functions, further program analysis is necessary to ensure accurate detection.

Changes in Entropy. The process of decryption usually decreases information entropy. Hence, entropy can be used as an indicator of encrypted versus plaintext data. Several techniques [20], [29] have used entropy to distinguish encrypted blocks from regular ones. However, relying on entropy can also lead to false positives in certain circumstances [21].

I/O Mapping. Cryptographic functions sometimes have one-to-one mappings between their input to output, i.e., given a key and the plaintext, we might always get the same output, or given the input to a hash function, we should always get the same output. However, these approaches rely on the accurate extraction of the key and input/output from the memory, as any wrong extraction can defeat the usefulness of unique mapping.

Data-Flow Isomorphism. In this approach [10], a Data-Flow Graph (DFG) [30] is generated from the corresponding assembly code, and subgraphs in the DFG are checked to see whether they are isomorphic to the graph signature of a particular cryptographic algorithm. However, this approach is limited in the case of conditional statements, which are more prominent in asymmetric algorithms. Additionally,

DFG can vary in effectiveness depending on data obfuscations techniques used in the implementation, such as data-splitting.

S-Box. Substitution boxes (S-Box) [31] are a fundamental component used in many symmetric key ciphers. For instance, the BLOWFISH algorithm is a 64-bit block cipher that has an S-Box consisting of 1024 elements [28] which are usually sequentially stored in memory, allowing tools to detect BLOWFISH by locating this S-Box in memory. Similar techniques can be applied to Permutation Boxes (P-Box) [32], which are methods of bit-shuffling to permute or transpose bits across S-box inputs.

Instruction Sequences. An instruction sequence is a series of instructions or commands that a CPU executes in a specific order to perform tasks within a program. Many fundamental cryptographic operations will consist of fingerprintable instruction sequences and hence can be used for the purpose of identification. Nevertheless, this method may lose its effectiveness when dealing with control-flow obfuscations such as control-flow flattening [33].

3. Overview

This paper focuses on comprehensive replication and reproduction studies for existing work on identifying cryptography code in binary applications. To aid in this, we propose a novel analysis framework aimed at facilitating a comprehensive evaluation of cryptographic function detection tools' replicability. Our framework is introduced in Section 4. Subsequently, we conduct two evaluations: a reproducibility evaluation in Section 5, and a replicability evaluation in Section 6. Additionally, we provide further discussion on the results of our reproducibility and replicability evaluations, as well as point out a number of important takeaways and directions for future work, in Section 7. Finally, to support future research in this domain, we present a summary cryptographic detection approaches and tools in Section 2 as part of our background.

3.1. Cryptographic Algorithm Detection Tools

A number of cryptographic detection tools have been developed based on the different approaches discussed in Section 2.1. We perform our reproduction and replication studies on all existing work that we were able to find from 2000 until now:

- Aligot [17]
- CryptoHunt [18]
- CryptoKnight [27]
- DRACA [11]
- FindCrypt2 [16]
- FALKE-MC [26]
- HCD [14]
- Kerckhoff [13]
- PEiD KANAL [12]
- SignSrch [15]
- Softmax Classifier [28]
- Where's Crypto [34]

Our focus is on evaluating tools specifically designed for identifying cryptographic functions and comparing these tools with their stated performance. As such, we exclude other (broader) binary analysis tools, such as Binshape [35], a general-purpose binary analysis tool, or CIS [36], which focuses on testing cryptographic heuristics rather than actual algorithms.

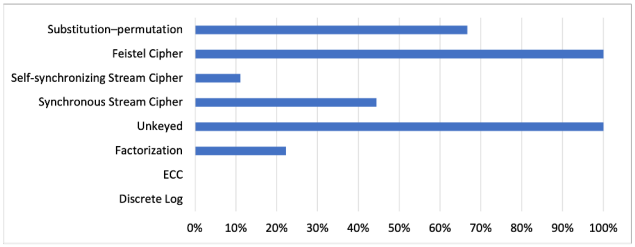

Figure 3. Types of cryptographic functionality and designs targeted by different tools and approaches, by percentage of tools surveyed

We also employ a systematic approach to categorize, classify, and interrelate the diverse array of knowledge elements that constitute such tools from several directions in Section 3.2. Due to space constraints, we include brief discussions of each tool, along with their strengths and weaknesses, in Appendix C.

We comprehensively summarize the detection abilities of each cryptographic function detection tool. We did a thorough check of each tool's documents and signatures to see what types of cryptographic algorithms they can find. Figure 3 shows the distribution of various algorithms that existing work has focused on. After that, we gather all the information we found and explain it in more detail in Table 8. This helps us better understand what each tool can do in terms of detecting different cryptographic algorithms.

Note that we compiled these results based on information provided by the developers of each tool. Through our comprehensive analysis, we found that commercial tools like DRACA [11], FindCrypt2 [16], and Signsrch [15] can detect a diverse range of cryptographic algorithms. However, their performance falter when encountering obfuscation or atypical situations. Certain academic-developed tools such as Aligot [17], CryptoHunt [18], and Where's Crypto [34] offer superior analysis results in specific scenarios, such as detecting cryptographic algorithms from obfuscated binaries. Many of them can only detect a few cryptographic algorithm, but they can be easily expanded.

It is important to recognize that a simple summary may not fully capture the current capabilities and detection abilities of each tool. Moreover, many of these tools were developed years ago, and they may not be compatible with current programming languages, compilers, and/or hardware architectures. This is why our reproduction and replication study is so crucial. It not only highlights these major gaps in existing work as the space has progressed, but allows us to provide insights on the space more broadly, and on a variety of important avenues for future work.

Tool	Method	Dependent Library	Language	Requires Compiling	Expandable	R+R	Obf. Binary Design	AI/ML Approach
Aligot	I/O Parameters	Intel Pin	Python	Yes	Yes	Reproducible	No	No
CryptoHunt	Bit-precise Symbolic Loop	Intel Pin	C++	Yes	Yes	Unable	Yes	No
CryptoKnight	Machine Learning Based	Intel Pin, PyTorch	Python	Yes	Yes	Both	No	Yes
DRACA	Unknown	None	Executable	No	No	Replicable	No	No
FindCrypt2	Magic Constants	IDA Pro	C++	Yes	No	Replicable	No	No
FALKE-MC	Neural Network	None	Unknown	Unknown	Unknown	Unable	Yes	Yes
HCD	Unknown	None	Executable	No	No	Replicable	No	No
Kerckhoff	Multiple	Intel Pin	Python	No	Yes	Unable	No	No
PEiD KANAL	Signature Searching	None	Executable	No	Yes	Replicable	No	No
Signsrch	Signature Searching	None	Executable	No	No	Replicable	No	No
Softmax Classifier	Neural Network	Unknown	Python	Unknown	Unknown	Unable	No	Yes
Where's Crypto	DFG Isomorphism	IDA SDK	C++	Yes	Yes	Both	No	No

Table 1. CATEGORIZATION OF EXISTING TOOLS AND THEIR IMPORTANT PROPERTIES

3.2. Categorization of Cryptographic Algorithm Detection Tools

We employ a systematic approach to categorize, classify, and interrelate the diverse array of knowledge elements that constitute such cryptographic detection tools from several directions. Table 1 contains an overview and summary of each tool, including its reproduction and replication status in this work, as well as a variety of information such that one can easily find or organize existing (and future) tools based on the per-column categories.

We begin by categorizing the detection approach (broken down by the different methods discussed previously in Section 2.1) and the programming language in which the tool's source code is written. We note that some tools are not open source; they provide only an executable file, and as a result, we lack information about the source language. Some tools require specific dependent libraries to run, which can limit or hinder their usability, and as such we list if there are any required dependencies. Some tools are installation-free and do not require compilation, offering an easy, user-friendly experience. And a few tools provide the capability for developers to insert new cryptographic algorithm signatures, thereby expanding their detection capabilities as the field advances.

4. Analysis framework for open-source detection tools

Previous tools have been evaluated only with certain basic cryptographic algorithms, and often seem to lack evaluation for false positives, real world applications, and large scale projects. Therefore, as the first step in our replication study, we build a new benchmarking framework with a number of evaluation metrics, which will not only encompass all evaluations conducted by previous tools but also establish a unified more comprehensive evaluation benchmark moving forward.

4.1. Benchmarking Framework

During our study, we identified a significant gap in the field: the absence of a standardized method for evaluating existing tools and their efficacy. We believe that to do a fair evaluation in any domain requires a standard benchmark. We propose a benchmarking framework that will help us understand the scalability and effectiveness of any given detection approach. We have four categories in our framework: i) basic cryptographic functions, ii) microbenchmarks, iii) libraries, and iv) large projects. Our framework is flexible and can be easily updated as the field of cryptography advances.

Basic cryptographic functions. The basic cryptographic functions category contains cryptographic algorithms from various standard classes within the cryptographic space. A single cryptographic algorithm may have multiple versions, or may use non-standard implementations, so we may select many different implementations or versions for certain algorithms. The cryptographic algorithms we select are:

- AES256
- DES
- ECC
- MD5
- RC4
- RC5
- RSA
- SHA1
- SHA256
- TEA
- XTEA
- XXTEA

We have chosen these algorithms for several reasons. First, we have included those recommended by the National Institute of Standards and Technology (NIST) [37], as they are widely utilized and cover a broad range of real-world applications. Second, we have incorporated algorithms like TEA and MD5, which, although once popular, are now considered unsafe or outdated, but still might have historical significance in applications like malware. Last, we have selected various variants of a single cryptographic algorithm to assess a tool's ability to handle different implementations effectively.

Microbenchmarks. Our microbenchmarks category contains small programs on file manipulation, networking, I/O heavy programs, math heavy programs, matrix and array, and so-called "golden implementations" of well-known cryptographic algorithms. We have chosen small programs such as I/O heavy and math-heavy programs since they have similar instruction sets or behavior patterns as cryptographic functions. This allows us to test for false positives in a detection framework.

Libraries. We have chosen cryptographic libraries as well as encoding and compression libraries that have similar behavior to cryptographic functions including:

- openssl (3.3.1)
- libgcrypt (1.8.11)
- libsodium (1.0.20)
- mbedTLS (3.6.1)
- gnuTLS (3.8.6)
- bzip2 (1.0.8)
- zlib (1.3.1)
- ffmpeg (7.0.2)
- libgsm (1.0.17)
- libjpeg (6b)
- libpng (1.6.43)

This allows us to again test for false positives, but also to test for a variety of different cryptographic algorithms implemented in different ways. We select open source libraries only, so that we are able to compile them based on our evaluation metrics. For each tool, we used the latest version available at the time of evaluation. Our selection includes a diverse range of libraries, from those with frequent recent updates, such as OpenSSL, to those that are a bit older, such as libjpeg.

Large projects. We also select Signal, an application designed for encrypted messaging, to evaluate a tool's ability to both scale for large codebases as well as still detect cryptography within them. We believe that to understand the scalability of a mechanism, it is crucial to determine that the tool performs well irrespective of the code size and its applications. Signal was selected because it both utilizes multiple (modern) cryptographic schemes and is open source. This again means we can recompile Signal with our evaluation metrics, plus establish ground truth for testing. However, as our framework is open-source, one could easily extend it to include any future desired large-scale open-source projects, such as Tor.

4.2. Evaluation Metrics

To ensure a thorough exploration of current tools (and to provide rigorous and fair evaluation metrics for the future), we have meticulously crafted a series of evaluation experiments. These experiments are thoughtfully designed to align with the challenges we discuss in Section 7.6, maximizing our ability to uncover the current state of these tools and elucidate their strengths and weaknesses. Our evaluation metrics are as follows:

- **Based on optimization levels:** Evaluate based on different compiler optimizations including -O0, -O1, -O2, -O3, -Os, and -Ofast
- **Based on different combinations of obfuscation mechanisms:**
  - **Control-flow obfuscation:** Adding bogus control-flow, control-flow flattening, substitution of instructions, polymorphism
  - **Data-flow obfuscation:** Data aggregation, data-splitting, variable transformation
  - **Modified versions of algorithms:** For example, modified TEA (Russian TEA) used in malware as suggested in [18]
  - **Layout Obfuscation:** Address obfuscation, obfuscating debug information, address layout/memory layout randomization
  - **Combination of all prior techniques**
- **Based on different compilers:** Evaluate based on different compilers, including GCC, CLANG, MSVC

The full list of optimizations, obfuscations, and their combinations we select for our evaluation can be found in Appendix A.

We assessed each tool with these benchmarks and metrics to thoroughly explore accuracy and detection capabilities when handling binary programs subjected to specific compiler or optimization/obfuscation techniques.[1] For instance, a tool **A** might successfully detect MD5 when all optimization flags are applied but fail to do so in the presence of obfuscation. Conversely, a tool **B** may be able to detect TEA when compiled with GCC but not when compiled with CLANG. This exploration provides us with a more comprehensive understanding of the current state of cryptographic function detection tools, revealing their weaknesses and indicating areas for future development from various perspectives.

5. Reproduction

We start by assessing the reproducibility of each tool by employing consistent experimental setups, procedures, and operating conditions to replicate their purported capabilities.

In our reproducibility evaluation we follow the ACM guidelines on reproducibility [38], using the same measurement procedure, system setting, and operating conditions as the tool's original test cases[2]. This will help us to understand the basic fundamental reliability for each tool. When tools do not pass the reproduction evaluation, we also provide a brief discussion for why. Table 2 shows the reproducibility results.

Aligot	Crypto Hunt	Crypto Knight	DRACA	FindCrypt2	FALKE-MC
✓	✗	✓	–	–	✗

HCD	Kerckhoff	PEiD KANAL	SignSrch	Softmax Classifier	Where's Crypto
–	✗	–	–	✗	✓

Table 2. REPRODUCIBILITY OUTCOME FOR EACH TOOLS. ✓INDICATES A TOOL THAT HAS PASSED THE REPRODUCTION EVALUATION, WHILE ✗REPRESENTS A TOOL THAT HAS EITHER FAILED THE REPRODUCTION EVALUATION OR IS NOT ACCESSIBLE. –DENOTES A TOOL THAT IS AVAILABLE FOR USE, BUT THE DEVELOPER HAS NOT PROVIDED THE ORIGINAL TEST CASES.

Note that for certain tools, the developers only provided the executable binary program without the accompanying source code or test cases necessary for evaluation, making a reproducibility assessment unfeasible (denoted –). Some tools are not publicly available in either source code or executable binary form, making a reproducibility evaluation impossible. As a result, they are marked with a ✗. Additionally, we encountered crashes when running Pin-based tools with the latest version of Intel Pin, indicating that the design of older cryptographic detection tools may not

1. All benchmarks and metrics we evaluate are available at https://github.com/BARC-Purdue/CryptoBinary.

2. For the precise details and steps of our reproducibility evaluation see Appendix B.1.

be compatible with newer Pin versions. However, Aligot is the one exception to this. Aligot provides step-by-step test cases, and, based on this, we have successfully reproduced each step with the expected results. Therefore, we consider Aligot reproducible, as we were able to replicate the test results using their original measurement procedures.

5.1. Non-Reproducible Tools

CryptoHunt has been successfully executed, but we are unable to provide the complete benchmarking framework and performance evaluation results. CryptoHunt detects cryptographic functions in binary code through a bit-precise symbolic loop mapping approach. However, depending on the binary's size, CryptoHunt may identify up to 10,000 loops within the binary. Without additional filtering, the process of comparing each of these loops with reference loops becomes exceedingly time-consuming and infeasible.

Kerckhoff cannot be evaluated in this paper due to its incompatibility with existing available versions of the Pin tracing tool, which cause it to crash. We have encountered a similar issue in the replication evaluation as well, which will be discussed in Section 6.

Softmax Classifier and **FALKE-MC** cannot be evaluated in this paper since they are not available for public use, as neither the source code nor the compiled binary is provided. Only the original research paper is available, with no accompanying software or executable.

6. Replication

We now undertake a comprehensive replicability evaluation of existing work across different categories of cryptographic algorithms with our newly proposed analysis framework, in order to understand their consistency, robustness, and effectiveness. Our replication evaluation follows the ACM guidelines on replicability [38]. We rebuild each tool (if the source code is available) and run it across different system environments using our newly developed evaluation benchmark as the test cases[3]. This evaluation encompasses all work across this domain, allowing us to delve into their performance and potential limitations. For each cryptographic algorithm, we considered different optimization/obfuscation levels, compilers, and implementations. These experiments are designed to help us understand the consistency and robustness of existing work and techniques, the current status of cryptographic function detection performance and to explore potential future research questions, as well as to highlight existing challenges in cryptographic function detection. We also involve micro-benchmarks, libraries, and large scale projects to examine each tool's performance in real-world applications. We present the full results of our evaluation in Table 9 in Appendix D, and discuss a selection of the results, as well as their implications, now.

3. For the precise details and steps of our replicability evaluation see Appendix B.2.

Tool	Cryptographic Algorithm								
	AES	DES	MD5	RC4	RC5	RSA	SHA1	SHA256	TEA
DRACA	✗	✗	✓		✓		✓		✓
CryptoKnight	✓		✗	✗		✓			
FindCrypt2	✗	✓	✓		✓		✓	✓	✓
HCD	–	–	–	–	–	–	–	–	–
PEiD KANAL	–	–	–	–	–	–	–	–	–
SignSrch	✓	✗	✓		✓		✓	✓	✓
Where's Crypto	✓	✓	✓				✓	✓	✓

Table 3. COMPARATIVE ANALYSIS OF DETECTION APPROACHES VERSUS STATED CAPABILITIES IN ORIGINAL DOCUMENTATION. CROSS MARK DENOTES INCONSISTENCIES (FALSE POSITIVE OR FALSE NEGATIVE), CHECK MARK SUCCESSFUL REPLICATION.

Based on our replicability evaluation, we find that some tools cannot be replicated with our framework. We also highlight some interesting new results, which may help direct future research in this field. This will be addressed in both this section and in Section 7.

6.1. Non-Replicable Tools

Aligot is a cryptographic function identification tool developed based on Pin, a dynamic binary instrumentation framework developed by Intel. We successfully reproduce the test cases that Aligot provided. However, Aligot was tested with Pin 2.10 and Pin 2.12 by its developer, but unfortunately, these versions are no longer available. Consequently, we attempted to use an older version of Pin (3.22) but we were unable to execute Aligot correctly. We believe this to be due to the fact that this tool does not support and cannot be executed on recent CPU architectures.

6.2. Unexpected Detection Results

We begin by discussing evaluation results that were either unexpected or inconsistent based on prior stated findings. During our evaluation, we encountered instances where certain tools yielded false positives or false negatives in their detection outcomes. These findings underline the importance of a more in-depth and rigorous analysis to ensure the attainment of dependable detection results. As a result, it is evident that further development is required to enhance reliability.

Unreplicable cryptographic function detection. In Table 8, we list the purported performance for each tool. However, our evaluation could not verify the detection capabilities for certain cryptographic functions in some of the tools. Table 3 presents the outcomes of our assessment relative to the detection capabilities as claimed by the respective tools' original documentation. If a tool claims to have the ability to detect a cryptographic algorithm, but we cannot reproduce the same result in any of our benchmarks or evaluation metrics, we mark it with a red exclamation mark. Otherwise, if we can confirm the detection ability, we mark it with a green circle. A blank means the tool makes no claim about the given algorithm. Due to the unavailability of documentation for HCD and PEiD KANAL, a comparison of our evaluation results with their claimed detection capabilities was not possible.

Micro-benchmark false positives. In our micro-benchmark evaluations, we also uncovered new false positive results. None of the tools falsely considered operations like file handling, I/O, network connection, or heavy math operations as cryptographic functions. However, some tools, such as CryptoKnight and Signsrch, mistakenly identify a mathematical matrix as a cryptographic function like AES. Since AES includes the Rijndael S-box (often coded as a matrix), tools that rely on static signature searches may incorrectly recognize a matrix as an AES function. This highlights the importance of broader and more standardized test cases as well, as things like this were not considered by previous evaluations. To better understand these results, in Table 7 (Appendix D) we present the detailed breakdown of percentage of false positives and false negatives observed for each tool across various cryptographic functions, offering a comprehensive comparison of their performance in these error metrics. This allows us to gain insight into the likelihood of a tool raising errors when attempting to detect specific cryptographic functions.

Undetected cryptographic functions in libraries. Finally, besides standard stand-alone cryptographic functions, we also selected some real-world cryptographic libraries and libraries with similar behavior to cryptographic functions to test on. We find that many tools successfully detect cryptographic functions in these libraries, which is expected, as some, including openssl and libgcrypt, are indeed primarily cryptographic libraries. However, it is interesting to note that this is not true of all tools. Some work fails to detect cryptographic functions in this setting. For example, tools like DRACA, HCD, and Signsrch are unable to detect any cryptographic functions from openssl, while CryptoKnight and HCD fail to detect cryptographic functions in libgcrypt. We are not sure why this is the case, and leave this as an interesting avenue for future work. This also points to the fact that existing work may have unreliable performance in real world (large) programs.

Future Research: Certain cryptographic-specific libraries seem to pose challenges for existing tools; exploring why this is the case would be an interesting avenue for future work. Also, existing tools have more false positives/negatives than initially expected, so placing more emphasis on improving detection accuracy and reducing false positives or false negatives seems highly relevant.

6.3. Operating System Limitations

The usage of certain tools may be constrained by their exclusive compatibility with specific operating systems, impacting their efficacy and analysis capabilities. Many tools are designed and built with specific operating systems in mind. Fortunately, with the help of other tools, such as WineHQ, which enables software developed for Microsoft Windows to run on Unix-like operating systems, developers can for example use some Windows cryptographic detection tools in Linux. However, tools are not just limited to running on specific operating systems, but also restricted to handling binaries compiled for selected operating systems. These inherent limitations potentially reduce the scope and depth of analysis a tool can provide, depending on the operating system selected and targeted in the original work.

Tool	Supported System			Supported Binary		
	Windows	macOS	Linux	Windows	macOS	Linux
Aligot	●	●	●	●	●	●
CryptoHunt	○	○	●	○	○	●
CryptoKnight	●	●	●	●	●	●
DRACA	●	◐	◐	●	●	●
FindCrypt2	●	●	●	●	●	●
HCD	●	◐	◐	●	○	○
Kerckhoff	●	○	○	●	○	○
PEiD KANAL	●	◐	◐	●	●	●
Signsrch	●	◐	◐	●	●	●
Where's Crypto	●	○	○	●	●	●

Table 4. TOOL COMPATIBILITY WITH DIFFERENT OPERATING SYSTEMS; ●REPRESENTS A TOOL THAT SUPPORTS THE DESIRED SYSTEM, WHILE ○INDICATES A TOOL THAT DOES NOT SUPPORT THE DESIRED SYSTEM; ◐ REPRESENTS CASES WHERE A TOOL CANNOT BE EXECUTED ON A UNIX-LIKE SYSTEM BUT BECOMES FUNCTIONAL WITH THE ASSISTANCE OF WINEHQ

In Table 4, we evaluate the capability of exisiting work to analyze binaries across three different operating systems and for binaries compiled specifically for each of these operating systems. We use a ◐ to indicate cases where a tool cannot be executed on a Unix-like system but becomes functional with the assistance of WineHQ.

Takeaway: Ideally cryptographic function detection tools need to support all operating systems to effectively detect cryptographic functions or malware across diverse environments and improve overall detection measures.

6.4. Impact of Compiler Variations

To study the impact of compilers on cryptographic function detection, we compiled each program in our benchmarking framework with GCC/G++, CLANG/CLANG++, and MSVC. Somewhat as expected given the impact a compiler can have on the resulting binary, our evaluation shows that tools may output quite different results for different compilers. We choose MD5 as an illustrative example, shown in Figure 4, though we have observed the same general trends across most cryptographic schemes.

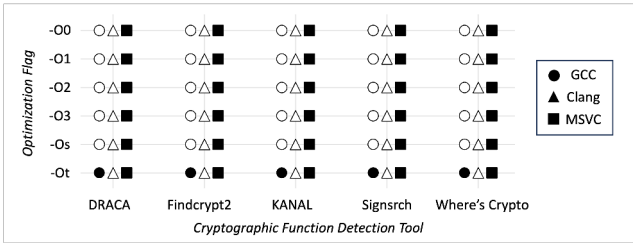

Figure 4. Evaluation result for MD5 with different compilers. A full marker indicates that function can be detected and an empty marker that it cannot.

In this example, DRACA, Findcrypt2, PEiD KANAL, Signsrch, and Where's Crypto support detecting the MD5 algorithm. However, their detection capabilities are influenced by the compiler used. All of these tools can detect

the MD5 algorithm in binaries compiled with the MSVC compiler, regardless of the optimization flag. In contrast, for binaries compiled with GCC, they can only detect the MD5 algorithm when the -O0 optimization flag is applied. Furthermore, when using the Clang compiler, none of these tools were able to detect the MD5 algorithm in the binary. This suggests that differences between compilers may significantly impact a tool's analysis results. In general, we found that the MSVC compiler has less impact on detection results compared to GCC and CLANG. Specifically, if a tool can detect a certain cryptographic algorithm in a program compiled by GCC or CLANG, it will also be able to detect that algorithm in the same program compiled by MSVC.

For example, Signsrch shows inconsistent performance with Clang, GCC, and MSVC, identifying the SHA-1 hash function in certain programs compiled with identical optimization flags but using a different compiler. We also observe same behaviour for DARACA when detect XXTEA algorithm (see Table 9 for details).

The structure of binaries produced by different compilers, such as MSVC, GCC, and LLVM, can vary significantly due to differences in object file formats, optimization strategies, and code generation techniques. Binary analysis often relies on symbolic information, padding, and inlined data, all of which are heavily influenced by the compiler version [39]. These variations can hinder the effectiveness of cryptographic function detection tools, potentially leading to inaccurate or incomplete detection results.

Future Research: At present, it is notable that none of the cryptographic function detection tools consider the impact of the compiler. Our evaluation results have revealed that the choice of compiler plays a pivotal role in cryptographic function analysis. The exploration of compiler effects and their incorporation into cryptographic function detection tools represents an enticing avenue for future research. Moreover, understanding and accommodating these compiler-related influences could significantly enhance both accuracy and reliability.

6.5. Impact of Optimization Flags

Optimization flags and obfuscation can significantly impact the analysis results for cryptographic functions [40]. CryptoHunt[18] and FALKE-MC [26] have explored this direction. CryptoHunt primarily focuses on specific optimization flags such as `-O0`, `-O1`, and `-O2`, while FALKE-MC includes programs compiled with `-Os` and `-Ofast` for training. However, we are unable to run either of these tools, so we cannot confirm their detection performance with optimization/obfuscations flags. In order to comprehensively investigate this topic, we have expanded our study to include additional optimization flags, obfuscation techniques, and their combinations, thereby providing a more thorough exploration of their impact on cryptographic algorithm detection.

Our findings indicate that both optimization flags and obfuscations can affect detection capability. For instance, based on our results in Table 9, we can see that Findcrypt2, PEiD KANAL, Signsrch, and Where's Crypto can detect MD5 when compiled without optimization/obfuscation flags or compiled with `-O1`. However, none of these tools can detect MD5 if the binary program is compiled with advanced optimization/obfuscation flags such as `-O2`, `-Ofast`, or `-obs_sub`. In addition, we observed that Signsrch exhibits inconsistent performance for SHA-1 or SHA-256, and DRACA shows unstable performance with XTEA and XXTEA when running with optimization or obfuscation flags. Optimization flags significantly alter the structure and behavior of a binary program by improving performance, reducing size, and modifying control flow. Techniques like function inlining, loop unrolling, and dead code elimination obscure the original source code structure, making it more difficult for analysts to identify cryptographic functions.

Future Research: The effects of optimization and obfuscation have been noted in works such as CryptoHunt, which is designed specifically for obfuscated binaries. However, limitations such as slow performance due to the large trace size, may reduce its applicability for large scale projects. Considering that many tools seem to face challenges with obfuscated binaries, we assert that investigating the effects of optimization and obfuscation continues to be a compelling and critical avenue for future research.

6.6. Impact of Different Algorithm Versions

We also carried out evaluations to determine whether variations in a cryptographic algorithm affects the success of its detection. As a case study, we analyzed different variants of the Tiny Encryption Algorithm (TEA), including standard TEA, XTEA, and XXTEA. While all the tools were able to successfully detect TEA binaries with any of these three variants, we observed some interesting details in the detection results.

Algorithm	DRACA	Signsrch
TEA	* RC5 or RC6 - 50% * TEA - 67%	* TEA1_DS [32.le.4] * TEA encryption/decryption
XTEA	* RC5 or RC6 - 50% * TEA - 34%	* TEA1_DS [32.le.4]
XXTEA	* RC5 or RC6 - 50% * TEA - 67%	* TEA1_DS [32.le.4]

Table 5. OUTPUT WHEN EVALUATING DRACA AND SIGNSRCH WITH DIFFERENT VARIANTS OF TEA

For example, as shown in Table 5, when using DRACA to analyze binaries containing TEA or XXTEA, the tool provides a confidence score. It tends to determine that a binary program is more likely to contain TEA functions if TEA or XXTEA is present. However, when analyzing binary programs containing XTEA, DRACA may incorrectly identify RC5 or RC6 as being present in 50% of cases, and TEA functions in only 34% of cases. In contrast, Signsrch is able to detect TEA functions in all three binary programs containing TEA, XTEA, or XXTEA, but it may identify different TEA signatures in each binary.

Future Research: Our evaluations indicates that the variant or version of a cryptographic algorithm plays a crucial role in its detection, suggesting that existing tools might struggle to accurately identify certain variants based on their signatures alone. Currently, there is no approach that focuses on this difference, which could be an interesting future research path.

6.7. Impact of Intentional Evasion Tactics

We selected multiple (ransomware) malware binaries, including CryptoLocker [41], CryptoWall [42], Locky [43], TeslaCrypt [44], and Win32Dircrypt [45], to evaluate if cryptographic detection tools are better or worse at finding cryptographic algorithms in malware binaries (where authors might intentionally try to obfuscate the presence of cryptography) versus normal applications.

We tested the tools with these malware samples and present the results in Table 6. It is worth noting that malware may have different versions depending on the discovery date, and detection results are highly dependent on these versions. We found that the tools could detect cryptographic functions in TeslaCrypt version 2 but failed to do so in versions 1 and 3. This suggests that the malware may employ non-standard algorithms to evade binary analysis in certain versions, increasing the difficulty of detection. Furthermore, cryptographic functions in CryptoLocker's September 10, 2013 version were detectable by the tools, but in the November 20, 2013, and January 22, 2014 versions, they were no longer detectable. In the latter two versions, some tools identified the presence of anti-debugging techniques in the malware, which might explain why cryptographic functions could not be detected anymore.

Malware	DRACA	PEiD KANAL	Signsrch	Note
CryptoLocker_10Sep2013	✗	✓	✓	
CryptoLocker_20Nov2013	✗	✗	✗	anti-debug
CryptoLocker_22Jan2014	✗	✗	✗	anti-debug
CryptoWall	✗	✗	✗	anti-debug
Locky	✗	✗	✗	
TeslaCrypt_v1	✗	✗	✗	
TeslaCrypt_v2	✓	✓	✓	
TeslaCrypt_v3	✗	✗	✗	
Win32Dircrypt	✗	✓	✓	

Table 6. EVALUATION RESULTS FOR CRYPTOGRAPHIC FUNCTION DETECTION IN DIFFERENT MALWARE VERSIONS

Future Research: Cryptographic function detection can be a useful component in identifying malware, and according to our investigation, current tools do not consistently detect cryptographic functions within the malware samples. This challenge has been noted in other research as well, for example, CryptoHunt identified environment-sensitive malware as a potential source of false negatives in detection efforts. Therefore, cryptographic function detection primarily focused on targeting malware could be a highly relevant source of future research in this area.

7. Discussion

We now provide a discussion backed by our evaluation results and examination of existing work and implemented tools, delving into broader issues that we discovered. This includes discussing additional overarching future research directions and key areas where improvements can be made moving forward.

7.1. Approaches to Algorithm Detection in Tools

We start by discussing the detection results of a single tool for different algorithms. We opted to evaluate 12 distinct algorithms (AES, DES, ECC, MD5, RC4, RC5, RSA, SHA-1, SHA-256, TEA, XTEA, and XXTEA) along with our microbenchmarks, various libraries, and a large-scale project.

Based on our evaluation (see Table 9), we found that PEiD KANAL, SignSrch, and Where's Crypto can detect most algorithms we selected. This was somewhat expected as Where's Crypto is a more modern detection tool. HCD demonstrated average performance in our evaluation results. Nonetheless, we came across cases where tools did not detect cryptographic functions they were expected to identify, indicating potential need to enhance detection accuracy.

Future Research: Current tools seemingly employ various (often distinct) approaches compared to one another. Many of these tools also provide developers with the opportunity to expand and enhance their detection capabilities. Therefore, future research could focus on improving existing technologies within individual tools and existing approaches to achieve broader applicability with higher detection rates, and to make it so that users do not need to use different tools to best identify different algorithms.

7.2. AI/ML Approach

Artificial intelligence and machine learning (AI/ML) have been extensively applied across various domains to address complex problems. Previous works, like CryptoKnight, have utilized machine learning techniques for cryptographic function detection. However, in our evaluation, we found that these tools did not exhibit satisfactory performance. We attribute this issue not to the approach itself, but rather to the fact that these tools typically only offer a demonstration to showcase the potential application of AI/ML in this field. The main problem lies in the lack of sufficient signatures or analysis provided by these tools, which is likely the primary reason for their low performance.

Future Research: While current AI/ML based tools demonstrate the feasibility of utilizing AI/ML in cryptographic function detection, they fall short in providing the robust and comprehensive signature or analysis capabilities necessary for accurate detection. These seem like promising future research directions.

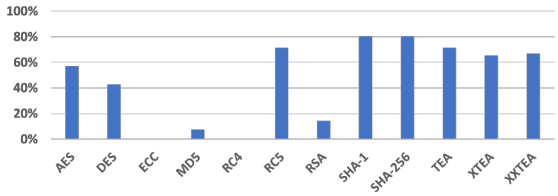

Figure 5. Detection percentage by algorithm across all tools

7.3. Lack of Support for Modern Cryptographic Algorithms

Many modern cryptographic algorithms, such as general Elliptic-Curve Cryptography (ECC) and more specifically ECDSA and ECDHE, have found widespread use in diverse applications [46] due to their advantages in terms of smaller private key size and improved efficiency at equivalent or higher security levels. ECDSA is utilized in a number of areas, including electronic healthcare, banking, commerce, and vehicles [47], and is currently supported by most major libraries, including cryptlib, Crypto++, libgcrypt, LibreSSL, and OpenSSL. However, our evaluation reveals that existing work lacks the ability to find such modern algorithms. As such, this could be a rich area for future development and improvement.

Future Research: As the prevalence of applications adopting ECC continues to rise, the ongoing development of cryptographic function detection has led to a disconnect between the outcomes of these efforts and the real-world applications that people are eager to test, leading to an important direction for future research.

7.4. Comparative Detection of Similar Algorithms Across Various Tools

Our evaluation helps provides insights from an algorithmic perspective on how well different tools can detect the same cryptographic algorithm. In our evaluation, we test the tools' abilities to find cryptographic functions in different settings (optimization flags, compilers, etc.). Figure 5 shows the percentage of algorithms that can be successfully detected by a tool across all evaluation experiments that we ran. We can see that TEA, RC5, and SHA can be successfully detected in most experiments. However, AES, and MD5 can only be found in a few experiments, with algorithms like RC4 and ECC not even being detected at all.

From our observations, it is not evident that there exists a consistent rule governing the ease of detecting specific categories of cryptographic functions. For instance, when considering two hash functions, MD5 and SHA, we note markedly different results. Even within the same family we find variations; TEA and RC5, both belonging to the block cipher family, exhibit detectability, while DES, another block cipher, proves more challenging to detect.

Future Research: Some cryptographic algorithms appear to be more challenging to detect than others. However, we have not observed a distinct pattern suggesting that algorithms within a specific category consistently present greater difficulty in detection. Therefore, future research could focus on how to detect cryptographic algorithms that appear to be harder to identify or investigate the underlying factors that contribute to variations in the detectability of algorithms within the same class.

7.5. User Experience

Considering the experience and insights we have garnered from our evaluation process, we believe that discussing the developer experience is also an intriguing topic, in terms of how developers interact with these tools and their ease of use.

Older cryptographic detection tools, such as DRACA, PEiD KANAL, and SignSrch, do not require developers to compile and build the tools from source code. Developers can readily utilize these tools with supported operating systems. However, when compared to some better performing tools, these user-friendly alternatives exhibit lower performance and utility. For example, obfuscation can disturb the binary analysis process of Signsrch, leading to false negatives in the detection of certain cryptographic algorithms. Furthermore, DRACA struggles to accurately analyze packed executable programs, and consequently, it can only provide a rudimentary analysis outcome, despite its ease of use.

In addition to the aforementioned challenges, some tools pose difficulties for users due to technological complexity or data-related issues. Take, for instance, IDA Scope, Findcrypt2, and Where's Crypto, which all rely on IDA Pro. IDA Pro is not a free, open-source software, and these tools cannot be utilized by developers without it. Conversely, tools such as CryptoKnight and other machine learning-based techniques require users to provide substantial amounts of data, which further compounds the difficulty of using these tools.

However, these tools generally exhibit superior performance when compared to user-friendly alternatives. For instance, CryptoHunt consistently yields accurate analysis results for identification within obfuscated programs. Where's Crypto, one of the most recent cryptographic detection tools, broadens the scope of analysis to encompass unfamiliar and proprietary cryptographic primitives, without relying on heuristics to select code fragments.

7.6. Challenges for Future Research

Based on our evaluation, we have identified several challenges that make cryptographic function identification difficult. In this subsection, we discuss some of these challenges, as we believe they may help inform future tool development and areas of research.

Obfuscation. One of the major challenges in cryptographic function identification is obfuscation. The initial

purpose of obfuscation was to protect software intellectual property [48] from malicious reverse engineering attempts. However, malware authors adopted obfuscation techniques as a way to avoid detection. Various types of obfuscation techniques have been implemented to thwart any form of detection. These obfuscation techniques can be primarily categorized as: control-flow obfuscation, data obfuscation, and layout obfuscation. For a variety of reasons, one might still wish to identify cryptographic code even in the face of such obfuscation techniques, particularly given its prevalence nowadays.

Control-flow obfuscation is the most common type of obfuscation. A control-flow graph (CFG) [49] embodies the graphical representation of the flow of a program. To hinder CFG-based detection, malware authors introduce techniques such as including bogus control-flow [50], control-flow flattening [33], and opaque predicates [51]. Similar to control flow obfuscation, data obfuscation techniques, like data aggregation and data splitting, attempt to hinder approaches based on input/output relationships. For example, one might split a single variable into two to prevent detection. Layout obfuscation [52] scrambles the layout of a program's instructions while keeping the original syntax intact. Layouts can be obfuscated by adding meaningless classifiers, stripping redundant symbols, separating related code, as well as adding junk code.

Implementation Variation. Despite having a well-defined specification, cryptographic algorithms can be implemented in a number of ways while still achieving the same result. In addition, cryptography is not always implemented correctly or exactly to spec. For example, buggy implementations of the TEA algorithm were found [17] in malware such as Storm Worm and Silent Banker. Hence, even if a detection approach can find an ideal implementation of an algorithm, there is no guarantee it can also detect all the implementation variations of the same algorithm. Our evaluations in this space are discussed in Sections 6.6 and 6.7.

Differences in Cryptographic Functions. There are fundamental differences in different classes of cryptographic algorithms. Cryptographic features (see Section 2.2) present in one algorithm may not be present in others. For instance, the data avalanche effect [19] which says that an insignificant change in the input parameters can make significant differences in the output, works well for identifying block ciphers. In the case of stream ciphers, there are no such observations like data avalanching. Tools must be designed to account for these differences if they wish to identify broad classes of algorithms.

Compilers. The choice of compiler can significantly impact the binary structure of a program in various ways such as format, linking, inline functions, and compatibility with different architectures. The incorporation of anti-analysis techniques further adds to the complexity of cryptographic function detection. Consequently, working with different compilers can introduce numerous challenges. Our study (Section 6.4) demonstrates how a tool's detection capabilities can vary greatly depending on the compiler used.

Comparison to General Binary Analysis. Binary code similarity analysis is an active area of research in the software community. There has been a plethora of research [53], [54], [55], [56], [57], [58], [59], [60] to compare two or more binaries to identify their similarities or variances. While such generic binary analysis techniques can be leveraged in the detection of cryptographic functions, future researchers should bear in mind that detecting cryptographic functions in binary code remains a multifaceted challenge. Cryptographic algorithms exhibit significant variability in terms of complexity and implementation. They often involve intricate mathematical operations and employ various techniques such as bit manipulation, modular arithmetic, and logical operations. This complexity can pose difficulties in identifying cryptographic functions without specific knowledge of the algorithm being utilized. In addition, they can be implemented in diverse ways and may not adhere to consistent naming or calling conventions, rendering them more challenging to identify through static analysis alone.

8. Conclusion

Cryptographic function identification has been a popular area of study in both academia and industry. However, we noticed a lack of consistent means for evaluation and comparison among tools. As such, in this work we conduct comprehensive reproduction and replication studies on all existing work in the space. To complement the traditional R+R studies, we developed a comprehensive testing and evaluation framework which includes a number of modern cryptographic algorithms and real world examples, that allow for the comparison of both existing and future work in a uniform manner. Finally, we highlight major gaps in existing work, especially as they relate to modern cryptographic primitives and real-world use cases, and discuss a variety of important avenues for future work.

Acknowledgment

Yongming Fan and Christina Garman's work was partially supported by NSF grant CNS-2047991.

References

[1] "Openssl," https://www.openssl.org/.

[2] "Wannacry ransomware," https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[3] "Petya ransomware," https://www.proofpoint.com/us/glossary/petya.

[4] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE S&P*, 2008.

[5] I. Harvey, "Cipher hunting: How to find cryptographic algorithms in large binaries," *NCipher Corporation Ltd*, 2001.

[6] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, 2008.

[7] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *ACM CCS*, 2007.

[8] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *ESORICS*, 2009.

[9] R. Chang, L. Jiang, H. Shu, and H. He, "Cryptographic algorithms analysis technology research based on functions signature recognition," in *CIS*, 2014.

[10] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated identification of cryptographic primitives in binary code with data flow graph isomorphism," in *ASIACCS*, 2015.

[11] "Draft crypto analyzer," http://www.literatecode.com/draca.

[12] "Kanal - krypto analyzer for peid," http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm.

[13] "Kerckhoffs," https://github.com/felixgr/kerckhoffs.

[14] "Hash & crypto detector (hcd)," https://webscene.ir/tools/show/Hash-and-Crypto-Detector-1.4.

[15] "Signsrch," http://aluigi.altervista.org/mytoolz/signsrch.zip.

[16] "Findcrypt2," http://www.hexblog.com/?p=28.

[17] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *ACM CCS*, 2012.

[18] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *IEEE S&P*, 2017.

[19] X. Li, X. Wang, and W. Chang, "Cipherxray: Exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE TDSC*, 2012.

[20] N. Lutz, "Towards revealing attacker's intent by automatically decrypting network traffic," *Mémoire de maitrise, ETH Zürich, Switzerland*, 2008.

[21] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *RAID*, 2011.

[22] J. Park and Y. Park, "Symmetric-key cryptographic routine detection in anti-reverse engineered binaries using hardware tracing," *Electronics*, 2020.

[23] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security*, 2015.

[24] J. L. Wright and M. Manic, "Neural network approach to locating cryptography in object code," in *ETFA*, 2009.

[25] L. Benedetti, A. Thierry, and J. Francq, "Detection of cryptographic algorithms with grap." *IACR Cryptology ePrint Archive*, 2017.

[26] A. Aigner, "Falke-mc: A neural network based approach to locate cryptographic functions in machine code," in *ARES*, 2018.

[27] G. Hill and X. Bellekens, "Cryptoknight: Generating and modelling compiled cryptographic primitives," *Information*, 2018.

[28] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, "A neural network-based approach for cryptographic function detection in malware," *IEEE Access*, 2020.

[29] S. Lee, N.-s. Jho, D. Chung, Y. Kang, and M. Kim, "Rcryptect: Real-time detection of cryptographic function in the user-space filesystem," *Computers & Security*, 2022.

[30] "Data flow graph," http://bears.ece.ucsb.edu/research-info/DP/dfg.html.

[31] "S-box," https://en.wikipedia.org/wiki/S-box.

[32] "P-box," https://en.wikipedia.org/wiki/Permutation_box.

[33] "Control-flow flattening," https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-Flattening.

[34] C. Meijer, V. Moonsamy, and J. Wetzels, "Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code," in *USENIX Security*, 2021.

[35] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *DIMVA*, 2017.

[36] F. Matenaar, A. Wichmann, F. Leder, and E. Gerhards-Padilla, "Cis: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware," in *MALWARE*, 2012.

[37] "Cryptographic standards and guidelines," https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines.

[38] Association for Computing Machinery, "Artifact Review and Badging Policy," 2024, accessed: 2024-09-16. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[39] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *EuroS&P*, 2017.

[40] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *PLDI*, 2021.

[41] A. Hansberry, A. Lasse, and A. Tarrh, "Cryptolocker: 2013's most malicious malware," *Retrieved February*, 2014.

[42] K. Cabaj, P. Gawkowski, K. Grochowski, and D. Osojca, "Network activity analysis of cryptowall ransomware," *Przeglad Elektrotechniczny*, 2015.

[43] . . p. U. Sean Gallagher Feb 17 and L. S. Palatinae, ""locky" crypto-ransomware rides in on malicious word document macro," Feb 2016. [Online]. Available: https://arstechnica.com/security/2016/02/locky-crypto-ransomware-rides-in-on-malicious-word-document-macro/

[44] Y. Lemmou and E. M. Souidi, "Infection, self-reproduction and over-infection in ransomware: the case of teslacrypt," in *Cyber Security*, 2018.

[45] M. Corporation, "Microsoft." [Online]. Available: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32%2FDircrypt

[46] R. Harkanson and Y. Kim, "Applications of elliptic curve cryptography: A light introduction to elliptic curves and a survey of their applications," in *CISRC*, 2017.

[47] M. Al-Zubaidie, Z. Zhang, and J. Zhang, "Efficient and secure ecdsa algorithm and its applications: A survey," *arXiv preprint arXiv:1902.10313*, 2019.

[48] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[49] "Control flow graph," https://en.wikipedia.org/wiki/Control-flow_graph.

[50] Y. Li, Z. Sha, X. Xiong, and Y. Zhao, "Code obfuscation based on inline split of control flow graph," in *ICAICA*, 2021.

[51] D. Xu, "Opaque predicate: Attack and defense in obfuscated binary code," 2018.

[52] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: a taxonomy of software obfuscation techniques for layered security," *Cybersecurity*, 2020.

[53] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *PLDI*, 2017.

[54] A. Saebjornsen, *Detecting fine-grained similarity in binaries*. University of California, Davis, 2014.

[55] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *ESEC/FSE*, 2018.

[56] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *FSE*, 2016.

[57] D. McKee, N. Burow, and M. Payer, "Software ethology: An accurate, resilient, and cross-architecture binary analysis framework," *arXiv preprint arXiv:1906.02928*, 2019.

[58] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE S&P*, 2019.

[59] J. Ming, D. Xu, Y. Jiang, and D. Wu, "{BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *USENIX Security*, 2017.

[60] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *ACM CCS*, 2017.

[61] "Ida pro," https://hex-rays.com/ida-pro/.

Appendix A. List of Optimizations, Obfuscations, and their Combinations for Our Evaluation Metrics

0) No identification
1) Identification with optimization flag -O0
2) Identification with optimization flag -O1
3) Identification with optimization flag -O2
4) Identification with optimization flag -O3
5) Identification with optimization flag –Os
6) Identification with optimization flag: -Ofast
7) Identification with instruction substitution obfuscation: -obs_sub
8) Identification with control-flow flattening: -obs_fla
9) Identification with bogus control-flow: -obs_bcf
10) Identification with combination of obfuscation: -obs_sub, obs_fla, obs_bcf
11) Identification with combination of obfuscation and optimization: -obs_sub, obs_fla, obs_bcf, -O3

Appendix B. Procedure for R+R Evaluation

B.1. Steps in our Reproduction Evaluation

1) Obtain access to each tool; compile the tool if required, following the provided instructions.
2) Set up the environment to replicate the tool's original configuration (follows the same operating conditions).
3) Obtain the original test cases for each tool. If unavailable, the reproduction evaluation will be skipped (follows the same measuring system).
4) Execute each tool with the provided test cases to reproduce the results (follows the same measurement procedure).

B.2. Steps in our Replication Evaluation

1) Obtain access to each tool; recompile the tool on our system if the source code is available.
2) Build and run each tool on Windows 10, Ubuntu 18.04, and macOS Sonoma (follows the different measuring system).
3) Compile our custom benchmarks with the evaluation metrics into binaries for testing.
4) Execute each tool with our benchmarks to assess replication capability (follows the different location on multiple trials).

Appendix C. Tools

We provide introductions to the tools analyzed in this work. For each tool, we give a high level overview, highlight key features for a user, and provide a discussion on its strengths and weaknesses, as relevant.

C.1. Aligot

Summary. Aligot [17] is based on the assumption that the input-output relations in a cryptographic function remain unchanged even if there is obfuscation. It performs an interloop data-flow analysis to identify cryptographic parameters and then performs a comparison with a reference implementation. The authors successfully evaluated the tool to identify four different cryptographic functions: TEA, RC4, AES and MD5.

Highlights for User.

- Requires the user to compile the tool with Intel Pin
- Requires an earlier version of Intel Pin and Python library *networkx* (latest does not work)

Other Discussion.

- Aligot has not been updated to accommodate the latest dependency changes. Specifically, the Python library "networkx" uses different data types that cause crashes in Aligot
- Additionally, the Aligot tracer has not been adapted to the latest version of Intel Pin, and the tested version appears to be discontinued

C.2. CryptoHunt

Summary. CryptoHunt [18] introduces the *bit-precise symbolic loop mapping* technique to identify cryptographic functions. Using this technique the tool captures the semantics of possible cryptographic algorithms within a loop and then performs guided fuzzing to efficiently match boolean formulas with known reference implementations. It shows its efficacy on commonly used cryptographic functions such as TEA, AES, RC4, MD5, and RSA under different control and data obfuscation scheme combinations.

Highlights for User.

- Requires the user to compile the tool with Intel Pin (2.13, 3.2, and 3.28 are tested)
- Requires the user to indicate the reference loop
- Higher accuracy rate and detection results in obfuscated binaries

Other Discussion.

- Depending on the binary program size, CryptoHunt may find 10,000 loops in the binary, which needs to be compared with the reference one by one. The long analysis time may not suitable for a large program.

C.3. CryptoKnight

Summary. CryptoKnight [27] employs a Dynamic Convolutional Neural Network (DCNN) to learn from variable-length control flow diagnostics output obtained from a dynamic trace, and is able to detect AES, RC4, Blowfish, MD5, and RSA.

Highlights for User.

- Can easily integrate new samples through the scalable synthesis of customizable cryptographic algorithms
- Automated and limits user interaction
- High accuracy rate (96%)

Other Discussion.

- Has been evaluated on GnuPG and a real-world ransomware binary 'GonnaCry' and can successfully identify the use of AES and RSA

C.4. Draft Crypto Analyzer (DRACA)

Summary. DRACA [11] targets preliminary detection and analysis of cryptographic algorithms within executables. Although DRACA is implemented as a command line utility for x86/Win32, it can also analyze Unix ELF binaries, Java applets, as well as 16- and 32-bit DOS Windows executables.

Highlights for User.

- No compiling is needed
- Can be directly run by passing the target binary program

Other Discussion.

- DRACA can only provide a rough idea of what kind of algorithms to look at without actual spending time on decompilation and code analysis.
- Detection result cannot be used without further analysis.

C.5. FindCrypt2

Summary. FindCrypt2 [16] has been implemented as a plug-in support for IDA Pro [61] which searches constants used in the initialization of cryptographic algorithms. It can detect commonly used encryption algorithms.

Highlights for User.

- Requires IDA Pro, a commercial disassembler software, which is not free

C.6. HCD

Summary. HCD [14] detects roughly 90 hash and cryptographic algorithms and compiles for portable executable files with a GUI interface and Shell integration, Command line support.

Highlights for User.

- No compiling is needed
- Can be directly run by passing the target binary program

Other Discussion.

- Can only detect cryptographic functions in executable file for Microsoft Windows (.exe). This tool does not support the analysis of Unix based binary programs.

C.7. PEiD KANAL

Summary. PEiD KANAL [12] is a plug-in for PEiD, searching for cryptographic algorithms with a specific signature such as fixed s-boxes, permutation tables, and initialization values. It searches for connections to identified code or data and find its associated address.

Highlights for User.

- No compiling is needed
- Can be directly run by passing the target binary program

C.8. Signsrch

Summary. Signsrch [15] is another IDA Pro plug-in which aims to detect not just encryption algorithms, but also compression algorithms, multimedia, etc. The tool searches for known signatures as well as allows a user to add their own signature.

Highlights for User.

- No compiling is needed
- Can be directly run by passing the target binary program

C.9. Where's Crypto

Summary. Where's Crypto [34] is an IDA SDK based tool, aiming to automatically identify cryptographic primitives in binaries. Where's Crypto can detect cryptographic functions that has as-of-yet unknown primitives if it falls within a taxonomical class of well-defined primitives.

Highlights for User.

- Requires IDA Pro, a commercial disassembler software, which is not free
- Supports unknown cryptographic primitives

Appendix D. Additional Tables

This appendix contains the complete results of all of our replication and reproduction studies.

Tool		Cryptographic Algorithm									Micro-Benchmark				
		AES	DES	MD5	RC4	RC5	RSA	SHA1	SHA256	TEA	File	I/O	Math	Matrix	Network
DRACA	False Positive	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	False Negative	100%	100%	69%	0%	0%	0%	0%	0%	28%	0%	0%	0%	0%	0%
	True Positive	0%	0%	31%	0%	100%	0%	100%	0%	72%	0%	0%	0%	0%	0%
	True Negative	0%	0%	0%	100%	0%	100%	0%	100%	0%	100%	100%	100%	100%	100%
CryptoKnight	False Positive	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	74%	0%
	False Negative	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	True Positive	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%
	True Negative	0%	100%	0%	0%	100%	0%	100%	100%	100%	100%	100%	100%	26%	100%
Findcrypt2	False Positive	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	False Negative	100%	0%	69%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	True Positive	0%	100%	31%	0%	100%	0%	100%	100%	0%	0%	0%	0%	0%	0%
	True Negative	0%	0%	0%	100%	0%	100%	0%	0%	100%	100%	100%	100%	100%	100%
Signsrch	False Positive	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%
	False Negative	0%	100%	69%	0%	0%	0%	31%	0%	0%	0%	0%	0%	0%	0%
	True Positive	100%	0%	31%	0%	100%	0%	69%	100%	100%	0%	0%	0%	0%	0%
	True Negative	0%	0%	0%	100%	0%	100%	0%	0%	0%	100%	100%	100%	0%	100%
Where's Crypto	False Positive	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	False Negative	0%	0%	69%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	True Positive	100%	100%	31%	0%	0%	0%	100%	100%	100%	0%	0%	0%	0%	0%
	True Negative	0%	0%	0%	100%	100%	100%	0%	0%	0%	100%	100%	100%	100%	100%

Table 7. PERFORMANCE METRICS FOR TOOL EVALUATION: FALSE POSITIVES, FALSE NEGATIVES, TRUE POSITIVES, AND TRUE NEGATIVES

Tool	Aligot	Crypto-Hunt	Crypto-Knight	DRACA	FindCrypt2	FALKE-MC	HCD	Kerckhoff	PEiD KANAL	SignSrch	Softmax Classifier	Where's Crypto
ADLER32							*Not Available*		*Not Available*		✓	
AES (Rijndael)	✓	✓	✓	✓	✓	✓		✓		✓		✓
BASE64										✓	✓	
Blowfish			✓	✓	✓					✓	✓	
Camellia					✓							
CAST					✓					✓		
CAST-256				✓	✓					✓		
CRC32				✓	✓					✓	✓	
DES				✓	✓			✓		✓	✓	✓
EC										✓		
GOST					✓					✓		
HAVAL					✓					✓	✓	
MARS				✓	✓					✓		
MD2					✓					✓		
MD4					✓					✓		
MD5	✓	✓	✓	✓	✓			✓		✓	✓	✓
RC2				✓	✓					✓		
RC4	✓	✓	✓			✓		✓				
RC5				✓	✓					✓	✓	
RC6				✓	✓					✓	✓	
Ripemd-160				✓						✓		
RSA	✓	✓	✓					✓				
SAFER				✓	✓					✓		
SHA-1				✓	✓					✓	✓	✓
SHA-256					✓					✓	✓	✓
SHA-512					✓					✓		✓
SHARK					✓					✓		
Skipjack				✓	✓					✓		
Square					✓					✓		
TEA	✓	✓		✓						✓		
Tiger				✓	✓					✓		
Twofish				✓	✓					✓		
WAKE					✓					✓		
Whirlpool					✓					✓		
XTEA												✓

Table 8. CRYPTOGRAPHIC ALGORITHM DETECTION SUPPORT AS STATED IN THE PAPER OR DOCUMENTATION FOR EACH TOOL

Table 9. DETAILED EVALUATION RESULTS FOR EACH TOOL BASED ON OUR UNIFORM BENCHMARKING SUITE

Tool	Flag	Basic Cryptographic Function												Micro-Benchmark					Library											Large Project
		AES	DES	ECC	MD5	RC4	RC5	RSA	SHA1	SHA256	TEA	XTEA	XXTEA	File	I/O	Math	Matrix	Network	openssl	libgcrypt	libsodium	mbedTLS	gnuTLS	bzip2	zlib	ffmpeg	libgsm	libjpeg	libpng	signal
DRACA	-O0	○△□	○△□	○△□	●△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	○△□	●△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	-O1	○△□	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	-O2	○△□	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	-O3	○△□	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	-Os	○△□	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	●▲□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	-Ofast	○△□	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	○△□	●▲■	●▲□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲□
	obs_sub	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_fla	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_bcf	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf, -O3	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
CryptoKnight	-O0	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	-O1	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	-O2	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	-O3	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	-Os	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	-Ofast	●▲□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	●▲□	○△□	○△□	○△□	○△□	○△□	○△□	●▲□	○△□	○△□	○△□	○△□
	obs_sub	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□
	obs_fla	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□
	obs_bcf	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□
	obs_sub, obs_fla, obs_bcf	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□
	obs_sub, obs_fla, obs_bcf, -O3	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○△□
Findcrypt2	-O0	○△□	●▲■	○△□	●△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O1	○△□	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O2	○△□	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O3	○△□	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Os	○△□	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Ofast	○△□	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■
	obs_sub	○△□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_fla	○△□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_bcf	○△□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf	○△□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf, -O3	○△□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
HCD	-O0	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	-O1	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	-O2	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	-O3	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	-Os	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	-Ofast	○△■	○△□	○△□	○△■	○△□	○△□	○△□	○△■	○△■	○△■	○△■	○△■	○△□	○△□	○△□	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△■	○△■	○△■	○△□	○△□	○△□	○△■
	obs_sub	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□
	obs_fla	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□
	obs_bcf	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□
	obs_sub, obs_fla, obs_bcf	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□
	obs_sub, obs_fla, obs_bcf, -O3	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□	○△□
PEiD KANAL	-O0	●▲■	●▲■	○△□	●△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O1	●▲■	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O2	●▲■	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O3	●▲■	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Os	●▲■	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Ofast	●▲■	●▲■	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	obs_sub	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_fla	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_bcf	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf, -O3	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○▲□	○△□	○△□	○△□	○▲□
Signsrch	-O0	●▲■	○△□	○△□	●△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O1	●▲■	○△□	○△□	○△■	○△□	●▲■	○△□	●▲■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O2	●▲■	○△□	○△□	○△■	○△□	●▲■	○△□	●△■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-O3	●▲■	○△□	○△□	○△■	○△□	●▲■	○△□	○△■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Os	●▲■	○△□	○△□	○△■	○△□	●▲■	○△□	●△■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	-Ofast	●▲■	○△□	○△□	○△■	○△□	●▲■	○△□	○△■	●▲■	●▲■	●▲■	●▲■	○△□	○△□	○△□	●▲■	○△□	○△□	●▲■	●▲■	○△□	○△□	●▲□	●▲■	●▲■	○△□	○△□	○△□	●▲■
	obs_sub	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_fla	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_bcf	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf, -O3	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□	○△□	○△□	○▲□	○▲□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○▲□
Where's Crypto	-O0	●▲■	●▲■	○△□	●△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	-O1	●▲■	●▲■	○△□	○△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	-O2	●▲■	●▲■	○△□	○△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	-O3	●▲■	●▲■	○△□	○△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	-Os	●▲■	●▲■	○△□	○△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	-Ofast	●▲■	●▲■	○△□	○△■	○△□	○△□	○△□	●▲■	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	○△□	●▲■	●▲■	●▲■	○△□	○△□	○△□	○△□	●▲■	○△□	○△□	○△□	●▲■
	obs_sub	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_fla	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_bcf	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□
	obs_sub, obs_fla, obs_bcf, -O3	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○△□	○▲□	○▲□	○▲□	○△□	○△□	○△□	○△□	○▲□	○△□	○△□	○△□	○▲□

Note: - ● represents the results of binaries compiled by GCC; ▲ represents the results of binaries compiled by LLVM; ■ represents the results of binaries compiled by MSVC.
- A full black shape indicates that the cryptographic function is detected in the binary, while an empty white shape indicates the opposite.
- Only LLVM supports obfuscation; flags with `obs_sub`, `obs_fla`, and `obs_bcf` are not available for GCC or MSVC (represented by dashed shapes).