Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism

Pierre Lestringant* Frédéric Guihéry* Pierre-Alain Fouque†

*AMOSSYS
R&D Security Lab
Rennes, France
first.last@amossys.fr

†Université de Rennes 1 and
Institut Universitaire de France
Rennes, France
first.last@univ-rennes1.fr

ABSTRACT

Softwares use cryptographic algorithms to secure their communications and to protect their internal data. However the algorithm choice, its implementation design and the generation methods of its input parameters may have dramatic consequences on the security of the data it was initially supposed to protect. Therefore to assess the security of a binary program involving cryptography, analysts need to check that none of these points will cause a system vulnerability. It implies, as a first step, to precisely identify and locate the cryptographic code in the binary program. Since binary analysis is a difficult and cumbersome task, it is interesting to devise a method to automatically retrieve cryptographic primitives and their parameters.

In this paper, we present a novel approach to automatically identify symmetric cryptographic algorithms and their parameters inside binary code. Our approach is static and based on Data Flow Graph (DFG) isomorphism. To cope with binary codes produced from different source codes and by different compilers and options, the DFGs is normalized using code rewrite mechanisms. Our approach differs from previous works, that either use statistical criteria leading to imprecise results, or rely on heavy dynamic instrumentation. To validate our approach, we present experimental results on a set of synthetic samples including several cryptographic algorithms, binary code of well-known cryptographic libraries and reference source implementation compiled using different compilers and options.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring—*reverse engineering, and reengineering*; D.4.6 [**Information Systems Applications**]: Cryptographic controls; I.2.8 [**Problem Solving, Control Methods, and Search**]: Graph and tree search strategies

Keywords

Static Binary Analysis, Reverse Engineering, Cryptography

ASIA CCS'15, April 14–17, 2015, Singapore.

ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714639.

1. INTRODUCTION

When assessing software security, analysts pay special attention to cryptographic algorithms for several reasons. First the algorithm choice may be problematic. Algorithms that have not been thoroughly analyzed or for which practical attacks have been published are insecure and should not be used. For instance, practical collision can be found for the MD5 hash function [33], however it is still not uncommon to find it in applications. Second, the algorithm's implementation may not be compliant with the specifications. It was the case for GnuPG that used a faster implementation of the signature scheme DSA by relying on smaller nonces [23]. This was instrumental in an attack that enabled to recover the signer's private key in less than a second. Because such liberties taken from the specifications may reduce the algorithm security, they should be detected so that they can be carefully analyzed. Third, the algorithm's implementation may leak information about secret data. Today, it is well-known that the most efficient attacks use side-channel information. Thus it is essential to check if these attacks exist on the implementation being evaluated. The cache timing attack against the table implementation of AES [26] and the recent attack based on acoustic leakage against RSA [12] are good examples of practical side-channel attacks. Finally, despite a secure implementation, poorly generated input parameters can still affect the system security. Therefore the analyst will be particularly interested in how the key, the IV or the padding are generated. For instance, attacks known as padding oracle [21, 28] target insecure padding scheme.

The ability to conduct these analyses at the binary level is important because source code is not always available. And even when it is, the compiler may introduce security breaches [5, 30] that can only be revealed by looking at the binary code. It is clear that a first prerequisite will be to find and identify cryptographic primitives in binary code. Therefore, designing automated mechanisms to address this specific problem should be highly profitable for security experts who have a limited amount of time to assess the security of software.

The solution we proposed, based on static analysis and more particularly on data flow graph isomorphism is able to identify and accurately locate cryptographic algorithms and their parameters inside binary executables. Our solution targets symmetric cryptographic algorithms. We choose not to address the problem of code obfuscation because the objective of our method is to deal with the kind of general software that is typically reviewed during product certification

and not to specifically target applications such as malwares that would have involved heavy obfuscation. In summary, this paper makes the following contributions:

- We present a sound and efficient approach to automatically identify and locate symmetric cryptographic algorithms and their parameters in binary code.
- We propose a normalization process and a signature matching scheme that is resistant against compiler optimizations and source code variations.
- We present the results of an experimental study that demonstrates the efficiency of our approach on well-known cryptographic libraries.

The remainder of the paper is structured as follows. In Section 2 we present more formally the previous works on the field of cryptographic identification. Then, we present at a high-level our solution in Section 3: the DFG is described in Section 4, the normalization process in Section 5, the signature design in Section 6 and the subgraph isomorphism algorithm due to Ullman is recalled in Section 7. Finally, in Section 8 we give experimental results.

2. RELATED WORK

The topic of cryptographic primitives identification in binary programs has been previously addressed. In this Section we will first present and discuss the limitations of two characteristics used to identify cryptographic code: data constant and input/output relationship. Finally we will briefly list other lines of work that are closely related.

2.1 Data Constant

Symmetric cryptographic algorithms often contain specific constants. These constants range from a single value of a few bits to large lookup tables of several kilobytes. Because it is unlikely to find them in different algorithms, they can be used to identify cryptographic code. This is a widely used technique and it has been implemented in several publicly available tools, such as Findcrypt2 (IDA plugin) [14], KANAL (PEiD plugin), or H&C Detector, to name but a few. As far as we know all tools based on constant identification solely rely on static analysis. However we should bear in mind that constant identification can also be performed using dynamic analysis. This is particularly useful in the case of packed programs, or for constants that are dynamically computed.

Despite its simplicity and its efficiency the constant identification technique does not meet our precision and reliability requirements. We exhibit some of the limitations of constant identification with the following example, that uses an AES table implementation as a target. A reminder on possible AES implementations is given in Appendix A. Given a binary program, let us assume the AES S-Box has been detected by one of the previously listed tools. The precise location of the AES encryption/decryption routines still needs to be investigated. In fact multiple parts of the program can access the S-Box, such as: the AES key schedule (either for encryption or decryption) or a 4 kilobytes lookup tables generation routine. Moreover the parameters (including the key size) have not been identified. Finally, the detected algorithm could be another cryptographic primitive that uses the AES S-Box, such as the Fugue hash function [15] or the LEX stream cipher [6].

To conclude, constant identification may be a very effective first step, but it should not be used as a standalone technique to precisely and completely uncover cryptographic primitives.

2.2 Input/Output Relationship

The I/O relationship identification technique relies on the following hypothesis: given a cryptographic primitive $f : I \rightarrow O$, the relationship between an input value and the corresponding output value identifies f with an overwhelming probability. In other words, if during an execution a program fragment reads a value $i \in I$ and writes a value $o \in O$ such that $f(i) = o$, we can conclude that this fragment implements the primitive f.

From a practical perspective, the targeted program is first executed in a Dynamic Binary Instrumentation (DBI) environment. The exact values manipulated by the program are recorded in an execution trace. The execution trace is then split into fragments using the loop abstraction according to the hypothesis that cryptographic code manipulates their arguments within loops. This observation is used to narrow the search space by only considering what happens inside loops' body. For each fragment F a set of input and output arguments (called respectively I_F and O_F) are reconstructed from its instructions' operands. Finally, given a database of cryptographic reference implementations P, we search $f \in P$ such that: $i \in I_F$ and $f(i) \in O_F$.

Gröbert *et al.* [13] were the first to use the I/O relationship to recognize cryptographic primitives. Zhao *et al.* [34] also used the I/O relationship, but instead of using the loop abstraction to extract candidates, they split the trace in terms of functions. More recently, Calvet *et al.* [7] proposed a complete and detailed tool named Aligot based on the I/O relationship. Aligot includes a new loop definition and the analysis of the data flow between loops to produce the best candidate for I/O testing.

However in our opinion two problems remain unaddressed by this approach. The first problem is fragment extraction in the case of unrolled loops. Loop unrolling is a common technique used by software engineers or compilers to reduce branch penalties and increase instruction level parallelism. It is not uncommon to encounter unrolled loops in cryptographic code as performance is a major concern. However according to our observations even Calvet loop's definition is unable to cope with unrolled loop: since loop unrolling is not the ultimate step of an optimization process, unrolled loops are not the exact repetition of a sequence of dynamic instructions.

The second problem concerns the parameters reconstruction. Instruction's operands have to be aggregated to form input and output parameters for reference implementations. Previous works [13, 34, 7] have provided rules to concatenate memory operands (based on spatial proximity either in code or in memory). However nothing has been proposed to concatenate register operands. Without external information, we must resort to a brute force approach. Given n operands, the number of parameters resulting from every combination and permutation of k operands, is: $\frac{n!}{(n-k)!}$. This is not tractable in practice if we consider that the same parameter may be distributed in the memory and in registers, as it is for instance often the case for the states of the

AES or MD5. In that case n is the sum of the number of registers operands an the number of memory operands.

To conclude, the I/O relationship identification method has several qualities: parameters identification, no false positive and resilient against implementation variations. However we have identified two blind spots: unrolled loops and parameters reconstruction.

2.3 Miscellaneous

In this section we list works that do not directly address the problem of cryptographic identification, but are closely related. A first line of work is automatic decryption of encrypted communication received by a binary executable. Given a known or an unknown decryption algorithm, the objective is to automatically retrieve decrypted data from the process memory. This problem deserves to be mentioned here, since it requires to detect where and when the cryptographic code is and implies at least to identify the output parameter (or conversely the input parameter for encryption). Lutz [20] was the first to address this problem in the context of malware analysis. He devised a dynamic technique based on data tainting and several heuristics including loops, integer arithmetic operations and entropy (an entropy drop may indicate a decryption). ReFormat [31] is a tool designed to automate the protocol reverse engineering of encrypted messages. It also uses data tainting to track the encrypted data in memory, but solely relies on arithmetic and bitwise instructions to discern the decryption phase. Finally Wang *et al.* [29] proposed a novel approach for automated Digital Rights Management (DRM) removal from streaming media. Their approach is driven by real-time constraints. The decryption algorithm is revealed by a randomness decrease in memory buffers manipulated by loops. According to this study randomness should be preferred over entropy to dissociate encrypted data from compressed data.

A second closely related area is binary clone detection. It is a more general problem but some of the techniques used might be of some interest for cryptographic primitive identification. Sæbjørnsen *et al.* [25] based their binary clone detection on the comparison of features vectors, which characterize a code fragment in terms of instruction mnemonics and operands. Finally, Rendezvous [19] identifies binary code fragment using statistical model relying on instruction mnemonics, Control Flow Graph (CFG) subgraph and data constants. Experiments carried out by the authors show resilience to the use of different compilers and compiler's options.

3. SOLUTION OVERVIEW

DFGs are a natural way to represent the dependencies between operations. They are also convenient to rewrite program code, i.e. to modify the code without breaking its semantics, as illustrated by their many uses in the compiler field for code optimizations [4]. For those two reasons the DFG representation appears well suited to address the problem of cryptographic algorithm identification. First it will be easy to extract specific subsets of related operations to form accurate signatures. Second, it will be possible to remove some of the variations that exist between different instances of the same algorithm, by rewriting the DFG. These variations have either been introduced in the source code or during the compilation. Ideally there should be a unique signature that matches every instance. Thus these variations should be removed leading to a canonical form that contains the signature.

In a nutshell, our identification method is a three-step process. First given as input a piece of assembly code we build the corresponding DFG. Second we normalize this DFG using rewrite rules. Third we search for subgraphs in the DFG that are isomorphic to the graph signature of a given cryptographic algorithm. If such a subgraph is found, we conclude that the assembly code implements the corresponding algorithm. Figure 1 shows the flowchart of our identification method.

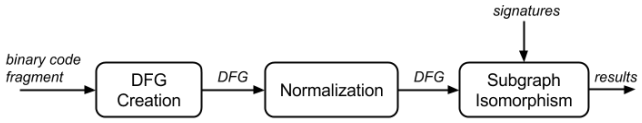

Figure 1: Simple flowchart of the DFG signature based identification technique

It is natural to wonder if CFGs could be used in a similar process. However, due to performance and security considerations (typically to resist timing attacks), the implementation of symmetric cryptographic algorithms tends to avoid conditional instructions. For example there is no conditional instruction in a standard MD5 implementation (Appendix B). Furthermore CFG may vary from one algorithm instance to another due to loop unrolling. For those reasons, control flow information does not seem relevant for symmetric cryptographic identification and we choose to solely rely on the DFG. To take full advantage of this simplification, we make the assumption of straight-line programs where every loop has been unrolled and every function call has been inlined.

To achieve acceptable performances, our method should not be applied directly to the code of a whole program. In fact the second and third steps are computationally intensive and the smaller the DFG is, the faster they will be executed. Therefore, only preselected program fragments should be submitted to the identification process. However unlike the I/O identification method (described in Section 2.2), there is no hard constraint on the fragment extraction mechanism in our case. Obviously the cryptographic code should be included in one fragment, but using large fragments does not affect the method reliability. In fact, the subgraph isomorphism algorithm is able to detect signatures even though they are surrounded by additional elements. Some of the criteria that may be used to extract suitable code fragments are discussed in Section 8.

4. DATA FLOW GRAPH CONSTRUCTION

4.1 Model of DFG

A DFG is a Directed Acyclic Graph (DAG) that represents the data dependency between a set of operations. A vertex represents either an arithmetic/logic operation or an input variable. An edge from vertex v_1 to vertex v_2 means that v_1 (or the value produced by v_1, if v_1 is an operation) is an input operand for operation v_2. Each operation produces one result and takes a non-empty unordered set of operands as input. For non commutative operations, edges may be tagged using special labels to clearly identify the role of each

operand. We also distinguish constant from non-constant input variables. Constant variables have a fixed known value derived from immediate operands of the x86 assembly code; non-constant variables have an unknown value defined either by a register or by a memory location. Memory accesses are potentially considered as both input/output variables and operations, and as such they are handled separately.

4.2 From Assembly to DFG

From the assembly code of a program fragment F, we build the corresponding DFG: $G_F = (V, E)$ (V is a set of vertices and E is a set of edges) by iterating over the instructions of F. Each instruction $i \in F$ is translated into a set of operations O_i which can be empty (if i has no effect on G_F, for instance because it is a branch instruction) or contain one to several vertices (multiple operations may be required to reproduce the behavior of complex instructions). Depending on the type of the instruction's input operands, we take the following actions:

Immediate. We add a constant input variable to G_F. This vertex holds the value of the immediate operand and is linked by an edge to O_i.

Register. We add an edge between the last definition of the register and O_i. In practice, we maintain an array that associates each register with the vertex holding the reference to its current value. This reference can be null if the register has not yet been used in F (In that case a new input vertex is added to the graph), or it can point to either an input variable (the register was read but not set in F) or an operation (the last to have written in the register).

Memory. Memory operands are accessed through special operations: `load` for memory read and `store` for memory write. These two operations take as input operand an address which computation is explicitly transcribed in G_F. We also keep track of the order in which memory accesses are made in the program fragment.

4.3 Example

We illustrate our DFG model and the translation process with an example. This example is based on a custom Even-Mansour cipher [11] with a 32-bit substitution box as the public permutation. If we note p the plaintext, k the key and S the substitution box, the ciphertext is equal to the following expression: $S(p \oplus k) \oplus k$. An x86 assembly implementation of this encryption algorithm is given in Figure 2 and the corresponding DFG is given in Figure 3. Input variables are represented inside rectangles and operations inside circles. The relative order of memory accesses is specified by an index. This example will be pursued through Section 5 and Section 6.

```
1  mov   ebx,DWORD PTR [esp+0x8]      ; load key
2  pop   ecx                          ; load plaintext
3  xor   ebx,ecx                      ; = p ^ k
4  mov   ecx,DWORD PTR [SBOX+ebx*4]   ; = S(p ^ k)
5  mov   ebx,DWORD PTR [esp+0x4]      ; load key
6  xor   ecx,ebx                      ; = S(p ^ k) ^ k
```

Figure 2: A possible x86 assembly code for the custom encryption algorithm. Given a plaintext p, an encryption key k, the custom algorithm computes $S(p \oplus k) \oplus k$ where S is a substitution box.

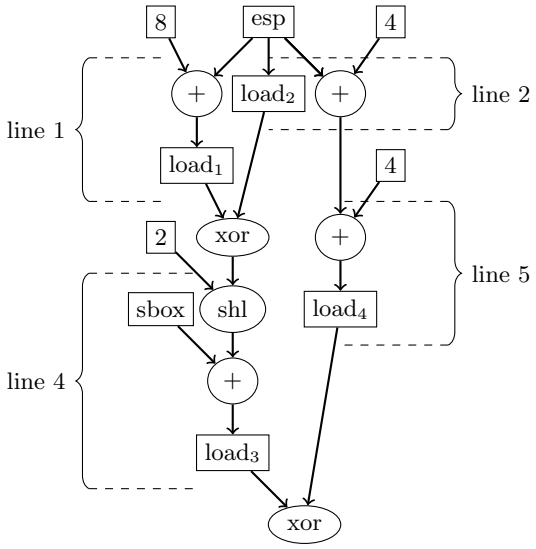

Figure 3: DFG derived from the assembly code of the custom encryption algorithm given in Figure 2. Input variables are represented inside rectangles and operations inside circles.

5. NORMALIZATION

The goal of normalization is to modify the DFG without breaking its semantics in order to maximize the chance of finding an algorithm's signatures. In other words we try to remove the variations that may have been introduced by either the developer, the optimizations of the compiler or the translation into machine code.

Once the DFG of a code fragment has been constructed, we normalize it using a set of rewrite rules. Since it is hard to understand how a particular rule can affect the others, we simply iterate the application of the rewrite rules until we reach a fixed-point, which we consider to be the canonical form of the DFG. We assume that the set of rewrite rules does not contain conflicting rules that will cause an endless loop. In practice the canonical form is reached in less than ten iterations for most cases.

We use three types of rewrite rules: normalization rules, memory simplification rules and general simplification rules.

5.1 Normalization Rules

Normalization rules are used when several instructions can be used to perform the same operation. We arbitrarily choose one as the normalized version and we create rules to convert from the others. Examples of normalization rules that we use, are given in the table below:

Original	Normalized	
$\mathtt{a} \oplus \mathtt{0xff..f}$	$\sim \mathtt{a}$	
$\mathtt{rol(a, cst_1)}$	$\mathtt{ror(a, cst_2)}$	with $\mathtt{cst_2} = \mathtt{size(a)} - \mathtt{cst_1}$
$\mathtt{a} - \mathtt{cst_1}$	$\mathtt{a} + \mathtt{cst_2}$	with $\mathtt{cst_2} = -\mathtt{cst_1}$
$\mathtt{a} \oplus \mathtt{a}$	$\mathtt{0}$	
$\mathtt{a} \times 2^{\mathtt{n}}$	$\mathtt{a} << \mathtt{n}$	

5.2 Memory Access Simplification Rules

During the generation of the DFG, every memory access has been replaced by a `load` or a `store` operation. However, many of them are caused by registers filling and spilling

and are irrelevant to the algorithm identification. Most of all, it is necessary for normalized graphs to be independent from where local variables are stored (either in registers or memory). A consequence is that normalized graphs should be free of any memory operation except for the ones corresponding to input or output variables. We devise three rules to simplify memory accesses. For each memory address, i.e. each vertex of the DFG that is used as an address operand by at least one memory operation, we compute the corresponding sequence of memory operations. Then, we traverse the sequences and perform simplifications for the following patterns:

- `store` after `store`: the first `store` has no effect, it can be removed.
- `load` after `store`: the output of the `load` is equal to the input of the `store`, the `load` can be removed.
- `load` after `load`: the two operations output the same value, they can be merged.

However, the problem of aliasing is yet to be solved. Aliasing happens when two different vertices are equal to the same address value. In such cases, the address sequence computed for both vertices will be incomplete. As a first consequence, we may miss additional simplification cases. This is not a concern however, since we can assume that these will eventually be dealt with by other simplification rules (such as common subexpression elimination). The second consequence is more problematic: simplifications performed with incomplete sequences may break semantics of the code irreversibly. This happens in the following cases:

- if an aliased `load` happens between two memory `store`.
- if an aliased `store` happens between a memory `store` and a memory `load`.
- if an aliased `store` happens between two memory `load`.

To address this problem, we can use a may-alias analysis. When a possible aliasing issue is detected we split the sequences of memory operations. The result is a set of sequences that are free of any aliasing conflict and that can be safely simplified. However, if the may-alias analysis is too inclusive, several legitimate simplification cases will be discarded. This is the main drawback of that method.

An example of memory access simplification is given in the right hand side of Figure 4. Two `load` operations are performed for the same address $(esp + 8)$, there is no memory write in-between (that is to say no possible aliasing issue), thus according to the simplification rules they can be merged.

5.3 General Simplification Rules

We wish to achieve two goals with the general simplifications rules. The first goal is to find which memory accesses are made at a same address in order to enable the memory simplifications described previously. The second goal is to optimize the DFG of non optimized code. In fact compiler optimizations are not necessarily reversible. Hence if we have to reach the same normalized representation from two versions: one well optimized and the other not, the only possibility left is to optimize code that is not well optimized. For example let us assume the following sequence of instructions and its optimized counterpart:

$$\begin{array}{l} \texttt{b} = \texttt{a} >> 16 \\ \texttt{c} = \texttt{b} \wedge \texttt{0xff} \\ \texttt{d} = \texttt{c} << 2 \end{array} \quad \longrightarrow \quad \begin{array}{l} \texttt{e} = \texttt{a} >> 14 \\ \texttt{d} = \texttt{e} \wedge \texttt{0x2fc} \end{array}$$

Let us assume that a certain compiler happens to perform this optimization. It is obvious that it will be hard to be undone: how do we guess there was a right shift operation at the end of the sequence? Its normalized representation should definitely be the optimized one. As a result, every time the original sequence is encountered it will have to be optimized.

At this point one might think that the amount of work required to match modern compilers data flow optimizations is going to be tremendous. However due to the straight line hypothesis we made, it is simpler than it might look. In fact the program fragment F can be seen as a single basic block with one entry point and one exit point. Thus the simplification rules required to catch up with maximal optimization levels only have to be applied locally to a single basic block. These rules can be divided in two main mechanisms: common subexpression elimination and constant simplification.

Common Subexpression Elimination

Common subexpression elimination is a classical compiler technique to remove redundant operations. If two operations share the same set of input operands, then they obviously produce the same output. Consequently one of them can safely be removed from the graph.

Common subexpression elimination is especially important for memory addresses. In fact in x86 code, the effective address computation is generally performed every time a memory access is made. As a consequence it is hard to detect if two memory accesses are made at the same location, since their address operand systematically belongs to different vertices. Common subexpression elimination will merge effective address computations resulting from the same set of operands (base, index, scale and displacement). As a result some memory accesses will explicitly share address vertex in the graph and thence it will be possible to perform the memory simplification rules described previously.

An example of subexpression elimination is given on the center of Figure 4. Two additions take as input *ESP* and 8. One of them was removed, leaving the graph with two memory `load` sharing the same address vertex.

Constant Simplification Rules

We can perform constant simplification in the following case:

1. If every input operands of an arithmetic/logic operation is a constant variable. The operation can be replaced by the result.
2. If an arithmetic/logic operation has an operand which is equal to either the identity element or the absorbing element of that operation (if they exist).

However it is sometimes required to rearrange sequences of associative operations to perform constant simplification. This requirement is illustrated on the left hand side of Figure 4 where two consecutive additions involving constant variables can be simplified if they are rearranged. Another kind of rearranging strategy that enables further constant simplification is distribution. It is especially important for memory address simplification due to the scale parameter of the x86 addressing mode.

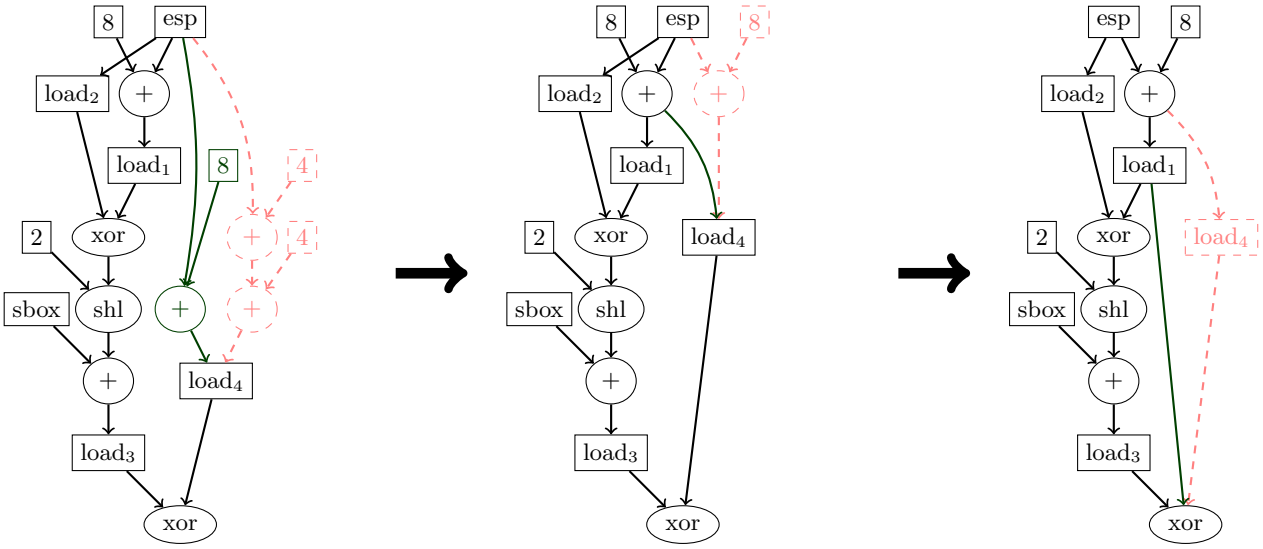

Figure 4: During the normalization process three transformation rules can be applied to the DFG of Figure 3. For each, the initial state is recalled in dashed red and the result is given in dark green. From the left to the right we successively perform: a constant simplification, a common subexpression elimination and a memory simplification.

For that purpose we design two additional rewrite rules. The first rule merges consecutive associative operations involving constant variables. The second rule distributes distributive couple of operations involving constant variables.

6. SIGNATURE

A signature is a distinctive subgraph shared between the normalized DFG of every instance of an algorithm. Signatures should be as complete and precise as possible in order to reduce the number of false positives and to reveal every feature of the algorithm that we might be interested in. Since our objective is to precisely locate the algorithm and its parameters, the signatures must cover the full length of the algorithm and must not be limited to some specific fragments. But conversely, signatures should not be too restrictive so as to match the largest range of possible implementations. Ideally the normalization process should be able to transform any instance of an algorithm into a unique representation. However, the level of analysis and understanding required is sometimes far beyond the reach of our rewrite rules. When it happens, one may favor a more generic signature that leaves the part subject to variations unspecified rather than to multiply the number of required signatures.

6.1 Signature Creation

We currently do not implement an automatic mechanism to create new signatures, and they have to be generated manually. Obviously, this requires some knowledge of the assembly language and a good understanding of the algorithm implementation. However, there is no prerequisite regarding the normalization process in theory. In fact, to be compliant with the rewrite rules, a signature can always be normalized after its creation, like any other DFG.

6.2 Macro Signature

Despite the normalization step and careful signature designs, multiple signatures are sometimes required to cover a wide range of implementations. In such cases, to reduce their number, we introduce macro signatures to model context-free graph grammar. The key idea is to append to the DFG a special vertex every time a signature is detected. It makes possible to issue higher level queries through macro signatures containing reference to signatures that have already been encountered. More formally a macro signature is a graph containing usual DFG vertices (terminal symbols) and also vertices representing other signatures (non terminal symbols). We use a bottom-up parsing algorithm relying on a subgraph isomorphism solver (discussed in Section 7). Starting from the signatures containing only terminal symbols, a special vertex is appended to the DFG every time an instance of a signature is found. These vertices are labeled according to their corresponding signature. Notice that different signatures can result in vertices of equal labels. A macro signature s_1 can be searched when every signature s_2 such that $s_1 \Rightarrow s_2$ was searched.

Macro signature are of high interest to reduce the number of signatures that need to be tested. In fact, variations that affect disjoint parts of a DFG can be searched independently. Let us assume an algorithm divided into n disjoint parts, each of them with $a_i, 1 \leq i \leq n$ different ways of being implemented. Without macro signatures, the number of signatures that have to be tested to cover the full algorithm is $\prod_{i=1}^{n} a_i$. However, using macro signatures, each part can be searched independently, covering every case with only $\sum_{i=1}^{n} a_i$ signatures. A practical example is given in Section 8.1.

Additionally, macro signatures may be used to break down large graphs into smaller ones, leading to better performances. This strategy is illustrated in Section 8.2.

6.3 Example

We continue the running example and create a signature for our toy cipher. Assuming we want to locate the parameters of our algorithm, the signature should at least contain

one vertex for each of them: the plaintext, the key and the ciphertext and at least a path linking them together. Although macro signatures were far from being mandatory in that case, we intentionally choose to use one for illustrating purpose. We first create a signature that covers the access to the substitution box (address computation plus memory read). This signature is then reused in another signature where the two `xor` operations and the parameters' vertices have been appended. This second signature is complete and should be able to accurately identify our algorithm. These signatures and the way they match the normalized DFG of Figure 4 are shown in Figure 5. A specific label $*$ is used for input variables. This label can be matched with any other label (it is mandatory for input variables since they may have produced by any operation). The two steps of the parsing process are illustrated in Figure 5. First, on the left hand side the substitution box signature is being searched. Second, on the right hand side after appending a new vertex (drawn in a rounded rectangle with the label sig_1) corresponding to the successfull match of the first signature, the complete signature is being searched.

7. SUBGRAPH ISOMORPHISM

This is the final step of our solution. Given a normalized DFG and a set of signatures, we want to know which signatures are contained in the DFG. To this end we use the subgraph isomorphism algorithm proposed by Ullmann [27]. A short description of this algorithm is given as follows.

As a reminder, a graph $G_A = \{V_A, E_A\}$ is a subgraph of $G_B = \{V_B, E_B\}$ if $V_A \subset V_B$ and $E_A \subset E_B$. A graph $G_A = \{V_A, E_A\}$ is said to be isomorphic to $G_B = \{V_B, E_B\}$ if there is a function $f : V_A \mapsto V_B$ such that: $(v, w) \in E_A \Leftrightarrow (f(v), f(w)) \in E_B$. Given a signature $S = \{V_S, E_S\}$ and a normalized DFG $D = \{V_D, E_D\}$, we want to find all subgraphs of D that are isomorphic to S. With respect to the above definitions, our problem can be reformulated as follows: we want to enumerate every function $f : V_S \to V_D$ such that:

$$(v, w) \in E_S \Rightarrow (f(v), f(w)) \in E_D \quad (1)$$

Subgraph isomorphism can be achieved using a rather simple depth-first tree-search procedure. For each vertices of the signature: $v \in V_S$, we maintain a set of possible assignment called A_v. A_v is initialized with vertices of D that have the same label than v. The algorithm works by recursively picking one element in each possible assignment sets such that condition 1 hold true. The f function is defined by: $f(v) = u$ where u is the vertex that has been picked in A_v.

Ullmann introduced an additional refinement procedure that takes advantage of the vertices that have already been picked to reduce the possibilities for future picks. Let v and w be vertices of S such that $(v, w) \in E_S$. Given $x \in A_u$, if there is no vertex $y \in A_w$ such that $(x, y) \in E_D$, then x can be removed from A_v. In fact, picking x from A_v will necessarily break condition 1. Every time an element is removed or picked from one of the possible assignment sets, we apply this new criteria trying to remove as many elements as possible from the other possible assignment sets. A pseudo code for Ullmann subgraph isomorphism algorithm is given in Algorithm 1.

Despite a high theoretical complexity (the subgraph isomorphism problem is NP complete) we were able to achieve acceptable performance using Ullmann algorithm in our context. Experimental data regarding the execution time are presented and discussed in Section 8.2.

Algorithm 1 Ullmann Subgraph Isomorphism

```
 1: function SUBGRAPH ISOMORPHISM(S, D)
 2:     ∀v ∈ V_S initialize A_v. Δ ← {A_v, v ∈ V_S}
 3:     f is undefined for every vertex of V_S
 4:     RECURSIVE SEARCH(S, D, f, Δ)
 5: end function

 6: function RECURSIVE SEARCH(S, D, f, Δ)
 7:     if ∀v ∈ V_S, f(v) is defined then
 8:         f defines a valid subgraph isomorphism ¤
 9:     else
10:         UPDATE(Δ)
11:         pick v ∈ V_S such that f(v) is undefined
12:         while A_v ≠ ∅ do
13:             pick u ∈ A_v. f(v) ← u
14:             RECURSIVE SEARCH(S, D, f, copy(Δ))
15:             remove u from A_v and set f(v) undefined
16:         end while
17:     end if
18: end function

19: function UPDATE(Δ)
20:     for all (v, w) ∈ E_S | f(v) is undefined do
21:         for all x ∈ A_v do
22:             if f is defined for w then
23:                 A_w ← {f(w)}
24:             end if
25:             if {y ∈ V_D, (x, y) ∈ E_D} ∩ A_w = ∅ then
26:                 remove x from A_v
27:                 UPDATE(Δ) return
28:             end if
29:         end for
30:     end for
31: end function
```

8. EXPERIMENTAL EVALUATION

The goal of the experimental evaluation is to demonstrate the validity of our approach. It involves two claims in particular: first that the normalization process is able to effectively remove implementation variations, and second that it is possible to create graph signatures that match every instance of an algorithm once normalized. For that purpose, we have implemented a prototype and run it on multiple test programs. As a first evaluation of our method, we thought it was more relevant to analyze the performance on synthetic samples rather than to directly confront our prototype with real life programs. Synthetic samples are convenient to thoroughly evaluate the solution in well controlled environments. Furthermore as mentioned in Section 3 our method takes preselected fragments of binary code as input. It is clear that the prototype performances depend more on the fragment's code rather than on the fragment's origin. Thus synthetic samples sound perfectly relevant for our testing purpose.

Although we do not address the problem of fragment extraction in our work, we list here few heuristics that can be used as front end filters to extract candidates. These heuristics are taken from the related works discussed in Section 2.

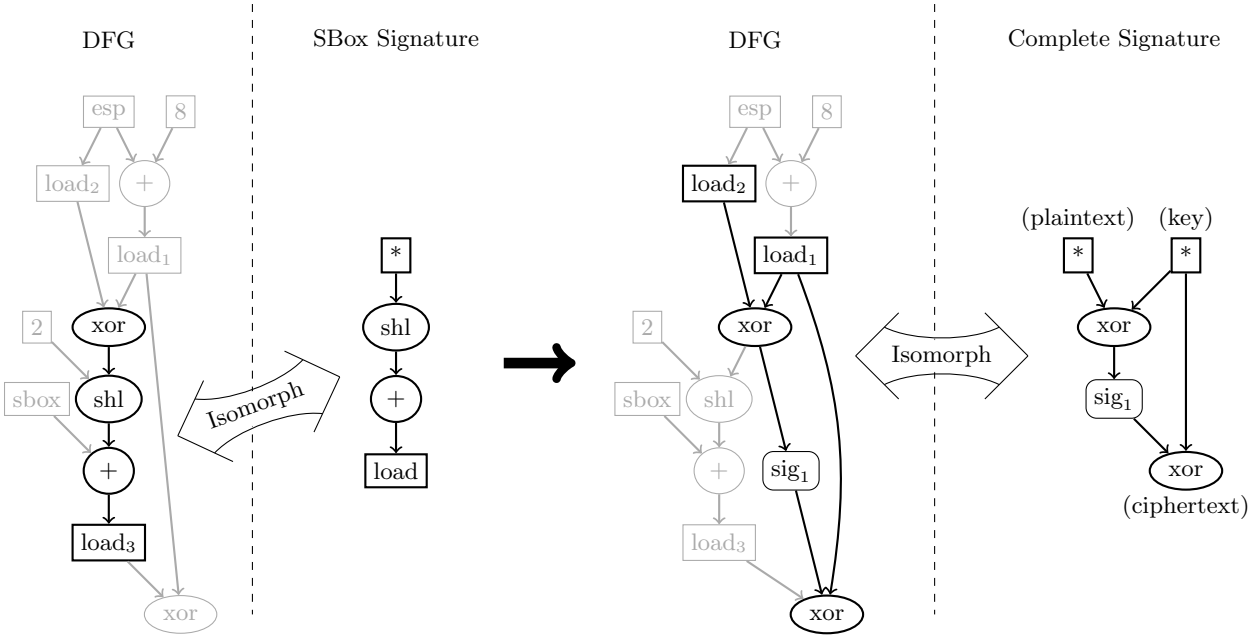

Figure 5: Signature detection: for the two steps of the parsing process we recall the DFG and the signature that is being searched. The signature match is drawn in black and the rest of the DFG in light gray.

Function. Symmetric cryptographic algorithms are usually implemented in a single function that does not call any sub function. Based on this observation we can extract every function which maps to a vertex with no direct successor in the call graph.

Constant. As previously stated constants can be used to identify cryptographic code. Code regions surrounding identified constants are good candidates for implementing cryptographic primitives.

Mnemonic. The DFG normalization may modify some instruction's mnemonic. Nevertheless most of them will remain unmodified (or at least will be changed in an easily predictable way). A first and effective step would be to filter code regions based on the mnemonics of their instructions.

From an implementation perspective the straight line code requirement is obtained using DBI. During a program execution, every executed instruction is recorded and then analyzed statically by our prototype.

In this section we present the results we obtained for three cryptographic primitives: the XTEA block cipher, the MD5 hash function and the AES block cipher. For each of them two groups of tests were performed. The first group is designed to extensively evaluate the influence of the compiler and its options on the primitive detection. Given a publicly available reference source code, we compiled it using several compilers and multiple optimization levels. The second group of tests is aimed at assessing the efficiency of our method on *real* cryptographic binary code and to show its resilience to source code variations. To do so we created synthetic samples that use well known cryptographic libraries that were used as distributed in their respective Debian package.

Finally we discuss the performance of our approach.

8.1 Results on Samples Testing

We will try to justify here why our method does not produce any false positive. First, the normalization process preserves the semantics. This is mandatory since semantics discrepancies amplified by the whole set of rewriting rules could have dramatic effects on both the false positive and the false negative ratio. Second, Ullmann algorithm for the subgraph isomorphism problem returns exact solutions and not approximations. Thence when a signature is detected, the code fragment does implement the mechanism described by the signature. Of course we do not deny that if the signature is not distinctive enough, it will appear in the DFG of other algorithms.

XTEA

The first algorithm we have tested is XTEA (a reminder on XTEA and its implementation is given in Appendix C). We used one signature for the encryption algorithm and one signature for the decryption algorithm. They are composed of approximately 500 vertices each and cover the 32 cycles of a classical implementation. We evaluated the resilience to variations in the compilation process by using the C source code given on the Wikipedia page of XTEA [1]. The results are presented in Table 1.

We use three different compilers with four optimization levels ranging from `-o0` (not optimized) to `-o3` (fully optimized). Only two levels have been used for MSVC (Microsoft Visual Studio Compiler), they correspond to the standard settings of the debug and release mode. Our prototype was able to correctly identify the algorithm in every case for both encryption and decryption.

The second testing phase involved three cryptographic libraries: Crypto++ [2], LibTomCrypt [10] and Botan [1].

[1] `http://en.wikipedia.org/w/index.php?title=XTEA&oldid=618892433`

Table 1: XTEA signature detection for different conditions of compilation.

	GCC 4.9.1 (Linux 32-bit)	Clang 3.5.0 (Linux 32-bit)	MSVC 17.00 (Windows 32-bit)
`-o0`	ok	ok	ok
`-o1`	ok	ok	-
`-o2`	ok	ok	ok
`-o3`	ok	ok	-

The identification was successful for these three libraries. The only difficulty we faced was about the key scheduling. The XTEA key scheduling is fairly simple, thus it can either be performed in an early initialization phase resulting in a round key buffer (as in LibTomCrypt) or it can be directly computed within the main encryption/decryption loop (as in Crypto++ or Botan). This difference cannot be removed by the normalization step, hence it should be dealt with at the signature level. The option we chose was to remove the key scheduling from the signature to keep a unique signature for both versions. Even though the key remains unidentified it is acceptable since at least, the round key buffer (first version) or the key scheduling final operations (second version) are identified as part of the signature input variables.

MD5

The second algorithm we have tested is MD5 (a reminder on MD5 and its implementation is given in Appendix B). For reasons that will soon become apparent we used macro signatures. Let us assume for the moment the following layout for the signatures: one signature for each different round functions and a macro signature that congregates the round signatures to form the 64 rounds of the Feistel network. For the first series of tests we took the C implementation given in the appendices of the RFC [24] and we recompiled it using different compilers and optimization levels. The results are presented in the Table 2.

Table 2: MD5 signature detection for different conditions of compilation.

	GCC 4.9.1 (Linux 32-bit)	Clang 3.5.0 (Linux 32-bit)	MSVC 17.00 (Windows 32-bit)
`-o0`	ok	ok	Partial
`-o1`	ok	ok	-
`-o2`	ok	ok	Partial
`-o3`	ok	ok	-

The identification of the final macro signature representing the Feistel network was successful except for MSVC. In the case of MSVC not all of the 64 round signatures were correctly detected for the second message chunk. Consequently the final signature, that depends on the correct identification of every round, was not detected for the second message chunk. This problem can be explained in two points that are detailed as follows.

Rotation. Every round function includes a rotation. Because there is no rotation operator in the C language, it has to be implemented using two `shift` and an `or` operator. Some compilers recognize this specific pattern and translate it to the rotation instruction of the x86 instructions set. However MSVC does not always perform this translation and sometimes keeps the expanded form. Due to the optimizations performed later by MSVC and to our own rewrite rules (particularly distributing left shifts over additions for constant operand, refer to Section 5.3) it would have been challenging to design a rewrite rule to detect and replace the expanded form by a single rotation vertex. We choose to tackle this problem at the signature level. It is a perfect example to illustrate the interest of macro signatures. Each round, after the normalization process, can still be implemented in two different ways: either with the rotation operation or with the expanded form. The overall number of combinations for the 64 rounds is 2^{64}. It is obviously impossible to test 2^{64} signatures. Instead, with macro signatures we only have to double the number of round signatures (from 4 to 8).

Constant state. For the first message chunk the state is initialized with constant values. Because of the rewrite rules designed to promote numeric simplification (Section 5.3) operations involving the initial state are merged with surrounding operations. It happens in the case of MSVC for the expanded form of the rotation at the beginning of the second chunk, scrambling the expected signature pattern. This issue is still being investigated and we believe that a more complete common subexpression elimination algorithm (capable of removing common subexpression distributed over several vertices and not only two) might be able to solve this issue.

We based our second testing phase on the following libraries: Crypto++, LibTomCrypt and OpenSSL [3]. The result were successful and we faced none of the previously described difficulties since rotations were always implemented by the x86 instruction and the initial state was never perceived as constant due to the API design which is such that the initialization is performed in another function outside the code fragment.

AES

The last algorithm we tested is the table implementation of AES (a reminder on AES and its implementation is given in Appendix A). We built three signatures, one for each key size. The signatures are the same for encryption and decryption, only the look up tables change. We choose the source code provided by Gladman on his website for the first series of test. The results are presented in the Table 3.

Table 3: AES signature detection for different conditions of compilation.

	GCC 4.9.1 (Linux 32-bit)	Clang 3.5.0 (Linux 32-bit)	MSVC 17.00 (Windows 32-bit)
`-o0`	ok	ok	ok
`-o1`	ok	ok	-
`-o2`	ok	ok	ok
`-o3`	ok	ok	-

Our prototype was able to successfully detect the signature in every case. For the second testing phase involving well known cryptographic libraries we used Crypto++, LibTomCrypt and Botan. We were only able to test the decryption for the Crypto++ library, since the encryption uses MMX and SSE2 instructions which were not supported by our DFG creation routine at the time when the tests

were done. Aside from Crypto++'s encryption algorithm the identification was successful in every cases.

8.2 Performance

The subgraph isomorphism problem is a well known NP-complete problem, but it can be solved efficiently in the majority of the cases encountered in our context. Table 4 presents some of the execution times we obtained with our prototype for the subgraph isomorphism step on a common laptop computer. Each column corresponds to a code fragment (taken from the LibTomCrypt library) and each row corresponds to a signature.

Table 4: Execution times for the signature matching step, on a common laptop computer.

	TEA 753 vertices	MD5 904 vertices	AES_{256} 1687 vertices
TEA 1 cycle (16 vertices)	5ms	< 1ms	< 1ms
TEA 32 cycles (450 vertices)	102ms	< 1ms	< 1ms
MD5 1 round (10 vertices)	< 1ms	9ms	< 1ms
MD5 64 rounds (372 vertices)	< 1ms	26ms	< 1ms
AES 1 round (76 vertices)	< 1ms	< 1ms	442ms
AES 14 rounds (1012 vertices)	< 1ms	< 1ms	2.22s

When the signature was not detected, the cell is colored in gray. We first notice that our prototype quickly (< 1ms) eludes cases where the code fragment does not match the signature. It is a reassuring result, since it will be the most common scenario while testing large databases of signatures with weak fragment selection heuristics. Second, we observe that large signatures (the ones covering several rounds) take significantly more time than smaller ones (covering just one round). Based on that observation, it may be tempting to use macro signatures to reduce the signature's size and thus to achieve better performances. For instance if we split the AES 14-round signature into two signatures: the first one covering one round (approximatively 70 vertices) and the second one linking the 14 rounds together (approximatively 130 vertices), the execution time is reduced: 1.3s instead of 2.3s. However the number of vertices is not the only parameter to influence the execution time. Sometimes larger signatures imply stronger structural constrains between their vertices and the subgraph isomorphism algorithm will be able converge more rapidly towards the solution.

Regarding the normalization step, every rewrite rule is linear with the number of vertices, except for the common subexpression rule which is quadratic (at least for its naive implementation). However as previously stated, the rewrite rules are iteratively applied until a fixed-point is reached. Thus, the execution time also depends on the distance between the original DFG and its normalized version. In practice the execution time of the normalization does not exceed the execution time of the signature matching.

To conclude, despite the theoretical complexity of the underlying algorithm, the execution times we observed on synthetic samples seem acceptable.

9. CONCLUSION

In this paper we presented a new method for symmetric cryptographic algorithms identification in binary programs. To this end we introduced a DFG representation. This representation was first used in a normalization step designed to increase the detection capability, by erasing the peculiarities of each instance of an algorithm. Then, the normalized DFG was compared to the signatures of database using a subgraph isomorphism algorithm. Signatures cover the full length of the algorithm and not only an isolated group of distinctive instructions. Thus our approach does not produce any false positive and the input and output parameters of the cryptographic primitive are automatically identified as part of the signature boundary. We built a prototype and tested it against several synthetic samples covering three cryptographic algorithms: XTEA, MD5 and AES. We showed detailed results proving that our approach was resistant to a large range of compilation conditions. We also provided results on well known cryptographic libraries.

As future work, we plan to cover block cipher modes of operation by leveraging the concept of macro-signature we have presented in this paper and we envisage to extend our approach to automatically identify public key cryptographic algorithms. In order to make our contribution more practical for security auditors to support more algorithms, we aim at automatically generating signatures from reference implementation. Finally we left for future researches the delicate problem of dealing with obfuscated code.

10. ACKNOWLEDGMENTS

The authors are grateful to Pierre Karpman for his help and his useful feedbacks.

11. REFERENCES

[1] Botan. `http://botan.randombit.net/`.
[2] Crypto++. `http://www.cryptopp.com/`.
[3] Openssl. `https://www.openssl.org/`.
[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.
[5] G. Balakrishnan and T. W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.
[6] A. Biryukov. The design of a stream cipher lex. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 67–75. Springer, 2006.
[7] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: cryptographic function identification in obfuscated binary programs. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 169–182. ACM, 2012.
[8] C. Clavier and K. Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.
[9] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[10] T. S. Denis. Libtomcrypt. http://libtom.org/.

[11] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. *J. Cryptology*, 10(3):151–162, 1997.

[12] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[13] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In R. Sommer, D. Balzarotti, and G. Maier, editors, *RAID*, volume 6961 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2011.

[14] I. Guilfanov. Findcrypt2. http://www.hexblog.com/?p=28, February 2006.

[15] S. Halevi, W. E. Hall, and C. S. Jutla. The hash function "fugue". *IACR Cryptology ePrint Archive*, 2014:423, 2014.

[16] M. Hamburg. Accelerating aes with vector permute instructions. In Clavier and Gaj [8], pages 18–32.

[17] E. Käsper and P. Schwabe. Faster and timing-attack resistant aes-gcm. In Clavier and Gaj [8], pages 1–17.

[18] J. Kelsey, B. Schneier, and D. Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and TEA. In Y. Han, T. Okamoto, and S. Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11-14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1997.

[19] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: a search engine for binary code. In T. Zimmermann, M. D. Penta, and S. Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 329–338. IEEE / ACM, 2013.

[20] N. Lutz. Towards revealing attacker's intent by automatically decrypting network traffic. Master's thesis, ETH Zurich, July 2008.

[21] J. Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, 2001.

[22] M. Matsui and J. Nakajima. On the power of bitslice implementation on intel core2 processor. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2007.

[23] P. Q. Nguyen. Can we trust cryptographic software? cryptographic flaws in GNU privacy guard v1.2.3. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 555–570. Springer, 2004.

[24] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[25] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su. Detecting code clones in binary executables. In G. Rothermel and L. K. Dillon, editors, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 117–128. ACM, 2009.

[26] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[27] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[28] S. Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In L. R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

[29] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Steal this movie: Automatically bypassing DRM protection in streaming media services. In S. T. King, editor, *USENIX Security*, pages 687–702. USENIX Association, 2013.

[30] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013.

[31] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2009.

[32] D. J. Wheeler and R. M. Needham. Tea, a tiny encryption algorithm. In B. Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1994.

[33] T. Xie, F. Liu, and D. Feng. Fast collision attack on MD5. *IACR Cryptology ePrint Archive*, 2013:170, 2013.

[34] R. Zhao, D. Gu, J. Li, and R. Yu. Detection and analysis of cryptographic data inside software. In X. Lai, J. Zhou, and H. Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2011.

APPENDIX

A. AES IMPLEMENTATION REMINDER

The Advanced Encryption Standard (AES) [9] is a Substitution Permutation Network (SPN) that can be instantiated using three different key bit-lengths: 128, 192, and 256. The 128-bit plaintext initializes the internal state viewed as a 4×4 matrix of bytes seen as elements of the finite field $GF(2^8)$, which is defined via the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ over $GF(2)$. Depending on the version of the AES, N_r rounds are applied to that state: $N_r = 10$ for AES-128, $N_r = 12$ for AES-192 and $N_r = 14$ for AES-256. Each of the N_r AES round applies four operations to the state matrix (except the last one where the *MixColumns* is omitted):

- *AddRoundKey*: adds a 128-bit subkey to the state.
- *SubBytes*: applies the same 8-bit to 8-bit invertible S-Box S 16 times in parallel on each byte of the state.
- *ShiftRows*: shifts the i-th row left by i positions.
- *MixColumns*: replaces each of the four column C of the state by $M \times C$ where M is a constant 4×4 maximum distance separable matrix over $GF(2^8)$.

After the N_r-th round has been applied, a final subkey is added to the internal state to produce the ciphertext. A key expansion algorithm is used to produce the $N_r + 1$ sub keys required for all AES variants.

From an implementation perspective, the *ShiftRows* and the *MixColumns* can be combined with the *SubBytes* resulting in 4 lookup tables of 1 kilobyte each. We introduce the following notations: A_i is the state at round i, divided in four 32-bits word and T_i $(0 \leq i \leq 3)$ is a function that given a 32-bit word, extracts the i^{th} most significant byte and returns the associated 32-bit word in the i^{th} lookup table. For each full round $(1 \leq i \leq N_r - 1)$, the combination of the *ShiftRows*, *MixColumns* and *SubBytes* can be implemented using the following pseudo code (we have omitted the *AddRoundKey* for brevity):

$$
\begin{aligned}
A_{i+1}[0] &= T_0(A_i[0]) \oplus T_1(A_i[1]) \oplus T_2(A_i[2]) \oplus T_3(A_i[3]) \\
A_{i+1}[1] &= T_0(A_i[1]) \oplus T_1(A_i[2]) \oplus T_2(A_i[3]) \oplus T_3(A_i[0]) \\
A_{i+1}[2] &= T_0(A_i[2]) \oplus T_1(A_i[3]) \oplus T_2(A_i[0]) \oplus T_3(A_i[1]) \\
A_{i+1}[3] &= T_0(A_i[3]) \oplus T_1(A_i[0]) \oplus T_2(A_i[1]) \oplus T_3(A_i[2])
\end{aligned}
$$

This implementation is the most widespread and it is usually referred to as the tables implementation. However, it is not the only way to efficiently implement the AES. Matsui *et al.* [22] and Käsper *et al.* [17] proposed two bitsliced implementations. In bitsliced modes, several blocks are processed in parallel taking advantage of the SMID architecture. Nevertheless bitsliced can only be used in parallel modes of operation (such as the counter mode for instance). Hamburg [16] demonstrated that it is feasible to implement a single block AES encryption with vector permute instructions. Finally recent CPUs have dedicated AES instructions to reach the best performances and the highest security levels. These alternative implementations have been mentioned here for completeness. This work only covers the tables implementation.

B. MD5 IMPLEMENTATION REMINDER

MD5 [24] is a cryptographic hash function that given a message of any size produces a 128-bit hash. The message is divided into 512-bit chunks with a padding being applied to the last chunk. The MD5 algorithm is based on a four-branch Feistel network that operates on a 128-bit state. It is composed of 64 rounds. We use the following notations: f_i is the round function of round i, $\{k_i, 1 \leq i \leq 64\}$ is a set of specific constants, M is an input message chunk and $(A_{1,i}, A_{2,i}, A_{3,i}, A_{4,i})$ is the 128-bit state at the beginning of round i, divided into four words of 32 bits. At each round $(1 \leq i \leq 64)$ the state is updated according to the following equations:

$$
\begin{aligned}
A_{1,i+1} &= A_{4,i} \\
A_{2,i+1} &= f_i(A_{1,i}, A_{2,i}, A_{3,i}, A_{4,i}, M, k_i) \\
A_{3,i+1} &= A_{2,i} \\
A_{4,i+1} &= A_{3,i}
\end{aligned}
$$

The round function f_i contains a boolean function that changes every 16 rounds (4 distinct boolean functions are used for the 64 rounds). Apart from this boolean function, the round function remains the same for every round. Usually the 64 rounds are directly unrolled in the source code. This is for instance the case of the C source code given in the appendix of the RFC).

For the first two series of rounds (round 1 to 16 and round 17 to 32) two different formulas might be used to compute the boolean functions. A different formula that the one provided in the RFC might be used for improved efficiency (it can be implemented using fewer bitwise instructions).

C. TEA & XTEA IMPLEMENTATION REMINDER

Tiny Encryption Algorithm (TEA) [32] is a 64-bit block cipher with a 128-bit key. It is based on a two-branch Feistel network that operates on a 64-bit state. Rounds are usually regrouped in pairs forming cycles. The recommended number of cycles is 32. However TEA suffers from related-key attacks [18]. To solve this weakness Needham and Wheeler proposed an extended version of TEA named XTEA. XTEA has a different key scheduling and a different round function.

Both TEA and XTEA have been designed as small C programs performing simple operations on 32-bit words. The only implementation variation we are aware of concerns the key scheduling. Since the key scheduling is extremely simple some implementations do not compute the round keys separately but they do it directly in each round.