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

Carlo Meijer
Institute for Computing and Information Sciences
Radboud University Nijmegen
cmeijer@cs.ru.nl

Veelasha Moonsamy
Institute for Computing and Information Sciences
Radboud University Nijmegen
email@veelasha.org

Jos Wetzels
Midnight Blue Labs
j.wetzels@midnightbluelabs.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. [40]. 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 [40], 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 [34, 37, 61], the use of proprietary cryptography has persisted across many industry verticals ranging from physical access control systems [1, 57, 61, 64, 65, 67] and telecommunications [23, 27, 52] to machine-to-machine authentication [11, 61].

Particularly affected are embedded systems in industries with complicated multi-tiered supply chains where system integrators and OEMs end up integrating a significant number of black-box components, leading to system opacity compounded by the frequent absence of a Software Bill of Materials. Consequently, the use of a proprietary cryptographic primitive high up in the supply chain can end up trickling down in a wide variety of products by vendors essentially unaware they are incorporating it.

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 [61], 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 particulars. To this end, we
utilize a taxonomy based on [4, 36, 43, 47] 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. Various ciphers could be classified as part of multiple taxonomical classes but this presents no problem to our purposes.

Figure 1: Taxonomical tree of algorithm classes

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 [40] are overcome through augmentation with symbolic execution which allows us to specify structural signatures for taxonomic classes of cryptographic primitives and analyze binary code for matches. Additionally, our focus for this paper is on symmetric and unkeyed primitives only.

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. Clear guidelines are provided on using the new DSL along with several examples.

Finally, a free and open-source proof-of-concept (PoC) implementation, Where’s Crypto?, is made available1 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 semantically equivalent and 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 and we can leverage this fact to recognize compiler-generated equivalences.

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 declare this class of information flows out of scope.

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 PoC’s coverage. However, this is not an inherent limitation of our approach but merely of our PoC 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.

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 [61], namely stream- and block ciphers and hash functions, we do not consider this a practical issue.

False positives Certain cryptographic primitives are essentially underdefined. That is to say, their definitions are a subset of other algorithms and code patterns and as such, their matches are prone to false positives. We do not consider this

---

1https://github.com/wheres-crypto/wheres-crypto
a serious practical problem, however, since 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.

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, either inherently or as the result of function inlining, 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) [44], library imports or dedicated instructions (e.g. AES-NI). This approach is inherently incapable of detecting unknown algorithms and as such unsuited for our purposes.

Data signatures The most common approach employed in practice [3, 33, 41, 42, 49, 53, 54, 68] 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). This approach is unsuitable also for detecting known algorithms that do not rely on easily identifiable fixed data or have certain implementations, such as dynamically generating S-Boxes rather than embedding them.

Code heuristics Another series of approaches relies on code heuristics which are applied either statically or dynamically. One set of heuristics are code-augmented data signatures, like mnemonic-constant tuples [32, 39], 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 [16, 32, 39, 44, 53] 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. [35] have proposed 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 [17, 32, 44, 50]. One approach here 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 [69] utilizes dynamic instrumentation and symbolic execution to translate candidate cryptographic algorithms into boolean formulas for subsequent comparison to reference implementations using guided fuzzing. However, the reliance of this approach on dynamic instrumentation and its inherent inability to recognize unknown algorithms render it unsuitable for our purposes.

Finally, there is the DFG isomorphism approach as proposed by [40] which extracts DFGs from a given binary and matches it against graphs of known cryptographic algorithms through the use of Ullmann’s subgraph isomorphism algorithm [60]. 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 [40] 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 any sort of 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 once the algorithm is known one
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. [40] 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 the symbolic analysis encounters a conditional instruction, it may decide to explore both the true and false execution paths. In such a case, the partially constructed DFG is copied 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 assembly 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 [40]. 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.
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 [40] 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).

A broker request for \( \text{op} \) preserves the following properties:
(i) For all \( v \in V \), \( v = h(v) \), i.e. all nodes in \( G \) are normalized.
(ii) For all \( v_1, v_2 \in V \), \( h(v_1) = h(v_2) \iff v_1 = v_2 \), 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 \( q = h(\text{op}(v_1, v_2)) \) 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 \( h(q) = q \), 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 \( h(p) = h(q) \). By (i), we get \( h(p) = p \), and hence \( p = h(q) \). By definition, \( q = h(\text{op}(v_1, v_2)) \) and hence \( p = h(\text{op}(v_1, v_2)) \). By (1), we get \( p = h(\text{op}(v_1, v_2)) \) and thus \( p = q \). This contradicts \( q \notin V \), and hence no \( p \in V \) exists such that \( h(p) = h(q) \). Therefore, (ii) holds for \( V' \).

Since (i) and (ii) trivially hold for the base case, i.e., an empty graph \( G \), where \( V = \emptyset \), and the above shows preservation during the step case, the properties hold for any \( G, E \) is also subject to modifications, but they are not relevant for the proof.

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.

\[
\begin{array}{c}
4 \\
+ \\
12 \\
\rightarrow \\
16
\end{array}
\]

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’s result 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, 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 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, 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 $\phi$ 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 $P$ restricts the domain of node $v_1 \in V$. Then, a condition depending on $v_2 \in V$, can be evaluated immediately 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 $S = (G, P, B)$, where $G = (V, E)$ is the partially constructed DFG. $P$ is a predicate consisting of conditions that apply. These conditions restrict unknown variables to a certain domain. We define predicate $P$ so that satisfaction of $P$ guarantees 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 $i^{th}$ occurrence of the underdetermined conditional instruction located at execution address $e$.

The graph construction begins by initializing $S = (G, P, B)$ to the empty state, i.e, $G$ is an empty graph, $P = \text{true}$, 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, e.g., a branch instruction, in which case, we continue the construction 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 callee 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. \( c \) may be a determined condition. If either \( v_1 \) or \( v_2 \) is non-constant, \( c \) need not be underdetermined, since predicate \( P \) may sufficiently restrict \( v_0 \) or \( v_1 \) so that \( c \) is determined. Conversely, in case \( c \) is underdetermined, we are forced to choose which execution path to follow. Alternatively, we may follow both paths, by creating a copy of state \( S \), 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 \( S \). 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 \( P \land c = true \) then
    Evaluate instruction at \( e \)
else if \( P \land c = false \) then
    Skip over instruction at \( e \)
else
    \( d \leftarrow po.query(e, B) \)
    if \( d = \text{TAKE_TRUE} \) then
        \( P \leftarrow P \land c \)
        \( B \leftarrow B \cup (e, true) \) \hspace{1cm} \triangleright \text{expand } P \text{ with } c \)
    else if \( d = \text{TAKE_FALSE} \) then
        \( P \leftarrow P \land \neg c \)
        \( B \leftarrow B \cup (e, false) \) \hspace{1cm} \triangleright \text{append decision to backlog} \)
    else if \( d = \text{TAKE_BOTH} \) then
        \( S' \leftarrow S.fork() \) \hspace{1cm} \triangleright \text{fork } S \text{, skipped for } S' \)
```

For every decision made by the path oracle, \( P \) and \( B \) in \( S \) 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
Through experimental evaluation, we settled on the following policy for the path oracle: the goal of the policy is, for some number \( n \), to obtain a DFG representing exactly \( n \) iterations of a primitive with variable input length. The primitive can subsequently be identified by searching for a pattern consisting of exactly \( n \) iterations in the resulting DFG.

We define \( d_{i,j} \in \{ \text{TAKE_TRUE, TAKE_FALSE, TAKE_BOTH} \} \) as the path oracle’s decision for the \( i^{th} \) query for the conditional instruction found at execution address \( e \). The policy for the path oracle is defined as follows:

\[
\begin{align*}
    d_{e,0} & := \text{TAKE_BOTH} \\
    d_{e,1} & := \begin{cases} 
        \text{TAKE_TRUE} & \text{iff } B_{e,0} = \text{true}, \\
        \text{TAKE_FALSE} & \text{iff } B_{e,0} = \text{false} 
    \end{cases} \quad \forall i \in [1, n - 1] \\
    d_{e,j} & := \begin{cases} 
        \text{TAKE_FALSE} & \text{iff } B_{e,0} = \text{true}, \\
        \text{TAKE_TRUE} & \text{iff } B_{e,0} = \text{false} 
    \end{cases} \quad \forall i \in [n, \infty]
\end{align*}
\]

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. Furthermore, since each conditional instruction may introduce at most one additional DFG, the total number of DFGs generated from \( f \) is finite.

7 Purging process

Once the construction is complete, graph \( G \) represents the input/output relation of function \( f \), given predicate \( P \). However, it contains other information as well, such as nodes created from temporary STORE operations to the stack, and expressions rewritten by the broker, leaving the source nodes unused.
For example, suppose that \( v \) represents \( \text{ADD}(x, y) \). Then, a request to the broker for \( \text{ADD}(v, z) \) yields node \( w \), representing \( \text{ADD}(x, y, z) \). \( w \) 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 \text{STORE} operation, and the target address is not relative to the \text{SP} register. Thus, information is stored outside of the stack, or

(iii) a \text{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 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 diagram of the \text{expression} data type is given in Figure 5. It is recursively defined, and hence allows for nested subexpressions. The + keyword denotes the addition of two or more subexpressions. The \textlangle\rangle\textrangle keywords denote a left and right shift, respectively. The \text{label} data type is a reference to a previously defined graph node. The \text{literal} data type denotes a constant value. The \text{STORE}, \text{LOAD}, \text{XOR}, \text{OR}, \text{AND}, \text{MULT} and \text{ROTATE} keywords followed by subexpressions surrounded by parentheses provoke creation of a new graph node. The following subexpressions serve as input nodes. Finally, the \text{OPAQUE} keyword signifies a special wildcard node. A comparison against 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 \text{clamp-label} data type allows one to assign a name to the node type. Consequently, a comparison against a node of any other type yields true, with the restriction that all opaque nodes carrying the same type label must map to nodes of the same type. We refer to this practice as \text{type clamping}.

Figure 4: High-level overview of state machine for the signature DSL

Figure 5: State machine for ‘expression’ type
Within the realm of identifying unknown primitives, defining a special wildcard applicable to a group of nodes would make sense. However, to the best of our knowledge, the nature of subgraph-isomorphism algorithms 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 opaque 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. A particular syntax could trigger the translation automatically. However, the number of variants grows exponentially in the number of times this hypothetical syntax is used, dramatically affecting the running time of the analysis. Therefore, we omit the syntax altogether and enforce the 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 [60] is a recursion backtracking algorithm with pruning. Our framework implements this algorithm, with support for type clamping (see Section 8). For brevity, we avoid further details about Ullmann’s algorithm and our optimizations to it here, and refer the reader to the documentation included with our framework’s source code instead.

Support for type clamping is implemented in the prune routine of the algorithm. Each time a candidate node \( c \) in the code graph is assigned to another node \( e \) in the signature graph, a check is introduced whether \( e \) carries the opaque type, and a clamp label. If so, the clamp label \( l \) is checked for previous assignment to a particular node type. If not, then \( l \) is assigned to \( \text{type}(c) \), the type of the code node. At that point, all candidates \( c' \) for opaque signatures nodes carrying clamp label \( l \), for which \( \text{type}(c) \neq \text{type}(c') \), are invalidated.

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.

10.1 AES, MD5, SHA1

Despite this paper’s strong focus on unknown primitives, there are several arguments to be made for the inclusion of AES, MD5 and SHA1. First, they allow for corroboration of, and straight comparison with the result by [40]. Second, and more importantly, they demonstrate that our proposed symbolic execution 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. Hence, a generic classifier capable of identifying them poses an interesting use case. In a Feistel cipher, a plaintext block \( P \) is split in two pieces \( L_0 \) and \( R_0 \). Then, for each round \( i \in [0, 1, \ldots, n] \), compute

\[
L_{i+1} = R_i
\]

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

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, one yields the graph provided in Figure 6.

![Figure 6: DFG of a Feistel structure](image)

The next step is to construct a signature that represents the DFG from Figure 6. However, \( F \) is an algorithm-specific set of operations, of which thus no properties are known. The OPAQUE operator (see Section 8), is applicable to a single node only, 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 several variants of the signature. In the first, we substitute \( F \) with OPAQUE\((R_i)\), in the second with OPAQUE(OPAQUE\((R_i)\)), etc., until we reach 8 levels of 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 designed for use in embedded devices. Both hardware and software implementations of (N)LFSRs are common.
An (N)LFSR $R$ is initialized with a seed value. Each round, a new bit is generated using the feedback function $L$ from a subset of the bits in the current state of $R$. $R$ is an LFSR, if $L$ is linear, e.g., an exclusive-or over the input bits. 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. Subsequently, an output bit is generated by feeding $R$ to some function $F$. Hence, we have $R_0 = seed$. Then, for each round $i \in [0,1,\ldots,n]$, $R_{i+1} = (R_i \ll 1) \mid L(R_i)$, where $\ll x$ denotes a left shift by $x$ bits and $\mid$ denotes bitwise OR.

Figure 7 depicts a translation of the definition given 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).

### 10.4 Sequential Block Permutation

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

Let $H_i$ be the $i$th output block of a sequential block permutation function, $B_i$ be the $i$th 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

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

A DFG representation of this definition is given in Figure 8. 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 $B_0, B_1, \ldots, B_n$.

In order to express this in a signature definition, we may opt to use an approach similar to how the Feistel cipher signature definition is constructed. However, Figure 8 does not define 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). Hence, we must also take property (ii) into account. Let $c_i$ be the $i$th instance of $c$. The number of arithmetic/logical operations in $c_i$ transforming the output of $c_{i-1}$ need not be related to that of input block $B_i$. Therefore, in order to translate $c$ into several variants of the signature, we have to perform a translation for both the path between $c_i$ and $c_{i-1}$, and that between $c_1$ and $B_0$. The number of variants grows exponentially in the number of such translations. On top of that, the compression function $c$ can be vastly more complex than a round function in a Feistel cipher. For example, 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.

In all, we conclude that the number of signature variants, and therewith the running time complexity of the analysis, becomes prohibitively large.

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 $B_0, B_1, \ldots, B_n$ 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. From these observations, we devise 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 8).

(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 tolerate constant inputs and inputs of type LOAD in this step.

The definition given in this section resembles exactly that of the Merkle-Damgård construction, with the omission of the padding function applied to the last block. We studied the possibility of creating a Merkle-Damgård classifier built on the sequential block permutation classifier. This appeared to be infeasible due to a combination of factors. First, no concrete data dependencies are defined for the padding function, but only a set of properties (see [31] for so-called MD-compliant padding). If we assume that the padding function encodes the input length, then the final instance of compression function $c$ should take a node representing it as an input. Therefore, we have to reliably determine the input length, which need not coincide with $n$ (see Section 2), identify the final instance of $c$, which depends on whether the encoded input length fits within the padding space, and make additional assumptions about the length encoding. As such, we conclude the distinction of the Merkle-Damgård construction within the broader class of sequential block permutation functions relies on too many assumptions, and leave it as a manual effort for the human analyst.

11 Experimental evaluation

We evaluate our solution’s performance with respect to accuracy and running time on the following three test sets: (a) a collection of shared libraries and executables part of the OpenWRT network equipment firmware, (b) a collection of proprietary cipher implementations built from public sources, and (c) a collection of representative real-world embedded firmwares from relevant verticals (PLCs, ECUs).

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 of our work 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.

11.1 Performance on OpenWRT binaries

The empirical evaluation is conducted on an AMD Ryzen 3600 machine with 16 GB of RAM, which is considered to be mid-range hardware nowadays. The version of OpenWRT used is 19.07.2, which is the latest version at time of writing. We begin by investigating the impact of the parameters defined throughout this paper. We do this by running the analysis on libcrypto.so.1.1, taking on different parameter values, and measuring performance in terms of running time and accuracy. We then choose sensible parameter values based on a trade-off between the two, and use them for the remainder of the section, unless specified otherwise.

Section 6.1.1 defines $n$, the target number of iterations. The (N)LFSR and sequential block permutation classifiers are affected by this value. The others target a single instance only. The latter identifies two successive instances of some compression function $c$. The IV and length padding may provoke differences in the DFG in the first and the last instance of $c$, respectively. Taking $n = 4$ circumvents this. A pattern of 4 successive (N)LFSR rounds is also sufficiently distinctive. Therefore, for the remainder of this section, we take $n = 4$.

Formally, the construction of a DFG from binary code, as described in 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 $t_{\text{timeout}}$. Figure 9a 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 $t_{\text{timeout}} = 10s$.

Figure 9b depicts the time taken to complete the entire analysis pipeline over every function in libcrypto.so.1.1, under the influence of varying inline depth $d$. Figure 9c contains accuracy measurements for each signature. True negatives are omitted since they cover an overwhelming majority of results, and thus impact readability. Before discussing the results, we will first provide guidelines on how the chart should be interpreted. 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 9c are aggregated on a per-function level.

Let $f$ be any function in the binary under analysis, and let signature $s_\alpha$ denote a signature whose goal is to identify prim-
Not implement primitive $\alpha$, yet its signature is found in at least one graph in $F$. A result is marked as a true negative if $\neg\text{imp}(f, \alpha) \land \exists G, G \in F \land \text{match}(s_\alpha, G)$. A result is a false negative if $\text{imp}(f, \alpha) \land \neg\exists G, G \in F \land \text{match}(s_\alpha, G)$.

The results in Figure 9c 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 take $d = 2$ for the remainder of this section, unless specified otherwise.

At this point, sensible values for $n$, $d$ and $t_{\text{timeout}}$ have been selected. We continue the evaluation by feeding various binaries within OpenWRT containing cryptographic primitives to our analysis framework. The results are listed in Table 1. Each cell in the table depicts the symbol name in the corresponding column.

### Table 1: Analysis result for various binaries in OpenWRT

| Algorithm | signature | description | libcrypto.so.1 | libcrypto.so.2.16 | libcrypto.so.2.16.3 | libnettle.so.7.0 |
|-----------|-----------|-------------|----------------|-------------------|-------------------|------------------|
| AES       | ✓         | ✓           | ✓              | ✓                 | ✓                 | ✓                |
| sha1      | ✓         | ✓           | ✓              | ✓                 | ✓                 | ✓                |
| bl.perm.  |           |             |                |                   |                   |                  |
| sha256    | ✓         | ✓           | ✓              | ✓                 | ✓                 | ✓                |
| AES       | ✓         | ✓           | ✓              | ✓                 | ✓                 | ✓                |
| bl.perm.  |           |             |                |                   |                   |                  |

Figure 9: Effect of inline depth $d$ and $t_{\text{timeout}}$ for libcrypto.so.1.1

In this section, we turn our attention to proprietary algorithms. Most algorithms included in this section 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.

### 11.2 Performance on proprietary algorithms

The above results indicate our solution is capable of successfully identifying the vast majority of cryptographic primitives present in various binaries in timely manner. Should accuracy take precedence over performance, it is possible to tune the discussed parameters to improve detection.

#### 11.2.1 Cryptographic algorithms

Table 2 depicts the analysis results for various proprietary algorithms. A description, the analysis result, and other relevant information is condensed into a single table due to space restrictions. All algorithms are successfully identified, with the exception of Red Pike. We found that the cipher’s structure only loosely resembles a Feistel network; an addition is used instead of the exclusive-or operation (see Section 10.2).

Hence, it is arguably not a Feistel cipher. The possibility that
The test set of representative real-world firmwares consists of images are 'flat' binaries and hence symbol names are absent. The best of our knowledge, the table covers all cryptographic algorithms present in the sample set of firmware images. The size, nature and complexity of these images ensure test-set realism. PLC and Volkswagen IPC. The size, nature and complexity of these images ensure test-set realism.

11.2.2 Firmware Images

The test set of representative real-world firmwares consists of images for the Emerson ControlWave Micro RTU, Emerson ControlWave XPC flow computer, Schneider Electric M340 PLC and Volkswagen IPC. The size, nature and complexity of these images ensure test-set realism.

Table 3 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. [63] 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, even when aided by heuristics.

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. 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.
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] L 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. https://www.bbc.com/news/technology-23487928, 2013.

[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] 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.

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

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

[12] 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.

[13] M Briceno, I Goldberg, and D Wagner. An implementation of comp128, 1998. http://www.iol.ie/kooltek/a3a8.txt.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.
[25] 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.

[26] 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.

[27] EN ETSI. 300 392-7 v3.1 (2012-07) european standard (telecommunication series) terrestrial trunked radio (tetra); voice plus data (v+d); part 7: Security. European Telecommunications Standards Institute (ETSI) Site. URL: http://webapp.etsi.org/workprogram/Report_WorkItem.asp.

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

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

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

[31] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography. *Summer course “Cryptography and computer security” at MIT*, 1999:1999, 1996.

[32] 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.

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

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

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

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

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

[38] 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.

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

[40] 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.

[41] IO Levin and F Yarochkin. Draft crypto analyzer (draca), 2013.

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

[43] 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.

[44] 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.

[45] 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.

[46] Carlo Meijer and Roel Verduilt. 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.

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

[48] Microchip. Hopping code decoder using a PIC16C56, AN642. https://web.archive.org/web/20080916043223/http://www.keeloqboom.ru/decryption.pdf, 1998.
[49] Mr Paradox, AT4RE. Hash & crypto detector (hcd), 2009. https://web.archive.org/web/20091203010936/http://www.at4re.com/download.php?view.8.

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

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

[52] 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.

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

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

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

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

[57] 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.

[58] 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.

[59] J Tamas. Secrets of the sim, 2013. http://www.hackingprojects.net/2013/04/secrets-of-sim.html.

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

[61] Roel Verdult. The (in) security of proprietary cryptography. PhD thesis, [SI: sn], 2015.

[62] 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 ({USENIX} Security 12), pages 237–252, 2012.

[63] 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 (Supplement to {USENIX} Security 15), pages 703–718, 2015.

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

[65] 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.

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

[67] 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.

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

[69] 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.
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, ... R5, [R4, R6] ; store result at R6[R4]
9 ADD R4, R4, #1 ; increment R4
10 B _begin ; continue at beginning
11 _end:
```

Figure 10: Example stream cipher ARM assembly snippet

A practical example Suppose the graph construction is run on the example ARM assembly snippet displayed in Figure 10. We start with \( S = (G, P, B) \), with \( P = \text{true} \). Line 4 contains conditional instruction Branch Greater/Equal (BGE). During the first visit of this instruction, we have \( i = 0 \), \( P = \text{true} \), and \( c = (R8 \leq 0) \). Since \( R8 \) is unknown, \( c \) is underdetermined. The path oracle policy prescribes \( \text{TAKE\_BOTH} \). Thus, we get \( P = (R8 \leq 0) \), \( B_{4,0} = \text{true} \), and \( S' = (G', P', B') \), with \( P' = (R8 > 0) \) and \( B_{4,0}' = \text{false} \). For state \( S \), the instruction is evaluated, and thus the construction continues on line 11, and hence terminates. For \( S' \), the instruction is skipped, thereby visiting the body of the loop. Eventually, \( S' \) revisits the instruction at line 4. This time we have \( c = (R8 \leq 1) \), \( i = 1 \), \( P' = (R8 > 0) \) and \( B_{4,0}' = \text{false} \). Since \( P' \wedge c \) is underdetermined, we query the path oracle, and obtain \( \text{TAKE\_FALSE} \), causing another visit of the loop’s body. Finally, at \( i = n \), we get \( c = R8 \leq n \) and \( P' = (R8 > n - 1) \). We obtain \( \text{TAKE\_TRUE} \) from the path oracle. Thus, the construction terminates. We obtain two graphs; one corresponding to predicate \( R8 \leq 0 \), and another corresponding to \( R8 = n \). 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 here.

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

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
```

D Implementation

An implementation of the framework described in this paper is available for download\(^4\). It comes in the form of a plug-in for the popular IDA disassembler. At the time of writing, support is implemented for 32 bits ARM binaries. The architecture is modular, and expanding support to other architectures is relatively straightforward. Figure 11 shows a sample analysis report, and a DFG plot generated by our implementation.

\(^{14}\)https://github.com/wheres-crypto/wheres-crypto
Figure 11: An impression of the implementation of our framework