Cryptographic Function Identification in Obfuscated Binary Programs

REcon 2012

Joan Calvet – j04n.calvet@gmail.com

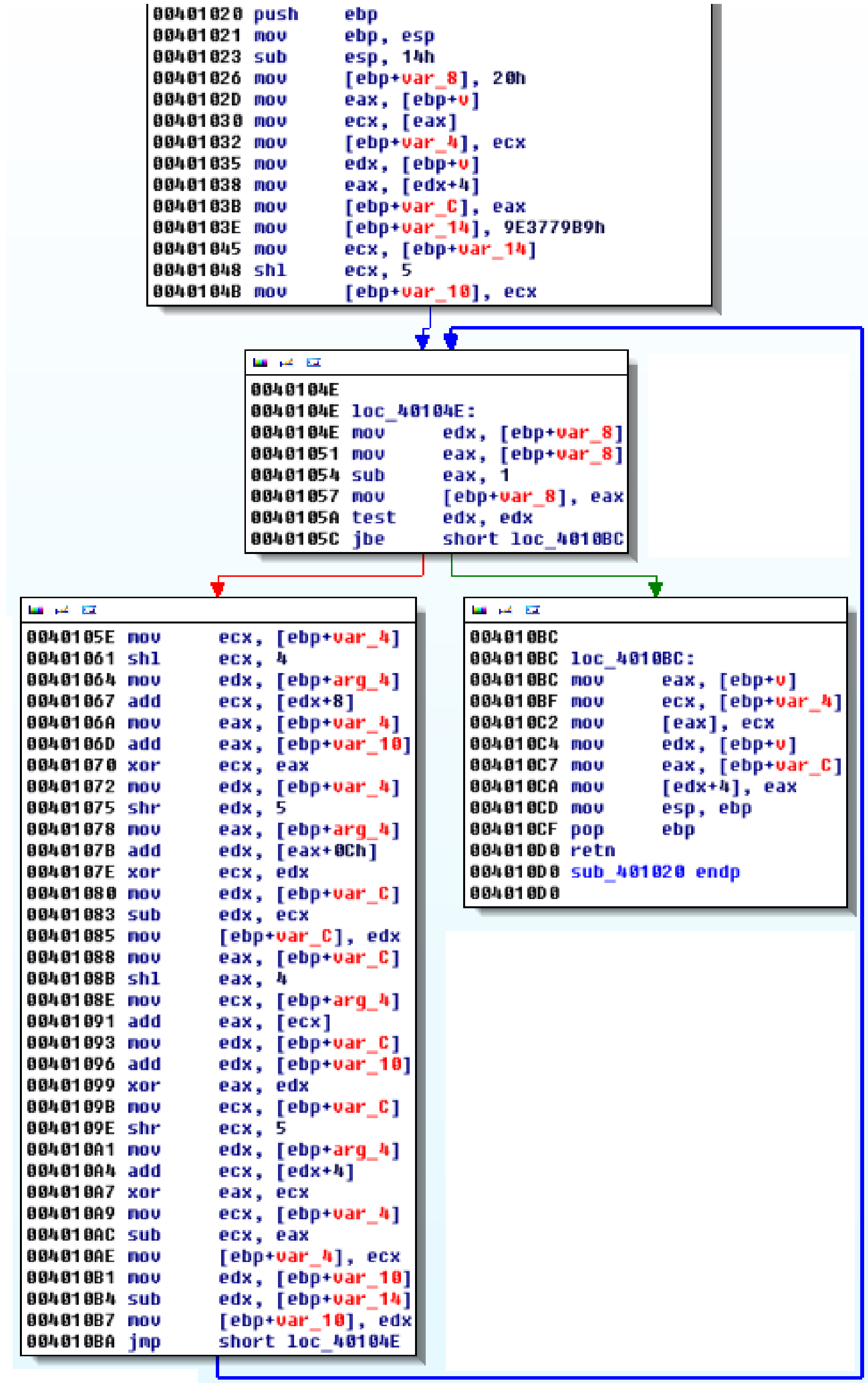
Loria
Laboratoire lorrain de recherche en informatique et ses applications

École Polytechnique Montréal

Presentation Outline

- Introduction to the Problem
- Proposed Solution
- Examples
- What's Next ?

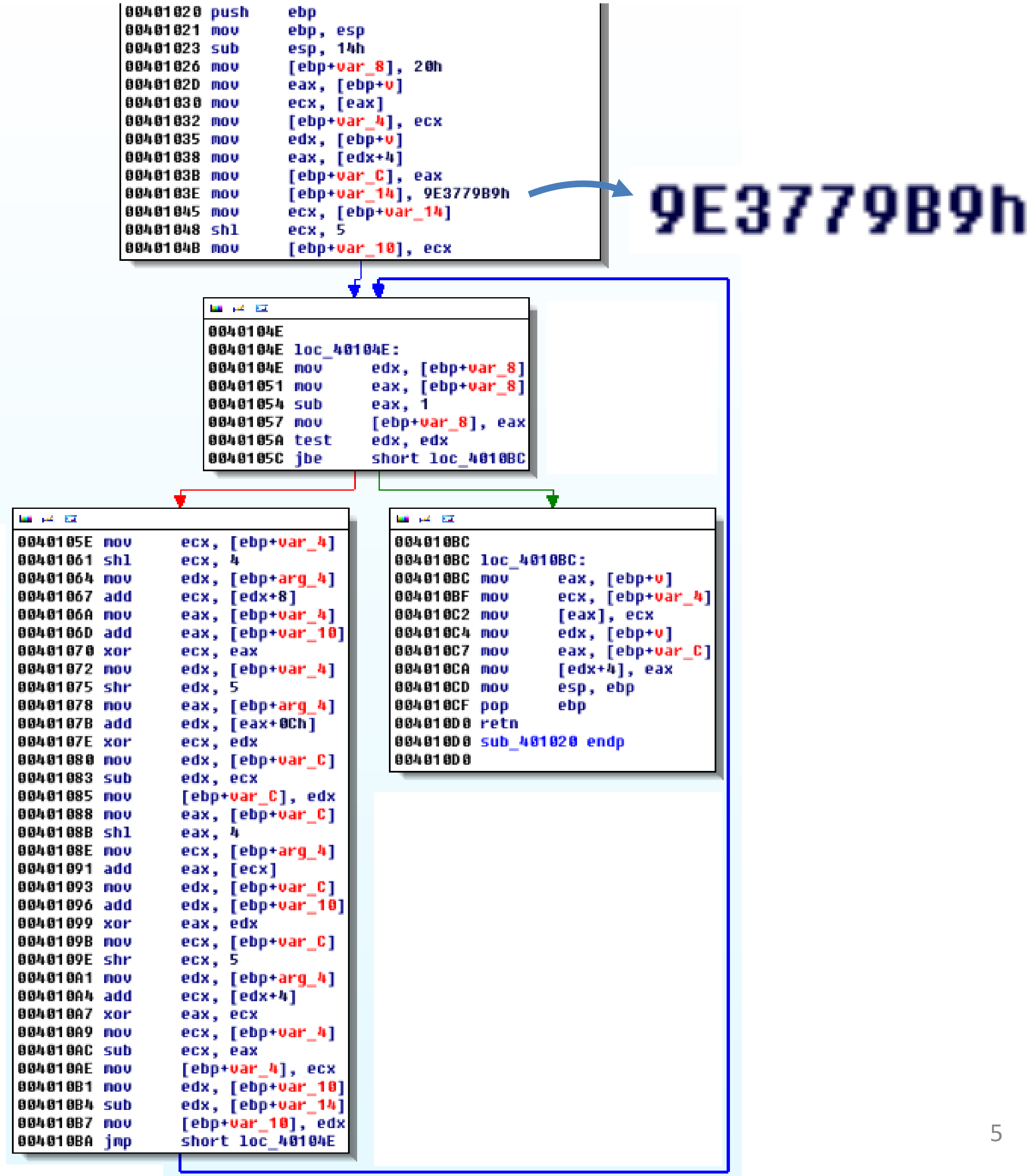
INTRODUCTION TO THE PROBLEM



00401020 push ebp
00401021 mov ebp, esp
00401023 sub esp, 14h
00401026 mov [ebp+var_8], 20h
0040102D mov eax, [ebp+v]
00401030 mov ecx, [eax]
00401032 mov [ebp+var_4], ecx
00401035 mov edx, [ebp+v]
00401038 mov eax, [edx+4]
0040103B mov [ebp+var_C], eax
0040103E mov [ebp+var_14], 9E3779B9h
00401045 mov ecx, [ebp+var_14]
00401048 shl ecx, 5
0040104B mov [ebp+var_10], ecx
0040104E
0040104E loc_40104E:
0040104E mov edx, [ebp+var_8]
00401051 mov eax, [ebp+var_8]
00401054 sub eax, 1
00401057 mov [ebp+var_8], eax
0040105A test edx, edx
0040105C jbe short loc_4010BC
0040105E mov ecx, [ebp+var_4]
00401061 shl ecx, 4
00401064 mov edx, [ebp+arg_4]
00401067 add ecx, [edx+8]
0040106A mov eax, [ebp+var_4]
0040106D add eax, [ebp+var_10]
00401070 xor ecx, eax
00401072 mov edx, [ebp+var_4]
00401075 shr edx, 5
00401078 mov eax, [ebp+arg_4]
0040107B add edx, [eax+0Ch]
0040107E xor ecx, edx
00401080 mov edx, [ebp+var_C]
00401083 sub edx, ecx
00401085 mov [ebp+var_C], edx
00401088 mov eax, [ebp+var_C]
0040108B shl eax, 4
0040108E mov ecx, [ebp+arg_4]
00401091 add eax, [ecx]
00401093 mov edx, [ebp+var_C]
00401096 add edx, [ebp+var_10]
00401099 xor eax, edx
0040109B mov ecx, [ebp+var_C]
0040109E shr ecx, 5
004010A1 mov edx, [ebp+arg_4]
004010A4 add ecx, [edx+4]
004010A7 xor eax, ecx
004010A9 mov ecx, [ebp+var_4]
004010AC sub ecx, eax
004010AE mov [ebp+var_4], ecx
004010B1 mov edx, [ebp+var_10]
004010B4 sub edx, [ebp+var_14]
004010B7 mov [ebp+var_10], edx
004010BA jmp short loc_40104E
004010BC
004010BC loc_4010BC:
004010BC mov eax, [ebp+v]
004010BF mov ecx, [ebp+var_4]
004010C2 mov [eax], ecx
004010C4 mov edx, [ebp+v]
004010C7 mov eax, [ebp+var_C]
004010CA mov [edx+4], eax
004010CD mov esp, ebp
004010CF pop ebp
004010D0 retn
004010D0 sub_401020 endp
004010D0

What's this ?

What’s this ?


00401020 push ebp
00401021 mov ebp, esp
00401023 sub esp, 14h
00401026 mov [ebp+var_8], 20h
0040102D mov eax, [ebp+v]
00401030 mov ecx, [eax]
00401032 mov [ebp+var_4], ecx
00401035 mov edx, [ebp+v]
00401038 mov eax, [edx+4]
0040103B mov [ebp+var_C], eax
0040103E mov [ebp+var_14], 9E3779B9h
00401045 mov ecx, [ebp+var_14]
00401048 shl ecx, 5
0040104B mov [ebp+var_10], ecx
9E3779B9h
0040104E
0040104E loc_40104E:
0040104E mov edx, [ebp+var_8]
00401051 mov eax, [ebp+var_8]
00401054 sub eax, 1
00401057 mov [ebp+var_8], eax
0040105A test edx, edx
0040105C jbe short loc_4010BC
0040105E mov ecx, [ebp+var_4]
00401061 shl ecx, 4
00401064 mov edx, [ebp+arg_4]
00401067 add ecx, [edx+8]
0040106A mov eax, [ebp+var_4]
0040106D add eax, [ebp+var_10]
00401070 xor ecx, eax
00401072 mov edx, [ebp+var_4]
00401075 shr edx, 5
00401078 mov eax, [ebp+arg_4]
0040107B add edx, [eax+0Ch]
0040107E xor ecx, edx
00401080 mov edx, [ebp+var_C]
00401083 sub edx, ecx
00401085 mov [ebp+var_C], edx
00401088 mov eax, [ebp+var_C]
0040108B shl eax, 4
0040108E mov ecx, [ebp+arg_4]
00401091 add eax, [ecx]
00401093 mov edx, [ebp+var_C]
00401096 add edx, [ebp+var_10]
00401099 xor eax, edx
0040109B mov ecx, [ebp+var_C]
0040109E shr ecx, 5
004010A1 mov edx, [ebp+arg_4]
004010A4 add ecx, [edx+4]
004010A7 xor eax, ecx
004010A9 mov ecx, [ebp+var_4]
004010AC sub ecx, eax
004010AE mov [ebp+var_4], ecx
004010B1 mov edx, [ebp+var_10]
004010B4 sub edx, [ebp+var_14]
004010B7 mov [ebp+var_10], edx
004010BA jmp short loc_40104E
004010BC
004010BC loc_4010BC:
004010BC mov eax, [ebp+v]
004010BF mov ecx, [ebp+var_4]
004010C2 mov [eax], ecx
004010C4 mov edx, [ebp+v]
004010C7 mov eax, [ebp+var_C]
004010CA mov [edx+4], eax
004010CD mov esp, ebp
004010CF pop ebp
004010D0 retn
004010D0 sub_401020 endp
004010D0

[Tiny Encryption Algorithm - Wikipedia, the free encyclopedia]
en.wikipedia.org/wiki/Tiny_Encryption_Algorithm - Traduire cette page
The magic constant, 2654435769 or **9E3779B9**16 is chosen to be 232/ϕ, where ϕ is the golden ratio. TEA has a few weaknesses. Most notably, it suffers from **...**
↳ Properties - Versions - Reference code - See also
Vous avez consulté cette page 15 fois. Dernière visite : 07/04/12

[PDF] [The RC5 Encryption Algorithm?]
www.engr.uconn.edu/~zshi/.../rc5.pdf - États-Unis - Traduire cette page
Format de fichier: PDF/Adobe Acrobat - Afficher
de RL Rivest - Cité 827 fois - Autres articles
Q32 = 10011110001101110111100110111001 = **9e3779b9**. P64 =
1011011111100001010100010110001010001010111011010010101001101011 **...**
Vous avez consulté cette page le 05/04/12.

[Internet Security: Cryptographic Principles, Algorithms and Protocols - Résultats Google Recherche de Livres]
books.google.fr/books?isbn=0470852852**...**
Man Young Rhee - 2003 - Computers - 405 pages
... b7e!5163 + **9e3779b9** = 5618cblc 5[2] = 5[1] + Q32 = 5618cblc + **9e3779b9** = f45044d5 5[3] = S[2] + Q32 = f45044d5 + **9e3779b9** = 9287be8e S[25] = S[24] + **...**

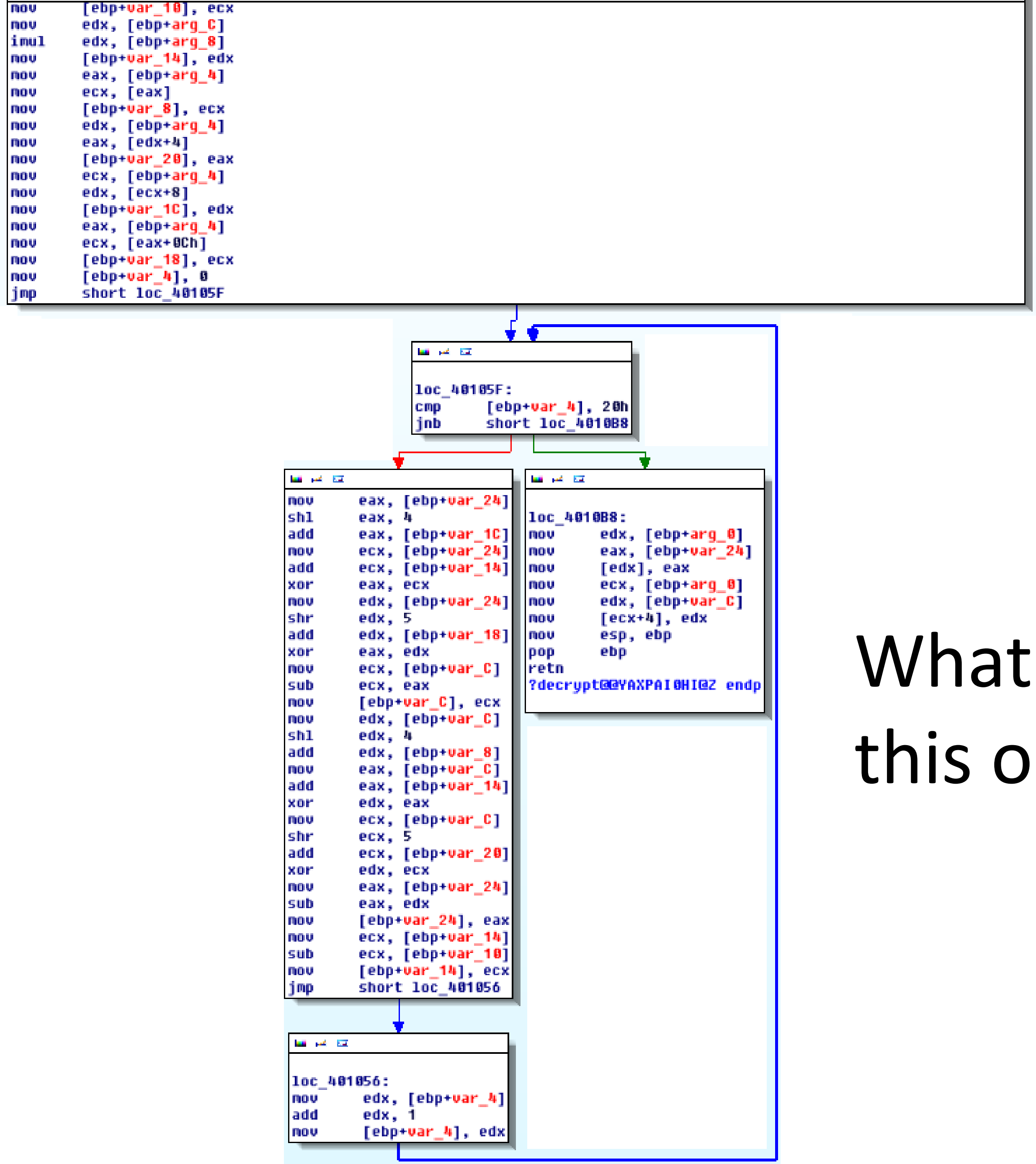
[Changeset 329 – CrypTool 2.0]
https://www.cryptool.org/trac/CrypTool2/changeset/329
28 May 2009 – The magic constant, 2654435769 (Decimal) or **9E3779B9** (Hex) is chosen to be (2^32 / phi) where phi is the golden ratio.</Run> **...**

[TEA Encryption Algorithm, Source code]
www.shokhirev.com/nikolai/.../uTeaSet_pas.html - Traduire cette page
... XTeaEncrypt/XTeaDecrypt Thanks to Pedro Gimeno Fortea <parigalo@formauri.es>
} interface const Delta: longword = $**9e3779b9**; type TLong2 = array[0.

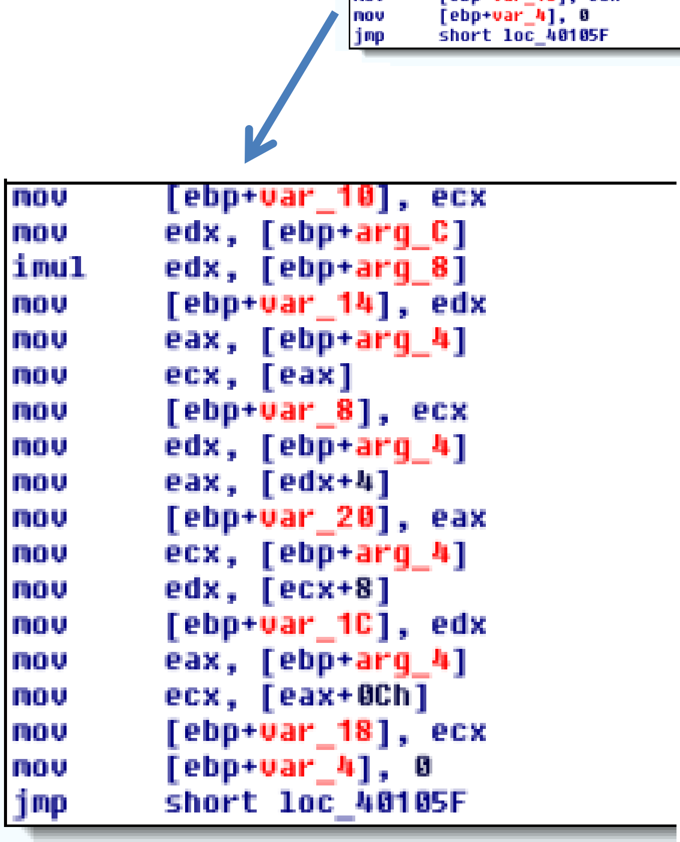
Tools	Answer
Crypto Searcher	“TEA”
Draca v0.5.7b	“TEA/RC5/RC6”
Findcrypt v2	Ø
Hash & Crypto Detector v1.4	“TEA/XTEA/TEAN”
PEiD KANAL v2.92	“TEA/N, RC5, RC6”
Kerckhoffs	Ø
Signsrch 0.1.7	“TEA”
SnD Crypto Scanner v0.5b	Ø

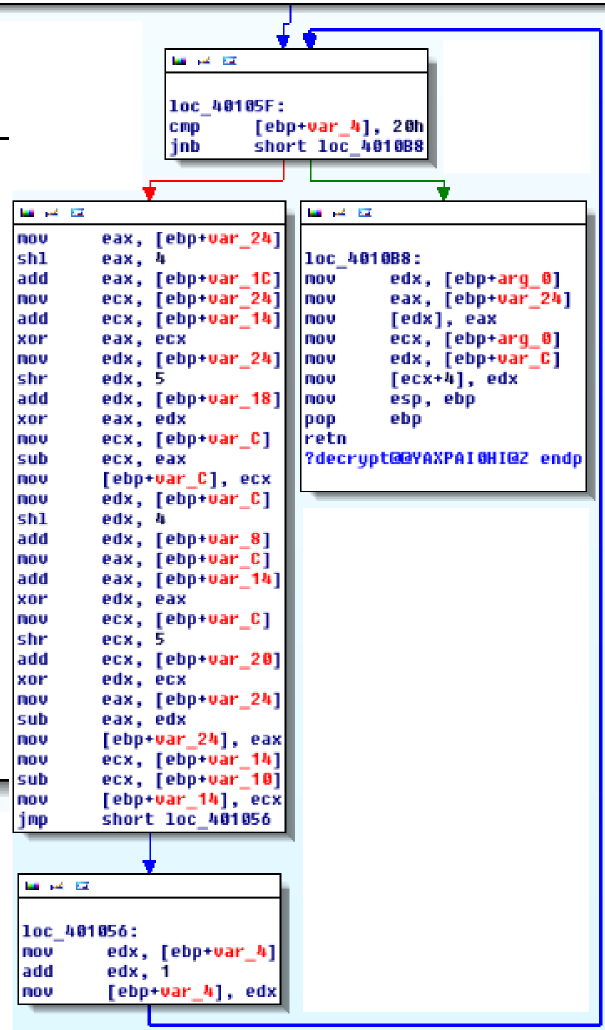
Tools	Answer
Crypto Searcher	“TEA”
Draca v0.5.7b	“TEA/RC5/RC6”
Findcrypt v2	Ø
Hash & Crypto Detector v1.4	“TEA/XTEA/TEAN”
PEiD KANAL v2.92	“TEA/N, RC5, RC6”
Kerckhoffs	Ø
Signsrch 0.1.7	“TEA”
SnD Crypto Scanner v0.5b	Ø

That’s indeed the Tiny Encryption Algorithm!



What about this one?





What about this one?

No particular constants

Tools	Answer
Crypto Searcher	Ø
Draca v0.5.7b	Ø
Findcrypt v2	Ø
Hash & Crypto Detector v1.4	Ø
PEiD KANAL v2.92	Ø
Kerckhoffs	Ø
Signsrch 0.1.7	Ø
SnD Crypto Scanner v0.5b	Ø

Tools	Answer
Crypto Searcher	Ø
Draca v0.5.7b	Ø
Findcrypt v2	Ø
Hash & Crypto Detector v1.4	Ø
PEiD KANAL v2.92	Ø
Kerckhoffs	Ø
Signsrch 0.1.7	Ø
SnD Crypto Scanner v0.5b	Ø

Sigh.. That was still TEA!

What Can We Do ?

- How to recognize different TEA implementations in a more reliable way ?

- Is there something such implementations *have to* share ?

(If so, we could use it in obfuscated programs!)

Input-Output Relationship

- For a key **K** and an encrypted text **C**, *any* TEA implementation produces the *same* decrypted text **C’**.

Input-Output Relationship

- For a key **K** and an encrypted text **C**, *any* TEA implementation produces the *same* decrypted text **C'**.

Could we identify TEA implementations by using their deterministic I/O relationship ?

(or any other cipher)

PROPOSED SOLUTION

How To Use Input-Output Relationship ?

- Let’s say **P** is a program implementing an unknown cryptographic algorithm.

How To Use Input-Output Relationship ?

- Let’s say **P** is a program implementing an unknown cryptographic algorithm.

- First idea: execute **P** on *all* possible input states and check if the outputs are the same than a known cryptographic algorithm.

(not realistic!)

How To Use Input-Output Relationship ?

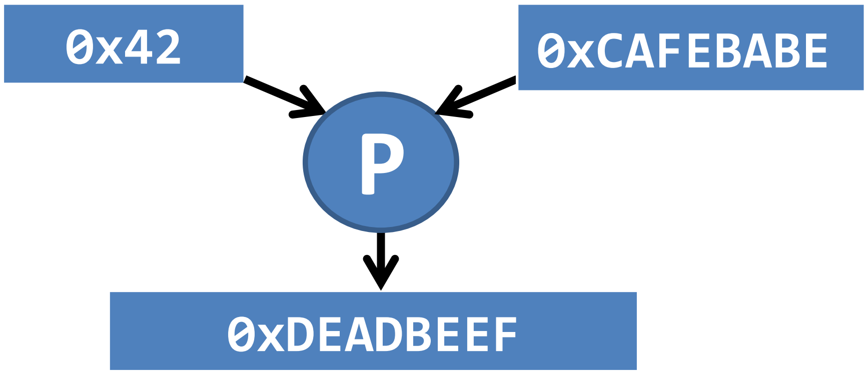
- Let’s say **P** is a program implementing an unknown cryptographic algorithm.

- First idea: execute **P** on *all* possible input states and check if the outputs are the same than a known cryptographic algorithm.

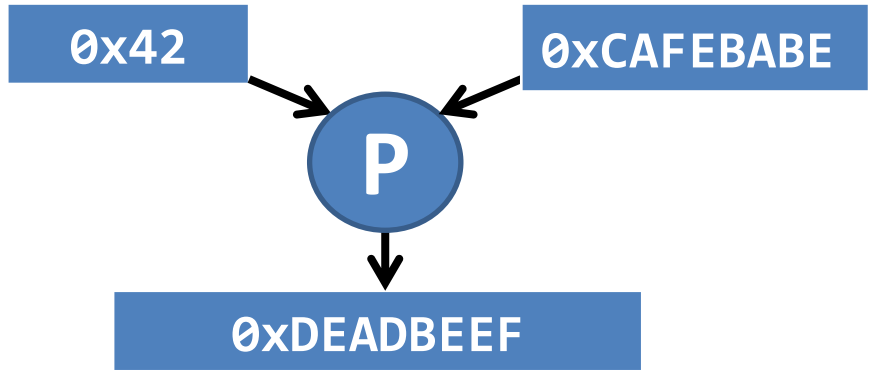
  (not realistic!)

- But we can observe one particular **P** execution and collect its input-output parameter values...

For example:



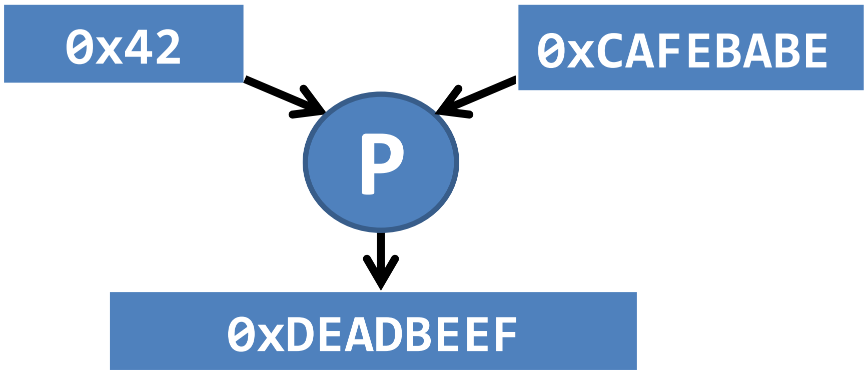
For example:



- Now imagine that when we execute a <u>reference implementation</u> of **TEA** with the key 0x42 and the input text 0xCAFEBABE, it produces 0xDEADBEEF.

What does it mean for **P**?

For example:



- Now imagine that when we execute a <u>reference implementation</u> of **TEA** with the key 0x42 and the input text 0xCAFEBABE, it produces 0xDEADBEEF.

What does it mean for **P**?

- It proves that **P** implements **TEA** <u>on these particular input values</u>.

Final Goal

- We are going to prove that a particular program **P** behaves like a known cryptographic algorithm <u>during a particular execution</u>.

- It means that we are **not** going to prove a general semantic equivalence between **P** and a cryptographic algorithm.

Workflow

Given a program **P**:

- **Step 1:** Collect **P** execution trace.
- **Step 2:** Extract possible cryptographic algorithms *with their parameters* from **P** execution trace (here is the magic).
- **Step 3:** Identify these algorithms by comparing their I/O relationship with those of known algorithms.

STEP 1: COLLECT EXECUTION TRACE

Execution Trace

- Pin: Dynamic Binary Instrumentation framework.

Address	Instruction	Read Registers	Written Registers	Read Memory	Written Memory
4012b3	push ebp	ebp *0012de28* esp *0012bd98*	esp *0012bd94*		12bd94 *0012de28*
4012b4	mov ebp, esp	esp *0012bd94*	ebp *0012bd94*		
4012b6	push ebx	ebx *02f00010* esp *0012bd84*	esp *0012bd80*		12bd80 *2f00010*
		...			

STEP 2: CRYPTOGRAPHIC ALGORITHM EXTRACTION

How To Find Crypto Code ? (1)

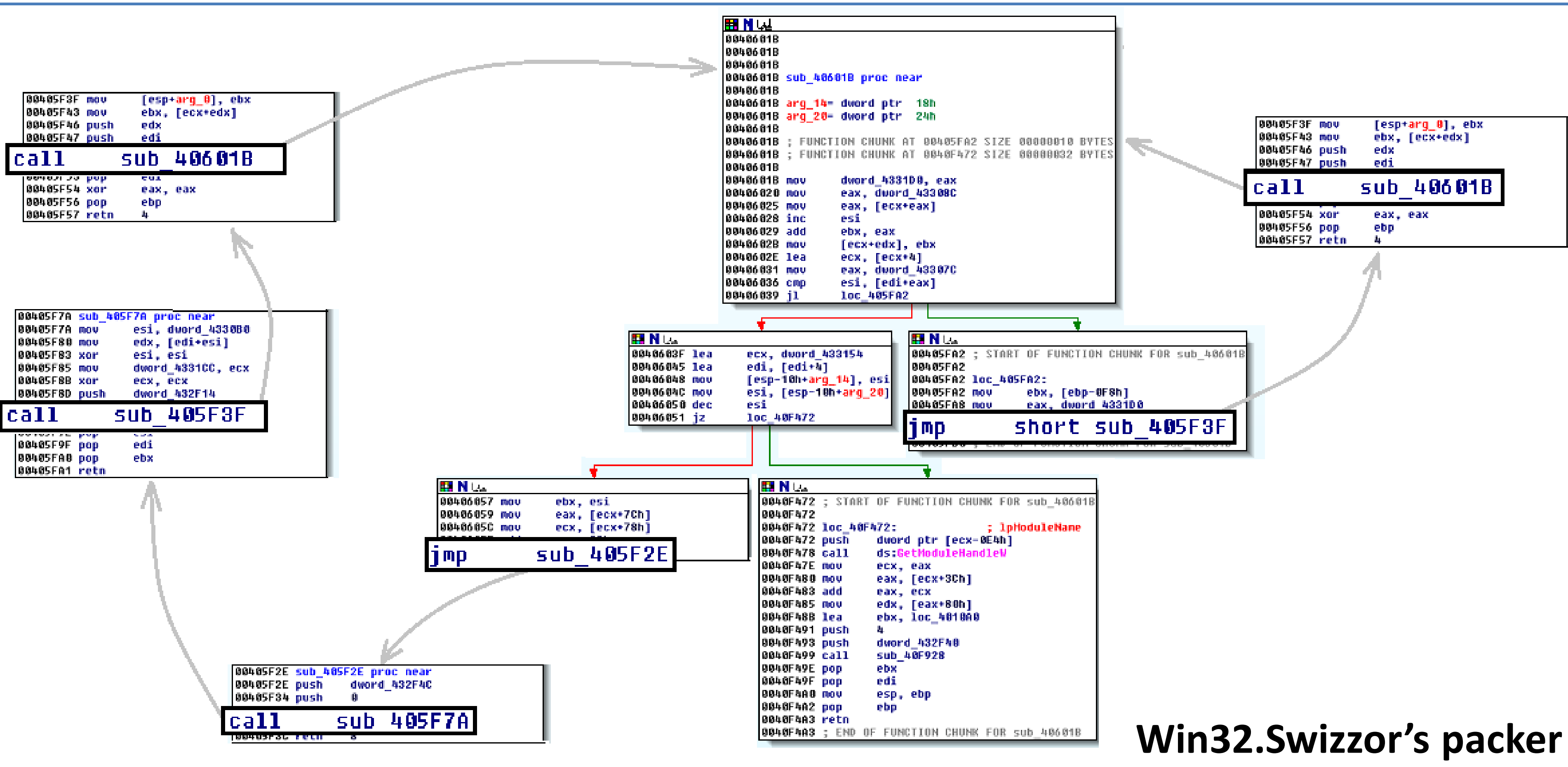
- Cryptographic code constitutes only a part of programs, we need a way to find it.

How To Find Crypto Code ? (1)

- Cryptographic code constitutes only a part of programs, we need a way to find it.

- As we want to play with *obfuscated* programs, IDA functions will not be enough...

In obfuscated programs, such things can happen:



Win32.Swizzor's packer

How To Find Crypto Code ? (2)

- Cryptographic algorithms usually apply *a same treatment* on their input-output parameters.

How To Find Crypto Code ? (2)

- Cryptographic algorithms usually apply *a same treatment* on their input-output parameters.

- It makes **loops** a cryptographic code feature.

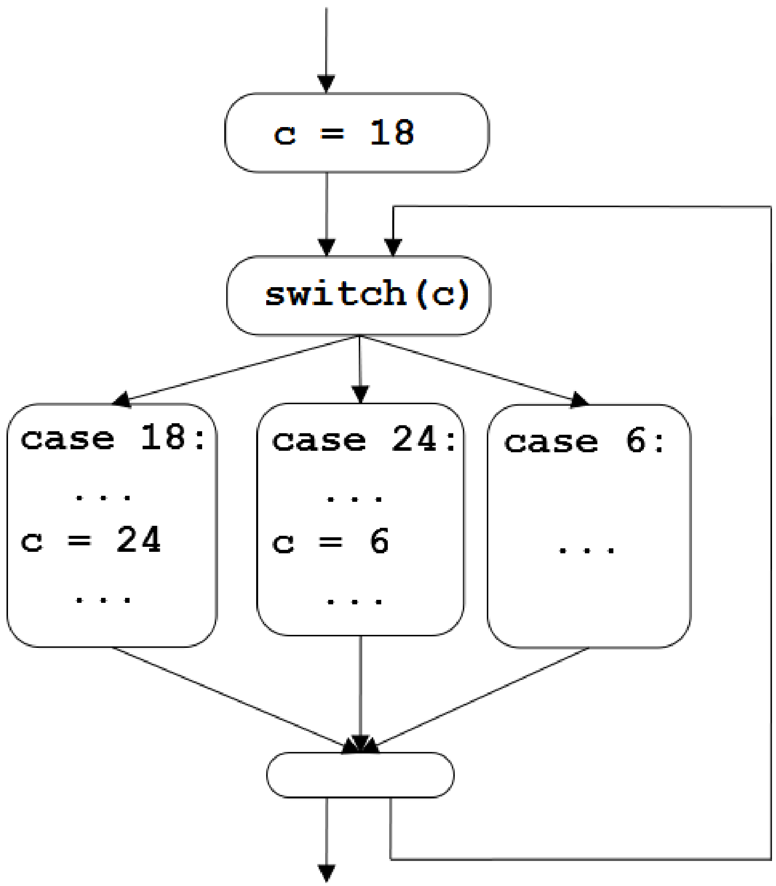
How To Find Crypto Code ? (2)

- Cryptographic algorithms usually apply *a same treatment* on their input-output parameters.

- It makes **loops** a cryptographic code feature.

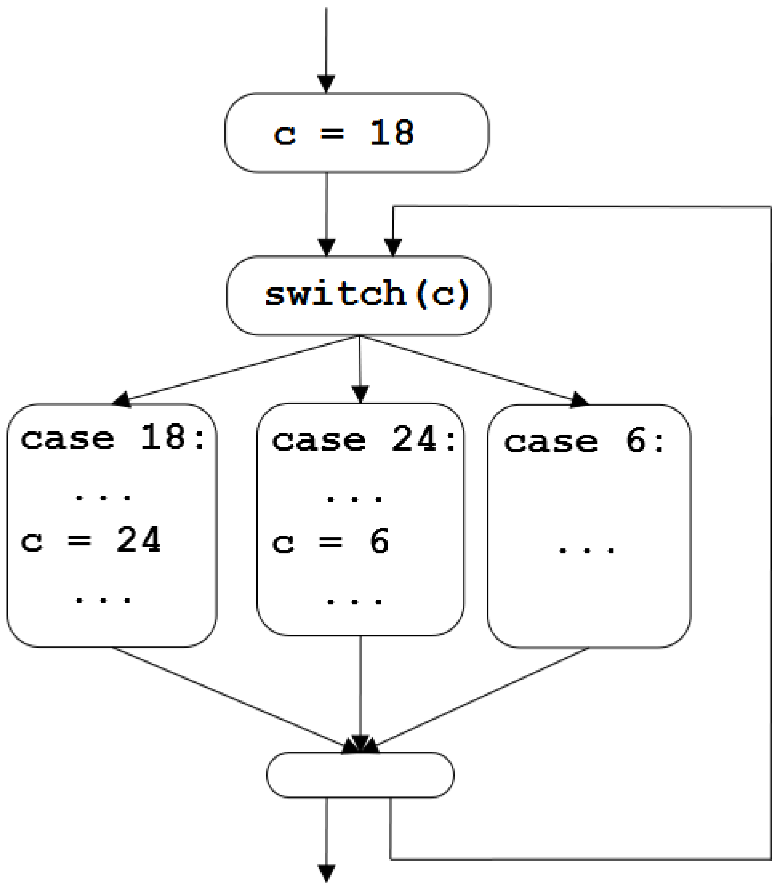
- But there are loops everywhere, not only in crypto... What kind of loops are we looking for ?

Loops ?

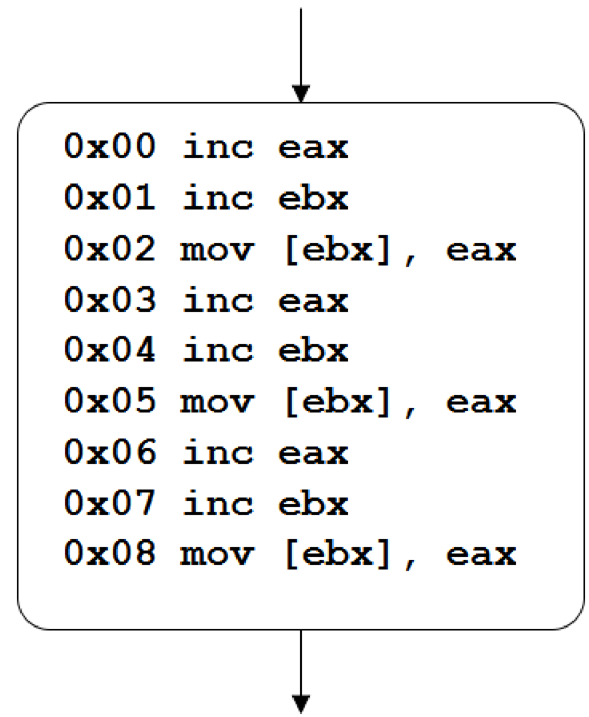


Win32.Mebroot

Loops ?

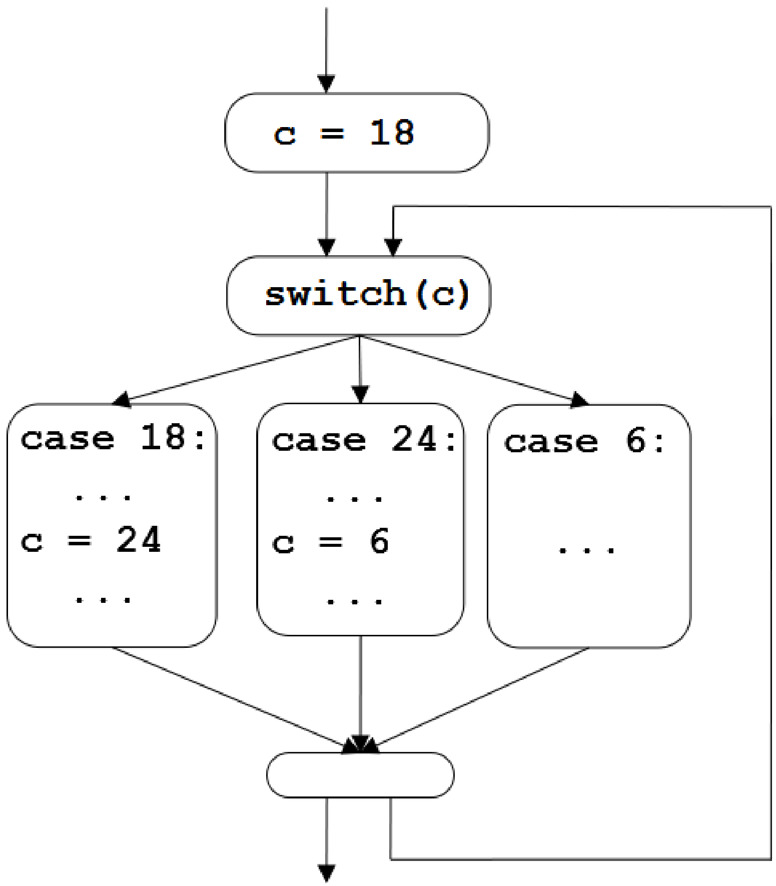


Win32.Mebroot

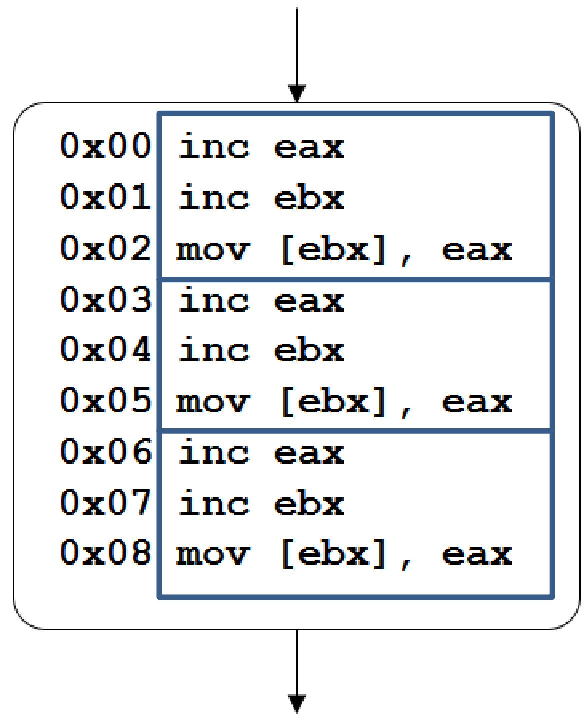


Unrolling optimization

Loops ?



Win32.Mebroot



Unrolling optimization

Looooops

- We look for **the same operations applied repeatedly** on a set of data.

Looooops

- We look for **the same operations applied repeatedly** on a set of data.

“A loop is the repetition of a same sequence of machine instructions at least two times.”

(This sequence of instructions is the loop body.)

Example

Execution Trace

…	…
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
…	…

Example

Execution Trace

…	…
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
…	…

Iteration 1

Iteration 2

Example

Execution Trace

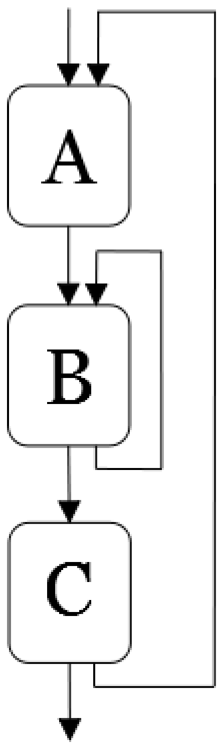
...	...
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
401325	add ebx, edi
401327	sub edx, ebx
401329	dec dword ptr [ebp+0xc]
40132c	jnz 0x401325
...	...

Iteration 1

Iteration 2

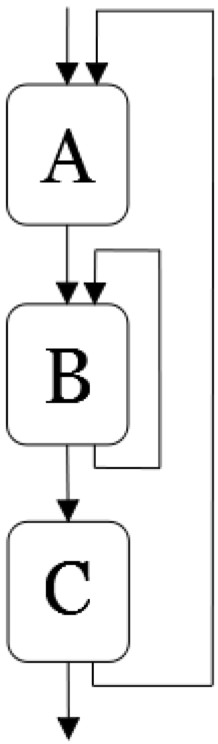
Loop

What About Nested Loops ?



Simplified CFG

What About Nested Loops ?

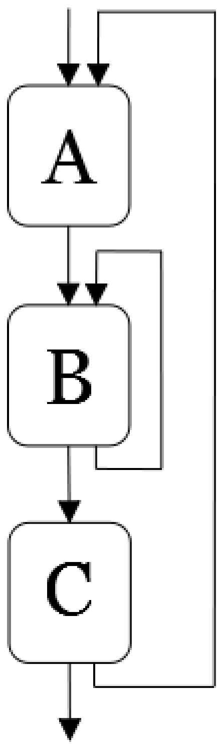


Simplified CFG

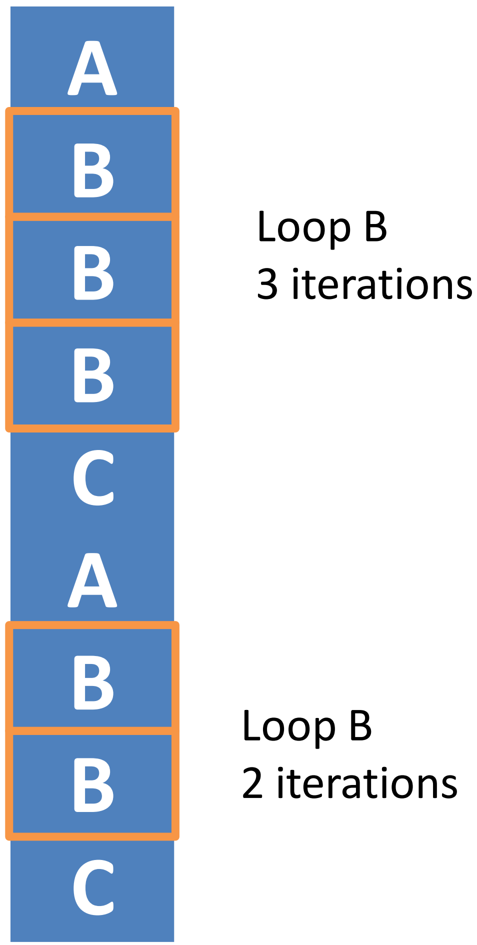
A
B
B
B
C
A
B
B
C

Execution trace

What About Nested Loops ?

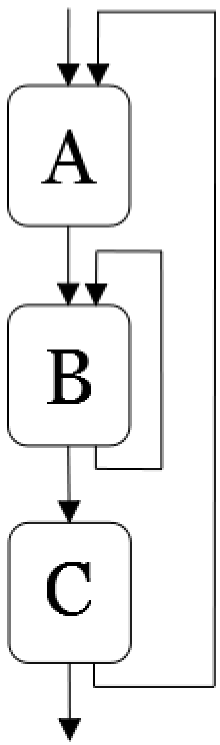


Simplified CFG



Execution trace

What About Nested Loops ?

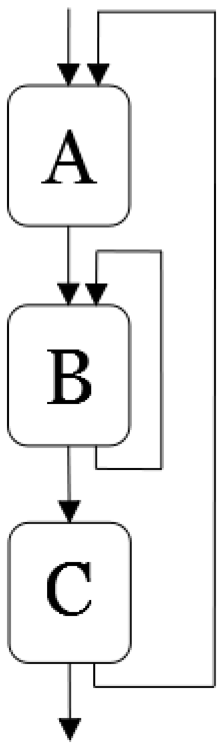


Simplified CFG

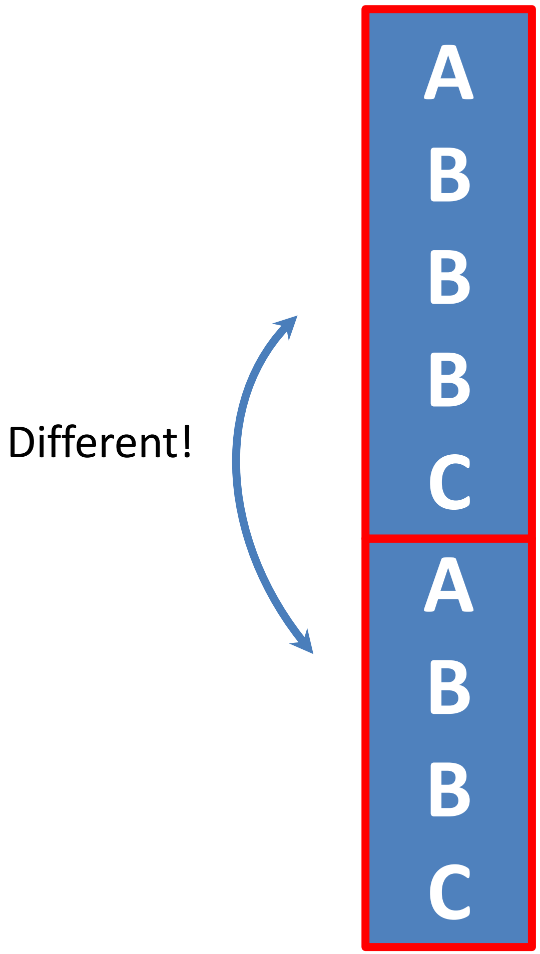
A
B
B
B
C
A
B
B
C

Execution trace

What About Nested Loops ?

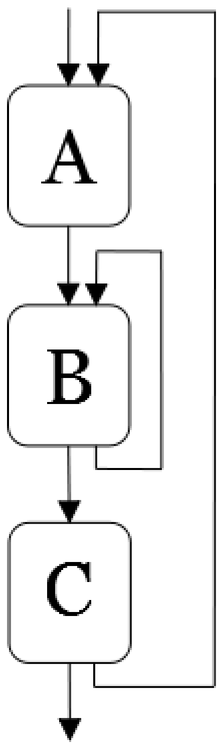


Simplified CFG

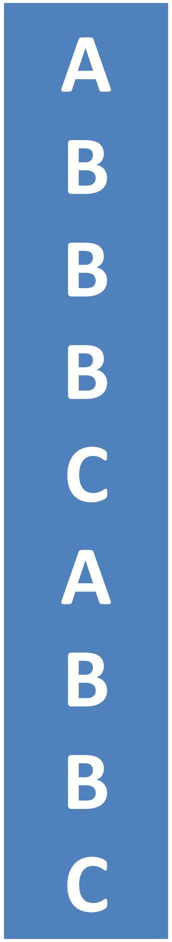


Execution trace

What About Nested Loops ?

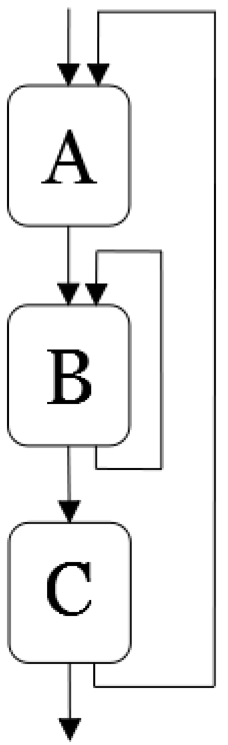


Simplified CFG

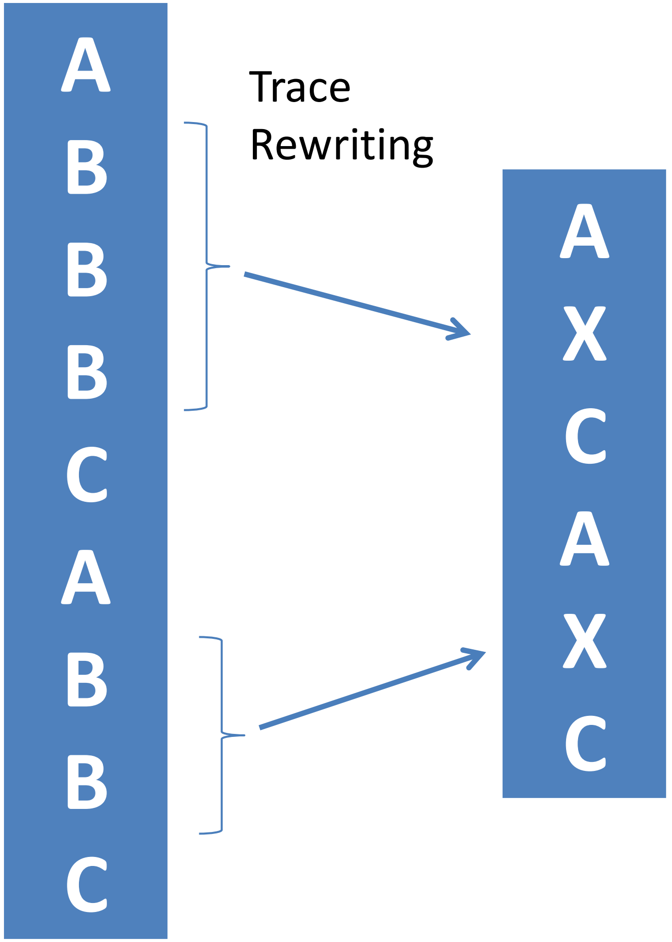


Execution trace

What About Nested Loops ?

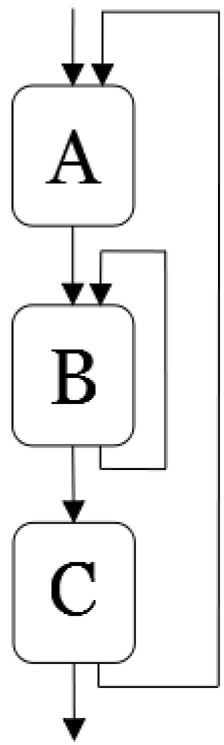


Simplified CFG

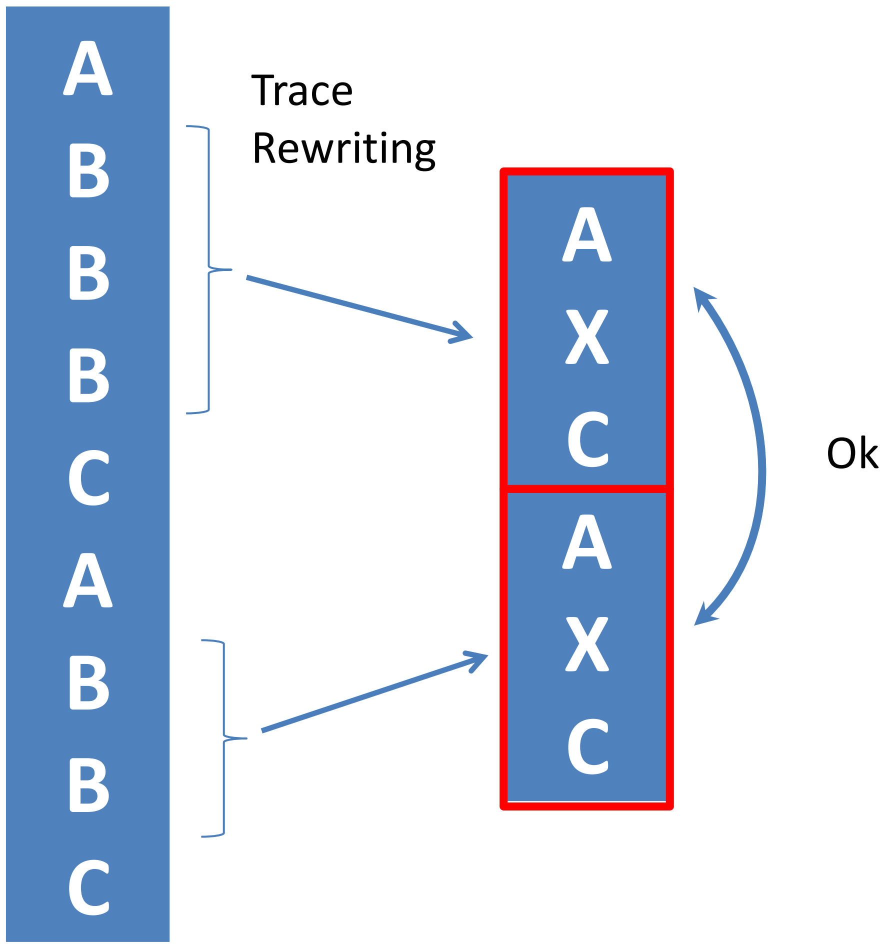


Execution trace

What About Nested Loops ?



Simplified CFG



Execution trace

Loop Detection Algorithm

1. Detects two repetitions of a loop body in the execution trace.

   (non trivial, language w.w is non-context-free)

2. Replaces in the trace the detected loop by a symbol representing their body.
3. Goes back to step 1 if new loops have been detected.

What’s Next ?

- We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

What's Next ?

- We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

- For the moment, we assume that each possible cryptographic algorithm corresponds to *one single* loop.

What’s Next ?

- We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

- For the moment, we assume that each possible cryptographic algorithm corresponds to *one single* loop.

- How can we define *parameters* from the *bytes* read and written in the execution trace ?

Loop Parameters (1)

- Distinction between input and output bytes in the execution trace:
  - **Input** bytes have been **read without having been previously written**.
  - **Output** bytes have been **written**.

Loop Parameters (2)

- We want to group together bytes belonging to the same cryptographic parameter (key, input text...).

Loop Parameters (2)

- We want to group together bytes belonging to the same cryptographic parameter (key, input text...).

What criteria can we use ?

Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*

Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by the same instruction in the loop body**.

Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by the same instruction in the loop body**.

```
add ebx, edi
mov eax, [ebx]
...
add ebx, edi
mov eax, [ebx]
...
```

Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by the same instruction in the loop body**.

Iteration 1

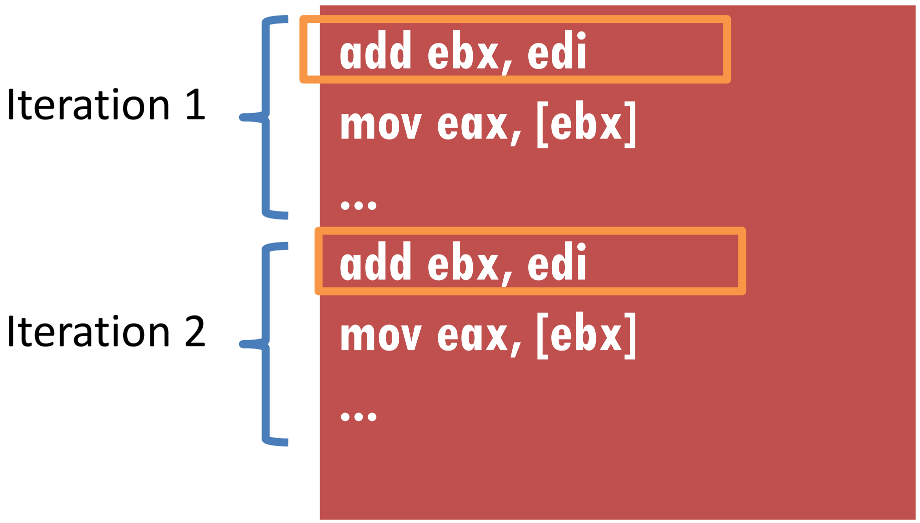
```
add ebx, edi
mov eax, [ebx]
...
```

Iteration 2

```
add ebx, edi
mov eax, [ebx]
...
```

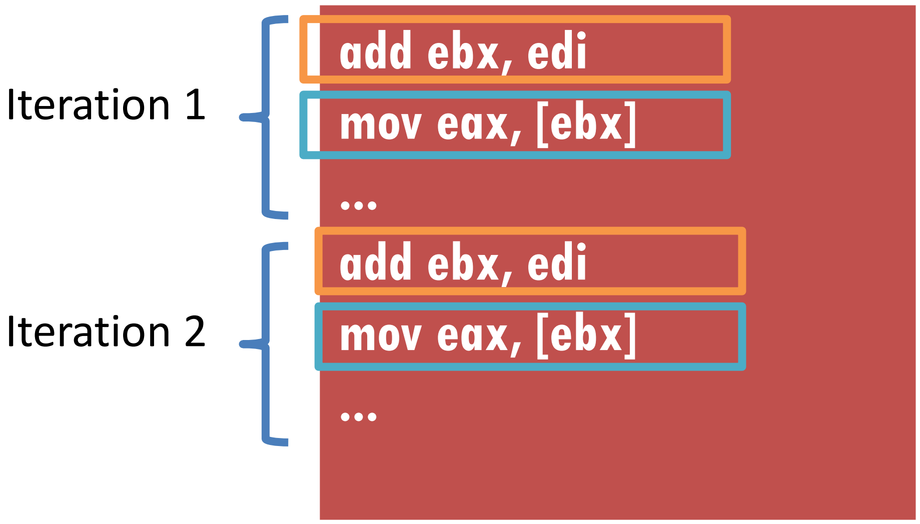
Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by the same instruction in the loop body**.



Loop Parameters (3)

- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by the same instruction in the loop body**.



Loop Parameters (4)

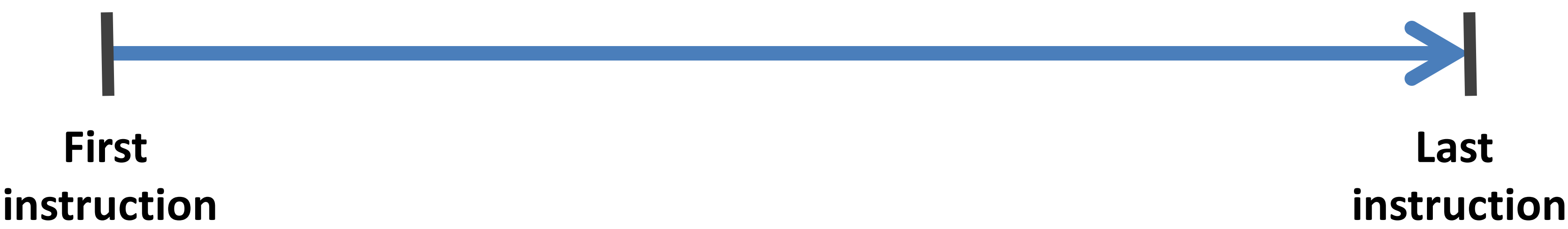
- A parameter is then defined by:
  - An identifier: *“(memory address|register name):size”*
  - A value

Let's Recap With a Use-Case

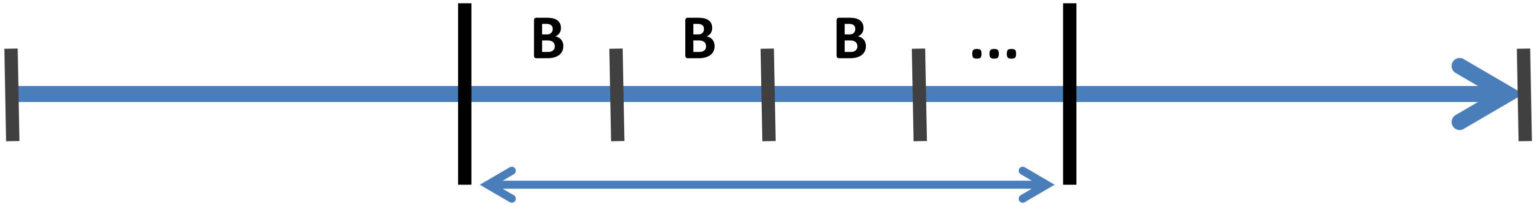
- Tiny Encryption Algorithm:
  - Block cipher
  - 16-byte key
  - 8-byte input text
  - Magic constant ***delta*** (0x9E3779B9)

- We built a toy program calling the TEA decryption function on:
  - Key : **`0xDEADBEE1...DEADBEE4`**
  - Encrypted text: **`0x0123456789ABCDEF`**

Step 1 : Gather Execution Trace

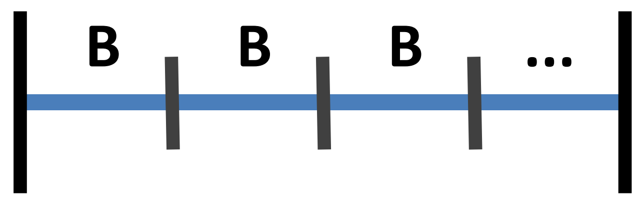


Step 2 : Recognize Loops

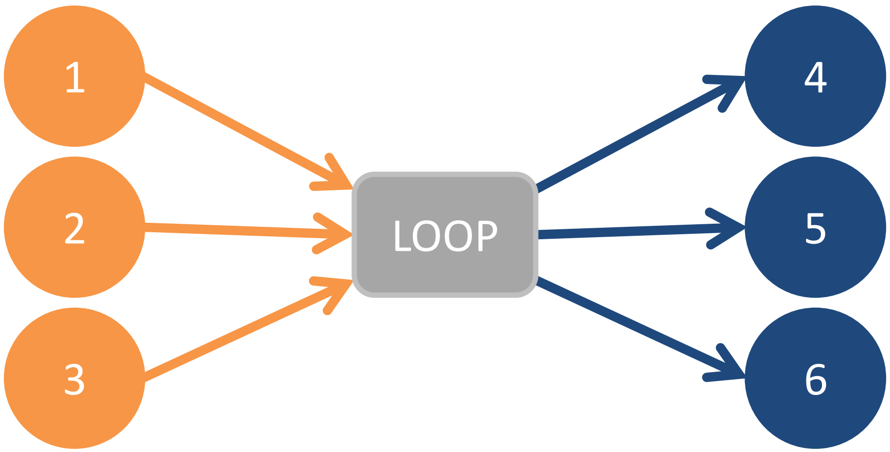


Machine instruction sequence B is repeated

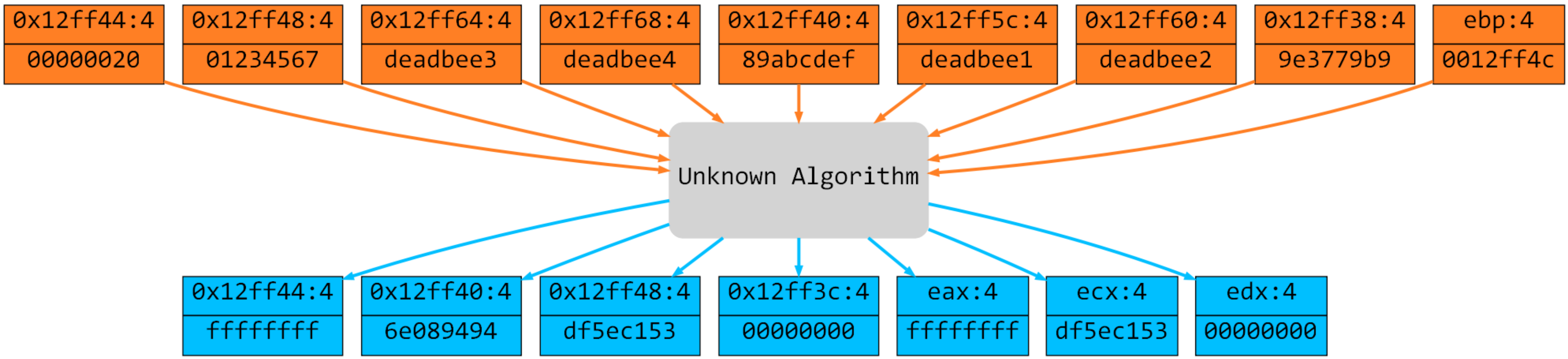
Step 2 : Recognize Loops



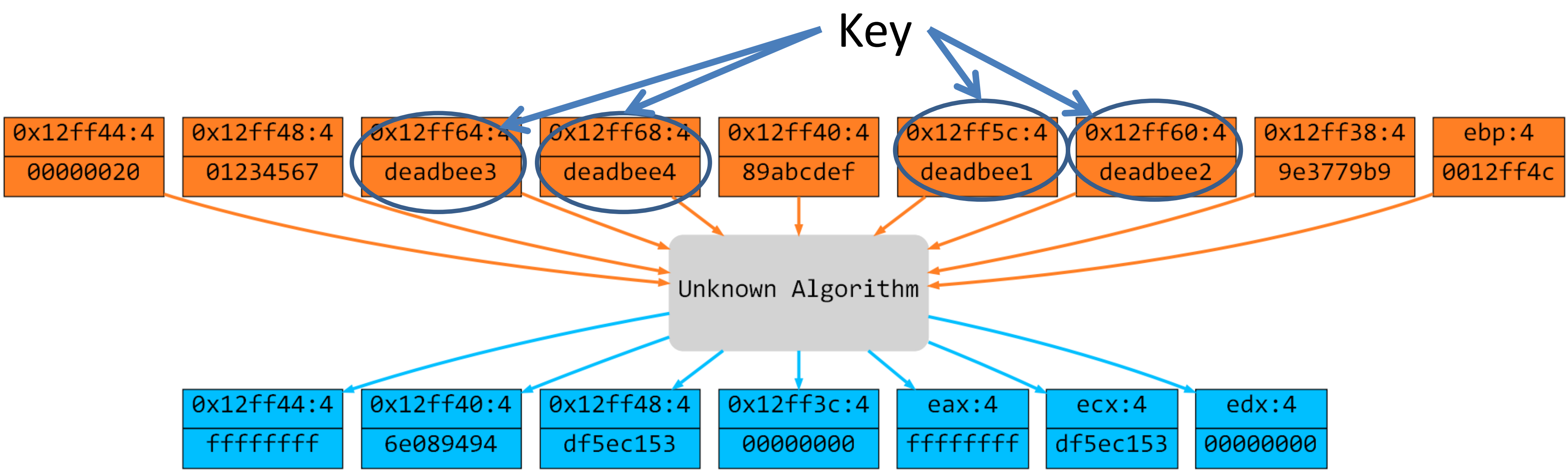
Step 3 : Define Loop Parameters



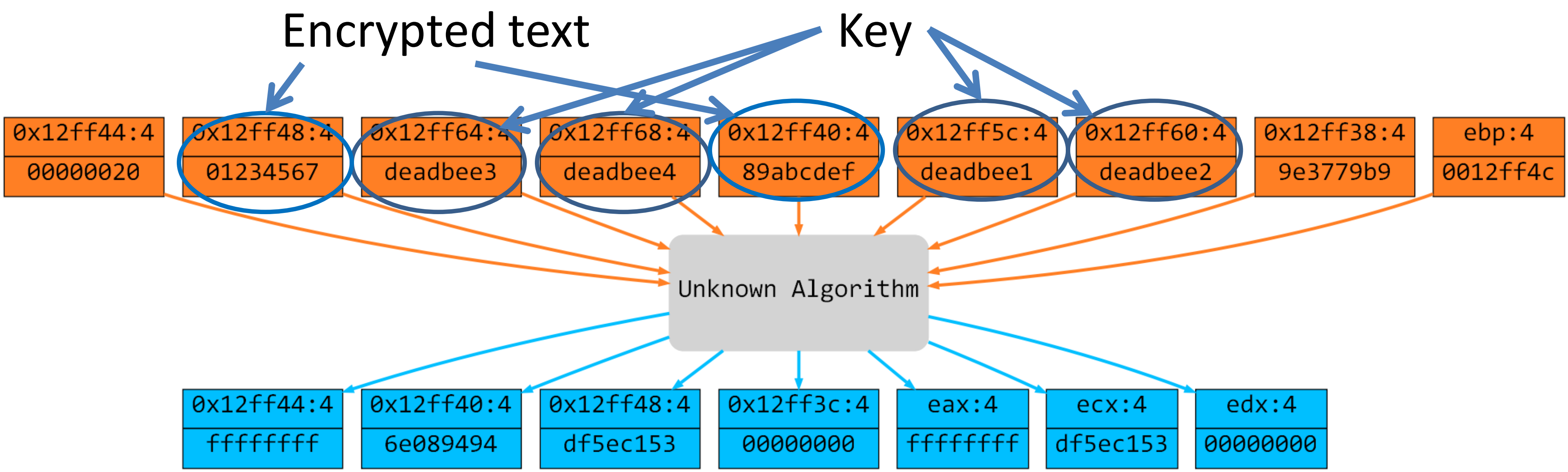
Each loop is then a possible cryptographic algorithm!


0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153
edx:4
00000000

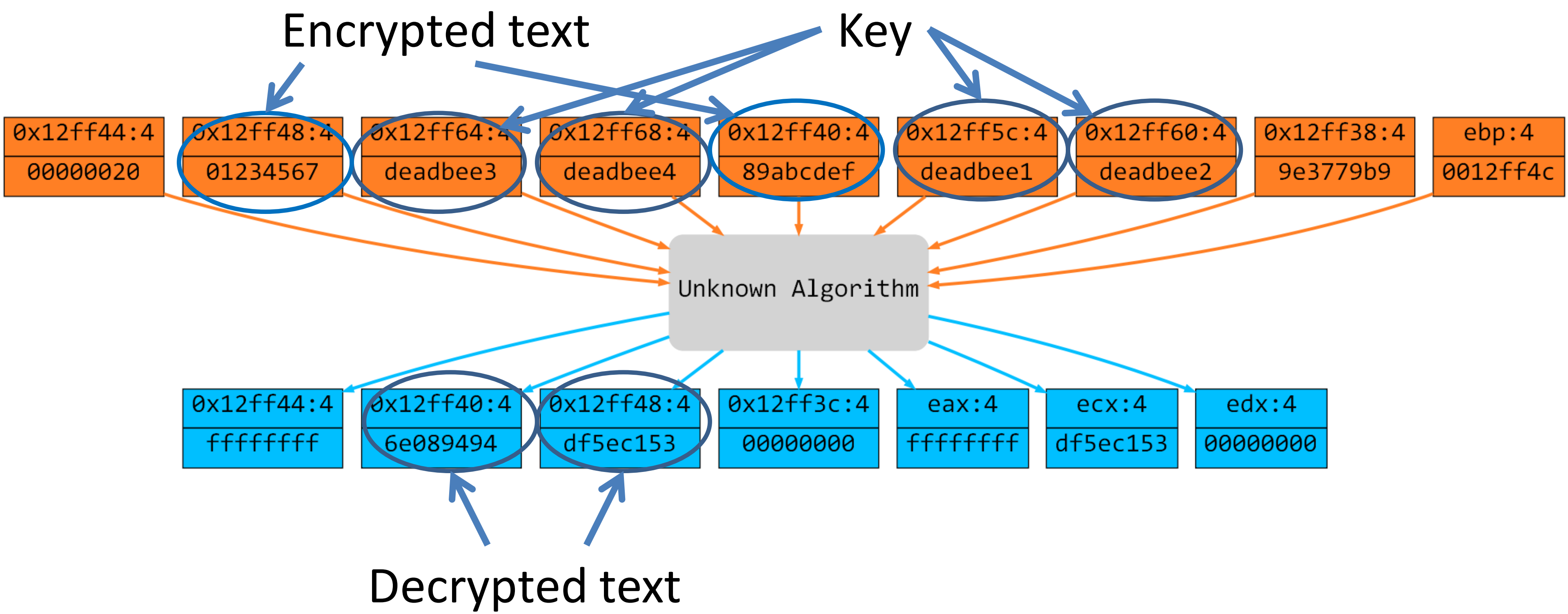
Final Model


Key
0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153
edx:4
00000000

Final Model


Encrypted text
Key
0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153
edx:4
00000000

Final Model

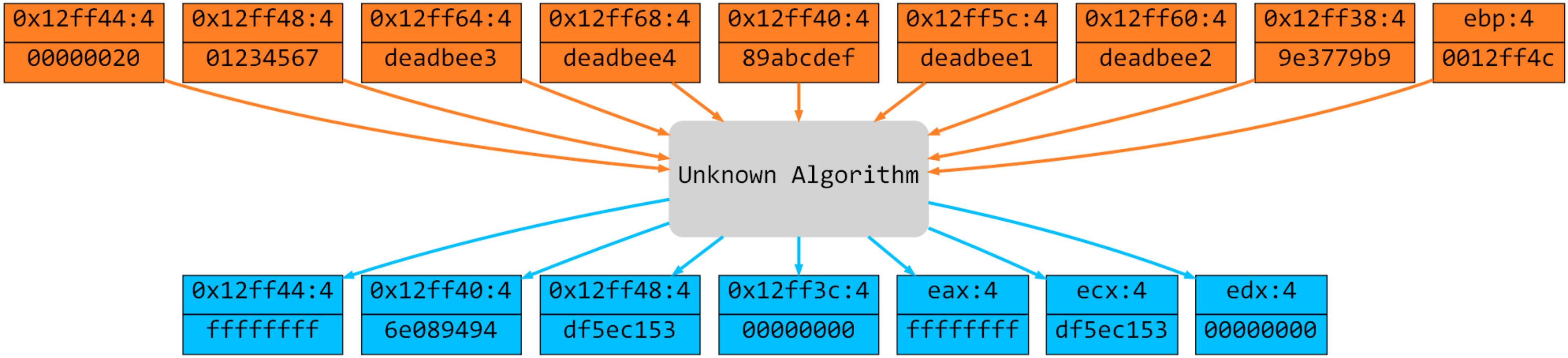


Encrypted text
Key
0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153
edx:4
00000000
Decrypted text

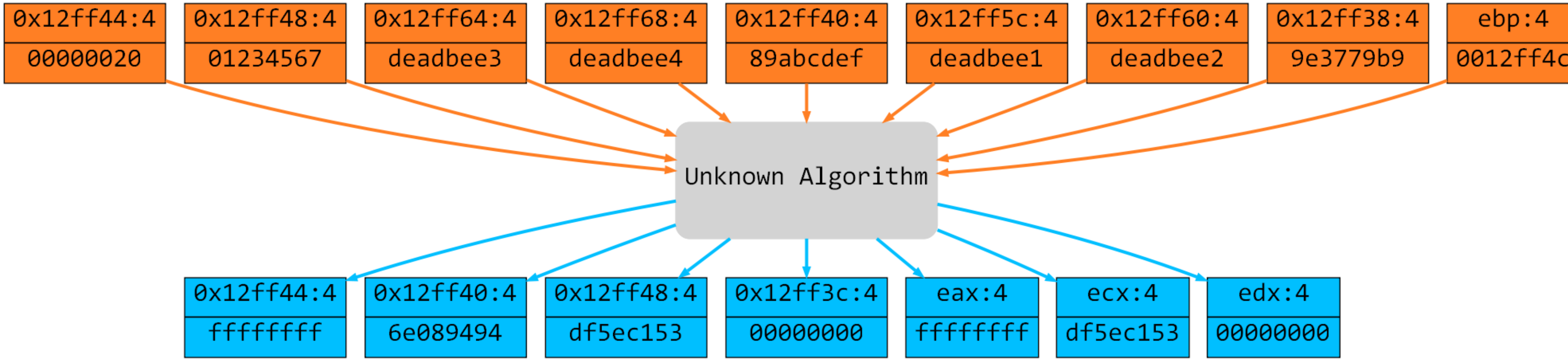
Final Model

STEP 3: CRYPTO ALGORITHM IDENTIFICATION

Input 1: unknown algorithm **A** with its parameter values



Input 1: unknown algorithm **A** with its parameter values



Input 2: reference implementations for common crypto algo

def tea (input_text, key): ...	def xtea (input_text, key): ...	def rc4 (input_text, key): ...

Question

- Is there a way to combine **A** input values such that *tea(), xtea()* or *rc4()* would produce a combination of **A** output values ?

Question

- Is there a way to combine **A** input values such that *tea()*, *xtea()* or *rc4()* would produce a combination of **A** output values ?

- Some difficulties:
  - **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.

Question

- Is there a way to combine **A** input values such that *tea()*, *xtea()* or *rc4()* would produce a combination of **A** output values ?

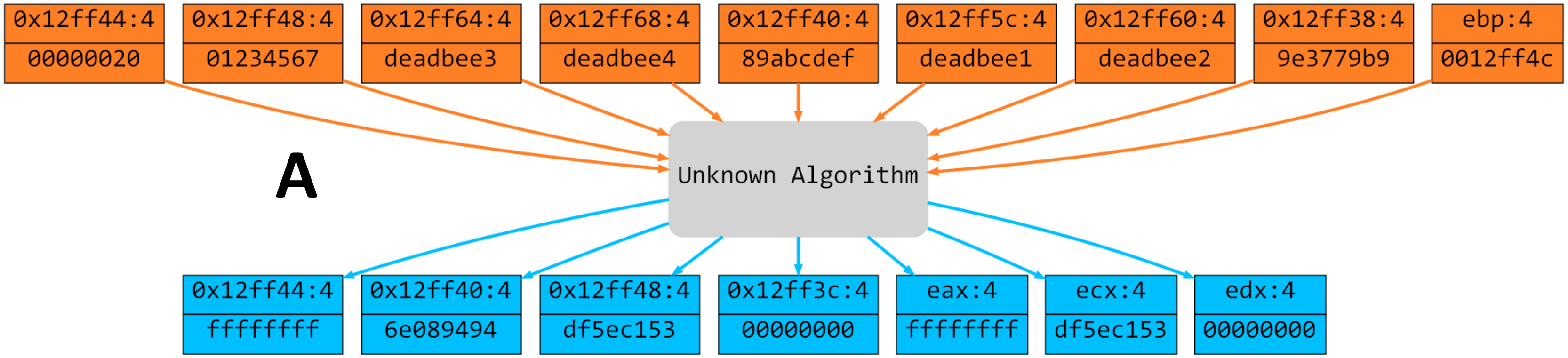
- Some difficulties:
  - **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  - **Parameter order:** no particular order for **A** parameters.

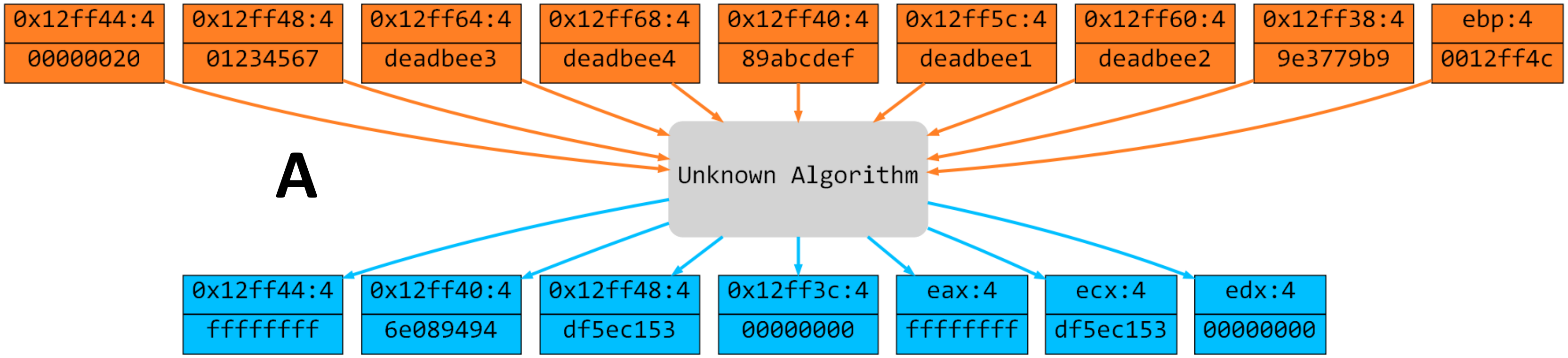
Question

- Is there a way to combine **A** input values such that *tea()*, *xtea()* or *rc4()* would produce a combination of **A** output values ?

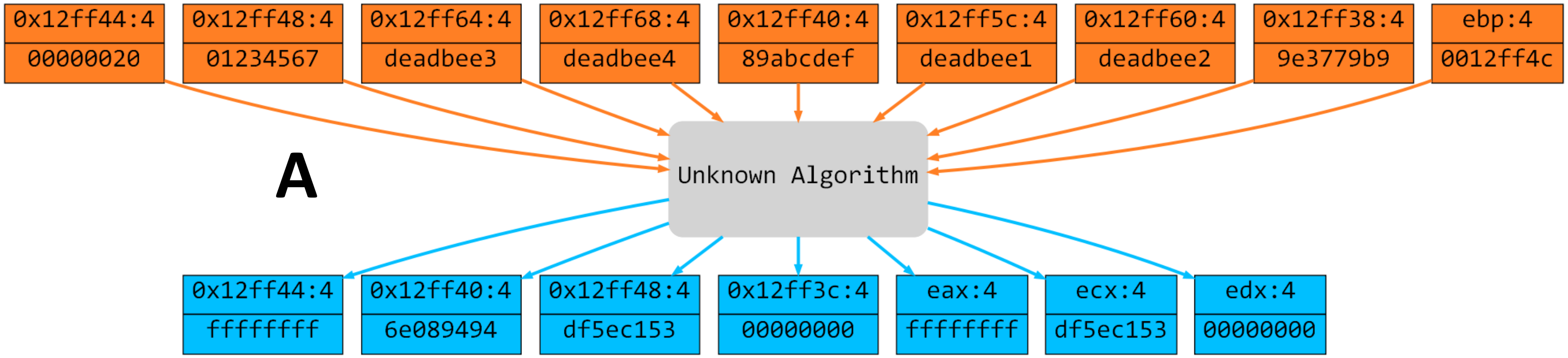
- Some difficulties:
  - **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  - **Parameter order:** no particular order for **A** parameters.
  - **Parameter number:** we collect more than the cryptographic parameters.

Brute-Force!

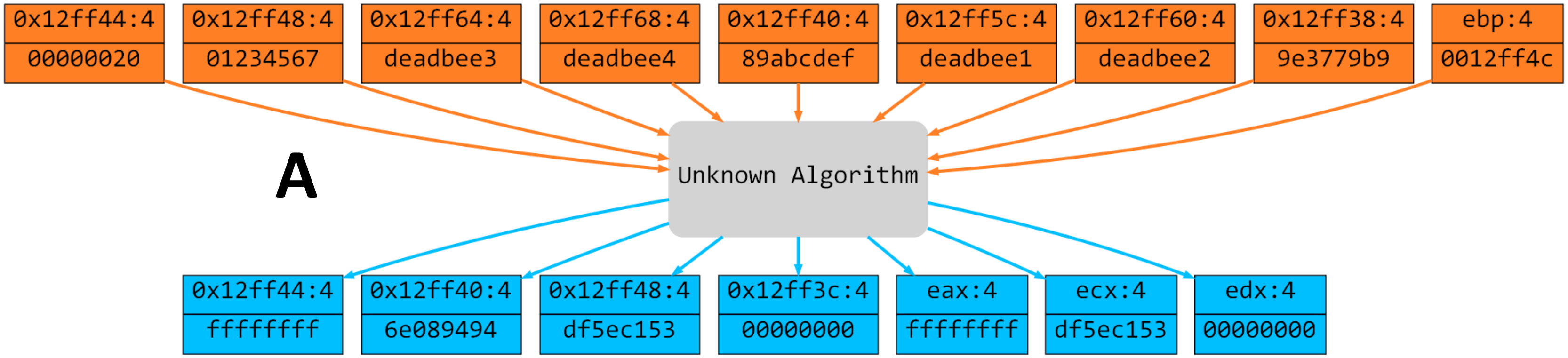

0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
A
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153
edx:4
00000000



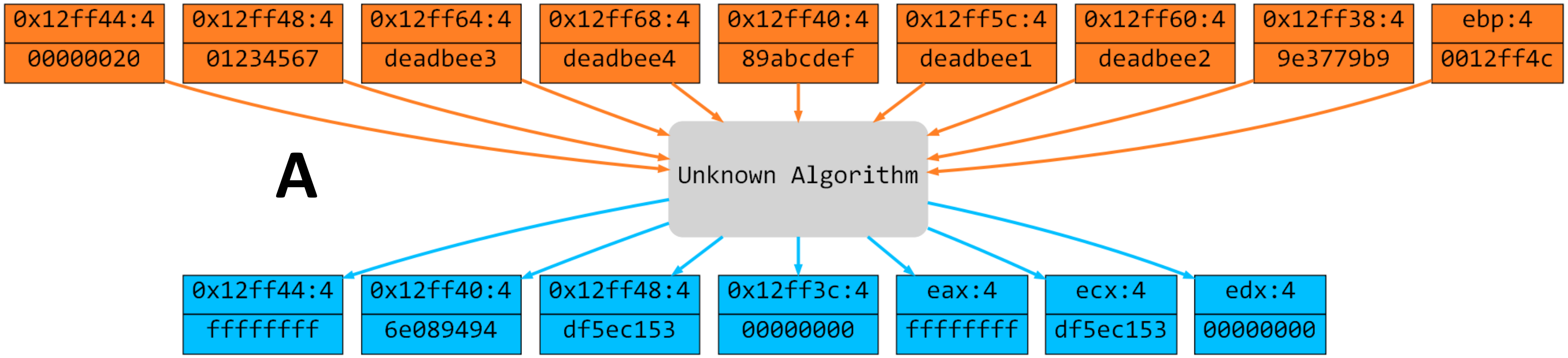
1. Generate all possible values with **A** input parameters:
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...



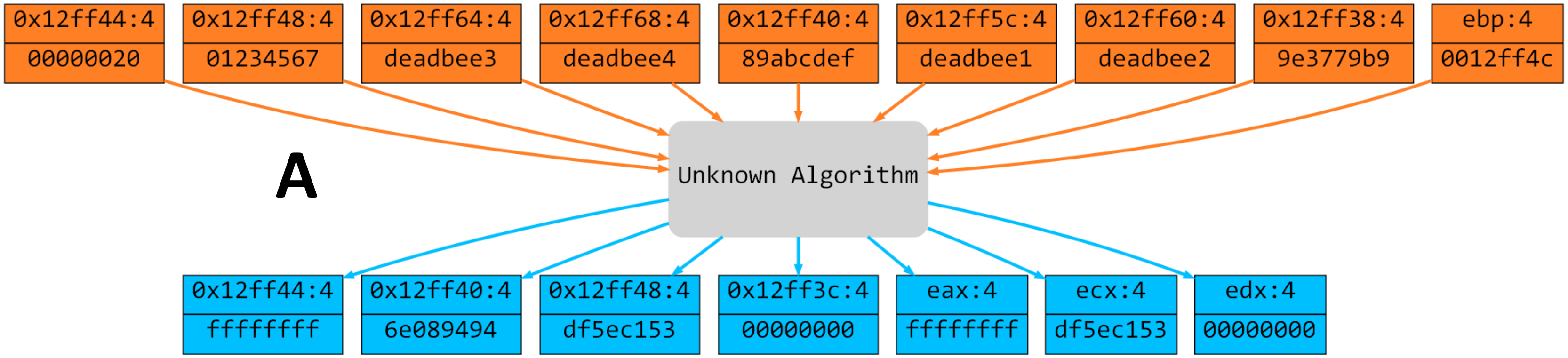
1. <u>Generate all possible values with **A** input parameters:</u>
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...
2. <u>Same thing with **A** output parameters.</u>



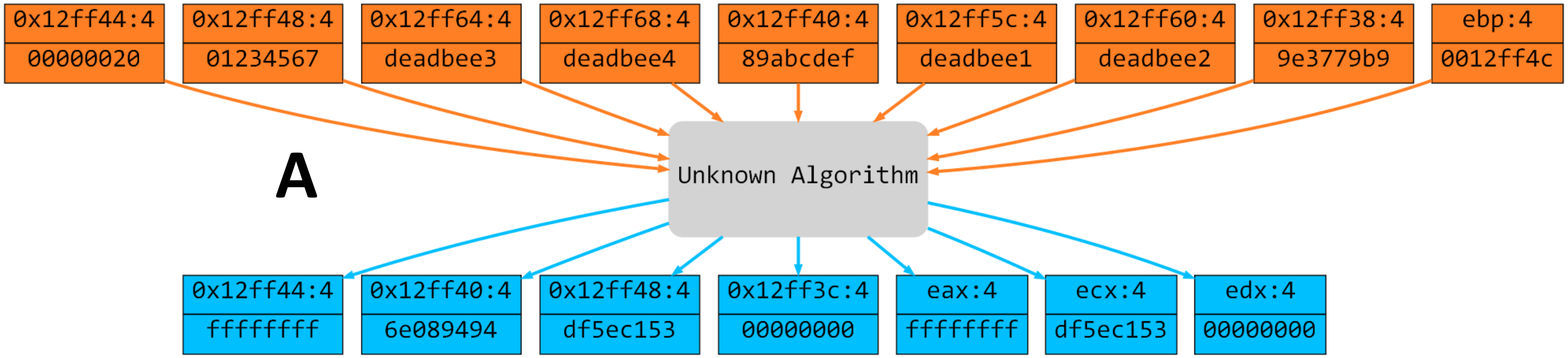
1. <u>Generate all possible values with **A** input parameters:</u>
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...
2. <u>Same thing with **A** output parameters.</u>
3. <u>For TEA reference implementation:</u>
    1. Possible input texts (8 bytes): *0000002001234567*,...



1. Generate all possible values with **A** input parameters:
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...
2. Same thing with **A** output parameters.
3. For TEA reference implementation:
    1. Possible input texts (8 bytes): *0000002001234567,...*
    2. Possible keys (16 bytes): ...



1. Generate all possible values with **A** input parameters:
   1. Length 4: *00000020*, *01234567*, *deadbee3*...
   2. Length 8: *0000002001234567*, *00000020deadbee3*,..
   3. ...
2. Same thing with **A** output parameters.
3. For TEA reference implementation:
   1. Possible input texts (8 bytes): *0000002001234567*,...
   2. Possible keys (16 bytes): ...
   3. Execute our TEA reference implementation on each possible pair (input text, key)



1. <u>Generate all possible values with **A** input parameters:</u>
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...
2. <u>Same thing with **A** output parameters.</u>
3. <u>For TEA reference implementation:</u>
    1. Possible input texts (8 bytes): *0000002001234567,...*
    2. Possible keys (16 bytes): ...
    3. Execute our TEA reference implementation on each possible pair (input text, key)
    4. If the output has been produced during step 2: success!

```
STARTED AT
2012-04-08 08:59:58.284000
** Crypto Algorithm Identification starting !**
9 input parameters
7 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption...
** Found TEA decryption **

 ===> Key (16 bytes) : deadbee1deadbee2deadbee3deadbee4

 ===> Crypted text (8 bytes) : 0123456789abcdef

 ===> Decrypted text (8 bytes) :    df5ec1536e089494
ENDED AT
2012-04-08 09:01:37.832000
```

~ 2 minutes

Malware And TEA

EXAMPLES!

Storm Worm

- Several internet references about the use of TEA in the Storm Worm packer (aka Tibs).
- Let’s take a look to the code...

```
push    edi
mov     ebx, [edi]
mov     ecx, [edi+4]
mov     edx, 9E3779B9h
mov     eax, edx
shl     eax, 5
mov     edi, 20h ; ' '
```

```
loc_1FA1:
mov     ebp, ebx
shl     ebp, 4
sub     ecx, ebp
mov     ebp, [esi+8]
xor     ebp, ebx
sub     ecx, ebp
mov     ebp, ebx
shr     ebp, 5
xor     ebp, eax
sub     ecx, ebp
sub     ecx, [esi+0Ch]
mov     ebp, ecx
shl     ebp, 4
sub     ebx, ebp
mov     ebp, [esi]
xor     ebp, ecx
sub     ebx, ebp
mov     ebp, ecx
shr     ebp, 5
xor     ebp, eax
sub     ebx, ebp
sub     ebx, [esi+4]
sub     eax, edx
dec     edi
jnz     short loc_1FA1
```

```
pop     edi
mov     [edi], ebx
mov     [edi+4], ecx
retn
```

```
push    edi
mov     ebx, [edi]
mov     ecx, [edi+4]
mov     edx, 9E3779B9h
mov     eax, edx
shl     eax, 5
mov     edi, 20h ; ' '
```

TEA delta

TEA round number

Classic TEA operations

```
loc_1FA1:
mov     ebp, ebx
shl     ebp, 4
sub     ecx, ebp
mov     ebp, [esi+8]
xor     ebp, ebx
sub     ecx, ebp
mov     ebp, ebx
shr     ebp, 5
xor     ebp, eax
sub     ecx, ebp
sub     ecx, [esi+0Ch]
mov     ebp, ecx
shl     ebp, 4
sub     ebx, ebp
mov     ebp, [esi]
xor     ebp, ecx
sub     ebx, ebp
mov     ebp, ecx
shr     ebp, 5
xor     ebp, eax
sub     ebx, ebp
sub     ebx, [esi+4]
sub     eax, edx
dec     edi
jnz     short loc_1FA1
```

```
pop     edi
mov     [edi], ebx
mov     [edi+4], ecx
retn
```

```
push    edi
mov     ebx, [edi]
mov     ecx, [edi+4]
mov     edx, 9E3779B9h
mov     eax, edx
shl     eax, 5
mov     edi, 20h ; ' '
```

TEA delta

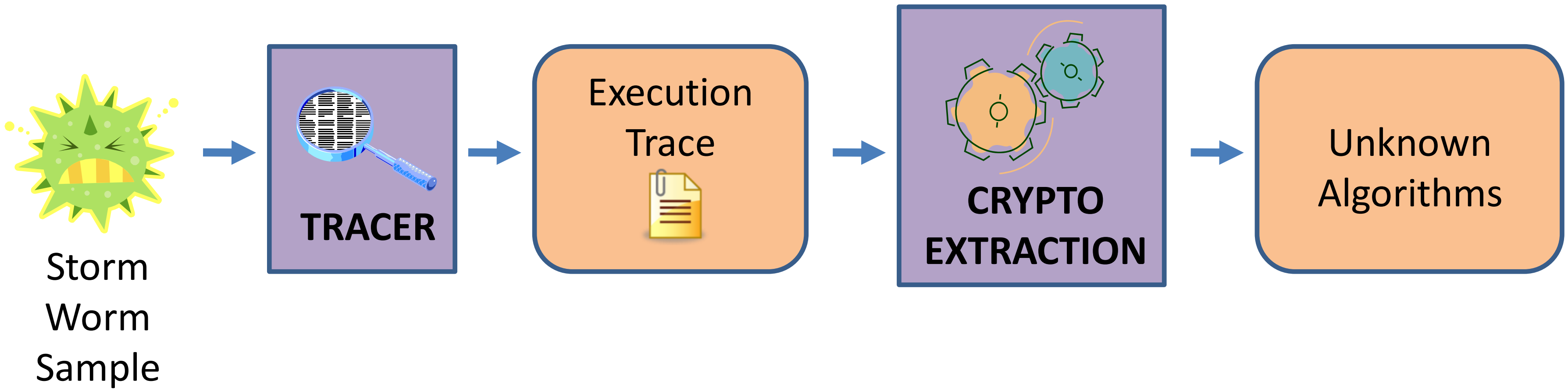
TEA round number

```
loc_1FA1:
mov     ebp, ebx
shl     ebp, 4
sub     ecx, ebp
mov     ebp, [esi+8]
xor     ebp, ebx
sub     ecx, ebp
mov     ebp, ebx
shr     ebp, 5
xor     ebp, eax
sub     ecx, ebp
sub     ecx, [esi+0Ch]
mov     ebp, ecx
shl     ebp, 4
sub     ebx, ebp
mov     ebp, [esi]
xor     ebp, ecx
sub     ebx, ebp
mov     ebp, ecx
shr     ebp, 5
xor     ebp, eax
sub     ebx, ebp
sub     ebx, [esi+4]
sub     eax, edx
dec     edi
jnz     short loc_1FA1
```

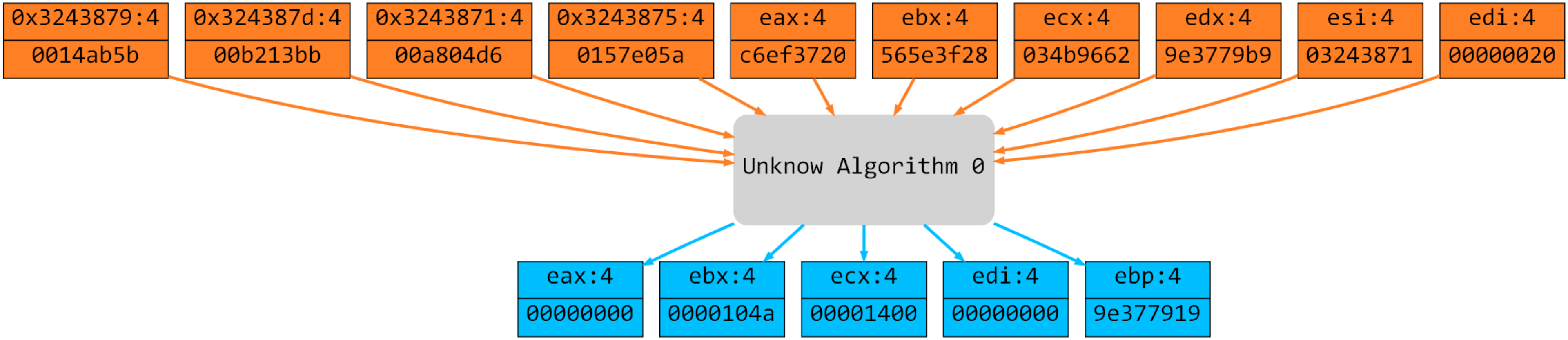
Classic TEA operations

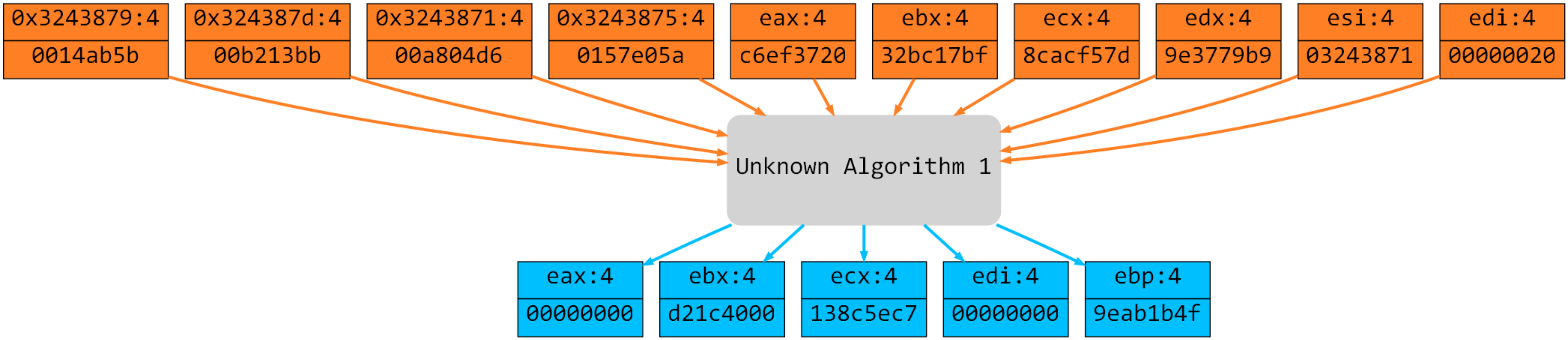
```
pop     edi
mov     [edi], ebx
mov     [edi+4], ecx
retn
```

Let’s try our tool…

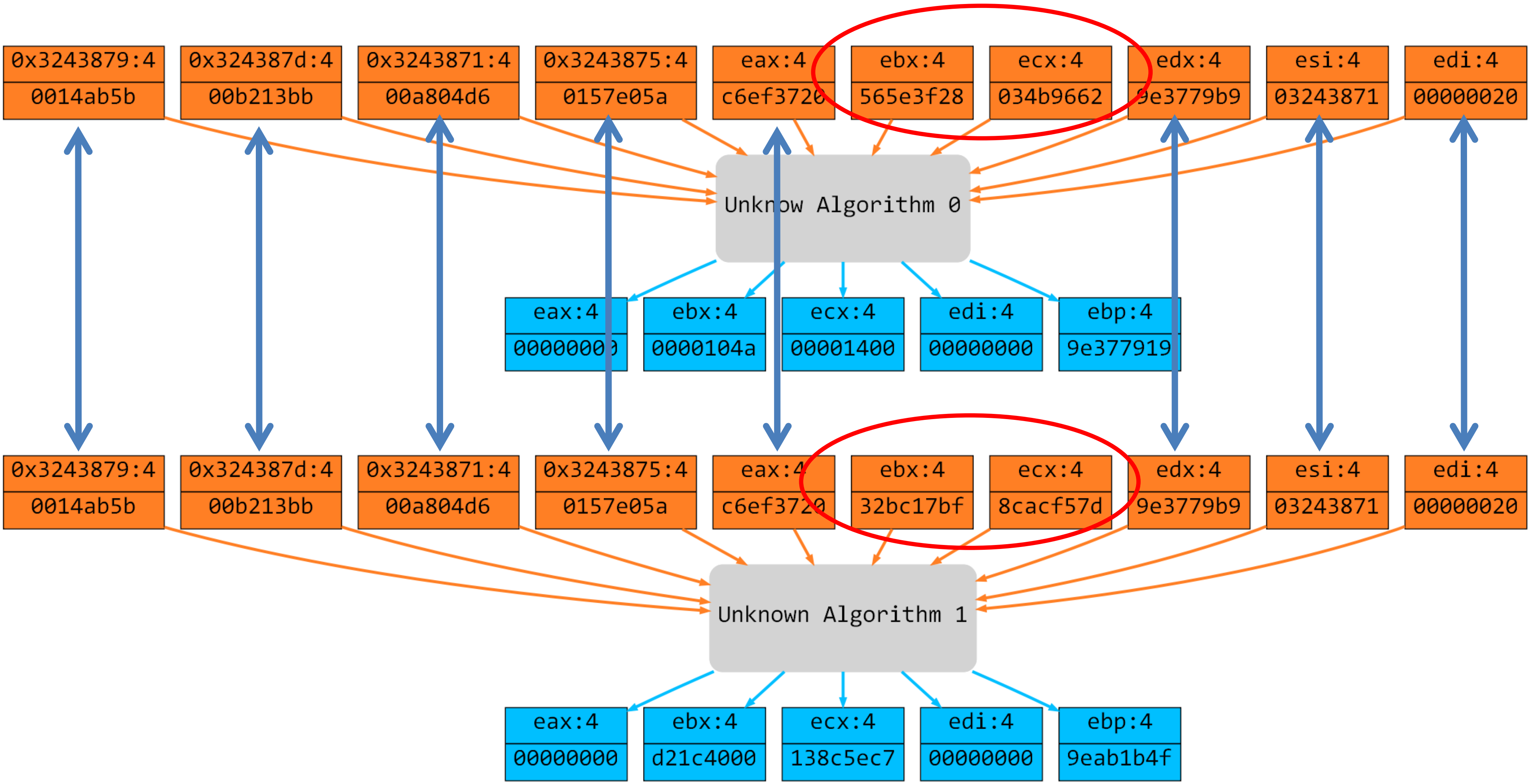

Storm Worm Sample
TRACER
Execution Trace
CRYPTO EXTRACTION
Unknown Algorithms

- For the previous loop, we extracted many unknown algorithms like these ones:

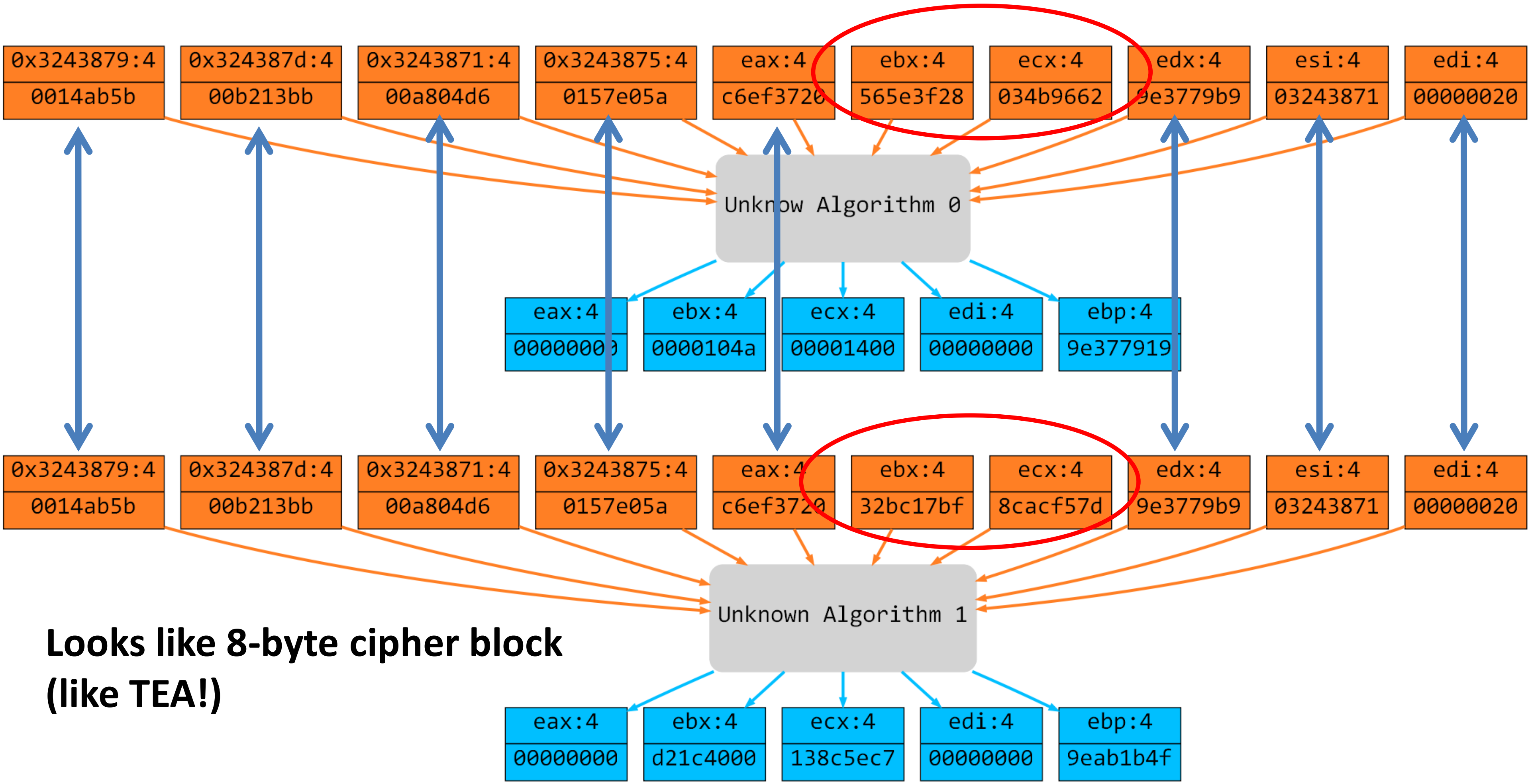




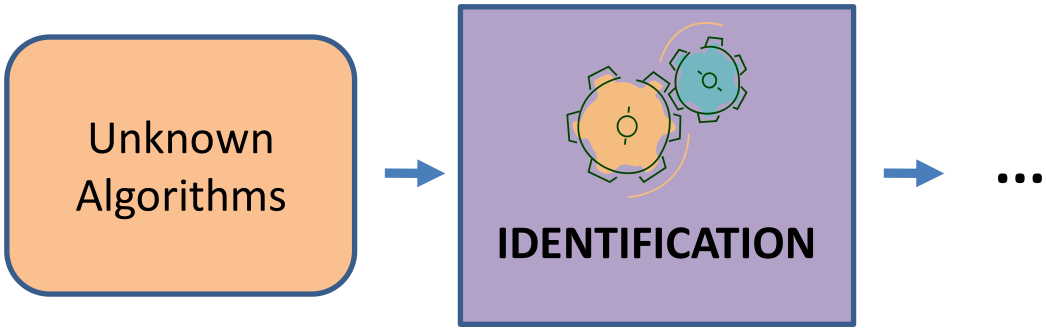
- For the previous loop, we extracted many unknown algorithms like these ones:



- For the previous loop, we extracted many unknown algorithms like these ones:



Looks like 8-byte cipher block (like TEA!)


Unknown Algorithms
IDENTIFICATION
…

```
STARTED AT
2012-04-08 14:20:30.858000
** Crypto Algorithm Identification starting !**
8 input parameters
5 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption... Unknown Algorithm stays
unkwown!
Done!
ENDED AT
2012-04-08 14:21:11.328000
```

WTF ?

Original TEA source code

```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```

```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code

```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```

```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code

```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```

```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code

```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```

```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

This is not TEA: parenthesis at the wrong place!

```
STARTED AT
2012-04-10 13:22:02.192000
** Crypto Algorithm Identification starting !**
8 input parameters
5 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for *Storm Worm* TEA decryption...
** Found *Storm Worm* TEA decryption **

 ===> Key (16 bytes) : 00a804d60157e05a0014ab5b00b213bb

 ===> Crypted text (8 bytes) : 565e3f28034b9662

 ===> Decrypted text (8 bytes) : 0000104a00001400
ENDED AT
2012-04-10 13:22:13.877000
```

Ok, Storm Worm implementation added to the base...

(this is *not* TEA)

Trojan.SilentBanker

- Several internet references about the use of TEA in SilentBanker.

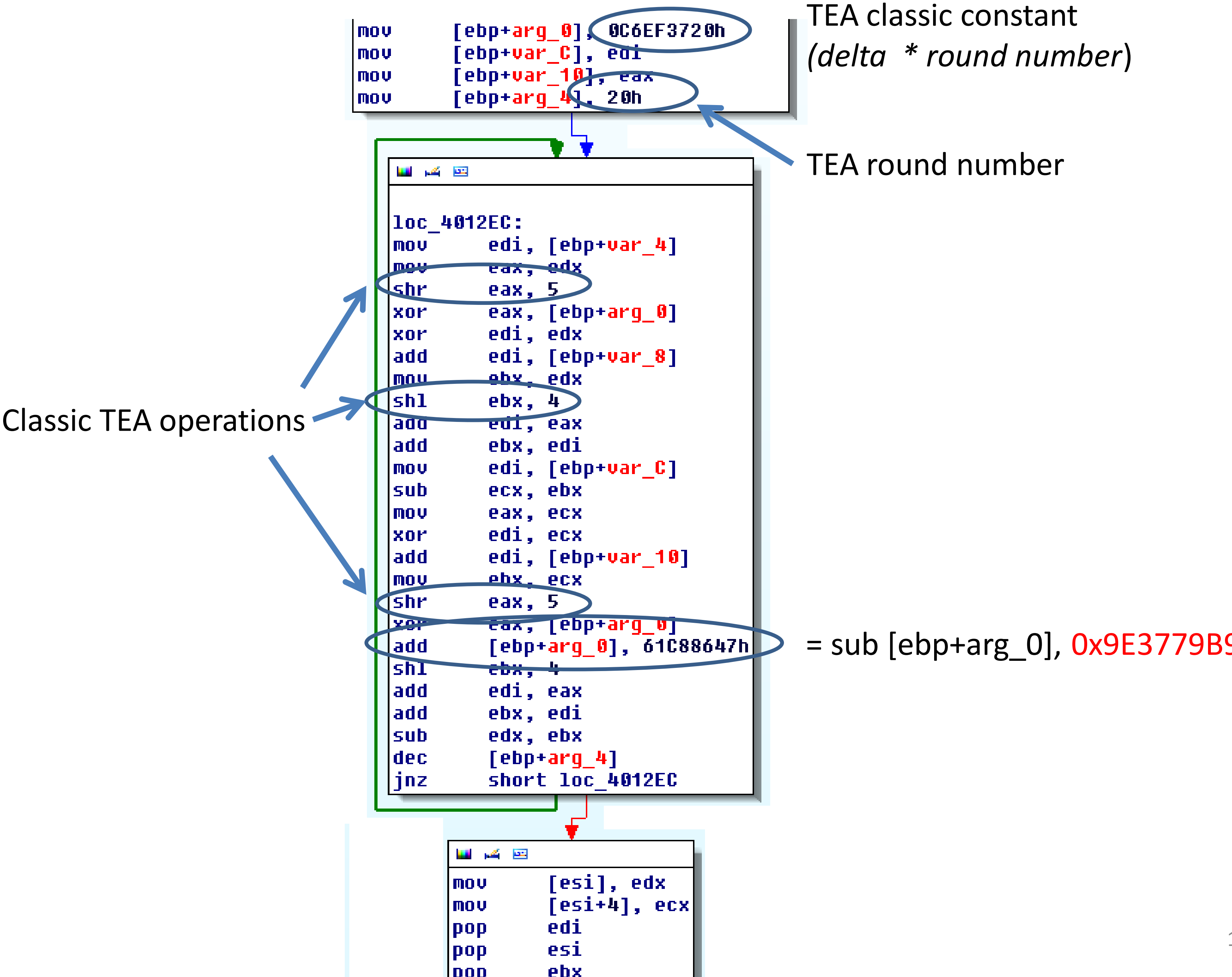
- Let's take a look to the code...

(sounds familiar, isn't it ?)

```
mov     [ebp+arg_0], 0C6EF3720h
mov     [ebp+var_C], edi
mov     [ebp+var_10], eax
mov     [ebp+arg_4], 20h
```

```
loc_4012EC:
mov     edi, [ebp+var_4]
mov     eax, edx
shr     eax, 5
xor     eax, [ebp+arg_0]
xor     edi, edx
add     edi, [ebp+var_8]
mov     ebx, edx
shl     ebx, 4
add     edi, eax
add     ebx, edi
mov     edi, [ebp+var_C]
sub     ecx, ebx
mov     eax, ecx
xor     edi, ecx
add     edi, [ebp+var_10]
mov     ebx, ecx
shr     eax, 5
xor     eax, [ebp+arg_0]
add     [ebp+arg_0], 61C88647h
shl     ebx, 4
add     edi, eax
add     ebx, edi
sub     edx, ebx
dec     [ebp+arg_4]
jnz     short loc_4012EC
```

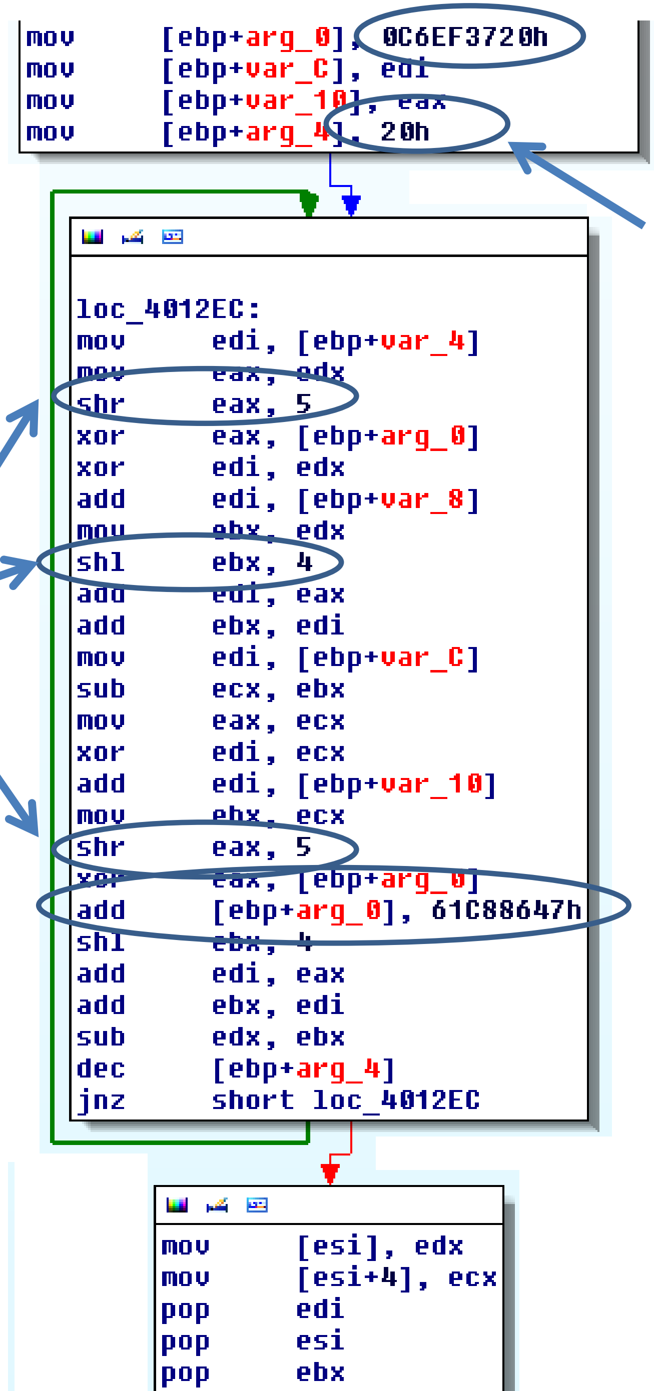
```
mov     [esi], edx
mov     [esi+4], ecx
pop     edi
pop     esi
pop     ebx
```


mov [ebp+arg_0], 0C6EF3720h
mov [ebp+var_C], edi
mov [ebp+var_10], eax
mov [ebp+arg_4], 20h
TEA classic constant
(delta * round number)
TEA round number
loc_4012EC:
mov edi, [ebp+var_4]
mov eax, edx
shr eax, 5
xor eax, [ebp+arg_0]
xor edi, edx
add edi, [ebp+var_8]
mov ebx, edx
shl ebx, 4
add edi, eax
add ebx, edi
mov edi, [ebp+var_C]
sub ecx, ebx
mov eax, ecx
xor edi, ecx
add edi, [ebp+var_10]
mov ebx, ecx
shr eax, 5
xor eax, [ebp+arg_0]
add [ebp+arg_0], 61C88647h
shl ebx, 4
add edi, eax
add ebx, edi
sub edx, ebx
dec [ebp+arg_4]
jnz short loc_4012EC
Classic TEA operations
= sub [ebp+arg_0], 0x9E3779B9
mov [esi], edx
mov [esi+4], ecx
pop edi
pop esi
pop ebx

TEA classic constant
*(delta * round number)*

TEA round number

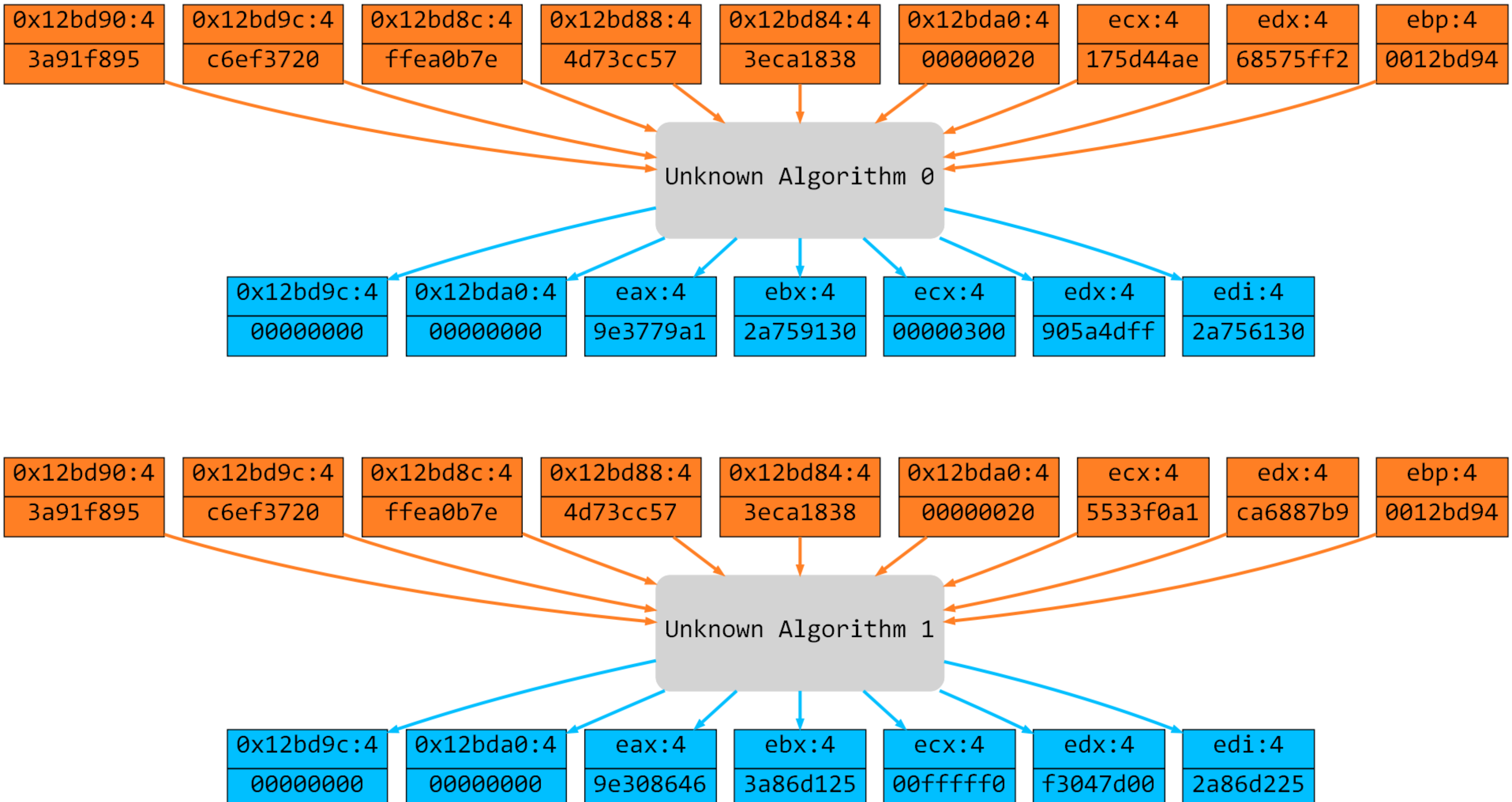
Classic TEA operations



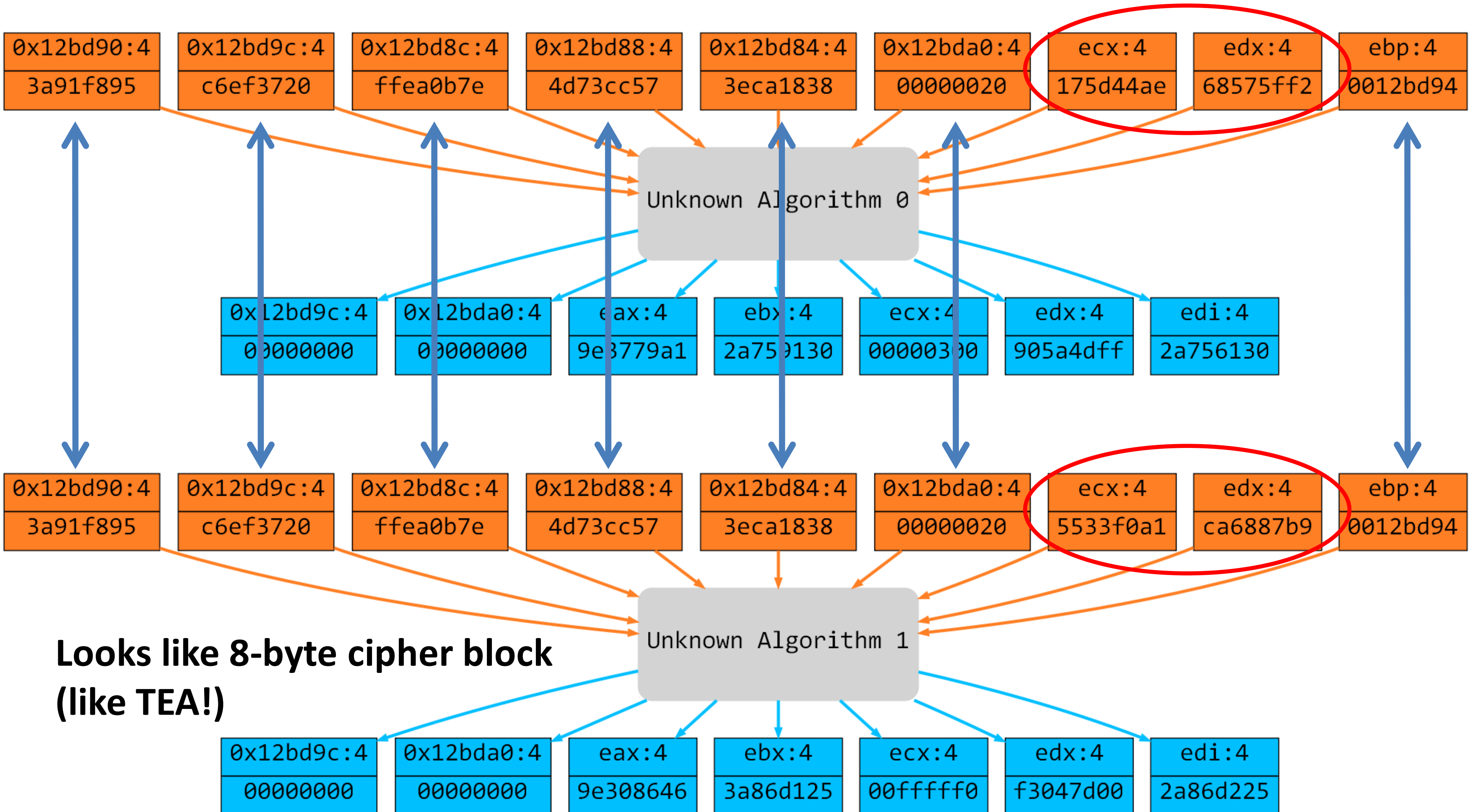
= sub [ebp+arg_0], 0x9E3779B9

Let’s try our tool…

- For the previous loop, we extracted many unknown algorithms like these ones:



- For the previous loop, we extracted many unknown algorithms like these ones:



Looks like 8-byte cipher block (like TEA!)

```
STARTED AT
2012-04-08 16:05:14.288000
** Crypto Algorithm Identification starting !**
9 input parameters
7 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption... Unknown Algorithm stays
unkwown!
Done!
```

Fail.. Again !?

```
STARTED AT
2012-04-08 16:15:22.255000
** Crypto Algorithm Identification starting !**
9 input parameters
7 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for *Storm Worm* TEA decryption...
** Found *Storm Worm* TEA decryption **

 ===> Key (16 bytes) : 4d73cc573eca18383a91f895ffea0b7e

 ===> Crypted text (8 bytes) : 68575ff2175d44ae

 ===> Decrypted text (8 bytes) : 905a4dff300
ENDED AT
2012-04-08 16:16:22.552000
```

☺ !!

Same implementation than in the Storm Worm!

- They probably both copied/pasted a wrong source code from the internet.

- Started to look for it: Google, TEA Wikipedia page,... nothing!

- At some point, I remembered something these two malware families have in common...

They both came from Russia!

Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages

Deutsch
Español
Français
Hrvatski
Italiano
Polski
Русский

Tiny Encryption Algorithm

From Wikipedia, the free encyclopedia

In cryptography, the **Tiny Encryption Algorithm** (**TE**
few lines of code. It was designed by David Wheeler a
Fast Software Encryption workshop in Leuven in 1994

The cipher is not subject to any patents.

Contents [hide]

Properties

TEA operates on two 32-bit unsigned integers (could
a suggested 64 rounds, typically implemented in pairs
exactly the same way for each cycle. Different multiple
rounds. The magic constant, 2654435769 or 9E3779

TEA has a few weaknesses. Most notably, it suffers fr
effective key size is only 126 bits.[3] As a result, TEA i
hacking Microsoft's Xbox game console, where the ci
requires 2^{23} chosen plaintexts under a related-key pai
designed.

Ссылки

- Страница алгоритма шифрования TEA
- Roger M. Needham and David J. Wheeler. «TEA, a Tiny Encryption Algorithm»(PDF)
- Andew Hang. «Hacking the Xbox: an introduction to reverse engineering»[1]
- Журнал «Хакер Онлайн», TEA: блочный шифр своими руками, (2004)[2]
- Paris Kitsos,Yan Zhang. «RFID Security: Techniques, Protocols and System-On-Chip Design»[3]
- David J. Wheeler and Roger M. Needham. «Correction to xtea.» Technical report, Computer Laboratory,
- Roger M. Needham and David J. Wheeler. «Tea extensions.» Technical report, Computer Laboratory, Ur
- Test vectors for TEA
- JavaScript implementation of XXTEA with Base64

Журнал Новости Форум Видео
Bugtrack Статьи Блог Поиск

Взлом как образ мысли

Интервью с человеком, который, как оказалось, является не только талантливым пентестером в одной из крупных ИБ-компании, но и хакером-ветераном, который уверенными шагами вышел на свет и прикоснулся к истории российской хак-сцены....

Полный гı голосоваı

Конкурсы с гс посетителей, интерес вызы лакомые приз

TEA: блочный шифр своими руками

Дата: **22.04.2004** | slon

Оценка: 11 статьи

Tweet

В данном тексте хотелось бы затронуть такую животрепещущую тему, как шифрование файлов. Вообще нужно различать два вида шифрования файлов:

- шифрование для себя (чтобы ваши файлы никто, кроме вас не "читал")
- шифрование для других (чтобы ваши файлы "читал" только адресат)

Russian Website TEA source code

```
push edi ; Сохраняем
mov ebx,v0 ; Кладём
mov ecx,v1 ; В ecx кл
mov edx,9e3779b9h ;
mov eax,edx ; Кладём
shl eax,5 ; Сдвиг eax
mov edi,32 ; Кладём
DLoopR:
mov ebp,ebx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ecx,ebp ; Отнима
mov ebp,k2 ; Кладём
xor ebp,ebx ; XOR'им
sub ecx,ebp ; Отнима
mov ebp,ebx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ecx,ebp ; Отнима
sub ecx,k3 ; Отнимае
;
mov ebp,ecx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ebx,ebp ; Отнима
mov ebp,k0 ; Кладём
xor ebp,ecx ; XOR'им
sub ebx,ebp ; Отнима
mov ebp,ecx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ebx,ebp ; Отнима
sub ebx,k1 ; Отнимае
sub eax,edx ; Отнима
dec edi ; Уменьшаем
jnz DLoopR ; Дешифр

pop edi ; Вынимаем
mov v0,ebx ; Кладём
mov v1,ecx ; В отведё
ret ; Возврат из подп
```

```
push edi ; Сохраняем
mov ebx,v0 ; Кладём
mov ecx,v1 ; В ecx кл
mov edx,9e3779b9h ;
mov eax,edx ; Кладём
shl eax,5 ; Сдвиг eax
mov edi,32 ; Кладём в
DLoopR:
mov ebp,ebx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ecx,ebp ; Отнима
mov ebp,k2 ; Кладём
xor ebp,ebx ; XOR'им
sub ecx,ebp ; Отнима
mov ebp,ebx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ecx,ebp ; Отнима
sub ecx,k3 ; Отнимаем
;
mov ebp,ecx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ebx,ebp ; Отнима
mov ebp,k0 ; Кладём
xor ebp,ecx ; XOR'им
sub ebx,ebp ; Отнима
mov ebp,ecx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ebx,ebp ; Отнима
sub ebx,k1 ; Отнимае
sub eax,edx ; Отнима
dec edi ; Уменьшаем
jnz DLoopR ; Дешифр

pop edi ; Вынимаем и
mov v0,ebx ; Кладём
mov v1,ecx ; В отведё
ret ; Возврат из подп
```

Russian Website TEA source code

=

```
push    edi
mov     ebx, [edi]
mov     ecx, [edi+4]
mov     edx, 9E3779B9h
mov     eax, edx
shl     eax, 5
mov     edi, 20h ; ' '

loc_1FA1:
mov     ebp, ebx
shl     ebp, 4
sub     ecx, ebp
mov     ebp, [esi+8]
xor     ebp, ebx
sub     ecx, ebp
mov     ebp, ebx
shr     ebp, 5
xor     ebp, eax
sub     ecx, ebp
sub     ecx, [esi+0Ch]
mov     ebp, ecx
shl     ebp, 4
sub     ebx, ebp
mov     ebp, [esi]
xor     ebp, ecx
sub     ebx, ebp
mov     ebp, ecx
shr     ebp, 5
xor     ebp, eax
sub     ebx, ebp
sub     ebx, [esi+4]
sub     eax, edx
dec     edi
jnz     short loc_1FA1

pop     edi
mov     [edi], ebx
mov     [edi+4], ecx
retn
```

Storm Worm

RC4

MORE EXAMPLES!

RC4 (1)

- RC4 algorithm:
  - Stream cipher
  - Variable-length key
  - Two loops generate a pseudorandom stream into a 256 bytes substitution-box (S-BOX).
  - A final loop does the actual decryption.

- We have to extend our model to **regroup different loops into a same algorithm**.

Interlude: Loop Data Flow

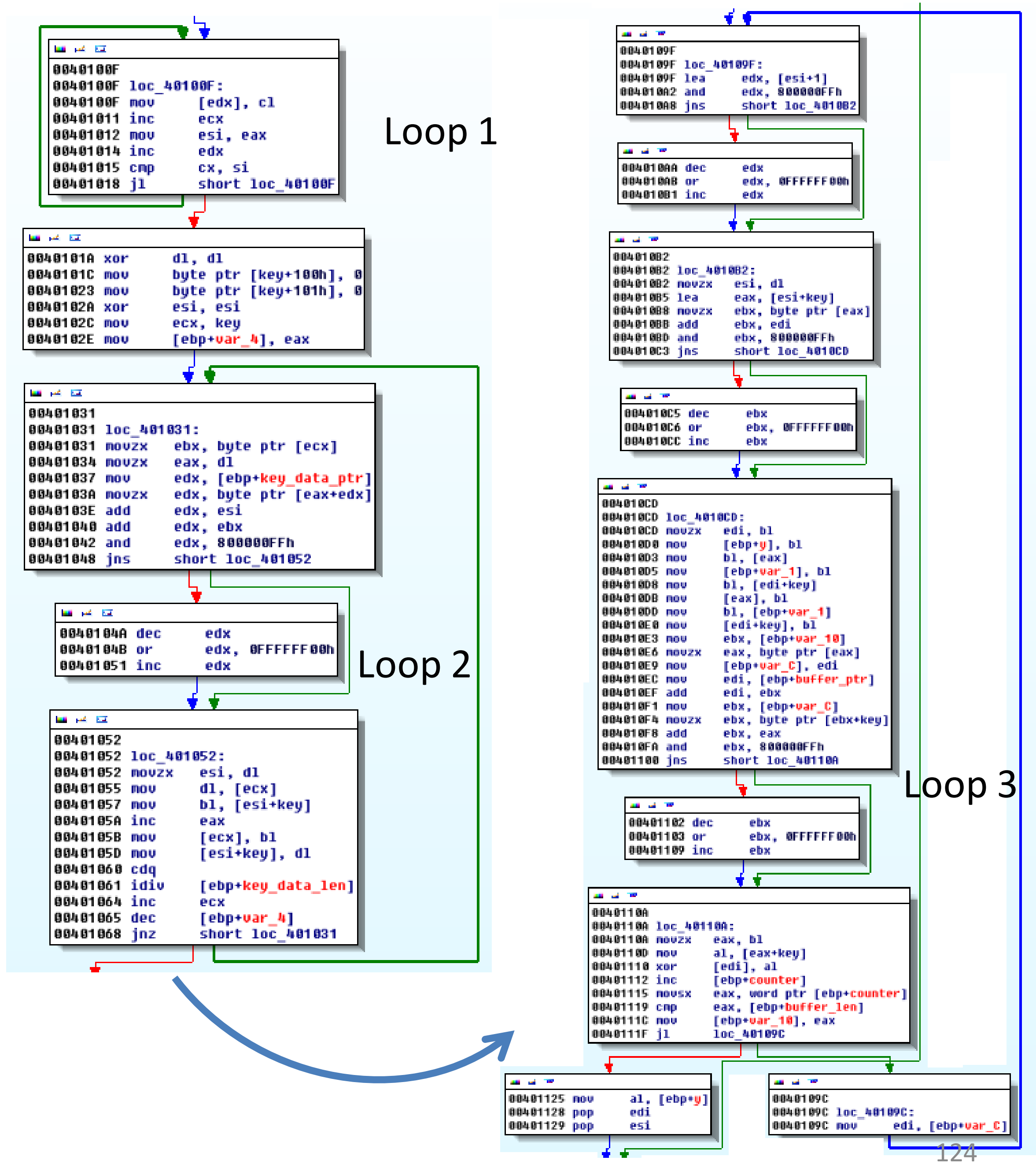
- Two loops *L1* and *L2* are in the same algorithm:
  - If *L1* started before *L2* in the trace.
  - If *L2* uses as input parameter an output parameter of *L1.*

(or the contrary!)

RC4 (2)

- We built a toy program calling the RC4 decryption function on:
  - Key : **“`SuperKeyIsASuperKey`” `(19 bytes)`**
  - Encrypted text: **“`AAA....AA`” `(1024 bytes)`**

Statically speaking it looks like this...



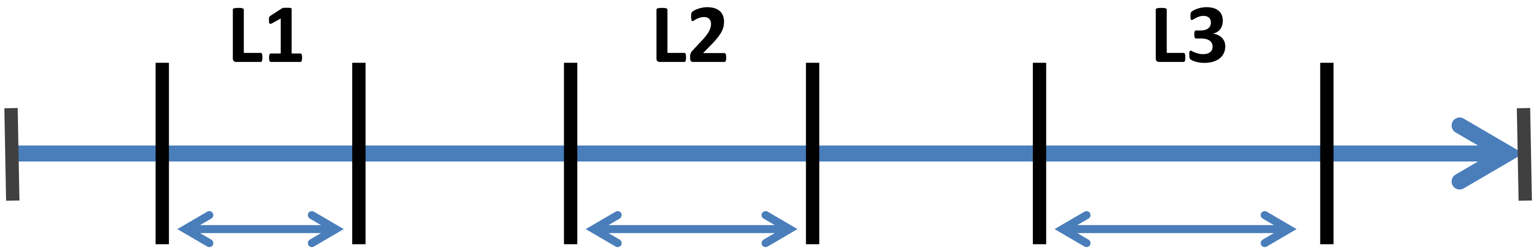
Tools	Answer
Crypto Searcher	∅
Draca v0.5.7b	∅
Findcrypt v2	∅
Hash & Crypto Detector v1.4	∅
PEiD KANAL v2.92	∅
Kerckhoffs	∅
Signsrch 0.1.7	∅
SnD Crypto Scanner v0.5b	∅

Tools	Answer
Crypto Searcher	Ø
Draca v0.5.7b	Ø
Findcrypt v2	Ø
Hash & Crypto Detector v1.4	Ø
PEiD KANAL v2.92	Ø
Kerckhoffs	Ø
Signsrch 0.1.7	Ø
SnD Crypto Scanner v0.5b	Ø

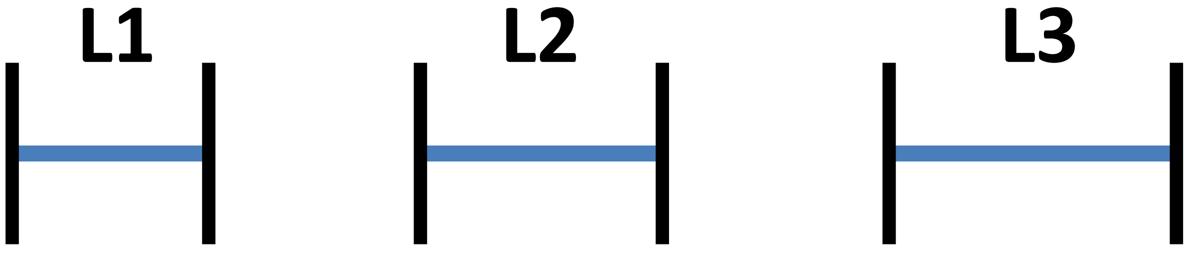
Let's try our tool...

Step 1 : Gather Execution Trace

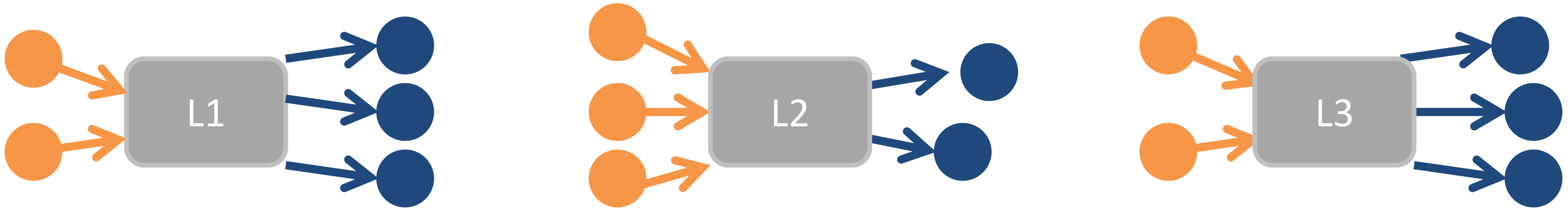
Step 2 : Recognize Loops



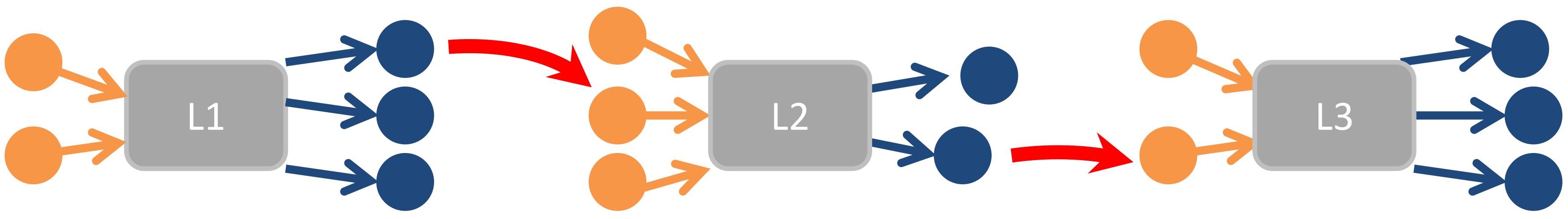
Step 2 : Recognize Loops



Step 3 : Define Loop Parameters



Step 4 : Connect Loops With Data-Flow

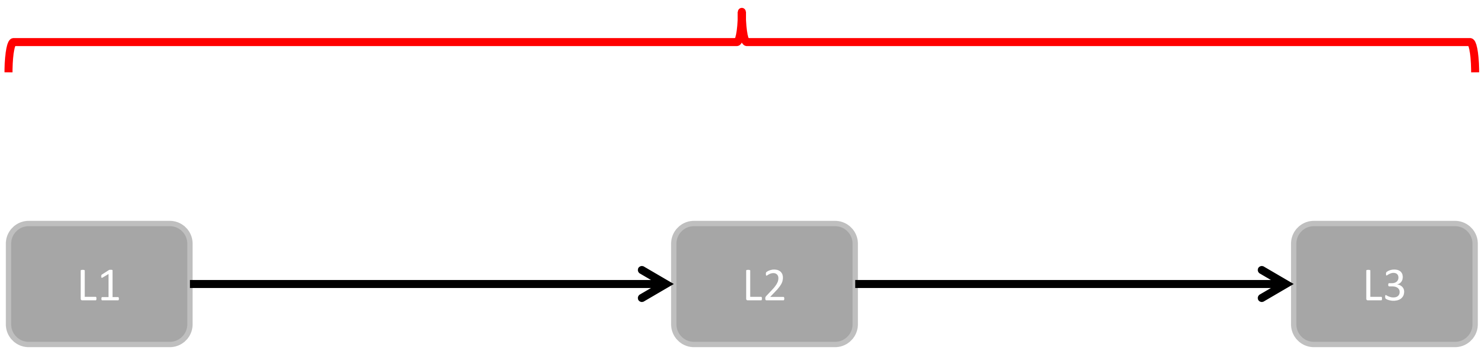


Loop Data Flow Graph (oriented, acyclic)

We consider each path in the graph as a possible cryptographic algorithm!

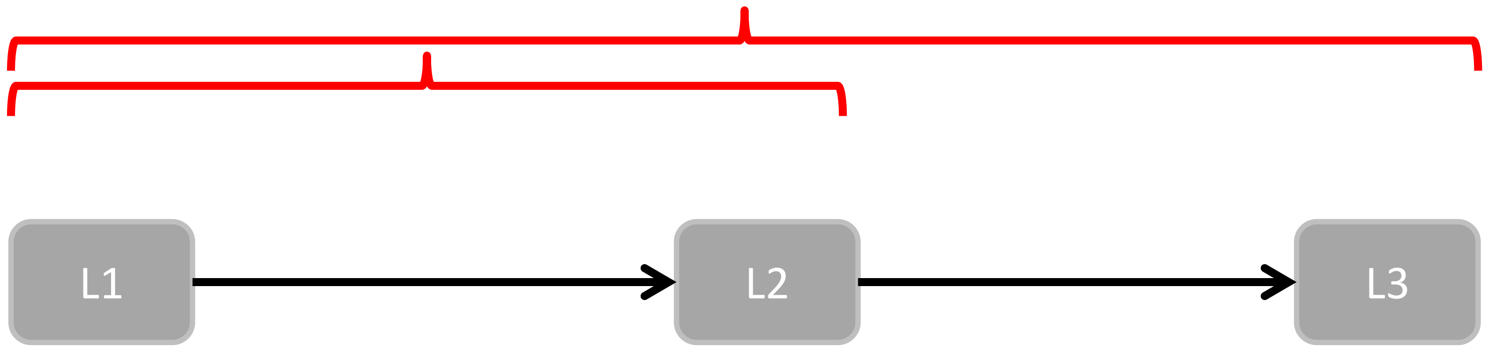
(in order to deal with algorithm combinations)

We consider each path in the graph as a possible cryptographic algorithm!



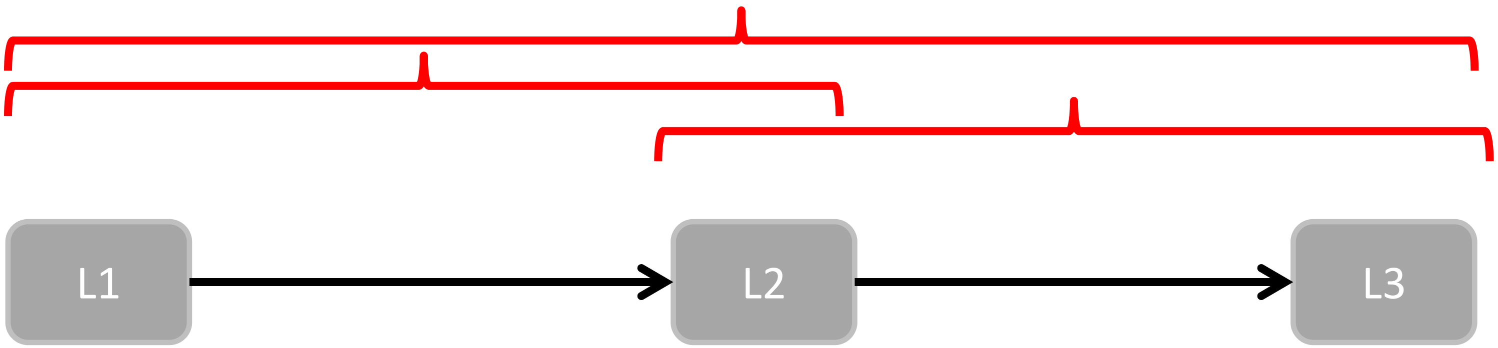
(in order to deal with algorithm combinations)

We consider each path in the graph as a possible cryptographic algorithm!



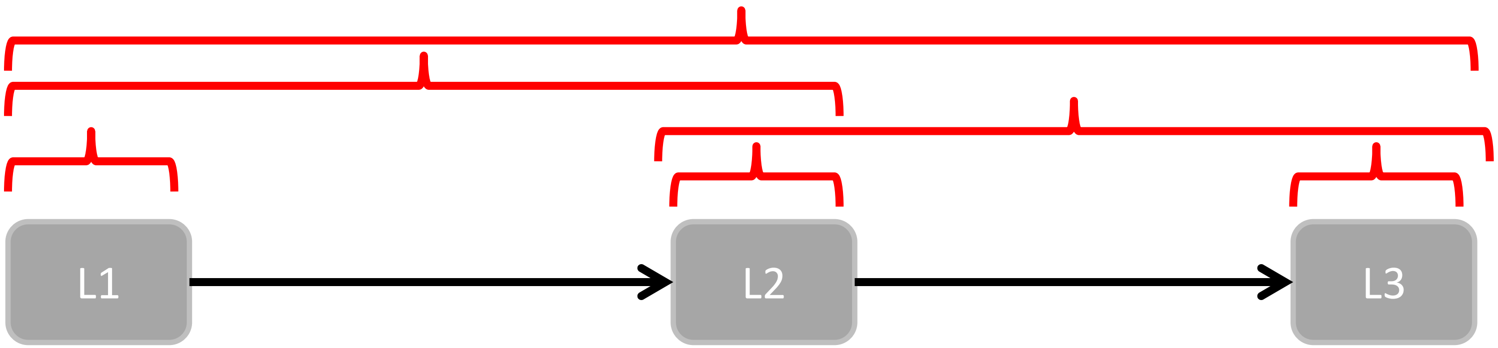
(in order to deal with algorithm combinations)

We consider each path in the graph as a possible cryptographic algorithm!



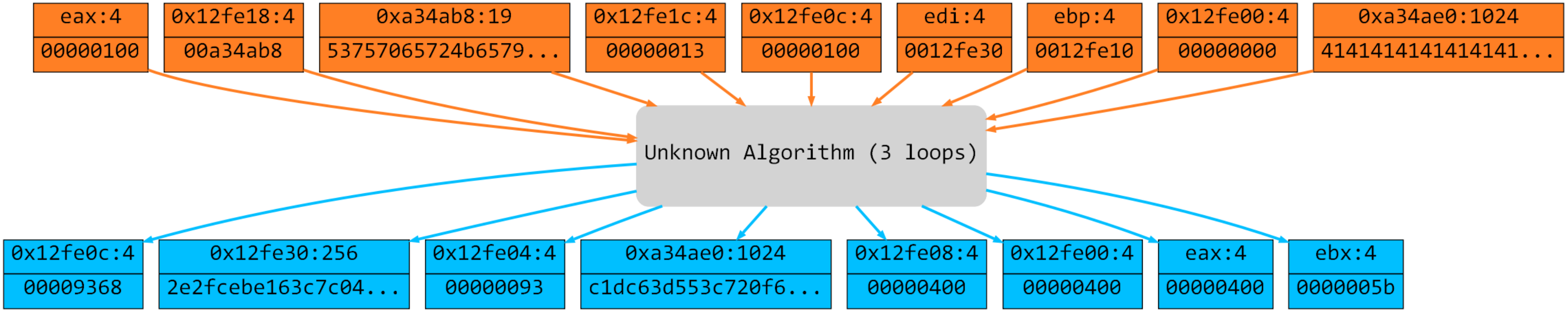
(in order to deal with algorithm combinations)

We consider each path in the graph as a possible cryptographic algorithm!

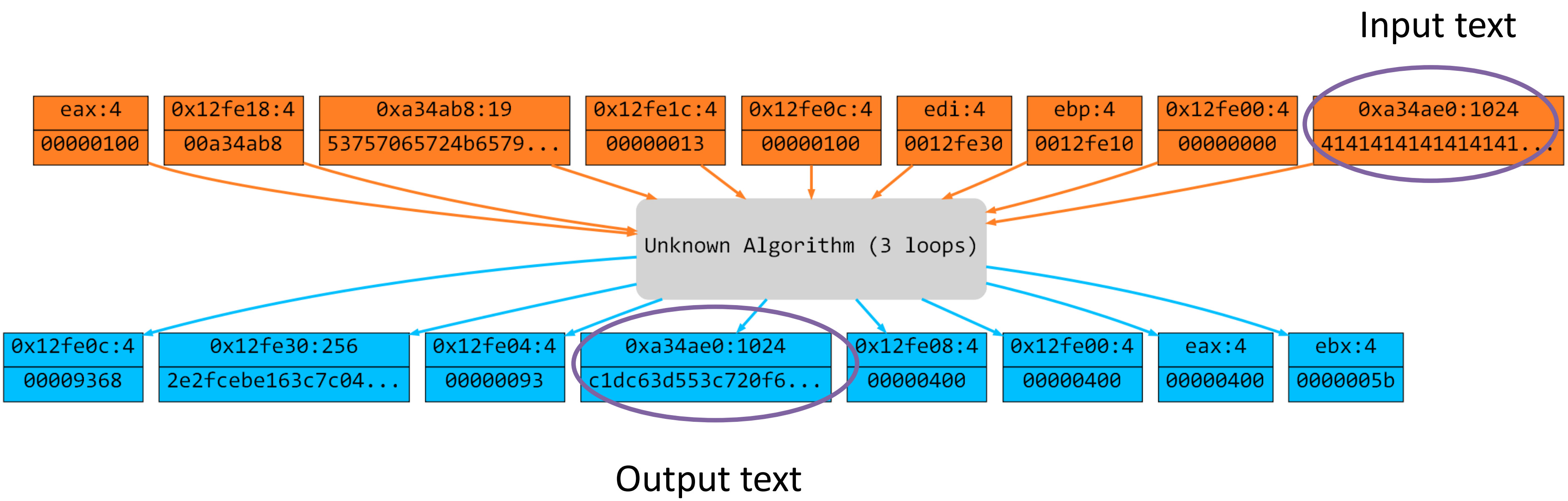


(in order to deal with algorithm combinations)

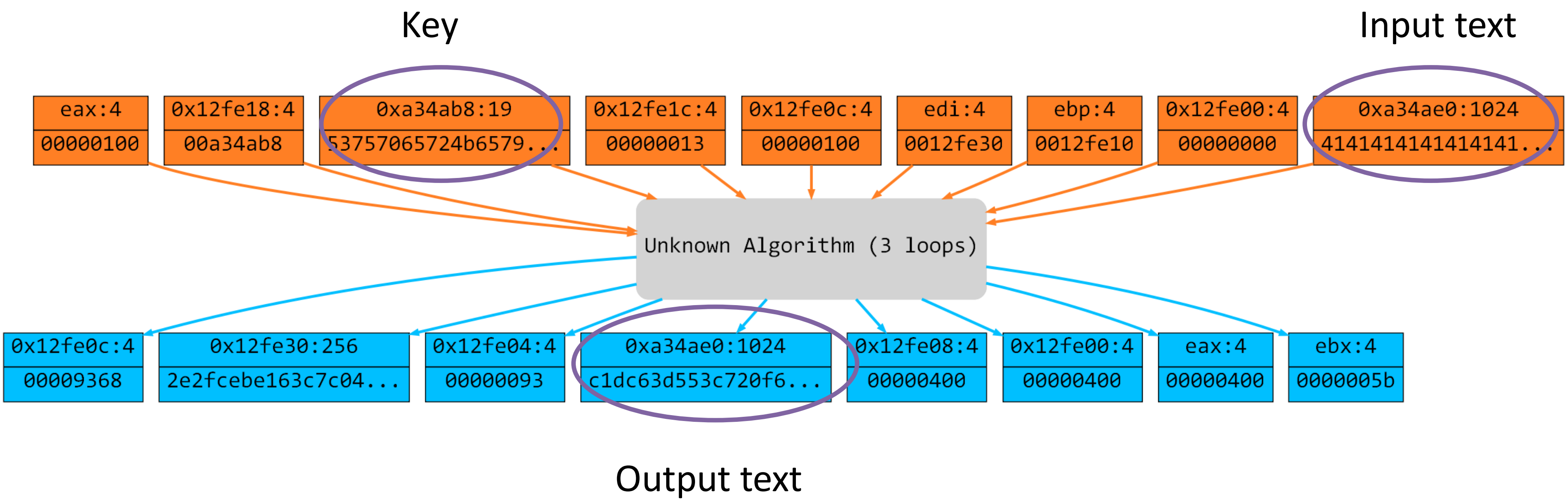
Final model for the longest path



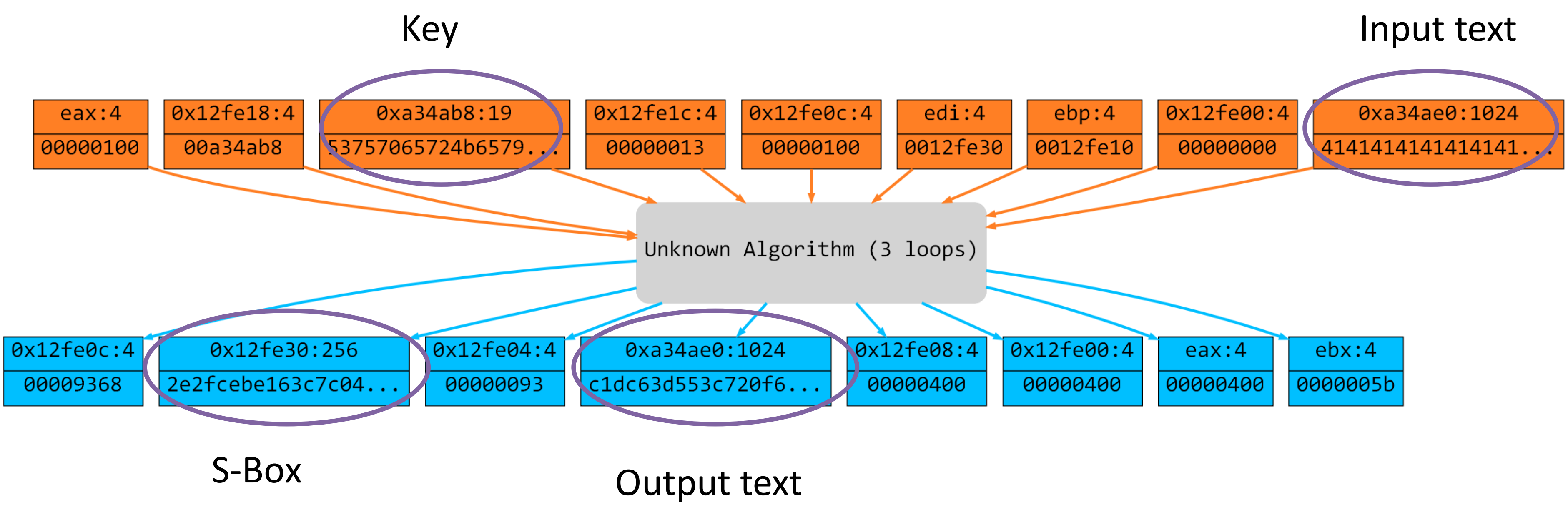
Final model for the longest path



Final model for the longest path



Final model for the longest path



```
STARTED AT
2012-04-08 19:34:12.959000
** Crypto Algorithm Identification starting !**
3 input parameters
3 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for RC4...

** Found RC4 **

 ===> Key (19 bytes) : 53757065724b657949734153757065724b6579

 ===> Crypted text (1024 bytes) :  41414141414141...

 ===> Decrypted text (1024 bytes) :  c1dc63d553c720f6...
ENDED AT
2012-04-08 19:34:12.990000
```

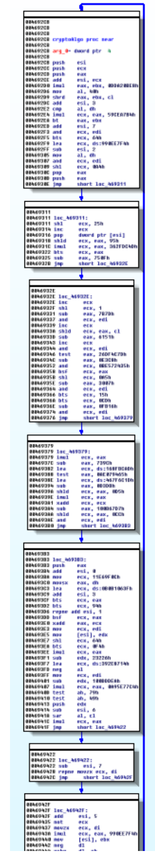
Win32.Sality.AA

- Several internet references about the use of RC4 in Sality.AA protection layers.

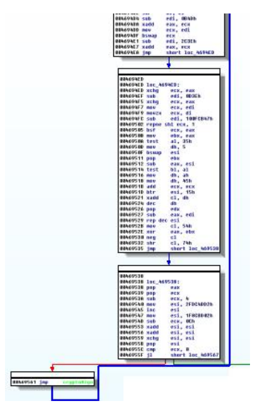
- Let's take a look...

(suspense...)

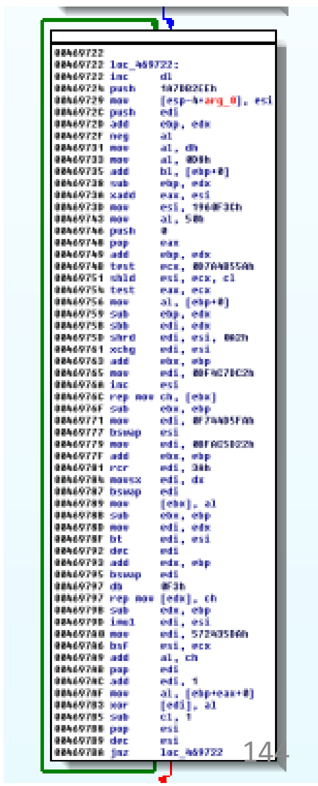
Loop 1

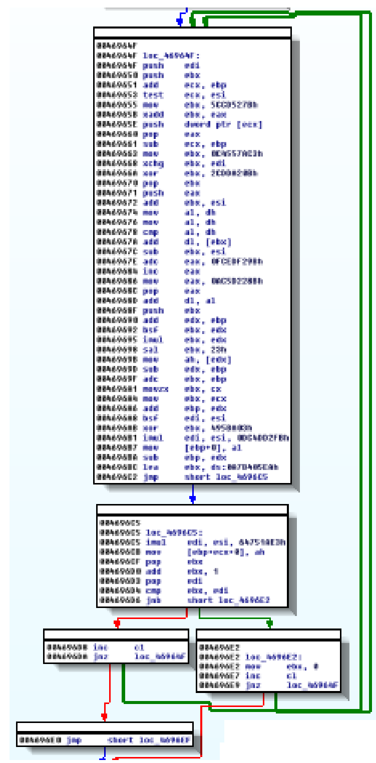


....

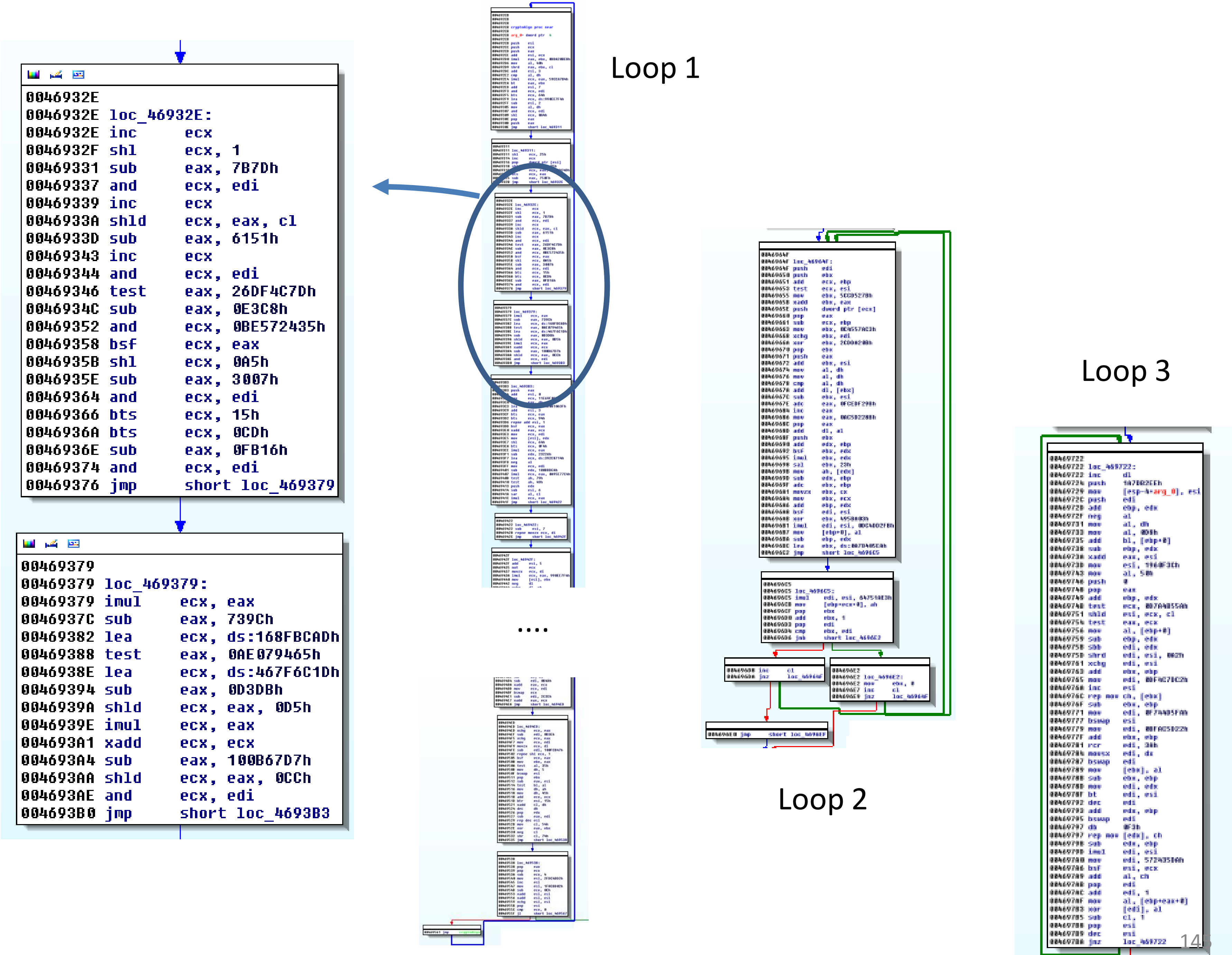


Loop 3





Loop 2



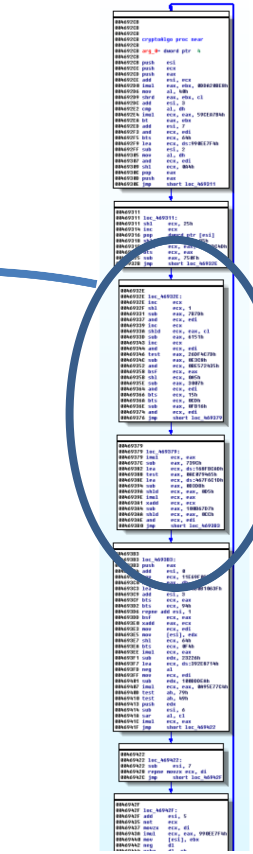
0046932E
0046932E loc_46932E:
0046932E inc ecx
0046932F shl ecx, 1
00469331 sub eax, 7B7Dh
00469337 and ecx, edi
00469339 inc ecx
0046933A shld ecx, eax, cl
0046933D sub eax, 6151h
00469343 inc ecx
00469344 and ecx, edi
00469346 test eax, 26DF4C7Dh
0046934C sub eax, 0E3C8h
00469352 and ecx, 0BE572435h
00469358 bsf ecx, eax
0046935B shl ecx, 0A5h
0046935E sub eax, 3007h
00469364 and ecx, edi
00469366 bts ecx, 15h
0046936A bts ecx, 0CDh
0046936E sub eax, 0FB16h
00469374 and ecx, edi
00469376 jmp short loc_469379
00469379
00469379 loc_469379:
00469379 imul ecx, eax
0046937C sub eax, 739Ch
00469382 lea ecx, ds:168FBCADh
00469388 test eax, 0AE079465h
0046938E lea ecx, ds:467F6C1Dh
00469394 sub eax, 0D3DBh
0046939A shld ecx, eax, 0D5h
0046939E imul ecx, eax
004693A1 xadd ecx, ecx
004693A4 sub eax, 100B67D7h
004693AA shld ecx, eax, 0CCh
004693AE and ecx, edi
004693B0 jmp short loc_4693B3
Loop 1
....
Loop 2
Loop 3

Hmpf.. Let's try!

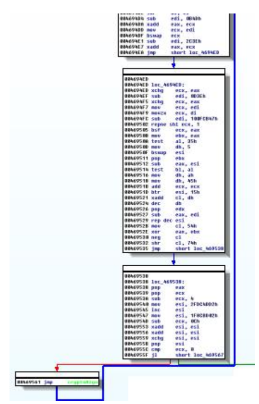
```
0046932E
0046932E loc_46932E:
0046932E inc     ecx
0046932F shl     ecx, 1
00469331 sub     eax, 7B7Dh
00469337 and     ecx, edi
00469339 inc     ecx
0046933A shld    ecx, eax, cl
0046933D sub     eax, 6151h
00469343 inc     ecx
00469344 and     ecx, edi
00469346 test    eax, 26DF4C7Dh
0046934C sub     eax, 0E3C8h
00469352 and     ecx, 0BE572435h
00469358 bsf     ecx, eax
0046935B shl     ecx, 0A5h
0046935E sub     eax, 3007h
00469364 and     ecx, edi
00469366 bts     ecx, 15h
0046936A bts     ecx, 0CDh
0046936E sub     eax, 0FB16h
00469374 and     ecx, edi
00469376 jmp     short loc_469379
```

```
00469379
00469379 loc_469379:
00469379 imul    ecx, eax
0046937C sub     eax, 739Ch
00469382 lea     ecx, ds:168FBCADh
00469388 test    eax, 0AE079465h
0046938E lea     ecx, ds:467F6C1Dh
00469394 sub     eax, 0D3DBh
0046939A shld    ecx, eax, 0D5h
0046939E imul    ecx, eax
004693A1 xadd    ecx, ecx
004693A4 sub     eax, 100B67D7h
004693AA shld    ecx, eax, 0CCh
004693AE and     ecx, edi
004693B0 jmp     short loc_4693B3
```

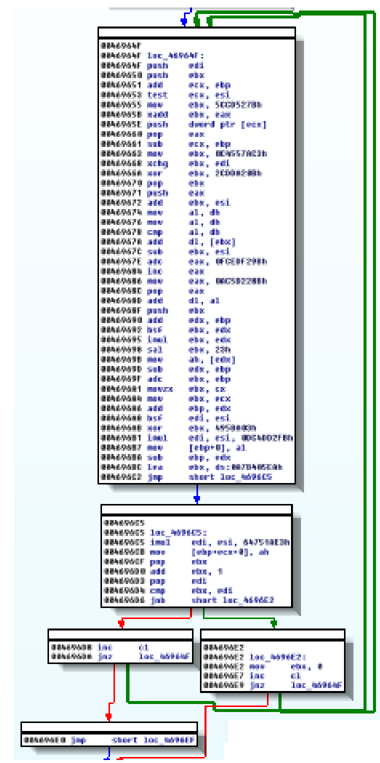
Loop 1



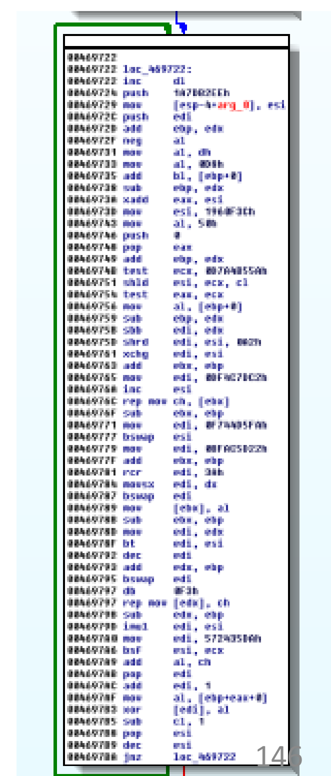
....

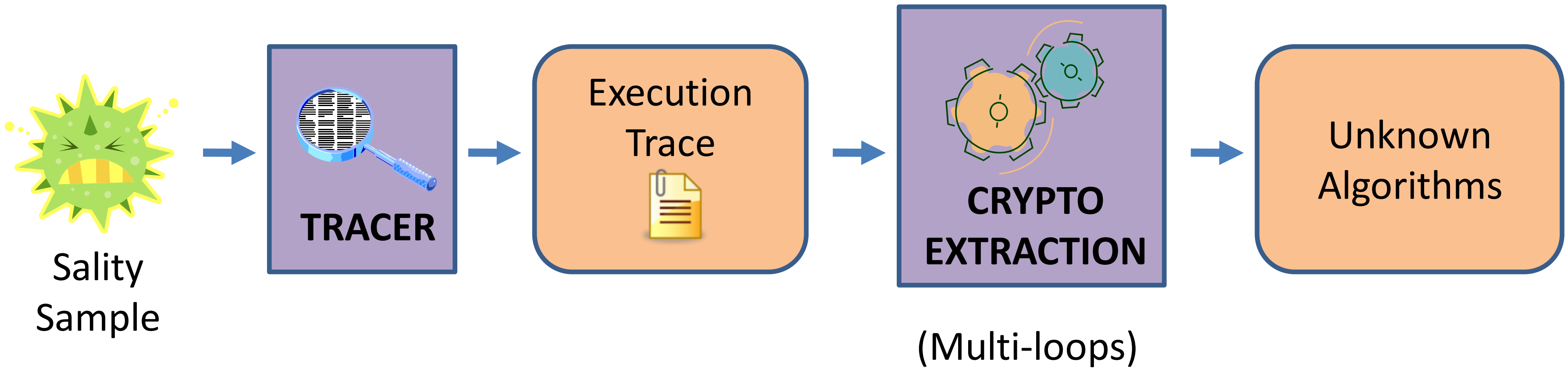


Loop 2

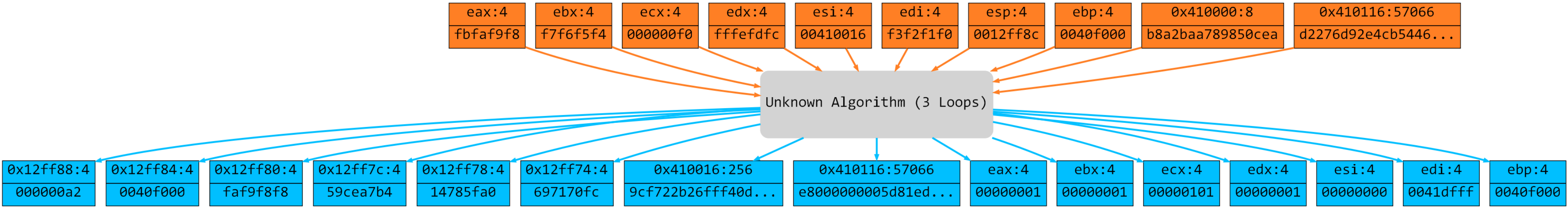


Loop 3

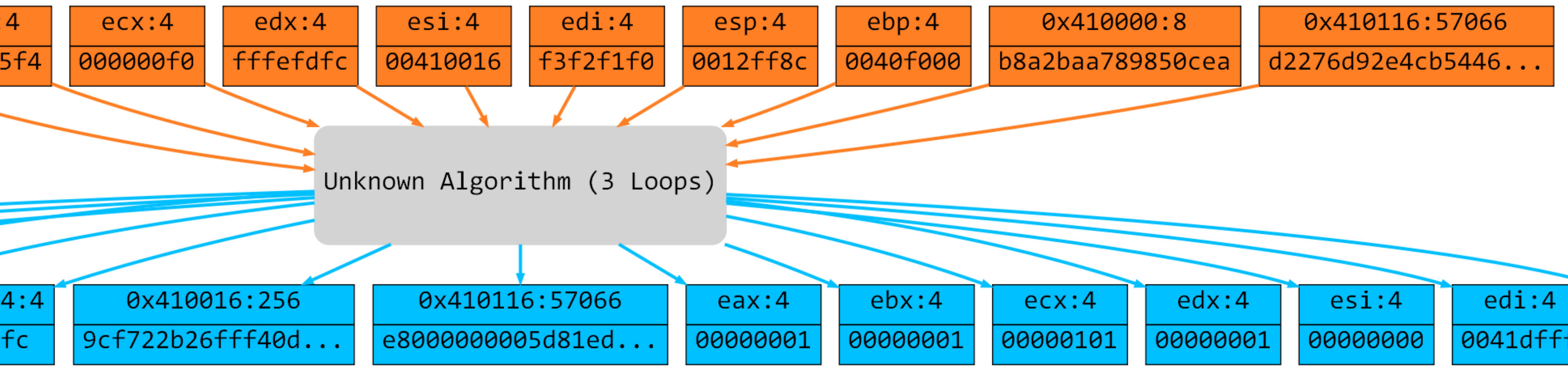



Sality
Sample
TRACER
Execution
Trace
CRYPTO
EXTRACTION
(Multi-loops)
Unknown
Algorithms

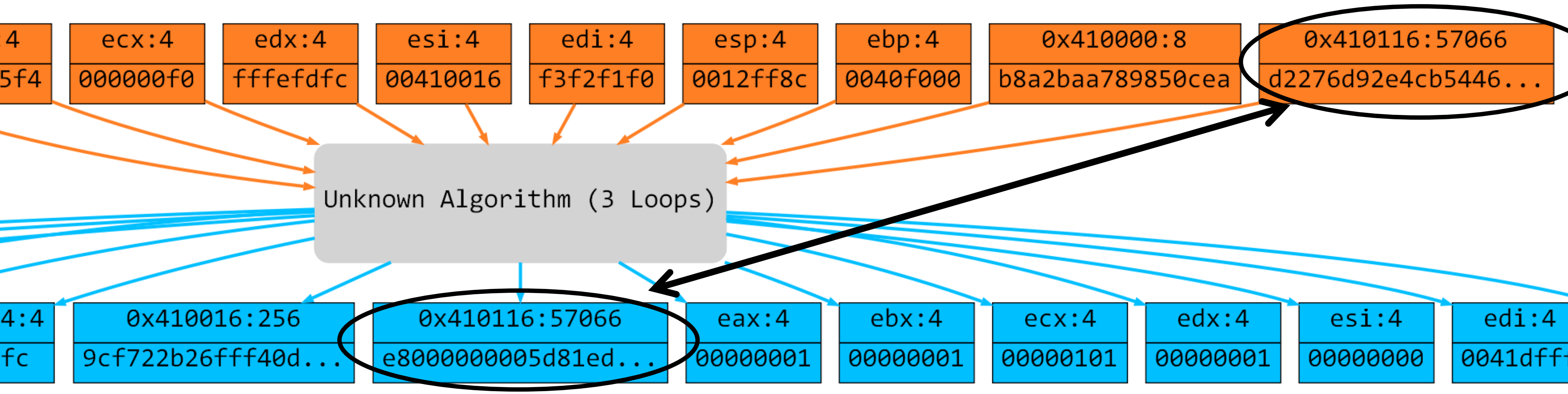
For the previous 3 loops, we extracted one algorithm:



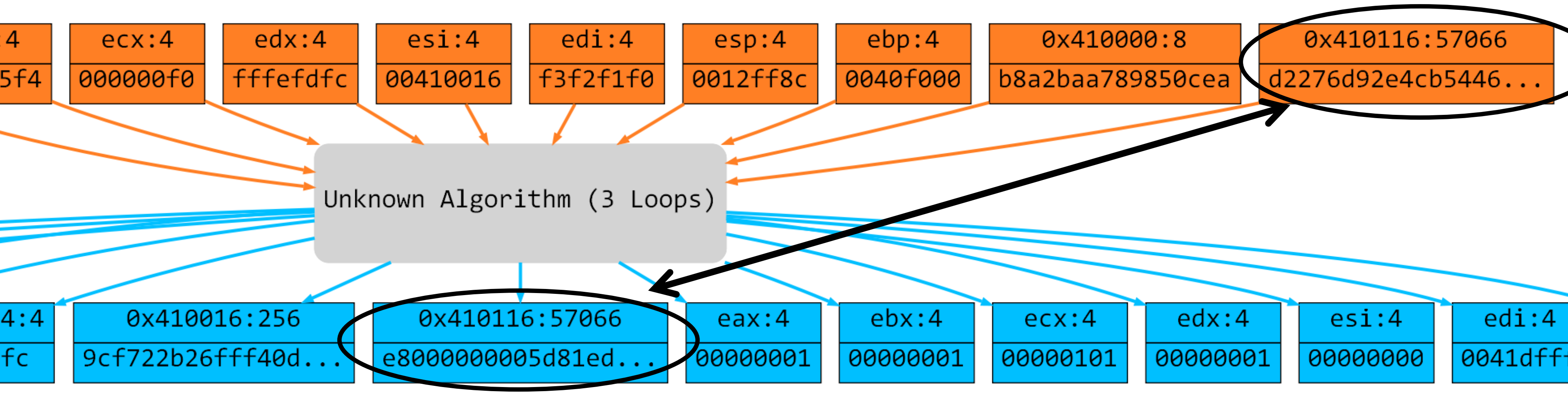
For the previous 3 loops, we extracted one algorithm:



For the previous 3 loops, we extracted one algorithm:

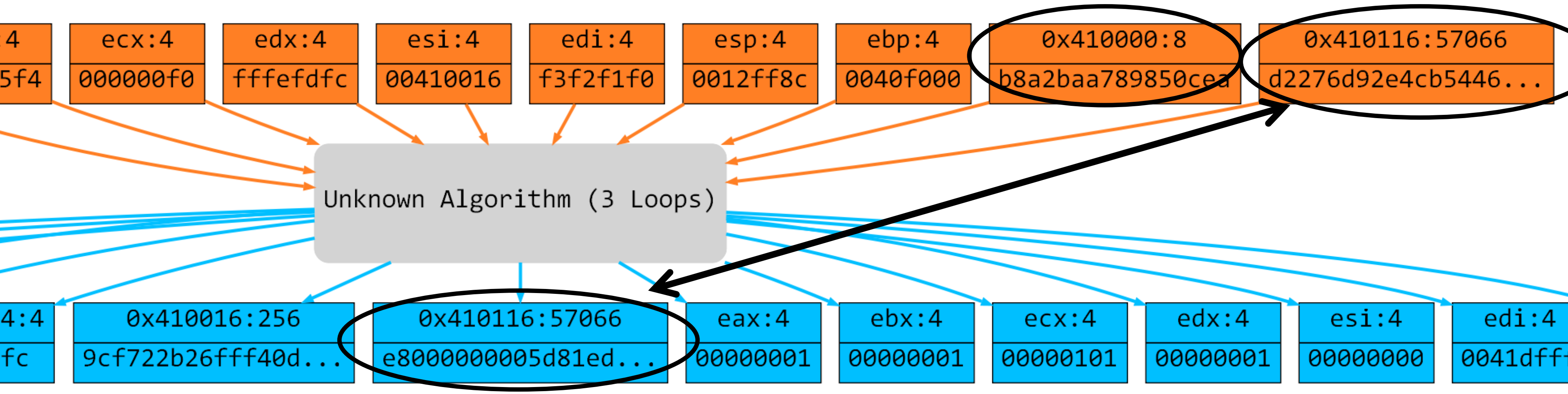


For the previous 3 loops, we extracted one algorithm:



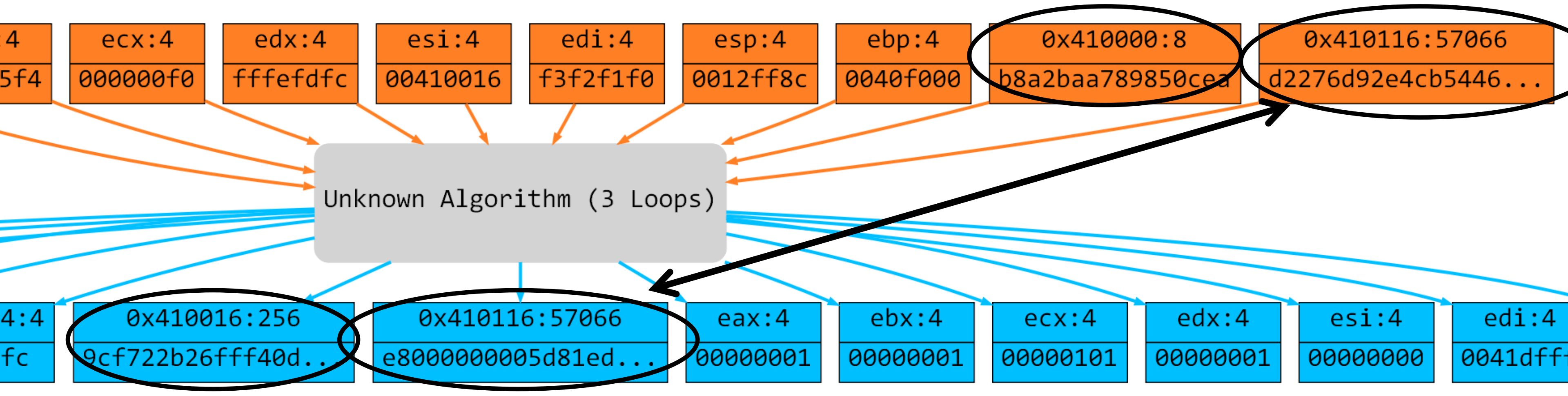
X86 ExecutableCode!

For the previous 3 loops, we extracted one algorithm:

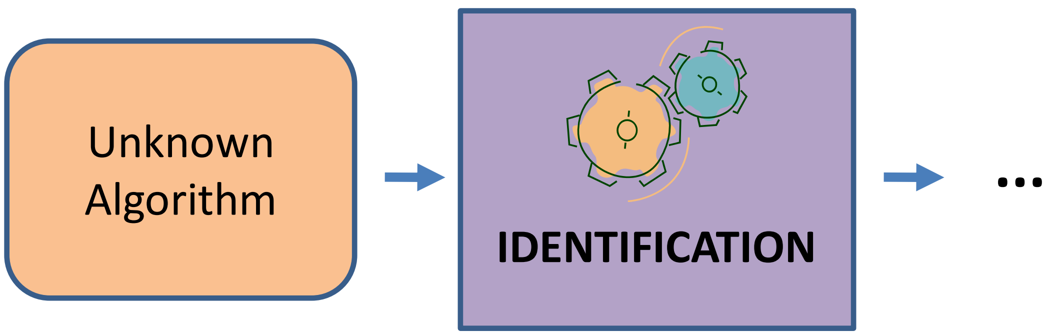


X86 ExecutableCode!

For the previous 3 loops, we extracted one algorithm:

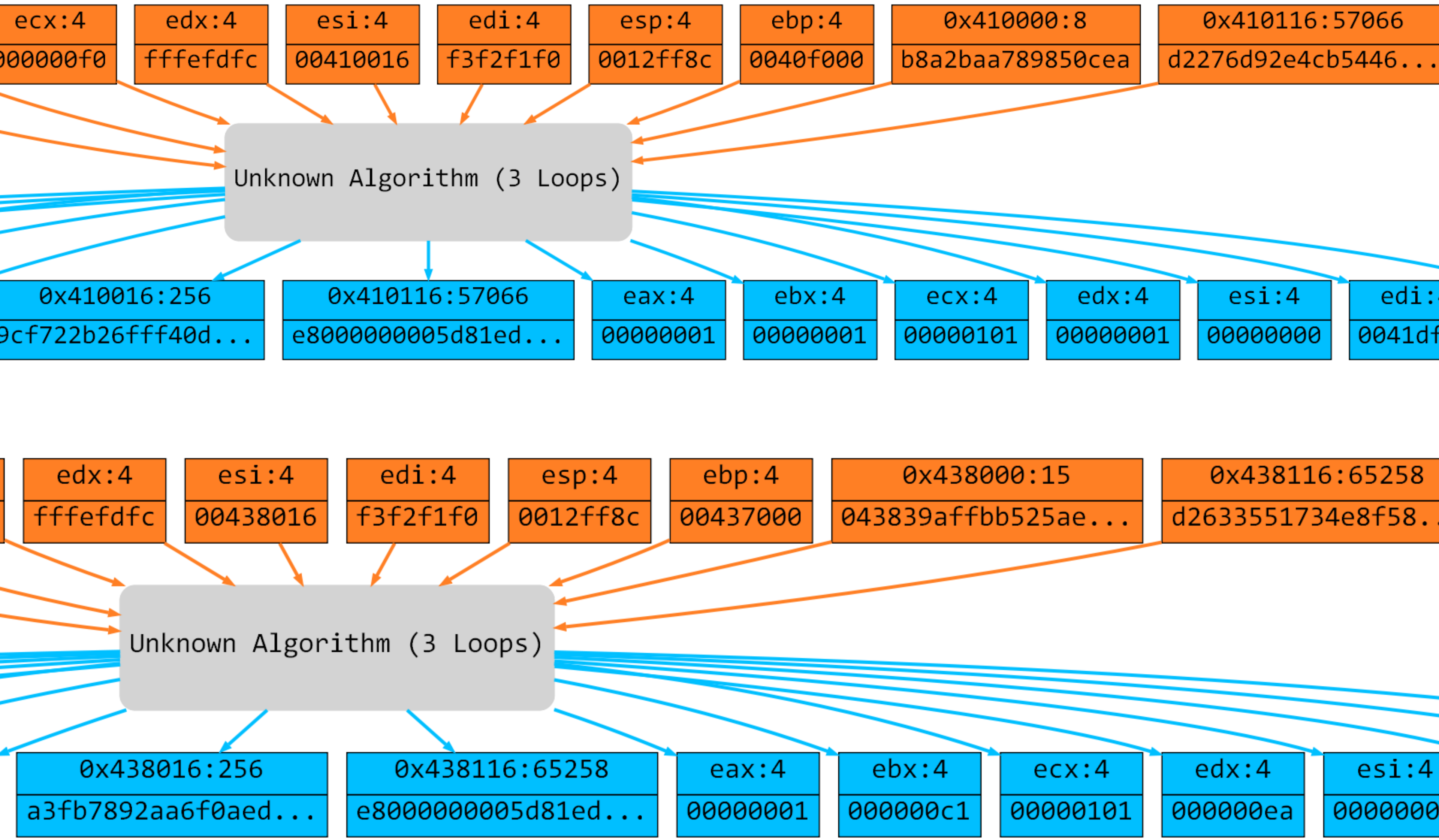


X86 ExecutableCode!

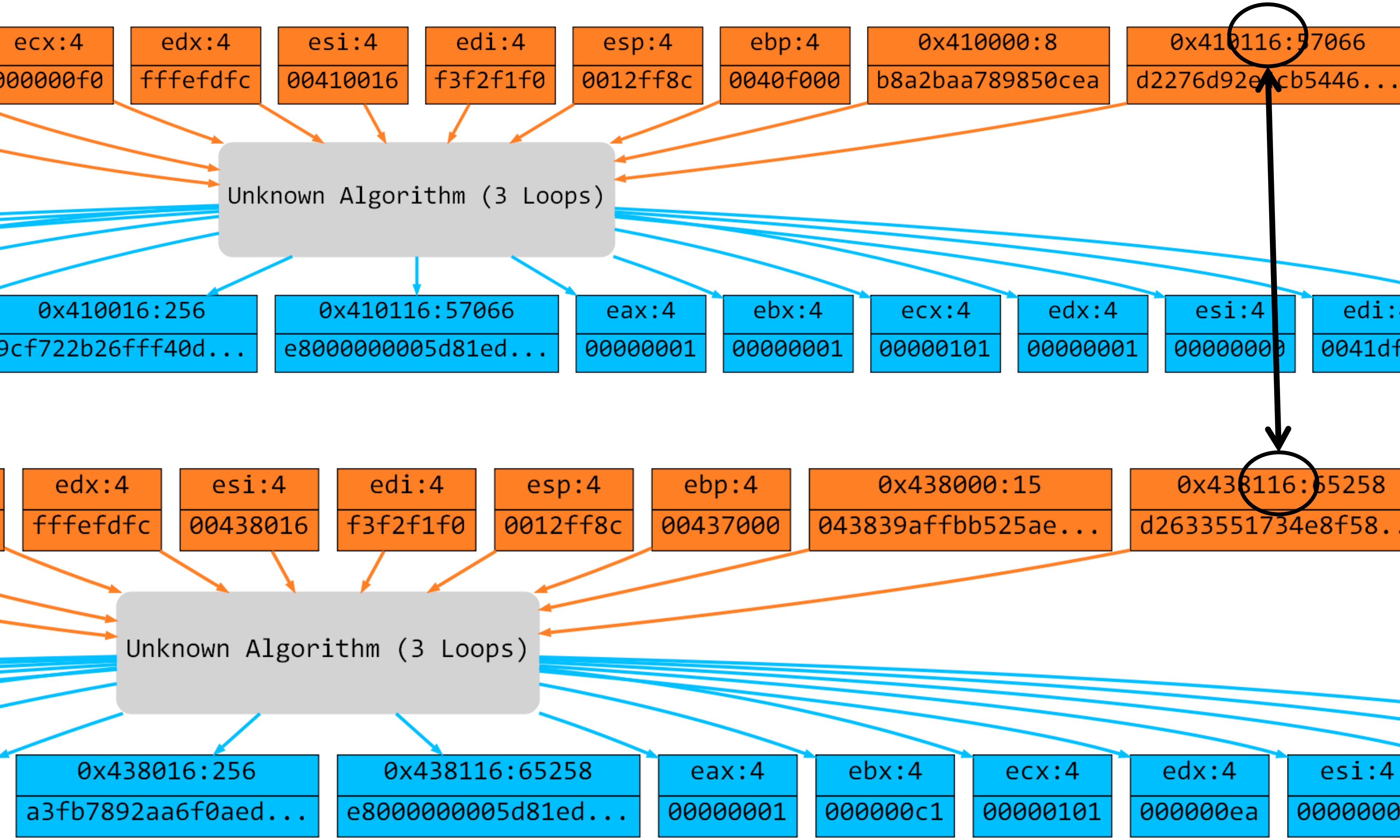

Unknown Algorithm
IDENTIFICATION
...

```
STARTED AT
2012-04-10 16:29:05.135000
** Crypto Algorithm Identification starting !**
5 input parameters
4 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for RC4...

** Found RC4 **

 ===> Key (8 bytes) : b8a2baa789850cea

 ===> Crypted text (57066 bytes) :  d2276d92e4cb5446...

 ===> Decrypted text (57066 bytes) :  e8000000005d81ed...
ENDED AT
2012-04-10 16:29:06.929000
```

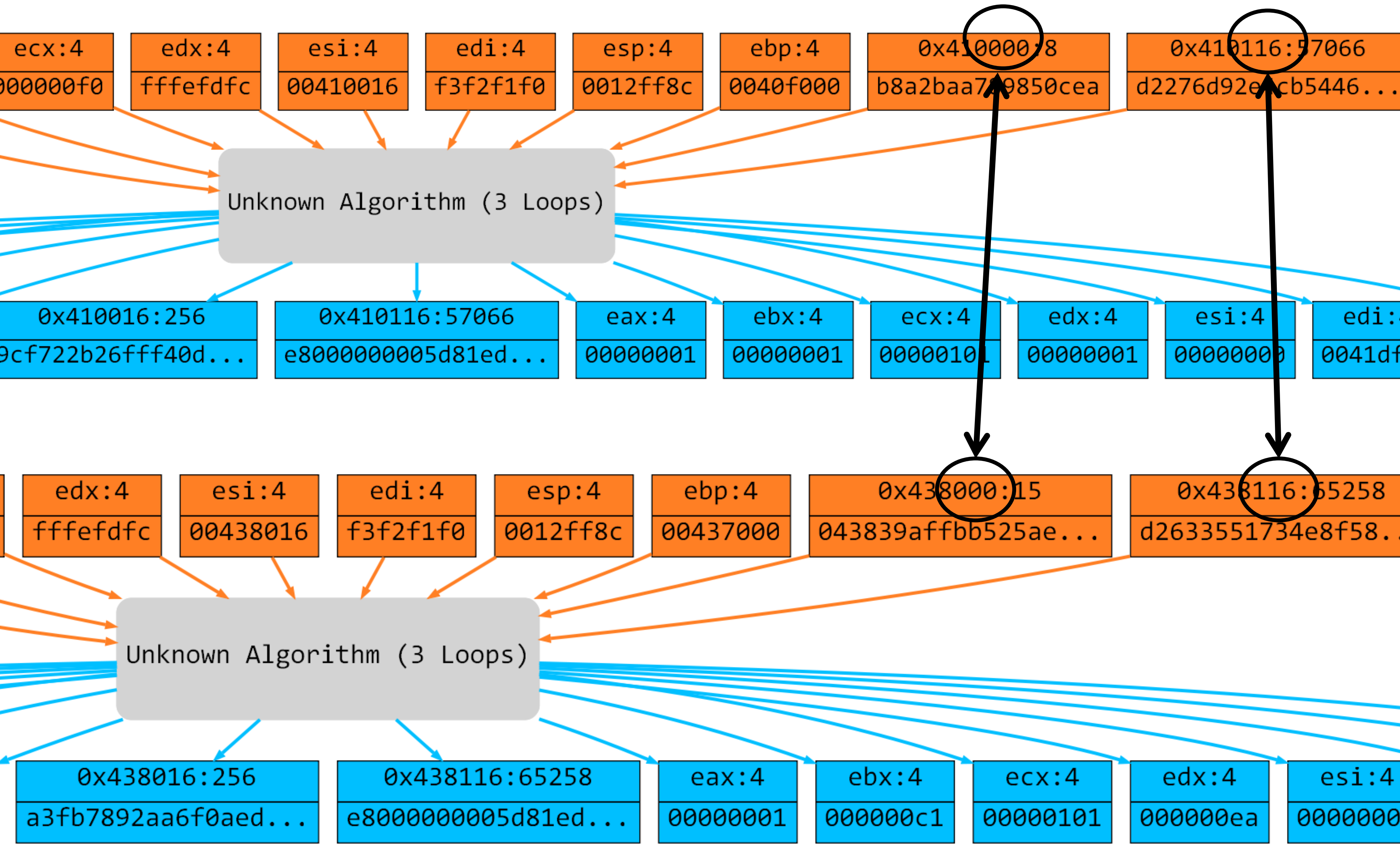
RC4 extracted from two Sality.AA binaries



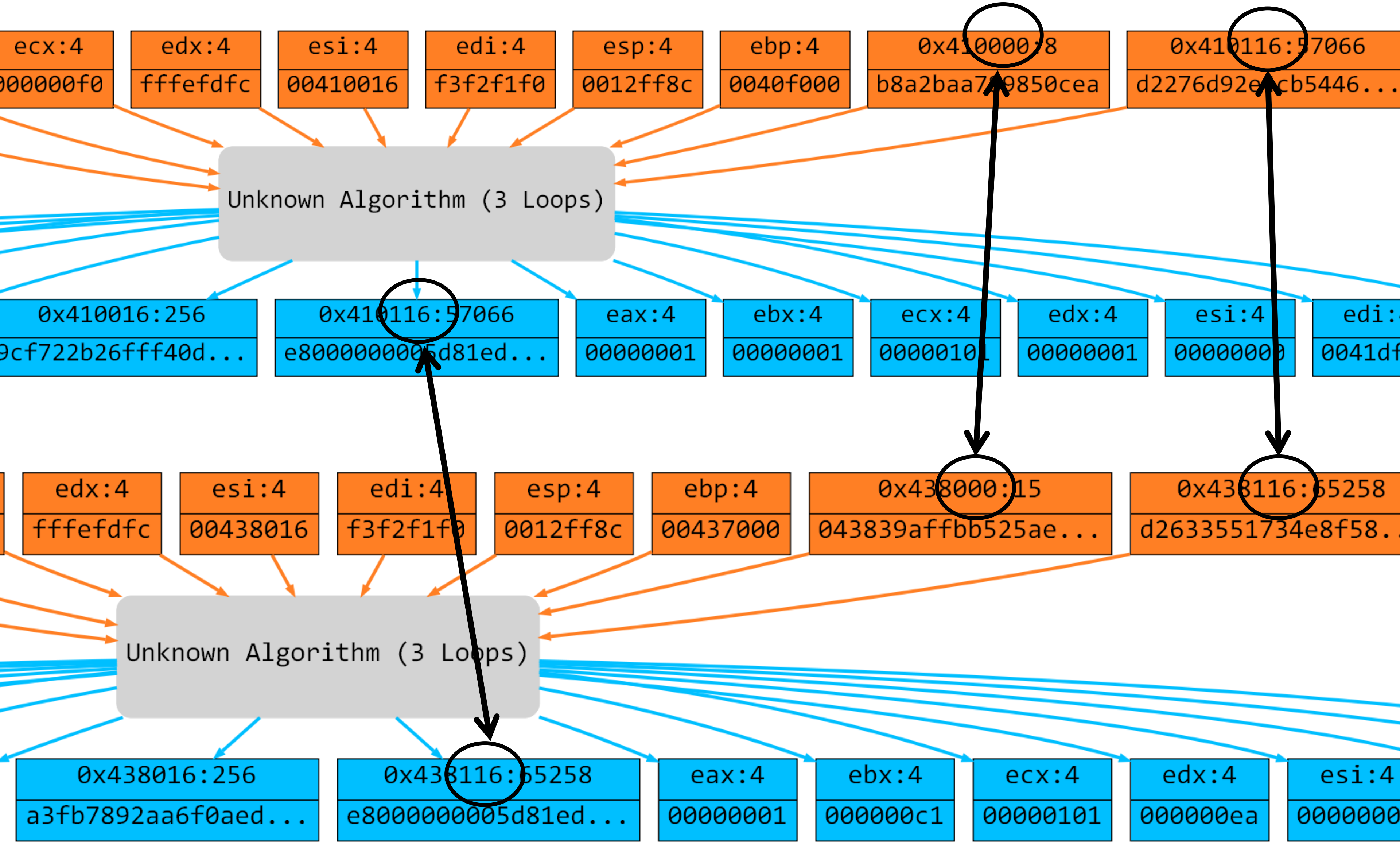
RC4 extracted from two Sality.AA binaries



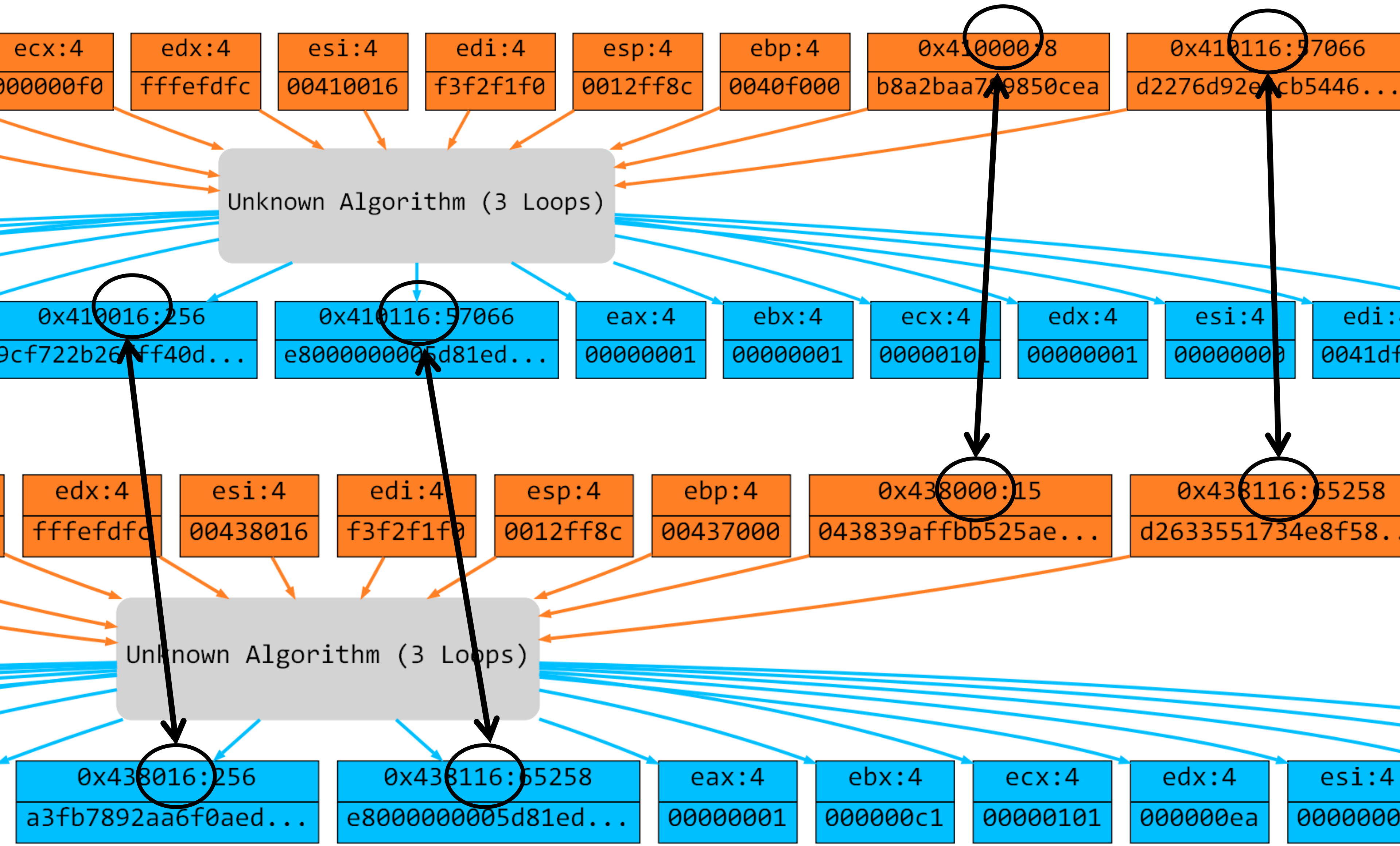
RC4 extracted from two Sality.AA binaries



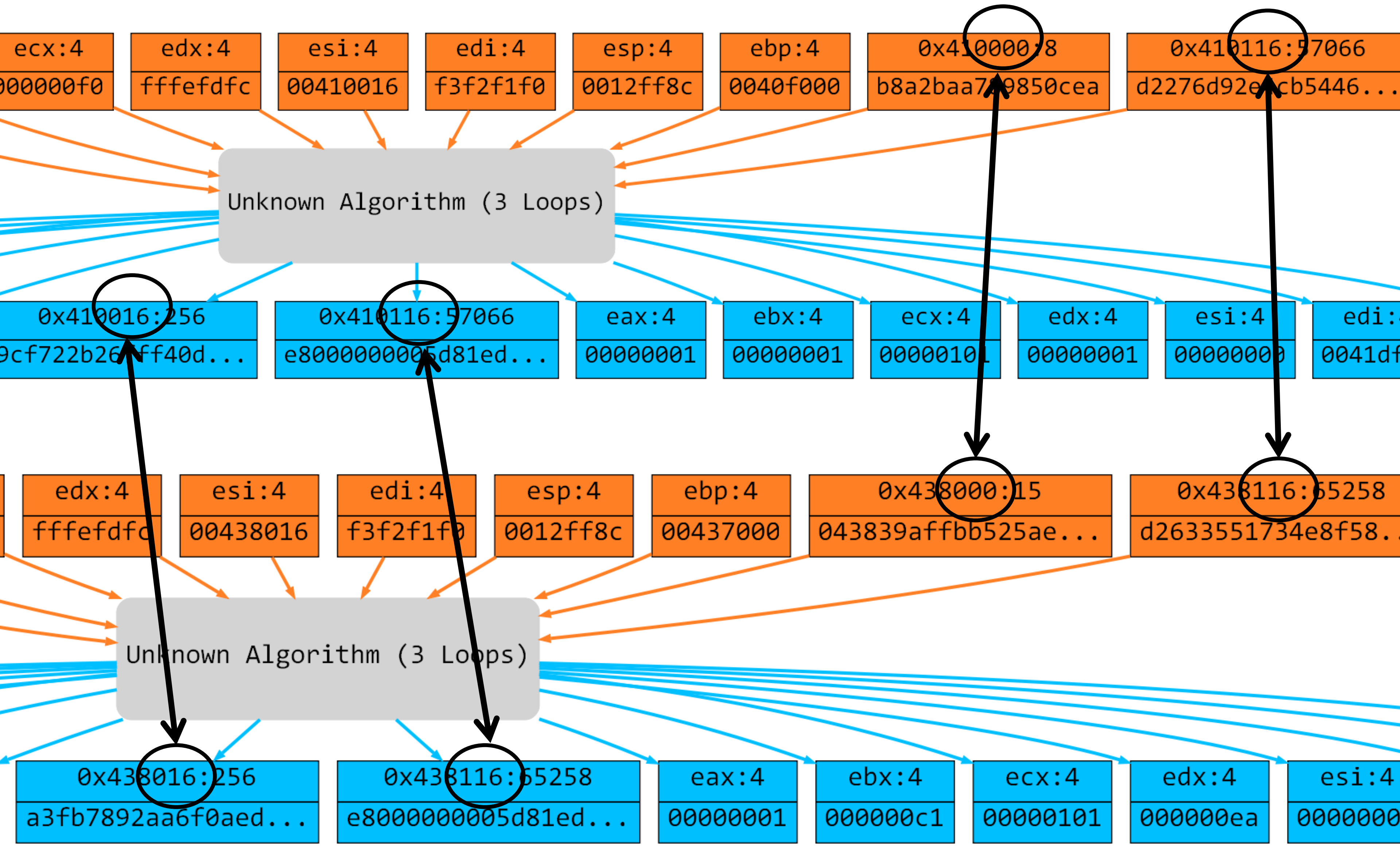
RC4 extracted from two Sality.AA binaries



RC4 extracted from two Sality.AA binaries



RC4 extracted from two Sality.AA binaries

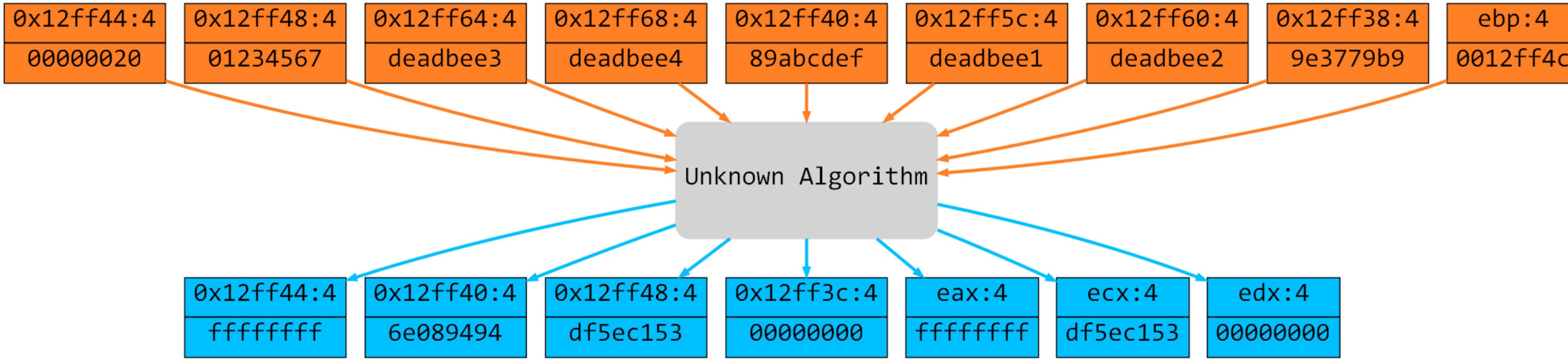


Crypto parameters always at the same offsets!

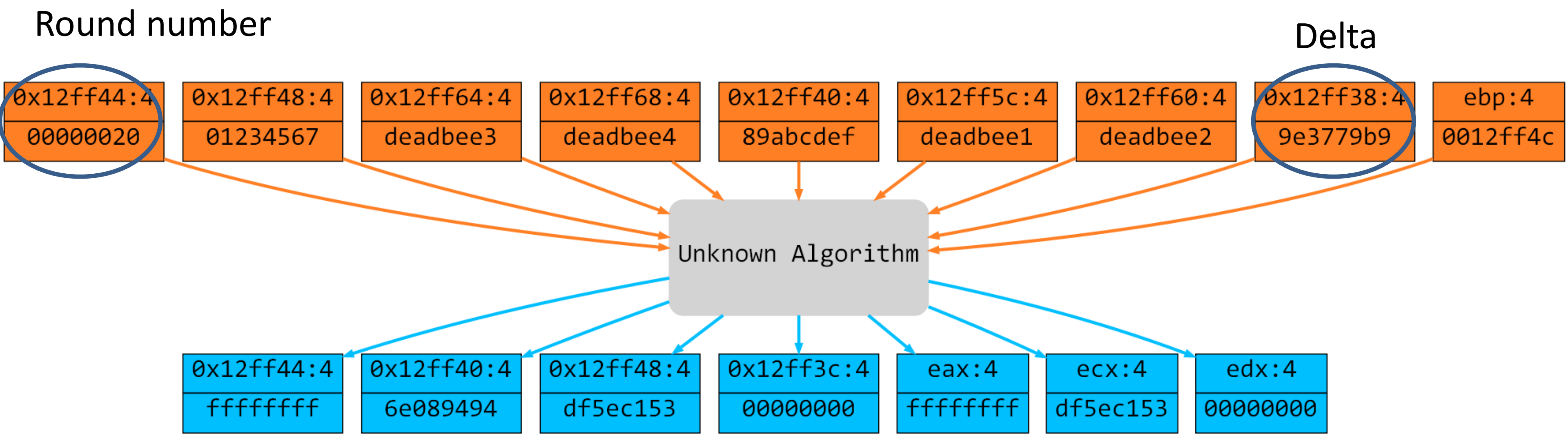
Modified TEA

MORE EXAMPLES!

Remember This ?



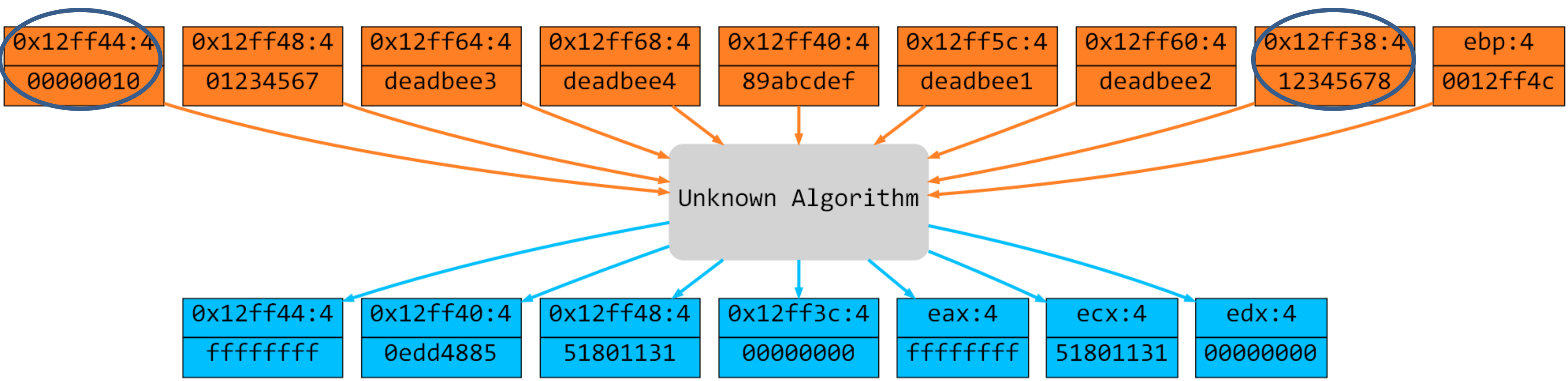
Remember This ?



The magic TEA constant (*delta*) and the round number are seen as input parameters, because they are initialized *before* the loop and used inside.

Modified TEA Implementation

- delta = 0x12345678 *(normally 0x9E3779B9)*
- round number = 16 *(normally 32)*



- TEA reference implementation extended:

```
def  tea (input_text, key):
    ...
```

- TEA reference implementation extended:

```
def  tea (input_text, key, delta, round_number):
    ...
```

- TEA reference implementation extended:

```
def  tea (input_text, key, delta, round_number):
    ...
```

```
STARTED AT
2012-04-08 13:16:27.918000
** Crypto Algorithm Identification starting !**
8 input parameters
3 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA *modified* decryption...
** Found *modified* TEA decryption **

 ===> Key (16 bytes) :   deadbee1deadbee2deadbee3deadbee4

 ===> Delta (4 bytes) :   12345678

 ===> Round number (4 bytes) :   00000010

 ===> Crypted text (8 bytes) :   0123456789abcdef

 ===> Decrypted text (8 bytes) : 51801131edd4885
ENDED AT
2012-04-08 13:21:11.562000
```

Example: Mozilla CTF

- Challenge “Awesome Corp. Secured Ranges”

- Binary program protected by PE Spin

- In the core binary, a strange algorithm...

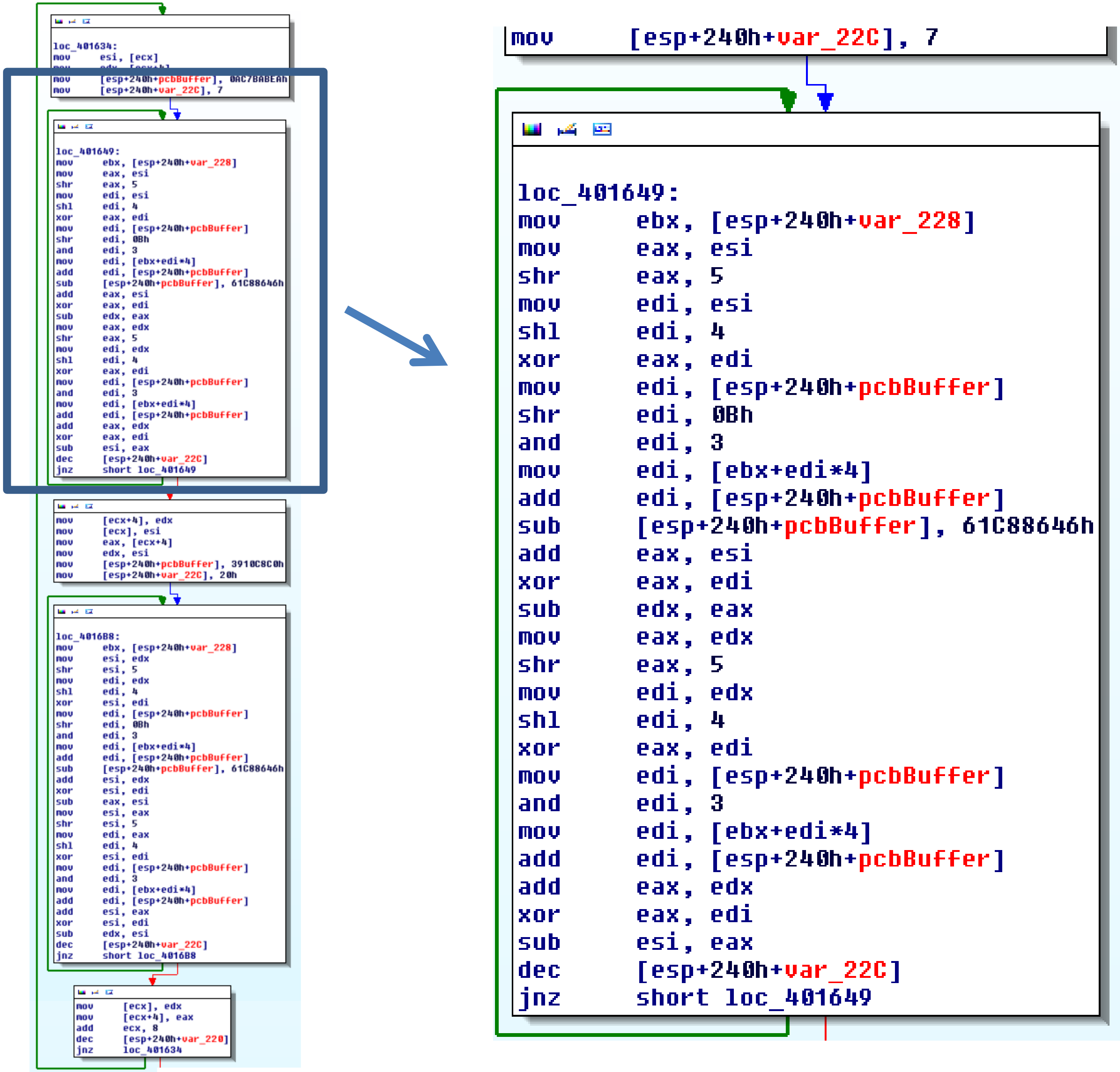
```
loc_401634:
mov     esi, [ecx]
mov     edx, [ecx+4]
mov     [esp+240h+pcbBuffer], 0AC7BABEAh
mov     [esp+240h+var_22C], 7
```

```
loc_401649:
mov     ebx, [esp+240h+var_228]
mov     eax, esi
shr     eax, 5
mov     edi, esi
shl     edi, 4
xor     eax, edi
mov     edi, [esp+240h+pcbBuffer]
shr     edi, 0Bh
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
sub     [esp+240h+pcbBuffer], 61C88646h
add     eax, esi
xor     eax, edi
sub     edx, eax
mov     eax, edx
shr     eax, 5
mov     edi, edx
shl     edi, 4
xor     eax, edi
mov     edi, [esp+240h+pcbBuffer]
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
add     eax, edx
xor     eax, edi
sub     esi, eax
dec     [esp+240h+var_22C]
jnz     short loc_401649
```

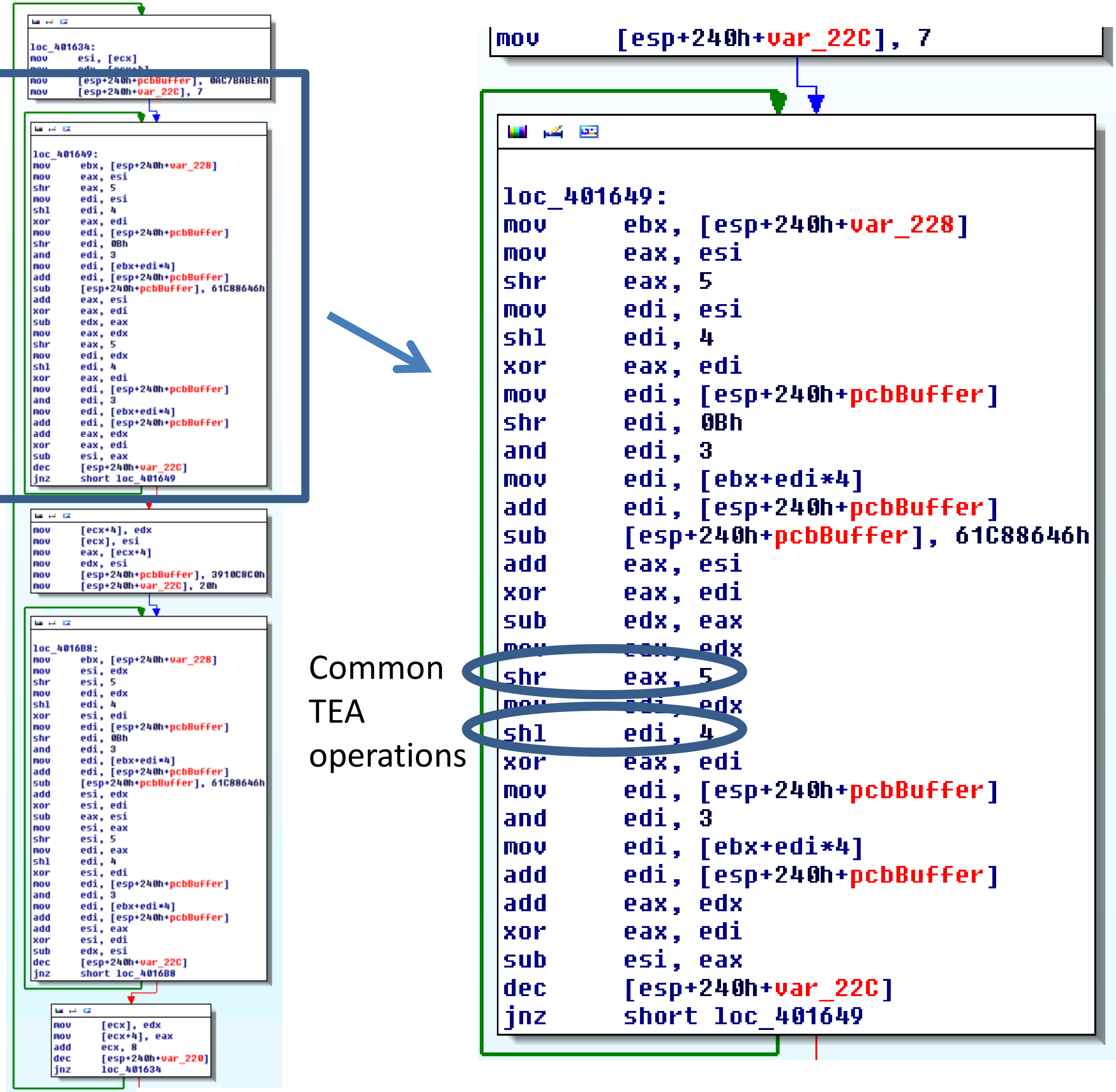
```
mov     [ecx+4], edx
mov     [ecx], esi
mov     eax, [ecx+4]
mov     edx, esi
mov     [esp+240h+pcbBuffer], 3910C8C0h
mov     [esp+240h+var_22C], 20h
```

```
loc_4016B8:
mov     ebx, [esp+240h+var_228]
mov     esi, edx
shr     esi, 5
mov     edi, edx
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
shr     edi, 0Bh
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
sub     [esp+240h+pcbBuffer], 61C88646h
add     esi, edx
xor     esi, edi
sub     eax, esi
mov     esi, eax
shr     esi, 5
mov     edi, eax
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
add     esi, eax
xor     esi, edi
sub     edx, esi
dec     [esp+240h+var_22C]
jnz     short loc_4016B8
```

```
mov     [ecx], edx
mov     [ecx+4], eax
add     ecx, 8
dec     [esp+240h+var_220]
jnz     loc_401634
```


mov [esp+240h+var_22C], 7
loc_401649:
mov ebx, [esp+240h+var_228]
mov eax, esi
shr eax, 5
mov edi, esi
shl edi, 4
xor eax, edi
mov edi, [esp+240h+pcbBuffer]
shr edi, 0Bh
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
sub [esp+240h+pcbBuffer], 61C88646h
add eax, esi
xor eax, edi
sub edx, eax
mov eax, edx
shr eax, 5
mov edi, edx
shl edi, 4
xor eax, edi
mov edi, [esp+240h+pcbBuffer]
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
add eax, edx
xor eax, edi
sub esi, eax
dec [esp+240h+var_22C]
jnz short loc_401649

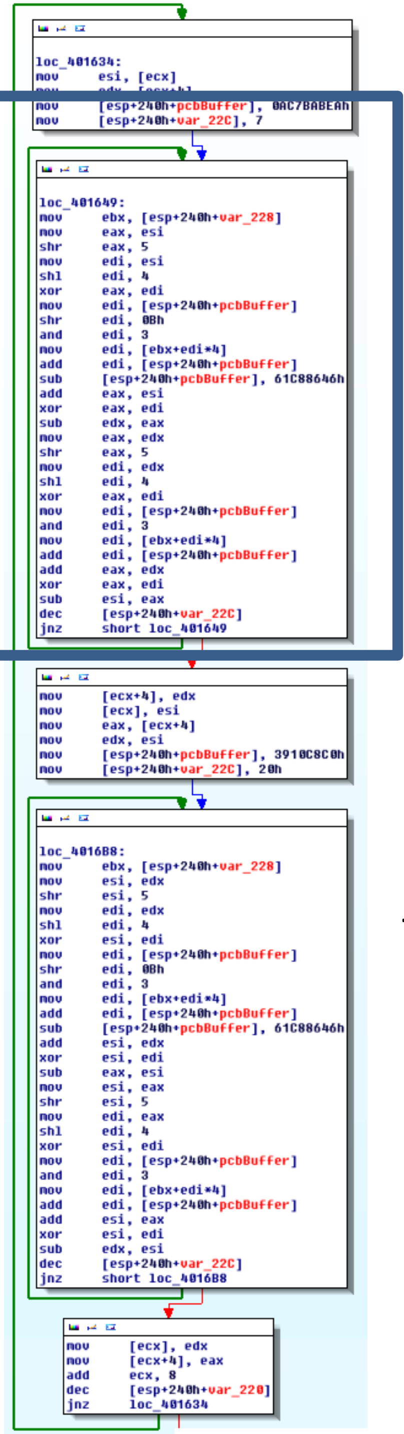
LOOP 1

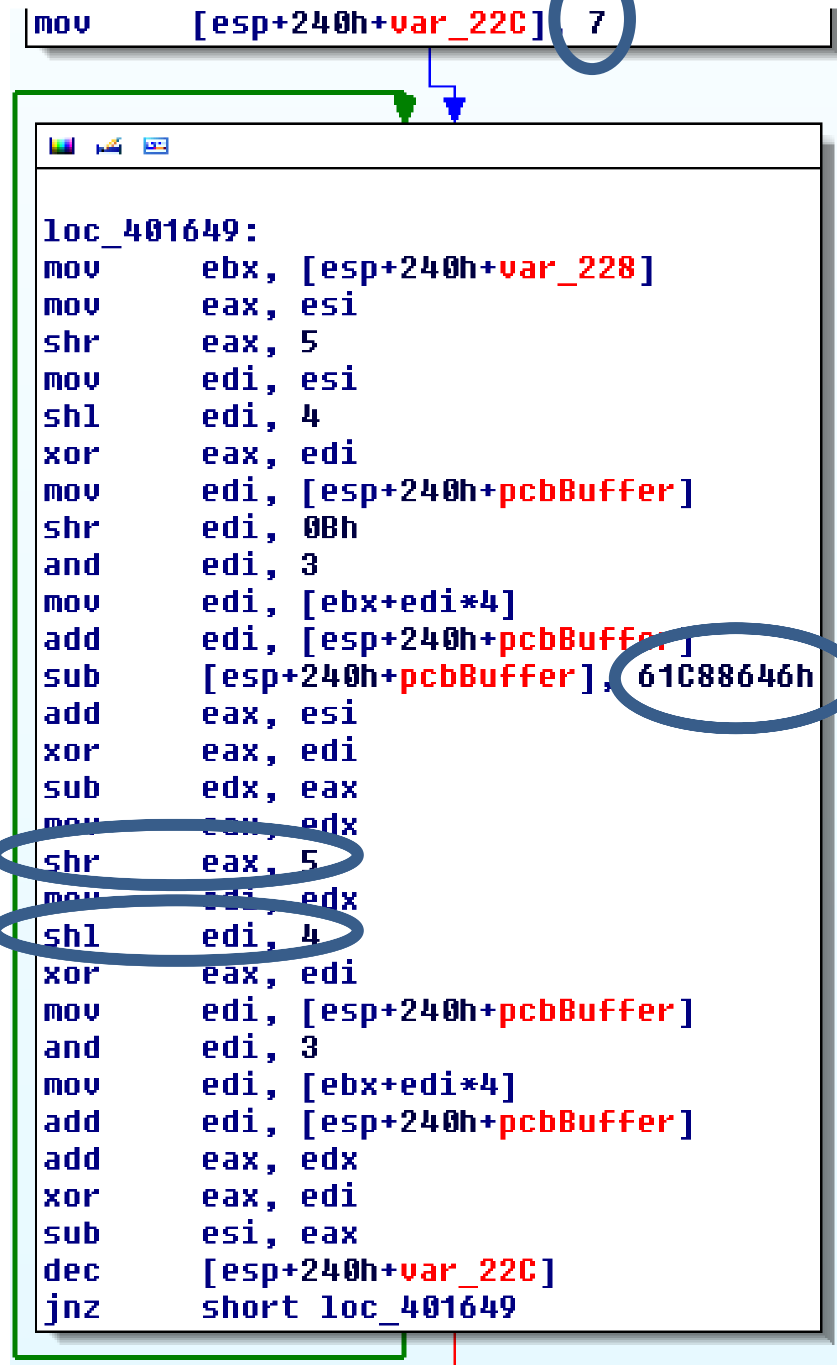


Common TEA operations

LOOP 1

Not the TEA round number

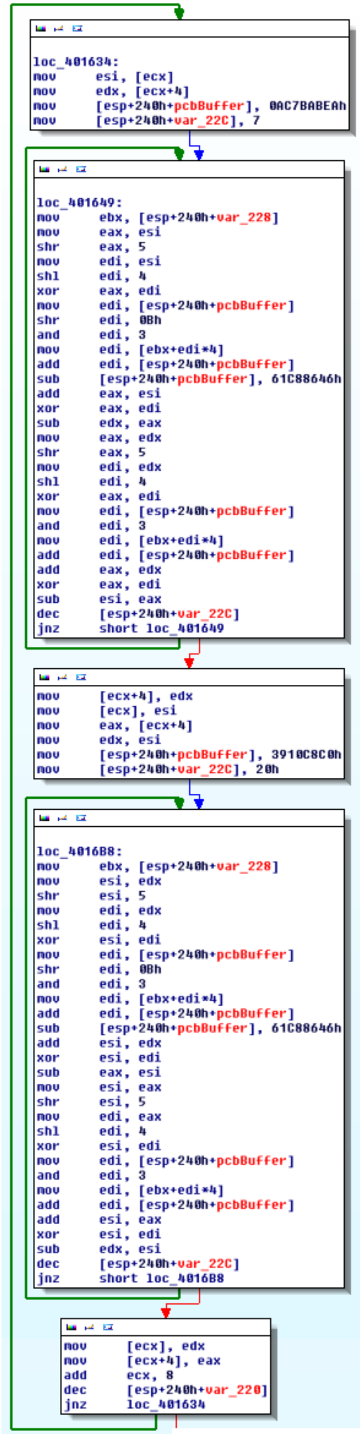




LOOP 1

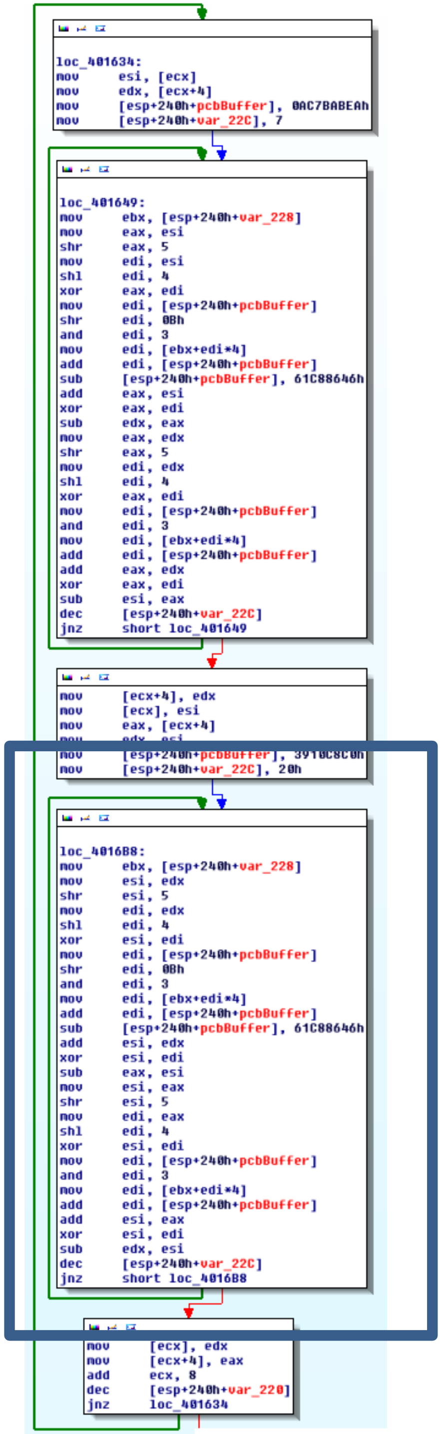
Not the TEA delta

Common TEA operations


loc_401634:
mov esi, [ecx]
mov edx, [ecx+4]
mov [esp+240h+pcbBuffer], 0AC7BABEAh
mov [esp+240h+var_22C], 7
loc_401649:
mov ebx, [esp+240h+var_228]
mov eax, esi
shr eax, 5
mov edi, esi
shl edi, 4
xor eax, edi
mov edi, [esp+240h+pcbBuffer]
shr edi, 0Bh
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
sub [esp+240h+pcbBuffer], 61C88646h
add eax, esi
xor eax, edi
sub edx, eax
mov eax, edx
shr eax, 5
mov edi, edx
shl edi, 4
xor eax, edi
mov edi, [esp+240h+pcbBuffer]
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
add eax, edx
xor eax, edi
sub esi, eax
dec [esp+240h+var_22C]
jnz short loc_401649
mov [ecx+4], edx
mov [ecx], esi
mov eax, [ecx+4]
mov edx, esi
mov [esp+240h+pcbBuffer], 3910C8C0h
mov [esp+240h+var_22C], 20h
loc_4016B8:
mov ebx, [esp+240h+var_228]
mov esi, edx
shr esi, 5
mov edi, edx
shl edi, 4
xor esi, edi
mov edi, [esp+240h+pcbBuffer]
shr edi, 0Bh
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
sub [esp+240h+pcbBuffer], 61C88646h
add esi, edx
xor esi, edi
sub eax, esi
mov esi, eax
shr esi, 5
mov edi, eax
shl edi, 4
xor esi, edi
mov edi, [esp+240h+pcbBuffer]
and edi, 3
mov edi, [ebx+edi*4]
add edi, [esp+240h+pcbBuffer]
add esi, eax
xor esi, edi
sub edx, esi
dec [esp+240h+var_22C]
jnz short loc_4016B8
mov [ecx], edx
mov [ecx+4], eax
add ecx, 8
dec [esp+240h+var_220]
jnz loc_401634

```
mov     [esp+240h+var_22C], 20h

loc_4016B8:
mov     ebx, [esp+240h+var_228]
mov     esi, edx
shr     esi, 5
mov     edi, edx
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
shr     edi, 0Bh
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
sub     [esp+240h+pcbBuffer], 61C88646h
add     esi, edx
xor     esi, edi
sub     eax, esi
mov     esi, eax
shr     esi, 5
mov     edi, eax
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
add     esi, eax
xor     esi, edi
sub     edx, esi
dec     [esp+240h+var_22C]
jnz     short loc_4016B8
```

LOOP 2



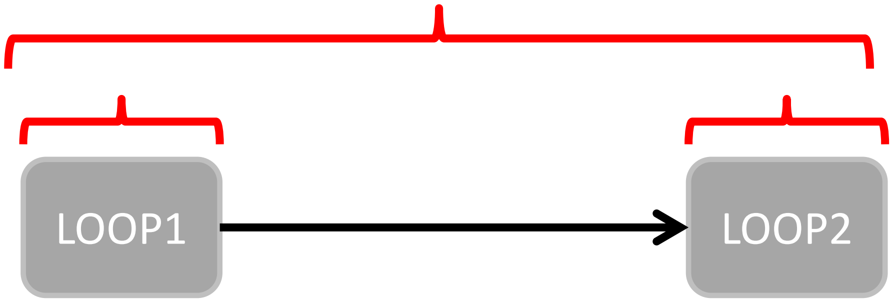

```
mov     [esp+240h+var_22C], 20h
```

TEA
round number!

```
loc_4016B8:
mov     ebx, [esp+240h+var_228]
mov     esi, edx
shr     esi, 5
mov     edi, edx
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
shr     edi, 0Bh
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
sub     [esp+240h+pcbBuffer], 61C88646h
add     esi, edx
xor     esi, edi
sub     eax, esi
mov     esi, eax
shr     esi, 5
mov     edi, eax
shl     edi, 4
xor     esi, edi
mov     edi, [esp+240h+pcbBuffer]
and     edi, 3
mov     edi, [ebx+edi*4]
add     edi, [esp+240h+pcbBuffer]
add     esi, eax
xor     esi, edi
sub     edx, esi
dec     [esp+240h+var_22C]
jnz     short loc_4016B8
```

LOOP 2

Loop Data Flow Graph



3 possible cryptographic algorithms

```
STARTED AT
2012-06-01 17:55:57.729000
** Crypto Algorithm Identification starting !**  8 inputs -  5 outputs
Comparison phase starting... Test for XTEA *modified* decryption... Unknown Algo
rithm stays unkwown!
** Crypto Algorithm Identification starting !**  6 inputs -  5 outputs
Comparison phase starting... Test for XTEA *modified* decryption...
** Found XTEA *extended* decryption **

 ===> Key (16 bytes) : 0x09c195dbb17945f5503a3b2bc61fa033

 ===> Delta (4 bytes) : 0x61c88646

 ===> Round number (4 bytes) : 0x00000007

 ===> Crypted text (8 bytes) : 0x2adc69d7832172aa

 ===> Decrypted text (8 bytes) : 0x75d081d699f152ae
** Crypto Algorithm Identification starting !**  6 inputs -  5 outputs
Comparison phase starting... Test for XTEA *modified* decryption...
** Found XTEA *extended* decryption **

 ===> Key (16 bytes) : 0x09c195dbb17945f5503a3b2bc61fa033

 ===> Delta (4 bytes) : 0x61c88646

 ===> Round number (4 bytes) : 0x00000020

 ===> Crypted text (8 bytes) : 0x75d081d699f152ae

 ===> Decrypted text (8 bytes) : 0x911b822660cfe1f1
ENDED AT
2012-06-01 17:57:43.659000
```

LOOP1 +
LOOP2

LOOP1

LOOP2

Method Recap

1. We collect an execution trace.

2. We extract possible cryptographic algorithms with their parameter values.

3. We compare the input-output relationship with known algorithms.

We prove that a program behaves like a known crypto algorithm during one particular execution path.

Conclusion (1)

- Interesting alternative to pure syntactic-identification for crypto algorithms:
  - Resistance against usual obfuscation techniques.
  - Gives the exact parameters.

- As any dynamic technique, you have to know how to exhibit interesting execution paths.

- It is easy to bypass, like any program analysis technique ☹

Conclusion (2)

- The identification process itself is generic:
  - Collect the execution trace
  - Extract the type of code you are looking for (here is the magic)
  - Get I/O values
  - Compare with reference implementations

Conclusion (3)

- Nice work: Felix Gröbert "**Automatic Identification of Cryptographic Primitives in Software**", 27th CCC

http://code.google.com/p/kerckhoffs/

What's Next ? (1)

- That's only the beginning! Just wanted to show that it is feasible and useful.
- What about more complex algorithms ? What about hash functions ? Compression algorithms ?
- What about proprietary algorithms ?

What's Next ? (2)

- Make a real tool. This one is just a PoC.
- How to use the analyst knowledge ? In practice **he often knows where the crypto is**, analyzing a complete execution trace is more an academic hobby.

Thank you for your attention ;-)

j04n.calvet@gmail.com