Where's Crypto?

Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code

Carlo Meijer (RU), Veelasha Moonsamy (RUB), Jos Wetzels (Midnight Blue)

AGENDA

1. Introduction
2. Prior Work
3. Solution Overview
4. Experimental Evaluation
5. Conclusions

INTRODUCTION

BACKGROUND

- 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**

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

CRITERIA

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

LIMITATIONS OF 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                                    ; CODE XREF: sub_3AAA8+12↓p
                                             ; sub_7608C+4↓p
                                             ; DATA XREF: ...
                PUSH    {R4,LR}
                MOVS    R1, #0               ; c
                MOVS    R2, #0x5C ; '\'      ; n
                MOVS    R4, R0
                BLX     memset
                LDR     R0, =0x67452301
                LDR     R1, =0xEFCDAB89
                LDR     R2, =0x98BADCFE
                LDR     R3, =0x10325476
                STM     R4!, {R0-R3}
                MOVS    R0, #1
                POP     {R4,PC}
; End of function sub_3A034
```

Initial values revealing MD5

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.

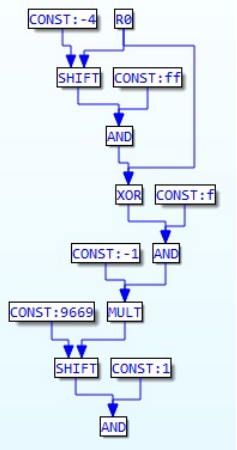
LIMITATIONS OF PRIOR WORK – DFG ISOMORPHISM

- **Data Flow Analysis – DFG Isomorphism**

  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.

```
PUSH    {R11}
ADD     R11, SP, #0
SUB     SP, SP, #0xC
MOV     R3, R0
STRB    R3, [R11,#var_5]
LDRB    R3, [R11,#var_5]
MOV     R3, R3,LSR#4
UXTB    R2, R3
LDRB    R3, [R11,#var_5]
EOR     R3, R3, R2
UXTB    R3, R3
AND     R3, R3, #0xF
LDR     R2, =0x9669
MOV     R3, R2,ASR R3
UXTB    R3, R3
AND     R3, R3, #1
UXTB    R3, R3
MOV     R0, R3
SUB     SP, R11, #0
POP     {R11}
BX      LR
```



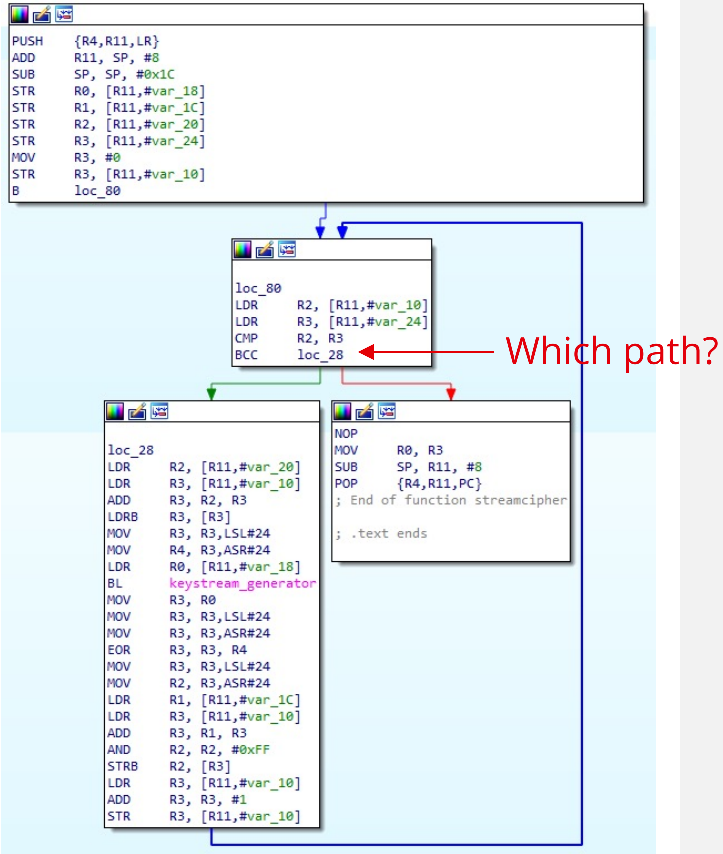
* *P. Lestringant et al. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. 2015.*

LIMITATIONS OF PRIOR WORK – DFG ISOMORPHISM

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:

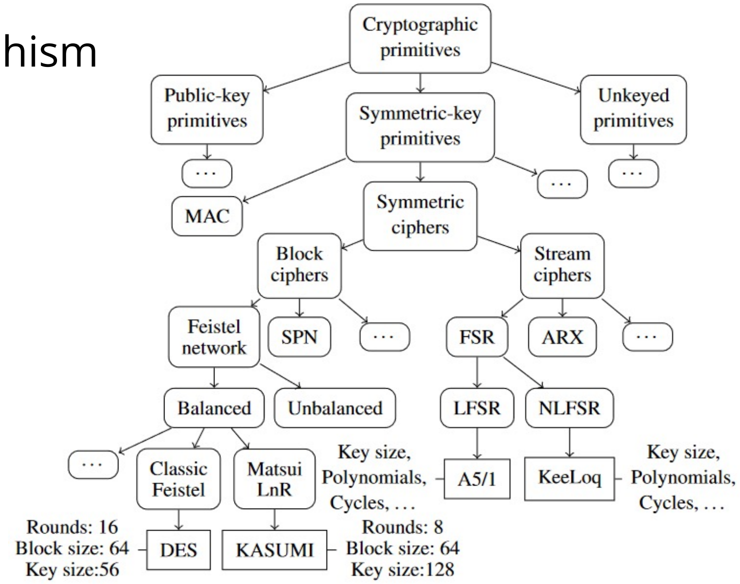
Suppose we have a proprietary stream cipher σ, containing a KSG operating in a loop driven by a length parameter. DFGs computed from basic blocks will represent at most a single iteration and hence do not show stream cipher characteristics.



* *P. Lestringant et al. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. 2015.*

APPROACH

- 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



* R. Avanzi. A salad of block ciphers. 2016.
L. Keliher. Linear cryptanalysis of substitution-permutation networks. 2003.
C. Manifavas et al. A survey of lightweight stream ciphers for embedded systems. 2016.
A. Menezes et al. Handbook of applied cryptography. 1996.

CONTRIBUTION

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.

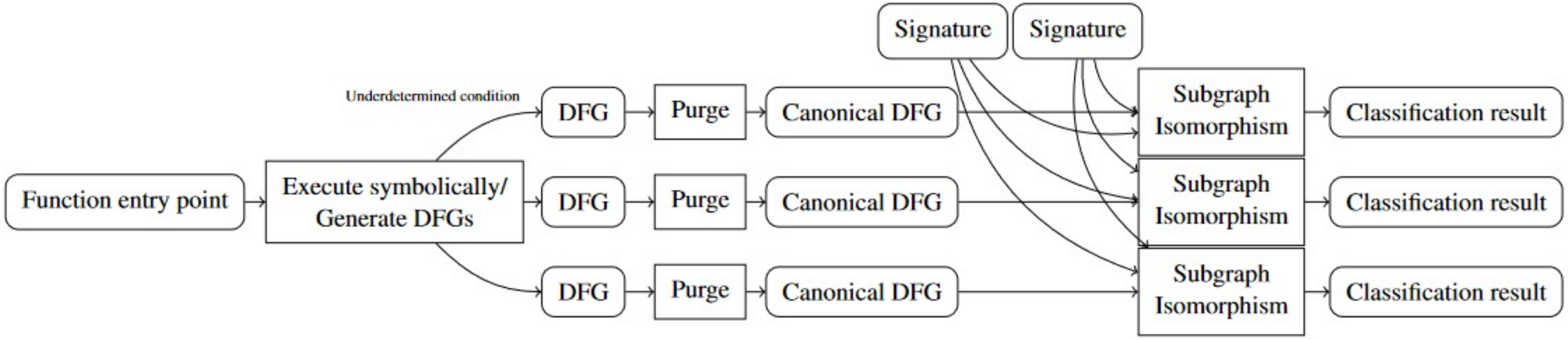
3. We provide a FOSS PoC implementation* and the corresponding evaluation in terms of analysis time and accuracy against real-world binaries.

* https://github.com/wheres-crypto/wheres-crypto

SOLUTION OVERVIEW

OVERVIEW

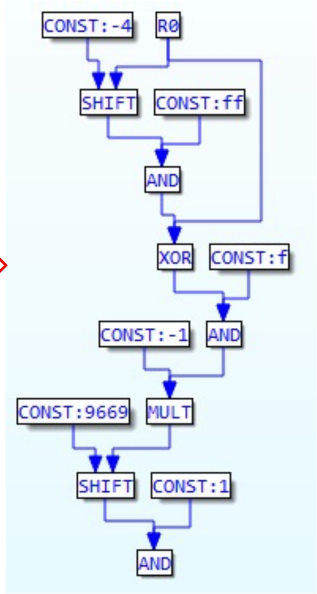
A schematic overview of the identification/classification pipeline



DATA FLOW GRAPH CONSTRUCTION

Build a graph incrementally as we pass over instructions

```
PUSH    {R11}
ADD     R11, SP, #0
SUB     SP, SP, #0xC
MOV     R3, R0
STRB    R3, [R11,#var_5]
LDRB    R3, [R11,#var_5]
MOV     R3, R3,LSR#4
UXTB    R2, R3
LDRB    R3, [R11,#var_5]
EOR     R3, R3, R2
UXTB    R3, R3
AND     R3, R3, #0xF
LDR     R2, =0x9669
MOV     R3, R2,ASR R3
UXTB    R3, R3
AND     R3, R3, #1
UXTB    R3, R3
MOV     R0, R3
SUB     SP, R11, #0
POP     {R11}
BX      LR
```



Node determined by operand type

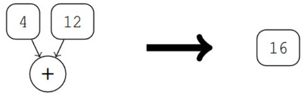
- *Immediate*: constant value.
- *Register*: create an edge to the node representing the value last written to that register.
- *Memory*: create LOAD/STORE operations.

DATA FLOW GRAPH CONSTRUCTION

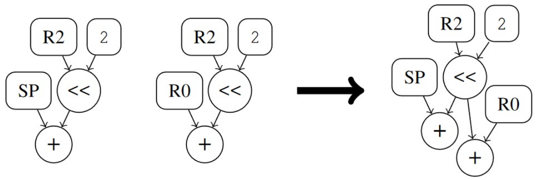
Semantically equivalent code often yields different DFGs, due to architectural, compiler- and implementation particularities.

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

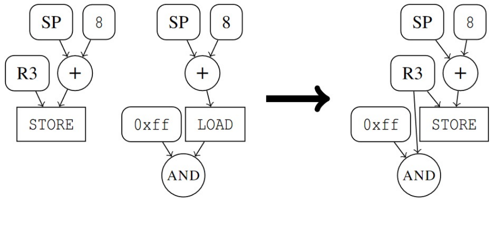
1. Operation simplification



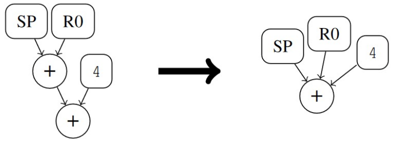
2. Common subexpression elimination



3. Memory access



4. Associativity merging



SYMBOLIC EXECUTION

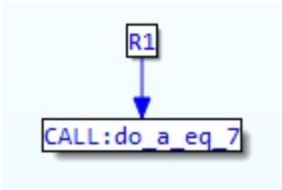
What path should we follow when we encounter a conditional instruction?

```
CMP     R3, #0
BEQ     loc_10748
```

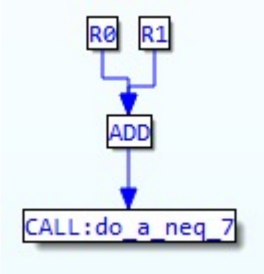
- In some cases, the evaluation outcome is determined by its preceding instructions
  → *determined* condition.
- In other cases, we don't know, e.g. when it depends on an unknown variable, e.g. a function parameter
  → *underdetermined* condition.
- For underdetermined conditions, we have to *choose*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

```
int some_function(int a, int b) {
    if(a == 7) {
        return do_a_eq_7(b);
    } else {
        return do_a_neq_7(a + b);
    }
}
```

a = 7



a ≠ 7

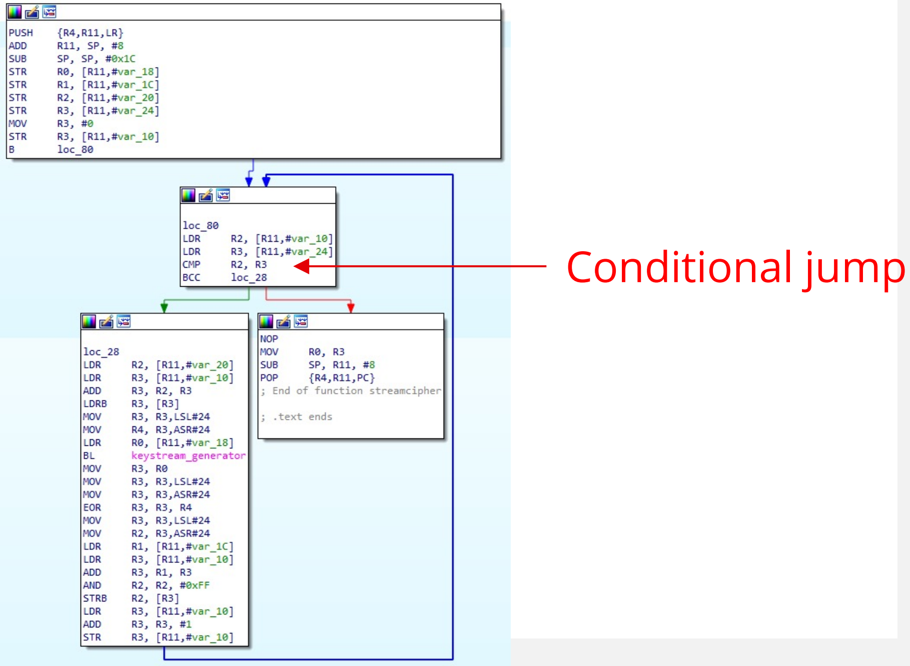


SYMBOLIC EXECUTION

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.

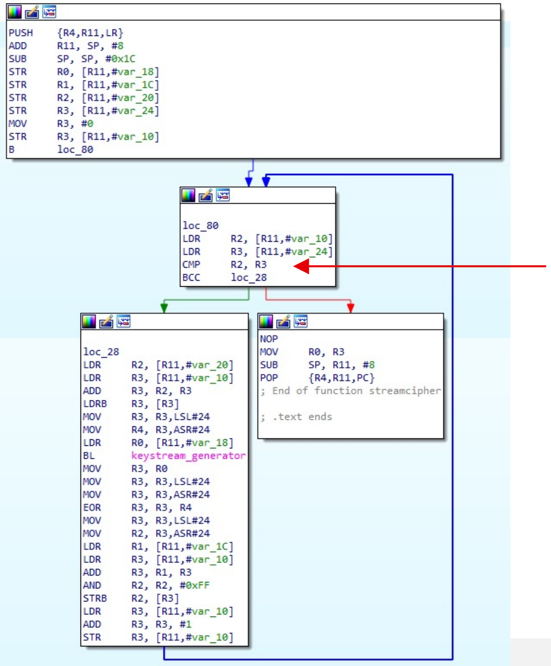
```
int streamcipher(void *ctx, uint8_t *dst,
 uint8_t *src, uint32_t len) {
    uint32_t i;
    for (i = 0; i < len; i++) {
        dst[i] = src[i] ^
        keystream_generator(ctx);
    }
}
```



SYMBOLIC EXECUTION

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.



Conditional jump

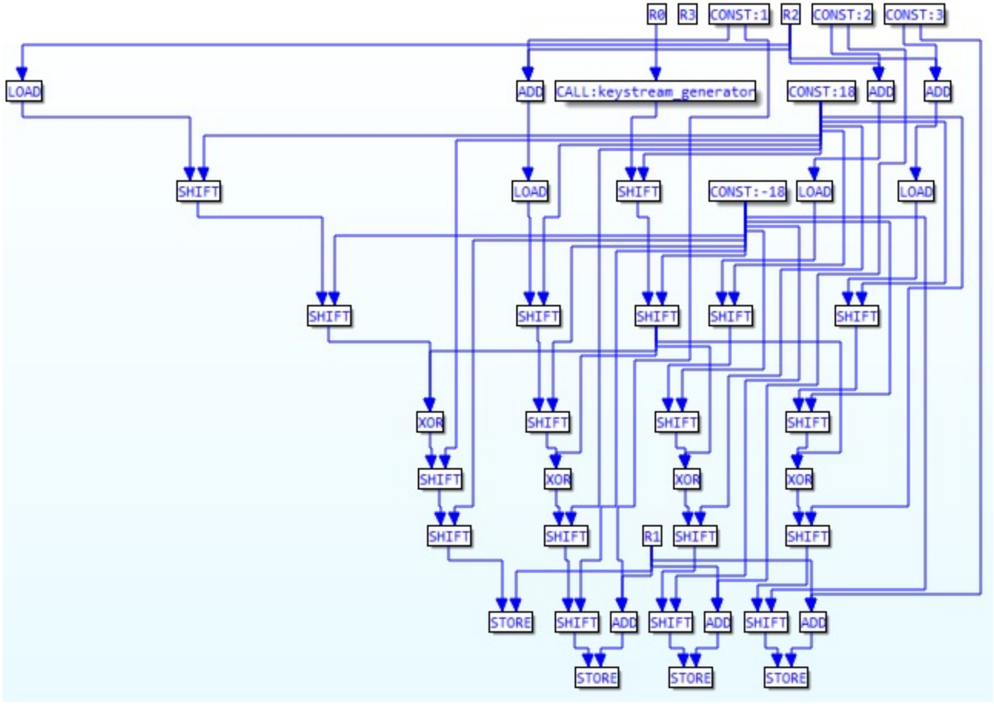
SYMBOLIC EXECUTION

Finally, we obtain two DFGs:

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

```
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);
    }
}
```

0 iterations



n iterations (n=4)

PURGING PROCESS

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:

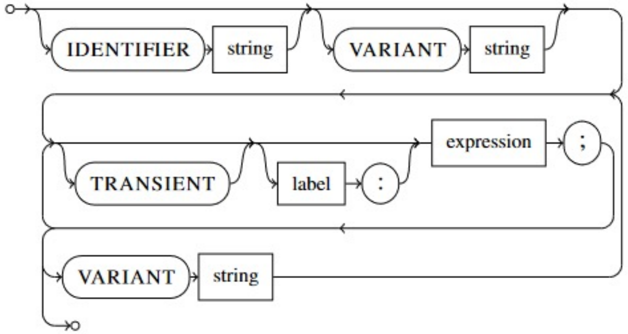
- It is the return value.
- It is a STORE operation to an address not relative to the stack pointer.
- It is a CALL operation.

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.

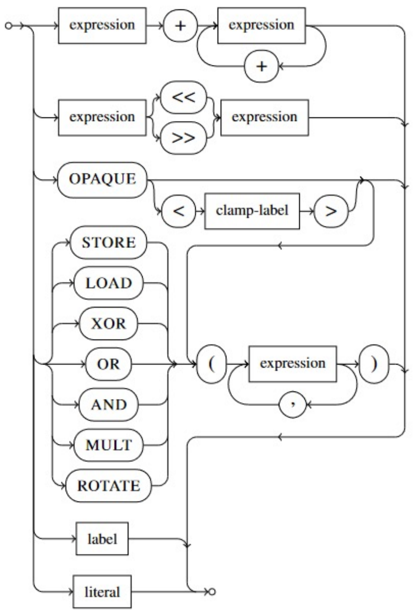
SIGNATURE EXPRESSION

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

```
IDENTIFIER Linear feedback shift register

VARIANT A
...

VARIANT B
...

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);

VARIANT D
...
```

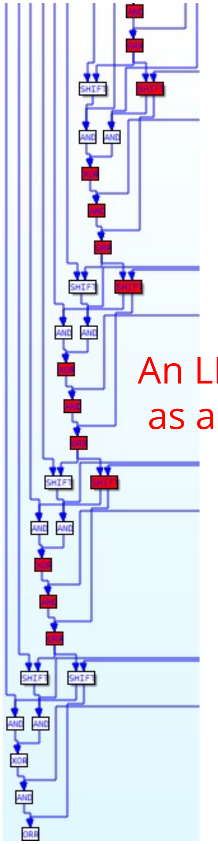
Example: (N)LFSR

SOLUTION OVERVIEW

SUBGRAPH ISOMORPHISM

We use Ullmann’s subgraph isomorphism* algorithm.

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



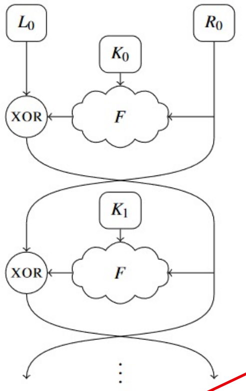
An LFSR signature is found to exist as a subgraph, highlighted in red.

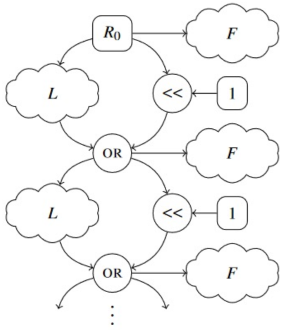
** Julian R Ullmann. An algorithm for subgraph isomorphism. 1976.*

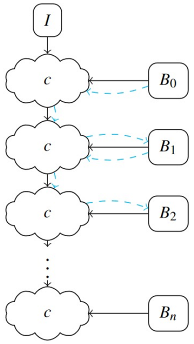
SIGNATURES

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

SETUP

- 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.

SETUP

- 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.

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.

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)	-

PERFORMANCE ON OPENWRT BINARIES

- Nearly all primitives identified using generic signatures.

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 $t_{\text{timeout}} \geq 30s$.

PERFORMANCE ON PROPRIETARY ALGORITHMS

- 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.

Algorithm	Type	Description	Target signature
CRYPTO1	Stream	Cipher used in the Mifare Classic family of RFID tags.	✓ (N)LFSR[1]
HITAG2	Stream	Cipher used in vehicle immobilizers.	✓ (N)LFSR[1]
A5-1	Stream	Provides over-the-air privacy for communication in GSM.	✓ (N)LFSR[1]
A5-2	Stream	GSM export cipher.	✓ (N)LFSR[1]
A5-GMR	Stream	Cipher used in GMR, a standard for satellite phones. Heavily inspired by A5/2.	✓ (N)LFSR[1]
RED PIKE	Block	Classified UK government encryption algorithm.	✗ Feistel cipher
COMP128	Hash	Family of algorithms used for session key and MAC generation in GSM.	✓ Block permutation
KASUMI	Block	Feistel cipher used for the confidentiality and integrity of 3G.	✓ Feistel cipher
MULTI2	Block	A block cipher used for broadcast scrambling in Japan.	✓ Feistel cipher
DST40	Block	Digital Signature Transponder cipher, often found in vehicle immobilizers.	✓ (N)LFSR
KEELOQ	Block	Block cipher used in remote keyless entry systems and home automation.	✓ (N)LFSR

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

Publicly available proprietary algorithms

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

Firmware images

CONCLUSIONS

CONCLUSIONS & FUTURE WORK

- 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.

CONCLUSIONS & FUTURE WORK

- **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

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

Carlo Meijer
Radboud University Nijmegen
c.meijer@midnightblue.nl
https://cs.ru.nl/~cmeijer/

Veelasha Moonsamy
Ruhr University Bochum
email@veelasha.org
https://veelasha.org

Jos Wetzels
Midnight Blue Labs
j.wetzels@midnightblue.nl
https://midnightblue.nl