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

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

AMOSSYS, R&D Security Lab, Rennes, France

University of Rennes, France

Institut Universitaire de France, Paris, France

Introduction

Introduction: Motivations

Why cryptographic implementations need to be reverse-engineered?

- cryptographic algorithms and their implementation are highly sensitive from a security perspective ;
- **source code** and even **specifications** are not always available or trustworthy.

Example: black box security audits, ransomware analysis.

Introduction: Idea

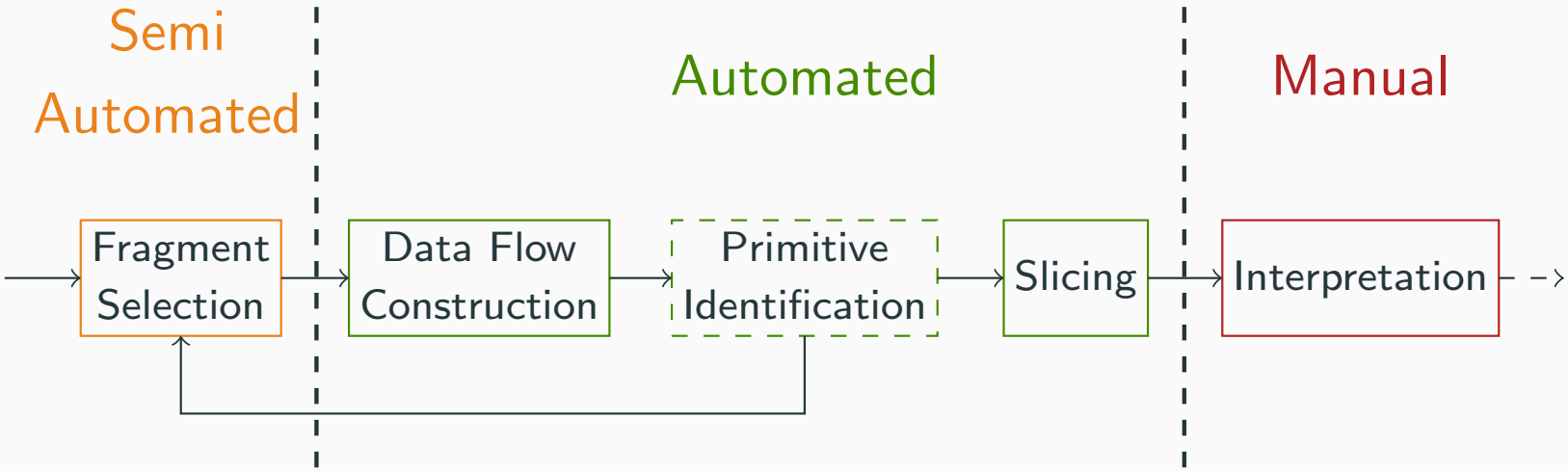
Techniques already exist to automatically identify primitives.
What can be done for modes of operation?

Solution: produce a synthetic representation of the data transfers between the primitives.

- works with any mode of operation ; [1]
- bridge the gap between automated and manual analysis. [1]

[1] probably not the case with automated pattern matching techniques.

Introduction: Solution Overview

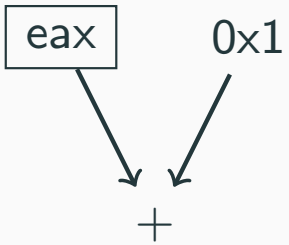


Data Flow

Data Flow: Model

The Data Flow is represented by a directed Graph (*abbr* DFG). A vertex corresponds to an operation. There is an edge to from u to v, if u is an operand of operation v.

```
1   add eax, 0x1
```

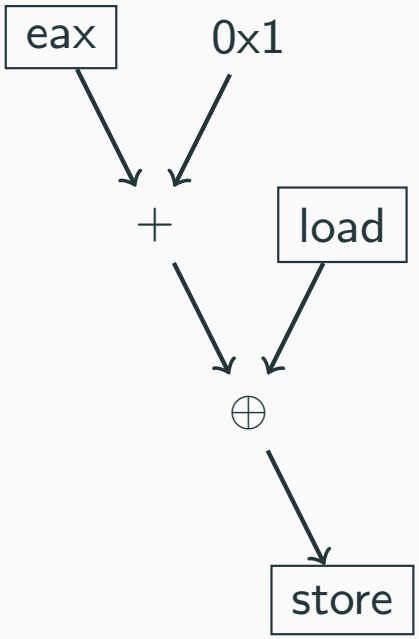


Data Flow: Model

A memory access does not depend on its address.

Justification: two variables accessed with the same pointer are not necessarily related.

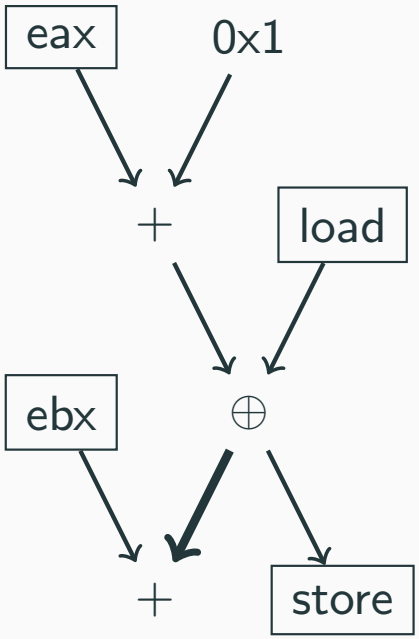
```
1  add eax, 0x1
2  xor [esp], eax
```



Data Flow: Model

A memory read depends on the last value that was written at its address: **load-value dependency**.

```
1  add eax, 0x1
2  xor [esp], eax
3  add ebx, [esp]
```



Data Flow: Construction

Straight Line Code:

Due to performance and security reasons, symmetric cryptographic implementations tend to **avoid** conditional statements.

- We dot not consider implicit dependencies:

```
out = 0;
if (in) out = 1;
```

- The data flow is constructed from a sequence of dynamic instructions.

Data Flow: Construction

Load-value dependencies:

Find the last value that was accessed at a given address. Easier said than done due to possible aliasing:

```
mov [esp], eax
mov [ebp], ebx
mov eax, [esp]
```

We need to compare the address of every memory access.

Load-value dependencies: Static Approach

Try to over-approximate the value of memory pointers (range analysis).

```
mov [esp+0x200], eax
movzx ebx, bl          ; ebx ∈ [0,0xff]
mov eax, [esp+ebx]     ; esp+ebx ≠ esp+0x200
```

Complex analysis with a limited efficiency when conducted locally.

Data Flow: Construction

Load-value dependencies: Dynamic Approach

For a given execution, we save the value of the memory addresses.

Load-value dependencies are perfectly constructed, **but** they reflect a particular execution.

Hypothesis: memory addresses do not depend on input values. [1]

[1] Sbox is a counterexample.

Primitives Identification

Primitives Identification: Goals

- Identify the primitive (type and name).
- Locate its parameters (vertexes in the DFG).
- Dissociate the data flow of the primitive from the data flow of the mode.

I/O relationship: [GWH11; CFM12]

For a code fragment, I denotes the set of values that are read and O the set of values that are written for a given execution.

If $\exists x \in I$ and $\exists y \in O$ such that $f(x) = y$, where f is a cryptographic function, then the code fragment **implements** f.

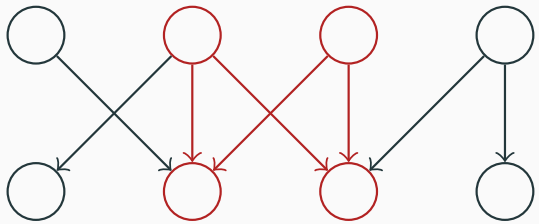
Primitives Identification: Existing Techniques

	Pros	Cons
I/O Relationship	No false positive	High combinatorial complexity
	Easy to implement	Sensitive fragment selection
		No data flow information

Avalanche Effect: [LWC14]

Every part of the input parameter influences **every** part of the output parameter.

Assuming that the parameters are stored in memory, find the largest sets of memory reads and memory writes that verify the avalanche property.

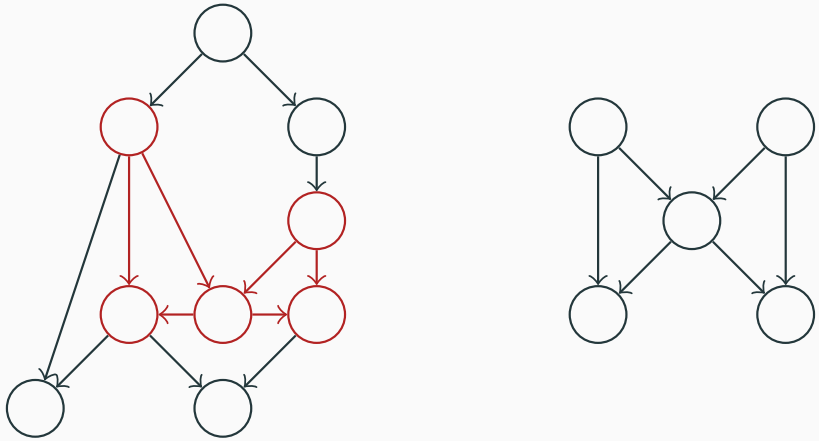


Primitives Identification: Existing Techniques

	Pros	Cons
I/O Relationship	No false positive Easy to implement	High combinatorial complexity Sensitive fragment selection No data flow information
Avalanche Effect	No signature needed	False positives No identification Sensitive fragment selection

Data Flow Isomorphism: [LGF15]

The DFG is used as a signature to identify primitives. The DFG is first normalized using code rewriting rules. Then, it is compared to a list of signatures using a subgraph isomorphism algorithm.



Primitives Identification: Existing Techniques

	Pros	Cons
I/O Relationship	No false positive Easy to implement	High combinatorial complexity Sensitive fragment selection No data flow information
Avalanche Effect	No signature needed	False positives No identification Sensitive fragment selection
▶ **Isomorphism**	Almost no false positive Data flow information	Complex rewriting rules Signatures are hard to create

Slice

Slice: Formal Definition

Definition

Given a DFG $D = (V_D, E_D)$ and a set of cryptographic parameters $P \subset V_D$, a slice $S = (V_S, E_S)$ is the smallest subgraph of D such that $P \subset V_S$ and: $\forall (u, v) \in P^2, dst_D(u, v) = dst_S(u, v)$

We define the distance between two vertices as the number of edges on the shortest **undirected** path (with non-zero dependence mask).

Slice: Completeness - Readability Tradeoff

Complete:

A slice is complete if it contains enough information to identify the mode of operation.

Distance preserving property $\Longrightarrow$ if two parameters are connected in the DFG, they are also connected in the slice.

Slice: Completeness - Readability Tradeoff

Readable:

A slice is readable if it is free of irrelevant element.

- Minimality property $\Longrightarrow$ if an element is not part of a path between two parameters, it will not appear on the slice.
- Distance preserving property $\Longrightarrow$ paths in the slice are the shortest.

Slice: Construction - Greedy Algorithm

Basic Idea:

1. Compute the shortest path for every pair of P^2 (BFS).
2. Take the union of the shortest paths.

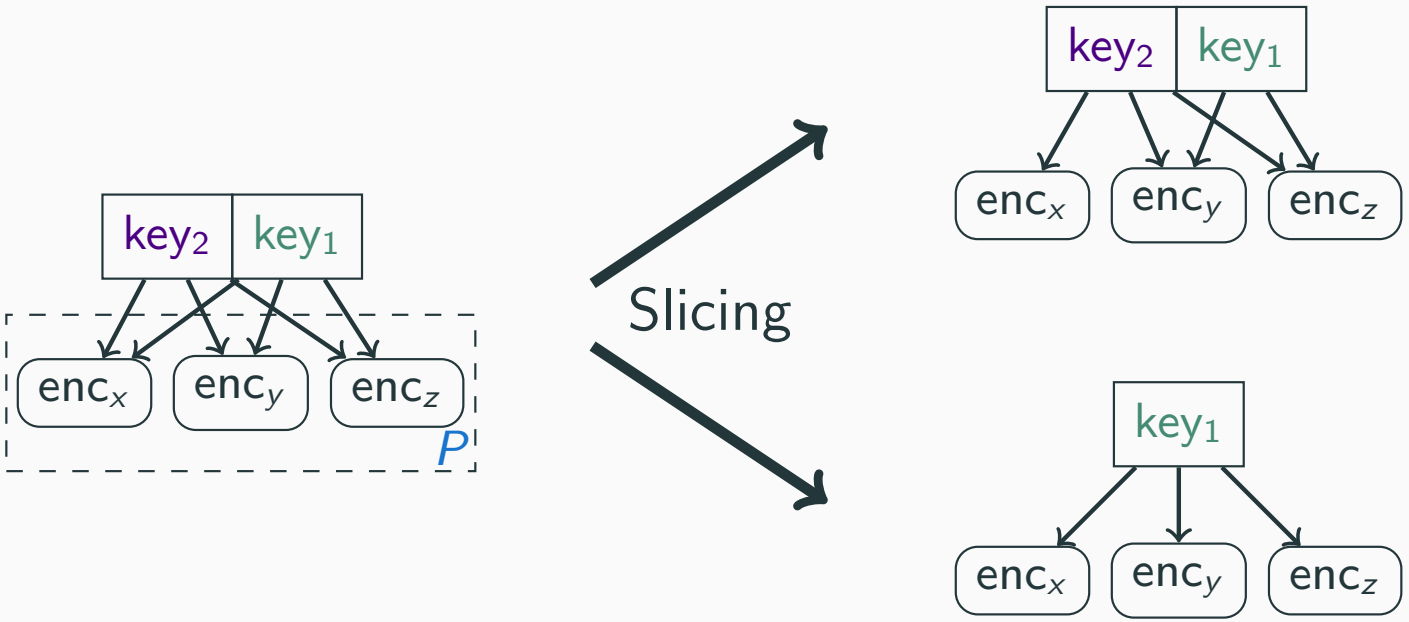
Problems:

- There are several shortest paths. [1]
- Which one to choose to obtain the smallest union?

[1] May be exponential to the number of vertices.

Slice: Construction - Greedy Algorithm

$(enc_x, key_2, enc_j) \cup (enc_x, key_2, enc_z) \cup (enc_y, key_1, enc_z)$



$(enc_x, key_1, enc_j) \cup (enc_x, key_1, enc_z) \cup (enc_y, key_1, enc_z)$

Min Coverage Problem:

Given a collection of sets $\{path_{u,v}, (u,v) \in P^2\}$, pick exactly one element from each set $p_{u,v} \in path_{u,v}$ such that their union $\bigcup p_{u,v}$ is minimal.

Slice: Construction - Greedy Algorithm

for all pairs (u, v) of P^2 **do**

$path_{u,v} = minPath(u, v)$

end for

Initialize S as an empty graph

repeat

pick an unprocessed pair (u, v) such that $|path_{u,v}|$ is minimal

pick a path $p \in path_{u,v}$ such that $|V_S \cup p|$ is minimal

add p to S and mark (u, v) as processed

until all pairs of P^2 have been processed

return S

Slice: Construction - Greedy Algorithm

Complexity: $\mathcal{O}(|V_D|.|P^2|)$ assuming that the number of path returned by *minPath* is capped.

No guarantee of obtaining the smallest graph **but** in practice:

- not every pair of parameters has several shortest paths ;
- the limit on the number of shortest paths is never reached ;
- some sets of shortest paths are disjoint.

Experimental Evaluation

Experimental Evaluation: Methodology

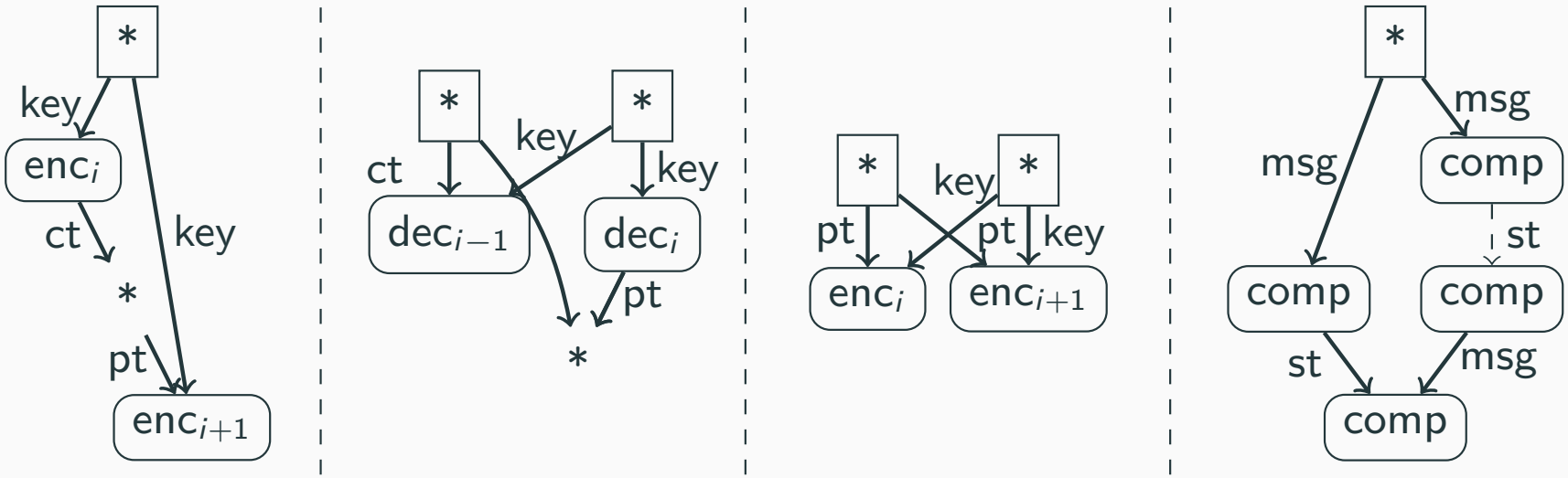
The completeness Cp and the readability Rd are defined as follows:

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

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

S_{opt} is the optimal pattern, Mcs is a function that returns, for a pair of graphs, their maximum common subgraph.

Experimental Evaluation: Ideal Slice S_{opt}



The * label may refer to any path that does not intersect the rest of the graph.

Experimental Evaluation: Results

	CBC	CTR	HMAC
Crypto++ 5.6.1	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 1
TomCrypt 1.17	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 1
Nettle 2.7.1	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 0.71[1,2]
OpenSSL 1.0.1f	Cp = 1, Rd = 1	Cp = 1, Rd = 1	Cp = 1, Rd = 0.83[1]

- Slices are always complete.
- Superfluous elements are not overwhelming.

[1] Both, the inner and the outer hash functions depend on the size of a block.
[2] An aligned memory read retrieves part of the key and part of the message.

Practical Examples

Practical Examples: OpenSSL's RNG

RAND_add: adds an entropy buffer B to the RNG's internal state St.

for $i = 0$ to n **do**

$md_{i+1} = sha1(md_i \| B_i \| St_{i+j} \| c_1 \| c_2)$

$St_{i+j} = St_{i+j} \oplus md_{i+1}$

$c_2 = c_2 + 1$

end for

B and St are divided into 20-byte blocks. n is the number of blocks in B. c_1 and c_2 are two 32-bit counters.

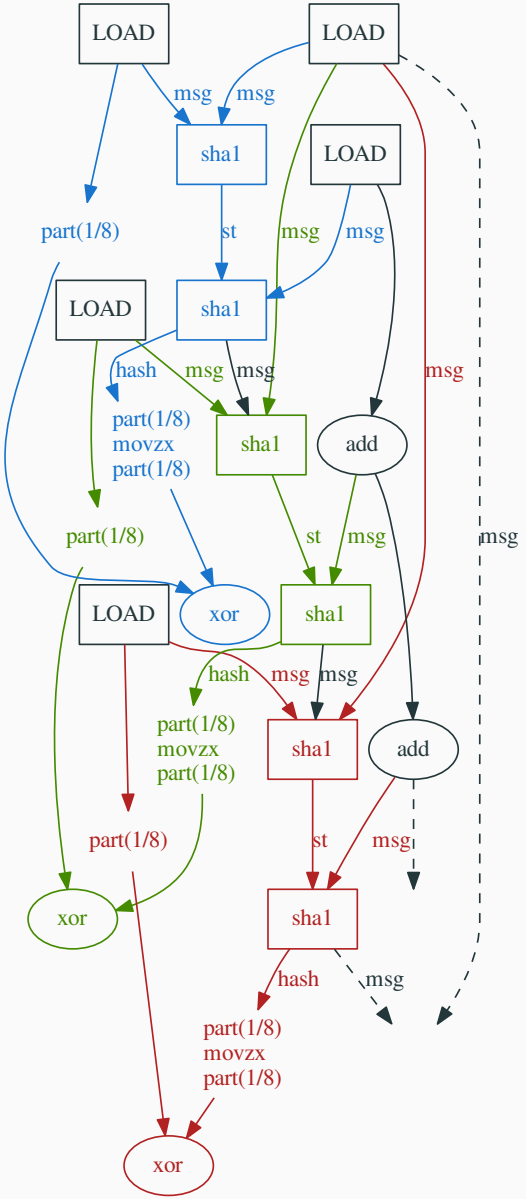
Practical Examples: OpenSSL's RNG

$$md_1 = sha1(md_0 \| B_0 \| St_j \| c_1 \| c_2)$$
$$St_j = St_j \oplus md_1$$
$$c_2 = c_2 + 1$$

$$md_2 = sha1(md_1 \| B_1 \| St_{j+1} \| c_1 \| c_2)$$
$$St_{j+1} = St_{j+1} \oplus md_2$$
$$c_2 = c_2 + 1$$

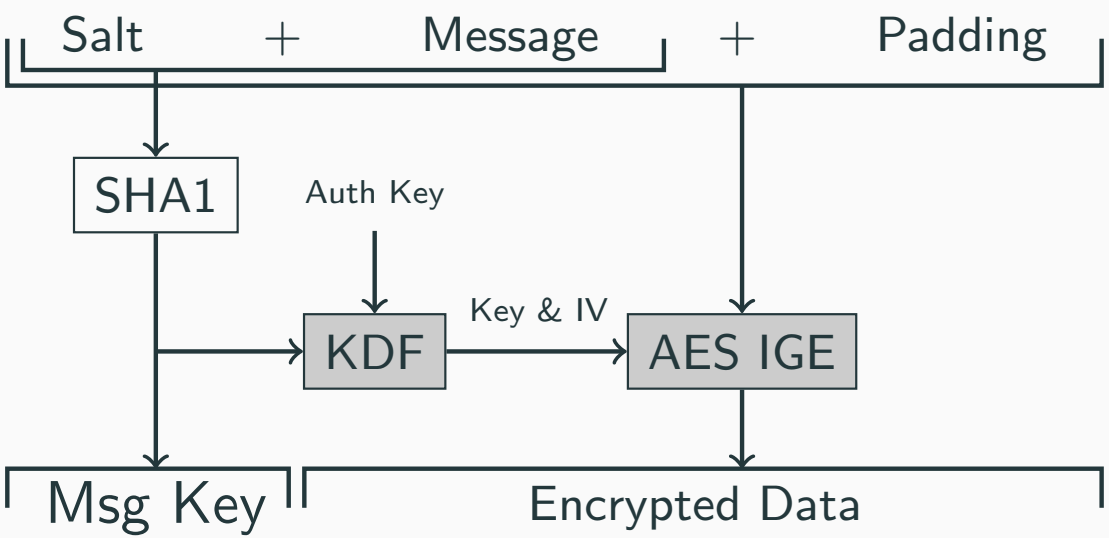
$$md_3 = sha1(md_2 \| B_2 \| St_{j+2} \| c_1 \| c_2)$$
$$St_{j+2} = St_{j+2} \oplus md_3$$
$$c_2 = c_2 + 1$$

...

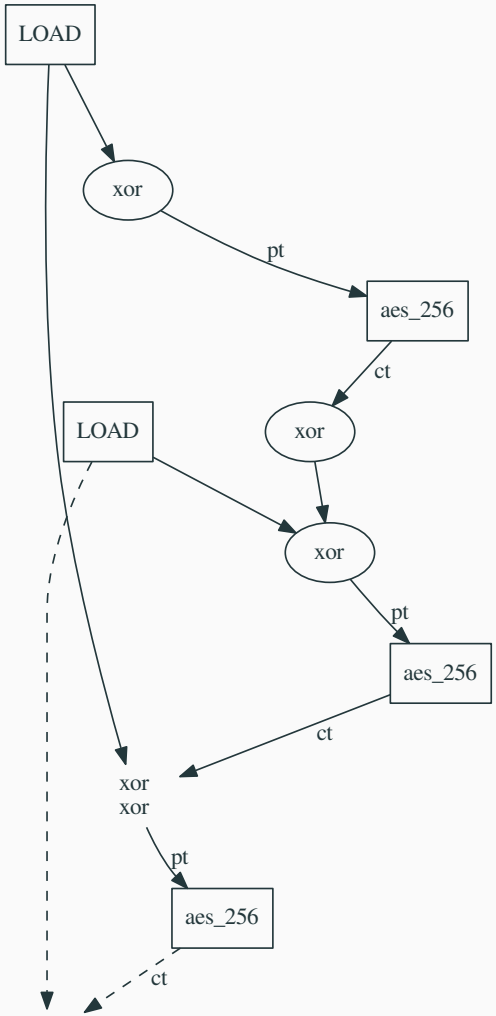


Practical Examples: Instant Messaging Application

Telegram is an instant messaging service that uses a custom encryption scheme called MtProto:



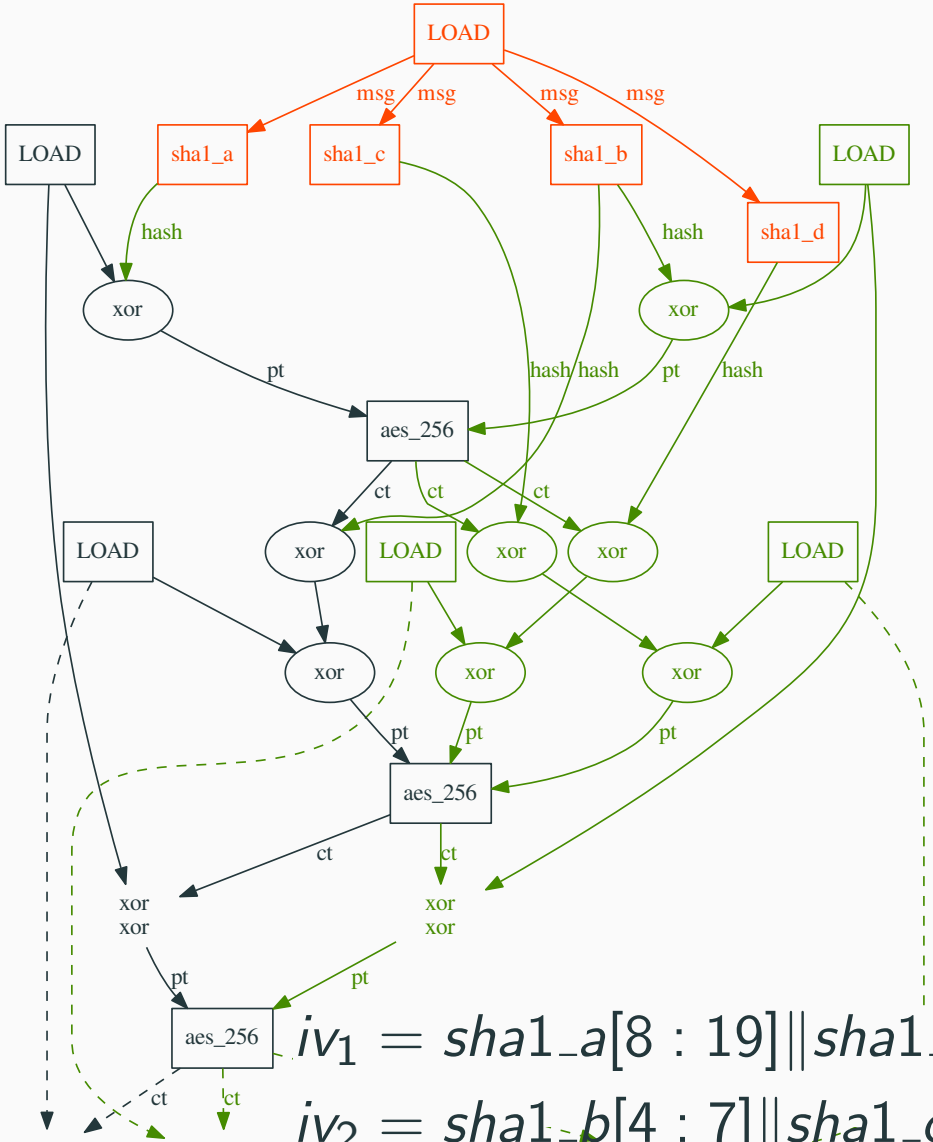
Practical Examples: Instant Messaging Application



IGE encryption:

$$C[i] = E_k(M[i] \oplus C[i-1]) \oplus M[i-1]$$

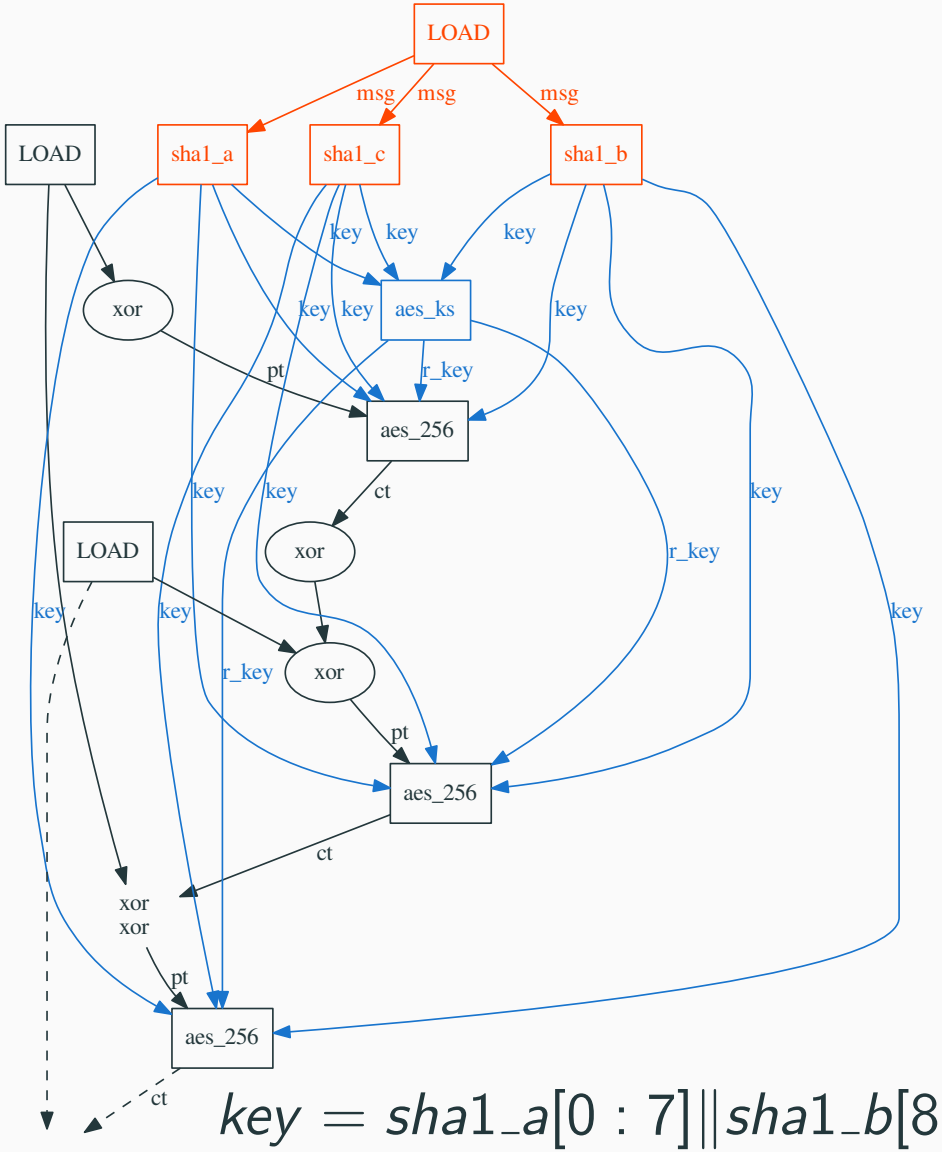
Practical Examples: Instant Messaging Application



$$iv_1 = sha1\_a[8:19] || sha1\_b[0:3]$$

$$iv_2 = sha1\_b[4:7] || sha1\_c[16:19] || sha1_d[0:7]$$

Practical Examples: Instant Messaging Application



$$key = sha1\_a[0:7] \| sha1\_b[8:19] \| sha1\_c[4:15]$$

Conclusion

Conclusion:

Summary: our solution takes as input an execution trace and produces a synthetic representation of the data transfers within the mode of operation.

This representation is a **tradeoff** between completeness and readability.

It should be profitable to reverse engineer mode of operation's implementations.