Where’s Crypto?

Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code

Carlo Meijer (RU), Veelasha Moonsamy (RUB), Jos Wetzels (Midnight Blue)
WHERE'S CRYPTO?: AUTOMATED IDENTIFICATION AND CLASSIFICATION OF PROPRIETARY CRYPTOGRAPHIC PRIMITIVES IN BINARY CODE

AGENDA

1. Introduction
2. Prior Work
3. Solution Overview
4. Experimental Evaluation
5. Conclusions
INTRODUCTION
Despite popular consensus, usage of proprietary cryptography persists, especially in embedded systems
  • E.g. Physical Access Control, Telecommunications, Machine-to-Machine Authentication

Presents significant obstacle to security evaluation efforts
  • Certification & Compliance
  • Secure Procurement
  • Time-boxed Penetration Tests

Manual RE effort required for determining presence & nature of proprietary algorithms. Might lead to false conclusions of robustness, NDAs or court injunctions* leave other affected parties to repeat expensive research

There is a concrete industry need for **automated detection of as-of-yet unknown cryptographic primitives in binary code**

1. Identification of as-of-yet unknown cryptographic algorithms falling within relevant taxonomical classes.

2. Efficient support of large, real-world embedded firmware binaries.

3. No reliance on full firmware emulation or dynamic instrumentation due to issues around platform heterogeneity and peripheral emulation in embedded systems.
PRIOR WORK
• **Dedicated Functionality Identification**
  Identification of native cryptographic APIs, libraries or hardware functionality is inherently incapable of detecting unknown algorithms.

• **Data Signatures**
  Identification using constants (IVs, NUMS) & LUTs is unsuitable for unknown algorithms as well as for known algorithms that don’t rely on fixed data or generate LUTs dynamically.

---

```
sub_3a034

PUSH {R5,LR}
MOVS R2, #0 ; c
MOVS R2, #0x5C ; `\' ; n
MOVS R4, R0
BLX memload
LDR R0, =0x67452301
LDR R1, =0xdeadbeef
LDR R2, =0x807a397e
LDR R3, =0x325476e
STH R4, {R0-R3}
MOVS R0, #1
POP {R4,PC}

; CODE XREF: sub_3a4a8+124p
; sub_79688c+44p
; DATA XREF: ...
```

Initial values revealing MD5
PRIOR WORK
LIMITATIONS OF PRIOR WORK

• **Code Heuristics**
  1. Matching mnemonic-constant tuples which suffers from essentially the same drawbacks as data signatures for our purposes.
  2. Matching routines with high ratios of bitwise arithmetic (BAR) instructions. Main drawbacks here are the lack of granular taxonomical identification as well as FP susceptibility, especially on embedded systems where heavy BAR usage is present as part of e.g. peripheral interaction.

• **Deep Learning**
  Usage of Dynamic Convolutional Neural Networks has been proposed but this approach is inherently unable to classify unknown algorithms and relies on dynamic binary instrumentation.
LIMITATIONS OF PRIOR WORK

- **Data Flow Analysis**
  1. Identification based on static relation between functions and their I/Os.
     Taint analysis and entropy change evaluation.
     Comparison of emulated/symbolically executed function I/O to collection of
     reference implementations / test vectors.
  2. Usage of dynamic instrumentation & symbolic execution to translate
     candidate algorithms into Boolean formulas for comparison to reference
     implementations.

Both approaches are unsuitable due to their reliance on emulation/dynamic
instrumentation and/or inherent inability to detect unknown algorithms.
Finally, there is the DFG isomorphism approach as proposed by Lestringant et al.*

- Generate a DFG from assembly instructions.
- Compare it to that of a known algorithms using Ullmann's subgraph isomorphism algorithm.

* P. Lestringant et al. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. 2015.
No systematic way to deal with data-dependent branches. Approach is limited to linear sequences of instructions:

1. No strategy for code fragment selection is proposed. Authors propose a set of heuristics, e.g. analyzing each basic block.

2. Class of cryptographic primitive often only becomes clear once analysis incorporates conditional instructions, consider:

Suppose we have a proprietary stream cipher \( \sigma \), containing a KSG operating in a loop driven by a length parameter. DFGs computed from basic blocks will represent at most a single iteration and hence do not show stream cipher characteristics.

* P. Lestringant et al. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. 2015.
Observation: the vast majority of proprietary cryptography falls within established primitive classes.

We aim to develop structural signatures capturing a taxonomical class while disregarding algorithm’s particulars. We developed an instrumental taxonomy based on prior work* in order to facilitate this.

Our approach leverages this taxonomy to specify structural signatures by building on two fundamentals:

- Data Flow Graph (DFG) isomorphism
- Symbolic Execution

1. Limitations of prior work are overcome by combining subgraph isomorphism with symbolic execution, rendering it suitable for identifying unknown ciphers. To the best of our knowledge there is no prior work in industry or academia that addresses this problem.

2. We propose a new domain-specific language (DSL) for defining structural properties of cryptographic primitives, along with several examples.

3. We provide a FOSS PoC implementation* and the corresponding evaluation in terms of analysis time and accuracy against real-world binaries.

* https://github.com/wheres-crypto/wheres-crypto
SOLUTION OVERVIEW
A schematic overview of the identification/classification pipeline
Build a graph incrementally as we pass over instructions

Node determined by operand type

- **Immediate**: constant value.
- **Register**: create an edge to the node representing the value last written to that register.
- **Memory**: create LOAD/STORE operations.
Semantically equivalent code often yields different DFGs, due to architectural, compiler- and implementation particularities.

- Problematic because we’d like to compare it to a single reference DFG.
- Normalization maps different variants to a single canonical form.

1. Operation simplification
2. Common subexpression elimination
3. Memory access
4. Associativity merging
What path should we follow when we encounter a conditional instruction?

- In some cases, the evaluation outcome is determined by its preceding instructions → *determined* condition.
- In other cases, we don’t know, e.g. when it depends on an unknown variable, e.g. a function parameter → *underdetermined* condition.
- For underdetermined conditions, we have to *choose* the evaluation outcome: true, false, or both (i.e. create two graphs):

```c
int some_function(int a, int b) {
    if (a == 7) {
        return do_a_eq_7(b);
    } else {
        return do_a_neq_7(a + b);
    }
}
```
Always taking both paths maximizes code coverage but is unfeasible.

- We have to come up with a strategy when to do so → *path oracle*.
- Our goal: obtain a DFG representing $n$ iterations of a cryptographic primitive.
- Consider the following toy example:
  - The conditional jump is underdetermined, as it depends on a variable.

```c
int streamcipher(void *ctx, uint8_t *dst,
                 uint8_t *src, uint32_t len) {
    uint32_t i;
    for (i = 0; i < len; i++) {
        dst[i] = src[i] ^
                 keystream_generator(ctx);
    }
}
```
Each encounter with the conditional instruction, we are met with two options:

- $0 < R3$: true → perform another iteration, false → return immediately.
- $1 < R3$: true → perform another iteration, false → return immediately.
- $2 < R3$: true → perform another iteration, false → return immediately.
- ...

- We want a generic approach that gives us $n$ iterations of a primitive, so:

- On the first encounter, we take both execution paths:
  → The false case will immediately return.
  → The true case takes us back to another underdetermined condition at the exact same execution address.

- For the second encounter and beyond, we keep replicating the decision that caused the revisit to occur until the $n$th visit, and then take the opposite path, i.e. return.
Finally, we obtain two DFGs:

- One representing 0 iterations, the other representing $n$.

```c
int streamcipher(void *ctx, uint8_t *dst,
                 uint8_t *src, uint32_t len) {
    uint32_t i;
    for (i = 0; i < len; i++) {
        dst[i] = src[i] ^
                 keystream_generator(ctx);
    }
}
```
Besides the actual semantics, the resulting DFG contains other information:

- Temporary LOADs/STOREs from/to the stack.
- Expressions translated through normalization, leaving their source nodes unused.

We consider a leaf node to be part of semantics if either:

- It is the return value.
- It is a STORE operation to an address not relative to the stack pointer.
- It is a CALL operation.

We continue to remove leaf nodes not part of semantics until we hit a fixed point. Then, all nodes are either leaves part of semantics, or an intermediary result.
Cryptographic primitive signatures must be expressed in some way

- The signature is ultimately nothing more than a DFG of a cryptographic primitive, which is fed to the subgraph isomorphism algorithm.
- In principle, we could simply generate it from assembly instructions as well.
- However, we wouldn’t be able to express wildcards and more. → Domain-specific language (DSL).

The high level state machine

The ‘expression’ type

Example: (N)LFSR
We use Ullmann's subgraph isomorphism* algorithm.

- Known to be NP-complete.
- Yet, performs quite well for our purpose.

An LFSR signature is found to exist as a subgraph, highlighted in red.

To showcase the applicability of our method, we propose several example signatures:

- Algorithm-specific ones:
  - AES, MD5, XTEA and SHA1
- Generic ones:
  - Feistel network
  - (Non-)Linear feedback shift register
  - Sequential block permutation

We use these signatures in order to evaluate performance in terms of accuracy and running time.
EXPERIMENTAL EVALUATION
We evaluate accuracy & running time on following test sets*:

- Sample set used in prior work by Lestringant et al.
  → Evaluate algorithm-specific identification performance without reliance on heuristics.

- OpenWRT firmware shared libraries & executables
  → Evaluate generic signature identification performance on redistributable binaries, easy to reproduce results.

- Public proprietary cipher implementations
  → Evaluate generic signature identification performance on proprietary ciphers publicly available in source form, harder to reproduce.

- Collection of real-world embedded firmwares (PLCs, ECUs)
  → Evaluate generic signature identification performance on real-world embedded firmwares, not reproducible.

Evaluation is conducted on a mid-range AMD Ryzen 3600 machine with 16 GB of RAM.
Recall: $n$ is the target number of algorithm iterations contained within a DFG. Value for $n$ should be low, as it correlates with size of constructed DFGs, but high enough to accommodate all signatures.

(N)LFSR & sequential block permutation classifiers are affected by this.

- Latter case works by identifying two successive instances of compression function $c$. Since normalization promotes numeric simplification, initialization & finalization steps may get merged with first & last instance of $c$, respectively.
- Thus, $n = 4$ allows for at least two successive instances of $c$ in the DFG while choosing $n > 4$ does not offer advantages in this regard.
- Wrt (N)LFSRs, falsely identifying 4 successive rounds is highly unlikely.

- Hence, we pick $n = 4$ for our evaluation.
EXPERIMENTAL EVALUATION
COMPARISON WITH LESTRINGANT ET AL.

- We use algorithm-specific signatures in order to warrant a fair comparison.
- All primitives are correctly identified.
- No heuristics for code fragment selection required, where Lestringant et al. does.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Compiler</th>
<th>-O0 / Debug</th>
<th>-O1</th>
<th>-O2 / Release</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTEA</td>
<td>GCC</td>
<td>ok (1ms)</td>
<td>ok (2ms)</td>
<td>ok (2ms)</td>
<td>ok (2ms)</td>
</tr>
<tr>
<td></td>
<td>Clang</td>
<td>ok (1ms)</td>
<td>ok (2ms)</td>
<td>ok (2ms)</td>
<td>ok (2ms)</td>
</tr>
<tr>
<td></td>
<td>MSVC</td>
<td>ok (1ms)</td>
<td>-</td>
<td>ok (2ms)</td>
<td>-</td>
</tr>
<tr>
<td>MD5</td>
<td>GCC</td>
<td>ok (267ms)</td>
<td>ok (335ms)</td>
<td>ok (345ms)</td>
<td>ok (348ms)</td>
</tr>
<tr>
<td></td>
<td>Clang</td>
<td>ok (286ms)</td>
<td>ok (241ms)</td>
<td>ok (272ms)</td>
<td>ok (265ms)</td>
</tr>
<tr>
<td></td>
<td>MSVC</td>
<td>ok (269ms)</td>
<td>-</td>
<td>ok (322ms)</td>
<td>-</td>
</tr>
<tr>
<td>AES</td>
<td>GCC</td>
<td>ok (64ms)</td>
<td>ok (61ms)</td>
<td>ok (53ms)</td>
<td>ok (56ms)</td>
</tr>
<tr>
<td></td>
<td>Clang</td>
<td>ok (37ms)</td>
<td>ok (32ms)</td>
<td>ok (32ms)</td>
<td>ok (27ms)</td>
</tr>
<tr>
<td></td>
<td>MSVC</td>
<td>ok (30ms)</td>
<td>-</td>
<td>ok (42ms)</td>
<td>-</td>
</tr>
</tbody>
</table>
Nearly all primitives identified using generic signatures.
Publicly available proprietary algorithms:
- Again, nearly all primitives identified using generic signatures.

Analysis of firmware images of embedded devices:
- All primitives correctly identified, except Megamos.
- Analysis reveals that reliance on implicit flows causes the identification to fail.
Despite solid public alternatives, proprietary crypto has persisted (especially in embedded systems), posing a time-consuming, labor-intensive obstacle to security analysis efforts.

**Solution Criteria**

- Should be capable of automatically & efficiently identifying unknown cryptographic algorithms in large, real-world embedded firmwares.
- Should not rely on emulation or binary instrumentation.

No prior work exists that satisfies these criteria.
Our novel approach

- Combines DFG isomorphism with symbolic execution
- Introduces specialized DSL to enable identification of unknown cryptographic algorithms
- Is architecture- and platform-agnostic
- Performs well in terms of accuracy & running time on real-world firmware images
Thank you

- See the paper `Where’s Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code`
- FOSS reference code
  https://github.com/wheres-crypto/wheres-crypto