# Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism

Pierre Lestringant
*AMOSSYS*

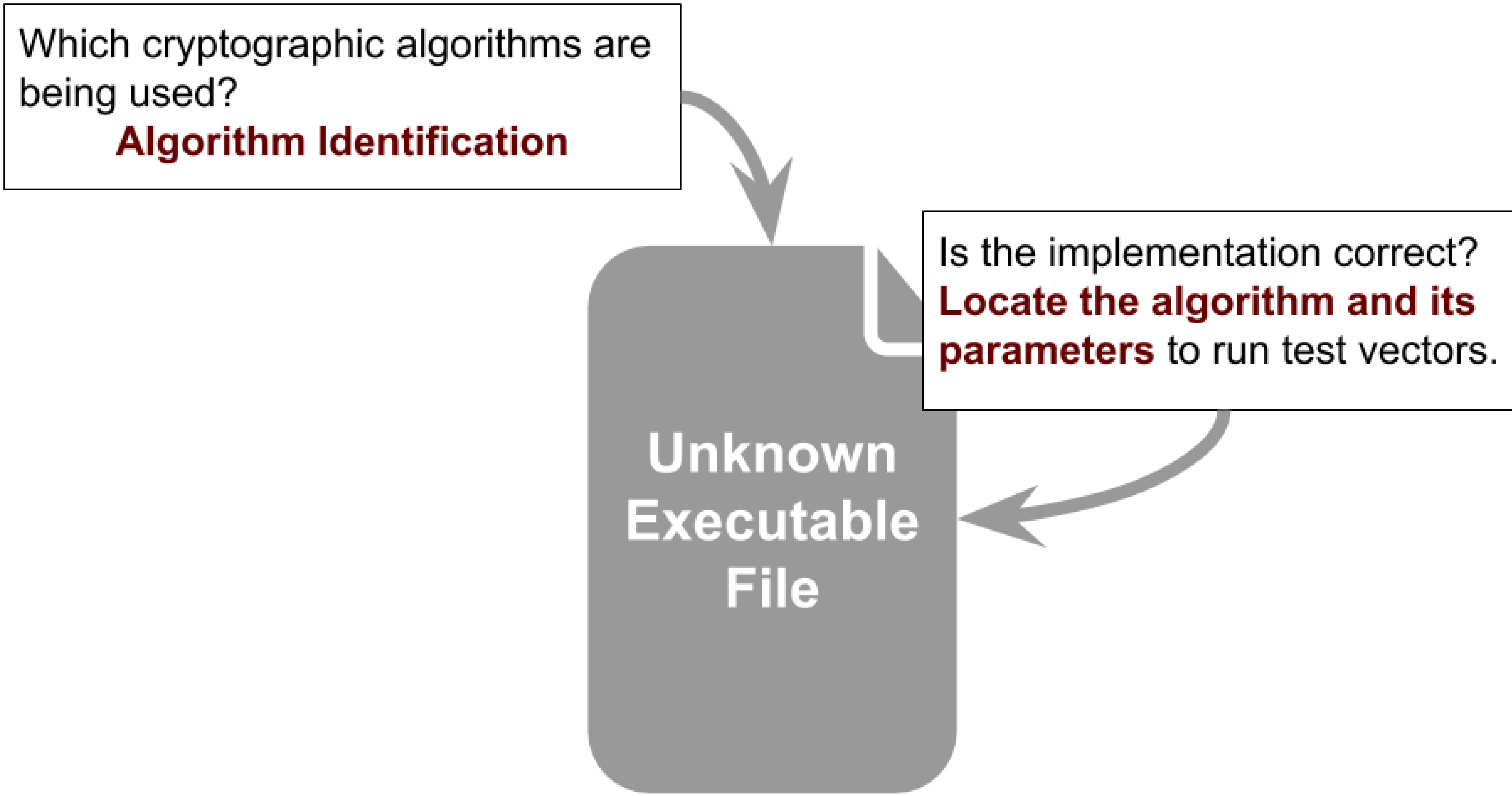
Frédéric Guihéry
*AMOSSYS*

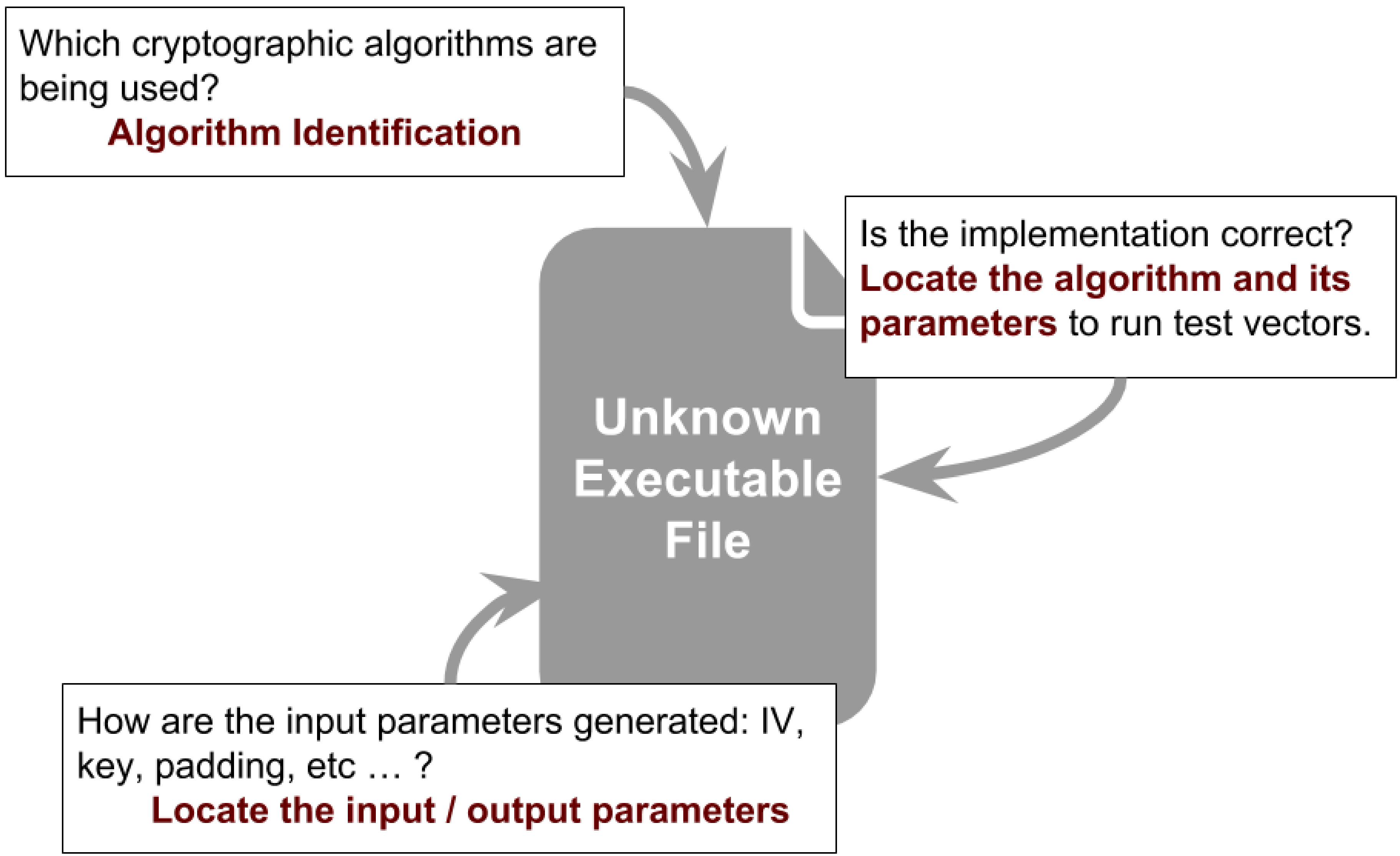
Pierre-Alain Fouque
*Rennes 1 University*

Unknown
Executable
File

Which cryptographic algorithms are being used?

**Algorithm Identification**


Which cryptographic algorithms are being used?
Algorithm Identification
Is the implementation correct?
Locate the algorithm and its parameters to run test vectors.
Unknown Executable File


Which cryptographic algorithms are being used?
Algorithm Identification
Is the implementation correct?
Locate the algorithm and its parameters to run test vectors.
Unknown Executable File
How are the input parameters generated: IV, key, padding, etc … ?
Locate the input / output parameters

# Related Work

# Related Work: Statistical Approach

A statistical model can involve the following features:

- Mnemonics
- Control Flow Graph
- Data Constant

This approach is simple and **efficient** but the **result is not precise enough**.

# Related Work: Input / Output Approach


If a code fragment *C* reads a value *i* and writes a value *o* such that *f(i) = o* then we conclude that *C* implements function *f*.

# Related Work: Input / Output Approach


If a code fragment *C* reads a value *i* and writes a value *o* such that *f(i) = o* then we conclude that *C* implements function *f*.

**Two open problems:**

- Code fragment must be precise.
  → **How can we extract precise code fragments?**
- Parameters are often fragmented.
  → **How can we regroup fragmented parameters?**

# Contribution

# Problem Scope & Hypothesis

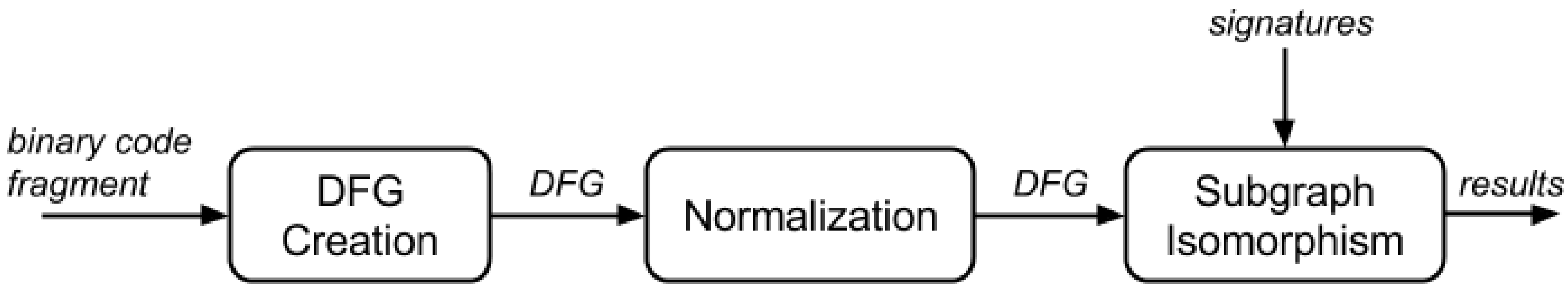
**Problem Scope**:

- Symmetric Cryptography
- No obfuscation
- To be applied on preselected code fragments (in practice must be used with a front end filter).

**Hypothesis:**

**Straight-line code** (loops are unrolled, function calls are inlined, no conditional branch).

# Solution Overview



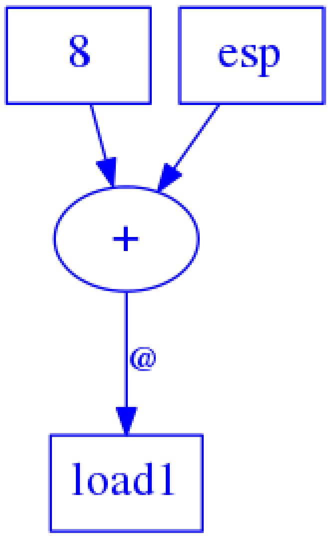
# Data Flow Graph (DFG)

A **DFG** represents the data dependencies between operations. A node is either an operation or an input value. An edge from $v_1$ to $v_2$ means that the result of $v_1$ is used by $v_2$.

- Convenient representation to rewrite the program code
- Easy to extract specific subset of related operations

Consider the toy cipher defined by the following formula:

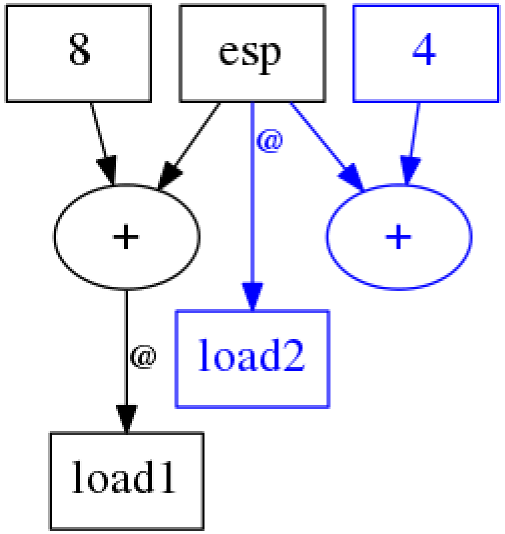
$$c = S(p+k)+k$$




```
mov     ebx,DWORD PTR [esp+0x8]     ; load key
pop     ecx                         ; load plaintext
xor     ebx,ecx                     ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]  ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]     ; load key
xor     ecx,ebx                     ; = S(p ^ k) ^ k
```

Consider the toy cipher defined by the following formula:

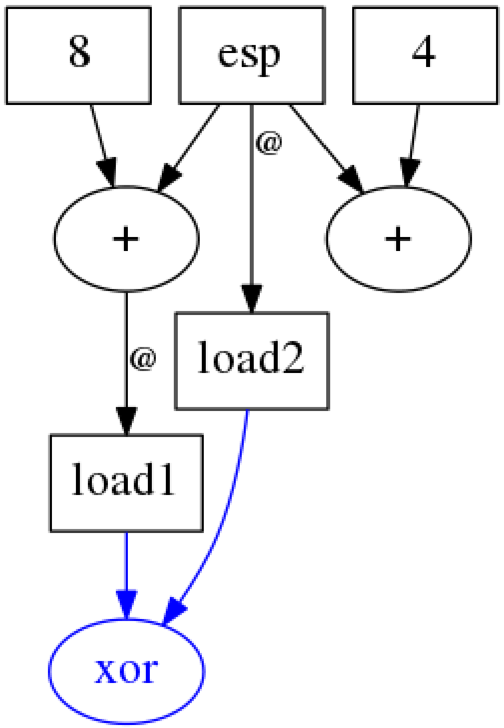
$$c = S(p+k)+k$$




```
mov     ebx,DWORD PTR [esp+0x8]     ; load key
pop     ecx                         ; load plaintext
xor     ebx,ecx                     ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]  ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]     ; load key
xor     ecx,ebx                     ; = S(p ^ k) ^ k
```

Consider the toy cipher defined by the following formula:

$$c = S(p+k)+k$$

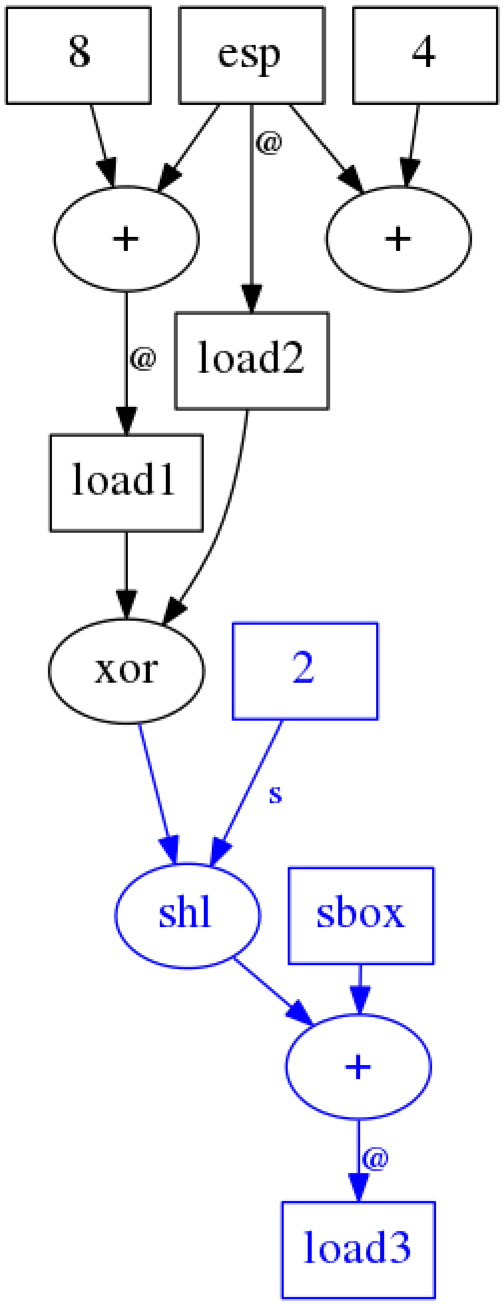



```
mov     ebx,DWORD PTR [esp+0x8]        ; load key
pop     ecx                            ; load plaintext
xor     ebx,ecx                        ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]     ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]        ; load key
xor     ecx,ebx                        ; = S(p ^ k) ^ k
```

Consider the toy cipher defined by the following formula:

$$c = S(p+k)+k$$

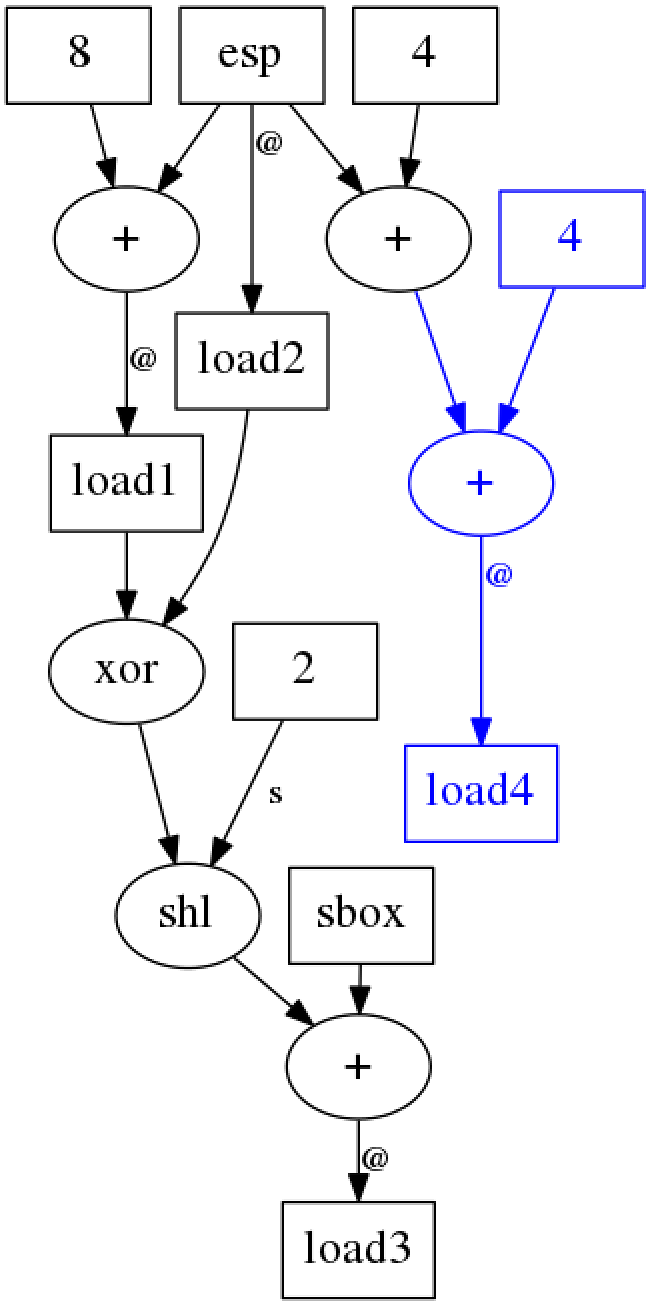
```
mov     ebx,DWORD PTR [esp+0x8]     ; load key
pop     ecx                         ; load plaintext
xor     ebx,ecx                     ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]  ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]     ; load key
xor     ecx,ebx                     ; = S(p ^ k) ^ k
```



Consider the toy cipher defined by the following formula:

$$c = S(p+k)+k$$

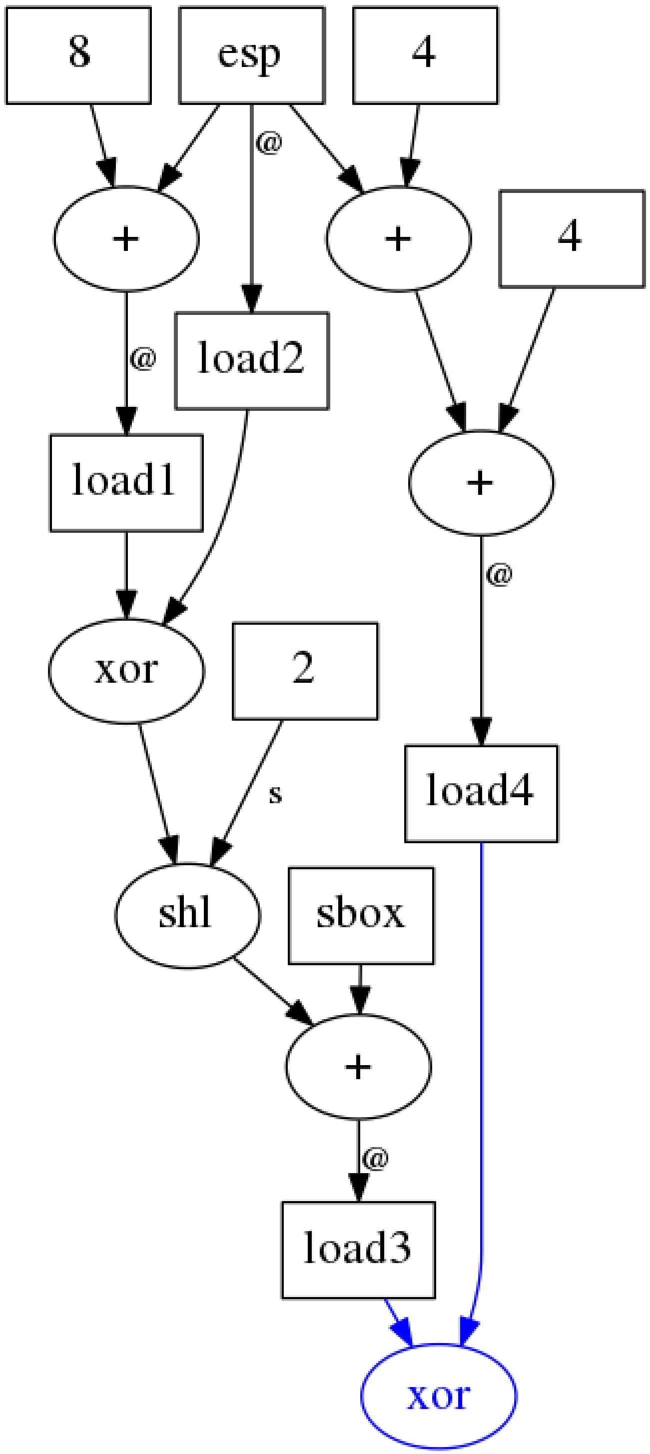
```
mov     ebx,DWORD PTR [esp+0x8]      ; load key
pop     ecx                          ; load plaintext
xor     ebx,ecx                      ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]   ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]      ; load key
xor     ecx,ebx                      ; = S(p ^ k) ^ k
```



Consider the toy cipher defined by the following formula:

$$c = S(p+k)+k$$

```
mov     ebx,DWORD PTR [esp+0x8]      ; load key
pop     ecx                          ; load plaintext
xor     ebx,ecx                      ; = p ^ k
mov     ecx,DWORD PTR [SBOX+ebx*4]   ; = S(p ^ k)
mov     ebx,DWORD PTR [esp+0x4]      ; load key
xor     ecx,ebx                      ; = S(p ^ k) ^ k
```



# Normalization

Modify the DFG with a set of **rewrite rules** to maximize the chance of finding the algorithm's signature.

Rewrite rules are applied until a fixed point is reached.

Four categories of rewrite rules:

- Constant simplification
- Subexpression elimination
- Memory simplification
- Operation rewriting

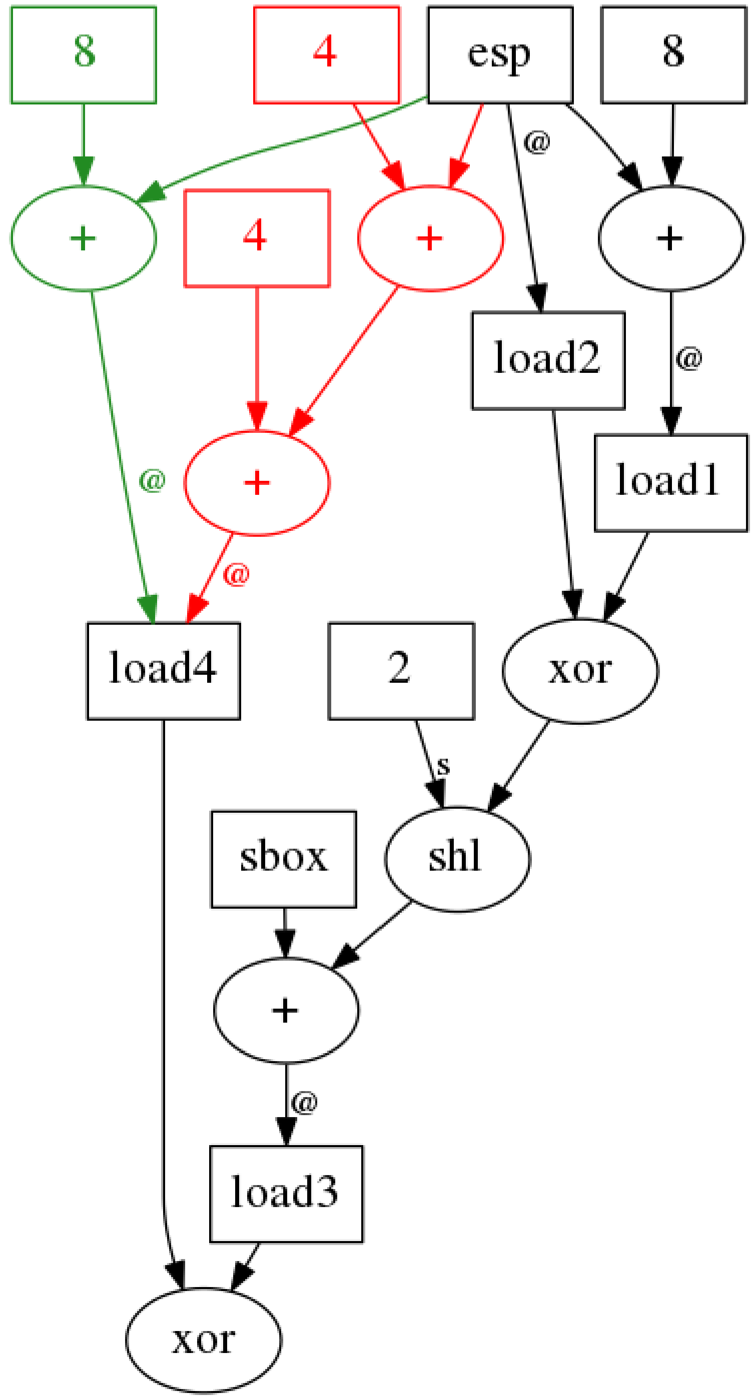
# Normalization: Constant Simplification

**Constant simplification** is performed in the following cases:

- Every operand has a known value
- An operand is equal to the identity / absorbing element

To maximize the number of constant simplifications:

- Rearrange sequence of associative operations
- Distribute


8
4
esp
8
+
4
+
+
@
load2
@
load1
+
@
@
load4
2
xor
s
sbox
shl
+
@
load3
xor

# Common Subexpression Elimination

If two operations share the same operands, they will produce the same result. They are redundant and one of them can be removed.

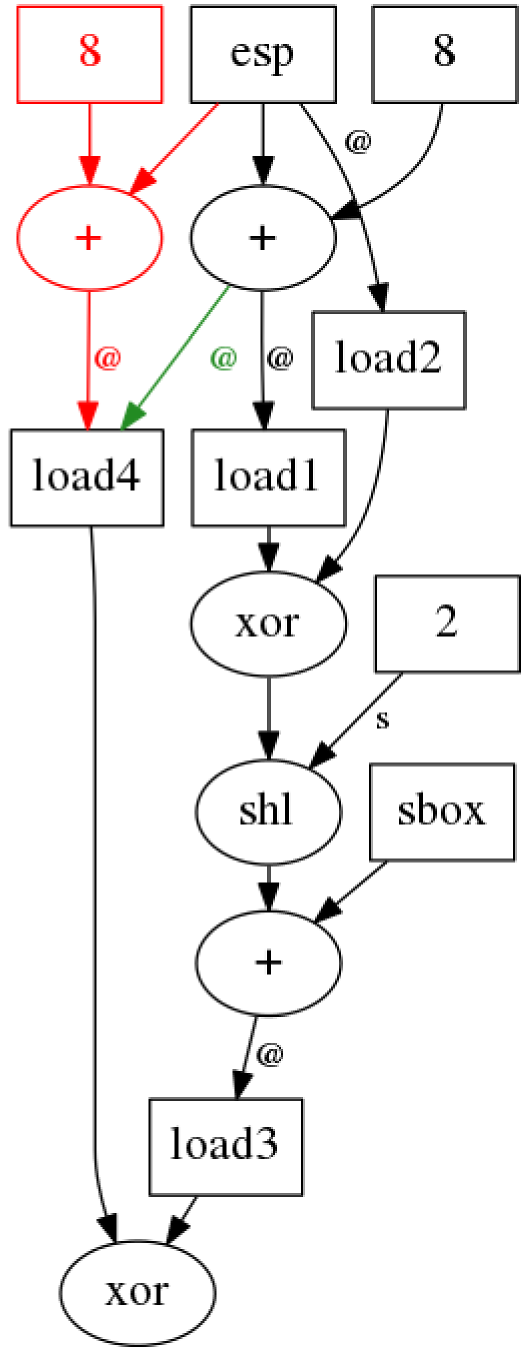
**Goals:**

- Deals with not optimized code (amplified by macros)

```
#define ROR(x, n) (((x) >> (n)) | ((x) >> (32 - (n))))
c = ROR(a + b, 5);
```

- Simplify effective address computation

```
mov eax,DWORD PTR [esp+edx*4+0x8]
mov ebx,DWORD PTR [esp+edx*4+0x8]
```


8
esp
8
@
+
+
@
@
@
load2
load4
load1
xor
2
s
shl
sbox
+
@
load3
xor

# Normalization: Memory Simplification

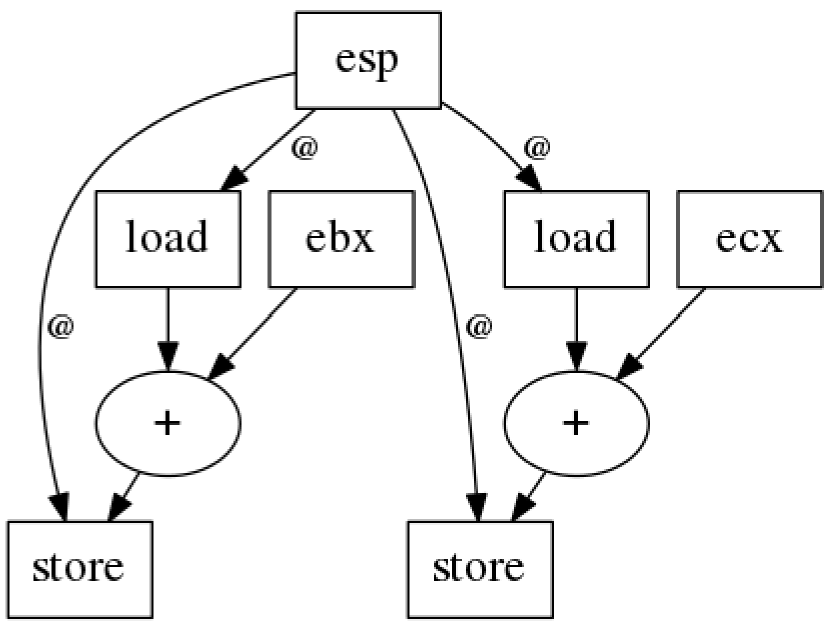
Register allocation is highly variable across different instances of a same algorithm.

```
add DWORD PTR [esp],ebx
; [...]
add DWORD PTR [esp],ecx
```

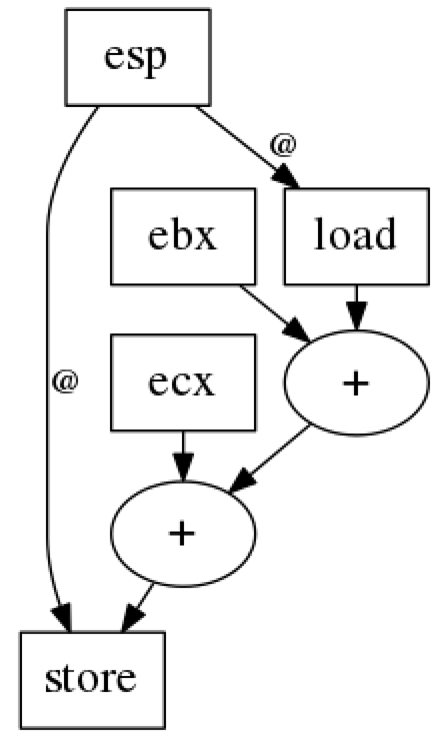
```
mov eax,DWORD PTR [esp]
add eax,ebx
; [...]
add eax,ecx
mov DWORD PTR [esp],eax
```

```
a = a + b;
/* [...] */
a = a + c;
```

# Normalization: Memory Simplification




```
add DWORD PTR [esp],ebx
; [...]
add DWORD PTR [esp],ecx
```




```
mov eax,DWORD PTR [esp]
add eax,ebx
; [...]
add eax,ecx
mov DWORD PTR [esp],eax
```

```
a = a + b;
/* [...] */
a = a + c;
```

# Normalization: Memory Simplification

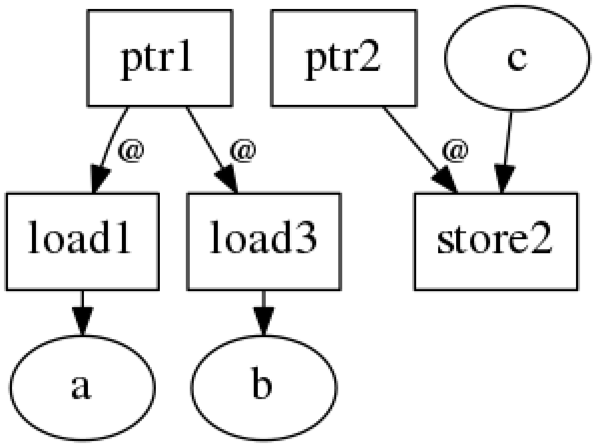
For a given address, the sequence of memory operations can be simplified in the following cases:

$\ldots, \text{Load}_n, \text{Load}_{n+1}, \ldots \rightarrow \ldots, \text{Load}_n, \ldots$

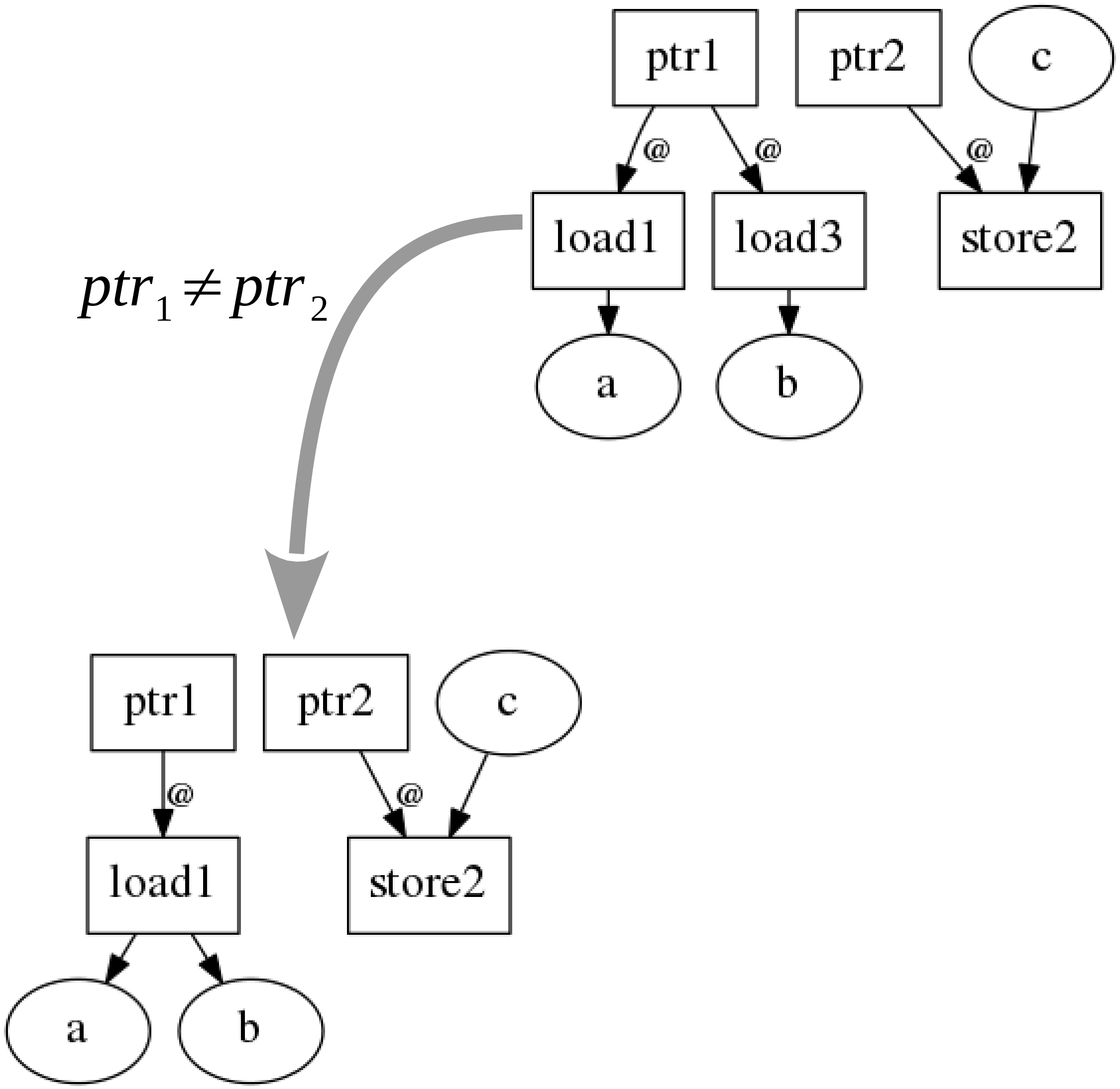
$\ldots, \text{Store}_n, \text{Store}_{n+1}, \ldots \rightarrow \ldots, \text{Store}_{n+1}, \ldots$

$\ldots, \text{Store}_n, \text{Load}_{n+1}, \ldots \rightarrow \ldots, \text{Store}_n, \ldots$

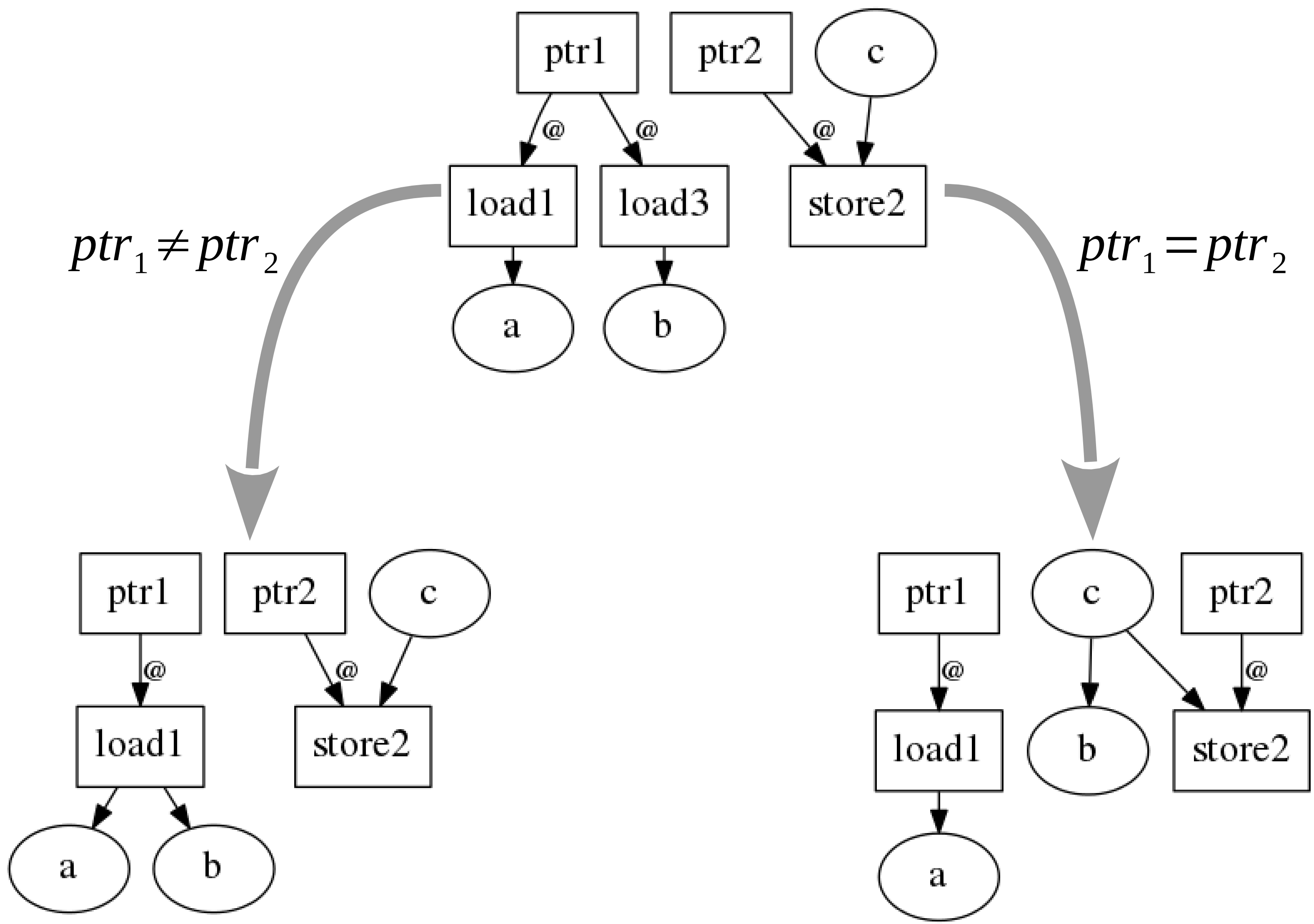
# Aliasing Issue

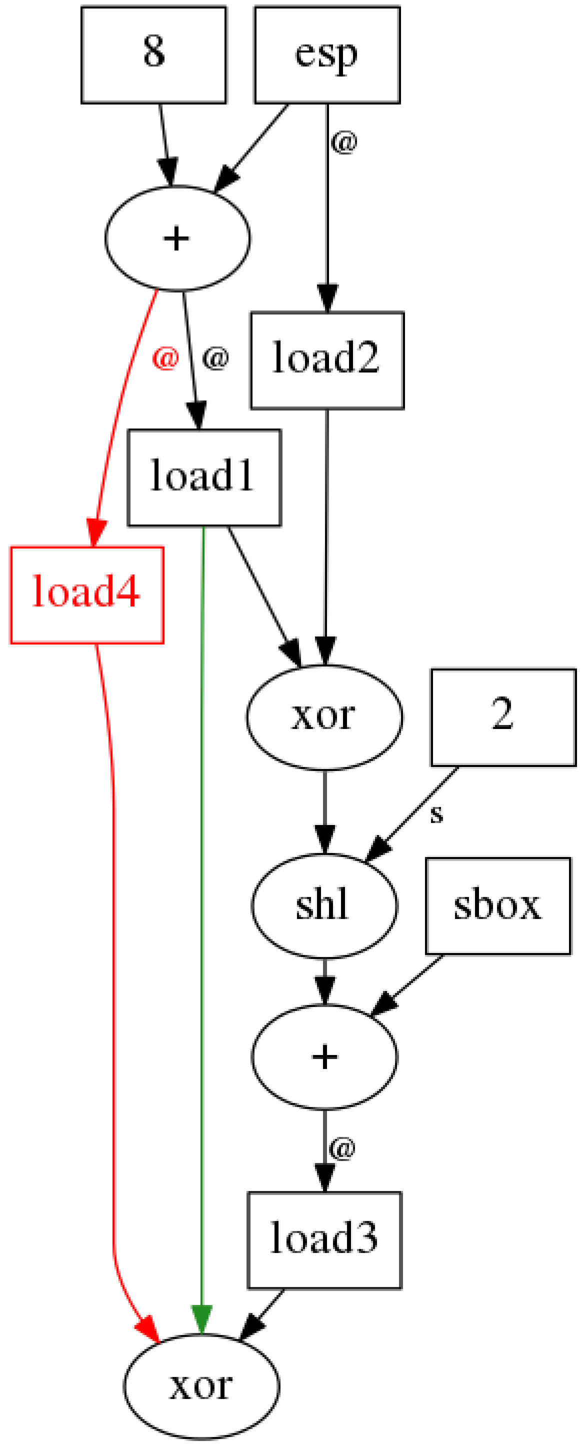


# Aliasing Issue



# Aliasing Issue




8
esp
@
+
@
@
load2
load1
load4
xor
2
s
shl
sbox
+
@
load3
xor

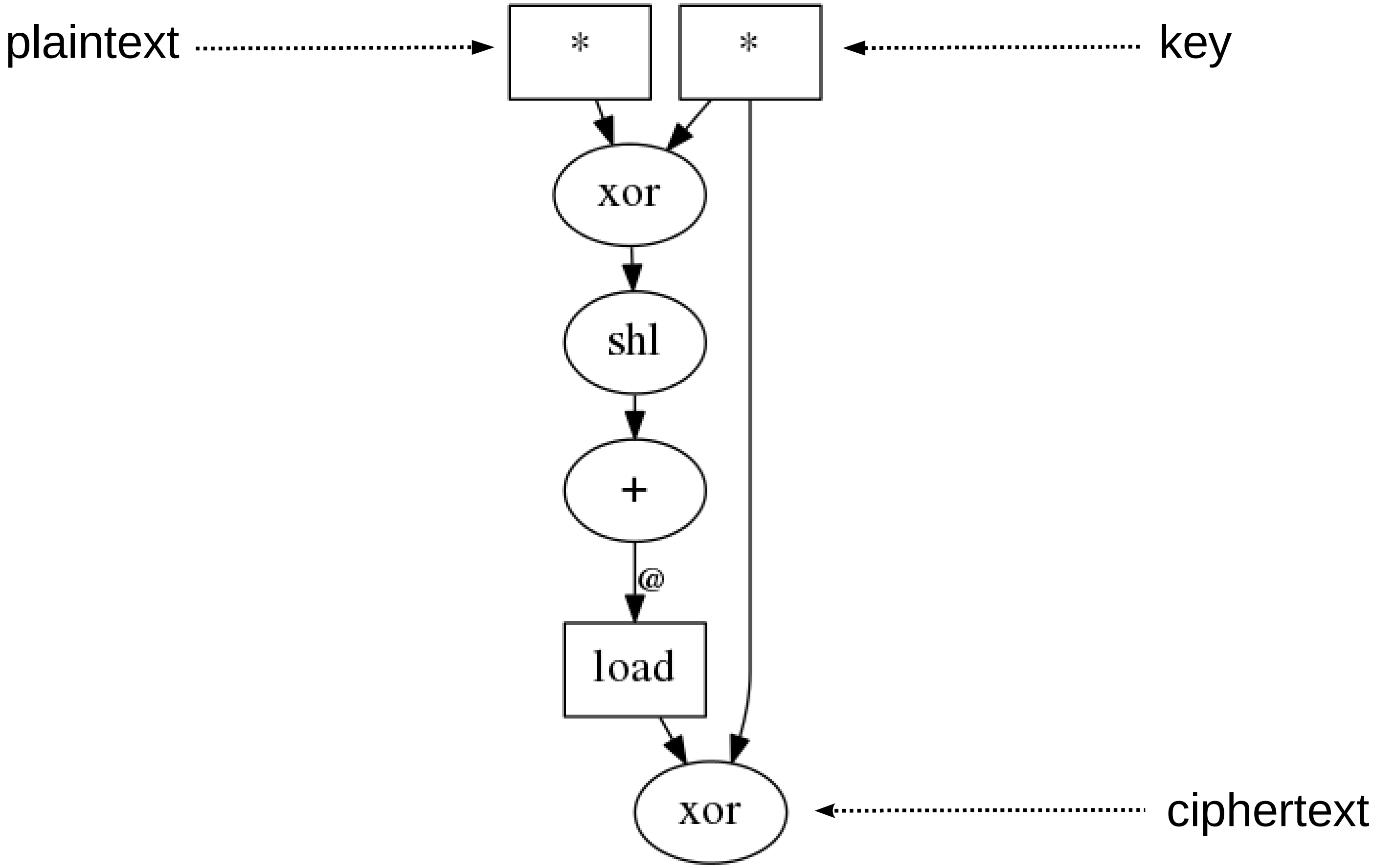
# Signature

A **signature** is a distinctive subgraph that is contained in the normalized DFG of every instance of an algorithm.

- Ideally, one signature per algorithm
- Signatures should cover as much of the algorithm as possible

  → *in particular should contain the IO parameters*
- Macro signature allows to combine signature together
- Signature creation is still a manual process

$$c = S(p + k) + k$$

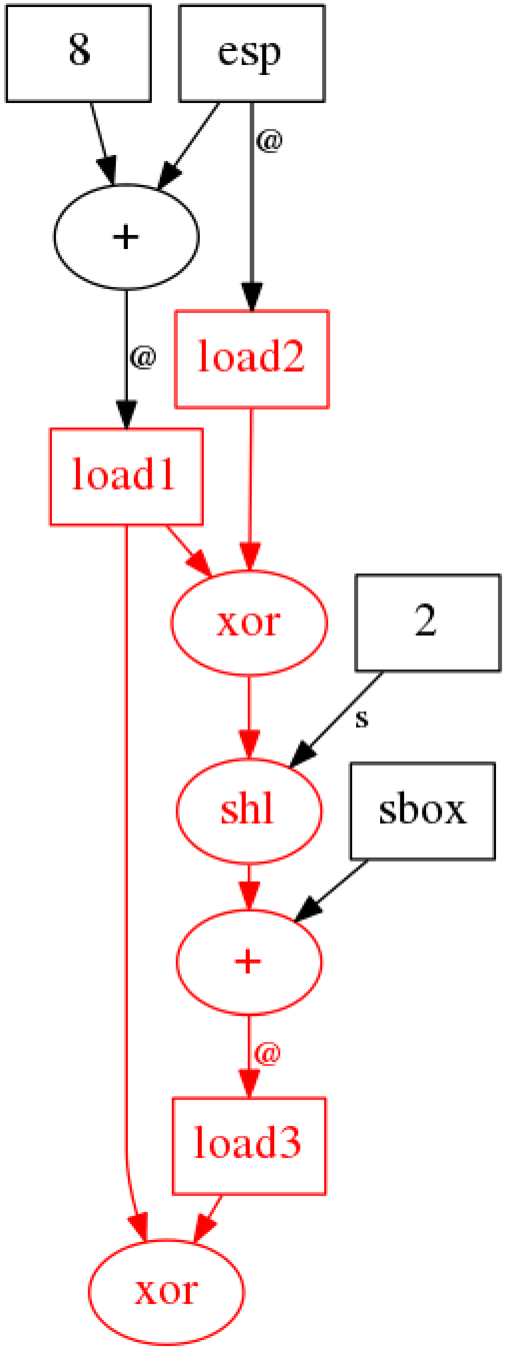


# Subgraph Isomorphism

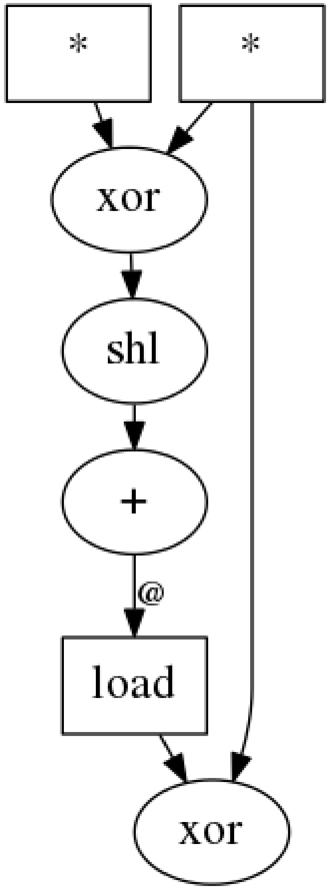
A graph $G_1 = (V_1,E_1)$ is isomorphic to a subgraph of $G_2=(V_2,E_2)$ if there is an injection $f:V_1 \rightarrow V_2$ such that:

$$\forall v_i, v_j \in V_1 if \left(v_i, v_j\right) \in E_1 then \left(f\left(v_i\right), f\left(v_j\right)\right) \in E_2$$

We use Ullman algorithm to find every subgraph of the DFG that are isomorph to the signature.




$$c = S(p+k)+k$$




Toy cipher's signature

# Experimental Evaluation

We have evaluated our solution for three cryptographic algorithms:

**XTEA**, **MD5**, **AES**

We performed tests on **synthetic samples**:

- Thorough evaluation in a well controlled environment
- Larger programs require efficient fragment extraction, which is not directly addressed by this work.

The straight line code requirement is obtained using **DBI**.

# Experimental Evaluation: Compilation

| | | GCC 4.9.1 *(Linux 32-bit)* | Clang 3.5.0 *(Linux 32-bit)* | MSVC 17.00 *(Windows 32-bit)* |
|---|---|---|---|---|
| XTEA *(Wikipedia's implementation)* | -o0 | **ok** | **ok** | **ok** |
| | -o1 | **ok** | **ok** | **-** |
| | -o2 | **ok** | **ok** | **ok** |
| | -o3 | **ok** | **ok** | **-** |
| MD5 *(RFC's implementation)* | -o0 | **ok** | **ok** | **partial** |
| | -o1 | **ok** | **ok** | **-** |
| | -o2 | **ok** | **ok** | **partial** |
| | -o3 | **ok** | **ok** | **-** |
| AES *(Gladman's implementation)* | -o0 | **ok** | **ok** | **ok** |
| | -o1 | **ok** | **ok** | **-** |
| | -o2 | **ok** | **ok** | **ok** |
| | -o3 | **ok** | **ok** | **-** |

It fails for the second message chunk, due to rotation and constant folding.

# Experimental Evaluation: Libraries

The libraries were used as configured and compiled in their respective Debian packages.

| | LibTomCrypt *(version 1.17)* | Crypto++ *(version 5.6.1)* | Openssl *(version 1.0.1f)* | Botan *(version 1.10.8)* |
|---|---|---|---|---|
| XTEA | **ok** | **ok** | **-** | **ok** |
| MD5 | **ok** | **ok** | **ok** | **ok** |
| AES | **ok** | **~ok** | **nok** | **ok** |

SSE instructions not yet supported by our implementation

# Conclusion

**Conclusion:**

- New approach to identify and locate cryptographic algorithms
- Robust due to the normalization step and the macro signatures

**Future work:**

- Cover block cipher **modes of operation** by leveraging the concept of macro signature.
- **Public key cryptography**
- **Automatically generate signature**
- Deal with **obfuscated code**

# Questions?