Assisted Identification of Mode of Operation in Binary Code with Dynamic Data Flow Slicing

Pierre Lestringant[1,2], Frédéric Guihéry[1], and Pierre-Alain Fouque[2,3]

[1] AMOSSYS, R&D Security Lab, Rennes, France
[2] Université de Rennes 1, Rennes, France
[3] Institut Universitaire de France, Paris, France

Abstract. Verification of software security properties, when conducted at the binary code level, is a difficult and cumbersome task. This paper is focused on the reverse engineering task that needs to be performed prior to any thorough analysis. A previous line of work has been dedicated to the identification of cryptographic primitives. Relying on the techniques that have been proposed, we devise a semi-automated solution to identify modes of operation. Our solution produces a concise representation of the data transfers occurring within a cryptographic scheme. Inspired by program slicing techniques, we extract from a dynamic data flow a slice defined as the smallest subgraph that is distance preserving for the set of cryptographic parameters. We apply our solution to several modes of operation including CBC, CTR, HMAC and OCB. For each of them, we successfully obtain a complete and readable representation. Moreover, we show with an example that our solution can be applied on non standard schemes to quickly discover security flaw.

Keywords: Binary Analysis, Reverse Engineering, Cryptography

1 Introduction

1.1 Problem Statement

Modes of operation are critical from a security perspective, since they have to guarantee the confidentiality, the integrity and the authenticity of sensitive data. However, they are subtle to securely devise and implement and they are subject to many security vulnerabilities. For instance, Katz and Schneier described an attack on OpenPGP which was applicable to many other e-mail encryption protocols [18]. Using a *chosen-ciphertext attack* on the Cipher Feedback (CFB) mode of operation, they were able to decrypt any message without recovering the secret key. Such attacks have been implemented in practice [17]. It is also well-known that the padding used in modes of operation is highly sensitive. Bad paddings have led to devastating attacks on many IETF standards [36] by Vaudenay. A practical attack has been successfully implemented using timing information [11]. Later, Paterson described many such attacks by carefully studying the interplay between modes of operation and various security protocols against TLS [29, 2, 30], against IPsec [14] and against SSH [1, 31].

Therefore, to ensure that the security properties provided by modes of operation are truly effective, security experts have to analyze their design and their implementations. When the source code is not available, this analysis needs to be conducted at the binary level. For instance, in black box security audits, security experts are limited to publicly

available information about the target. The objective of such audits is to simulate real-world scenarios. Unfortunately, even with a good understanding of the machine language, binary code analysis is still a difficult and time consuming task mostly due to the lack of high level structure. It would be highly beneficial for security analysts if some parts of the analysis could be automated. In particular, before digging into the details of padding verification or before looking for possible side channels, analysts have to identify the mode of operation and to locate its main components. In this paper we propose a solution to facilitate this first step.

1.2 Related Work

To the best of our knowledge the only previous work to address the problem of finding modes of operation in binary code, is CipherXRay [21]. CipherXRay is based on the avalanche effect of cryptographic functions. It identifies memory buffers that are highly dependent on one another. Specific dependencies patterns are proposed to distinguish some modes of operation. The problem of identifying modes of operation can be related to wider research fields such as generic algorithm identification and more generally binary analysis. Algorithm identification has been studied in the past few years for various reasons ranging from intellectual property protection to malware analysis and to vulnerability discovery. Identification techniques can be classified according to which code abstraction(s) they use to represent and compare binary code. The main abstractions are: bytes value [16], instruction mnemonics [33], Control Flow Graph (CFG) [9], program dependence graphs [27] and observations of runtime behavior [4]. One of the most recent result in that domain is Rendezvous [19] that relies on several abstractions: data constant, instruction mnemonics and CFG subgraphs.

In the case of modes of operation, it seems interesting to devise a specific solution. In fact, symmetric cryptographic algorithms share common characteristics. For instance, their implementations try to avoid conditional statements as much as possible due to performance and security considerations (typically to resist timing attacks). By taking them into account, a dedicated identification method will have a better efficiency and produce more relevant results. A previous line of work, dealing with primitive identification, provides a good starting point. The main primitive identification techniques are presented along with their advantages and drawbacks in Section 4.

1.3 Solution Overview

We choose to rely on the Data Flow Graph (DFG) to identify modes of operation. Modes of operation specify how cryptographic primitives are applied on data to achieve security properties. Thus, the data dependencies between the cryptographic primitives and, more generally, their organization in the data flow, are essential to identify modes of operation. We present our data flow model and how it can be obtained from a program execution in Section 3.

A classical approach would be to search for distinctive data flow patterns using automated pattern matching techniques [12]. However, this approach lacks flexibility and robustness. It is ineffective against modes of operation that have been modified or that have never been encountered before. Besides, these techniques often produce fully processed results that may be hard to seize by human analysts if they want to continue the analysis manually. Instead of using signatures to identify modes of operation, we chose

to produce a synthetic representation of the data transfers occurring between the cryptographic primitives. The interpretation of the synthetic representation is left to the human analyst. This solution seems ideal to bridge the gap between automated processing and manual analysis. Furthermore, human interpretation is much more flexible than any automated pattern matching techniques. This synthetic representation, called a slice, must contain enough information to accurately identify modes of operation and, at the same time, must be easily readable by a human analyst. A slice is defined in Section 5 as the smallest subgraph of the DFG that is distance preserving for the set of cryptographic parameters. A practical heuristic to extract a slice from a DFG is described in Section 5. Experimental results are presented in Section 6. Finally, three use cases are detailed in Section 7: the first one is about OCB an authenticated encryption mode, the second one deals with an uncommon use of a cryptographic primitive as part of an IV-replacement attack and the third one is about and instant messaging application that uses a custom encryption scheme. In summary, this paper makes the following contributions:

- We propose to facilitate the analysis of modes of operation by computing a representation that summarizes the data dependencies between the cryptographic primitives.
- We give a formal definition for this representation and we propose a practical algorithm to compute it. We discuss why this definition is a good tradeoff between completeness and readability.
- We present experimental results obtained for several modes of operation including CBC, CTR, HMAC and OCB on well-known cryptographic libraries.

2 Background on Modes of Operation

In this section, we review the main modes of operations that are used in this paper: CBC, IGE, CTR, HMAC and OCB. In cryptography, modes of operation rely on primitives of fixed input-length to provide security properties such as confidentiality, integrity and authentication for arbitrarily large messages.

CBC. The Cipher Block Chaining (CBC) mode of operation is one of the most used chaining mode. It is used to encrypt large messages using a block cipher. The idea consists in randomizing the input of the block cipher using a random value. The first block is randomized using an initialization vector, while block $M[i]$ is randomized using block $C[i-1]$. To encrypt a message M, it is split into blocks $M[i]$, the size of which is equal to the input length of the block cipher. If the length of the message is not a multiple of the block size, the message is padded. The most used padding scheme simply consists in adding a bit `1` and as many bits set to `0` as needed. We choose $C[0] := IV$ uniformly and iterately compute $C[i] := E_k(M[i] \oplus C[i-1])$. This makes the scheme non-parallelizable, but it has the advantage of being self-synchronizing: an error on one block infect only two blocks. The decryption is parallelizable and $M[i] := C[i-1] \oplus D_k(C[i])$.

IGE. The Infinite Garble Extension (IGE) mode is used to encrypt large messages using a block cipher. It is practically never used. It has the property to indefinitely propagate forward errors. The message is split in the same way as in the CBC mode and $C[i] = E_k(M[i] \oplus C[i-1]) \oplus M[i-1]$. The decrypted message is given by: $M[i] = D_k(M[i-1] \oplus C[i]) \oplus C[i-1]$.

CTR. The counter mode of operation consists in viewing the block-cipher as a pseudo-random permutation, meaning that the output bitstrings are indistinguishable from uniform bitstring up to the birthday paradox. Consequently, the encryption is a one-time pad with the output bitstring. To encrypt a message M, we first split it in the same way as in the CBC mode and $C[0] = ctr$ uniformly chosen and $C[i] = M[i] \oplus E_k(ctr + i)$. To decrypt, we just output $M[i] := C[i] \oplus E_k(C[0] + i)$ [6]. Thus, the decryption is exactly the same as the encryption. Moreover this mode is parallelizable.

OCB. The Offset Codebook Mode (OCB) has been defined by Rogaway [32]. It is an authenticated encryption mode of operation. It ensures confidentiality and authentication through a single pass over the message. The idea consists in XORing a random mask $Z[i]$ over the plaintext $M[i]$ and on the ciphertext: $C[i] := E_k(M[i] \oplus Z[i]) \oplus Z[i]$. The mask evolves between each call based on a linear relation $Z[i] := \gamma_i \cdot L \oplus R$, where $L := E_k(0^n)$ and $R := E_k(N \oplus L)$ and N is a random nonce and the multiplication is performed in some finite field of 2^{128} elements. There is a much more efficient way of computing $Z[i]$ from $Z[i-1]$ but we do not need such details. The final block is the encryption of the checksum, that is the XOR of all plaintext blocks $Checksum := \oplus_{i=1}^{m} M[i]$: $T := E_k(Checksum \oplus Z[m])$. Further in the text we describe a more precise version since there is some technicalities in order to make a ciphertext stealing mode, so that the output length of the ciphertext part is as long as the plaintext.

HMAC. It was defined by Bellare, Canetti and Krawczyk in 1995 and it has been extensively used in many RFCs [5] since. The Merkle-Damgård construction allows to hash arbitrary length messages using a fixed input-length function, called the compression function. The Merkle-Dangård mode of operation has some weaknesses to built a Message Authentication Code (MAC) from such construction. The idea to avoid length extension attack consists in hashing the output with another key and it has also been described as the envelop method: $H(k_1 \| M \| k_2)$. HMAC can be defined for a hashing function H such as MD5 or SHA-1 and simply outputs: $\text{HMAC}_k(M) = H(k \oplus \texttt{opad} \| H(k \oplus \texttt{ipad} \| M))$. We will use $\|$ in order to denote the concatenation of two bitstrings.

3 Data Flow

In this section we describe our data flow model and explain how it can be computed. As mentioned in the introduction, symmetric cryptographic algorithms try to avoid conditional statements as much as possible. Apart from the number of iterations over the message blocks, we do not expect the control flow to change significantly from one execution to one another. To take advantage of this observation we assume that the code to be analyzed is a sequence of instructions that is executed from the first to the last. This hypothesis greatly simplifies the data flow computation. Straight line code can be easily obtained in practice by recording a particular execution.

3.1 Data Flow Model

The data flow is represented by a directed graph. A vertex corresponds to an operation, and an edge to a data dependency between two operations. An operation depends on its operand(s). An input variable or a constant has no dependency. A memory access does

not depend on its address but only on the value it reads or writes. Let us consider the following x86 assembly code snippet:

```
mov eax, [ebp + 8]
mov ebx, [ebp + 16]
```

Taking load-address dependencies into account results in *eax* and *ebx* being connected through *ebp*. But as far as we know, *eax* and *ebx* may be perfectly independent (despite the fact that they are stored side by side). In the end, there is a risk that everything becomes interconnected through the stack pointer (at least when arguments are passed on the stack). Thus, we discard this type of dependency. In our model, a memory read depends only on the last value that was written at its address (if there is any, otherwise it is considered as an input value). This is essential to track values as they are written and read from memory. However, to build these dependencies, one has to find which memory accesses are performed at the same address. This issue is discussed in Section 3.2. In our data flow model we do not consider implicit dependencies. An example of implicit dependency is illustrated in the following code line:

```
for (y = 0; y < x; y++);
```

The final value of y is equal to the value of x, yet there is no direct assignment from x to y. This is an implicit dependency. As explained in the introduction there should be almost no conditional statements on cryptographic data. Thus, for simplicity we ignore implicit dependencies. Finally, it goes without saying that if the result of an operation is constant it will not depend on its operands. A typical example in x86 code is:

```
xor eax, eax
```

3.2 Concrete Memory Addresses

To obtain correct load-value dependencies we must be able to compare the address of memory accesses. It can be done either statically or dynamically.

Static Approach. Given two addresses, the goal is to over-approximate their difference. That is to say, to find a set that contains all the possible values that their difference could take. If this set is equal to the zero singleton, the two addresses are equal. If it does not contain the zero value, they are different. Otherwise, it is impossible to conclude. Thus, it is important to find the smallest over-approximation possible. One of the most advanced techniques for over-approximating memory addresses in binary code is Value Set Analysis (VSA) [3]. In our case, due to the straight line hypothesis, this technique can be greatly simplified.

One important design principle was to limit the analysis to a code window. In fact, we already know, where modes of operation are located in the program (code regions surrounding the cryptographic primitives call sites). Besides, applying analysis (such as VSA) to the whole program will dramatically increase the complexity of our solution without providing any guarantee on the information we will retrieve from it. However, lack of context information greatly reduces the efficiency of static address over-approximations. In fact, modes of operation manipulate several data buffers (at least plaintext, ciphertext,

key and nonce), the address of which is usually defined outside of the analysis window. Hence, whatever method is used, no good over-approximation can be computed for these addresses. Since these buffers are accessed for mixed reads and writes, aliasing issues arise. For instance, because we cannot decide if the address of the ciphertext buffer is different from the address of the key, any write access to the ciphertext buffer might also overwrite the key. It is unclear what should be the dependencies of the next read access to the key. A first possibility will be to consider that it does not have dependency. But in that case, the DFG will not reflect that, for instance, two consecutive cipher executions use the same key. A second possibility will be to consider both dependencies: one to ciphertext and one to previous key value. However, the ciphertext dependency is highly improbable and taking it into account can be misleading while interpreting the DFG.

Dynamic Approach. In order to be context sensitive without needing to analyze the whole program, we use concrete memory addresses. The resulting DFG reflects the particular execution, where the concrete memory addresses were recorded, and not necessarily the generic behavior of the program. However, the generality loss is not a big concern since we do not expect many addresses to be dependent on input values. A typical example of an address that depends on an input value is a substitution box. But implementations of modes of operation should be free of any substitution box access. Here again, the argument is that any complex transformation occurring inside the code of the mode can be seen as a distinct cryptographic primitive and be dealt with separately.

3.3 Filtering False Dependencies

It is not because two vertices are connected by path in the DFG, that one necessarily depends on the other. In the example of Figure 1, there is a path from A to C, and yet the value of C is independent of the value of A. The combination of several instructions along a path can nullify the influence of a variable. To measure at the bit granularity the influence of a vertex $v \in V$ on the rest of the graph we use a mask $M_v : V \mapsto \{0,1\}^n$ (V denotes the set of vertices). If the i^{th} bit of $M_v(u)$ is a 0, it means that the i^{th} bit of u is not influenced by v. Conversely, if is is a 1, it means that the i^{th} bit of u may be influenced by v. To compute M_v, we start with $M_v(v) = 111...1$ and we propagate the value along the edges. For each new vertices, we update the mask value depending on the nature of its operation and the over-approximation that can be obtained for its operands. In Figure 1, M_A is given in red. $M_A(C) = 0$, thus, the path from A to C should not be considered as a true dependency.

4 Identification of the Primitives and the Parameters

As a preliminary condition, the parameters and the code of the cryptographic primitives need to be identified and located inside the DFG. With this last requirement, concerning the code of the primitive, our goal is to be able to dissociate the data flow of the primitive from the external data flow of the mode of operation. Since we are only interested in the data connections happening at the mode level, we must be able to exclude the internal data flow of the primitive from our analysis.

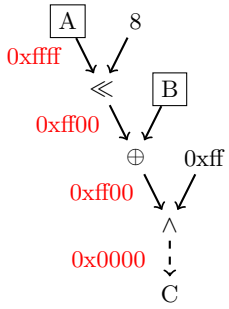


Fig. 1. In this DFG, C depends on B but not on A. M_A is given in red.

4.1 Existing Techniques

Cryptographic primitive identification has already been studied and practical solutions have been proposed. A first solution, described in [15] and [10], is based on the unique relationship that exists between the input and the output values of a cryptographic algorithm. If the data manipulated by a program fits that relationship, then we have not only identified the algorithm but also its parameters. However this solution suffers from a high combinatorial complexity. In fact, no good solution has been proposed to aggregate registers and memory accesses together in order to obtain parameters than can be test against standard implementations. CipherXRay (already mentioned in the introduction) is a second solution. It takes advantage of the avalanche effect of cryptographic functions. According to this effect, each byte of the input is expected to influence all the bytes of the output. CipherXRay searches for couples of memory buffers (continuous memory location accessed within a code fragment) that are subject to the avalanche effect. A third solution, presented in [20], is relying on DFG to build signatures for cryptographic algorithms. The DFG is first normalized using code rewrite mechanisms and then compared to the signatures of a database using a subgraph isomorphism algorithm. Signatures are a distinctive subgraphs. Parameter of cryptographic primitives are automatically identified as part of the signature boundary.

4.2 Selected Technique: DFG Signatures

We choose to rely on DFG signatures to retrieve cryptographic code and cryptographic parameters. The DFG model used in our method for mode identification is similar to the one that is used for primitive identification. Thus, it will only have to be created once for both methods. Moreover, this method has proven to be fast (execution time does not exceed a couple of seconds), efficient for non obfuscated programs and it does not require heavy instrumentation. And most of all, since it is based on DFG isomorphism, it tells very precisely which vertices and edges are part of the primitive and which ones are part of the mode. Despite the fact that primitive detection is not directly addressed by our work, it is critical for the success of our method. In Appendix A, we describe two problems that may greatly reduce the usability of the results for mode identification: parameter interdependence and parasitic detection.

5 Slicing

As explained in the introduction, to make it possible for a human analyst to interpret the data flow easily, it needs to be simplified. To this end, we propose to extract parts of the data flow that are connected to the cryptographic parameters. Described as such, this step can be seen as a program slicing process. As in program slicing, our goal is to extract parts of the program that are affected by or have an effect on points of interest (which are, in our case, the cryptographic parameters). But unlike the usual definitions of program slicing [35], we do not impose the slice to maintain semantics of the original program with respect to the points of interest. In fact, we favor readability over semantic equivalence. Thus, not every part of the data flow that is connected with the cryptographic parameters, is transcribed in the extracted graph. Because of the proximity to the program slicing domain, we borrow the terminology and call the extracted graph a slice. This section is structured as follow: first we give a formal definition of a slice; then we justify why this definition is a good compromise between completeness and readability; finally we describe a practical algorithm to compute an approximated slice.

5.1 Problem Formalization

Given a DFG $D = (V_D, E_D)$ and a set of cryptographic parameters $P \subset V_D$, a slice $S = (V_S, E_S)$ is the smallest subgraph of D such that $P \subset V_S$ and: $\forall (u, v) \in P^2, dst_D(u, v) = dst_S(u, v)$ (where dst_D and dst_S denotes respectively the distance in D and S). We define the distance between two vertices as the number of edges on the shortest *undirected* path.

5.2 Completeness-Readability Tradeoff

Completeness. A slice is said to be *complete* if it contains enough information to identify the mode of operation. The completeness is due to the distance preserving property. If two parameters are connected in the DFG, then they will also be connected in the slice. A first naive approach would be to consider a less generic definition where the slice is made only of predefined connections (instead of generic undirected paths) between subsets of cryptographic parameters. For instance, based on the CBC mode, it could be tempting to only extract the smallest directed path from an output parameter to an input parameter (chaining between two executions of the block cipher) and the lowest common ancestor between two input parameters (same key for two executions of the block cipher). However, there is a risk for this list of predefined connections to be incomplete and to become more and more complex as new types of connections are added. For instance, let us consider the simple construct to make a block cipher *tweakable*: $E_k(M \oplus h(T)) \oplus h(T)$ described in [26]. Part of the DFG for this construct is given in Figure 2. None of the connections previously mentioned for the CBC mode can describe the path between the input and the output of the block cipher in that case. To obtain complete slices without a priori knowledge of the types of connections that may be encountered, we consider undirected paths between every pair of parameters.

So far we have justified why the proposed definition is necessary to obtain complete slices with a large variety of modes of operation. Unfortunately, due to the minimality property, this definition does not guarantee the slice to always be complete. For instance, if one is interested in a particular path between two parameters, only the smallest is guaranteed to be reported. In the next section, this issue is illustrated by an example

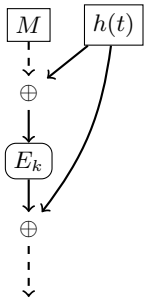


Fig. 2. DFG of a possible construct to obtained a tweakable block cipher from a block cipher. The connection between the input and the output is neither a directed path nor a lowest common ancestor

and a possible workaround is mentioned. In practice, as showed in Section 6 and 7, this definition has given good results.

Readability. A slice is said to be *readable*, if it does not contain significantly more information than what is strictly required to identify the mode of operation. The readability is ensured by the minimality property. It guarantees that the slice is free of irrelevant elements, that is to say, vertices or edges that are not connected to any cryptographic parameters.

However, the minimality property may also cause some perfectly relevant elements to be discarded. In fact, if they are not located on the shortest path between a pair of cryptographic parameters, they will not be included in the slice. This scenario is illustrated by an example in Figure 3. On the left there is a possible data flow of a CTR mode and on the right its corresponding slice. The counter is implemented using two variables, as it could be the case for a 128-bit counter on 64-bit architecture. For a large majority of executions, including the one used to build the data flow of the example, only the least significant part is being incremented. Of the two existing paths between the input of the block cipher, only the shortest is included in the slice. Thus, the information about the addition, which is useful to identify the CTR mode, is lost.

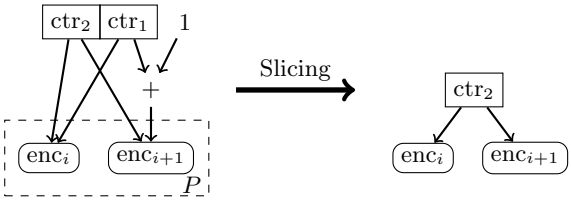


Fig. 3. DFG of a CTR implementation and its corresponding slice. The counter is implemented using two variables: ctr_1 (resp. ctr_2) is the least significant part (resp. most significant part).

Including any paths and not only the shortest one, is not a possible solution to this problem. In fact for some parameters, there are a lot of paths that are strictly equivalent.

For instance, the AES128 round key buffer is made of 44 32-bit words. Thus, there would be 44 paths for each pair of encryptions sharing the same round key buffer. To avoid redundant elements (representing the same information several times) we stick with the original slice definition. A possible workaround for the example of Figure 3 is to split the plaintext of the block cipher into two parts. Since the two parts are now seen as independent parameters, both paths will be included in the slice (that will be equivalent to the original data flow). Obviously, this solution requires a priori knowledge of the mode that is going to be identified. As such, it cannot be used directly in a first approach, but can be used later during a refinement phase.

5.3 Practical Greedy Algorithm

Finding a minimum distance preserving subgraph is a difficult task. A basic idea is to search for a shortest path for every pair of P^2 and to take their union. Since the path length is measured as the number of edges, a Breadth First Search (BFS) algorithm can be used to compute the shortest path between two vertices. For a sparse graph with a number of edges linear to the number of vertices (as it is the case in our DFG model) the complexity of the BFS algorithm is linear to the number of vertices. Thus, the overall complexity of this simplistic algorithm is $\mathcal{O}(|V_D|.|P|^2)$. However, the resulting subgraph is not necessarily the smallest. If there are several smallest paths for a pair of vertices, the size of the union may depend on which one is chosen. It is illustrated by an example in Figure 4. We want to find a slice for the data flow on the left assuming a set of parameters $P = \{enc_i, enc_j, enc_k\}$. By using the algorithm we just described, we may obtain the slice given on the top right which is equal to the union of (enc_i, key_2, enc_j), (enc_i, key_1, enc_k) and (enc_j, key_2, enc_k). However, the slice given on the bottom right is smaller. This problem is common in practice. In fact, a cryptographic parameter is almost always defined by a set of vertices. For instance on a 32-bit architecture, a 128-bit plaintext is usually split into four 32-bit fragments. One shortest path for each of these fragments is to be expected. Back to the example of Figure 4, key_1 and key_2 could be two fragments of a same key parameter.

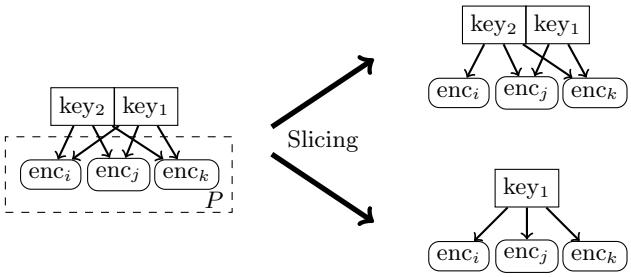


Fig. 4. A data flow with two possible distance preserving subgraphs

In the field of graph spanner, Coppersmith *et al.* [13] describe an approximate algorithm to compute pair-wise preservers. Given a graph $D = (V, E)$ and a set P^2 of pairs of vertices in P, a pair-wise preserver of D with respect to P is a subgraph $D' = (V, E')$ that is distance preserving for the elements of P. Their algorithm produces pair-wise preserver

the size of which is bounded by $\mathcal{O}(|V| + \sqrt{|V|}|P^2|)$. The idea behind their algorithm is to modify slightly the weight of the edges to enforce the uniqueness of the shortest paths. If this upper bound is relevant from the graph spanner perspective, in our case it does not provide any guarantee at all. The DFG is already sparse. Thus, all its subgraphs are under that bound. Apart from this work, we have not been able to find any work or study addressing directly our problem.

An exact solution can be computed using the following algorithm. First, search for the set of shortest paths for every pair of parameters. Then, pick one path from each set, such that their union is minimum. This algorithms suffers from a high complexity. The number of shortest paths can be exponential to the number of vertices. Evaluating every possible selection of paths to find the smallest union has also an exponential complexity.

To reduce its complexity and make it tractable in practice, we make the following modifications. First, we limit to a fixed amount the number of paths returned by the BFS shortest path computation. Second, we used a greedy algorithm to find the set of paths with the smallest union. Iteratively, for each pair of P^2, we insert its shortest path that shares the largest number of edges with the current selection. A pseudo-code for this new algorithm, called the greedy algorithm, is given in Algorithm 1.

Algorithm 1 Greedy Algorithm

```
for all pairs (u, v) of P^2 do
    path_{u,v} = minPath(u, v)
end for
Initialize S as an empty graph
repeat
    pick an unprocessed pair (u, v) such that |path_{u,v}| is minimal
    pick a path p ∈ path_{u,v} such that |V_S ∪ p| is minimal
    add p to S and mark (u, v) as processed
until all pairs of P^2 have been processed
return S
```

The complexity of the greedy algorithm is $\mathcal{O}(|V_D|.|P^2|)$. Although there is no theoretical guarantee that the returned subgraph would be the smallest, it is almost always the case in practice. A list of remarks is given as follow to justify this observation. First, the fixed upper bound on the number of shortest paths is almost never reached. In fact, as previously said, when several shortest paths are found it is often due to parameters fragmentation. Because fragments are rarely mix together outside of the cryptographic primitives, the number of shortest paths is almost always linear to the number of fragments. Second, not every pair of parameters has several shortest paths. Thus, the greedy selection mechanism starts with a non empty set of edges. As a consequence, the first path has not been chosen randomly and more generally we think it helps to stabilize the result. Finally, some sets of shortest paths are disjoints. For instance, for a usual mode of operation, the plaintext path will not intersect the key path. It mitigates the effect of the selection algorithm on the solution.

6 Experimental Evaluation

From an implementation perspective, we divided our solution into two parts. The first one, collects an execution trace of a program, using the PIN [28] framework. This execution trace contains the sequence of executed instructions along with the concrete memory addresses. The computation of the DFG and the extraction of the slice are performed off-line, in the second part. Results are printed in the DOT graph description language.

This section describes the experiments we conducted to evaluate our method. The data set is made of cryptographic implementations of some well-known cryptographic libraries.

6.1 Methodology

To save some space, we do not detail every slice that was obtained. Instead, to assess their usability by a human analyst, we provide measure of their completeness and their readability. These two notions are defined with respect to what should be an optimal data flow pattern in order to identify the mode of operation. The slice is called S, S_{opt} is the optimal pattern and Mcs is a function that returns, for a pair of graphs, its maximum common subgraph. The completeness Cp and the readability Rd are defined as follows:

$$\mathrm{Cp}(S) = \frac{|\,\mathrm{Mcs}(S, S_{opt})|}{|S_{opt}|} \qquad \mathrm{Rd}(S) = \frac{|\,\mathrm{Mcs}(S, S_{opt})|}{|S|}$$

Here, the size of a graph (denoted by $|.|$) is equal to its number of edges. If the slice is equal to the optimal pattern then both the completeness and the readability are equal to 1. During our experiments, the completeness and the readability were computed manually.

We performed experiments for three modes of operation: CBC (encryption and decryption), CTR and HMAC. We give in Figure 5 what we consider to be an optimal pattern for each of these modes. In that representation, the * label may refer not only to any vertices but also to any path that does not intersect the rest of the graph. Some edges have a label to specify to which parameter of the cryptographic primitives they are connected. These patterns contain only the minimal number of executions of the cryptographic primitives to be recognizable. If the analysis window contains more, they will need to be extended. A short explanation for each of these patterns is given as follows.

CBC. For both encryption and decryption, the pattern contains two executions of the block cipher. In both cases, they have the same key parameter. For encryption, the input of the second execution of the block cipher, depends on the output of the first. For decryption, the input of the first execution an the output of the second have a common descendant.

CTR. The pattern contains two executions of the encryption primitive. They have the same key parameter and their input, both depends on the counter initial value.

HMAC. The pattern contains four executions of the compression function (two for each execution of the hash function). The first message block for the inner and outer hash function, are both dependent on the key. The second message block of the outer hash function depends on the output of the inner hash function. The others edges are due to the Merkle-Damgård hash construction. The dashed edge marks the place where the pattern would have to be extended if a larger code window were to be analyzed.

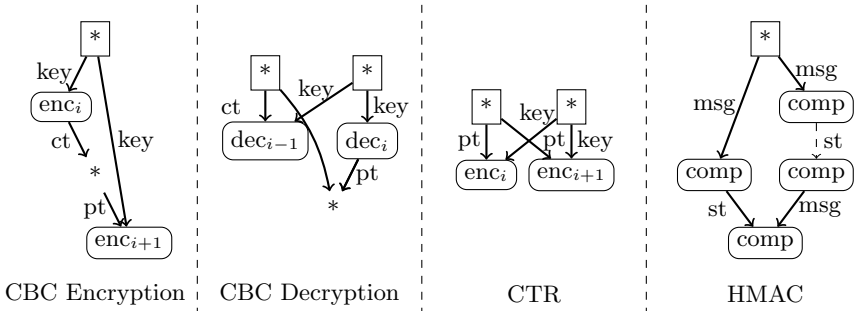


Fig. 5. Optimal data flow pattern for CBC, CTR and HMAC modes of operation

6.2 Results

We evaluated our method against the following cryptographic libraries: Crypto++ [22], LibTomCrypt [25], Nettle [23] and OpenSSL [24]. To be as close as possible to the reality, we did not recompiled these libraries, but took them as they were distributed in their respective Debian package. The CBC and CTR modes were tested with the AES and XTEA block cipher (when available) and the HMAC was tested with the MD5 hash function. For each scenario, we wrote a very simple program that calls the right library function on a small amount of data. We expected the same kind of results on larger programs. In fact, the analysis is limited to a small code window. For cryptographic libraries, this code window is not going to change depending on the amount of code surrounding it. Efficient heuristics that may be used to extract relevant code windows, are presented in Section 7.

Table 1. Measures of the completeness Cp and the readability Rd

	CBC	CTR	HMAC
Crypto++ 5.6.1	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 1
LibTomCrypt 1.17	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 1
Nettle 2.7.1	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 0.71
OpenSSL 1.0.1f	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 0.83

The completeness and the readability measures are given in Table 1. The completeness is always equal to one. It means that the slicing process has not missed any important connection specified by the optimal pattern. The majority of the readability values are also equal to one, meaning that the corresponding slices do not contain superfluous connections. However, smaller readability values were obtained for some HMAC implementations (Nettle and OpenSSL). These slices contain a common ancestor between the last block of the two executions of the hash function. After a thorough investigation, it appears that this common ancestor is the size of a message block. In fact, by the specification $k \oplus \texttt{opad}$ and $k \oplus \texttt{ipad}$ have the same size than a message block. Thus, the size of the messages $k \oplus \texttt{opad}||H(k \oplus \texttt{ipad}||m)$ and $k \oplus \texttt{ipad}||m$ both depend on the size of a message block. Since the message padding includes the length of the message, it is perfectly legitimate for

the last block to depend on the size of a block. Nevertheless we count it as a superfluous connection since it can be misleading for inexperienced analysts.

To conclude, our method have given promising results. In particular, every elements necessary to identify the mode of operation were obtained and the percentage of superfluous elements was never overwhelming.

7 Detailed Uses Cases

In this section we detail three application scenarios. First, we apply our solution on an OCB implementation to demonstrate that it can scale to more complex modes of operation. Second, we show that our solution can be used to quickly identify a malicious CBC implementation containing a backdoor. Finally, we confront our solution with an instant messaging application, to illustrate how it can be used on larger programs.

7.1 Authenticated Encryption: OCB

There are three versions of OCB. This example is based on the implementation of LibTomCrypt which corresponds to the first version, described in [32]. The slice given in Figure 6, was obtained after encrypting a 34-byte message with AES OCB.

To justify why this slice correctly reflects the algorithm and to underline some of its imprecisions we divide the graph into four parts. The first part, colored in blue at the top, computes the first offset which is defined by the following expression: $E_k(N \oplus E_k(0^n))$, where N denotes the nonce, E_k the encryption under the key k and 0^n n bits set to 0. The two AES executions and the XOR operation in between are visible in the graph. The second part, colored in orange at the bottom left, encrypts the two first message blocks by evaluating the expression: $E_k(M[i] \oplus Z[i]) \oplus Z[i]$, where $M[i]$ is the ith message block and $Z[i]$ the ith offset (random mask). Here again the slice perfectly transcribes the algorithm specification. The two message blocks correspond to the two LOAD vertices at the center of the graph. The offset $Z[i]$ is XORed two times, before and after the encryption. The OR and PART1_8 operators are due to size changes from 32-bit to 8-bit variables and conversely. The third part, colored in violet on the right, corresponds to the last block encryption defined by: $E_k(len(M) \oplus L(-1) \oplus Z[m]) \oplus M[m]$. The last message block $M[m]$ does not appear in the graph. $M[m]$ is read only once for the whole scheme. Thus, it does not belong to any path between cryptographic parameters and it was not reported in the slice. The last part, colored in green at the bottom right, computes the authentication tag defined by: $E_k(M[1] \oplus ... \oplus M[m-1] \oplus (C[m]||0^*) \oplus Y[m] \oplus Z[m])$, where $C[m]||0^*$ is the last encrypted block padded with zeros and $Y[m] = E_k(len(M) \oplus L(-1) \oplus Z[m])$. As previously said, the message used for this slice is 34-bytes long. Thus, the size of $C[m]$ is 2 bytes. These two bytes are obviously not on any shortest path, since they involved an additional XOR operation compared to $Y[m]$. With this remark in mind, the slice appears to contain the right dependencies: the two message blocks, $Y[m]$ and $Z[m]$ are XORed together and the result is encrypted. For brevity, we will not dig into how the different offsets are generated. As far as we have conducted our analysis, no inconsistency between the slice and the specifications has been found.

To conclude, our slicing model was able to capture most of the interesting connections even though some are missing (the XOR with $C[m]||0^*$ for instance). Obviously the complexity of this mode reduces the advantage of a graph representation for a human analyst.

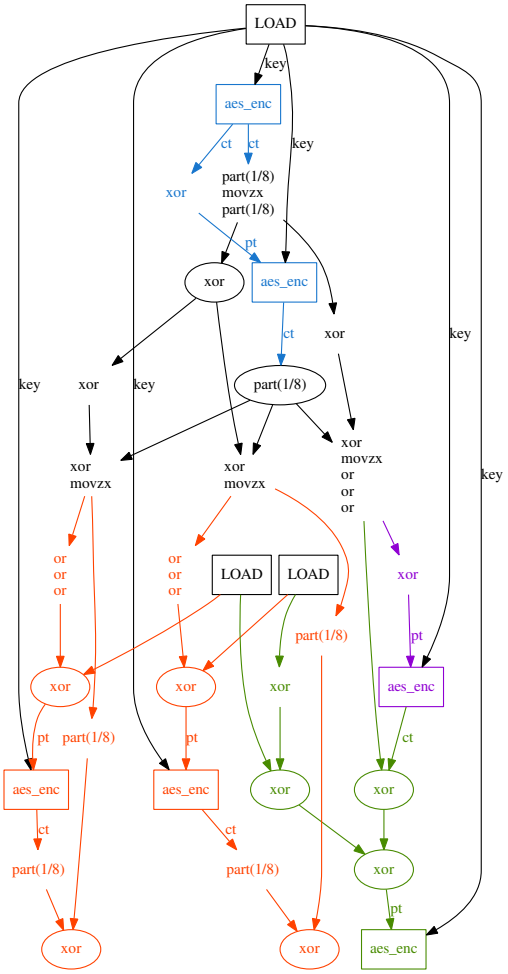
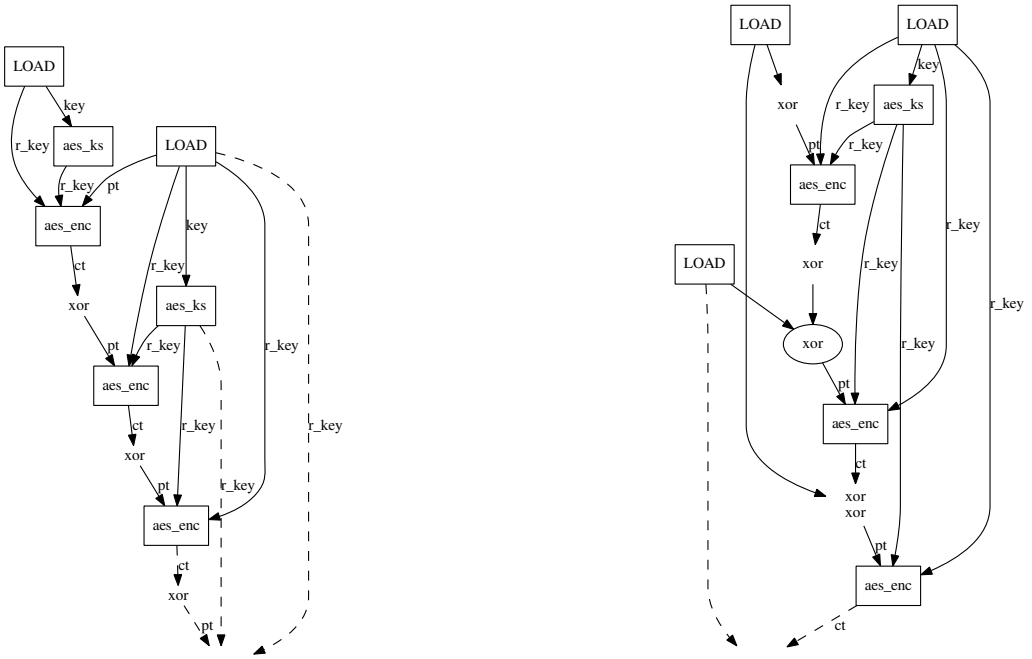


Fig. 6. Experimental slice obtained for the AES OCB implementation of the TomCrypt library executed on a three block message

However, as demonstrated in the last paragraph, it is still possible to understand it with the help of the specifications.

7.2 IV-Replacement Attack

Algorithm Substitution Attack (ASA) consists in replacing the original encryption algorithm by a malicious one containing backdoor capabilities. There has been a renewed attention in the last past years for ASA, as shown by recent publications in that domain [8, 7]. Closed source implementations of symmetric cryptography are attractive targets for ASA. Thus, while evaluating binary software, security experts could be interested in detecting ASA. This example shows that our method can automatically discloses an IV-replacement attack.



(a) AES CBC encryption subject to an IV-replacement attack.

(b) AES IGE used by Telegram to encrypt messages.

Fig. 7. Experimental slices

An IV-substitution attack is a simple ASA that was first described in [8]. It can be used against any encryption scheme that surfaces its IV, such as CBC or CTR. Two keys are used: the legitimate encryption key k defined by the user and a second key k' known only by the attacker. The IV is replaced by k encrypted under k'. Anyone with the knowledge of k' can decrypt the IV, recover k and finally decrypt the data.

For this experiment, we have implemented a very simple AES CBC encryption subject to an IV-replacement attack. The encryption key k is also encrypted using AES. It is a simplistic example, since in reality one will probably use public key cryptographic to correctly conceal the encryption key k. To start the analysis, we located the AES key schedule and the AES encryption, using primitive identification methods. The slice that was returned by our method is given in Figure 7a. It is easy to recognize three CBC patterns in the middle: encryption executions are chained by XOR operations. Notice that encryptions depend on both the result of the key schedule and the key (LOAD labels in the graph). It is perfectly correct since the first four round keys are equal to the key. The key schedule is executed two times: one for each k and k'. The IV generation happens on the top left corner: we noticed that the first AES encryption takes as a plaintext parameter a value read form the memory that is later used as input by a key schedule execution. This is the encryption key k. The IV-substitution pattern is thus clearly visible.

7.3 Instant Messaging Application

For simplicity reasons, every results provided so far were obtained on wrapper applications that just call few functions from cryptographic libraries. In this section, we apply our solution on a much larger program: the Telegram client for Linux. Telegram is an instant messaging service that uses a custom encryption scheme called MtProto [34]. Brief specifications of this protocol can be found on editor's website. Client applications are available for several operating systems and they are all open source. Thus, it will be easy to check the validity of our results.

To extract interesting code fragments for our analysis, we use three simple heuristics. First, we looked for large basic blocks (more 40 instructions). Symmetric cryptography algorithms have very few conditional statements resulting in large basic blocks. Second, we filtered the basic blocks that had a low ratio of logical bitwise instructions. Finally, we kept only functions that did not call any sub function and, of course, contained at least one of the previously selected basic blocks. These three heuristics returned, for an execution trace of nearly a billion dynamic instructions and more than 130000 basic blocks, only a dozen of functions. Among them, we found AES (encryption and decryption), SHA1 and MD5. The others are checksum or compression functions.

The slice we obtained for the encryption part of protocol is given in Figure 7b. It covers the encryption of the first two blocks of a message. The IGE mode of operation is perfectly recognizable.The two blocks of message (corresponding to the LOAD vertices on the left), are XORed with their previous ciphertext block, encrypted and XORed with their previous plaintext block.

8 Conclusion

In this paper we have presented an automated solution to produce synthetic representations of the principal data transfers occurring in modes of operation. A formal definition ensures that this representation, called a slice, is both, sufficiently complete to reliably identify the mode of operation and, easily readable to benefit from the flexibility of human interpretation. We have described how slices can be computed. First we generate a dynamic DFG from an execution trace containing the executed instructions and the concrete memory addresses. Then, we locate in the DFG, the code and the parameters of the cryptographic primitives, using a signature-based identification technique. Finally, we apply a greedy algorithm to find the smallest representation possible.

We have demonstrated with experimental results on CBC, CTR and HMAC that, in practice, the slices produced by our method are complete and readable. In the last section, we have described in details three application scenarios to illustrate the capability of our solution. For the three scenarios, a complex mode of operation, a modified one with a security flaw and a real world program, our method performed well. Security analysts can take advantage of the results provided our solution, to quickly identify modes of operation and to get a good understanding of their internal structure. As such it should be highly profitable for black box audits and any other activities that require to reverse engineer the binary code of mode of operation.

References

1. Albrecht, M.R., Paterson, K.G., Watson, G.J.: Plaintext recovery attacks against SSH. In: 30th IEEE Symposium on Security and Privacy (S&P 2009). pp. 16–26 (2009)
2. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy. pp. 526–540 (2013)
3. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004. pp. 5–23 (2004)
4. Bayer, U., Comparetti, P.M., Hlauschek, C., Krügel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2009 (2009)
5. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference. pp. 1–15 (1996)
6. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: 38th Annual Symposium on Foundations of Computer Science, FOCS '97. pp. 394–403 (1997)
7. Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1431–1440 (2015)
8. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. IACR Cryptology ePrint Archive 2014, 438 (2014)
9. Bonfante, G., Kaczmarek, M., Marion, J.: Morphological detection of malware. In: 3rd International Conference on Malicious and Unwanted Software, MALWARE 2008. pp. 1–8 (2008)
10. Calvet, J., Fernandez, J.M., Marion, J.: Aligot: cryptographic function identification in obfuscated binary programs. In: the ACM Conference on Computer and Communications Security, CCS'12. pp. 169–182 (2012)
11. Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In: Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference. pp. 583–599 (2003)
12. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI 18(3), 265–298 (2004)
13. Coppersmith, D., Elkin, M.: Sparse sourcewise and pairwise distance preservers. SIAM J. Discrete Math. 20(2), 463–501 (2006)
14. Degabriele, J.P., Paterson, K.G.: On the (in)security of ipsec in mac-then-encrypt configurations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010. pp. 493–504 (2010)
15. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011. pp. 41–60 (2011)
16. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011. pp. 63–72 (2011)
17. Jallad, K., Katz, J., Schneier, B.: Implementation of chosen-ciphertext attacks against PGP and gnupg. In: Information Security, 5th International Conference, ISC 2002. pp. 90–101 (2002)
18. Katz, J., Schneier, B.: A chosen ciphertext attack against several e-mail encryption protocols. In: 9th USENIX Security Symposium (2000)
19. Khoo, W.M., Mycroft, A., Anderson, R.: Rendezvous: a search engine for binary code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13. pp. 329–338 (2013)

20. Lestringant, P., Guihéry, F., Fouque, P.: Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15. pp. 203–214 (2015)
21. Li, X., Wang, X., Chang, W.: Cipherxray: Exposing cryptographic operations and transient secrets from monitored binary execution. IEEE Trans. Dependable Sec. Comput. 11(2), 101–114 (2014)
22. Crypto++. http://www.cryptopp.com/
23. Nettle. http://www.lysator.liu.se/~nisse/nettle/
24. Openssl. https://www.openssl.org/
25. Libtomcrypt. http://libtom.org/
26. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings. pp. 31–46 (2002)
27. Liu, C., Chen, C., Han, J., Yu, P.S.: GPLAG: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 872–881 (2006)
28. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. pp. 190–200 (2005)
29. Paterson, K.G., AlFardan, N.J.: Plaintext-recovery attacks against datagram TLS. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012 (2012)
30. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size does matter: Attacks and proofs for the TLS record protocol. In: Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security. pp. 372–389 (2011)
31. Paterson, K.G., Watson, G.J.: Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In: Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 345–361 (2010)
32. Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: a block-cipher mode of operation for efficient authenticated encryption. In: CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security. pp. 196–205 (2001)
33. Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D.J., Su, Z.: Detecting code clones in binary executables. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009. pp. 117–128 (2009)
34. Telegram. https://telegram.org/
35. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995), http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html
36. Vaudenay, S.: Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In: Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques. pp. 534–546 (2002)

A Primitive Identification Issues

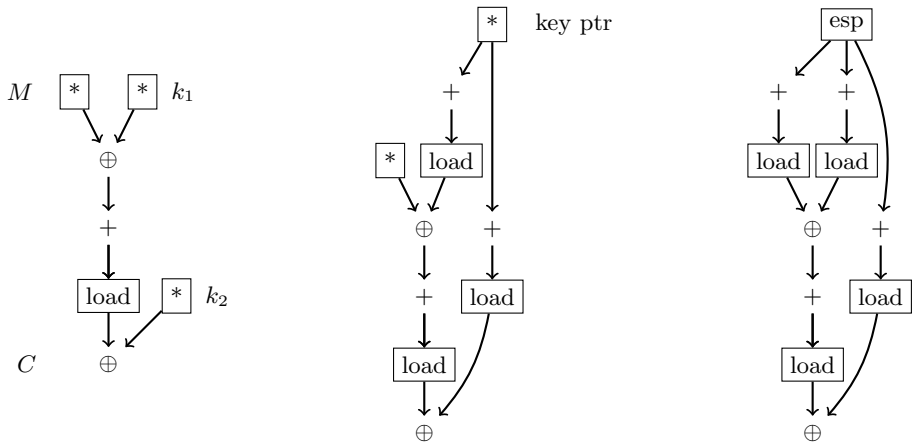
Independent Parameters. It is clear that as the signature's size increases, the harder it will be to detect the signature. In fact, the normalization phase is not always able to remove the variations that exist between the different implementations of a same algorithm. Complete and precise signatures are putting more pressure on the normalization phase and increases the number of false negative. Conversely, it might be tempting to limit signatures to only one part of the algorithm (that is less subject to variation). But in that case, the retrieved parameters will correspond to intermediate results or to already processed entries. It could be problematic if those parameters are dependent on one another. In fact, searching connections between those parameters instead of the real independent ones will cause irrelevant connections to be found, reducing the readability of the slice and in some extreme cases even concealing other connections.

Parasitic Detection. Given a graph G we note $Aut(G)$ its automorphism group. Let us consider a signature graph $S = (V_S, E_S)$ and a DFG $D = (V_D, E_D)$. We assume that $f : V_S \mapsto V_D$ is the edge preserving one-to-one function such that $f(V_S)$ is the signature match we would like to detect. We call parasitic detections of f every subgraph isomorphism of the form: $g \circ f \circ h$ where $h \in Aut(S)$ and $g \in Aut(D')$ with $g_{|f(V_S)} \neq id$ and D' a subgraph of D containing $f(V_S)$.

Of course, in practice it is impossible to tell which detection is a correct matching of the algorithm and which one is parasitic. Symmetries responsible for parasitic detection often concerns input parameters since they lack operand constraints (they have no operand and because they may have been produced by any operation, their label is left unspecified). Consequently, input parameter identification is prone to false positives. The example of Figure 8 illustrates how symmetries in the signature and in the DFG, can result in incorrect parameter detections. This example is based on a simple toy cipher defined by: $C = S(M \oplus k_1) \oplus k_2$ where C is the ciphertext, S a public permutation implemented as lookup table, M the plaintext and k_1, k_2 two keys.

Figure 8a represents what would be a typical signature for this cipher. We notice that this signature has a non trivial automorphism. The image of M by the automorphism is k_1 and vice versa. Thus, it is impossible with this signature to distinguish those two inputs. A possible solution to break the signature symmetry, is to add additional constrains on one of the parameters. We proposed an *enhanced* signature in Figure 8b. This signature will operate under the assumption that the keys are accessed through the same pointer. It has no symmetry any more. However, if we consider the DFG of Figure 8c, the enhanced signature will still produce parasitic detections. In fact, the three input parameters are read from the stack in a symmetric way. As a consequence there are two subgraphs which are isomorphic to the signature, each of them with different vertices for M and k_1. Hence, again we are unable to dissociate the plaintext from the key.

This issue happens for real cryptographic algorithms such as AES for instance (the first add round key is symmetric, thus it is hard to correctly dissociate the plaintext from the round key). A possible solution is to filter incorrect parameters using the IO relationship method described in Section 4.1.



(a) Simple signature: it has a non trivial automorphism.

(b) Enhanced signature: the key pointer is specified to break the symmetry.

(c) For this DFG, even the second signature cannot to distinguish M from k_1.

Fig. 8. A possible DFG and two signatures for the toy cipher $S(M \oplus k_1) \oplus k_2$