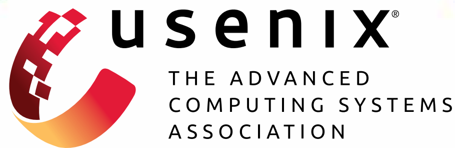


Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code

Carlo Meijer, *Radboud University;* Veelasha Moonsamy, *Ruhr University Bochum;*
Jos Wetzels, *Midnight Blue Labs*

https://www.usenix.org/conference/usenixsecurity21/presentation/meijer

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code

Carlo Meijer
Radboud University
The Netherlands
cmeijer@cs.ru.nl

Veelasha Moonsamy
Ruhr University Bochum
Germany
email@veelasha.org

Jos Wetzels
Midnight Blue Labs
The Netherlands
a.l.g.m.wetzels@gmail.com

Abstract

The continuing use of proprietary cryptography in embedded systems across many industry verticals, from physical access control systems and telecommunications to machine-to-machine authentication, presents a significant obstacle to black-box security-evaluation efforts. In-depth security analysis requires locating and classifying the algorithm in often very large binary images, thus rendering manual inspection, even when aided by heuristics, time consuming.

In this paper, we present a novel approach to automate the identification and classification of (proprietary) cryptographic primitives within binary code. Our approach is based on Data Flow Graph (DFG) isomorphism, previously proposed by Lestringant et al. [43]. Unfortunately, their DFG isomorphism approach is limited to known primitives only, and relies on heuristics for selecting code fragments for analysis. By combining the said approach with symbolic execution, we overcome all limitations of [43], and are able to extend the analysis into the domain of unknown, proprietary cryptographic primitives. To demonstrate that our proposal is practical, we develop various signatures, each targeted at a distinct class of cryptographic primitives, and present experimental evaluations for each of them on a set of binaries, both publicly available (and thus providing reproducible results), and proprietary ones. Lastly, we provide a free and open-source implementation of our approach, called *Where's Crypto?*, in the form of a plug-in for the popular IDA disassembler.

1 Introduction

Despite the widely-held academic consensus that cryptography should be publicly documented [37, 40, 67], the use of proprietary cryptography has persisted across many industry verticals ranging from physical access control systems [1,61,67,70,71,73] and telecommunications [26,30,55] to machine-to-machine authentication [13, 67].

This situation presents a significant obstacle to security-evaluation efforts part of certification, compliance, secure procurement or individual research since it requires resorting to highly labor-intensive reverse-engineering in order to determine the presence and nature of these algorithms before they can be evaluated. In addition, when a proprietary algorithm gets broken, details might not be published immediately as a result of NDAs or court injunctions [5] leaving other potentially affected parties to repeat such expensive efforts and hampering effective vulnerability management. As such, there is a real need for practical solutions to automatically scan binaries for the presence of as-of-yet unknown cryptographic algorithms.

Criteria In order to support the analysis of closed-source embedded systems for the use of proprietary cryptography, a suitable solution should meet the following criteria: (i) identification of as-of-yet unknown cryptographic algorithms falling within relevant taxonomical classes, (ii) efficient support of large, real-world embedded firmware binaries, and (iii) no reliance on full firmware emulation or dynamic instrumentation due to issues around platform heterogeneity and peripheral emulation. As discussed in Section 3, there is no prior work meeting all of these criteria.

Approach To meet the above criteria, our approach bases itself on a structural taxonomy of cryptographic primitives. The idea is that, since the vast majority of proprietary cryptography falls within established primitive classes [67], we can develop structural signatures allowing for the identification of any algorithm within these classes without having to rely on knowledge of the algorithm's particularites. To this end, we utilize a taxonomy based on [4, 39, 46, 50] and illustrated in Figure 1. Note that this taxonomy is purely instrumental and does not intend to be exhaustive or allow for an exclusive partitioning of algorithms.

Our approach is built on two fundamentals: Data Flow Graph (DFG) isomorphism and symbolic execution. As described in Section 4, the limitations of prior work on DFG isomorphism [43] are overcome through augmentation with symbolic execution which allows us to specify structural signatures for taxonomic classes of cryptographic primitives and

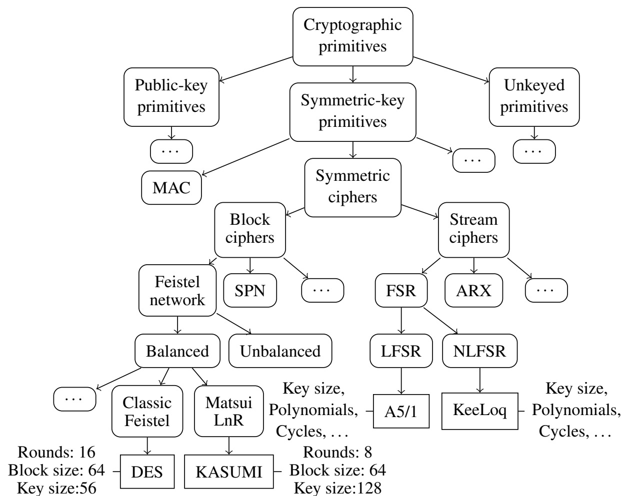


Figure 1: Taxonomical tree of algorithm classes

analyze binary code for matches. The focus of this paper is on symmetric and unkeyed primitives.

Contribution Our contribution is threefold. First, our novel approach combines subgraph isomorphism with symbolic execution, solving the open problem of fragment selection and eliminating the need for heuristics and thus, overcoming the limitations of prior work which rendered it unsuited to identifying unknown ciphers. To the best of our knowledge, as discussed in Section 3, there is currently no prior work in either industry or academia that addresses the problem of identifying unknown cryptographic algorithms. Second, we propose a new domain-specific language (DSL) for defining the structural properties of cryptographic primitives, along with several examples. Finally, a free and open-source proof-of-concept (PoC) implementation, *Where's Crypto?*, is made available[1] and evaluated in terms of analysis time and accuracy against relevant real-world binaries.

2 Scope and limitations

Normalization and optimization A single function can be represented as many different combinations of assembly instructions depending on architecture and compiler particularities. Attempting to construct a 1–to–1 mapping between semantic equivalence classes and DFGs is beyond the scope of this work. When our normalization maps two expressions to the same DFG node, they are considered to be semantically equivalent. While the inverse is not necessarily true, our approach can operate as if this were the case since, for a compiler to take advantage of semantic equivalences, it must be consistently aware of them. Therefore, we can leverage this fact to recognize compiler-generated equivalences.

[1] https://github.com/wheres-crypto/wheres-crypto

Implicit flows Data dependencies may also arise due to control-dependent assignments. For example, given two boolean variables a and b, statements $a \leftarrow b$ and `if` a `then` $b \leftarrow$ **true**; `else` $b \leftarrow$ **false** are semantically equivalent. In the former, b directly flows to a, and therefore the dependency is apparent in its corresponding DFG, whereas in the latter, the dependency information is lost. Since data-dependent branches increase side-channel susceptibility, developers should refrain from using them for cryptographic primitives. Therefore, we believe it is justified to declare implicit flows out of scope. Note that implicit flows is a concept different from data-dependent branches. Support for the latter is achieved by means of symbolic execution (Section 6).

Function entry points Our PoC implementation relies on IDA's recognition of function entry points as input to our algorithm. As such, inaccuracies in IDA's function recognition will reduce our coverage. However, this is not an inherent limitation of our approach but merely of the implementation.

Code obfuscation Since code obfuscation presents an inherent challenge to any binary-analysis approach, our approach assumes that the input it operates on is not obfuscated and delegates this de-obfuscation to a manual and/or automated pre-processing step. Automated binary deobfuscation is a well-established research field of its own which consists of a wide variety of static, dynamic, symbolic and concolic approaches [24, 57, 75, 77] drawing upon synthesis [9, 11], optimization [31], semantic equivalence [65] and machine learning [64] based techniques in order to make obfuscated binaries amenable to analysis.

Taxonomical constraints In our PoC evaluation and the examples of our DSL, we have limited our discussion to a subset of the taxonomy of cryptographic primitives. This is not an inherent limitation of our approach, but merely of our PoC and its evaluation. Our approach is essentially agnostic with respect to the employed taxonomy, which can be extended as users see fit, and only assumes that the algorithm the analyst is looking for is within one of its classes. Given that the vast majority of proprietary cryptography falls within a specific subset of established primitive classes [67], namely stream- and block ciphers and hash functions, we do not consider this a practical issue.

False positives Certain primitive classes are a subset of others and some instances fit the definition of several ones. As such, their matches are prone to false positives. Examples of such are discussed in Section 11.2.1. We do not consider this a serious practical problem as our solution is intended to assist a human analyst who will be easily capable of pruning a limited number of false positives compared to the burden of unassisted analysis required by the status quo.

Furthermore, certain primitive classes are essentially underdefined. That is to say, their definition is so broad that

characteristic properties are not distinctive enough for a meaningful identification. For example, the defining property of stream ciphers is two data streams being XOR-ed together. Obviously, identifying instances of XOR results in an overwhelming number of false positives. In case a signature for such a generic class is desired, an alternative approach is to craft signatures for every subclass contained within it.

Path oracle policy The path oracle policy discussed in Section 6.1 is chosen such that the resulting graph represents n iterations of an algorithm. While this typically satisfies our goals, there are a few exceptions to this rule. First, compilers sometimes ensure loop-guard evaluation during both entry and exit, resulting in a DFG representing $n+1$ iterations. Second, cryptographic primitives with a constant iteration length are beyond the control of the path oracle. Finally, loop unrolling will result in a DFG representing kn iterations, where k denotes the number of compiler-grouped iterations. In order to overcome this limitation, we suggest taking the possibility of iteration count deviating from n into account during signature construction as described in Section 10, for example by defining a minimum rather than an exact match.

3 Prior work

Prior work by academia and industry into the identification of cryptographic algorithms in binary code can be divided into (combinations of) the following approaches:

Dedicated functionality identification The most naive and straight-forward approach consists of identifying dedicated cryptographic functionality in the form of OS APIs (e.g. Windows CryptoAPI/CNG) [47], library imports or dedicated instructions (e.g. AES-NI). This approach is inherently incapable of detecting unknown algorithms.

Data signatures The most common approach employed in practice [3, 36, 44, 45, 52, 56, 58, 74] consists of identifying cryptographic algorithms on the basis of constants (e.g. IVs, Nothing-Up-My-Sleeve Numbers, padding) and lookup tables (e.g. S-Boxes, P-Boxes). The approach is unsuitable for detecting unknown algorithms. Moreover, the same applies for known algorithms that do not rely on fixed data, or those that do, but, for example, use dynamically generated S-Boxes, rather than embedded ones.

Code heuristics Another series of approaches rely on code heuristics, which are applied either statically or dynamically, like mnemonic-constant tuples [35, 42], which take into account word sizes, endianness, and multiplicative and additive inverses but otherwise suffer from the same drawbacks as data signatures.

A second heuristic relies on the observation that symmetric cryptographic routines tend to consist of a high ratio of bitwise arithmetic instructions [18, 35, 42, 47, 56] and attempt to classify functions based on a threshold. The drawback of this approach is that it lacks granular taxonomical identification capabilities as well as being highly prone to false positives, especially on embedded systems where heavy bitwise arithmetic is typically present as part of memory-mapped register operations required for peripheral interaction.

Deep learning Hill et al. [38] propose a Dynamic Convolutional Neural Network based approach which, however, is unsuited for our purposes due to its reliance on dynamic binary instrumentation and its inherent inability to classify unknown algorithms.

Data flow analysis One set of approaches to data flow analysis relies on the static relation between functions and their inputs and outputs [19, 35, 47, 53]. One plausible approach is to perform taint analysis and evaluate function I/O entropy changes, which relies on emulation and as such is unsuitable as per our criteria in Section 1. Another approach is to compare emulated or symbolically executed function I/O to a collection of reference implementations or test vectors, which is inherently incapable of detecting unknown algorithms.

Another approach [76] utilizes dynamic instrumentation and symbolic execution to translate candidate cryptographic algorithms into boolean formulas for subsequent comparison to reference implementations using guided fuzzing. However, its reliance on dynamic instrumentation and inherent inability to recognize unknown algorithms render the approach unsuitable for our purposes.

Finally, there is the DFG isomorphism approach as proposed by [43] which produces DFGs from a given binary and compares it against graphs of known cryptographic algorithms through the use of Ullmann's subgraph isomorphism algorithm [66]. A DFG is a Directed Acyclic Graph (DAG) representing the flow of data within a sequence of arithmetic/logic operations. A vertex represents either an operation, or an input variable. The presence of an edge between vertex v_1 and v_2 means that v_1 (or the result of operation v_1) is an input to operation v_2. Due to the nature of DFGs, code flow information cannot be expressed. As such, the contributions of [43] are limited to linear sequences of instructions. Moreover, the authors argue that since cryptographic implementations ought to avoid data-dependent branching due to side-channel susceptibility, one can assume all cryptographic code is free from data-dependent conditional instructions. This latter generalization introduces several limitations.

First, no straightforward strategy for selecting code fragments is proposed. Performing the analysis on a per-function basis is complicated by the fact that cryptographic implementations are commonly surrounded by some basic control logic, such as checks on input parameters. As a result, analysis can neither be applied to entire functions nor across function boundaries through inlining and hence the authors propose a limited set of selection heuristics constraining the work.

Second, the approach performs well when identifying known algorithms since one can take advantage of algorithm-

unique characteristics, but this does not hold when attempting to identify unknown algorithms. Furthermore, a common pattern is that the class of a cryptographic primitive often only becomes apparent once the analysis incorporates conditional instructions. We clarify this point using the following toy examples.

Suppose that we would like to identify a proprietary stream cipher σ. A typical implementation contains a key-stream generator, generating pseudo-random bytes in a loop. Inevitably, this loop contains a conditional instruction causing the program to either re-enter or exit the loop, depending on the length parameter. As there is no support for conditional instructions depending on non-constant values, DFG G, generated from σ will, at most, represent a single iteration, covering a single unit of input length (bytes or otherwise). In this typical example, clearly, a stream cipher pattern will not become apparent in G. The example can be generalized to any pattern that becomes apparent only after several iterations, where no additional properties of the target primitive are known.

Similarly, suppose that we would like to identify a proprietary hash function θ, based on a Merkle-Damgård construction. θ invokes compression function F, which processes blocks of fixed input length. The Merkle-Damgård construction is then used to allow variable input lengths. As such, in order to generate a DFG wherein the construction is apparent, we need it to incorporate several iterations, and perform inlining of F. The former is problematic (as per the stream cipher example), and so is the latter in case F performs some kind of input validation, for e.g. checking for NULL pointers.

4 Solution overview

Cryptographic primitives are essentially a set of arithmetic and logical operations representing an input/output relation. This structural relationship between operations and data can be expressed as a DFG. Since all particular algorithms will be structurally similar to the general primitive defining their taxonomical class, the problem of identifying an unknown algorithm assumed to belong to a well-defined taxonomical class can be formulated as a DFG subgraph isomorphism problem. However, due to slight differences in implementation and compiler peculiarities, DFG representations of semantically identical algorithms may differ and such representations require normalization before they can be subjected to isomorphism analysis. Lestringant et al. [43] demonstrated that, by repeatedly applying a set of rewrite rules to the DFG, a normalized version is obtained, wherein many of these variations are removed. Although no guarantee can be given that equivalent semantics will always map to the same DFG, the result is 'good enough' to serve as a data structure for the purpose.

The identification procedure consists of three stages. A diagram of the procedure is given in Figure 2. First, given the entry point of a function, we start executing it symbolically. A DFG is constructed during the execution, where each instruction adds a set of nodes and edges to the graph. In case a conditional instruction is encountered, the execution path belonging to the condition evaluating to **true**, **false**, or both paths are explored. In the latter case, the partially constructed DFG is duplicated and the construction continues independently for both execution paths. Hence, the final result of the DFG construction phase is, in fact, a set of DFGs describing the input/output relation corresponding to the execution path taken. Section 5 describes the construction phase in detail.

Second, once a DFG is fully constructed, we enter the purging phase. This phase is responsible for removing nodes from the graph that represent neither an output, nor a value used in the computation of any output. As such, the graph is reduced to a form in which it only represents the input/output relation, free from operations introduced due to register spilling and other possible implementation, compiler, and architecture-specific operations that are irrelevant to the function's semantics. Section 7 describes the purging phase in detail.

Last, with the finalized DFG at our disposal, we enter the pattern-matching phase, where we search for subgraphs in the DFG that are isomorphic to the graph signature of a given cryptographic primitive. If such a subgraph is identified, we conclude that the primitive is indeed present in the instructions from which the DFG was generated. We use Ullmann's subgraph isomorphism algorithm for searching the DFG. Section 8 describes the pattern-matching phase in detail.

5 Data Flow Graph construction

The approach of constructing the DFG from assembly instructions builds upon that of [43]. This section summarizes their approach, and indicates where ours departs from it.

Suppose we have a sequence of assembly instructions. We construct its corresponding DFG, $G = (V, E)$, by converting each instruction i into a set of operations O_i, which can potentially be empty (e.g., a `NOP` or branch), or contain multiple operations (e.g., a complex instruction). We distinguish three cases based on input type, as follows:

Immediate We create a vertex representing a constant value in G. It is linked by an edge to O_i.

Register In case an instruction takes a register as an input operand, we create an edge between the last value written to that register and O_i. In practice, this means we maintain an array containing, for each register, a reference to the vertex in G corresponding to that value.

Memory For operands that load or store from/to memory, we create `LOAD` and `STORE` operations. Both operations take a memory address vertex as input. Like any other vertex, the address can be a constant, or a more complex symbolic expression.

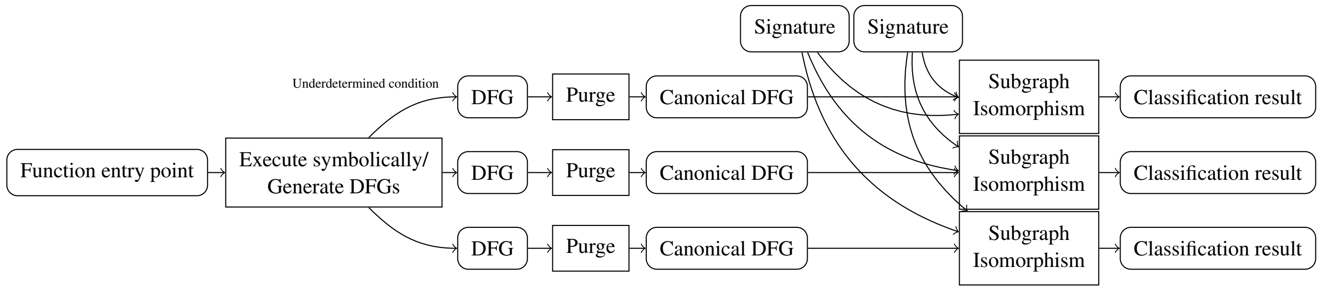


Figure 2: Diagram of primitive identification process

Ideally, we would like all code fragments within a semantic equivalence class to map to the same DFG, and have the end result represent the semantics only, free from architecture and compiler-specific traits. The approach followed by [43] is to take the generated DFG, and repeatedly apply normalization rewrite rules until a fixed-point is reached. This is where our approach deviates from theirs, as we apply normalization as well, but *continuously* during graph construction. This enhances performance, which we argue below in Section 5.1, and allows us to efficiently keep track of the conditions that apply during symbolic execution (Section 6).

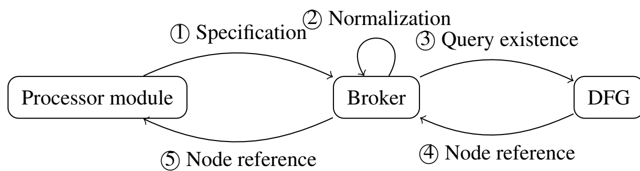


Figure 3: Flow of the graph-node creation process

A diagram of the graph-node creation process is given in Figure 3. More concretely: there is a *processor module*, written for a specific architecture that translates each instruction into graph nodes. The processor module cannot autonomously create new graph nodes. Instead, it must interact with the *broker*. The broker is responsible for the application of normalization rewrite rules and is processor-architecture agnostic. The processor module provides a specification of the desired node to the broker, which in turn applies normalization rewrite rules to the specification. As such, the result either matches the specification exactly, or a different one that is semantically equivalent. After normalization, the broker queries the DFG for whether a node conforming to the normalized specification already exists. If it does, a reference to it is returned, rather than a new node being created. Consequently, there cannot exist two distinct nodes in a graph conforming to the same specification, or equivalent under normalization. We prove this property in Lemma 1.

Lemma 1. *Let $G = (V,E)$ be a DFG, and $\hbar$ denote the normalization transform, for which holds: (1) $\hbar(\hbar(x)) = \hbar(x)$ for all $x \in U$ (universe). Consider arbitrary arithmetic/logical operation $op(v_1, v_2)$, where $v_1, v_2 \in V$.*

A broker request for op preserves the following properties: (i) For all $v \in V$, $v = \hbar(v)$, i.e. all nodes in G are normalized. (ii) For all $v_1, v_2 \in V$, $\hbar(v_1) = \hbar(v_2) \implies v_1 = v_2$, i.e. all nodes in G belong to a unique equivalence class under the normalization function.

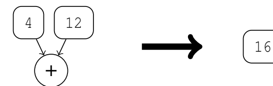
Proof. Assume (i) and (ii) hold for V. We define $q = \hbar(\mathrm{op}(v_1, v_2))$ and distinguish two cases.

If $q \in V$, then G is not modified and (i) and (ii) are trivially preserved. If $q \notin V$, then $V' = V \cup \{q\}$. By applying (1), we get $\hbar(q) = q$, and thus (i) holds for $\{q\}$. Since (i) already holds for V, (i) also holds for V'. Furthermore, suppose that there exists $p \in V$, for which $\hbar(p) = \hbar(q)$. By (i), we get $\hbar(p) = p$, and hence $p = \hbar(q)$. By definition, $q = \hbar(\mathrm{op}(v_1, v_2))$ and hence $p = \hbar(\hbar(\mathrm{op}(v_1, v_2)))$. By (1), we get $p = \hbar(\mathrm{op}(v_1, v_2))$ and thus $p = q$. This contradicts $q \notin V$, and hence no $p \in V$ exists such that $\hbar(p) = \hbar(q)$. Therefore, (ii) holds for V'.

Since (i) and (ii) trivially hold for the base case, i.e., an empty graph G, where $V = \varnothing$, and the above shows preservation during the step case, the properties hold for any G. □

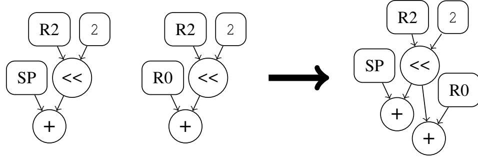
At this point, we are ready to describe the normalization rewrite rules; they include operation simplification, common-subexpression elimination, and subsequent memory access.

Operation simplification Suppose that we encounter an arithmetic/logic operation for which all input parameters are constants. Then, the operation can be replaced by its result.

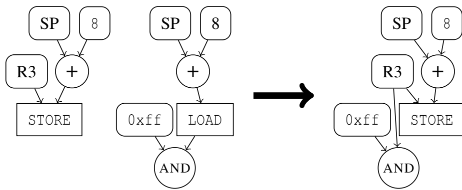


Likewise, in case an element is the identity element for the operation it serves as an input to, the operation has no effect and can be removed. In case an element is the zero element, the operation can be replaced by zero.

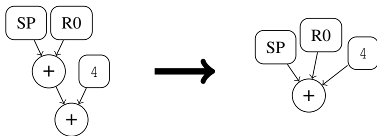
Common subexpression elimination Often within a code fragment, the same value is re-computed several times. This is especially true when the instruction set allows for expressing complex operands, for e.g. supporting offsets and shifts. Lemma 1 states that broker requests for nodes belonging to a certain equivalence class all result in references to the same graph node. Hence, common-subexpression elimination is already achieved by the design of the node-creation process.



Memory access Loading and storing of data from/to main memory is a common operation. However, this need not have a relation with semantics, but may be due to register filling and spilling. We attempt to correct for this by substituting each `LOAD` operation by its result, which is known in case a preceding `STORE` operation to the same memory address node exists. It is important to be able to identify the potential equivalence of memory address nodes passed to the `STORE` and `LOAD` operation. Like any other expression, memory addresses are represented by graph nodes. Given Lemma 1, all equivalent address nodes are mapped to a single graph node. By maintaining a lookup table during graph construction, for e.g., a hash table mapping address nodes to their corresponding stored value, the substitution can be performed in constant time.



For associative operations, the result does not depend on the order in which they are executed. Therefore we translate nested associative operations into a single operation taking all inputs.



Miscellaneous translations Besides the rewrite rules described above, we apply additional miscellaneous rules that do not fit any of the aforementioned categories. They are listed in Appendix B.

5.1 Advantages

Applying the normalization rewrite rules during construction of the graph has several advantages over doing so once the graph is fully generated. First, in case normalization function $\hbar$ has constant running time complexity, then the running time complexity of the construction phase, including normalization, grows linearly with the number of assembly instructions, whereas repeated application on a wholly generated DFG has quadratic complexity.

Second, by Lemma 1, equivalence of any pair of node references can be evaluated in constant time, simply by checking whether $v_1 = v_2$. As such, substitution of `LOAD` operations by their result can be achieved in constant time. The property is also utilized extensively during symbolic execution (Section 6). Suppose some predicate P involves node $v_1 \in V$. Then, a condition involving $v_2 \in V$, can be evaluated immediately under P without the need for proving equivalence of v_1 and v_2 first.

6 Symbolic execution

During the analysis of a function, we may encounter conditional instructions. By definition, a conditional instruction carries a condition. We define the terms *determined* and *underdetermined* conditions. These terms relate to the terminology used in the classification of systems of linear equations. For *determined* conditions, the input variables are restricted to a domain such that there is only a single possible evaluation result. For example, a conditional jump instruction at the end of a loop consisting of a fixed number of iterations. Conversely, for *underdetermined* conditions, the input variables are not restricted enough to determine a fixed outcome. Below we describe how we approach this class of conditions.

During the DFG construction of any function f, we keep a state $\mathcal{S} = (G, P, B)$, where $G = (V, E)$ is the partially constructed DFG. P is the path condition, which is constructed during symbolic execution; a predicate restricting unknown variables to a certain domain so that, if satisfied, the execution path follows the same path taken during the DFG construction. Phrased differently: satisfaction of P warrants that G represents the input/output relation of f. The inverse of this statement need not be true. Finally, backlog B is a mapping between an execution address and a list of booleans. For all underdetermined conditional instructions encountered during the construction of G, B keeps a record of which evaluation result was chosen (i.e., true/false). Since the analysis may encounter the same conditional instruction several times, a list is kept. We define $B_e[i] \in \mathbb{B}$, as the evaluation result chosen during the ith occurrence of the underdetermined conditional instruction located at execution address e.

The graph construction begins by initializing $\mathcal{S} = (G, P, B)$ to the empty state, i.e. G is an empty graph, $P =$ **true**, and B has no record of any evaluation result. Then, we begin the construction by processing the instruction located at the entry point of function f. Some instructions may manipulate the execution flow, for e.g., a branch instruction, in which case, we continue at its target address. The construction is complete when we encounter an instruction causing the execution flow to return to f's calling function. For example, in ARM assembly, this is achieved by writing the initial value of register `LR`, as set by the caller of f, to the program counter register `PC`.

We represent a condition c in the form of a tuple (v_1, o, v_2), where $v_1, v_2 \in V$, and $o \in \{<, \leq, =, \geq, >\}$ is the operator. In case either v_1 or v_2 is non-constant, c need not be underdetermined, as predicate P may sufficiently restrict v_0 or v_1 so that

c is determined. In case c is underdetermined, both execution paths are possible, and we are forced to choose which one to follow. Alternatively, we may follow both paths, by duplicating state $\mathcal{S}$, and subsequently assigning each execution path to one of the instances. This way, the resulting final graph construction consists of several DFGs; each one representing a different execution path. We refer to this practice as *forking* state $\mathcal{S}$. Forking at the occurrence of every underdetermined condition maximizes code coverage. However, it is infeasible due to the state explosion problem. Therefore, we should devise a balanced strategy for when to apply it – as elaborated below.

6.1 Path Oracle

The strategy of when to apply forking only loosely relates to the symbolic execution itself. Therefore, we introduce the *Path Oracle*, a separate entity that is queried during the graph construction phase, for every occurrence of an underdetermined condition c. It decides whether c should evaluate to **true** or **false**, or that the construction should fork and follow both execution paths.

Algorithm 1 Conditional Instruction

```
Require: S = (G,P,B), ExecutionAddress e, Condition c, PathOracle po
  if P ∧ c = true then
      Evaluate instruction at e
  else if P ∧ c = false then
      Skip over instruction at e
  else
      d ← po.query(e, B)
      if d = TAKE_TRUE then
          P ← P ∧ c                                  ▷ expand P with c
          B_e ← B_e ∪ {true}                         ▷ append decision to backlog
          Evaluate instruction at e
      else if d = TAKE_FALSE then
          P ← P ∧ ¬c
          B_e ← B_e ∪ {false}
          Skip over instruction at e
      else if d = TAKE_BOTH then
          S' ← S.fork()                              ▷ S' = (G',P',B')
          P ← P ∧ c
          B_e ← B_e ∪ {true}
          P' ← P' ∧ ¬c
          B'_e ← B'_e ∪ {false}
          e is evaluated for S, skipped for S'
```

For every decision made by the path oracle, P and B in $\mathcal{S}$ are updated accordingly. The pseudocode given in Algorithm 1 depicts how this is done. In short, predicate P is updated to include condition c (or the negation thereof), thereby maintaining satisfaction of its defining property, i.e. satisfaction of P guarantees G represents the input/output relation of f. An entry is added to backlog B, reflecting the decision made by the path oracle. B has no purpose beyond weighing into the decisions made by the path oracle.

6.1.1 Path Oracle Policy

The goal of the policy described below is, for some number n, to obtain a DFG consisting of exactly n iterations of a primitive with variable input length. The target primitive can subsequently be identified by searching for exactly n iterations in the resulting DFG.

We define $d_{e,i} \in \{\texttt{TAKE_TRUE}, \texttt{TAKE_FALSE}, \texttt{TAKE_BOTH}\}$ as the path oracle's decision for the ith query for the conditional instruction found at execution address e. The policy for the path oracle is defined as follows:

$$d_{e,0} := \texttt{TAKE_BOTH}$$
$$d_{e,i} := \left\{ \begin{array}{ll} \texttt{TAKE_TRUE} & \text{iff } B_e[0] = \textbf{true}, \\ \texttt{TAKE_FALSE} & \text{iff } B_e[0] = \textbf{false} \end{array} \right\} \ \forall_i \in [1, n-1]$$
$$d_{e,i} := \left\{ \begin{array}{ll} \texttt{TAKE_FALSE} & \text{iff } B_e[0] = \textbf{true}, \\ \texttt{TAKE_TRUE} & \text{iff } B_e[0] = \textbf{false} \end{array} \right\} \ \forall_i \in [n, \infty]$$

We justify the choice of policy by means of an example. Suppose that we encounter an underdetermined condition c at address e. We do not know which of the two possible execution paths leads to a cryptographic primitive (if any). Hence, for $i = 0$, i.e., the first occurrence, we fork the state and explore both. Suppose that, at a later point during the graph construction, one instance visits address e again, hence $i = 1$, and finds itself with another underdetermined condition c'. Since, at this point, P incorporates c (or $\neg c$), the outcome of c can be evaluated. As c' is underdetermined, $c \neq c'$ is guaranteed.

Such behavior is typical for a loop-guard statement. If this is indeed the case, the execution path taken at $i = 0$ made us revisit e. In light of our goal of constructing a DFG comprising of n iterations of a primitive, we replicate this path choice $n-1$ times, and subsequently take the opposite path, causing the execution flow to exit the loop. Finally, the construction phase yields two DFGs: one representing 0 iterations, and another representing n iterations. A description of the strategy being applied to a concrete example is given in Appendix A. The strategy does not produce exactly n iterations in every situation. Section 2 highlights typical exceptions.

7 Purging process

Once the construction is complete, graph G represents the input/output relation of f, under predicate P. However, it contains other information as well, such as nodes created from temporary loads/stores to the stack, and expressions rewritten by the broker, leaving the source nodes unused. For e.g., suppose that v represents $\texttt{ADD}(x,y)$. Then, a request to the broker for $\texttt{ADD}(v,z)$ yields node w, representing $\texttt{ADD}(x,y,z)$. w does not depend on v and, unless v is referenced independently elsewhere, v is not part of f's input/output relation.

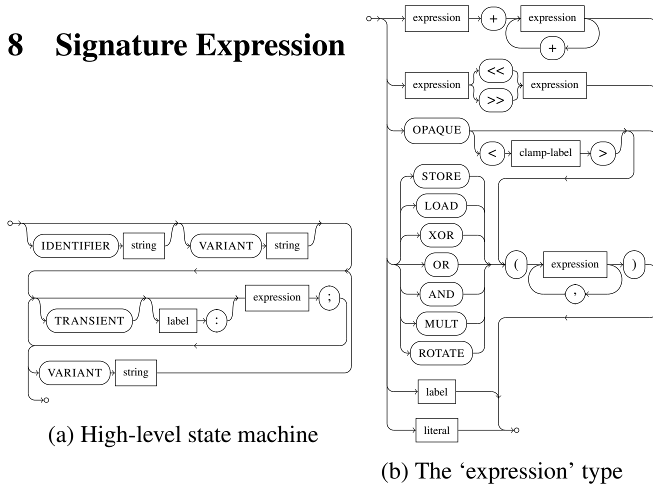
Leaf nodes are, by definition, graph nodes that are not used as an input to any arithmetic/logical operation. Our approach becomes the following: for each leaf node v, we check whether

it is part of f's semantics. We consider leaf node v to be part of f's semantics, if v is either:

(i) the return value of f,
(ii) a `STORE` operation, and the target address is not relative to the `SP` register. Thus, information is stored outside of the stack, or
(iii) a `CALL` operation, i.e. a function call not subject to inlining.

In case none of the above applies, v and its incoming edges can be removed from G, without affecting its semantics. The removal of leaf nodes continues repeatedly until no more nodes can be removed. Finally, by construction, all nodes in G are either leaf nodes that are part of f's semantics, or intermediate results contributing to some leaf.

8 Signature Expression



(a) High-level state machine

(b) The 'expression' type

Figure 4: Diagram representation of the DSL parser

In order to detect subgraph isomorphism, we need a means of expressing the signature graph. Figure 4a depicts a diagram of the signature domain-specific language (DSL). Appendix C provides a concrete example. The round boxes denote a keyword, whereas the square boxes denote a data type. New graph nodes are generated through the *expression* data type (Figure 4b). The IDENTIFIER keyword allows one to specify a friendly name for the signature. The VARIANT keyword enforces the creation of a new empty DFG. Subsequent expressions are added to this graph, thus, allowing one to specify multiple variants of a signature. Subgraph isomorphism detection is ultimately performed with all variants. The *label* data type is an optional field. It allows the node to be referenced by another expression, enabling node sharing between expressions. Analogous to DFGs generated from assembly instructions, a DFG declared in the DSL is also subject to normalization by the broker (Section 5), and purging (Section 7). In case the TRANSIENT keyword is specified, the node generated from the expression is considered to be non-essential, and may be removed during the purging process (i.e. in case it was translated by the broker).

Figure 4b depicts the *expression* data type. It is recursively defined, and hence allows for nested subexpressions. The '+' keyword denotes the addition of two or more subexpressions. '<<' / '>>' denote a left and right shift, respectively. The *label* data type is a reference to a previously defined graph node. The *literal* data type denotes a constant value. The STORE, LOAD, XOR, OR, AND, MULT and ROTATE keywords followed by subexpressions contained in parentheses provoke creation of a new graph node. The subexpressions serve as input nodes. Finally, the OPAQUE keyword signifies a special wildcard node. A comparison with a node of any other type by the subgraph-isomorphism algorithm always yields true. The opaque node type can have any number of input nodes, including zero. The optional *clamp-label* data type allows one to assign a name to the node type. Consequently, a comparison with a node of any other type yields true, with the added restriction that all opaque nodes carrying the same type label must map to nodes of the same type. We refer to this practice as *type clamping*.

Within the realm of identifying unknown primitives, a special wildcard applicable to a *group* of nodes would be useful. However, to our knowledge, the nature of subgraph-isomorphism does not allow for the augmentation of any such algorithm to support one-to-many mappings. Alternatively, one may declare several variants of a signature, where for each variant, the wildcard group is denoted by a different number of nested opaque operations, i.e. `OPAQUE`, `OPAQUE(OPAQUE)`, etc. This way, any group consisting of a finite number of operations can be expressed. Introducing a notation triggering the translation to multiple variants automatically has been considered. However, as the number of signature variants grows exponentially in the usage count of this hypothetical notation, we prefer to discourage its use. Hence, we omit the notation altogether, enforcing explicit declaration of multiple variants.

9 Subgraph isomorphism

Subgraph isomorphism is a well-documented problem, and is known to be NP complete. The solution proposed by Ullmann [66] is a recursive backtracking algorithm with pruning. Our framework implements this algorithm, with added support for type clamping (see Section 8). For further details about Ullmann's algorithm and the optimizations we applied to it, we refer the reader to the documentation included with our framework's source code.

10 Signatures

Before diving into the practical performance evaluation, we highlight the signatures used throughout the analysis, along with relevant details and a motivation as to why they are included. All signature definition files are included in our implementation of the framework. The list given below should

not be interpreted as an attempt to cover the entirety of cryptographic primitives in existence. Rather, they showcase the applicability of our framework. The selection of signatures was made with a strong focus on proprietary algorithms in embedded environments. As such, they consist of symmetric and unkeyed primitives only, although there is no fundamental incompatibility with asymmetric primitives. To our knowledge, no proprietary primitive exists to date that is studied in the scientific literature and does not fall within any of the classes covered in this section.

However, should an additional signature be desired, then it can be crafted. In broad terms, the approach is to formulate the primitive's defining properties, translate those to an abstract DFG, and finally into a signature definition expressed in the DSL. The process is somewhat ad-hoc in nature. However, the examples presented this section should provide sufficient guidance.

10.1 AES, MD5, XTEA, SHA1

Despite this paper's strong focus on unknown primitives, and hence generic signatures, algorithm-specific signatures, such as AES, MD5, XTEA and SHA1, can be defined and used. Doing so allows us to directly compare results with [43], and demonstrate that our approach effectively solves the code fragment selection problem without resorting to heuristics.

10.2 Feistel cipher

A Feistel cipher is a symmetric structure used in many block ciphers, including DES. In a Feistel cipher, a plaintext block P is split in two pieces L_0 and R_0. Then, for each round $i \in [0, 1, \ldots, n]$,

$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus F(R_i, K_i),$$

is computed, where $\oplus$ denotes bitwise exclusive-or, F the round function, and K_i the sub-key for round i. Translating this definition into a DFG yields the graph shown in Figure 5.

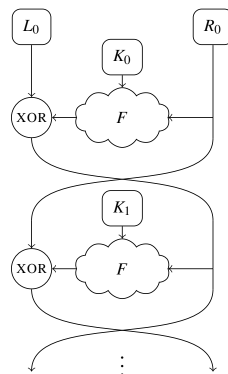


Figure 5: DFG of a Feistel structure

The next step is to construct a signature that represents the DFG from Figure 5. However, F is an algorithm-specific set of operations, of which thus no properties are known. The `OPAQUE` operator (see Section 8), only covers a single operation, whereas F consists of an unknown number of operations. F is known to take R_i and K_i as an input, where $i \in [0, 1, \ldots, n]$. No properties are known for K_i. Hence, we represent F by introducing multiple variants of the signature. In the first variant, we substitute F with `OPAQUE`(R_i), in the second with `OPAQUE`$($`OPAQUE`$(R_i))$, etc., until we reach 8 levels of nested operations. Thus, the signature identifies Feistel ciphers with an F whose input/output relation contains between 1 and 8 successive operations.

10.3 (Non-)Linear feedback shift register

(Non-)Linear feedback shift registers ((N)LFSRs) are often used in pseudo-random number generators, and key-stream generators for stream ciphers. When designed carefully, an (N)LFSR offers relatively strong randomness, whilst requiring very few logic gates, often making it an attractive choice for algorithms used in embedded devices. Both hardware and software implementations of (N)LFSRs are common.

Let R be an (N)LFSR. For each round, a new bit is generated using feedback function L from (a subset of) the bits in R. If L is linear, for e.g. an exclusive-or over the input bits, we refer to R as an LFSR. Conversely, R is an NLFSR if L is non-linear. All bits in register R are shifted one position to the left, discarding the most significant bit, and the newly generated bit is placed at position 0. Furthermore, an output bit is generated by feeding R to some function F. Hence, we have, for each round $i \in [0, 1, \ldots, n]$,

$$R_{i+1} = (R_i << 1) \mid L(R_i)$$
$$\text{output}_i = F(R_i),$$

where $<< x$ denotes a left shift by x bits and $\mid$ denotes bitwise or.

Figure 6 depicts a translation of the above into a DFG. In order to express this graph in a signature, we replace L and F with `OPAQUE` operators. The property that R_{i+1} depends on R_i via L is lost. However, the signature remains distinctive enough in order to warrant very few false positives (see Section 11).

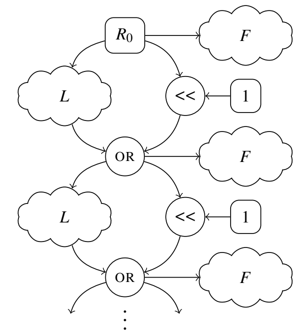


Figure 6: DFG of an (N)LFSR

10.4 Sequential Block Permutation

Variable-length primitives constructed from fixed-length ones are a common phenomenon. For e.g., all hash functions built on the Merkle-Damgård construction, such as MD5, SHA1 and SHA2, have this characteristic. Other examples include block ciphers in a chaining mode of operation. We refer to this concept as a *sequential block permutation*.

Let H_i be the ith output block of a sequential block permutation function, B_i be the ith input block, c be the fixed-length compression function, for $i \in [0, 1, \ldots, n]$. I denotes the initialization vector. Then, we define the sequential block permutation as:

$$H_0 = c(I, B_0)$$
$$H_i = c(H_{i-1}, B_i) \qquad \forall_i \in [1, n]$$

A DFG representation is given in Figure 7. On inspection, we find that it only provides structural guidance, and does not prescribe any arithmetic or logic operations. The definition of H prescribes that compression function c takes two inputs:

(i) The output of its preceding instance, except for the first instance, which depends on the IV.

(ii) Any of the input blocks $B_0, B_1, \dots, B_n$.

In order to express this in a signature definition, we may opt for an approach similar to how the Feistel cipher signature definition is constructed. However, Figure 7 does not contain any operation that serves as an 'anchor point' for c, analogous to the XOR-operation in the Feistel structure. As such, any pattern of repeated operations satisfies property (i), which is overtly generic. Hence, we must also take property (ii) into account. Let c_i be the ith instance of c. The number of arithmetic/logical operations on the path between c_{i-1} and c_i need not be related to that of the path between input block B_i and c_i. Therefore, in order to translate c into multiple variants of the signature, we have to perform a translation for both paths independently. Note that the number of variants grows exponentially in the number of translations. On top of that, the compression function c can be vastly more complex than a round function in a Feistel cipher. For e.g., the MD5 compression function in itself consists of 64 rounds. Therefore, the upper bound of the number of operations that c may consist of is an order of magnitude higher than what one would typically find in a Feistel cipher's round function. All in all, the number of signature variants, and therewith the running time of the analysis, becomes prohibitively large.

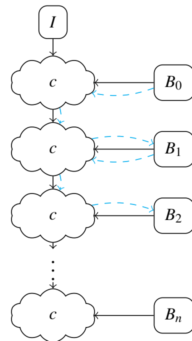


Figure 7: DFG of a sequential block permutation. The blue arrows depict the visitation order by the classifier

Fortunately, there is no need to restrict ourselves to subgraph isomorphism as a means of identifying primitives. Rather, we can apply any algorithm to the DFGs generated by the graph construction framework, which is our approach for the sequential block permutation use case. We take several observations into account. First, input blocks $B_0, B_1, \dots, B_n$ are typically loaded from a memory address. Second, c has a fixed (unknown) block size, and thus we can safely assume that the offsets between the load addresses of B_i, B_{i+1} and B_{i+2} are constant. We take the following approach:

(i) We identify all nodes representing LOAD(ADD(x,k)), where x is an arbitrary graph node, and k is a constant. For each instance of x, we construct a list of tuples (v_0, v_1, v_2), where v_i represents LOAD(ADD(x,k_i)). A tuple is valid only if $k_1 - k_0 \geq 16 \wedge k_1 - k_0 = k_2 - k_1$, i.e. the offsets between v_0, v_1 and v_2 are constant, and at least 16 bytes. As such, a DFG generated from a sequential block permutation function yields at least one tuple such that v_i maps to B_i, for all $i \in [0,1,2]$.

(ii) For all tuples, we determine the shortest path between v_0 and v_1. This can be done by means of a simple breadth-first search. If v_0 maps to B_0 and v_1 to B_1, then this path should take us through two instances of c (see Figure 7).

(iii) Suppose that such a path exists, then we would like to confirm that a similar path exists between v_1 and v_2. We take v_1 as a starting point, and traverse paths with edge directions and node types resembling those on the path between v_0 and v_1. Once such a path has been found, it should reach v_2. Satisfaction of this property is a strong positive indicator.

(iv) To gain more certainty, we also verify that the node types of all inputs and outputs for all the nodes on both paths match. However, in case v_0 maps to B_0, some inputs may originate from the IV, whereas they originate from computed values during the second round. Therefore, we treat constants and inputs of type LOAD as wildcards in this step.

11 Experimental evaluation

We evaluate our solution's performance with regards to accuracy and running time on the following four test sets: (a) the sample set used in [43], (b) a collection of shared libraries and executables part of the OpenWRT[2] network equipment firmware, (c) a collection of proprietary cipher implementations built from public sources, and (d) a collection of real-world embedded firmwares (PLCs, ECUs). The evaluation is conducted on an AMD Ryzen 3600 machine with 16 GB of RAM, which is considered mid-range hardware nowadays.

While not containing proprietary cryptography, the OpenWRT project is publicly available without legal issues around redistribution, contrary to firmwares which do. As such, this evaluation benefits the reproducibility of our work, as well as demonstrates the general principle, accuracy and performance on a test set representative of high-end embedded device firmware. Given the uncertainty over the legality of redistribution, we refer to the original sources of the proprietary cipher implementations rather than publish our binary test set. Due to copyright restrictions, we unfortunately lack permission to publish the real-world embedded firmwares.

Section 6.1.1 defines a tunable variable n, the target number of instances of an algorithm contained within a DFG. The value chosen for n should be low as it correlates with the size of the constructed DFGs, and hence running time, but high enough so that all signatures listed in Section 10 can be identified. The algorithm-specific and Feistel classifiers only target a single instance of an algorithm, and hence are not affected by n. Conversely, the (N)LFSR and sequential block

[2] https://openwrt.org/docs/techref/targets/mvebu

Signature	Compiler	-O0 / Debug	-O1	-O2 / Release	-O3
XTEA 4 rounds 70 vertices	GCC	ok (1ms)	ok (2ms)	ok (2ms)	ok (2ms)
	Clang	ok (1ms)	ok (2ms)	ok (2ms)	ok (2ms)
	MSVC	ok (1ms)	-	ok (2ms)	-
MD5 64 rounds 458-618 vertices	GCC	ok (267ms)	ok (335ms)	ok (345ms)	ok (348ms)
	Clang	ok (286ms)	ok (241ms)	ok (272ms)	ok (265ms)
	MSVC	ok (269ms)	-	ok (322ms)	-
AES 1 round 85-110 vertices	GCC	ok (64ms)	ok (61ms)	ok (53ms)	ok (56ms)
	Clang	ok (37ms)	ok (32ms)	ok (32ms)	ok (27ms)
	MSVC	ok (30ms)	-	ok (42ms)	-

Table 1: Signature matching step execution times, sample set of Lestringant et al.

permutation classifiers are, as they identify a primitive based on multiple instances. The latter identifies two successive instances of some unknown compression function c. Because the rewrite rules are designed to promote numeric simplification (Section 5), the initialization and finalization step of an algorithm may become merged with the first and last instance of c, respectively. Thus, by choosing $n = 4$, the presence of two successive instances of c in the DFG is warranted. Choosing a value beyond 4 clearly does not offer any advantages regarding this property. Furthermore, identifying 4 successive rounds of an (N)LSFR in a DFG produced from code that does not actually implement one is highly unlikely. Therefore, for the remainder of this section, we take $n = 4$.

11.1 Comparison with Lestringant et al.

Lestringant et al. [43] showcase the effectiveness of their method by successfully identifying AES, MD5 and XTEA in binary files. Unfortunately, their sample set was never published, and is compiled for x86, which our implementation currently does not support. Therefore, we constructed a new sample set for the ARM architecture that is as faithful as possible to theirs. The algorithms are taken from the cited sources[3],[4],[5], and subsequently compiled with GCC 9.3.0, Clang 9.0.8, and MSVC 19.16 on all available optimization levels (O0–O3, debug/release). We use algorithm-specific signatures in order to warrant a fair comparison. The results are depicted in Table 1. They show that all samples are identified successfully by (a variant of) their corresponding signatures, regardless of compiler and optimization level. This effectively demonstrates that our approach is equally capable of identifying these algorithms, without resorting to heuristics for fragment selection.

11.2 Performance on OpenWRT binaries

The version of OpenWRT used is 19.07.2, which is the latest at time of writing. The sample set consists of several binaries taken from the distribution and is known to contain cryptographic primitives.

DFG construction from binary code (Section 5) is a special case of execution, and is thus affected by the *halting* problem. As such, graph-construction is not guaranteed to terminate. Therefore, we introduce a graph construction timeout t_{timeout}. Figure 8a depicts a histogram of graph construction time t for all graphs constructed during the analysis of libcrypto.so.1.1. It shows that, for the vast majority of all graphs, construction completes within $10s$. Thus, we take $t_{\text{timeout}} = 10s$.

Furthermore, we must decide what action to take when the function under analysis invokes another function. Either we perform inlining, and hence incorporate the entire invocation in the resulting DFG, or we represent it by a single `CALL` operation. To address this issue, we define a tunable variable d, denoting the depth level to which function calls are inlined. We investigate the impact of d by running the analysis on *libcrypto.so.1.1*, while taking on different values, and measuring performance in terms of running time and accuracy. We then choose a sensible value based on a trade-off between the two, and use it for the remainder of this section. Figure 8b depicts the time taken to complete the entire analysis pipeline over every function in libcrypto.so.1.1, under the influence of d. Figure 8c contains accuracy measurements for each signature. True negatives are omitted since they cover an overwhelming majority of results, and thus impact readability.

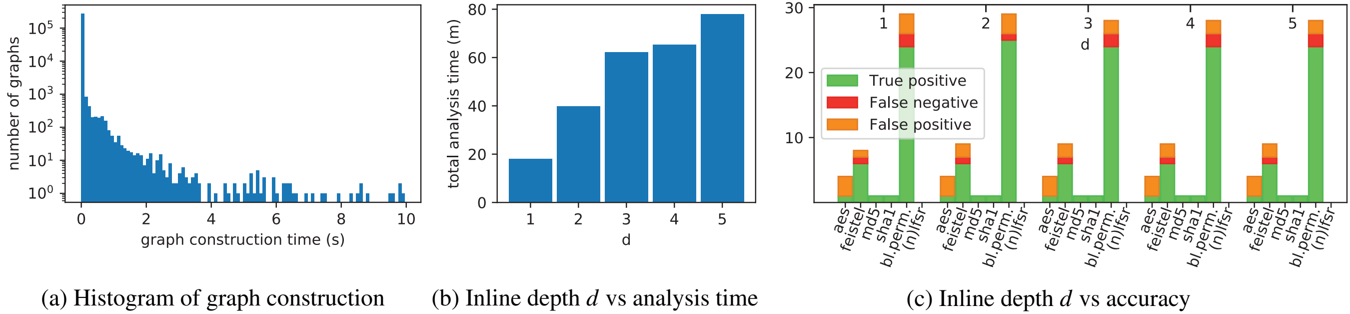
Recall that the signature evaluation is performed on graphs, and the graph construction step may yield several graphs. As such, several signature evaluation results may exist per function. The measurements provided in Figure 8c are aggregated on a per-function level.

Let f be any function in the binary under analysis, and let signature s_α denote a signature targeting primitive α. Furthermore, let F be the set of DFGs generated from f during the graph construction phase. Finally, $\text{match}(s_\alpha, G)$ indicates that signature s_α was identified in graph G, $\text{imp}(f,\alpha)$ denotes that f implements cryptographic primitive α.

A result is marked as a *true positive* if $\text{imp}(f,\alpha) \wedge \exists_G.G \in F \wedge \text{match}(s_\alpha, G)$, i.e. f implements cryptographic primitive α, and its signature is found in *at least one* graph in F. Indeed, there is no guarantee that all DFGs in F contain algorithm α, and hence it is expected that the signature is not found in every graph in F. A result is marked as a *false positive* if $\neg\text{imp}(f,\alpha) \wedge \exists_G.G \in F \wedge \text{match}(s_\alpha, G)$, i.e. f does not implement primitive α, yet its signature is found in at least one graph in F. A result is marked as a *true negative* if $\neg\text{imp}(f,\alpha) \wedge \neg\exists_G.G \in F \wedge \text{match}(s_\alpha, G)$. A result is a *false negative* if $\text{imp}(f,\alpha) \wedge \neg\exists_G.G \in F \wedge \text{match}(s_\alpha, G)$.

The results in Figure 8c show that accuracy does not substantially improve when choosing $d > 2$. However, doing so does impact the running time. We conclude that, for libcrypto.so.1.1, $d = 2$ is a reasonable trade-off between accuracy and running time. As such, we continue to use $d = 2$ for the remainder of this section, unless specified otherwise.

[3] https://en.wikipedia.org/w/index.php?title=XTEA
[4] https://tools.ietf.org/html/rfc1321
[5] https://github.com/BrianGladman/AES



(a) Histogram of graph construction (b) Inline depth d vs analysis time (c) Inline depth d vs accuracy

Figure 8: Effect of inline depth d and t_{timeout} for libcrypto.so.1.1

At this point, sensible values for n, d and t_{timeout} have been selected. We continue the evaluation by feeding the entire set of OpenWRT binaries to our analysis framework. The results are listed in Table 2. Each cell in the table depicts the symbol name in the corresponding binary of the first positive result, or, in case of a false negative, the symbol name where a positive result is expected. The results indicate our solution is capable of successfully identifying the vast majority of cryptographic primitives present in various binaries in a timely manner. Should accuracy take precedence over performance, it is possible to tune the parameters to improve detection.

Algorithm signature	dropbear	libcrypto.so.1.1	libmbedcrypto.so.2.16.3[1]	libnettle.so.7.0[2]
size	145 KB	1,735 KB	197 KB	237 KB
analysis time	6m44s	39m47s	6m56s	11m32s
SHA1				
sha1	✓ Unlabeled[3]	✓ SHA1_Update	✓ sha1_update_ret	✓ sha1_compress
bl.perm.	✓ Unlabeled[3]	✓ SHA1_Update	✓ sha1_update_ret	✓ sha1_update[4]
SHA256				
bl.perm.	✓ Unlabeled[3]	✓ SHA256_Update[5]	✓ sha256_update_ret	✓ sha256_update[4,5]
AES				
aes	✓ Unlabeled[3]	✓ AES_encrypt	✓ aes_encrypt	✓ aes_encrypt_armv6
MD4				
bl.perm.	N/A	✓ MD4_Update	N/A	✓ md4_update[4]
MD5				
md5	N/A	✓ MD5_Update	✓ md5_update_ret	✓ hmac_md5_update
bl.perm.	N/A	✓ MD5_Update	✓ md5_update_ret	✓ hmac_md5_update
RIPEMD160				
bl.perm.	N/A	✓ RIPEMD160_Update	N/A	✓ hmac_ripemd160_update
SHA512				
bl.perm.	N/A	✓ SHA512_Update[5]	✓ sha512_process[5]	✓ sha512_update[5]
SM3				
bl.perm.	N/A	✓ sm3_block_data_order	N/A	N/A
BLOWFISH				
feistel	N/A	✓ BF_encrypt	✓ blowfish_crypt_ecb[4]	✓ blowfish_encrypt
CAMELLIA				
feistel	N/A	✓ Camellia_EncryptBlock	N/A	✓ camellia_crypt
CAST				
feistel	N/A	✓ CAST_ecb_encrypt	N/A	✓ cast128_encrypt
DES				
feistel	N/A	✓ DES_encrypt2	N/A	✓ des_encrypt
RC2				
feistel	N/A	✗ RC2_encrypt	N/A	N/A
SEED				
feistel	N/A	✓ SEED_encrypt	N/A	N/A
SM4				
feistel	N/A	✓ SM4_encrypt	N/A	N/A
GOST				
feistel	N/A	N/A	N/A	✓ gosthash94_digest
MD2				
bl.perm.	N/A	N/A	N/A	✓ md2_update
TWOFISH				
feistel	N/A	N/A	N/A	✗ twofish_encrypt
SHA3				
bl.perm.	N/A	✓ SHA3_absorb	N/A	✓ sha3_update[4]

1 Symbols prefixed with `mbedtls_`
2 Symbols prefixed with `nettle_`
3 Misclassified by IDA as an integer array. Manual cast to function required.
4 Positive match for $d \geq 4$.
5 Positive match for $t_{\text{timeout}} \geq 30s$.

Table 2: Analysis result for various binaries in OpenWRT

11.2.1 Discussion of invalid results

Table 2 and Figure 8c contain several false positives and false negatives. In order to gain insights in the limitations of our approach, we highlight those instances here.

False negatives RC2 uses a regular addition, i.e. with carry over, rather than XOR, whereas the Feistel signature highlighted in Section 10.2 relies on the XOR operation being present. Therefore, RC2 is not identified as a Feistel cipher.

Furthermore, SHA512 is consistently among the false negatives for the sequential block permutation class of primitives. This is due to a DFG consisting of n (i.e. 4) instances of SHA512 being required for successful identification. However, said DFG consists of over 1,000,000 vertices, and causes the construction phase to exceed t_{timeout}. Increasing this value successfully mitigates the issue. However, it also affects the total analysis time. The exact same issue applies to SHA3 with $d \geq 3$, causing the Keccak-F function to be inlined, and consequently the construction to exceed t_{timeout}.

Twofish is a Feistel cipher with a complex round function. The Feistel signature used throughout the analysis supports a round function consisting of up to 8 consecutive arithmetic/logical operations, whereas the complexity of the Twofish round function goes beyond that. Unfortunately, extending the signature beyond 8 consecutive operations severely impacts the running time of our implementation.

False positives The AES key schedule is identified as a Feistel network. This is due to the fact that its structure can actually be formulated as one, i.e. each round $L_{i+1} = R_i$, and $R_{i+1} = L_i \oplus F(R_i, K_i)$, where i denotes the round number for some function F. This is a perfect example to illustrate that the taxonomical tree of cryptographic primitives is not necessarily clear-cut. Rather, a degree of ‘fuzziness‘ exists among different classes.

RC4 and ChaCha, both stream ciphers, are identified as sequential block permutations. Inspection reveals that both implementations keep an internal state of some size b. The state is used directly as the cipher's keystream. After the internal state is fully consumed, a new internal state is generated. As such, the structure can be viewed as a special case of a block cipher with a block size of b bytes.

Algorithm	Type	Description	Reverse-engineered	Cryptanalysis	Original source	Target signature
CRYPTO1	Stream	Cipher used in the Mifare Classic family of RFID tags.	[32, 54]	[20, 25, 32, 33, 49]	6	✓ (N)LFSR[1]
HITAG2	Stream	Cipher used in vehicle immobilizers.	[72]	[22, 59, 60, 62, 68]	7	✓ (N)LFSR[1]
A5-1	Stream	Provides over-the-air privacy for communication in GSM.	[16]	[6, 10, 48]	8	✓ (N)LFSR[1]
A5-2	Stream	GSM export cipher.	[16]	[34]	8	✓ (N)LFSR[1]
A5-GMR	Stream	Cipher used in GMR, a standard for satellite phones. Heavily inspired by A5/2.	[26]	[26, 27]	9	✓ (N)LFSR[1]
RED PIKE	Block	Classified UK government encryption algorithm.	[23]	-	10	✗ Feistel cipher
COMP128	Hash	Family of algorithms used for session key and MAC generation in GSM.	[15, 63]	[17]	11	✓ Block permutation
KASUMI	Block	Feistel cipher used for the confidentiality and integrity of 3G.	-	[8, 28, 41]	12	✓ Feistel cipher
MULTI2	Block	A block cipher used for broadcast scrambling in Japan.	-	[2]	13	✓ Feistel cipher
DST40	Block	Digital Signature Transponder cipher, often found in vehicle immobilizers.	[14]	[14]	14	✓ (N)LFSR
KEELOQ	Block	Block cipher used in remote keyless entry systems and home automation.	[51]	[7, 12, 21, 29]	15,16	✓ (N)LFSR

[1] Positive match for $d \geq 4$

Table 3: Analysis result for proprietary samples

Algorithm signature	CWM0576	CWX0470	M340	VW
size	1,717 KB	1,344 KB	4,133 KB	512 KB
analysis time	88m14s	45m53s	83m11s	11m45s
DES				
feistel	✓ Match	✓ Match	N/A	N/A
AES				
aes	✓ Match	N/A	N/A	N/A
bl.perm.	✓ Match	N/A	N/A	N/A
MD5				
md5	✓ Match	✓ Match	✓ Match	N/A
bl.perm.	✓ Match	✓ Match	✓ Match	N/A
MEGAMOS				
(n)lfsr	N/A	N/A	N/A	✗ No match

Table 4: Analysis result for various firmware images

Finally, CAST, ARIA and SM4 are all misidentified as AES. This is due to the fact that for all three primitives, either the algorithm itself, or its key schedule, is implemented by means of lookup tables in a fashion similar to that of AES. Ultimately, the transform completely depends on these tables, rather than information flows.

11.3 Performance on proprietary algorithms

Next, we turn our attention to various proprietary algorithms. Most algorithms were originally confidential, but have been leaked to the public or reverse engineered. As such, source code for all samples is publicly available. Due to uncertainty over the legality of redistribution, we point to the original sources for reference. Table 3 depicts the analysis results these algorithms. A description, the analysis result, and other relevant information is condensed into a single table due to space restrictions. All signatures target a generic class of primitives and none were pre-constructed to fit a particular sample. All algorithms are successfully identified, with the exception of Red Pike. Similar to RC2 from Section 11.2.1, Red Pike uses addition instead of exclusive-or, and is therefore not identified as a Feistel cipher.

Finally, the test set of representative real-world firmwares consists of images for the Emerson ControlWave Micro RTU, Emerson ControlWave XFC flow computer, Schneider Electric M340 PLC and Volkswagen IPC. The size, nature and complexity of these images ensure test-set realism. Table 4 depicts the analysis result for all the firmwares. To the best of our knowledge, the table covers all cryptographic algorithms present in the sample set of firmware images. The images are 'flat' binaries and hence symbol names are absent. The results show that all the cryptographic primitives were identified, except for the Megamos cipher. Verdult et al. [69] revealed that the Megamos cipher contains an NLFSR, and thus, the analysis should point this out. Further examination reveals that the non-linear feedback function is implemented as a subroutine, and the shift register is updated depending on its return value via an if-statement. This is a direct violation of the implicit flow limitation inherent to DFG-based approaches discussed in Section 2.

12 Conclusions

Despite the ubiquitous availability of royalty-free, publicly documented, and peer-reviewed cryptographic primitives and implementations, proprietary alternatives have persisted across many industry verticals, especially in embedded systems. Due to the undocumented and proprietary nature of said primitives, subjecting them to security analysis often requires locating and classifying them in often very large binary images, which is a time-consuming, labor-intensive effort.

In order to overcome this obstacle in an automated fashion, a solution should have the capability of identifying as-of-yet unknown cryptographic algorithms, support large, real-world firmware binaries, and not depend on peripheral emulation.

[6] https://github.com/nfc-tools/mfcuk/blob/master/src/crypto1.c
[7] http://cryptolib.com/ciphers/hitag2/
[8] https://cryptome.org/gsm-a512.htm
[9] https://github.com/marcelmaatkamp/gnuradio-osmocom-gmr/blob/master/src/l1/a5.c
[10] https://en.wikipedia.org/wiki/Red_Pike_(cipher)
[11] https://github.com/osmocom/libosmocore/blob/master/src/gsm/comp128.c
[12] https://github.com/osmocom/libosmocore/blob/master/src/gsm/kasumi.c
[13] https://github.com/OP-TEE/optee_os/blob/master/core/lib/libtomcrypt/src/ciphers/multi2.c
[14] https://github.com/jok40/dst40/blob/HEAD/software/dst40test/dst40.c
[15] https://github.com/hadipourh/KeeLoq
[16] http://cryptolib.com/ciphers/keeloq/

As of yet, no prior work exists that satisfies these criteria.

Our novel approach combines DFG isomorphism with symbolic execution, and introduces a specialized DSL in order to enable identification of unknown proprietary cryptographic algorithms falling within well-defined taxonomical classes. The approach is the first of its kind, is architecture and platform agnostic, and performs well in terms of both accuracy and running time on real-world binary firmware images.

Future work DFGs do not allow for the expression of code flow information. Potentially valuable indicators, such as whether two nodes originate from the same execution address, hinting to a round function, are therefore lost. We leave the incorporation of code flow information for future work.

13 Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CaSa - 390781972.

References

[1] Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.

[2] Jean-Philippe Aumasson, Jorge Nakahara, and Pouyan Sepehrdad. Cryptanalysis of the isdb scrambling algorithm (multi2). In *International Workshop on Fast Software Encryption*, pages 296–307. Springer, 2009.

[3] Luigi Auriemma. Signsrch tool. *tool for searching signatures inside files*, 2013.

[4] Roberto Avanzi. A salad of block ciphers. *IACR Cryptology ePrint Archive*, 2016:1171, 2016.

[5] BBC News. Car key immobiliser hack revelations blocked by uk court. 2013. https://www.bbc.com/news/technology-23487928.

[6] Eli Biham and Orr Dunkelman. Cryptanalysis of the a5/1 gsm stream cipher. In *International Conference on Cryptology in India*, pages 43–51. Springer, 2000.

[7] Eli Biham, Orr Dunkelman, Sebastiaan Indesteege, Nathan Keller, and Bart Preneel. How to steal cars a practical attack on keeloq. In *EUROCRYPT*, pages 1–18, 2008.

[8] Eli Biham, Orr Dunkelman, and Nathan Keller. A related-key rectangle attack on the full kasumi. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 443–461. Springer, 2005.

[9] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[10] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of a5/1 on a pc. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2000.

[11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *Proceedings of the 26th USENIX Security Symposium*, pages 643–659, 2017.

[12] Andrey Bogdanov. Cryptanalysis of the keeloq block cipher. *IACR Cryptology ePrint Archive*, 2007:55, 2007.

[13] Wouter Bokslag. An assessment of ecm authentication in modern vehicles.

[14] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled rfid device. In *USENIX Security Symposium*, volume 31, pages 1–16, 2005.

[15] Marc Briceno, Ian Goldberg, and David Wagner. An implementation of comp128. 1998. http://www.iol.ie/kooltek/a3a8.txt.

[16] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of the gsm a5/1 and a5/2 "voice privacy" encryption algorithms. *Originally published at http://www. scard. org, mirror at http://cryptome.org/gsm-a512. htm*, 26, 1999.

[17] Billy Brumley. A3/a8 & comp128. *T-79.514 Special Course on Cryptology*, pages 1–18, 2004.

[18] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634, 2009.

[19] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 169–182, 2012.

[20] Nicolas T Courtois. The dark side of security by obscurity and cloning mifare classic rail and building passes, anywhere, anytime. 2009.

[21] Nicolas T Courtois, Gregory V Bard, and David Wagner. Algebraic and slide attacks on keeloq. In *International*

Workshop on Fast Software Encryption, pages 97–115. Springer, 2008.

[22] Nicolas T Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the hitag2 stream cipher. In *International Conference on Information Security*, pages 167–176. Springer, 2009.

[23] Gmane Cypherpunk mailing list. Red pike cipher. 2004. http://permalink.gmane.org/gmane.comp.security.cypherpunks/3680.

[24] Robin David. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes*. PhD thesis, 2017.

[25] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D Garcia. A practical attack on the mifare classic. In *International Conference on Smart Card Research and Advanced Applications*, pages 267–282. Springer, 2008.

[26] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. Don't trust satellite phones: A security analysis of two satphone standards. In *2012 IEEE Symposium on Security and Privacy*, pages 128–142. IEEE, 2012.

[27] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. An experimental security analysis of two satphone standards. *ACM Transactions on Information and System Security (TISSEC)*, 16(3):1–30, 2013.

[28] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the kasumi cryptosystem used in gsm and 3g telephony. In *Annual cryptology conference*, pages 393–410. Springer, 2010.

[29] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme. In *Annual International Cryptology Conference*, pages 203–220. Springer, 2008.

[30] ETSI. 300 392-7 v3. 3.1 (2012-07) european standard (telecommunication series) terrestrial trunked radio (tetra); voice plus data (v+ d); part 7: Security. *European Telecommunications Standards Institute (ETSI)*, 2012. https://www.etsi.org/deliver/etsi_en/300300_300399/30039207/03.03.01_60/en_30039207v030301p.pdf.

[31] Peter Garba and Matteo Favaro. Saturn-software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 27–38, 2019.

[32] Flavio D Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter Van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling mifare classic. In *European symposium on research in computer security*, pages 97–114. Springer, 2008.

[33] Flavio D Garcia, Peter Van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a mifare classic card. In *2009 30th IEEE Symposium on Security and Privacy*, pages 3–15. IEEE, 2009.

[34] Ian Goldberg, David Wagner, and Lucky Green. The real-time cryptanalysis of a5/2. *Rump session of Crypto*, 99:16, 1999.

[35] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection*, pages 41–60, 2011.

[36] Ilfak Guilfanov. Findcrypt2, february 2006. http://www.hexblog.com/?p=28.

[37] Peter Gutmann. *Cryptographic security architecture: design and verification*. Springer Science & Business Media, 2003. pages 293.

[38] Gregory D Hill and Xavier JA Bellekens. Deep learning based cryptographic primitive classification. *arXiv preprint arXiv:1709.08385*, 2017.

[39] Liam Timothy Keliher. *Linear cryptanalysis of substitution-permutation networks*. Queen's University, 2003.

[40] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX:5–83, 161–191, 1883.

[41] Jongsung Kim, Seokhie Hong, Bart Preneel, Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key boomerang and rectangle attacks. *IACR Cryptology ePrint Archive*, 2010:19, 2010.

[42] Philippe Lagadec. Balbuzard, 2014. http://www.decalage.info/en/python/balbuzard.

[43] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 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*, pages 203–214. ACM, 2015.

[44] Literatecode. Draft crypto analyzer (draca). http://www.literatecode.com/draca, May 2013.

[45] Loki. Snd crypto scanner (olly/immunity plugin), 2008. https://web.archive.org/web/20080321134709/http://tuts4you.com/forum/index.php?showtopic=15447.

[46] Charalampos Manifavas, George Hatzivasilis, Konstantinos Fysarakis, and Yannis Papaefstathiou. A survey of lightweight stream ciphers for embedded systems. *Security and Communication Networks*, 9(10):1226–1246, 2016.

[47] Felix Matenaar, Andre Wichmann, Felix Leder, and Elmar Gerhards-Padilla. Cis: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 46–53. IEEE, 2012.

[48] Alexander Maximov, Thomas Johansson, and Steve Babbage. An improved correlation attack on a5/1. In *International Workshop on Selected Areas in Cryptography*, pages 1–18. Springer, 2004.

[49] Carlo Meijer and Roel Verdult. Ciphertext-only cryptanalysis on hardened mifare classic cards. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 18–30, 2015.

[50] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[51] Microchip. Hopping code decoder using a PIC16C56, AN642. 1998. https://web.archive.org/web/20080916043223/http://www.keeloq.boom.ru/decryption.pdf.

[52] Mr Paradox, AT4RE. Hash & crypto detector (hcd), 2009. https://web.archive.org/web/20091203010936/http://www.at4re.com/download.php?view.8.

[53] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[54] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. Reverse-engineering a cryptographic rfid tag. In *USENIX security symposium*, volume 28, 2008.

[55] Karsten Nohl, Erik Tews, and Ralf-Philipp Weinmann. Cryptanalysis of the dect standard cipher. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2010.

[56] Daniel Plohmann and Alexander Hanel. simplifire. idascope, 2012.

[57] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[58] snaker, Maxx. Kanal - krypto analyzer for peid, 2015. http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm.

[59] Mate Soos. Enhanced gaussian elimination in dpll-based sat solvers. In *POS@ SAT*, pages 2–14, 2010.

[60] Petr Štembera and Martin Novotny. Breaking hitag2 with reconfigurable hardware. In *2011 14th Euromicro Conference on Digital System Design*, pages 558–563. IEEE, 2011.

[61] Daehyun Strobel, Benedikt Driessen, Timo Kasper, Gregor Leander, David Oswald, Falk Schellenberg, and Christof Paar. Fuming acid and cryptanalysis: Handy tools for overcoming a digital locking and access control system. In *Annual Cryptology Conference*, pages 147–164. Springer, 2013.

[62] Siwei Sun, Lei Hu, Yonghong Xie, and Xiangyong Zeng. Cube cryptanalysis of hitag2 stream cipher. In *International Conference on Cryptology and Network Security*, pages 15–25. Springer, 2011.

[63] Jos Tamas. Secrets of the sim. 2013. http://www.hackingprojects.net/2013/04/secrets-of-sim.html.

[64] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 3–14, 2019.

[65] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-Ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, pages 1–12, 2018.

[66] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[67] Roel Verdult. *The (in) security of proprietary cryptography*. PhD thesis, [Sl: sn], 2015.

[68] Roel Verdult, Flavio D Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In *Presented as part of the 21st USENIX Security Symposium*, pages 237–252, 2012.

[69] Roel Verdult, Flavio D Garcia, and Baris Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *Supplement to the Proceedings of 22nd USENIX Security Symposium*, pages 703–718, 2015.

[70] Aram Verstegen, Peter Schwabe, Iskander Kuijer, and Roel Verdult. Press to unlock: Analysis, reverse-engineering and implementation of hitag2-based remote keyless entry systems. 2018.

[71] Michael Weiner, Maurice Massar, Erik Tews, Dennis Giese, and Wolfgang Wieser. Security analysis of a widely deployed locking system. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 929–940, 2013.

[72] I.C. Wiener. Hitag2 specification, reference implementation and test vectors, 2007. http://cryptolib.com/ciphers/hitag2.

[73] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, May 2019.

[74] x3chun. Crypto searcher, 2004. https://web.archive.org/web/20050211180634/http://x3chun.wo.to/.

[75] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 442–458, 2018.

[76] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 921–937. IEEE, 2017.

[77] Babak Yadegari. Automatic deobfuscation and reverse engineering of obfuscated code. 2016.

A Path Oracle Policy – an example

```
1  MOV   R4, #0                  ; set R4 to 0
2  _begin:
3  CMP   R4, R8                  ; compare R4 to R8
4  BGE   _end                    ; break loop if R4 >= R8
5  LDRB  R5, [R4, R7]            ; load R7[R4] into R5
6  BL    <keystream_generator>   ; call generator
7  EOR   R5, R0, R5              ; XOR output byte with R5
8  STRB  R5, [R4, R6]            ; store result at R6[R4]
9  ADD   R4, R4, #1              ; increment R4
10 B     _begin                  ; continue at beginning
11 _end:
```

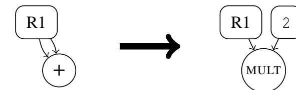
Figure 9: Example stream cipher ARM assembly snippet

Suppose the graph construction is run on the example ARM assembly snippet given in Figure 9. We start with $\mathcal{S} = (G,P,B)$, with $P = \textbf{true}$. Line 4 contains conditional instruction *Branch Greater/Equal* (BGE). During the first visit of this instruction, we have $i = 0$, $P = \textbf{true}$, and $c = (\texttt{R8} \leq 0)$. Since the value of R8 is unknown, c is underdetermined. The path oracle policy prescribes TAKE_BOTH. Thus, we get $P = (\texttt{R8} \leq 0)$, $B_4[0] = \textbf{true}$, and $\mathcal{S}' = (G', P', B')$, with $P' = (\texttt{R8} > 0)$ and $B'_4[0] = \textbf{false}$. For state $\mathcal{S}$, the instruction is evaluated, and thus the construction continues on line 11, and hence terminates. For $\mathcal{S}'$, the instruction is skipped, thereby visiting the body of the loop. Eventually, $\mathcal{S}'$ revisits the instruction at line 4. This time we have $c = (\texttt{R8} \leq 1)$, $i = 1$, $P' = (\texttt{R8} > 0)$ and $B'_4[0] = \textbf{false}$. Since $P' \wedge c$ is underdetermined, we query the path oracle, and obtain TAKE_FALSE, causing another visit of the loop's body. Finally, at $i = n$, we get $c = \texttt{R8} \leq n$ and $P' = (\texttt{R8} > n-1)$. We obtain TAKE_TRUE from the path oracle. Thus, the construction terminates. We obtain two graphs; one corresponding to predicate $\texttt{R8} \leq 0$, and another corresponding to $\texttt{R8} = n$. The latter describes n iterations of the algorithm, exactly conforming to our goal. The former describes zero iterations, and thus, contains a negligible amount of nodes. Therefore, we accept the small amount of overhead this graph induces during later stages of the analysis.

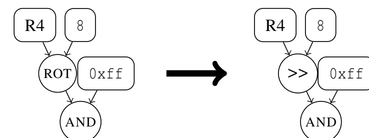
B Miscellaneous rewrite rules

Besides the rewrite rules already described, we apply additional miscellaneous rules. They were conceived through continuous application of our framework to code fragments from various sources, and subsequent stumbling upon variations between the processed result generated from supposedly semantically equivalent code. We highlight these rules below. Different compilers have different optimization strategies. As such, some finetuning of these rules may be necessary when analyzing code produced by a vastly different compiler than those already accounted for.

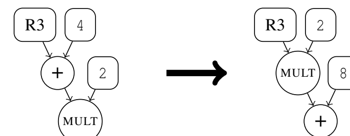
There are various means of doubling the value of an arbitrary expression x. For example, $\texttt{MULT}(x,2)$, but also $\texttt{ADD}(x,x)$ and x<<1. We represent all variants by $\texttt{MULT}(x,2)$.



Furthermore, suppose we have an arbitrary expression x, and constants c_1 and c_2. Then, the results of $\texttt{AND}(x >> c_1, c_2)$ and $\texttt{AND}(\texttt{ROTATE}(x,c_1),c_2)$, are equivalent if $c_2 < 2^{32-c_1}$ and $c_1 < 32$, for a 32-bit architecture. This equivalence is sometimes exploited by compilers. In such a scenario, we represent both variants by $\texttt{AND}(x >> c_1, c_2)$.



Lastly, we distribute multiplications over additions.



C Sample signature definition

Given below is a snippet taken from the (N)LFSR signature bundled with our implementation of the framework.

```
IDENTIFIER (Non-)Linear feedback shift register

VARIANT A
...

VARIANT C
TRANSIENT layer0:OR(AND(1,OPAQUE),OPAQUE<<1);
TRANSIENT layer1:OR(AND(1,OPAQUE),layer0<<1);
TRANSIENT layer2:OR(AND(1,OPAQUE),layer1<<1);
layer3:OR(AND(1,OPAQUE),layer2<<1);
```

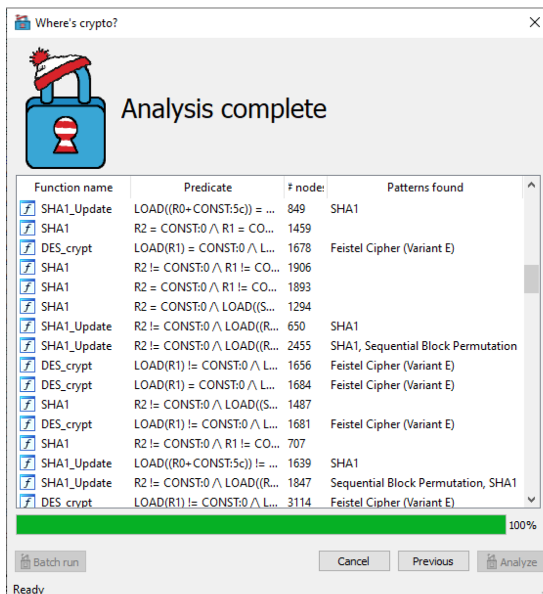
An (N)LFSR can be implemented in software by various means. For e.g., rather than shifting to the left, the register may shift to the right instead, placing the new bit generated by the feedback function at the most significant position. Furthermore, a left shift of one bit is equivalent to a multiplication with 2, or an addition with itself. Also, the newly generated bit is normally appended to the register through a bitwise or. However, directly after a shift operation is performed, the vacant bit is always 0. Hence, using an exclusive-or, or even an addition instead is equivalent. Due to these naturally occurring variations, several variants of the signature are defined. In this example, we take a closer look at variant `C`, which is the most typical.

As discussed in Section 11, we take $n = 4$. Hence, the signature should capture 4 iterations of an (N)LFSR. Each iteration, the register shifts one position to the left, and a new bit is generated by an unknown feedback function L and placed at position 0 by means of a bitwise or. Each round refers to the previous through its label, i.e. `layer[0-3]`. The initial state is the result of an unknown initialization function, hence represented by `OPAQUE`. L is also unknown, and thus represented by `OPAQUE`. However, it is known to produce a single output bit. Therefore, it can be assumed that the single bit is obtained through a bitwise-and with 1, before being inserted into the register by means of a bitwise or. Finally, all iterations except the last form intermediate steps towards the register's final value. By specifying the TRANSIENT keyword, we allow the broker to translate the intermediate steps into a more optimized DFG representation.

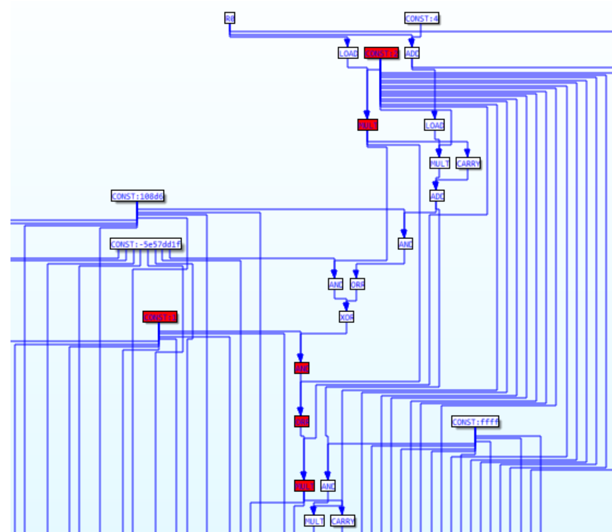
D Implementation

An implementation of the framework described in this paper is available for download[17]. It comes in the form of a plug-in for the popular IDA disassembler. At the time of writing, support is implemented for 32 bit ARM binaries. The architecture is modular, and expanding support to other architectures is relatively straightforward. Figure 10 shows a sample analysis report, and a DFG plot generated by our implementation.

[17] https://github.com/wheres-crypto/wheres-crypto



(a) Sample analysis report



(b) DFG plot generated from assembly, highlighting an LFSR

Figure 10: An impression of the implementation of our framework