Mnemocrypt: A Machine Learning Approach for Cryptographic Function Detection in x86 Executables

André Pacteau, Antonino Vitale, Davide Balzarotti, Simone Aonzo

▶ **To cite this version:**

André Pacteau, Antonino Vitale, Davide Balzarotti, Simone Aonzo. Mnemocrypt: A Machine Learning Approach for Cryptographic Function Detection in x86 Executables. BAR 2025, Workshop on Binary Analysis Research, Co-located with NDSS Symposium 2025, Usenix, Feb 2025, San Diego, United States. ⟨10.14722/bar.2025.23002⟩. ⟨hal-05127295⟩

HAL Id: hal-05127295

https://hal.science/hal-05127295v1

Submitted on 24 Jun 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Mnemocrypt: A Machine Learning Approach for Cryptographic Function Detection in x86 Executables

André Pacteau, Antonino Vitale, Davide Balzarotti, Simone Aonzo
EURECOM
firstname.lastname@eurecom.fr

***Abstract*—Cryptographic function detection in binaries is a crucial task in software reverse engineering (SRE), with significant implications for secure communications, regulatory compliance, and malware analysis. While traditional approaches based on cryptographic signatures are common, they are challenging to maintain and often prone to false negatives in the case of custom implementations or false positives when short signatures are used. Alternatively, techniques based on statistical analysis of mnemonics in disassembled code have emerged, positing that cryptographic functions tend to involve a high frequency of arithmetic and logic operations. However, these methods have predominantly been formulated as heuristics, with thresholds that may not always be optimal or universally applicable.**

In this paper, we present Mnemocrypt, a machine learning-based tool for detecting cryptographic functions in x86 executables, which we release as an IDA Pro plugin. Using a random forest classifier, Mnemocrypt leverages both structural and content-related metrics of functions at varying levels of granularity to make its predictions. The primary design goal of Mnemocrypt is to minimize false positives, as misleading results could lead analysts down incorrect investigative paths, undermining the efficacy of reverse engineering efforts.

Trained on a diverse dataset of cryptographic libraries compiled with different optimization levels, Mnemocrypt achieves robust detection capabilities without relying on predefined signatures or computationally expensive data flow graph analysis, ensuring high efficiency.

Our evaluation, conducted on 231 Portable Executable x86 Windows malware samples from different families, demonstrates that Mnemocrypt, when configured with a high confidence threshold, significantly outperforms existing solutions in terms of false positives. The few false positives detected by Mnemocrypt were only related to compression functions or complex data processing routines, further emphasizing the tool's precision in distinguishing algorithms that use instructions similar to cryptographic processes. Finally, with a median execution time of six seconds, Mnemocrypt provides the reverse engineering community with a practical and efficient solution for identifying cryptographic functions, paving the way for further studies to improve this type of model.

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23002
www.ndss-symposium.org

I. Introduction

In the domain of Software Reverse Engineering (SRE), the identification of cryptographic functions plays a pivotal role, as these functions underpin secure communication, data integrity, and confidentiality. Their correct implementation and usage are fundamental to the security of any system employing cryptographic measures as any errors, such as using outdated algorithms (e.g., MD5), hardcoded keys, or weak random number generators, can compromise defenses. SRE helps uncover these flaws, and therefore, it is essential for an analyst to be able to detect and identify the presence of cryptographic functions promptly. This can speed up the analysis of proprietary or non-standard protocols and help to expose weaknesses and design flaws in their implementations.

Additionally, compliance with regulations, like PCI, DSS, and GDPR, mandates secure cryptographic practices. In this case, SRE ensures adherence by identifying cryptographic functions and verifying their alignment with industry standards. Finally, cryptographic functions are frequently leveraged by malware authors, serving purposes such as obfuscation, secure communication with command-and-control servers, or encrypting stolen data for ransom. Identifying these functions within malware samples is critical not only for dissecting their operation but also for counteracting their malicious intent.

Developers have several options for implementing cryptographic primitives. They can use the APIs offered by the operating system (for example, the Windows "Cryptography API: Next Generation" [3]), resort to renowned libraries like OpenSSL, implement their own custom routines, or simply copy-paste the implementation of a popular cryptography algorithm from online sources. This latter case is also a popular choice for malware authors (e.g., the Dharma ransomware [4] uses an RSA implementation) because it is not detected by traditional sandboxes and dynamic analysis systems (since they usually just monitor APIs), and it does not require any external dependency.

From a reverse-engineering perspective, the usage of standard cryptographic APIs is trivial to study, while statically liked libraries or copy-pasted code makes the process much harder and time-consuming. However, cryptographic algorithms often include hard-coded constants (like prime num-

bers, S-boxes, or other predefined values), which can act as indicators for detecting the presence of these algorithms. For this reason, the SRE community often uses tools/plugins, like Findcrypt [6], that leverage this characteristic to quickly pinpoint cryptographic routines in a binary given a database of signatures. However, signatures fail to identify custom cryptographic algorithms since their characteristic patterns are unknown, generating false negatives.

Researchers have also proposed a wide array of alternative techniques based on statistical signatures, data-flow graphs, runtime information, and code similarity. As we will describe more in detail in Section II, all these techniques have pros and cons, but they often overfit with regard to reference cryptographic implementations or result in large numbers of false positives.

In this paper we present *Mnemocrypt*. It detects cryptographic functions within binary programs by using a machine learning classifier trained on features extracted from instructions mnemonics. *Mnemocrypt* is based on the work of Caballero et al. [11], where the ratio of certain classes of instructions in a function was used to determine whether the function could contain a cryptographic algorithm or not. We extended this idea by considering more features, and we developed a novel machine learning classifier trained on cryptographic libraries with different compiler optimizations. The main strength of *Mnemocrypt* is that it does not rely on signatures, so it is capable of detecting custom algorithms, and it is not computing the DFG (like [24], [27]), making it computationally fast.

Moreover, it was developed with the goal of minimizing false positives. This is a very important aspect, as false positives could otherwise lead analysts to follow a misleading path, thwarting the original purpose of supporting the SRE process. Our experiments show that it outperforms in terms of the number of false positives signatures-based solutions [6] as well as those based on mnemonics [11], [25], [26]. According to our tests performed on a dataset of real-world malware samples, *Mnemocrypt* configured to use a high value of confidence score threshold leads to 11.4% of false positives among flagged functions as cryptographic, while these percentages are 34.8% and 38.6% for Findcrypt and union of other state of the art heuristics respectively.

This paper is structured as follows. Section II presents the state-of-the-art and the building blocks of our work. Section III defines the scope of our work. Section IV describes the dataset we used. In Section V, we describe the empirical development of *Mnemocrypt*, while in Section VI, we tested it on some real-world malware samples. Finally, after discussing the limitations of our approach in Section VII, our paper concludes in Section VIII.

In the spirit of open science, we also release [1] the tool as an IDA Pro [20] plugin, but also the code needed to extract the features and train a new model, in the hope that this tool will be of help to the SRE community.

[1] https://github.com/theneonai/mnemocrypt

II. Related Work

A. *Signatures-based approaches.*

Today, reverse engineers mostly rely on byte signatures-based algorithms (such as the one implemented in the IDA plugin Findcrypt3 [6]), which search for known cryptography-related constants in their database. This type of approach is extremely fast, but the database needs to be constantly maintained, and this requires a considerable manual effort. Besides, the presence of relatively short cryptographic signatures in databases increases drastically the chances of having false positives, due to potential collisions of cryptographic signatures with byte patterns of non-cryptographic functions. On the other side, signatures cannot match custom cryptographic routines, increasing the chances of false negatives. Finally, this family of cryptography detection tools is very sensitive to obfuscation and does not take into account any semantics of the analyzed functions.

B. *Mnemonic-based approaches.*

A complementary approach was proposed in 2009 by Caballero et al. in Dispatcher [11] as part of automated reverse engineering of botnet message protocol. The authors proposed a heuristic (which we will call the *Caballero heuristic*) to count the percentage of arithmetic and bitwise instructions inside a function, flagging it as cryptographic if that percentage was higher than a fixed threshold. Other works have adopted the same idea. For instance, K-Hunt [25] narrowed the scope by flagging Basic Blocks instead of functions, whereas CIS [26], in addition to evaluating different techniques, proposed two new heuristics (*adjusted Caballero heuristic* and *asymmetric Caballero heuristic*) based on the selection of specific arithmetic and bitwise mnemonics mostly encountered in symmetric and asymmetric cryptography respectively. However, as we will see later in the paper, all of these approaches oversimplify the problem and produce a large number of false positives. Moreover, previous studies did not clearly specify the exact instructions to take into account, such as how the thresholds are experimentally computed and on which ground truth.

C. *DFG-based approaches.*

Solutions such as Crypto-DFG [24] and *Where's Crypto* [27] try to spot cryptographic functions by recognizing the Data Flow Graphs (DFG) of known cryptographic implementations. The basic idea is to build a database of signatures based on the DFGs of known cryptographic libraries. To analyze a sample, its DFG is manipulated and "standardized" so that it can be compared with the entries in the signatures database. The main drawback of this approach is that the solutions rely on external libraries for building the signatures. Even if the DFG of the algorithms is normalized to cover multiple variations, the signature is highly dependent on the reference implementation, and often results in both false positives and negatives. Moreover, proprietary ciphers cannot be identified with these methods.

D. Dynamic approaches.

Several solutions have been proposed to take advantage of additional runtime information regarding the state and execution of the target program. Most of these solutions [18], [12], [29], [31] focuses on loops, with the idea that, during both the encryption and decryption stages, loops are required to cycle through all the bytes of the input. Other works proposed alternative solutions, such as ReFormat [30], where the cumulative percentage of arithmetic and bitwise operations is used for detection, and CipherXRay, which exploits the avalanche effect of cryptographic functions. The main drawback of existing dynamic analysis tools is that they require considerable amounts of resources and time to execute the programs under analysis and trace the needed information, thus reducing the scalability of the solution.

E. Function similarity.

Finally, several works approached the problem through the lens of binary code similarity. An exhaustive list can be found in the 2021 SoK paper by Haq et al. [19]. The main problem of those approaches is that they require a reference to compare an unknown function against, which does not scale for large volumes of input functions, considering also that the reference set can easily become very large due to the presence of distinct cryptographic algorithms, several optimization levels, and different implementations.

III. Scope and Design Choices

The purpose of this work is to develop a *practical tool*, that we named *Mnemocrypt*, to support reverse engineers in statically identifying cryptographic functions. The choice of name is a reference to mnemonics, namely, the human-readable symbolic representations of these assembly instructions. They provide a more understandable way to write and read instructions without needing to deal with raw binary or hexadecimal machine code.

Mnemocrypt provides, for a given binary, a list of candidate cryptographic functions, each with its associated confidence score. By default, the classifier is tuned to reduce the number of false positives and provides the analyst with flagged functions' names along with their respective confidence scores, indicating the probability of being cryptography-related in the sense of our classifier.

In designing *Mnemocrypt*, we deliberately opted for a machine learning model rather than traditional signature-based approaches. This decision is rooted in the following considerations:

1) Adaptability to Variations: Signature-based methods rely on predefined patterns or rules specific to known cryptographic functions. While effective for detecting previously encountered functions, they struggle with novel, obfuscated, or modified implementations of cryptographic algorithms. Moreover, cryptographic implementations in binaries often vary significantly due to compiler optimizations or architecture-specific instructions. In contrast, an ML model learns to identify patterns from training data, enabling it to generalize to new or unseen functions.
2) Reduction of False Positives: One of the key objectives of *Mnemocrypt* is to minimize false positives to save analysts' time. Signature-based systems often lead to higher false positives when generalized to diverse binaries, as minor deviations from a known signature might still match. Moreover, signature-based solutions only provide a true-false result. In our case, by leveraging confidence scores in the ML model, *Mnemocrypt* provides more nuanced results, allowing analysts to adapt the tool to their needs and focus on functions with higher likelihoods of being cryptographic.
3) Function-Level Classification: Since SRE frameworks decompose programs into functions, we designed *Mnemocrypt* to classify at the function level by extracting rich features such as instruction patterns, software metrics, and statistical properties. This feature-driven approach leverages the strength of ML to capture complex patterns that would be difficult to encode as static signatures.

Finally, a primary challenge in automating the detection of cryptographic functions pertains to the definition of cryptographic functions themselves. First of all, cryptographic functions can be classified into several categories, including hash functions, encryption algorithms (subsequently divided into symmetric and asymmetric), and digital signatures — each with its distinct formal definition. Nevertheless, a reverse engineer would seek to identify them irrespective of their categorization. Despite the absence of any explicit definition for cryptographic functions, it is possible to identify the elements that are of particular concern to reverse engineers during the static analysis of binaries. These elements may pertain to the content or structure of functions. The features of our machine learning model correspond to these elements. We will elaborate on this point in the following sections.

IV. Training Dataset

In order to provide our supervised machine learning model with ground truth, we used the well-known cryptographic libraries Libsodium 1.0.20 [7] and OpenSSL 3.3.1 [9]. OpenSSL has been chosen as a comprehensive cryptographic library, including implementations of the most widely used symmetric and asymmetric cryptographic algorithms, as well as some cryptographic hashes. The purpose of including Libsodium is to provide the model with an alternative implementation of some cryptographic algorithms so that the model could generalize better.

The training set is composed of 32-bit binaries statically linked with OpenSSL and Libsodium, compiled with debugging symbols (to be able to read the names of functions later) using Clang v14.0 [2], GCC v11.4 [17] and MSVC v14.39 [28] compilers for all of their main optimization levels: `O0`, `O1`, `O2`, `O3`, `Ofast`, `Os` and `Oz` for Clang; `O0`, `O1`, `O2`, `O3`, `Ofast` and `Os` for GCC; `Od`, `O1`, `O2` and `Ox` for MSVC. This variety makes it possible for the model to learn variations

Table I: Number of functions in Libsodium and OpenSSL binaries from the training set.

Configuration	Libsodium Total	Libsodium Crypto	OpenSSL Total	OpenSSL Crypto
Clang O0	1,389	53	19,482	173
Clang O1	1,389	53	19,224	173
Clang O2	1,197	50	14,711	175
Clang O3	1,196	50	14,697	172
Clang Ofast	1,198	50	14,699	174
Clang Os	1,210	53	15,146	172
Clang Oz	1,274	60	15,523	168
GCC O0	1,394	53	21,060	173
GCC O1	1,228	59	15,283	173
GCC O2	1,229	59	15,346	174
GCC O3	1,212	55	14,760	175
GCC Ofast	1,214	55	14,762	175
GCC Os	1,259	59	15,437	171
MSVC Od	1,290	58	13,300	203
MSVC O1	1,175	62	12,444	203
MSVC O2	1,076	64	11,189	202
MSVC Ox	1,076	64	11,189	202

of mnemonics for the same functions but built with different configurations.

Ensuring the reliability of the ground truth is paramount for the development of any supervised machine learning model. Given the scarcity of cryptographic functions relative to the total number of functions, the presence of incorrectly labeled functions can have a substantial adverse impact on the model's generalization capacity and performance. Therefore, by manually inspecting the original source code, we labeled each function as *cryptographic* or *non-cryptographic*.

As shown in Table I, the only significant discrepancy in the number of cryptographic functions is observed in OpenSSL's MSVC-built binaries, where a substantial number of cryptographic functions were not inlined as in GCC and Clang.

It is worth reporting that during the labeling process, we realized that three subclasses of cryptographic functions can be distinguished: *core*, *hybrid*, and *auxiliary*. The *core* functions' purpose is to only perform cryptographic operations and that without calling any other cryptographic function. For example, most of the block encrypting functions are core. The *hybrid* functions perform some cryptographic operations, but in addition, call some other cryptographic function or repeatedly call some bitwise elementary functions such as rotation of a vector by a specific number of bits. Operation modes functions are typical examples of hybrid functions. The *auxiliary* functions are not themselves performing any encryption, decryption, hashing, cryptographic key expansion, or derivation but correspond to usual operations defined on complex objects, which are often manipulated in asymmetric cryptography. Functions defining basic operations on big numbers or cryptographic curves typically fall into this category.

V. Development

A. Categorization

The most fundamental yet effective approach to extracting information from mnemonics within a specific function involves identifying their overall behavior, which includes activities such as data movement, computation, and control flow management. In light of this, a mnemonics categorization is needed. We used the one from the ASM86 Language Reference Manual [22], which distinguishes six categories of mnemonics: data transfer, arithmetic, logic, string manipulation, control transfer, and process control.

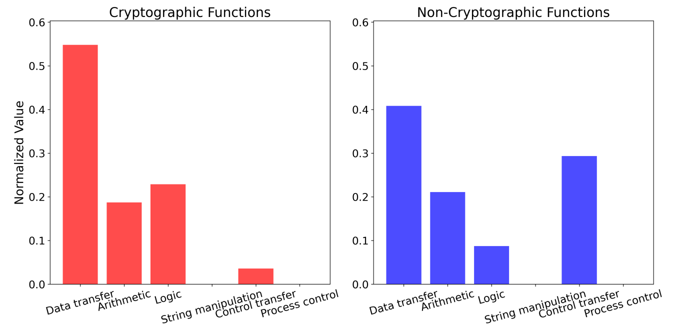

Figure 1: Normalized distributions of mnemonics categories in the training set (averaged over functions)

In Figure 1, we plot the normalized distributions of occurrences of each category in our training set. It shows that cryptographic functions have clearly different distributions compared to non-cryptographic ones. This difference could derive from the choice of the dataset, so even if metrics over categories can help the model to capture high-level information on functions, they are not enough, and some mnemonics such as logical AND, XOR, bit shifting, multiplication, addition, etc. can be more "cryptography-related" than others [26]. Thus, separate metrics for such mnemonics are relevant to use to capture details of function behavior.

B. Mnemonic Roots

Mnemonics are often appended with specific prefixes or suffixes, depending mainly on the type of data with which they are used [21]. We define a *root* as the most semantically meaningful morphological part of a given mnemonic. To avoid considering all these varieties separately, we regrouped such semantically close mnemonics under the same roots. They are essential for *Mnemocrypt* to process and analyze binary instructions effectively. Roots represent the fundamental semantic operations encoded in mnemonics, independent of the specific prefixes or suffixes coming with various instruction sets. For example, mnemonics like `vaddph` and `add` share the root (`ADD`), reflecting their common purpose of performing addition, regardless of their architectural or data-specific modifiers. Aggregating mnemonic occurrences under their corresponding roots generalizes and makes the following statistical analysis more relevant for our classifier.

When *Mnemocrypt* parses a given mnemonic from a function, it matches it using regex rules against a predefined root with an associated category. If the mnemonic matches several roots, only the longest one is supposed to be kept – so the

roots are preliminarily sorted by length to avoid unnecessary computations.

However, the roots are naturally prone to constraints of morphological nature. This sometimes required manual adjustments to make some roots immune to this kind of limitation. For example, for the shift operations there is no unique root long enough to be specific. Just to cite a few, once we matched the `SHL`, `SHR`, `SAL`, and `SAR` instructions, we aggregated them under a unique root, which we named `SH` for convenience. The same approach has been adopted for data move, addition, subtraction, and rotation operations because they are essentially the ones on which *Mnemocrypt* is computing statistics.

In our implementation, each root is deliberately assigned to only one category as a structuring choice. This ensures that roots form more semantically precise groups, offering finer granularity compared to the broader and more general groupings of categories. This led us to take additional precautions in the choice of roots in order to avoid potential inconsistency errors in matching. Thus, for example, the mnemonics `mov` and `movsx` from the data transfer category can not be all directly associated with the same root `MOV` because the root `MOVS`, belonging to string manipulation, would match `movsx` as it is longer than the root `MOV`. To avoid this, a separate root `MOVSX` has been introduced so that after the mnemonic-root matching stage, the occurrences of roots `MOV` and `MOVSX` are aggregated under the data move root named `MOV`. Full documentation of predefined prefixes and categories with their roots is available in the repository of *Mnemocrypt*.

C. Feature Selection.

In order to provide our model with information about the structure of functions, we considered the following state-of-the-art features [15], [16], [23]: number of basic blocks, number of loops, maximum depth of loops, and cyclomatic complexity. The choice of these control flow-related metrics comes from the fact that usual cryptographic functions present a well-structured and repetitive control flow.

In addition to the structural metrics described above, statistics on the categories and some of roots matched by the mnemonics processed by *Mnemocrypt* are also included as features. This information allows the model to gain insight into the kind of operations performed in the given function. We have included detailed statistics on categories (average occurrence per basic block, median, standard deviation, min, and max) as features because high-level information is more likely to be exploitable if analyzed from multiple perspectives. In addition, we have considered individual densities of the roots used in the computation of adjusted and asymmetric Caballero ratios, the density of a root in a function being defined as its occurrence divided by the total number of instructions of the function, without counting `MOV` instructions [26]. We have also included the Caballero ratios themselves in the features in order to emphasize the importance of these quantities for the model and to encompass Caballero heuristics.

We also decided to include average values per basic block of the 1-grams (thus equivalent to roots in our case) and bigrams

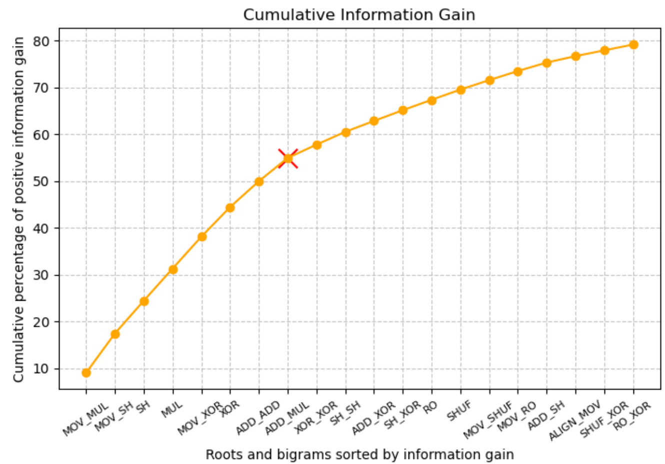

Figure 2: Cumulative information gain of roots and bigrams. Graph truncated after the top 20. The elbow is marked with a red cross.

representing the highest information gain (IG) in the training set with respect to our labeling. The strategy is to determine the most influential features by applying the elbow method to the IG distribution obtained by first computing the Term Frequency-Inverse Document Frequency (TF-IDF) scores of each feature candidate. In our case, the “Term Frequency” corresponds to the frequency of the feature over basic blocks of functions, and in the ”Inverse Document Frequency,” the term “document” corresponds to a function.

In Figure 2, we plot the cumulative IG of the top 20 roots and bigrams sorted in decreasing order of their contribution, the percentage being relative to the roots and bigrams having positive IG. A clear change of slope can be observed after the 8th feature candidate (bigram `ADD_MUL`); thus, according to the elbow method, we decided to keep the first 8 features. It is important to note that the bigrams here are undirected, i.e., a pair of roots where order does not matter. This decision comes from the fact that the cumulative IG curve with directed bigrams did not present any clear elbow, thus increasing the risk of making a sub-optimal feature selection decision.

D. True and False Positives Optimizations

Some mnemonics are particularly interesting in cryptography detection because they are part of Intel cryptographic extension, like `AES-NI` and `SHA`. If a function has one or more of these mnemonics, it is immediately considered cryptographic.

In addition to this, the algorithm eliminates in advance some potential false positives by directly flagging as non-cryptographic the functions that contain mnemonics involving floating point numbers manipulation. Such functions correspond in our case to encountering at least one mnemonic with the prefix ‘`f`,’ standing for `float` [21], or of at least one root corresponding to real mathematical function like `cos` or `sin`. This preliminary filter is due to the fact that cryptographic algorithms require exact computations, which

Table II: Xmllint FPs across compilers and their optimization levels. The numbers correspond to the order of optimization levels written in Section IV.

Trees	Clang	GCC	MSVC
100	5, 1, 1, 1, 1, 2, 1	9, 0, 0, 1, 1, 1	4, 1, 2, 2
1,000	4, 1, 1, 1, 1, 2, 2	9, 0, 0, 1, 1, 1	4, 2, 3, 3

is not guaranteed in floating-point arithmetic. We verified this claim, and we found no floating-point calculations among the functions of the training set that we labeled as cryptographic.

E. Training and Validation

Our training set is highly imbalanced (as can be seen in Table I), which is a constraint that not all supervised machine learning models can adapt to. In addition, the model should be able to handle non-linearity – as cryptographic functions detection is a complex problem. Another important, yet not necessary, criterion for the choice of the model is its explainability, as it allows one to choose more and more relevant features by training draft versions of the model and studying the importance of each feature. The Random Forest algorithm satisfies all these constraints and this is why we have chosen it as the basis for *Mnemocrypt*. We addressed the issue related to the imbalanced nature of the dataset by employing the Synthetic Minority Oversampling Technique (SMOTE) [13].

The use of the Random Forest model implies the tuning of hyperparameters: the depth of the trees and their number. In our study, we decided not to introduce any pruning because the training process was relatively fast, and we did not observe any overfitting during our preliminary tests. To choose the number of trees, we tested models trained with 100 and 1,000 trees (values that represent a compromise between training time and generalization ability of the model) on known non-cryptographic applications: Al-Khaser [1] and Xmllint from Libxml2 [8]. We chose these two open-source repositories because we know their codebase well and it does not contain cryptographic functions. Xmllint was built with the same settings (compiler/optimizations) as the training set. Al-Khaser was only built with the MSVC compiler and its different optimization levels because it is the only compiler supported by this project.

Moreover, even if our primary goal is to reduce false positives as much as possible, the ability to detect cryptographic functions is also important for the algorithm to be efficiently used in practice. For this reason, we also tested the models on manually crafted binaries implementing common cryptographic algorithms: AES, Blowfish, ChaCha20, DES, MARS, RC4, RC6, RSA, Skip32, TEA, and Twofish. They were generated with the same settings (compiler/optimizations) as the training set. In the following, we will refer to these programs as “toy libraries”. During model testing, the confidence threshold of the cryptographic class was set to 50%, i.e., any function with confidence below 0.5 would be classified as non-cryptographic.

Table III: toy libraries FNs across different optimization levels.

Trees	Clang	GCC	MSVC
100	20, 17, 15, 14, 14, 18, 21	17, 20, 23, 12, 12, 19	8, 22, 11, 12
1,000	21, 17, 14, 14, 14, 18, 20	16, 20, 22, 13, 13, 20	7, 22, 11, 11

For Al-Khaser, both trained models produced 2 false positives out of a total of 4,574 functions, while in the case of Xmllint, the model with 100 trees resulted in 33 false positives, and the one with 1,000 trees produced 36 false positives out of a total of 92,933 functions. We report the results for Xmllint in Table II. It is interesting to observe that the number of false positives is consistently higher for non-optimized binaries. This could be because non-optimized binaries typically contain more functions than their optimized counterparts. A more plausible explanation, however, is that optimized binaries are more similar to each other than they are to their corresponding non-optimized versions. Since our dataset includes more optimized binaries than non-optimized ones, the model is less trained on non-optimized cases, leading to the observed disparity.

A manual inspection of the false positives revealed that they are related to either non-cryptographic hash functions, parsers, or other complex data processing functions. This is due to the fact that these functions share certain characteristics with cryptographic functions, as they frequently involve the use of low-level operations, such as bitwise shifts, XORs, and modulo operations.

For the toy libraries, comprising a total of 177,610 functions, none of the models produced any false positive, with the exception of two non-cryptographic libc functions, `int_mallinfo` and `_dl_lookup_direct`, which were flagged in all executables built with Clang and GCC. These are not part of libc’s public APIs and are used internally for specific purposes related to memory allocation statistics and dynamic symbol lookup, respectively. The model trained with 100 trees resulted in slightly more false negatives than the one trained with 1,000 trees (275 vs 273, see Table III), with a total of 564 cryptographic functions.

Overall, based on these preliminary tests, we decided to use the model trained with 100 trees for the following tests, as it has fewer false positives in the validation set (which is the most important metric to minimize), even though this comes at the cost of potentially detecting slightly fewer cryptographic functions compared to the model trained with 1,000 trees.

F. Interpretability

Another advantage of Random Forest models is the ability to gain insight into feature ranking by examining the learned weights of the features. These weights are determined by the total information gain (reduction in Gini impurity) contributed by each feature across all decision trees in the forest. The weights represent the cumulative contribution of a feature to the model’s accuracy, normalized to sum to one. They are always positive and indicate the significance of each feature in differentiating between cryptographic and non-cryptographic functions in the training set.

Table IV: Top 15 features weights in the retained model. nb = number of

Rank	Feature Name	Weight
1	default_caballero_ratio	0.132
2	adjusted_caballero_ratio	0.111
3	mean_mov_sh	0.087
4	density_sh	0.084
5	mean_sh	0.081
6	max_nb_instr	0.071
7	mean_mov_xor	0.051
8	density_xor	0.049
9	mean_logic	0.046
10	nb_instr	0.044
11	mean_xor	0.032
12	mean_add_add	0.018
13	std_dev_data_transfer	0.017
14	std_dev_logic	0.017
15	density_and	0.017

We report the top 15 in Table IV. The fact that two of the three Caballero heuristics are in the top two positions suggests that their choice was well-motivated. However, the asymmetric Caballero ratio metric has a relatively low rank (i.e., 25th out of 43), with a weight of 0.006. This may be attributed to the potentially inaccurate approximation of asymmetric cryptographic functions or the influence of the training set, which exhibits a notable dominance of symmetric cryptography.

Another important observation is that this ranking is dominated by the influence of the `XOR` and `SH`ift instructions, which are commonly used in cryptographic functions due to their simplicity, efficiency, and versatility in bit-level manipulations. These instructions are computationally inexpensive and widely supported across all x86 processors, making them ideal for high-performance implementations. The shift instruction enables controlled bitwise movements, facilitating operations such as key scheduling and diffusion, while the `XOR` instruction is integral for combining data streams and introducing nonlinearity in encryption algorithms. Together, they form the foundation for many cryptographic primitives, offering a balance of speed and functionality critical for secure and efficient cryptographic processes.

VI. Testing

We have tested *Mnemocrypt* on a dataset of 231 PE x86 Windows malware samples, all belonging to different families. To build the dataset, we replicated and filtered the dataset used in 2023 by Dambra et al. [15] by excluding packed samples detected as such by Detect-It-Easy [5] and PackGenome [10]. Moreover, we removed the samples marked as corrupted by VirusTotal [14] and randomly selected one sample per each family. Even though this choice required us to manually verify the results (because there is no ground truth about the presence of cryptographic functions for the families in our dataset), we believe that malicious code analysis is the most typical use case for *Mnemocrypt*. The final dataset contains a total of 277,193 functions.

Table V: Manual labeling of Mnemocrypt with threshold 90%. CDP = Complex Data Processing

Function type	Count	Mean	StdDev	Median	Max	Min
True Positive	140	0.97	0.029	0.98	1.0	0.9
Compression	4	0.935	0.006	0.935	0.94	0.93
CDP	14	0.944	0.029	0.960	0.98	0.9

For this experiment, we set the confidence score to 90%. Namely, unless the confidence score of a given function is higher than 0.9 it will be classified as non-cryptographic. We chose to test *Mnemocrypt* with this threshold to achieve a sufficiently low number of flagged functions for manual inspection. However, *Mnemocrypt* is fully configurable, so that the analyst can manually tune the confidence score according to the experiment and expected results.

A. Results

When using the conservative threshold of 0.9, *Mnemocrypt* flagged 158 (0.055%) functions as cryptographic. We excluded from this measurement the cryptographic functions detected as such by the presence of mnemonics from Intel AES-NI or SHA extensions, as our model does not contribute to this discovery.

Upon a manual examination of all results, we concluded that 140 functions were indeed related to cryptographic algorithms, while 18 were false positives. A closer look at the false positives revealed that they were all related to compression and complex data processing. In the first category, we found functions performing discrete cosine transform used in JPEG compression, while complex data processing routines employ many instructions to perform bit operations on data.

Table V shows the confidence score *Mnemocrypt* assigned to the three types of functions. As we can see, the mean and median values of confidence scores of true positives are significantly higher than the ones of false positives. Moreover, if we look at the overall ranking of the 158 functions (in decreasing order of confidence), the first false positives appear only at position 62.

B. Comparison with Findcrypt and Caballero heuristics

To test the Caballero heuristics on our dataset, we considered the logical union of the adjusted and asymmetric Caballero heuristics [26]. This means that if at least one of the ratios is above the associated threshold, the given function is flagged as cryptographic. This combination arises from the fact that adjusted or asymmetric Caballero heuristics, if used alone, would, by design, lead to missing almost all asymmetric or symmetric cryptography respectively. As a result, we identified 663 functions flagged as cryptographic. Since this number is too high to analyze manually, we randomly extracted 158 functions (the number chosen to analyze the same number of functions for all tested approaches) and manually inspected them. According to our analysis, 61 are false positives (38.6%).

Findcrypt, in its original implementation [6], does not output functions but rather identifies cryptographic byte sequences.

Table VI: Execution time (in seconds) of Mnemocrypt, Caballero heuristics, and Findcrypt on malware samples.

Category	Mean	StdDev	Median	Max	Min
Mnemocrypt	12	15	6	148	4
Caballero	4	5	2	40	1
Findcrypt	2	3	2	31	1

Therefore, we considered functions flagged as cryptographic by Findcrypt to correspond to those obtained via cross-references on the byte sequences. Based on this approach, Findcrypt, when run on our malware dataset, flagged 530 functions as cryptographic. Among 158 randomly chosen functions that we analyzed, 55 were false positives (34.8%).

Thus, according to our tests, *Mnemocrypt*, used with a threshold of 90% on our malware dataset, outperforms Findcrypt and the combination of adjusted and asymmetric Caballero heuristics in terms of the number of false positives, namely, 11.4% FPs against 34.8% and 38.6% of the previous solutions.

C. Execution time

To measure the execution time, we performed all tests on a system with an Intel i7-6820HQ processor, 16GB RAM, running Windows 10 and using IDA Pro 8.3. The results are summarized in Table VI. Overall, *Mnemocrypt* is roughly three times slower than Findcrypt and the Caballero heuristics. However, we believe that a median time of six seconds to analyze an entire program is perfectly reasonable and almost negligible if compared to the human time required to reverse engineer the same amount of code.

VII. Limitations

This section discusses the limitations faced by *Mnemocrypt* and suggests avenues for improvement to enhance its capabilities and performance.

One significant limitation lies in the training dataset, which could benefit from expansion and diversification. The performance of supervised ML models is highly dependent on the quality and comprehensiveness of the training sets. By enriching the dataset, *Mnemocrypt* can improve its ability to reduce false positive rates while identifying a broader range of cryptographic functions. Specifically, the inclusion of additional cryptographic libraries written in diverse programming languages could complement the current dataset dominated by C-based libraries.

Extending *Mnemocrypt* to support x86-64 bit architecture is another important improvement avenue. Currently, the tool focuses primarily on the 32-bit x86 architecture, which limits its applicability to modern software that increasingly utilizes 64-bit architectures. Incorporating x86-64 support would enable *Mnemocrypt* to analyze a broader range of binaries, especially those built for contemporary systems, thus enhancing its practicality and scope. We prioritized 32-bit binaries as our primary focus was to support malware analysis, and a recent 2022 study [15] found that in the VirusTotal feed 88% of the malicious PE samples were 32-bit executables, 8% were DLLs (32-bit or 64-bit), and 4% were 64-bit executables. However, it is important to note that *Mnemocrypt* is not explicitly based on the hypothesis of being used on 32-bit binaries and, therefore, can technically also be used to analyze 64-bit executables.

The dataset could further be augmented by including non-cryptographic applications to reduce the false positives related to non-cryptographic functions. For instance, false positives involving compression algorithms, such as JPEG, could be reduced by including binaries containing such functions and labeling them as non-cryptographic. However, this requires careful validation to ensure that these binaries do not inadvertently contain cryptographic functions, such as hashes or authentication mechanisms, which are increasingly common in modern software due to security requirements.

Our solution relies only on assembly instruction mnemonics - a deliberate choice we made to keep the model lightweight and efficient. However, it might be possible to obtain better results by also utilizing the full information contained in the instruction operands. In addition, grouping mnemonics under common roots does not always capture nuances, such as distinctions between vectorized and scalar operations. For example, treating mnemonics from vectorized instructions as multiple occurrences of associated standard mnemonics could provide a more accurate statistical representation, enhancing the tool's effectiveness.

Finally, the efficiency of *Mnemocrypt* could also be improved by reimplementing the current Python-based implementation in a more performant language, such as C++. Leveraging the IDA C++ SDK instead of IDAPython would likely lead to a significant speed improvement, enhancing the tool's usability for large-scale or time-sensitive analyses.

By addressing these limitations, *Mnemocrypt* can evolve into a more comprehensive, accurate, and efficient framework for the detection and analysis of cryptographic functions.

VIII. Conclusion

In this paper, we presented *Mnemocrypt*, a static analysis tool designed to detect cryptographic functions within x86 executables. Built upon a supervised machine learning framework utilizing a random forest model, *Mnemocrypt* leverages 43 carefully selected features, encompassing structural metrics of functions and statistical metrics derived from the mnemonics of their assembly instructions. The tool was trained on manually labeled datasets generated from 32-bit builds of Libsodium and OpenSSL, compiled using Clang, GCC, and MSVC at various optimization levels, ensuring broad applicability and robustness.

Mnemocrypt serves as a generalization and enhancement of Caballero heuristics, incorporating and extending their principles. It is interpretable, offering insights into the feature weights of the trained model, and highly customizable, enabling users to enrich feature sets or adjust detection thresholds post-training. Our evaluation on a dataset of 231 real-world malware samples demonstrated that *Mnemocrypt* achieves a false positive rate of 11.4% relative to the number of flagged

cryptographic functions when applied with a 90% confidence threshold. Notably, all false positives were related to either compression functions or complex data processing, underlining the tool's effectiveness in narrowing down candidate cryptographic functions.

To facilitate adoption, we have released *Mnemocrypt* as open-source software, alongside an IDA Pro plugin that allows users to analyze binaries and obtain confidence scores for functions predicted as cryptographic with a confidence level above 0.5. Additionally, we adapted the Findcrypt3 plugin to integrate its cryptographic algorithm identification information into *Mnemocrypt*'s output, enhancing its usability and functionality.

Through its interpretability, flexibility, and demonstrated low number of false positives, *Mnemocrypt* provides a powerful, efficient solution for detecting cryptographic functions, addressing a critical need in cybersecurity analysis while paving the way for further advancements in automated binary analysis.

Acknowledgment

We express our gratitude to Slasti Mormanti for his invaluable cryptanalysis expertise, which was instrumental in the success of this study.

This work benefited from two government grants managed by the French National Research Agency with references: "ANR-22-PECY-0007" and "ANR-23-IAS4-0001".

References

[1] Al-khaser. https://github.com/ayoubfaouzi/al-khaser.
[2] Clang. https://clang.llvm.org/.
[3] Cryptography API: Next Generation. https://learn.microsoft.com/en-us/windows/win32/seccng/cng-portal.
[4] Dharma Ransomware Analysis: What It's Teaching Us. https://www.fortinet.com/blog/threat-research/dharma-ransomware--what-it-s-teaching-us.
[5] Die. https://github.com/horsicq/Detect-It-Easy.
[6] Findcrypt3. https://github.com/polymorf/findcrypt-yara.
[7] Libsodium. https://doc.libsodium.org/.
[8] Libxml2. https://github.com/GNOME/libxml2.
[9] OpenSSL. https://www.openssl.org/.
[10] Packgenome. https://github.com/packgenome/PackGenome-Artifacts.
[11] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security*, 2009.
[12] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *ACM Conference on Computer and Communications Security*, 2012.
[13] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
[14] Chronicle Security. VirusTotal. https://www.virustotal.com/, Accessed March 2, 2025.
[15] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 60–74, 2023.
[16] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp)*, pages 472–489. IEEE, 2019.
[17] GNU. GCC. https://gcc.gnu.org/.
[18] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *International Symposium on Recent Advances in Intrusion Detection*, 2011.
[19] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *Acm computing surveys (csur)*, 54(3):1–38, 2021.
[20] Hex-rays. IDA Pro. https://hex-rays.com/ida-pro/, Accessed March 2, 2025.
[21] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual.
[22] Intel Corporation. ASM86 Language Reference Manual, 1981, 1982, 1983.
[23] Sudesh Kumar Santhosh Kumar, Sandeep Pai Kulyadi, Pavitra Mohandas, MJ Shankar Raman, and VS Vasan. Computation of cyclomatic complexity and detection of malware executable files. In *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–5. IEEE, 2021.
[24] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *ACM Symposium on Information, Computer and Communications Security*, 2015.
[25] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *ACM Conference on Computer and Communication Security*, 2018.
[26] Felix Matenaar, Andre Wichmann, Felix Leder, and Elmar Gerhards-Padilla. Cis: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 46–53. IEEE, 2012.
[27] Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code. In *USENIX Security Symposium*, 2021.
[28] Microsoft Corporation. MSVC. https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options.
[29] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services. In *USENIX Security Symposium*, 2013.
[30] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14*, 2009.
[31] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *IEEE Symposium on Security and Privacy*, 2017.