Cryptographic Function Identification in Obfuscated Binary Programs

Hackito Ergo Sum 2012

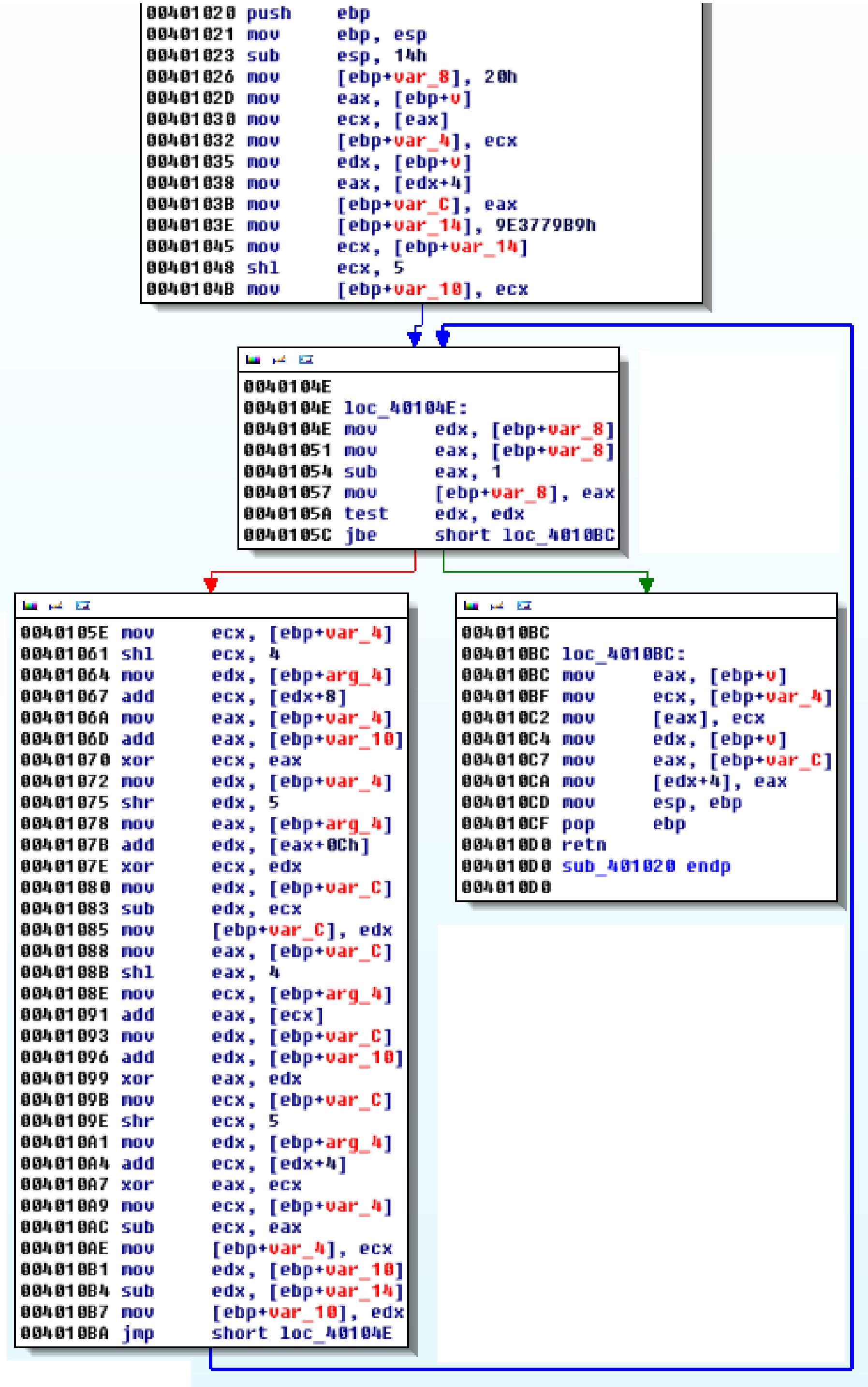
Joan Calvet – j04n.calvet@gmail.com

Presentation Outline

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

INTRODUCTION TO THE PROBLEM

What’s this ?



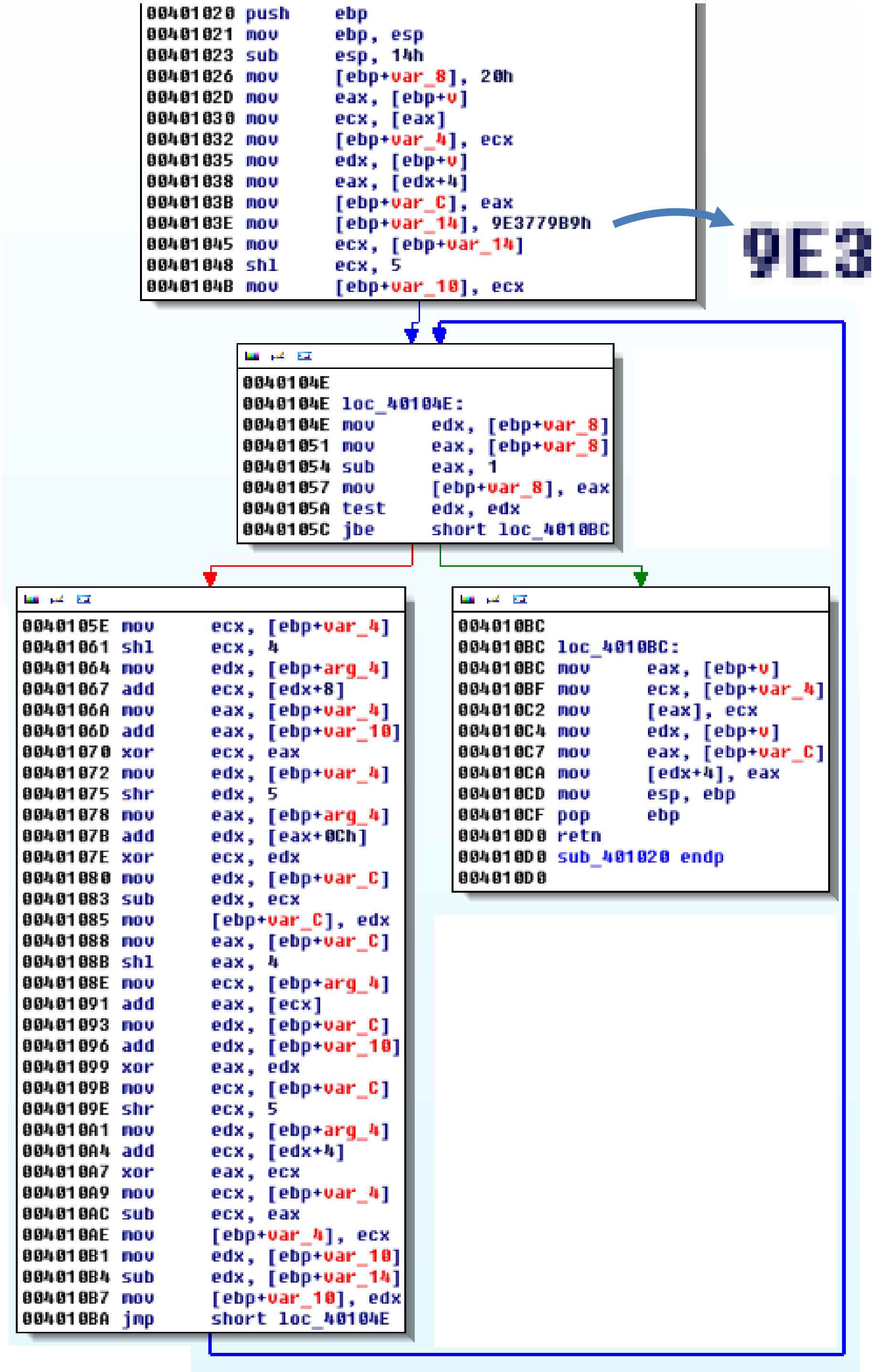
```
00401020 push    ebp
00401021 mov     ebp, esp
00401023 sub     esp, 14h
00401026 mov     [ebp+var_8], 20h
0040102D mov     eax, [ebp+v]
00401030 mov     ecx, [eax]
00401032 mov     [ebp+var_4], ecx
00401035 mov     edx, [ebp+v]
00401038 mov     eax, [edx+4]
0040103B mov     [ebp+var_C], eax
0040103E mov     [ebp+var_14], 9E3779B9h
00401045 mov     ecx, [ebp+var_14]
00401048 shl     ecx, 5
0040104B mov     [ebp+var_10], ecx
```

```
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 ?



```
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
```

9E3779B9h

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> **...**

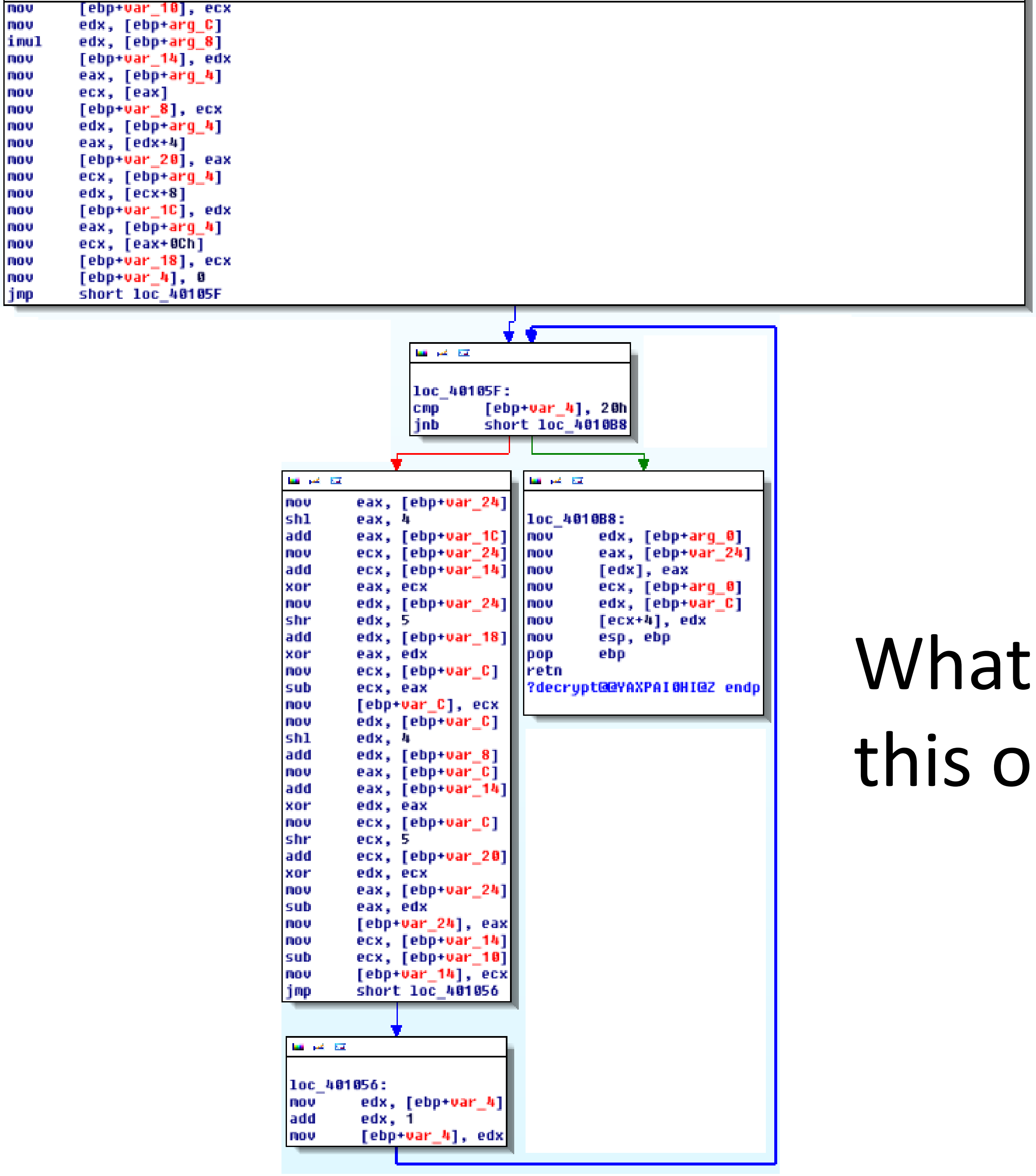
TEA Encryption Algorithm, Source code
www.shokhirev.com/nikolai/.../uTeaSet_pas.html - Traduire cette page
... XTeaEncrypt/XTeaDecrypt Thanks to Pedro Gimeno Fortea <parigalo@formauri.es> } interface const Delta: longword = $**9e3779b9**; type TLong2 = array[0.

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?



```
mov     [ebp+var_10], ecx
mov     edx, [ebp+arg_C]
imul    edx, [ebp+arg_8]
mov     [ebp+var_14], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax]
mov     [ebp+var_8], ecx
mov     edx, [ebp+arg_4]
mov     eax, [edx+4]
mov     [ebp+var_20], eax
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+8]
mov     [ebp+var_1C], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+0Ch]
mov     [ebp+var_18], ecx
mov     [ebp+var_4], 0
jmp     short loc_40105F
```

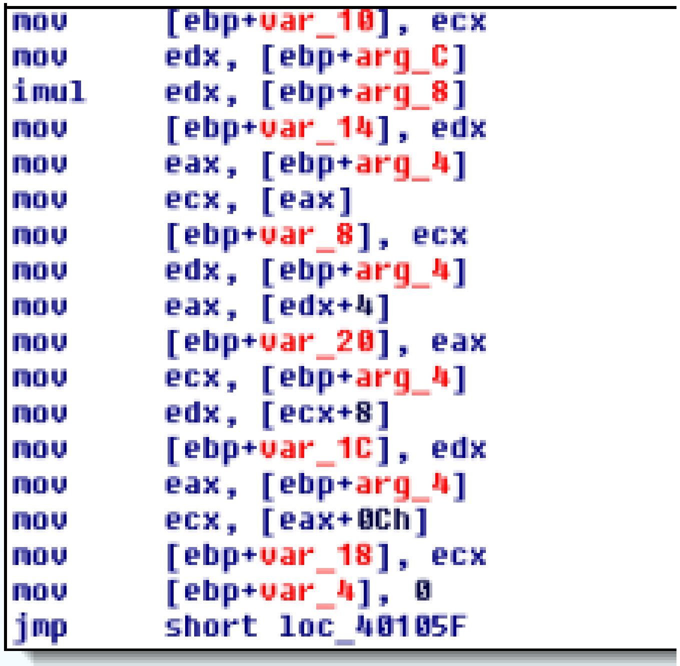
```
loc_40105F:
cmp     [ebp+var_4], 20h
jnb     short loc_4010B8
```

```
mov     eax, [ebp+var_24]
shl     eax, 4
add     eax, [ebp+var_1C]
mov     ecx, [ebp+var_24]
add     ecx, [ebp+var_14]
xor     eax, ecx
mov     edx, [ebp+var_24]
shr     edx, 5
add     edx, [ebp+var_18]
xor     eax, edx
mov     ecx, [ebp+var_C]
sub     ecx, eax
mov     [ebp+var_C], ecx
mov     edx, [ebp+var_C]
shl     edx, 4
add     edx, [ebp+var_8]
mov     eax, [ebp+var_C]
add     eax, [ebp+var_14]
xor     edx, eax
mov     ecx, [ebp+var_C]
shr     ecx, 5
add     ecx, [ebp+var_20]
xor     edx, ecx
mov     eax, [ebp+var_24]
sub     eax, edx
mov     [ebp+var_24], eax
mov     ecx, [ebp+var_14]
sub     ecx, [ebp+var_10]
mov     [ebp+var_14], ecx
jmp     short loc_401056
```

```
loc_4010B8:
mov     edx, [ebp+arg_0]
mov     eax, [ebp+var_24]
mov     [edx], eax
mov     ecx, [ebp+arg_0]
mov     edx, [ebp+var_C]
mov     [ecx+4], edx
mov     esp, ebp
pop     ebp
retn
?decrypt@@YAXPAI0HI@Z endp
```

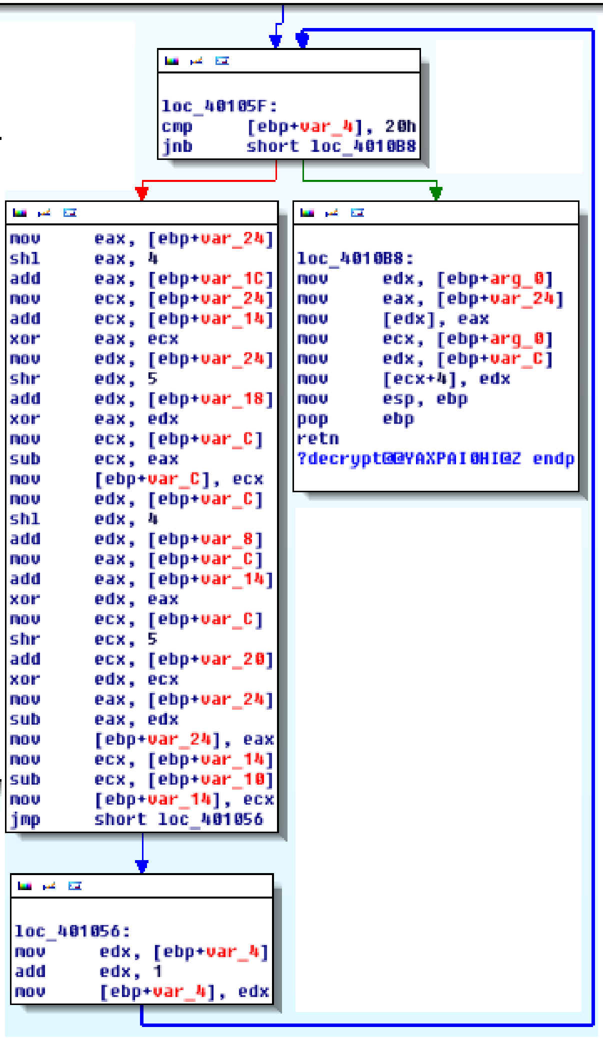
```
loc_401056:
mov     edx, [ebp+var_4]
add     edx, 1
mov     [ebp+var_4], edx
```

```
mov     [ebp+var_10], ecx
mov     edx, [ebp+arg_C]
imul    edx, [ebp+arg_8]
mov     [ebp+var_14], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax]
mov     [ebp+var_8], ecx
mov     edx, [ebp+arg_4]
mov     eax, [edx+4]
mov     [ebp+var_20], eax
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+8]
mov     [ebp+var_1C], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+0Ch]
mov     [ebp+var_18], ecx
mov     [ebp+var_4], 0
jmp     short loc_40105F
```



```
mov     [ebp+var_10], ecx
mov     edx, [ebp+arg_C]
imul    edx, [ebp+arg_8]
mov     [ebp+var_14], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax]
mov     [ebp+var_8], ecx
mov     edx, [ebp+arg_4]
mov     eax, [edx+4]
mov     [ebp+var_20], eax
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+8]
mov     [ebp+var_1C], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+0Ch]
mov     [ebp+var_18], ecx
mov     [ebp+var_4], 0
jmp     short loc_40105F
```

No particular constants



```
loc_40105F:
cmp     [ebp+var_4], 20h
jnb     short loc_4010B8
```

```
mov     eax, [ebp+var_24]
shl     eax, 4
add     eax, [ebp+var_1C]
mov     ecx, [ebp+var_24]
add     ecx, [ebp+var_14]
xor     eax, ecx
mov     edx, [ebp+var_24]
shr     edx, 5
add     edx, [ebp+var_18]
xor     eax, edx
mov     ecx, [ebp+var_C]
sub     ecx, eax
mov     [ebp+var_C], ecx
mov     edx, [ebp+var_C]
shl     edx, 4
add     edx, [ebp+var_8]
mov     eax, [ebp+var_C]
add     eax, [ebp+var_14]
xor     edx, eax
mov     ecx, [ebp+var_C]
shr     ecx, 5
add     ecx, [ebp+var_20]
xor     edx, ecx
mov     eax, [ebp+var_24]
sub     eax, edx
mov     [ebp+var_24], eax
mov     ecx, [ebp+var_14]
sub     ecx, [ebp+var_10]
mov     [ebp+var_14], ecx
jmp     short loc_401056
```

```
loc_4010B8:
mov     edx, [ebp+arg_0]
mov     eax, [ebp+var_24]
mov     [edx], eax
mov     ecx, [ebp+arg_0]
mov     edx, [ebp+var_C]
mov     [ecx+4], edx
mov     esp, ebp
pop     ebp
retn
?decrypt@@YAXPAI0HI@Z endp
```

```
loc_401056:
mov     edx, [ebp+var_4]
add     edx, 1
mov     [ebp+var_4], edx
```

What about this one?

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 ?

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 I/O relationship ?

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 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.

- We can not *realistically* use the I/O relationship to prove that **P** implements a particular crypto algorithm *on any inputs*.

(too many input states to test!)

How To Use Input-Output Relationship ?

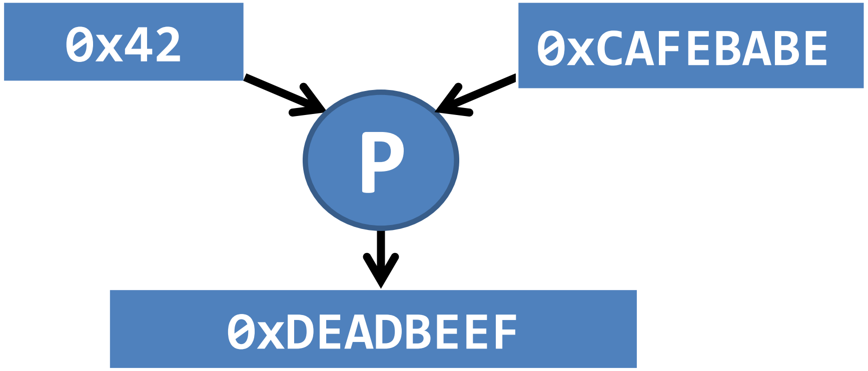
- Let’s say **P** is a program implementing an unknown cryptographic algorithm.

- We can not *realistically* use the I/O relationship to prove that **P** implements a particular crypto algorithm *on any inputs*.

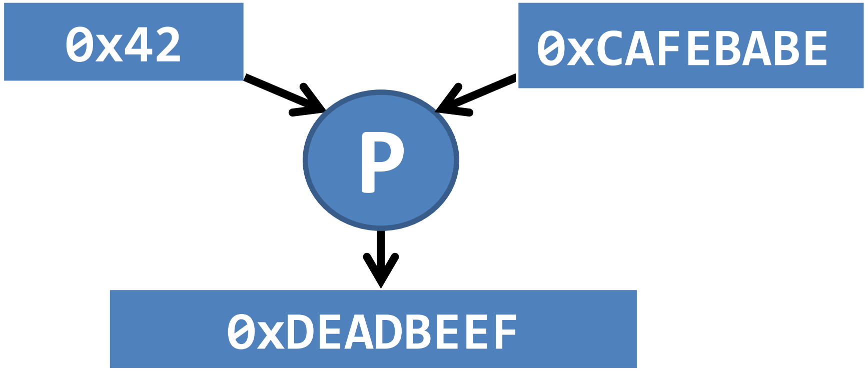
  (too many input states to test!)

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

For example:



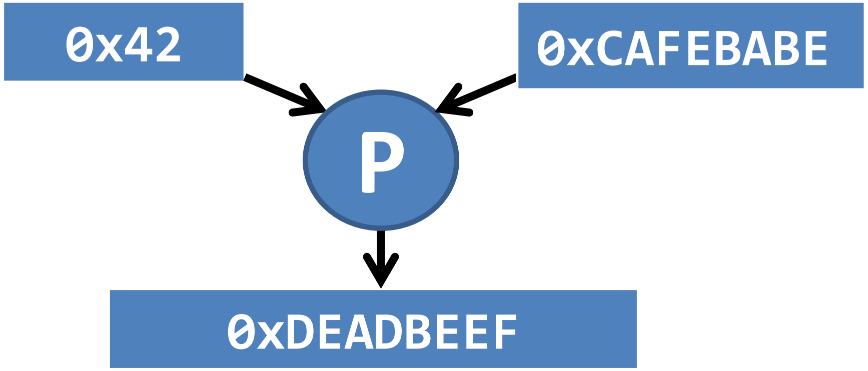
For example:



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

What does it mean ?

For example:


0x42

0xCAFEBABE

P

0xDEADBEEF

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

What does it mean ?

- It proves that **P** implements **TEA** on these particular input values.

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.

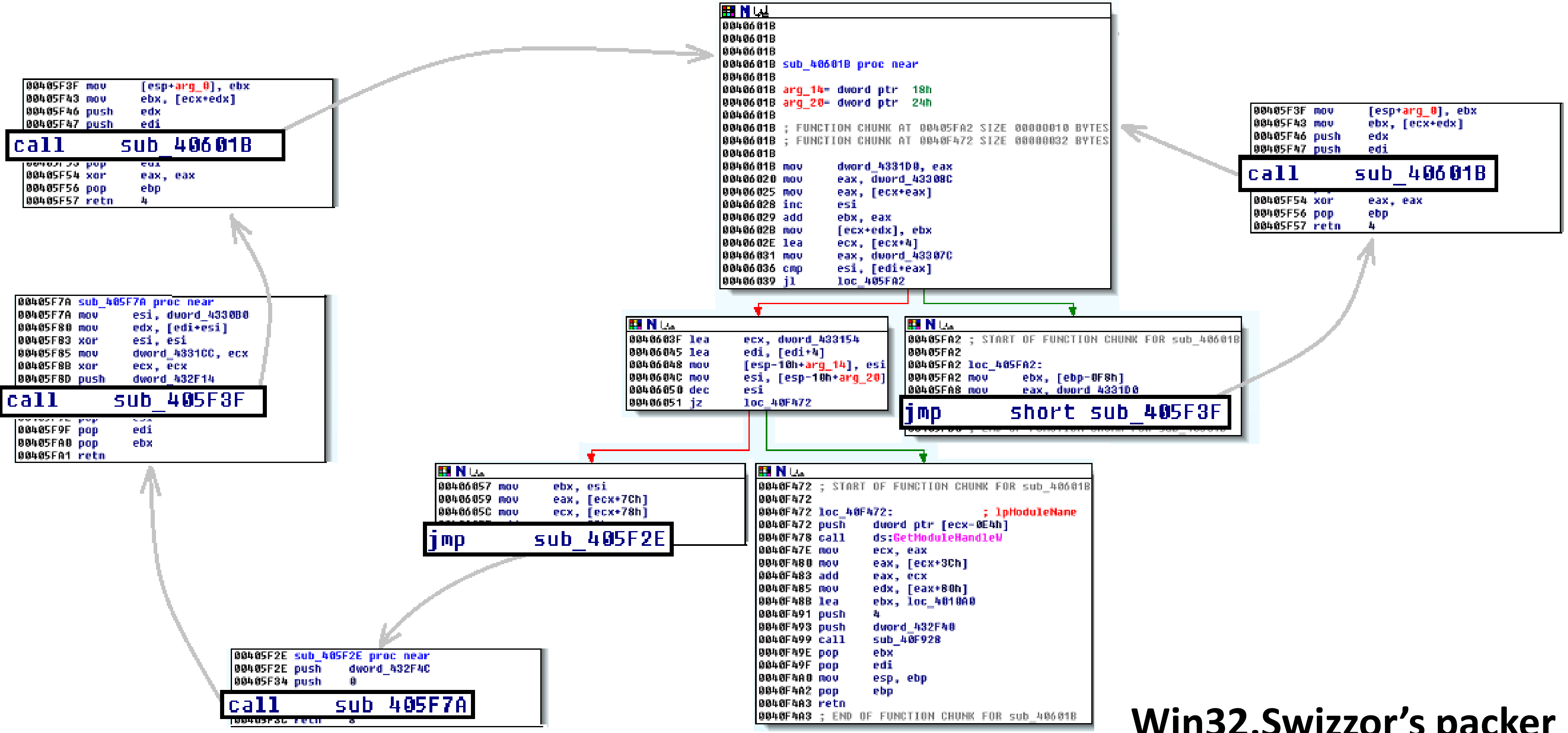
Address	Instruction	Read Registers	Written Registers	Read Memory	Written Memory
4012b3	push ebp	ebp *0012de28* esp *0012bd98*	esp *0012bd94*		12bd94 *0012de28*
4012b4	mov ebp, esp	esp *0012bd94*	ebp *0012bd94*		
4012b6	push ebx	ebx *02f00010* esp *0012bd84*	esp *0012bd80*		12bd80 *2f00010*
…					

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

In obfuscated programs, such things can happen:



```
00405F3F mov     [esp+arg_0], ebx
00405F43 mov     ebx, [ecx+edx]
00405F46 push    edx
00405F47 push    edi
call        sub_40601B
00405F54 xor     eax, eax
00405F56 pop     ebp
00405F57 retn    4
```

```
0040601B
0040601B
0040601B
0040601B sub_40601B proc near
0040601B
0040601B arg_14= dword ptr  18h
0040601B arg_20= dword ptr  24h
0040601B
0040601B ; FUNCTION CHUNK AT 00405FA2 SIZE 00000010 BYTES
0040601B ; FUNCTION CHUNK AT 0040F472 SIZE 00000032 BYTES
0040601B
0040601B mov     dword_4331D0, eax
00406020 mov     eax, dword_43308C
00406025 mov     eax, [ecx+eax]
00406028 inc     esi
00406029 add     ebx, eax
0040602B mov     [ecx+edx], ebx
0040602E lea     ecx, [ecx+4]
00406031 mov     eax, dword_43307C
00406036 cmp     esi, [edi+eax]
00406039 jl      loc_405FA2
```

```
00405F3F mov     [esp+arg_0], ebx
00405F43 mov     ebx, [ecx+edx]
00405F46 push    edx
00405F47 push    edi
call        sub_40601B
00405F54 xor     eax, eax
00405F56 pop     ebp
00405F57 retn    4
```

```
00405F7A sub_405F7A proc near
00405F7A mov     esi, dword_4330B0
00405F80 mov     edx, [edi+esi]
00405F83 xor     esi, esi
00405F85 mov     dword_4331CC, ecx
00405F8B xor     ecx, ecx
00405F8D push    dword_432F14
call        sub_405F3F
00405F9F pop     edi
00405FA0 pop     ebx
00405FA1 retn
```

```
0040603F lea     ecx, dword_433154
00406045 lea     edi, [edi+4]
00406048 mov     [esp-10h+arg_14], esi
0040604C mov     esi, [esp-10h+arg_20]
00406050 dec     esi
00406051 jz      loc_40F472
```

```
00405FA2 ; START OF FUNCTION CHUNK FOR sub_40601B
00405FA2
00405FA2 loc_405FA2:
00405FA2 mov     ebx, [ebp-0F8h]
00405FA8 mov     eax, dword 4331D0
jmp         short sub_405F3F
```

```
00406057 mov     ebx, esi
00406059 mov     eax, [ecx+7Ch]
0040605C mov     ecx, [ecx+78h]
jmp         sub_405F2E
```

```
0040F472 ; START OF FUNCTION CHUNK FOR sub_40601B
0040F472
0040F472 loc_40F472:                 ; lpModuleName
0040F472 push    dword ptr [ecx-0E4h]
0040F478 call    ds:GetModuleHandleW
0040F47E mov     ecx, eax
0040F480 mov     eax, [ecx+3Ch]
0040F483 add     eax, ecx
0040F485 mov     edx, [eax+80h]
0040F48B lea     ebx, loc_4010A0
0040F491 push    4
0040F493 push    dword_432F40
0040F499 call    sub_40F920
0040F49E pop     ebx
0040F49F pop     edi
0040F4A0 mov     esp, ebp
0040F4A2 pop     ebp
0040F4A3 retn
0040F4A3 ; END OF FUNCTION CHUNK FOR sub_40601B
```

```
00405F2E sub_405F2E proc near
00405F2E push    dword_432F4C
00405F34 push    0
call        sub 405F7A
```

Win32.Swizzor’s packer

How To Find Crypto Code ? (2)

How To Find Crypto Code ? (2)

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

How To Find Crypto Code ? (2)

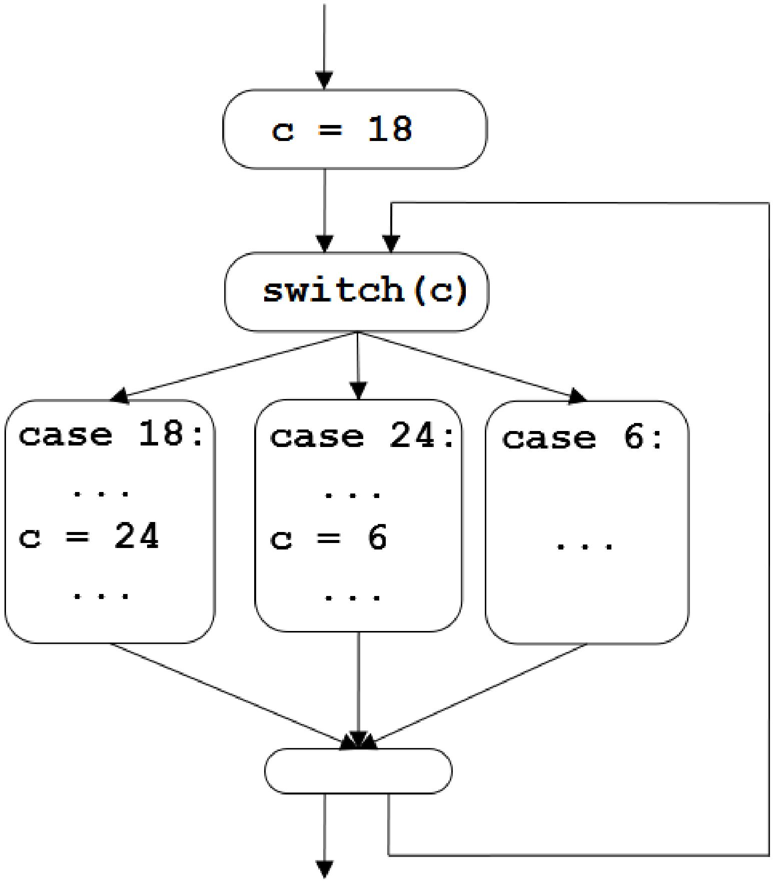
- Cryptographic algorithms usually apply *a same treatment* on their input-output parameters.

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

How To Find Crypto Code ? (2)

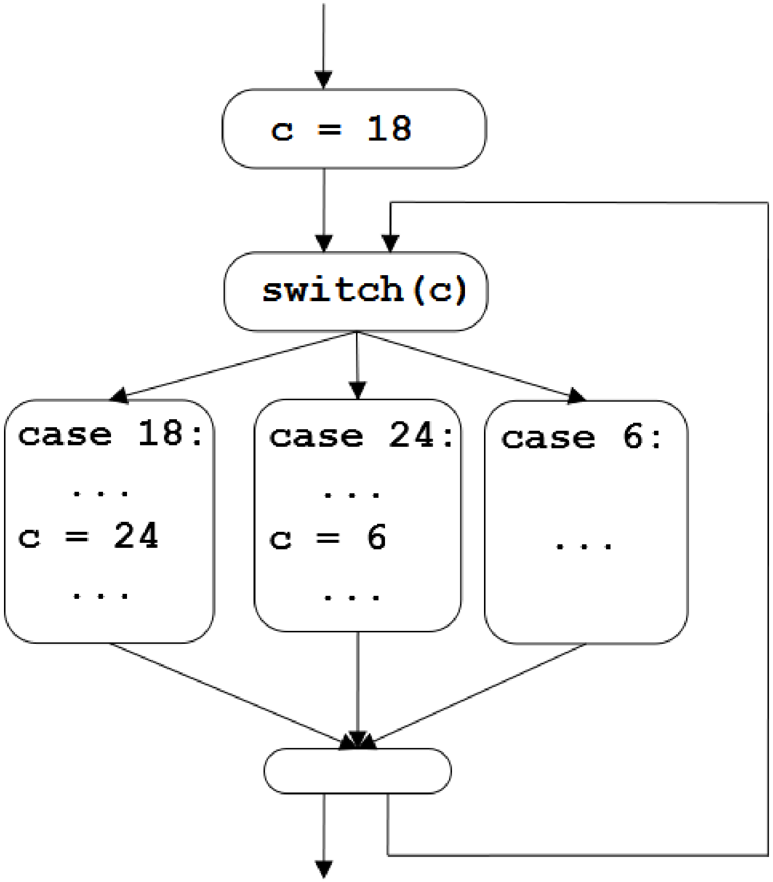
- Cryptographic algorithms usually apply *a same treatment* on their input-output parameters.
- It makes **loops** a cryptographic code feature.
- But there are loops everywhere, not only in crypto... What kind of loops are we looking for ?

Loops ?



Mebroot state-machine

Loops ?

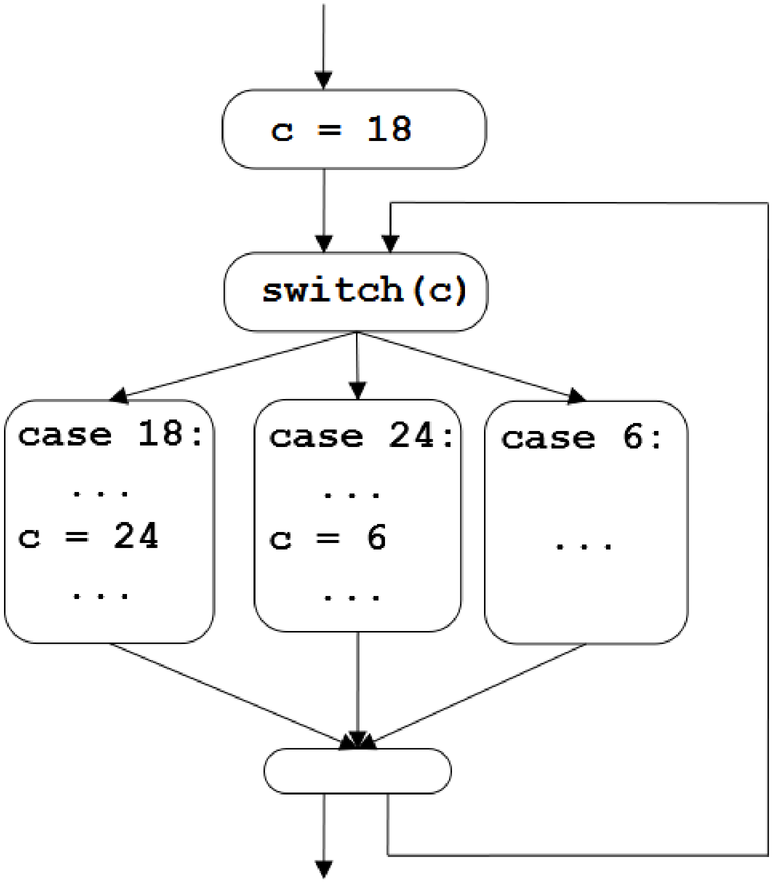


Mebroot state-machine

```
0x00 inc eax
0x01 inc ebx
0x02 mov [ebx], eax
0x03 inc eax
0x04 inc ebx
0x05 mov [ebx], eax
0x06 inc eax
0x07 inc ebx
0x08 mov [ebx], eax
```

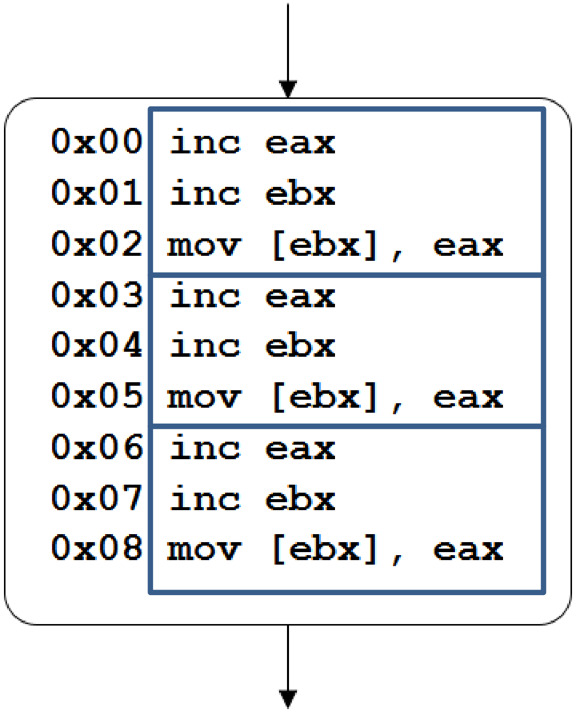
Unrolling optimization

Loops ?



```
c = 18
switch(c)
case 18:
  ...
c = 24
  ...
case 24:
  ...
c = 6
  ...
case 6:
  ...
```

Mebroot state-machine



```
0x00 inc eax
0x01 inc ebx
0x02 mov [ebx], eax
0x03 inc eax
0x04 inc ebx
0x05 mov [ebx], eax
0x06 inc eax
0x07 inc ebx
0x08 mov [ebx], eax
```

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.)

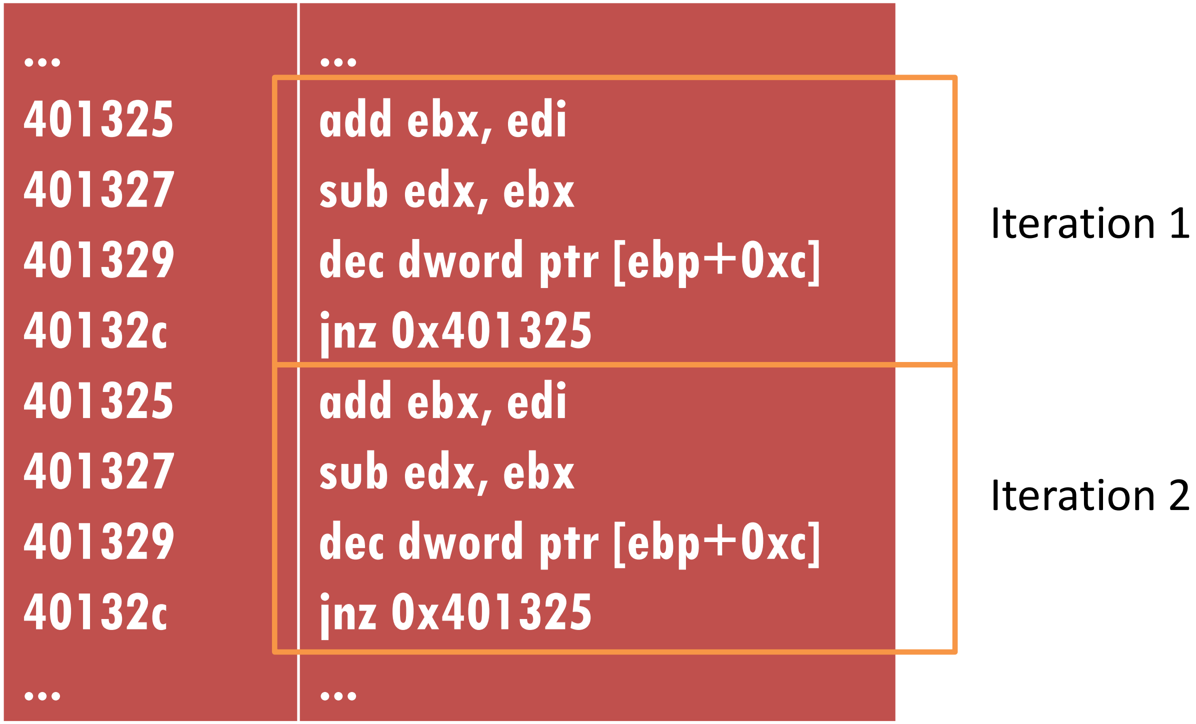
Example

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



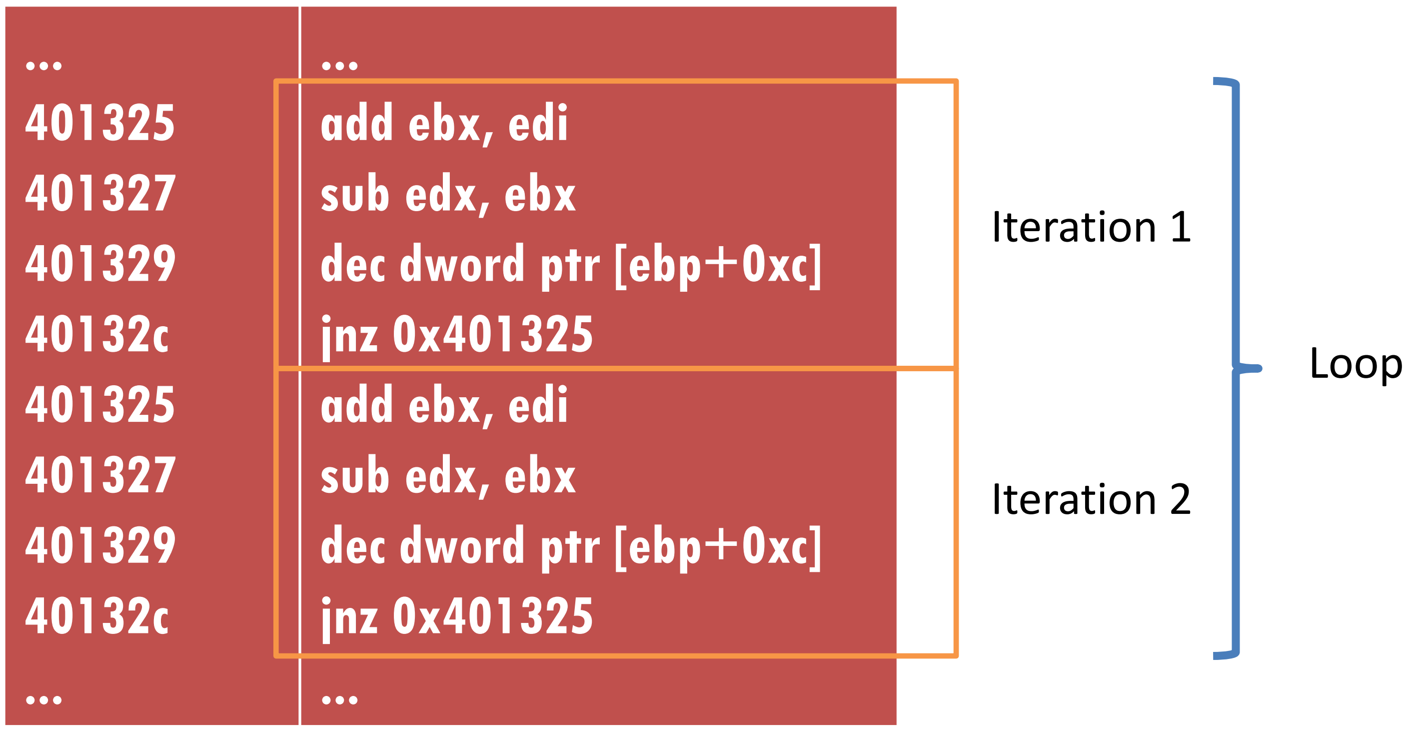
```
...       ...
401325    add ebx, edi
401327    sub edx, ebx
401329    dec dword ptr [ebp+0xc]
40132c    jnz 0x401325
401325    add ebx, edi
401327    sub edx, ebx
401329    dec dword ptr [ebp+0xc]
40132c    jnz 0x401325
...       ...
```

Iteration 1

Iteration 2

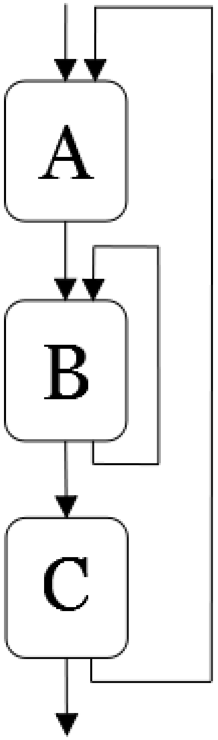
Example

Execution Trace



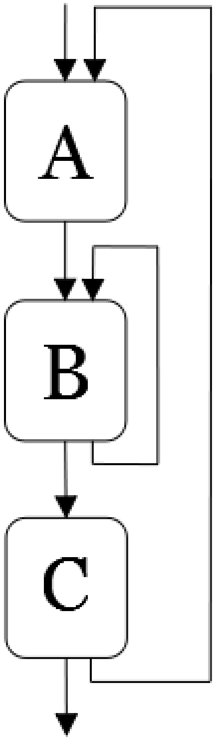
```
...       ...
401325    add ebx, edi
401327    sub edx, ebx
401329    dec dword ptr [ebp+0xc]
40132c    jnz 0x401325
401325    add ebx, edi
401327    sub edx, ebx
401329    dec dword ptr [ebp+0xc]
40132c    jnz 0x401325
...       ...
```

What About Nested Loops ?



Simplified CFG

What About Nested Loops ?

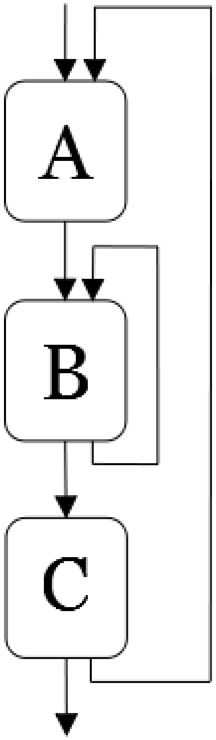


Simplified CFG

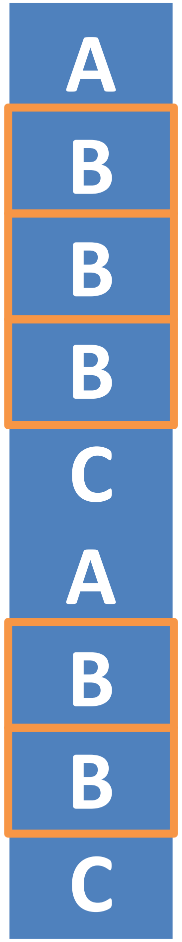
A
B
B
B
C
A
B
B
C

Execution trace

What About Nested Loops ?



Simplified CFG

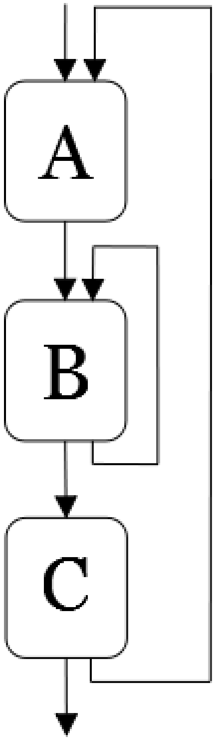


Loop B
3 iterations

Loop B
2 iterations

Execution trace

What About Nested Loops ?

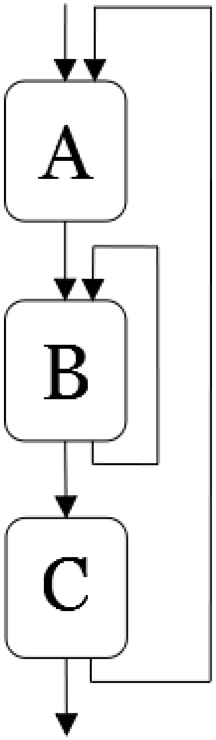


Simplified CFG

A
B
B
B
C
A
B
B
C

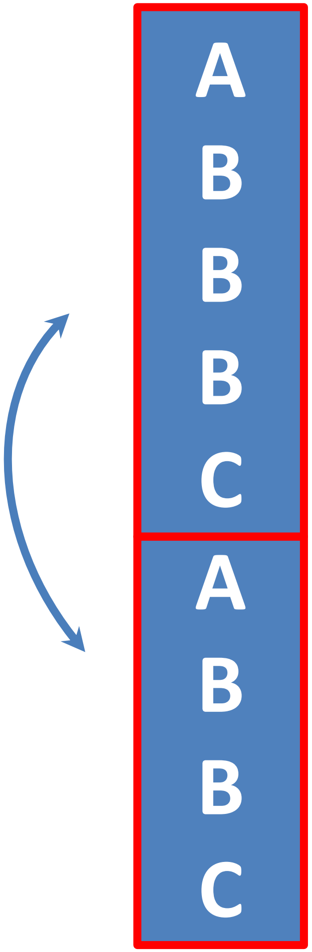
Execution trace

What About Nested Loops ?



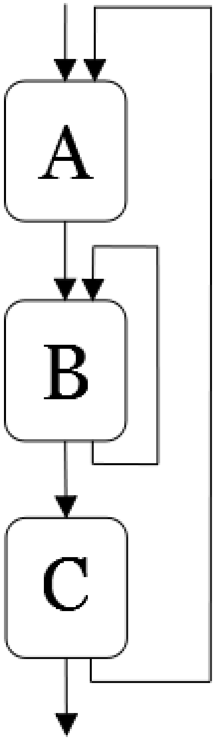
Simplified CFG

Different!



Execution trace

What About Nested Loops ?

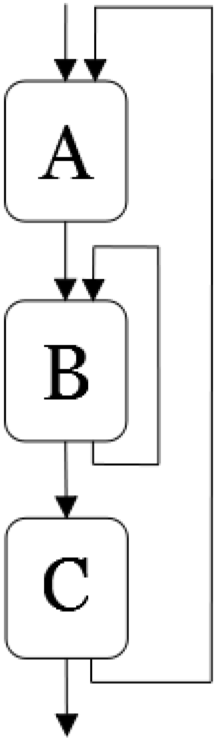


Simplified CFG

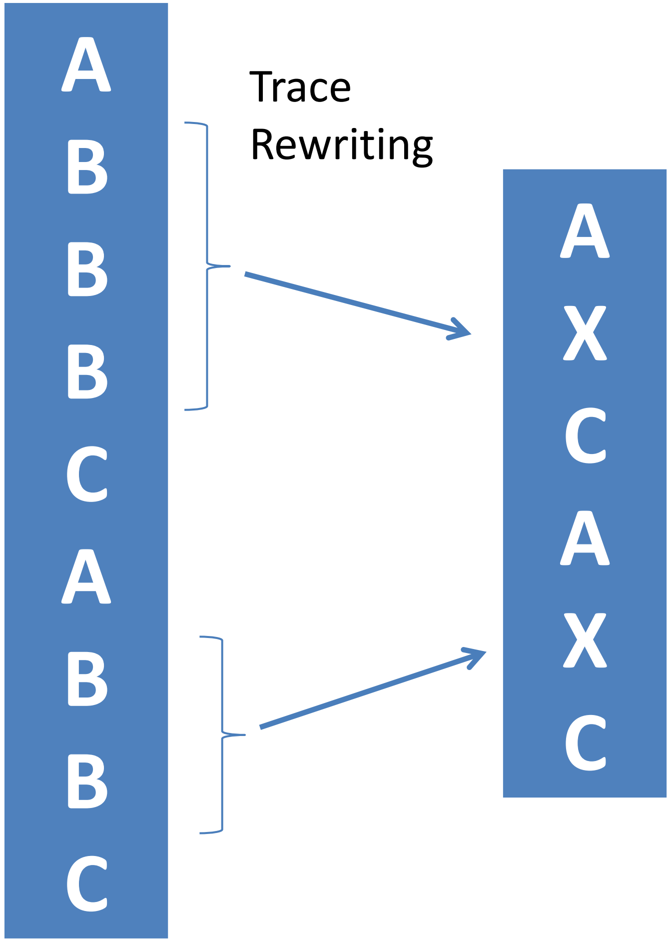
A
B
B
B
C
A
B
B
C

Execution trace

What About Nested Loops ?



Simplified CFG



A
B
B
B
C
A
B
B
C

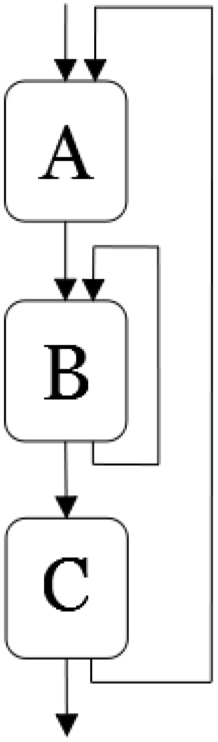
Trace Rewriting

A
X
C
A
X
C

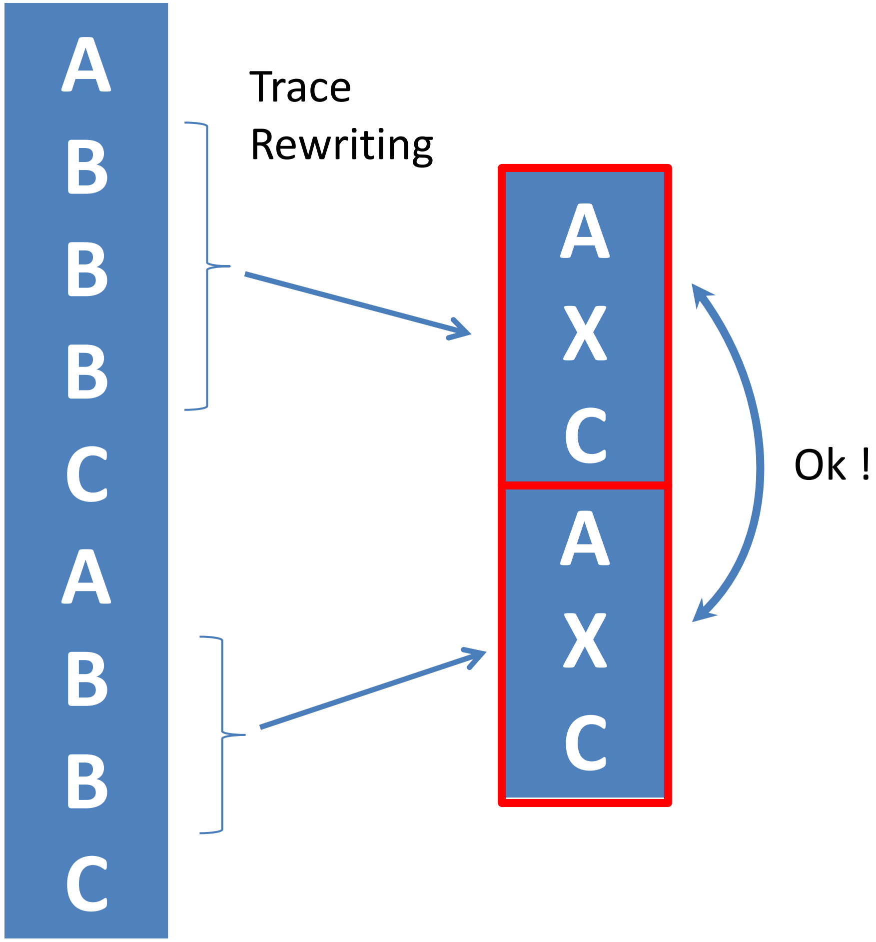
Ok !

Execution trace

What About Nested Loops ?



Simplified CFG



A
B
B
B
C
A
B
B
C

Trace Rewriting

A
X
C
A
X
C

Ok !

Execution trace

Loop Detection Algorithm

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

(non trivial, because there is an infinity of possible loop body)

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 a same instruction in the loop body**.

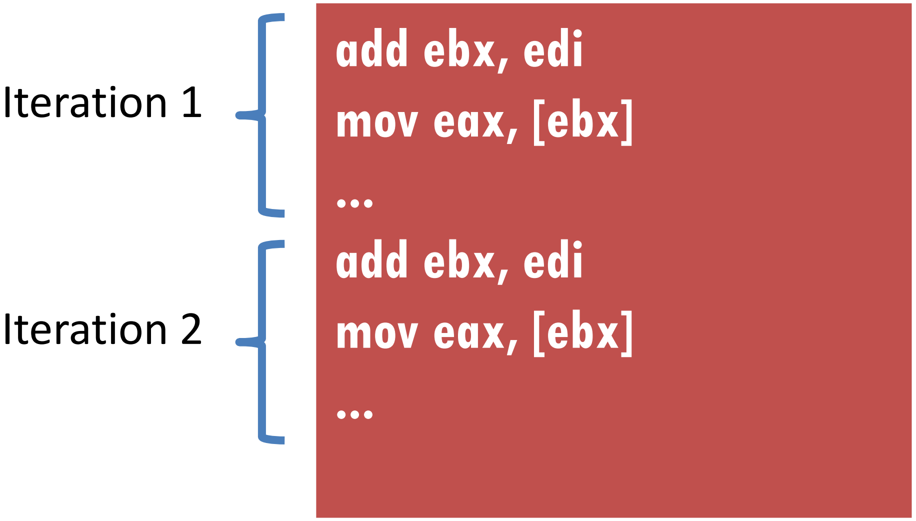
Loop Parameters (3)

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



Iteration 1

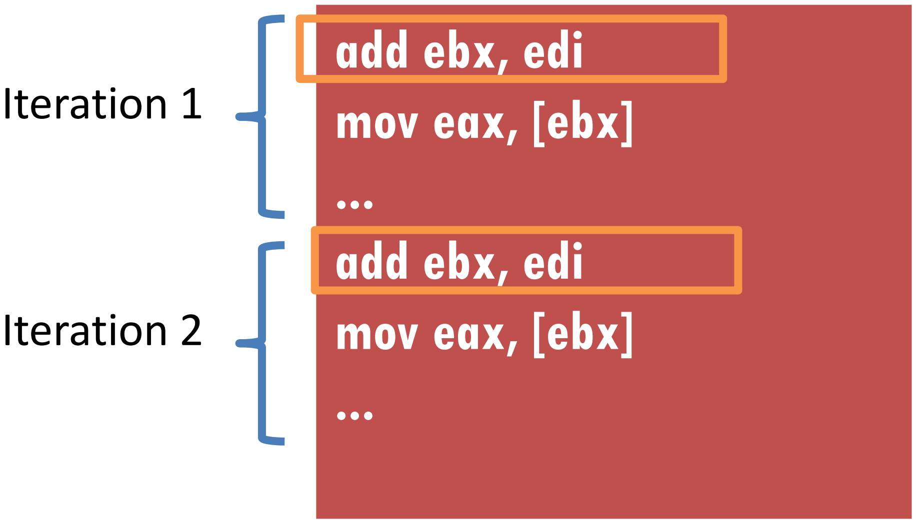
```
add ebx, edi
mov eax, [ebx]
...
```

Iteration 2

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

Loop Parameters (3)

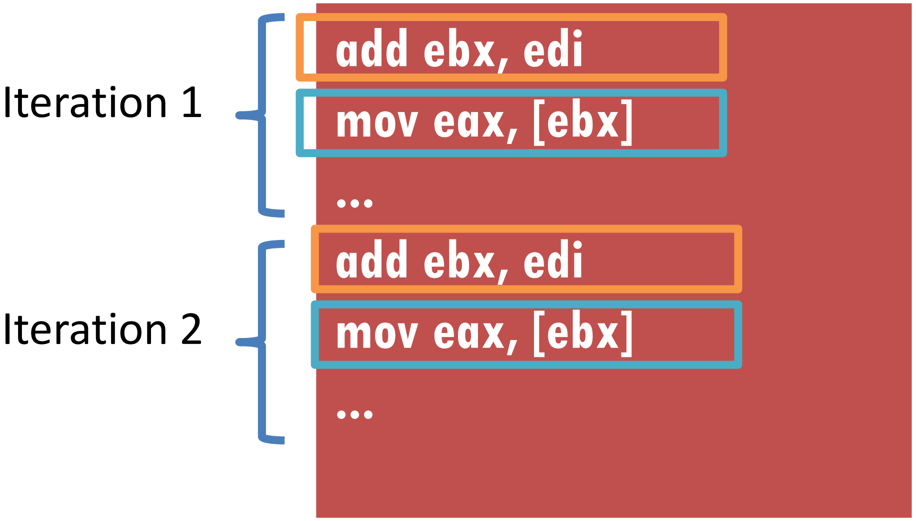
- Grouping of several bytes into the same parameter:
  1. If they are **adjacent in memory** *(too large!)*
  2. And if they are **manipulated by a same instruction in the loop body**.



```
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 a same instruction in the loop body**.



Iteration 1

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

Iteration 2

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

Loop Parameters (4)

- Once a parameter has been defined, we collect its value from the trace.

Loop Parameters (4)

- Once a parameter has been defined, we collect its value from the trace.

Parameter

`(memory address|register name):size`

`Value in hexadecimal`

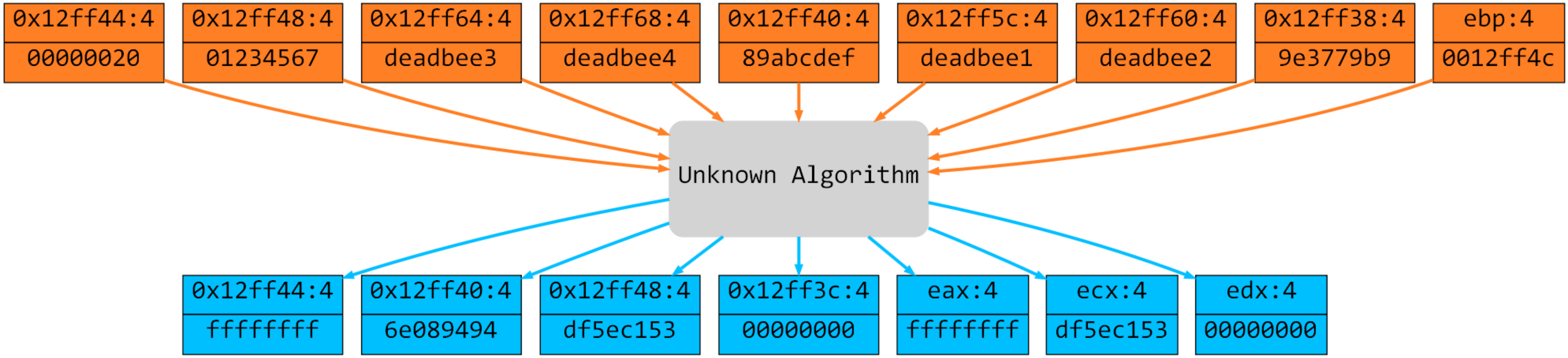
Let’s Recap With a Use-Case

- Tiny Encryption Algorithm:
  - Block cipher
  - 64 round Feistel network
  - 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`**

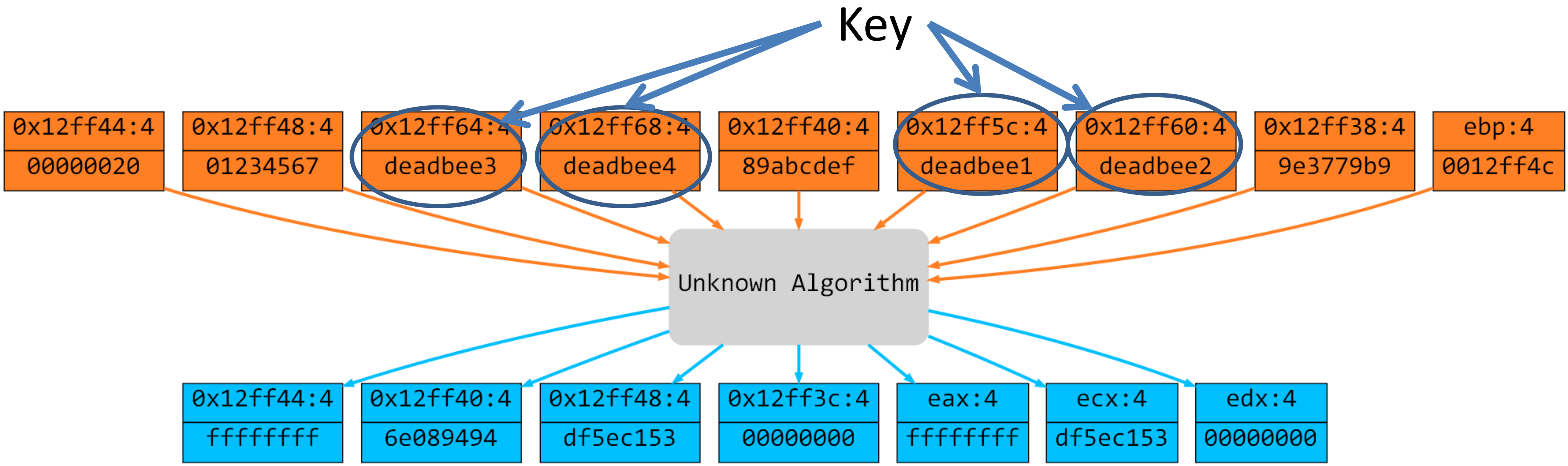
```
4010d0!push ebp!RR_ebp_12ffc0_esp_12ff80!WM_12ff7c_4_12ffc0!WR_esp_12ff7c
4010d1!mov ebp, esp!RR_esp_12ff7c!WR_ebp_12ff7c
4010d3!sub esp, 0x18!RR_esp_12ff7c!WR_esp_12ff64
4010d6!mov dword ptr [ebp-0x8], 0x01234567!RR_ebp_12ff7c!WM_12ff74_4_cafebabe
4010dd!mov dword ptr [ebp-0x4], 0x890ABCDEF!RR_ebp_12ff7c!WM_12ff78_4_cafebabe
4010e4!mov dword ptr [ebp-0x18], 0xdeadbee1!RR_ebp_12ff7c!WM_12ff64_4_deadbeef
4010eb!mov dword ptr [ebp-0x14], 0xdeadbee2!RR_ebp_12ff7c!WM_12ff68_4_deadbeef
4010f2!mov dword ptr [ebp-0x10], 0xdeadbee3!RR_ebp_12ff7c!WM_12ff6c_4_deadbeef
4010f9!mov dword ptr [ebp-0xc], 0xdeadbee4!RR_ebp_12ff7c!WM_12ff70_4_deadbeef
401100!lea eax, ptr [ebp-0x18]!RR_ebp_12ff7c!WR_eax_12ff64
                                  …
```

Step 1: collect the execution trace

Step 2: cryptographic algorithm extraction



Step 2: cryptographic algorithm extraction



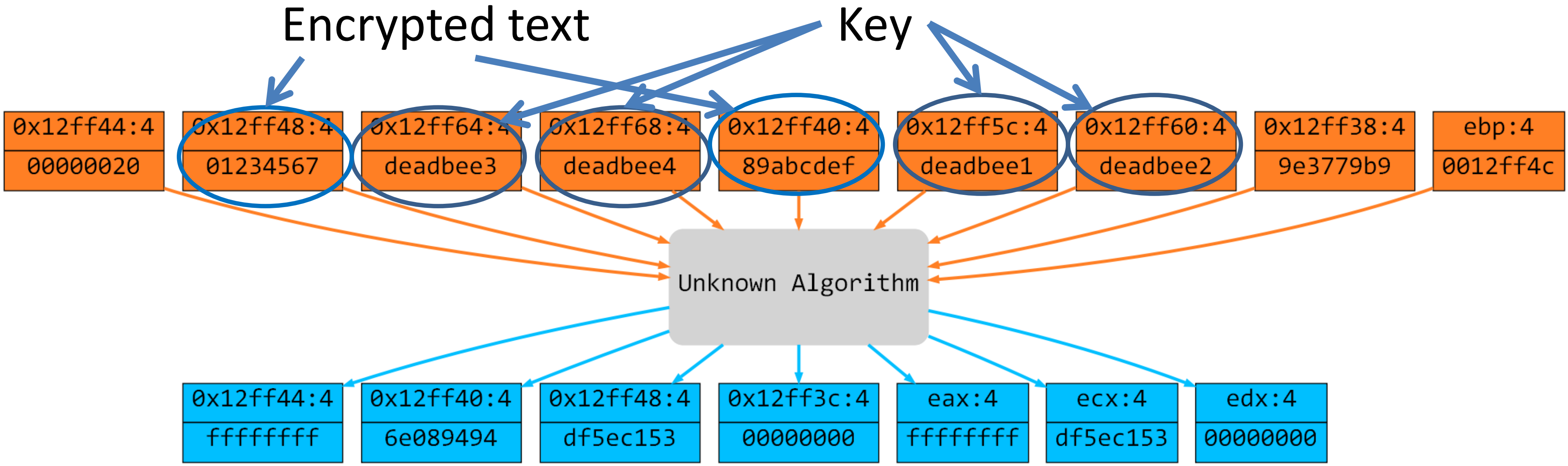
Key

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

Step 2: cryptographic algorithm extraction



Encrypted text

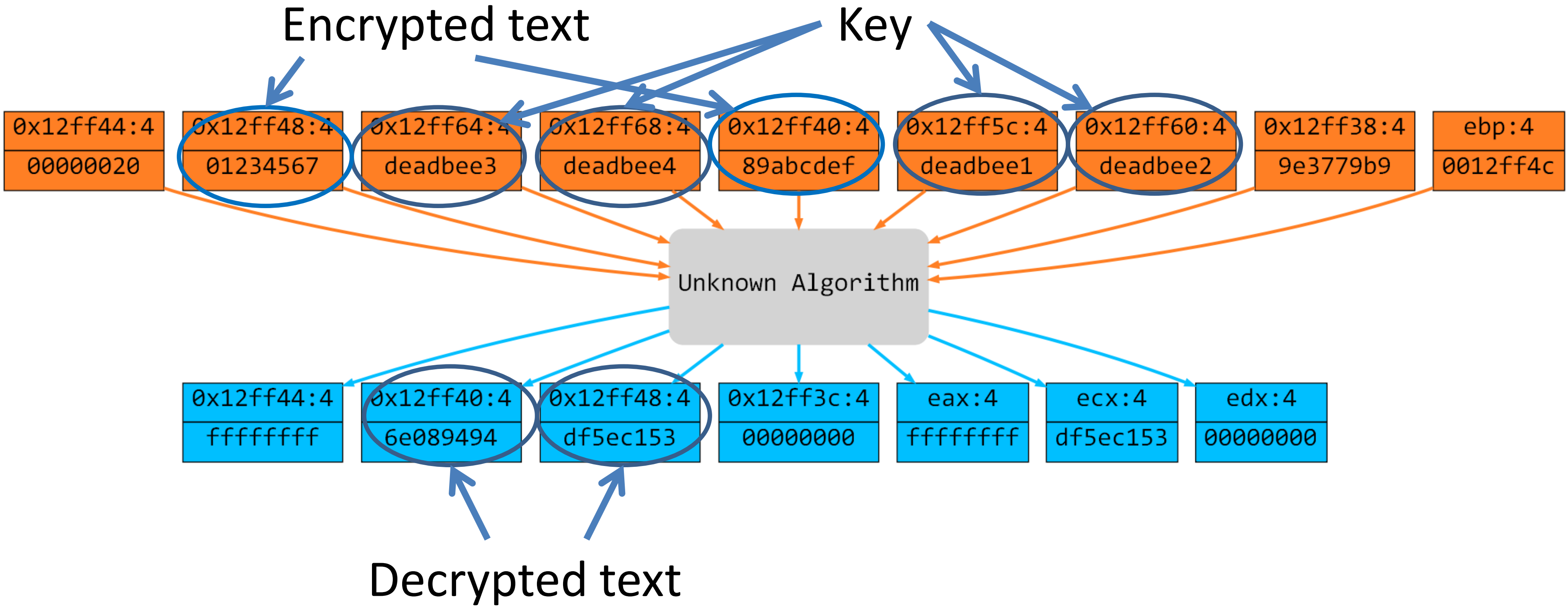
Key

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

Step 2: cryptographic algorithm extraction


Encrypted text

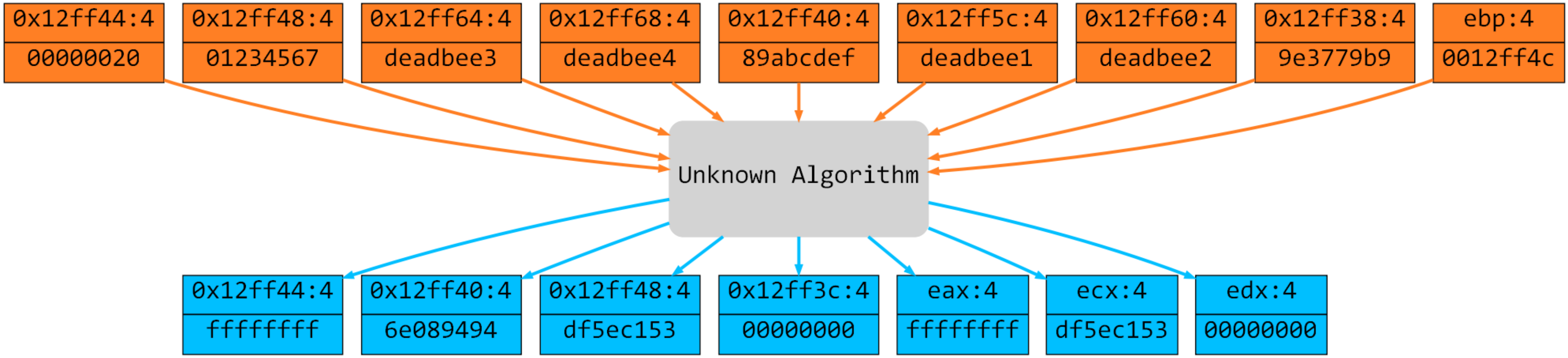
Key

Decrypted text

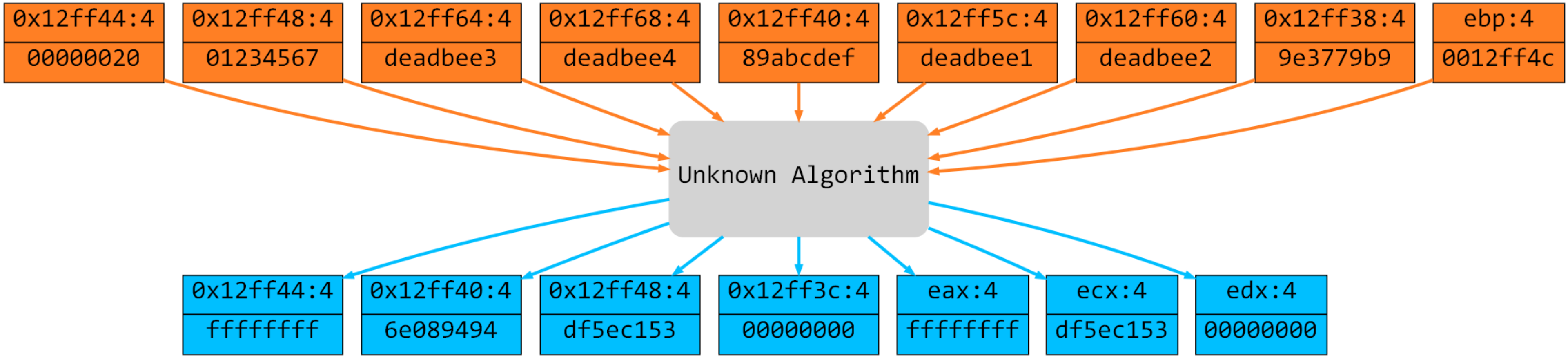
Step 2: cryptographic algorithm extraction

STEP 3: CRYPTO ALGORITHM IDENTIFICATION

<u>Input 1:</u> unknown algorithm **A** with its parameter values



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



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

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** output values ?

Question

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

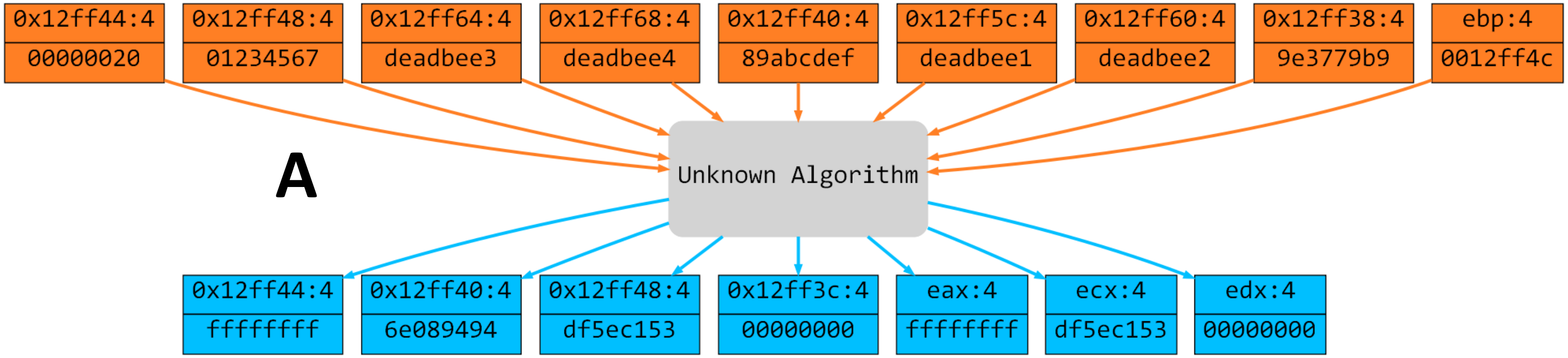
- Some difficulties:
  - **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  - **Parameter order:** no particular order for **A** parameters.

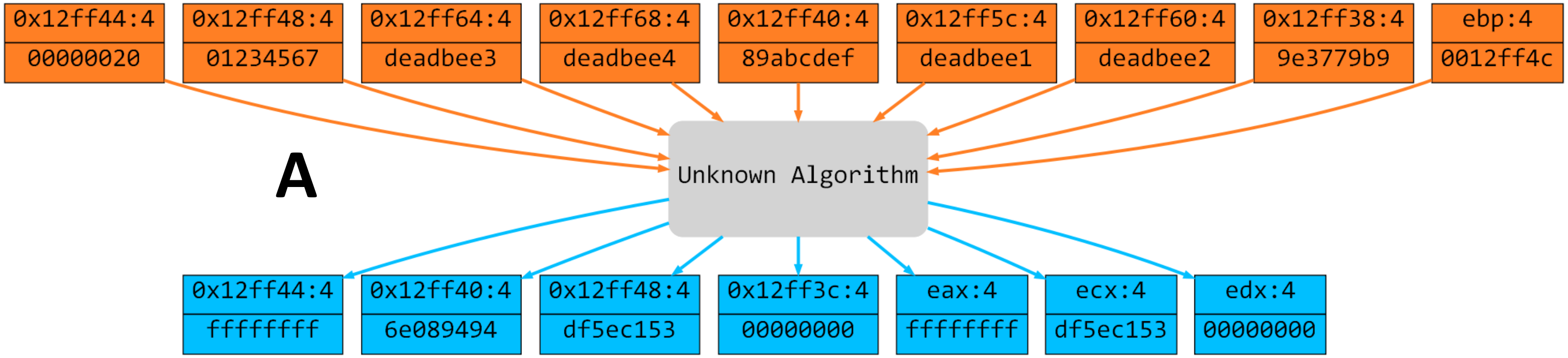
Question

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

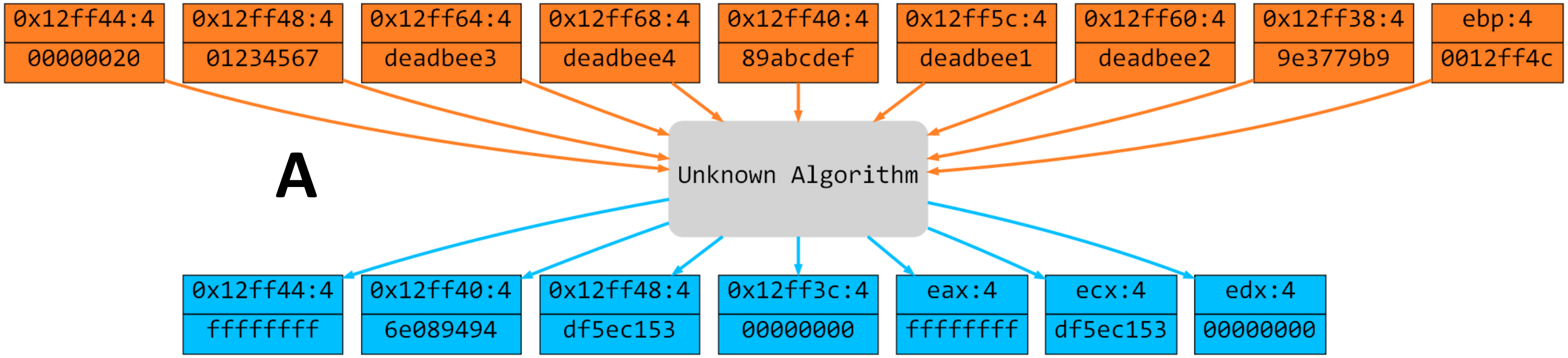
- Some difficulties:
  - **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  - **Parameter order:** no particular order for **A** parameters.
  - **Parameter number:** we collect more than the cryptographic parameters.

Brute-Force!

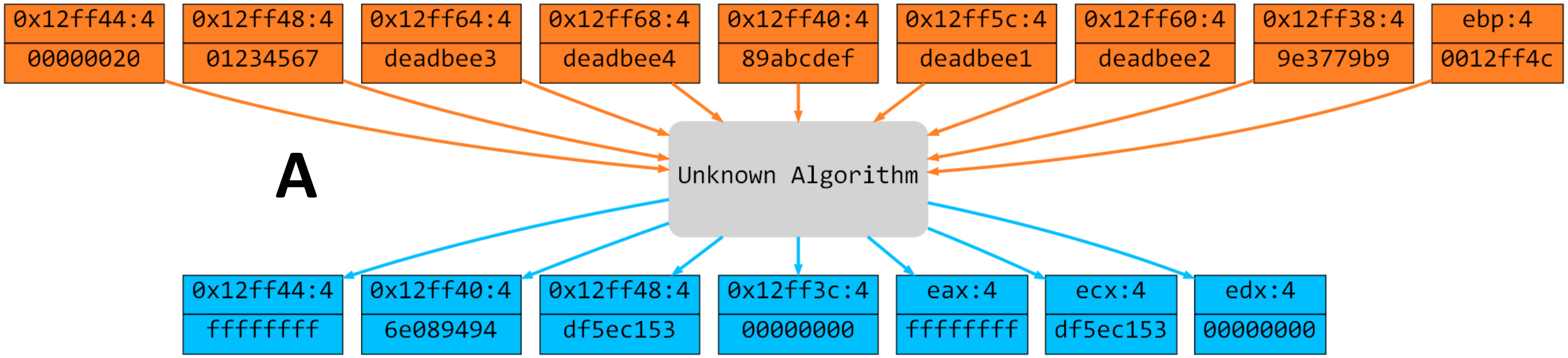

0x12ff44:4
00000020
0x12ff48:4
01234567
0x12ff64:4
deadbee3
0x12ff68:4
deadbee4
0x12ff40:4
89abcdef
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
ebp:4
0012ff4c
A
Unknown Algorithm
0x12ff44:4
ffffffff
0x12ff40:4
6e089494
0x12ff48:4
df5ec153
0x12ff3c:4
00000000
eax:4
ffffffff
ecx:4
df5ec153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



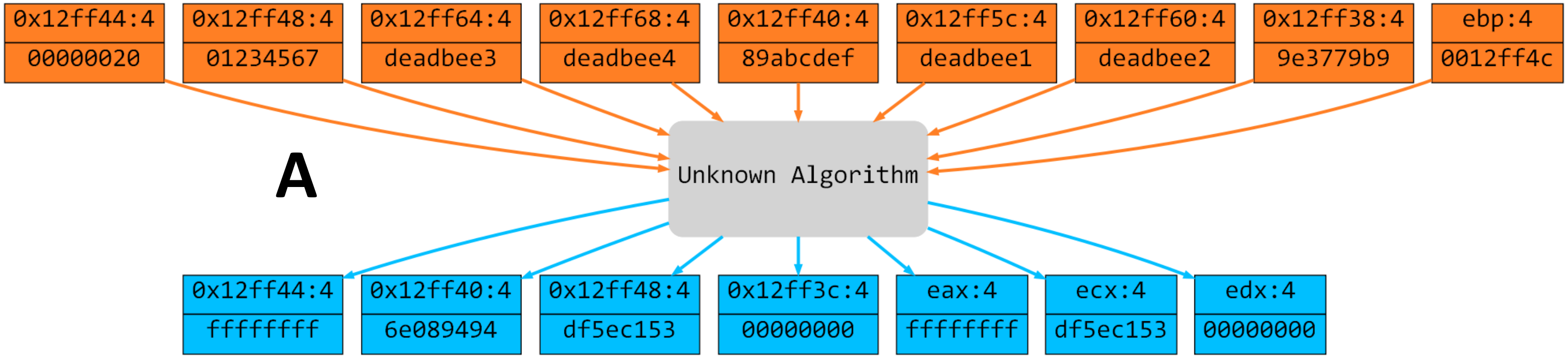
0x12ff44:4 00000020
0x12ff48:4 01234567
0x12ff64:4 deadbee3
0x12ff68:4 deadbee4
0x12ff40:4 89abcdef
0x12ff5c:4 deadbee1
0x12ff60:4 deadbee2
0x12ff38:4 9e3779b9
ebp:4 0012ff4c

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. …
2. Same thing with **A** output parameters.
3. For TEA reference implementation:
   1. Possible input texts (8 bytes): *0000002001234567*,…



A

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

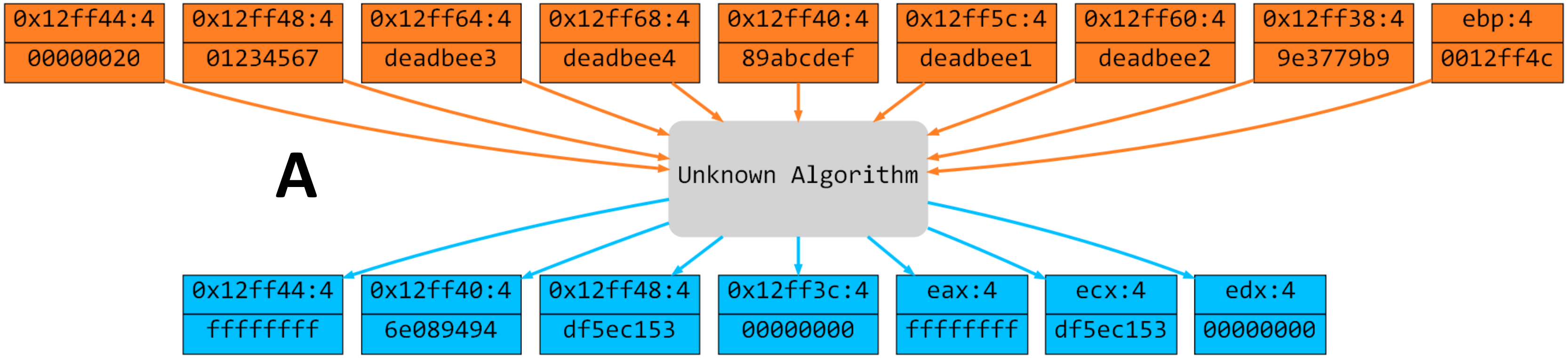
0x12ff3c:4 00000000

eax:4 ffffffff

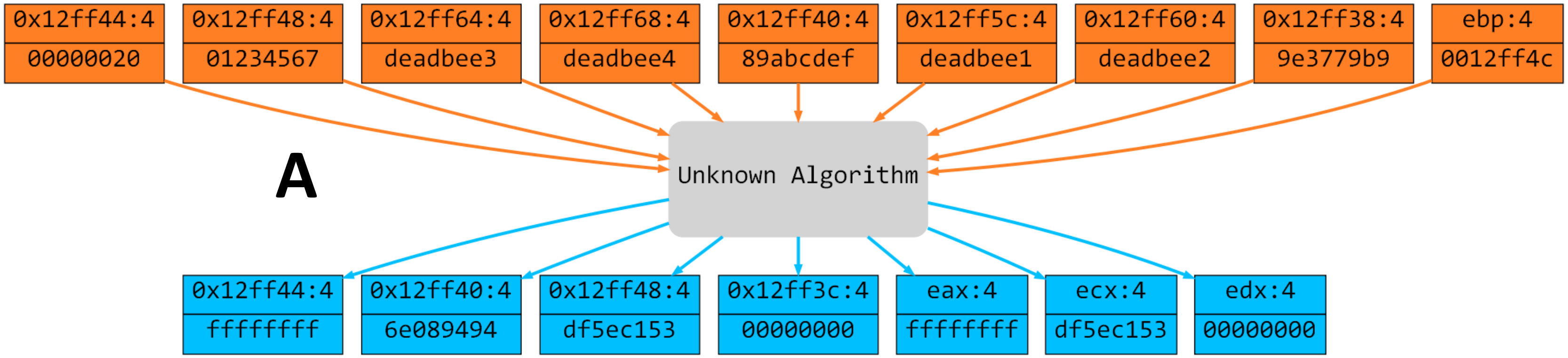
ecx:4 df5ec153

edx:4 00000000

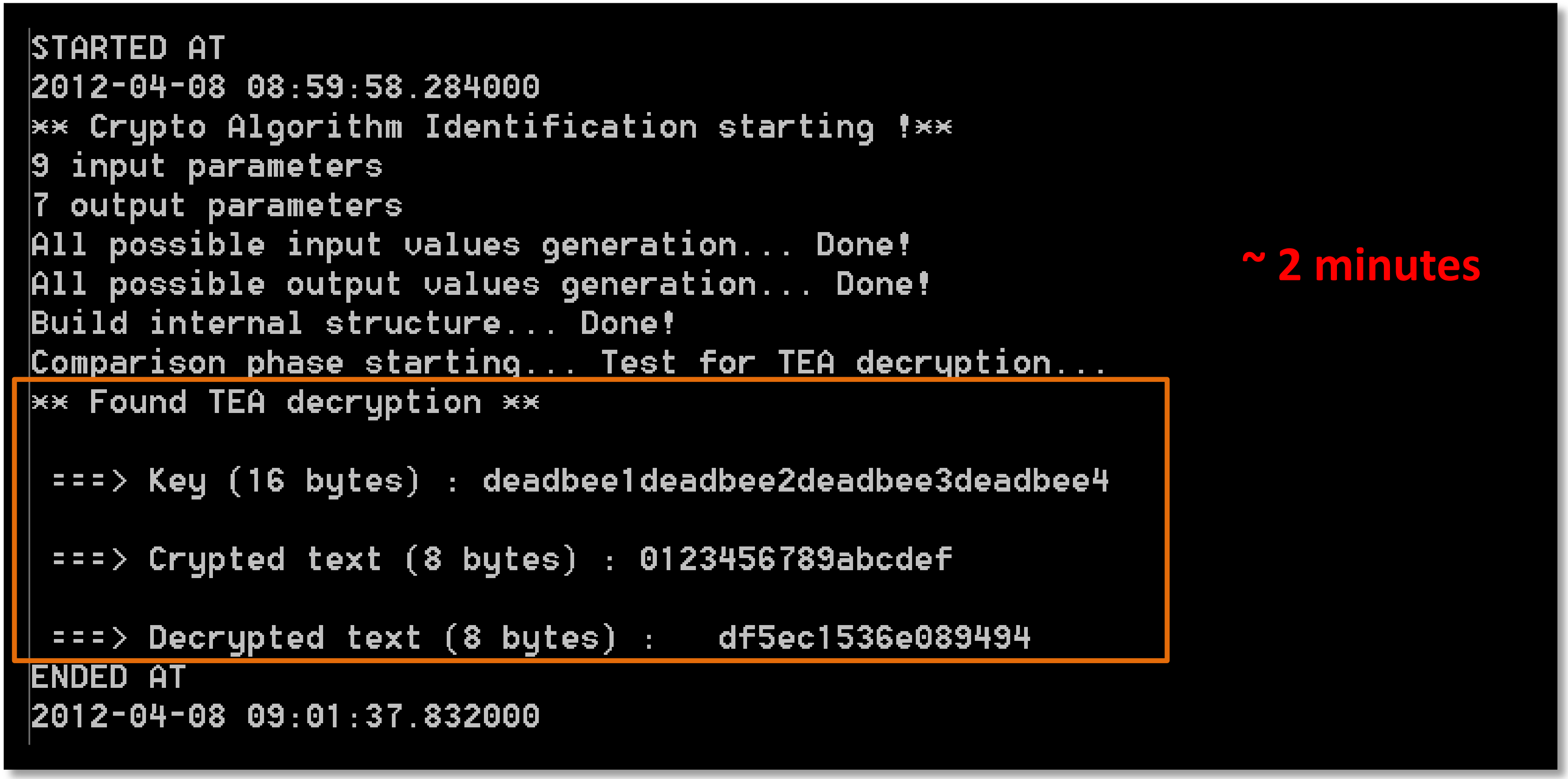
1. Generate all possible values with **A** input parameters:
    1. Length 4: *00000020*, *01234567*, *deadbee3*...
    2. Length 8: *0000002001234567*, *00000020deadbee3*,..
    3. ...
2. Same thing with **A** output parameters.
3. For TEA reference implementation:
    1. Possible input texts (8 bytes):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

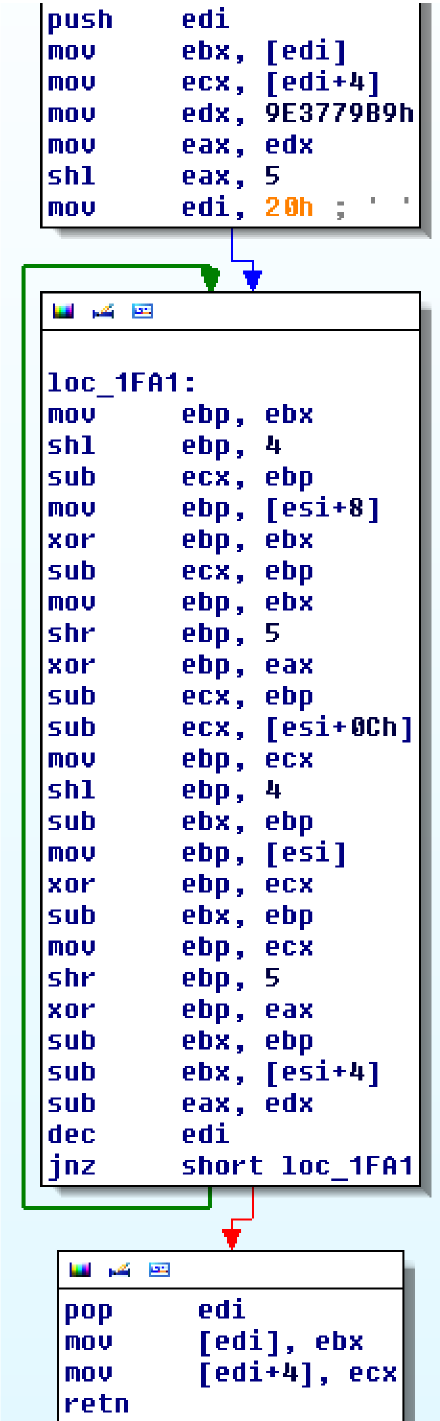
Malware And TEA

EXAMPLES!

Storm Worm

- Several internet references about the use of TEA in the Storm Worm packer (aka Tibs).

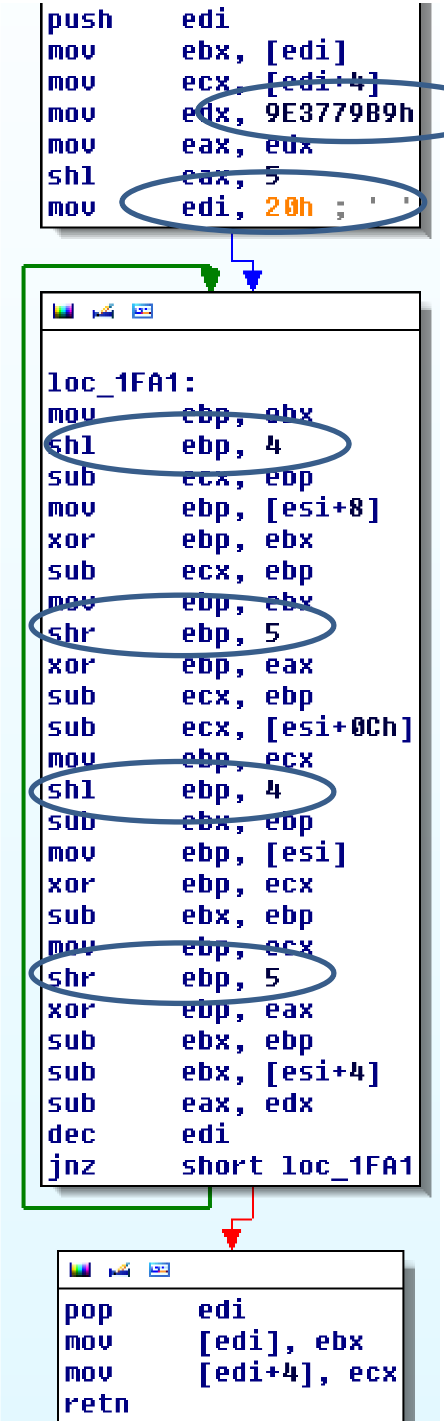
- Let’s take a look to the code...



```
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 ; ' '
```

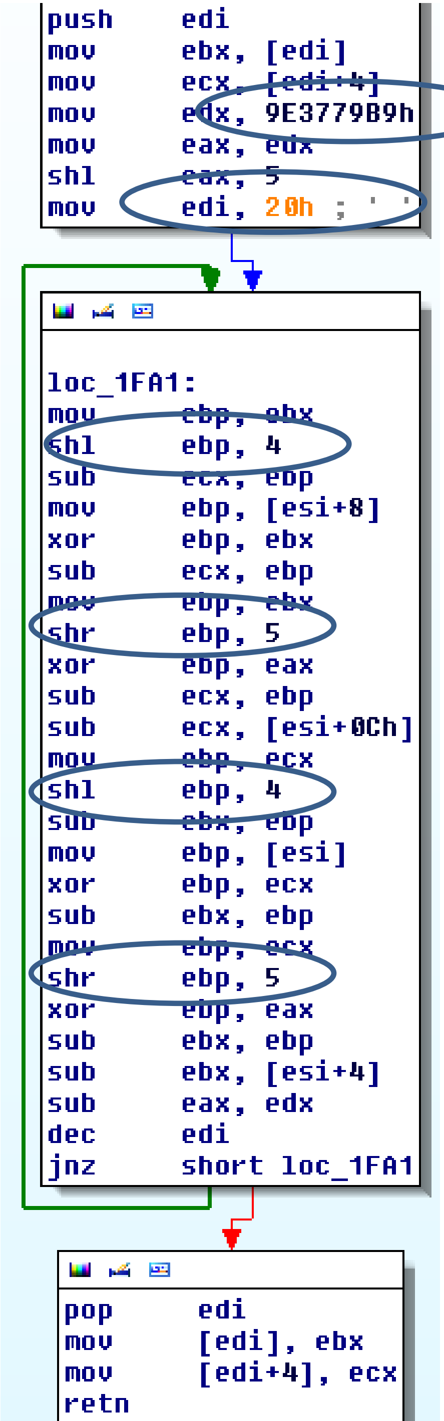
```
loc_1FA1:
mov     ebp, ebx
shl     ebp, 4
sub     ecx, ebp
mov     ebp, [esi+8]
xor     ebp, ebx
sub     ecx, ebp
mov     ebp, ebx
shr     ebp, 5
xor     ebp, eax
sub     ecx, ebp
sub     ecx, [esi+0Ch]
mov     ebp, ecx
shl     ebp, 4
sub     ebx, ebp
mov     ebp, [esi]
xor     ebp, ecx
sub     ebx,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

TEA delta

TEA round number

Classic TEA operations



```
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
```

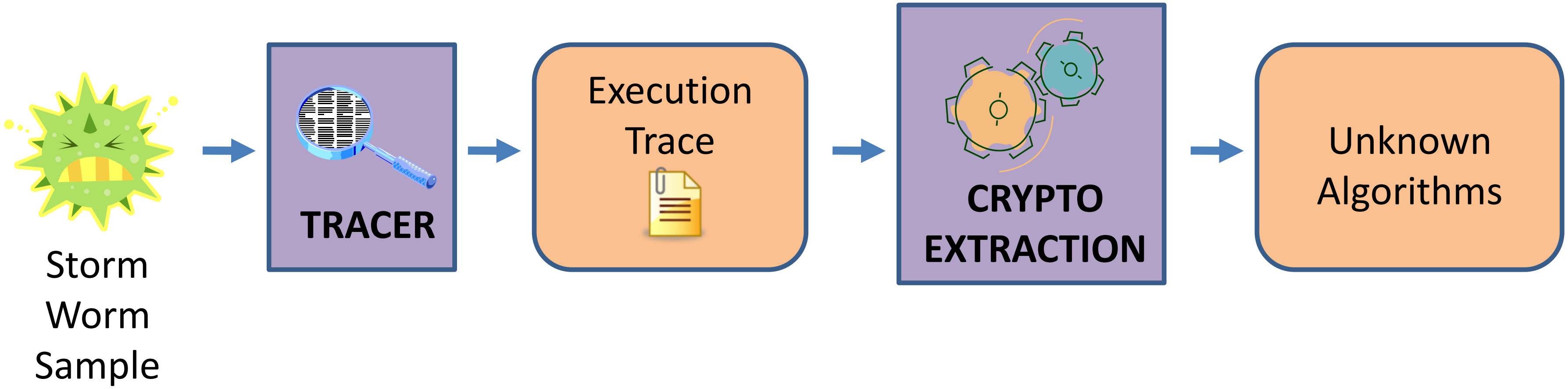
```
pop     edi
mov     [edi], ebx
mov     [edi+4], ecx
retn
```

TEA delta

TEA round number

Classic TEA operations

Let’s try our tool…



Storm Worm Sample

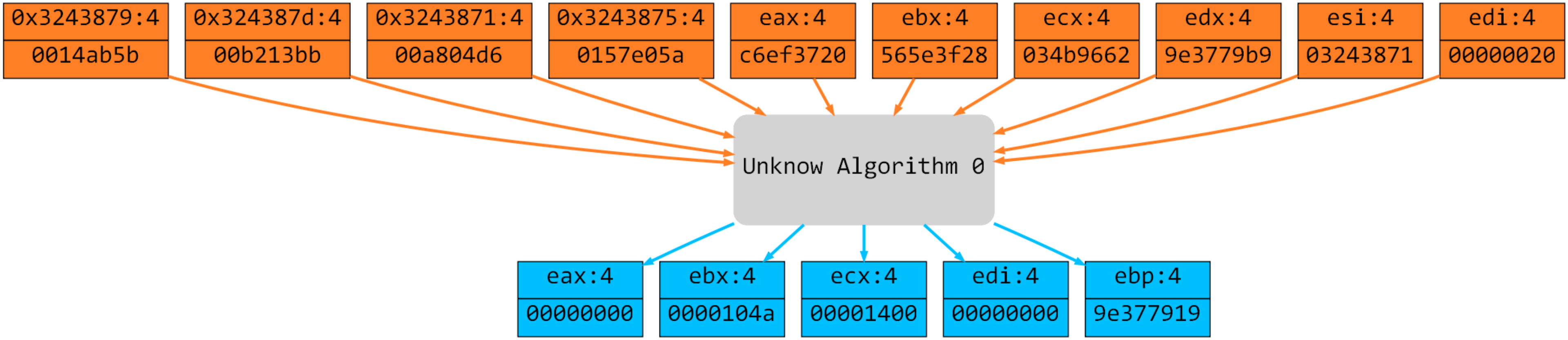
TRACER

Execution Trace

CRYPTO EXTRACTION

Unknown Algorithms

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



0x3243879:4
0014ab5b

0x324387d:4
00b213bb

0x3243871:4
00a804d6

0x3243875:4
0157e05a

eax:4
c6ef3720

ebx:4
565e3f28

ecx:4
034b9662

edx:4
9e3779b9

esi:4
03243871

edi:4
00000020

Unknow Algorithm 0

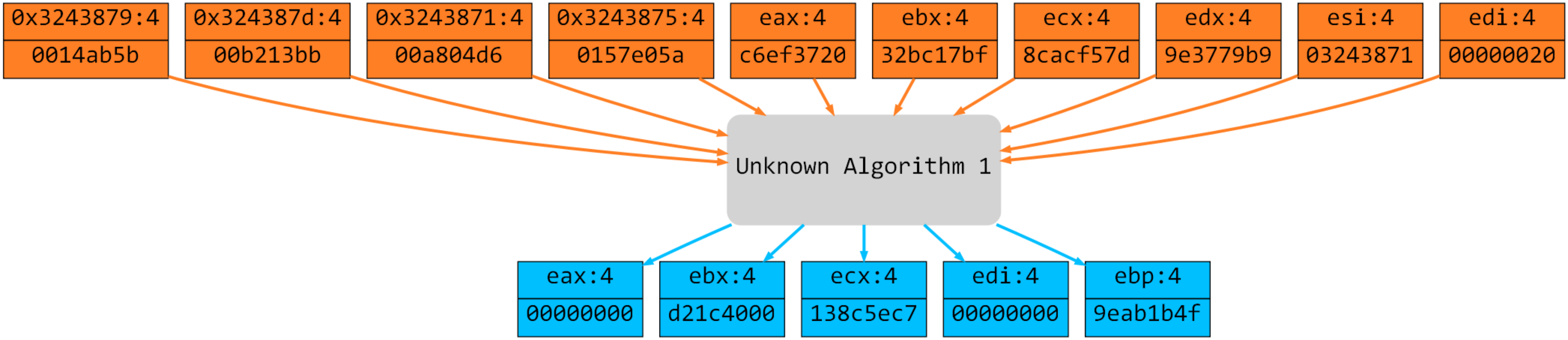
eax:4
00000000

ebx:4
0000104a

ecx:4
00001400

edi:4
00000000

ebp:4
9e377919



0x3243879:4
0014ab5b

0x324387d:4
00b213bb

0x3243871:4
00a804d6

0x3243875:4
0157e05a

eax:4
c6ef3720

ebx:4
32bc17bf

ecx:4
8cacf57d

edx:4
9e3779b9

esi:4
03243871

edi:4
00000020

Unknown Algorithm 1

eax:4
00000000

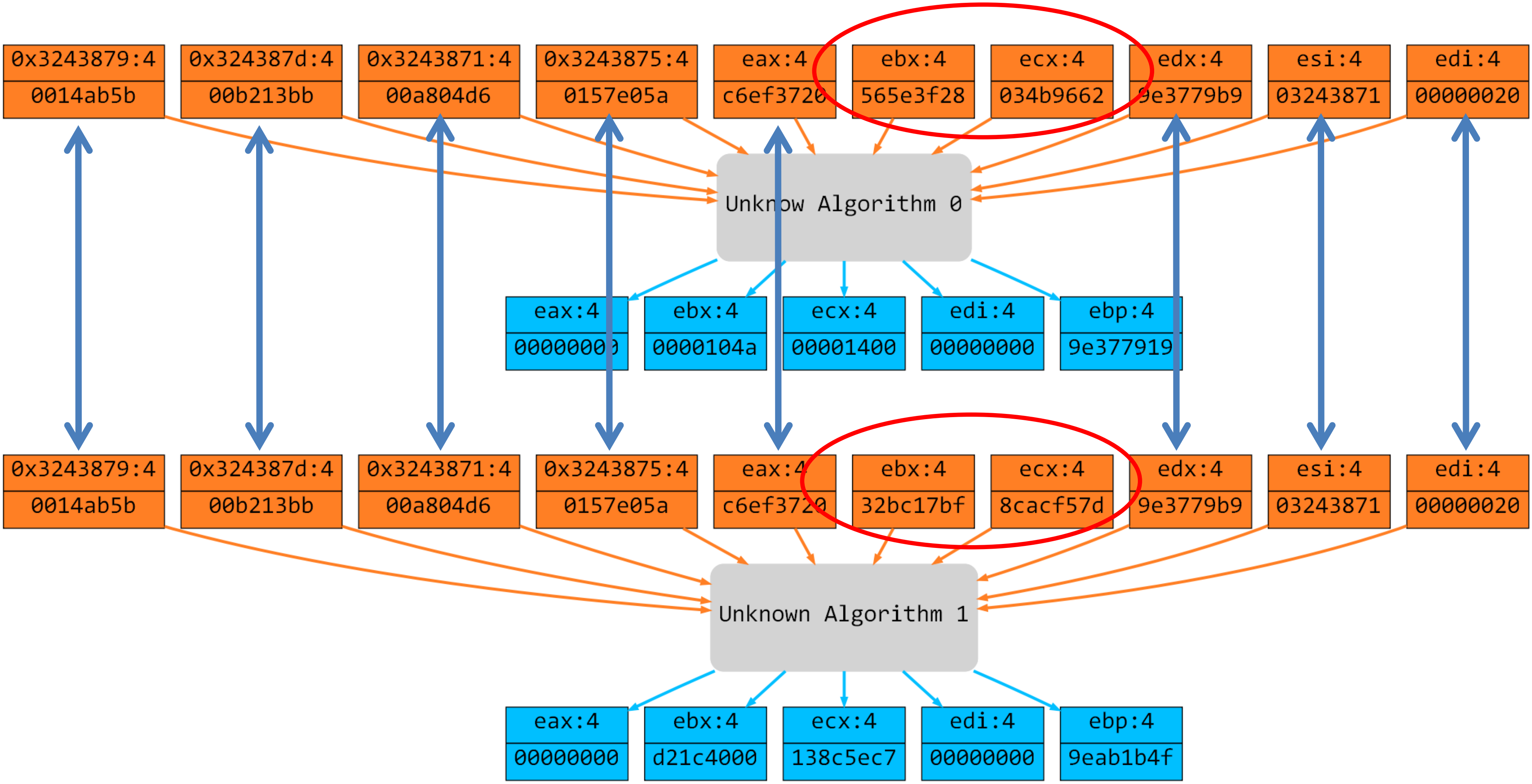
ebx:4
d21c4000

ecx:4
138c5ec7

edi:4
00000000

ebp:4
9eab1b4f

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



0x3243879:4	0x324387d:4	0x3243871:4	0x3243875:4	eax:4	ebx:4	ecx:4	edx:4	esi:4	edi:4
0014ab5b	00b213bb	00a804d6	0157e05a	c6ef3720	565e3f28	034b9662	9e3779b9	03243871	00000020

Unknow Algorithm 0

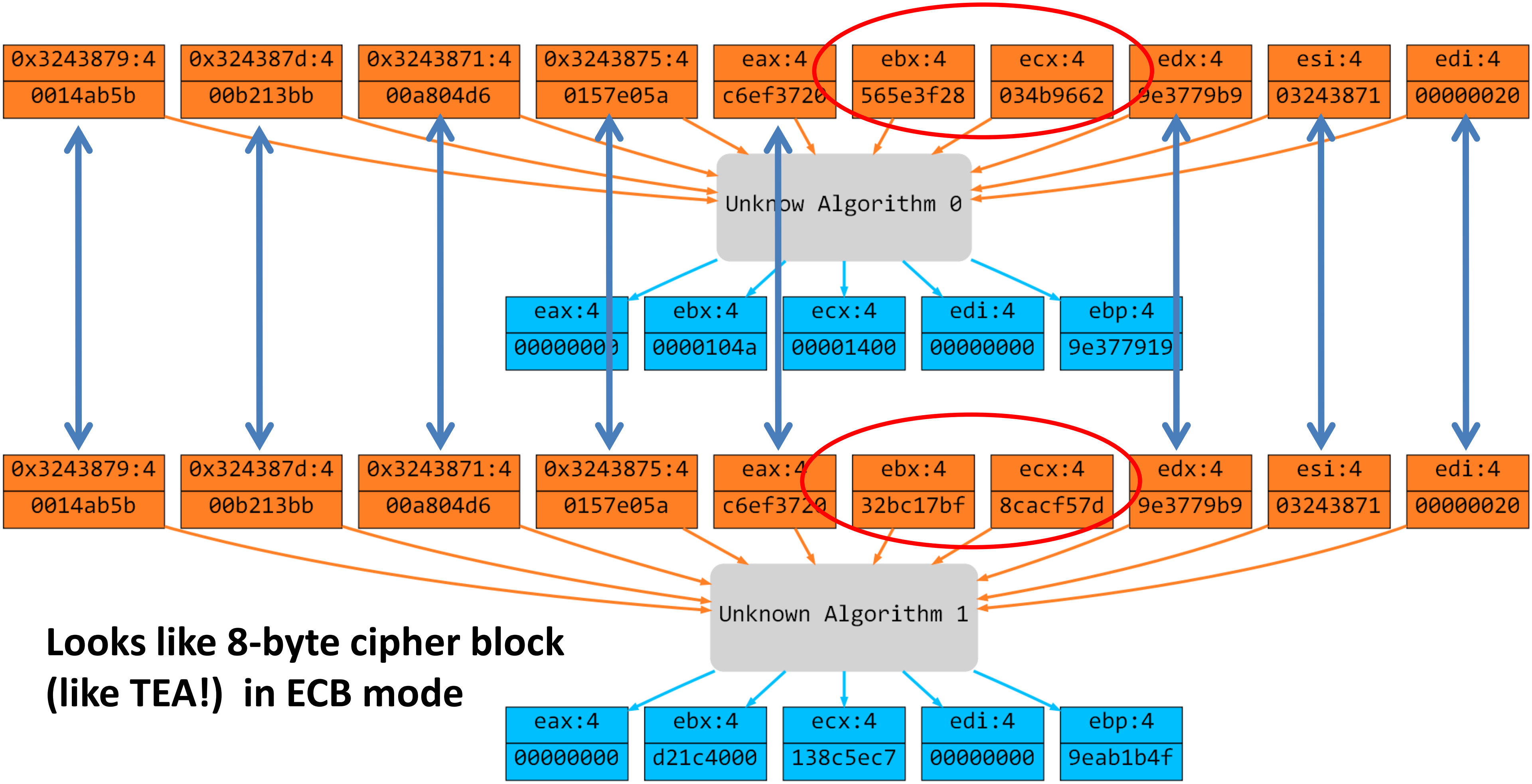
eax:4	ebx:4	ecx:4	edi:4	ebp:4
00000000	0000104a	00001400	00000000	9e377919

0x3243879:4	0x324387d:4	0x3243871:4	0x3243875:4	eax:4	ebx:4	ecx:4	edx:4	esi:4	edi:4
0014ab5b	00b213bb	00a804d6	0157e05a	c6ef3720	32bc17bf	8cacf57d	9e3779b9	03243871	00000020

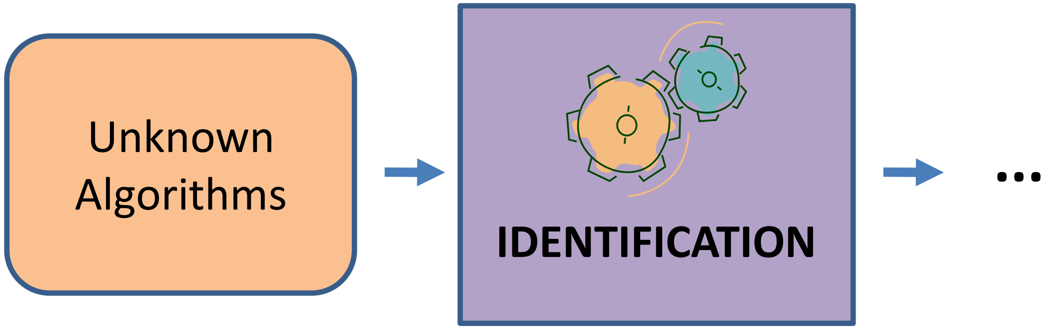
Unknown Algorithm 1

eax:4	ebx:4	ecx:4	edi:4	ebp:4
00000000	d21c4000	138c5ec7	00000000	9eab1b4f

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



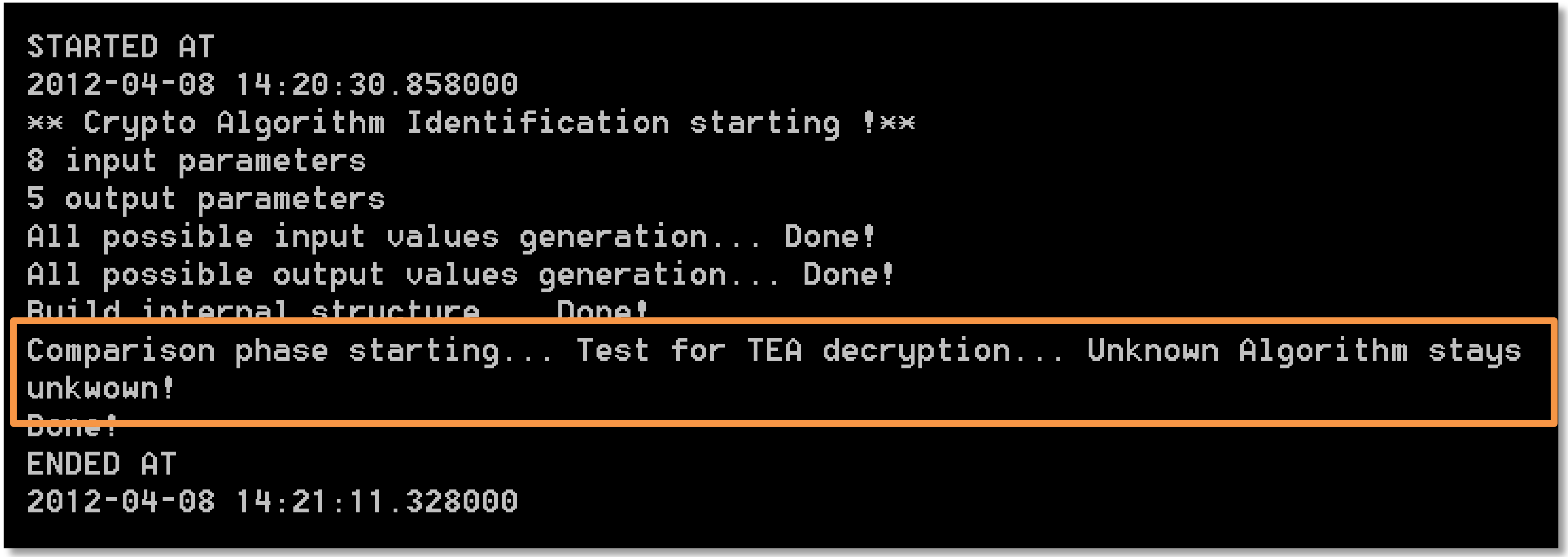
Looks like 8-byte cipher block (like TEA!) in ECB mode



Unknown Algorithms

IDENTIFICATION

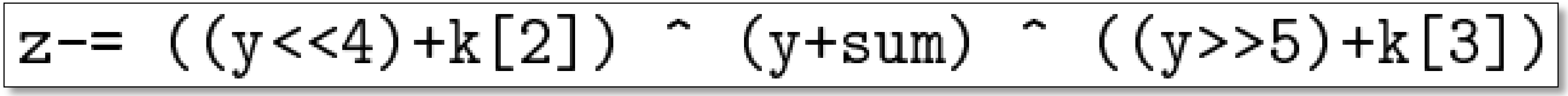
…



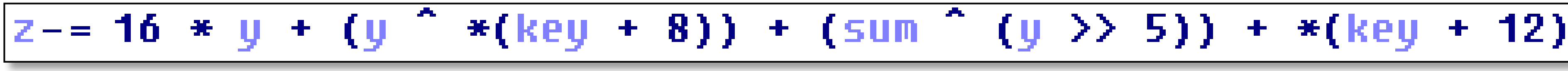
```
STARTED AT
2012-04-08 14:20:30.858000
** Crypto Algorithm Identification starting !**
8 input parameters
5 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption...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

WTF ?

Original TEA source code



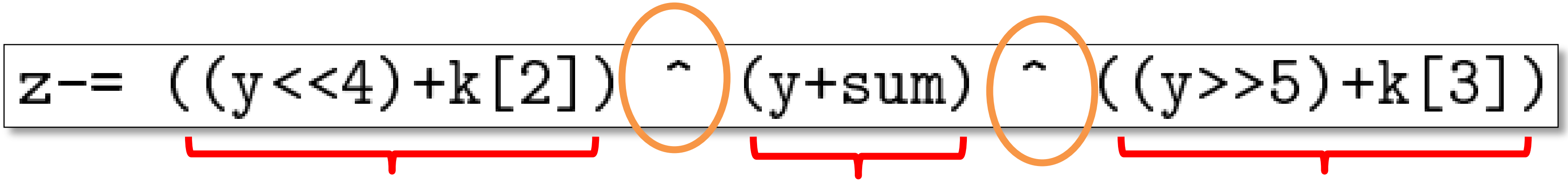
```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```



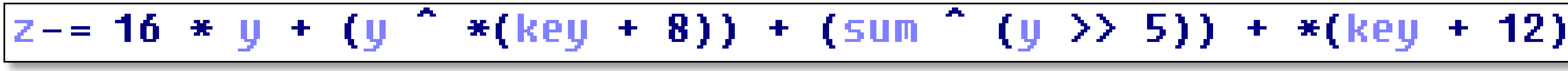
```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code



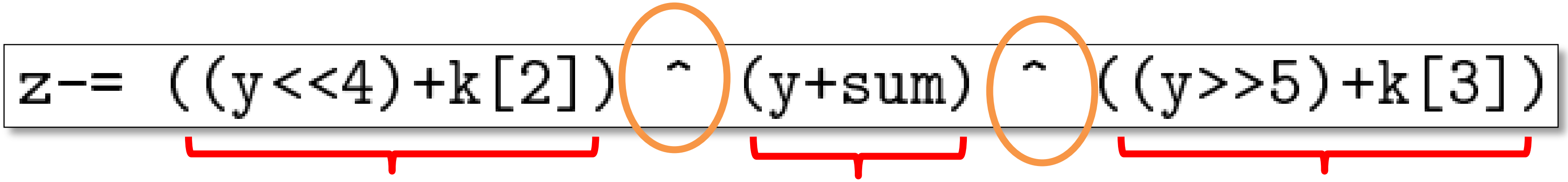
```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```



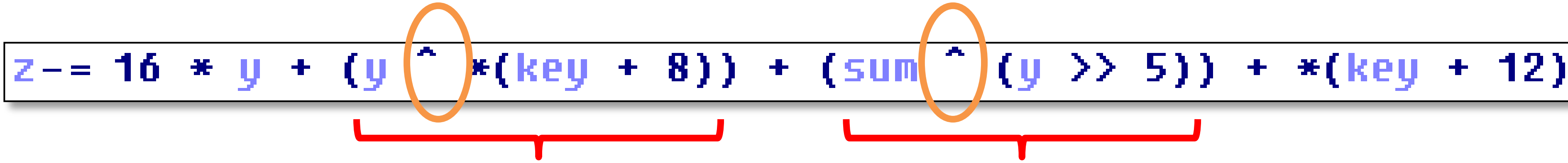
```
z-= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code



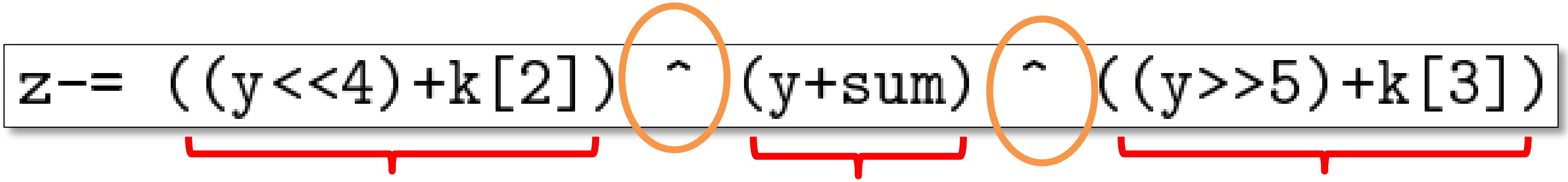
```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```



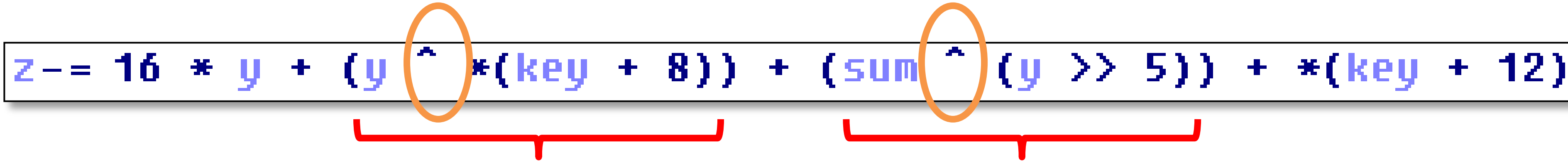
```
z -= 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```

Storm Worm implementation

Original TEA source code



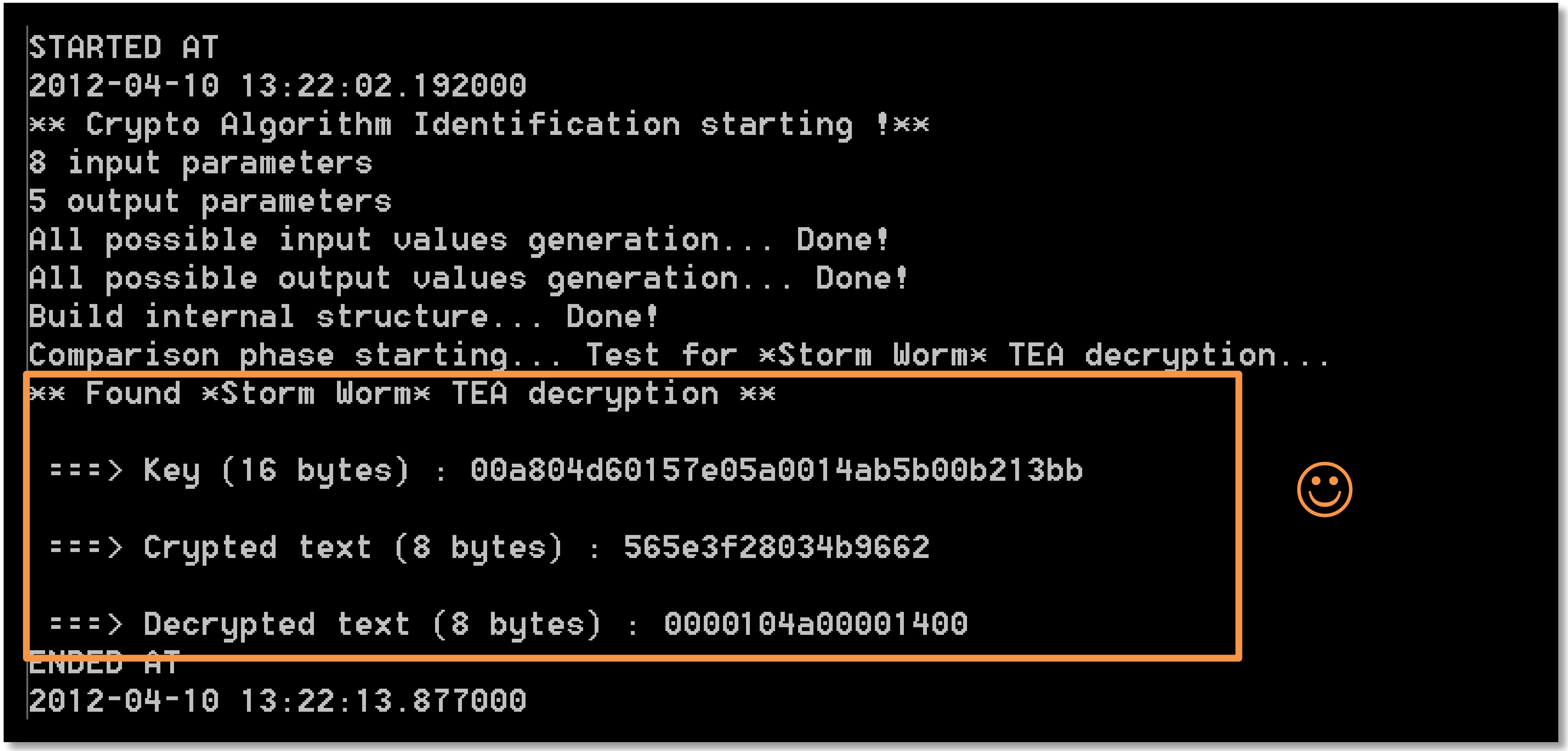
```
z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3])
```



```
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...

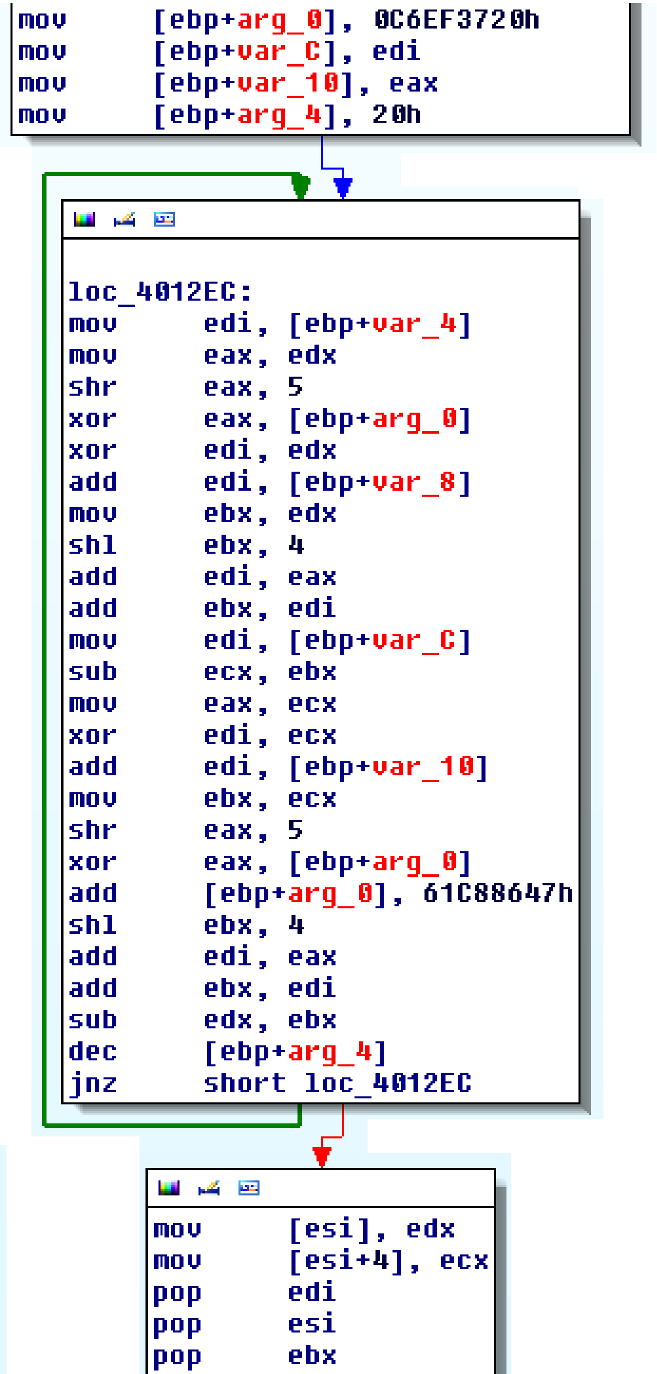
(this is *not* TEA)

Trojan.SilentBanker

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

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

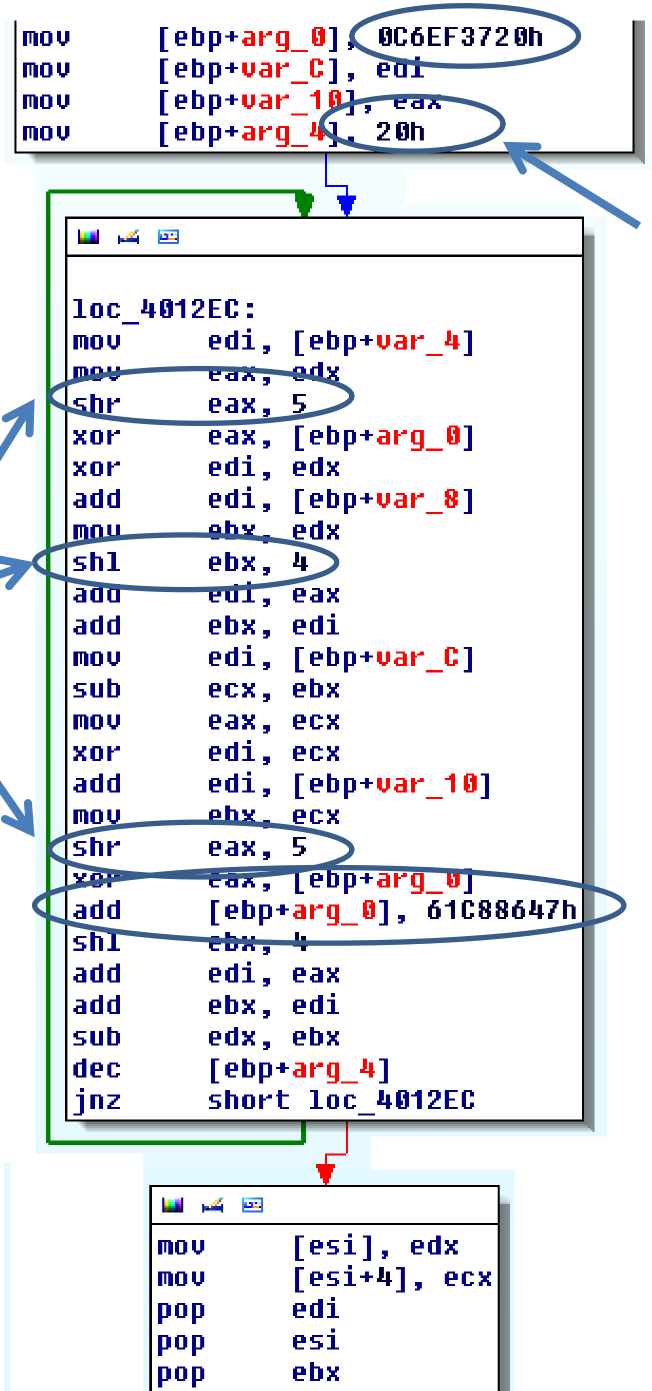
(sounds familiar, isn't it ?)



```
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
```

```
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
```

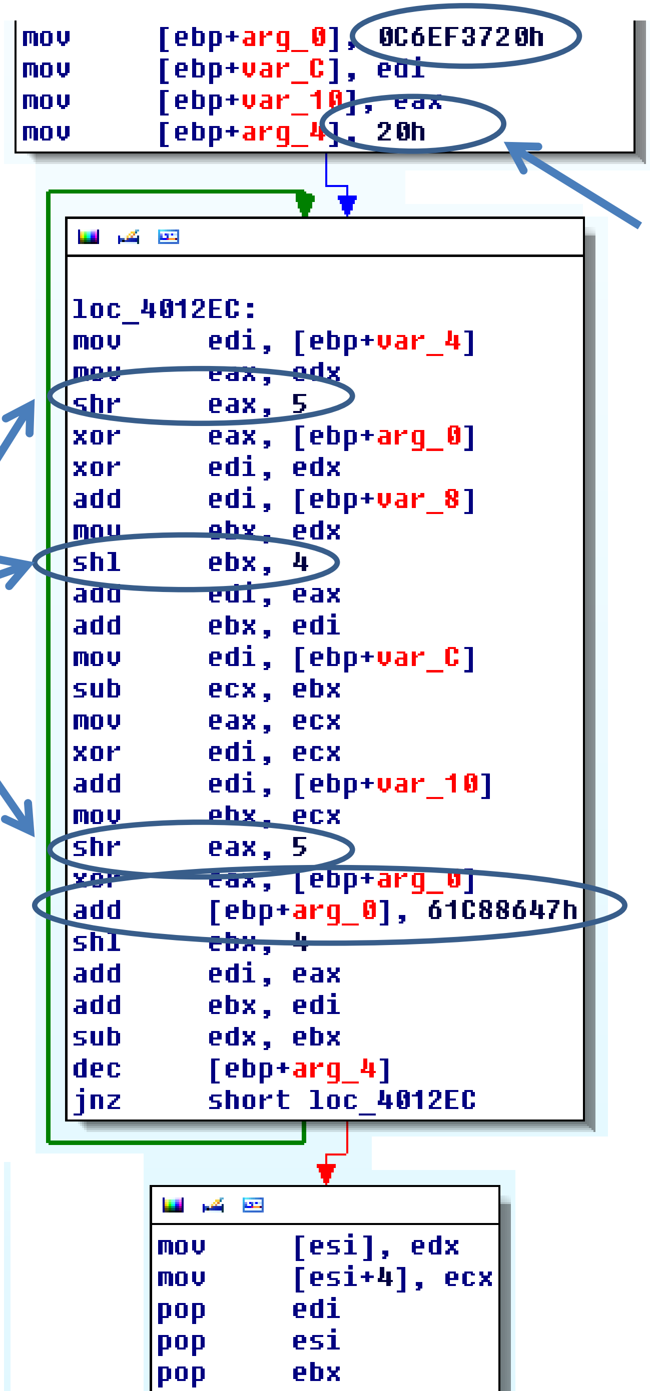
```
mov     [esi], edx
mov     [esi+4], ecx
pop     edi
pop     esi
pop     ebx
```

TEA classic constant
*(delta * round number*)

TEA round number

Classic TEA operations

= sub [ebp+arg_0], 0x9E3779B9



```
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
```

TEA classic constant
*(delta * round number)*

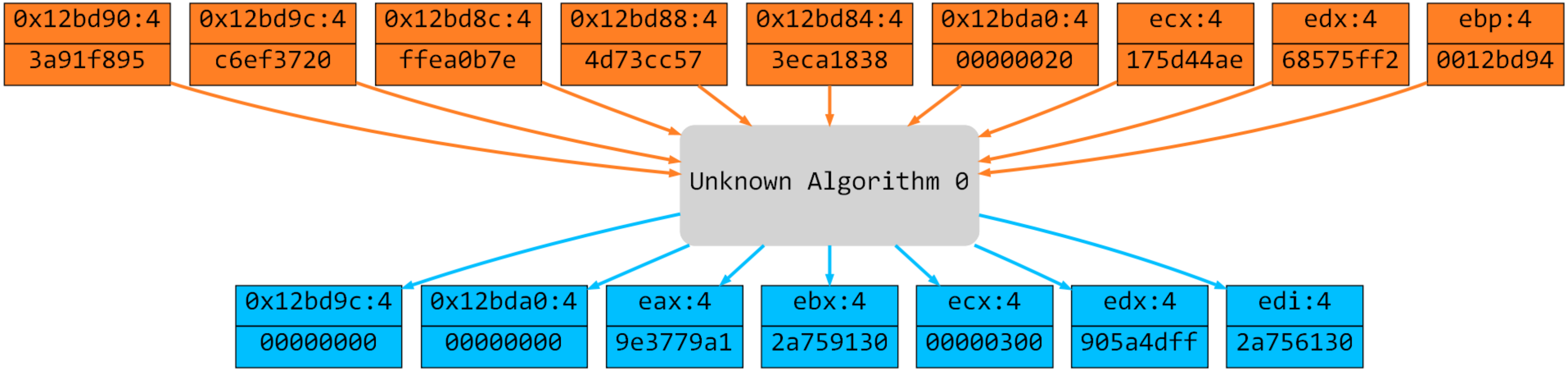
TEA round number

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:



0x12bd90:4
3a91f895

0x12bd9c:4
c6ef3720

0x12bd8c:4
ffea0b7e

0x12bd88:4
4d73cc57

0x12bd84:4
3eca1838

0x12bda0:4
00000020

ecx:4
175d44ae

edx:4
68575ff2

ebp:4
0012bd94

Unknown Algorithm 0

0x12bd9c:4
00000000

0x12bda0:4
00000000

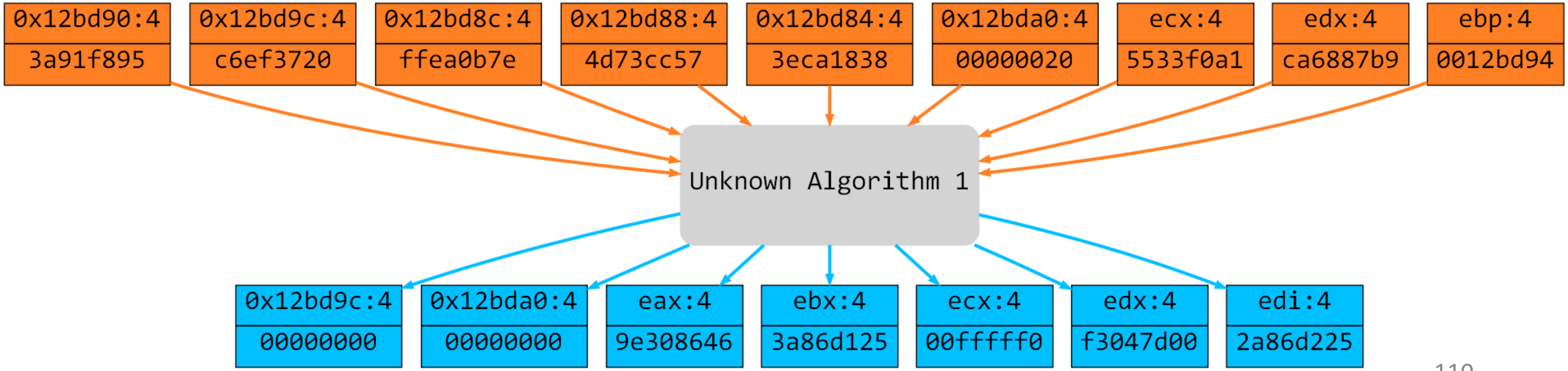
eax:4
9e3779a1

ebx:4
2a759130

ecx:4
00000300

edx:4
905a4dff

edi:4
2a756130



0x12bd90:4
3a91f895

0x12bd9c:4
c6ef3720

0x12bd8c:4
ffea0b7e

0x12bd88:4
4d73cc57

0x12bd84:4
3eca1838

0x12bda0:4
00000020

ecx:4
5533f0a1

edx:4
ca6887b9

ebp:4
0012bd94

Unknown Algorithm 1

0x12bd9c:4
00000000

0x12bda0:4
00000000

eax:4
9e308646

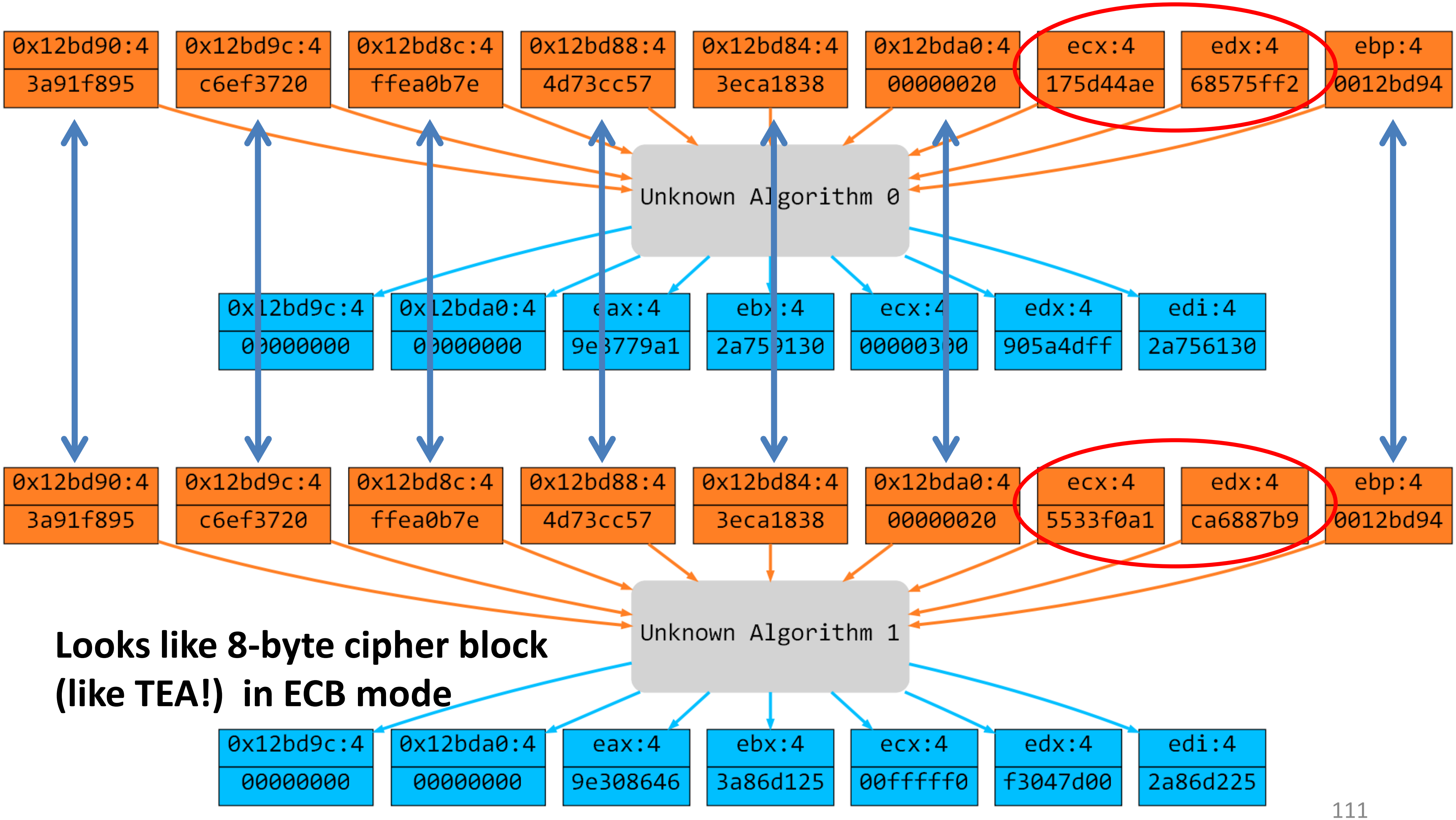
ebx:4
3a86d125

ecx:4
00fffff0

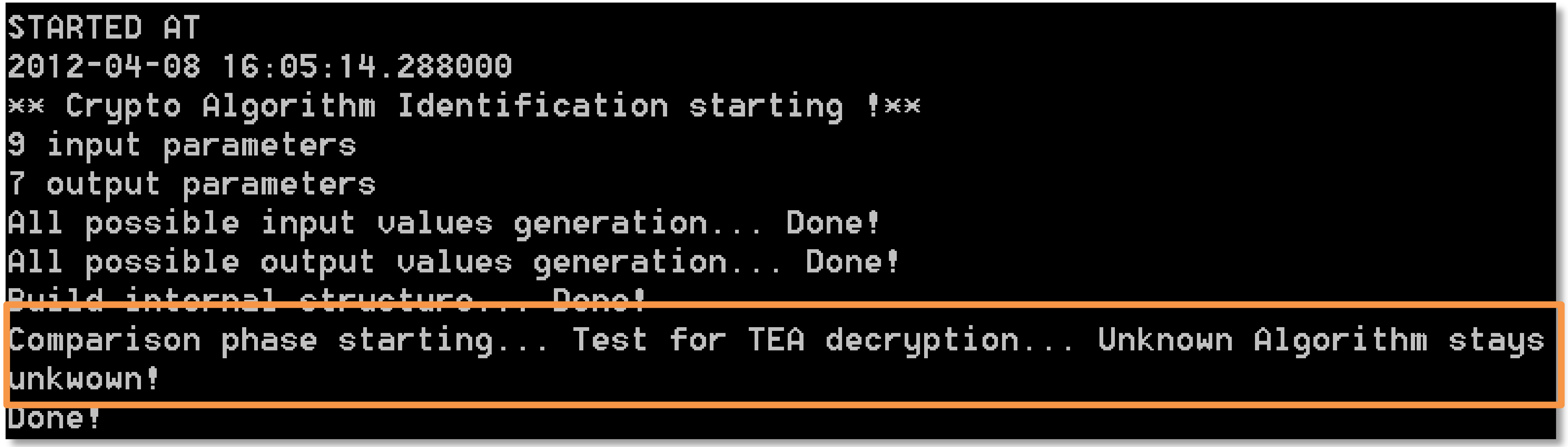
edx:4
f3047d00

edi:4
2a86d225

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

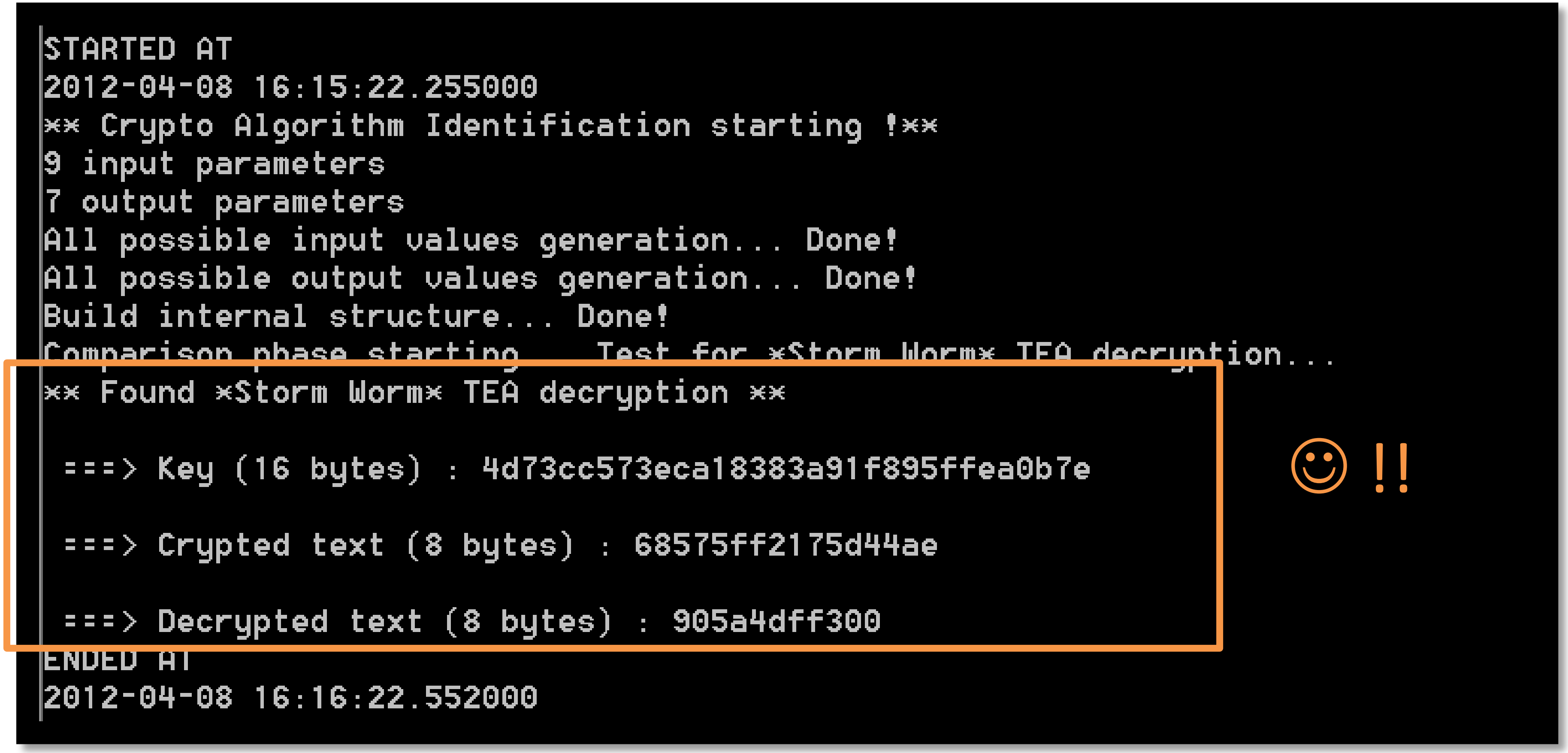


Looks like 8-byte cipher block (like TEA!) in ECB mode



```
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!

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.

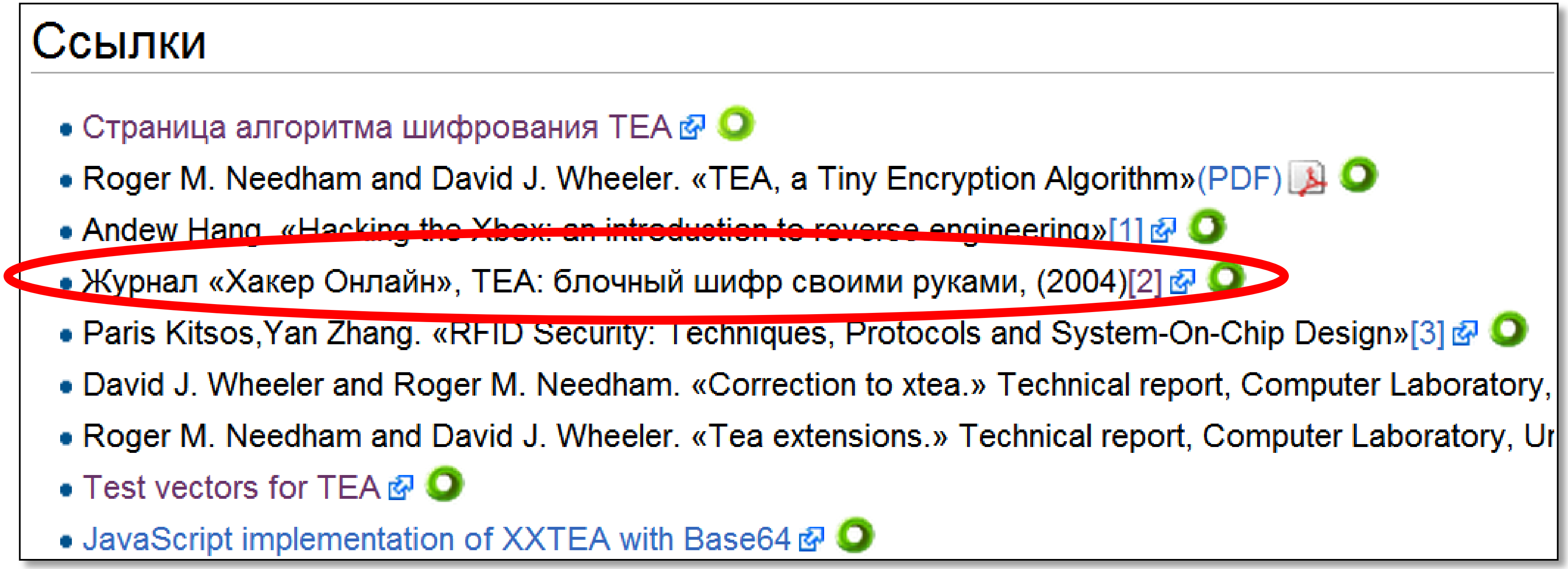
Contents [hide]

1 Properties
2 Versions
3 Reference code
4 See also
5 Notes
6 References
7 External links

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

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 г
голосован

Конкурсы с го
посетителей,
интерес вызы
лакомые приз

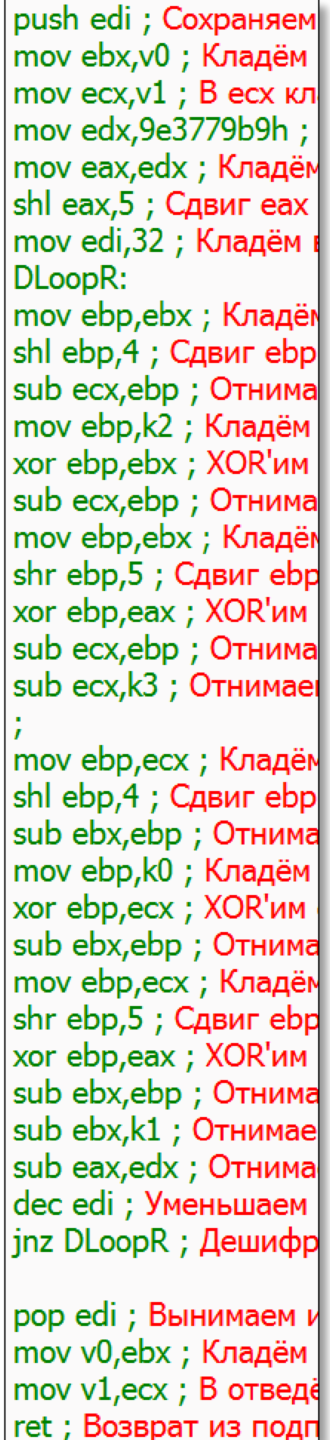
TEA: блочный шифр своими руками

Дата: **22.04.2004** | slon

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

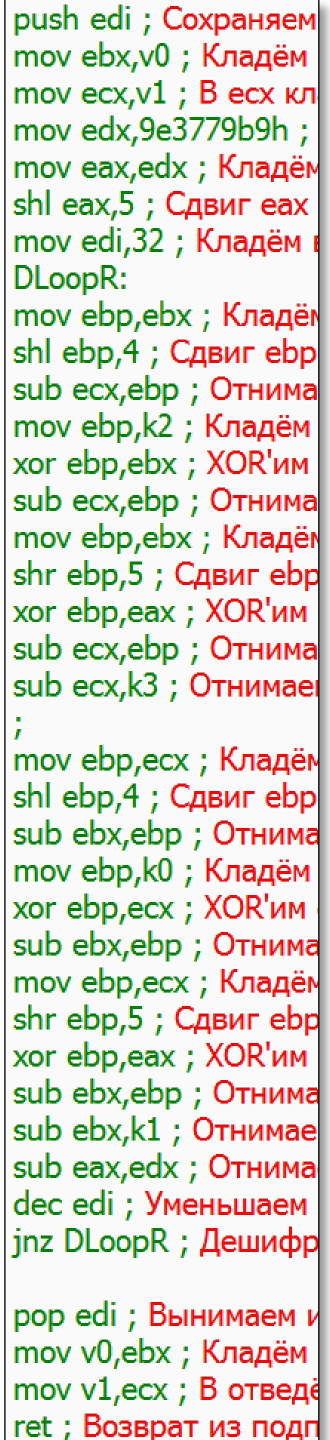
- шифрование для себя (чтобы ваши файлы никто, кроме вас не "читал")
- шифрование для других (чтобы ваши файлы "читал" только адресат)

Russian Website TEA source code



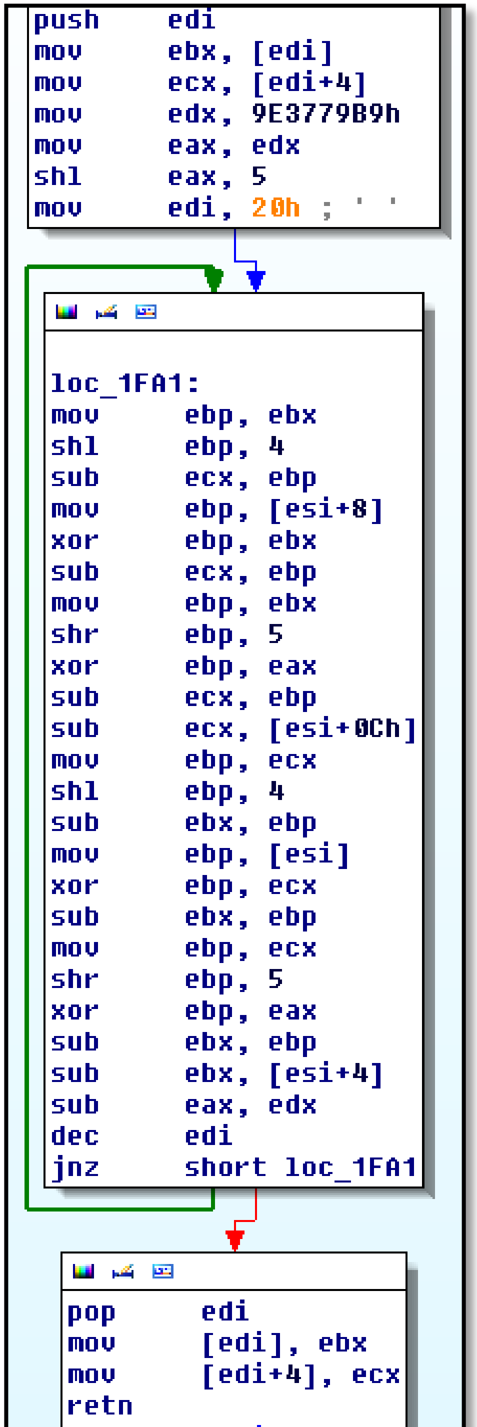
```
push edi ; Сохраняем
mov ebx,v0 ; Кладём
mov ecx,v1 ; В ecx кл
mov edx,9e3779b9h ;
mov eax,edx ; Кладё
shl eax,5 ; Сдвиг eax
mov edi,32 ; Кладём
DLoopR:
mov ebp,ebx ; Кладё
shl ebp,4 ; Сдвиг ebp
sub ecx,ebp ; Отнима
mov ebp,k2 ; Кладём
xor ebp,ebx ; XOR'им
sub ecx,ebp ; Отнима
mov ebp,ebx ; Кладё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

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
```

=



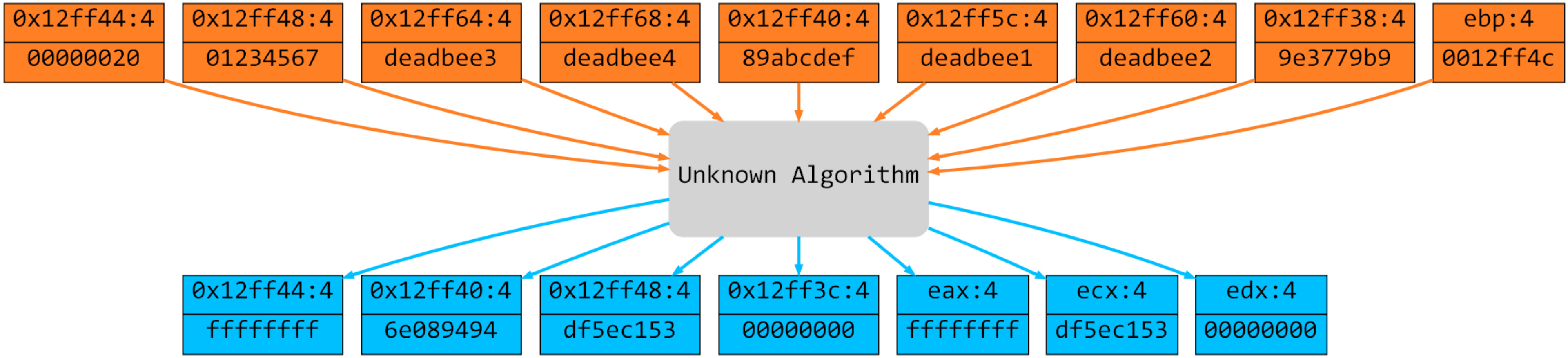
```
push    edi
mov     ebx, [edi]
mov     ecx, [edi+4]
mov     edx, 9E3779B9h
mov     eax, edx
shl     eax, 5
mov     edi, 20h ; ' '

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

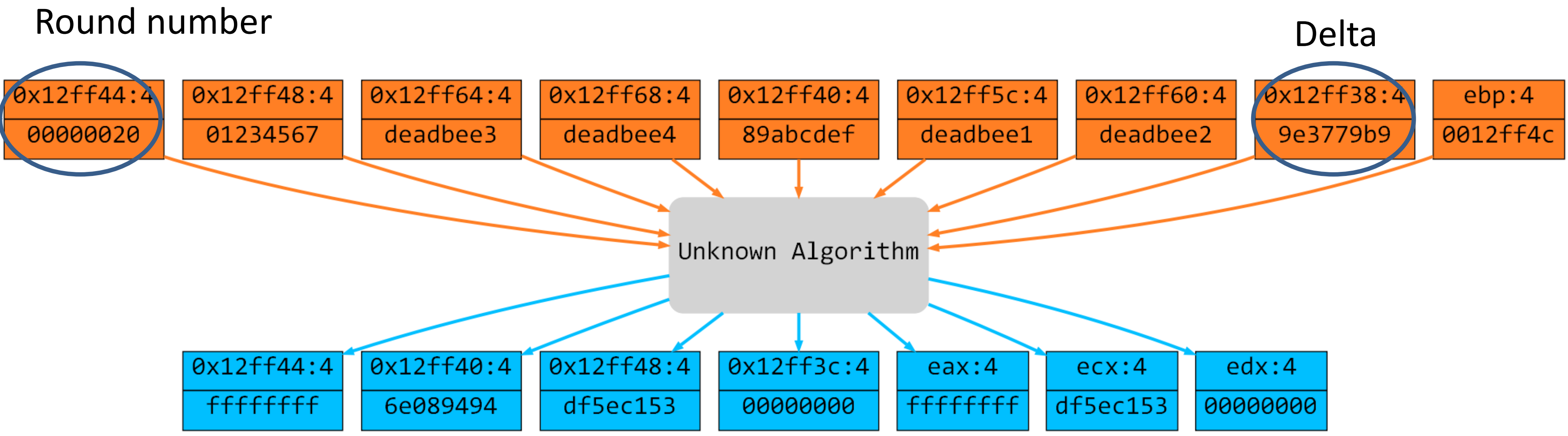
Modified TEA

MORE EXAMPLES!

Remember This ?



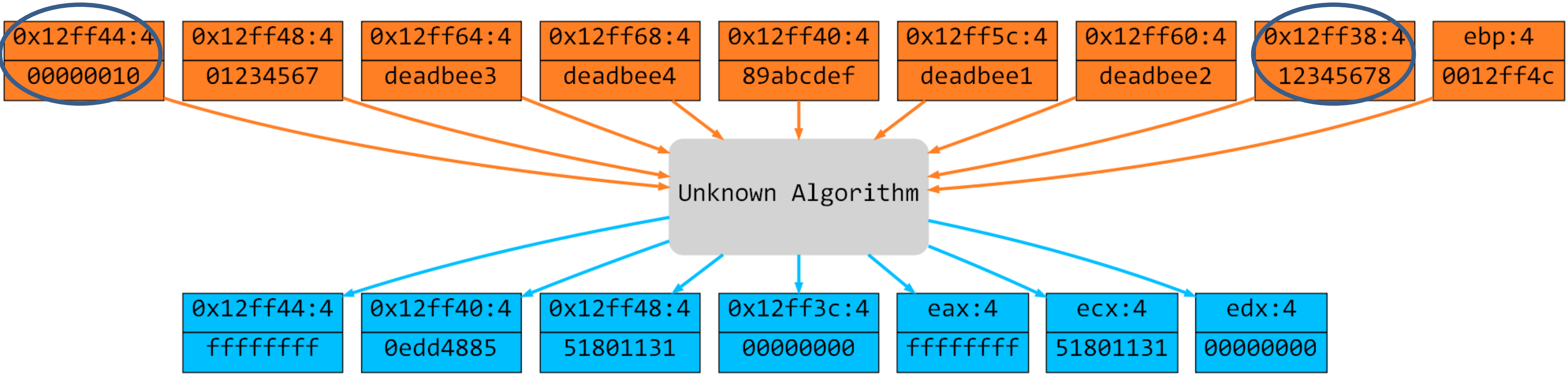
Remember This ?



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:

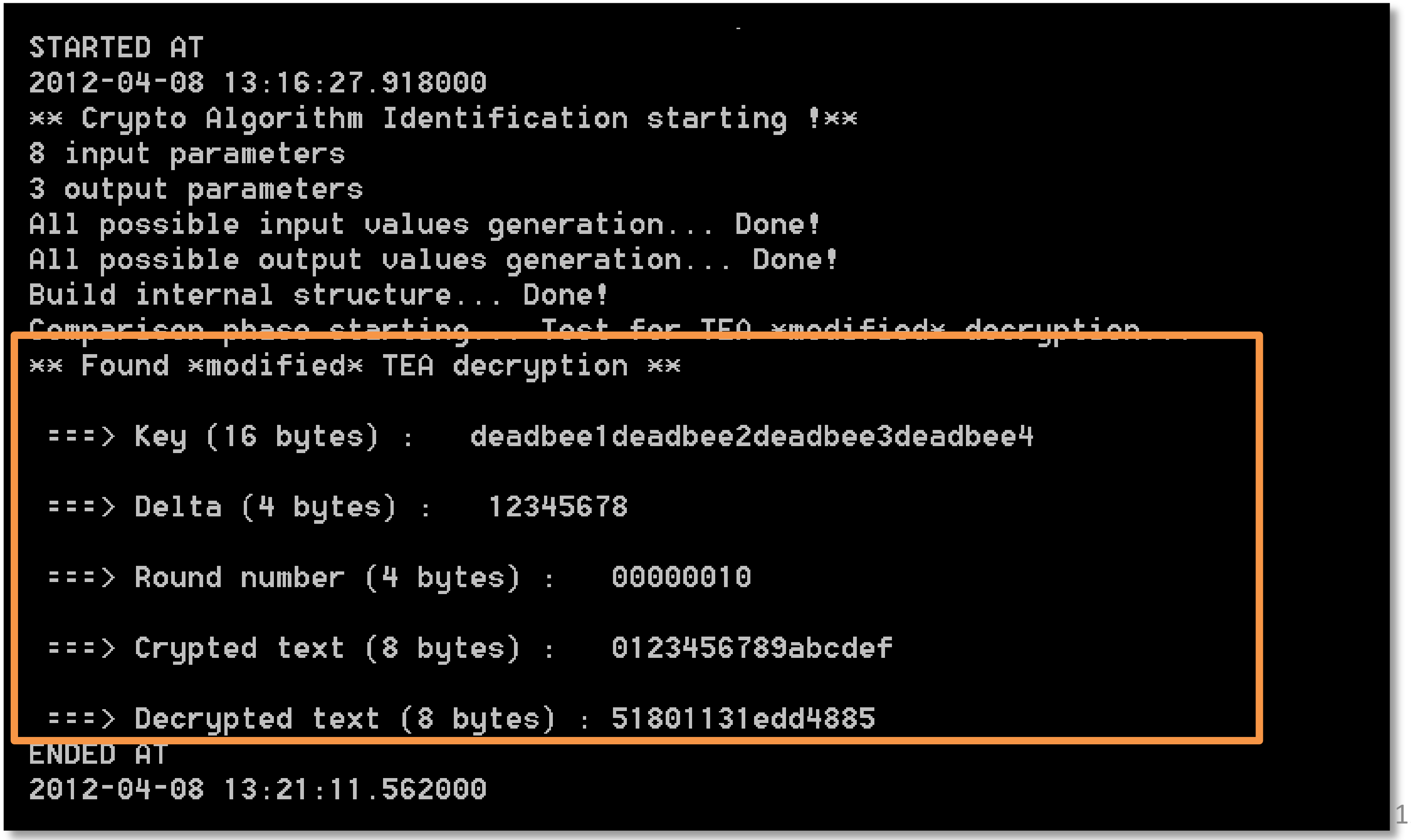
```
def  tea (input_text, key):
    ...
```

- 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
```

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

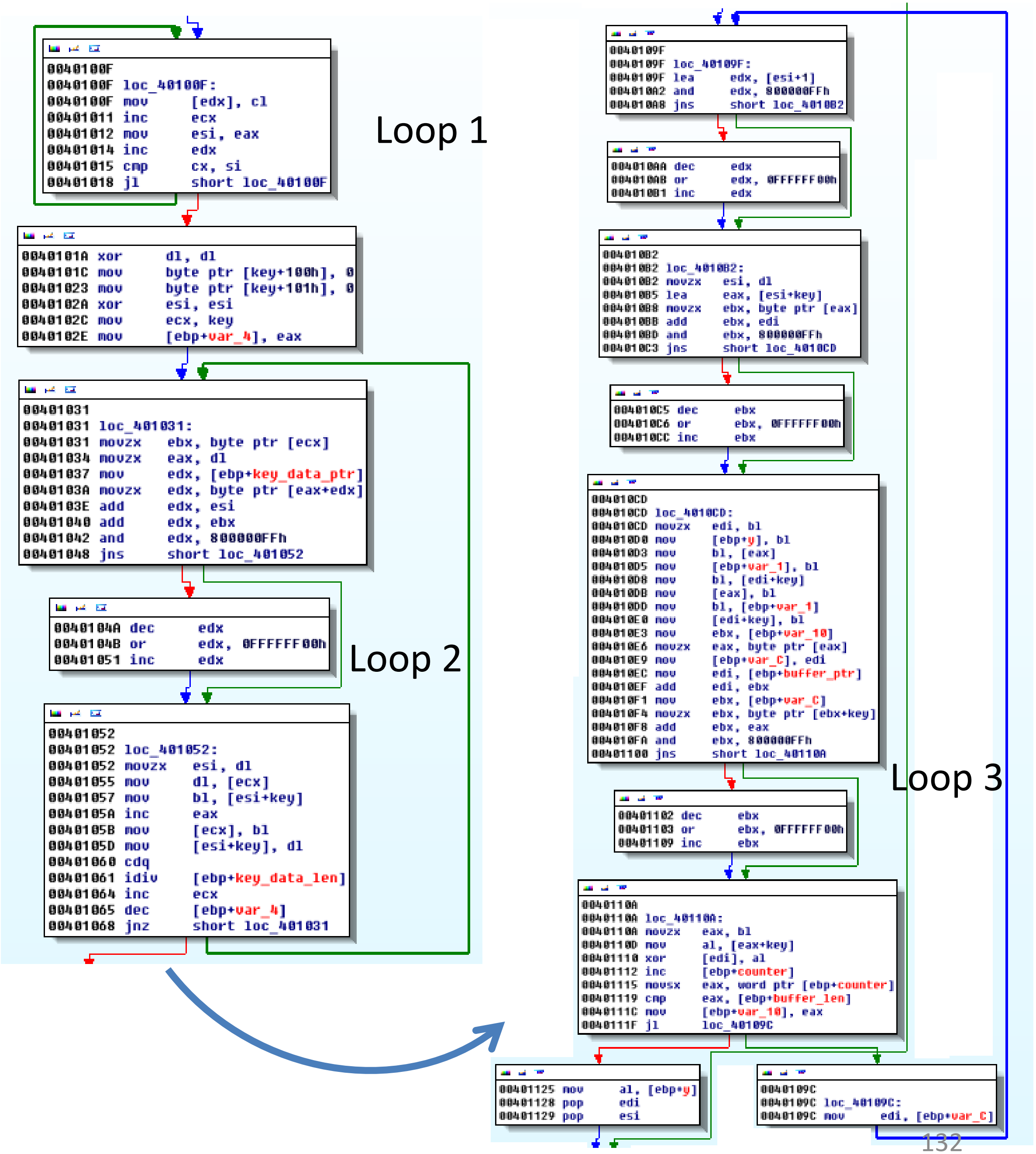
- Two loops *L1* and *L2* are in the same algorithm:
  - If *L1* started before *L2* in the trace.
  - If *L2* uses as input parameter an output parameter of *L1.*

(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…



```
0040100F
0040100F loc_40100F:
0040100F mov     [edx], cl
00401011 inc     ecx
00401012 mov     esi, eax
00401014 inc     edx
00401015 cmp     cx, si
00401018 jl      short loc_40100F
```

```
0040101A xor     dl, dl
0040101C mov     byte ptr [key+100h], 0
00401023 mov     byte ptr [key+101h], 0
0040102A xor     esi, esi
0040102C mov     ecx, key
0040102E mov     [ebp+var_4], eax
```

```
00401031
00401031 loc_401031:
00401031 movzx   ebx, byte ptr [ecx]
00401034 movzx   eax, dl
00401037 mov     edx, [ebp+key_data_ptr]
0040103A movzx   edx, byte ptr [eax+edx]
0040103E add     edx, esi
00401040 add     edx, ebx
00401042 and     edx, 800000FFh
00401048 jns     short loc_401052
```

```
0040104A dec     edx
0040104B or      edx, 0FFFFFF00h
00401051 inc     edx
```

```
00401052
00401052 loc_401052:
00401052 movzx   esi, dl
00401055 mov     dl, [ecx]
00401057 mov     bl, [esi+key]
0040105A inc     eax
0040105B mov     [ecx], bl
0040105D mov     [esi+key], dl
00401060 cdq
00401061 idiv    [ebp+key_data_len]
00401064 inc     ecx
00401065 dec     [ebp+var_4]
00401068 jnz     short loc_401031
```

```
0040109F
0040109F loc_40109F:
0040109F lea     edx, [esi+1]
004010A2 and     edx, 800000FFh
004010A8 jns     short loc_4010B2
```

```
004010AA dec     edx
004010AB or      edx, 0FFFFFF00h
004010B1 inc     edx
```

```
004010B2
004010B2 loc_4010B2:
004010B2 movzx   esi, dl
004010B5 lea     eax, [esi+key]
004010B8 movzx   ebx, byte ptr [eax]
004010BB add     ebx, edi
004010BD and     ebx, 800000FFh
004010C3 jns     short loc_4010CD
```

```
004010C5 dec     ebx
004010C6 or      ebx, 0FFFFFF00h
004010CC inc     ebx
```

```
004010CD
004010CD loc_4010CD:
004010CD movzx   edi, bl
004010D0 mov     [ebp+y], bl
004010D3 mov     bl, [eax]
004010D5 mov     [ebp+var_1], bl
004010D8 mov     bl, [edi+key]
004010DB mov     [eax], bl
004010DD mov     bl, [ebp+var_1]
004010E0 mov     [edi+key], bl
004010E3 mov     ebx, [ebp+var_10]
004010E6 movzx   eax, byte ptr [eax]
004010E9 mov     [ebp+var_C], edi
004010EC mov     edi, [ebp+buffer_ptr]
004010EF add     edi, ebx
004010F1 mov     ebx, [ebp+var_C]
004010F4 movzx   ebx, byte ptr [ebx+key]
004010F8 add     ebx, eax
004010FA and     ebx, 800000FFh
00401100 jns     short loc_40110A
```

```
00401102 dec     ebx
00401103 or      ebx, 0FFFFFF00h
00401109 inc     ebx
```

```
0040110A
0040110A loc_40110A:
0040110A movzx   eax, bl
0040110D mov     al, [eax+key]
00401110 xor     [edi], al
00401112 inc     [ebp+counter]
00401115 movsx   eax, word ptr [ebp+counter]
00401119 cmp     eax, [ebp+buffer_len]
0040111C mov     [ebp+var_10], eax
0040111F jl      loc_40109C
```

```
00401125 mov     al, [ebp+y]
00401128 pop     edi
00401129 pop     esi
```

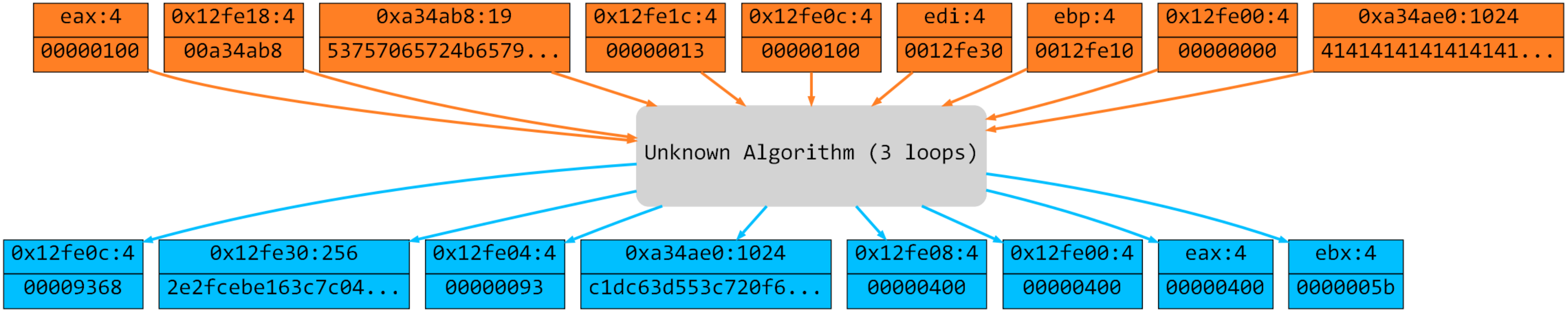
```
0040109C
0040109C loc_40109C:
0040109C mov     edi, [ebp+var_C]
```

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	∅
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	∅

Let’s try our tool...

- We extracted from our toy program:



eax:4
00000100

0x12fe18:4
00a34ab8

0xa34ab8:19
53757065724b6579...

0x12fe1c:4
00000013

0x12fe0c:4
00000100

edi:4
0012fe30

ebp:4
0012fe10

0x12fe00:4
00000000

0xa34ae0:1024
4141414141414141...

Unknown Algorithm (3 loops)

0x12fe0c:4
00009368

0x12fe30:256
2e2fcebe163c7c04...

0x12fe04:4
00000093

0xa34ae0:1024
c1dc63d553c720f6...

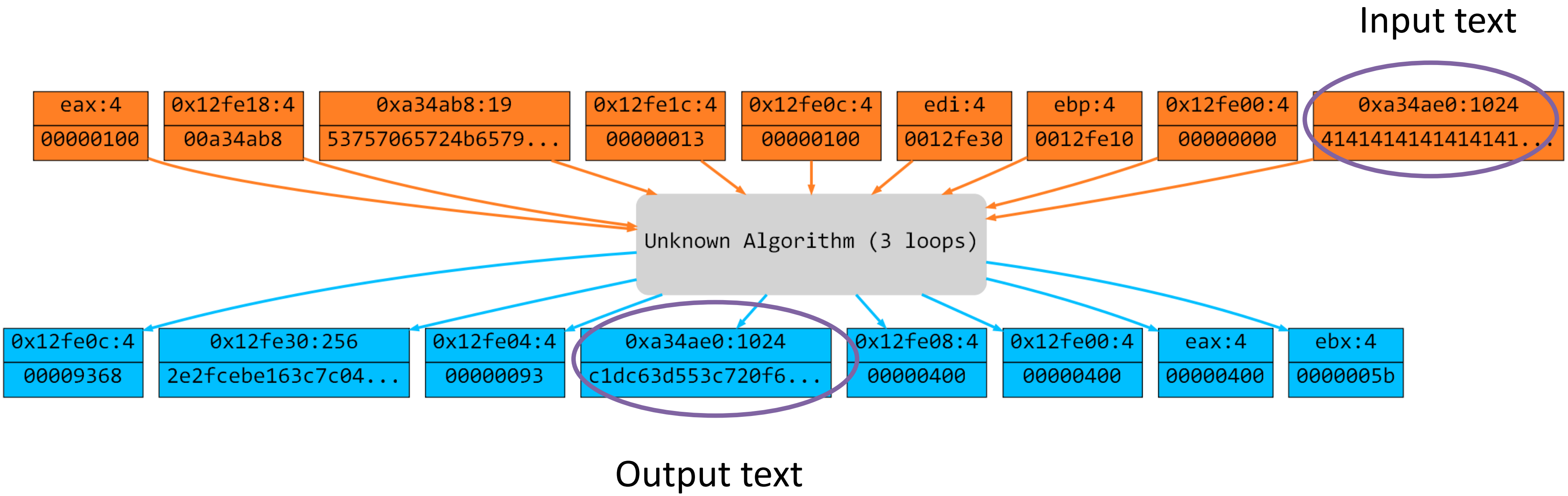
0x12fe08:4
00000400

0x12fe00:4
00000400

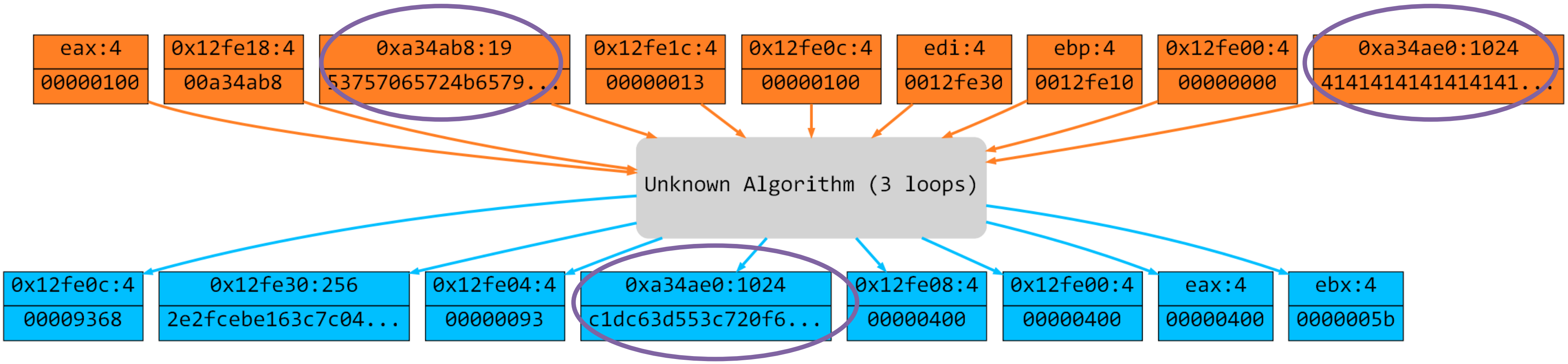
eax:4
00000400

ebx:4
0000005b

- We extracted from our toy program:



- We extracted from our toy program:



Key

Input text

eax:4
00000100

0x12fe18:4
00a34ab8

0xa34ab8:19
53757065724b6579...

0x12fe1c:4
00000013

0x12fe0c:4
00000100

edi:4
0012fe30

ebp:4
0012fe10

0x12fe00:4
00000000

0xa34ae0:1024
4141414141414141...

Unknown Algorithm (3 loops)

0x12fe0c:4
00009368

0x12fe30:256
2e2fcebe163c7c04...

0x12fe04:4
00000093

0xa34ae0:1024
c1dc63d553c720f6...

0x12fe08:4
00000400

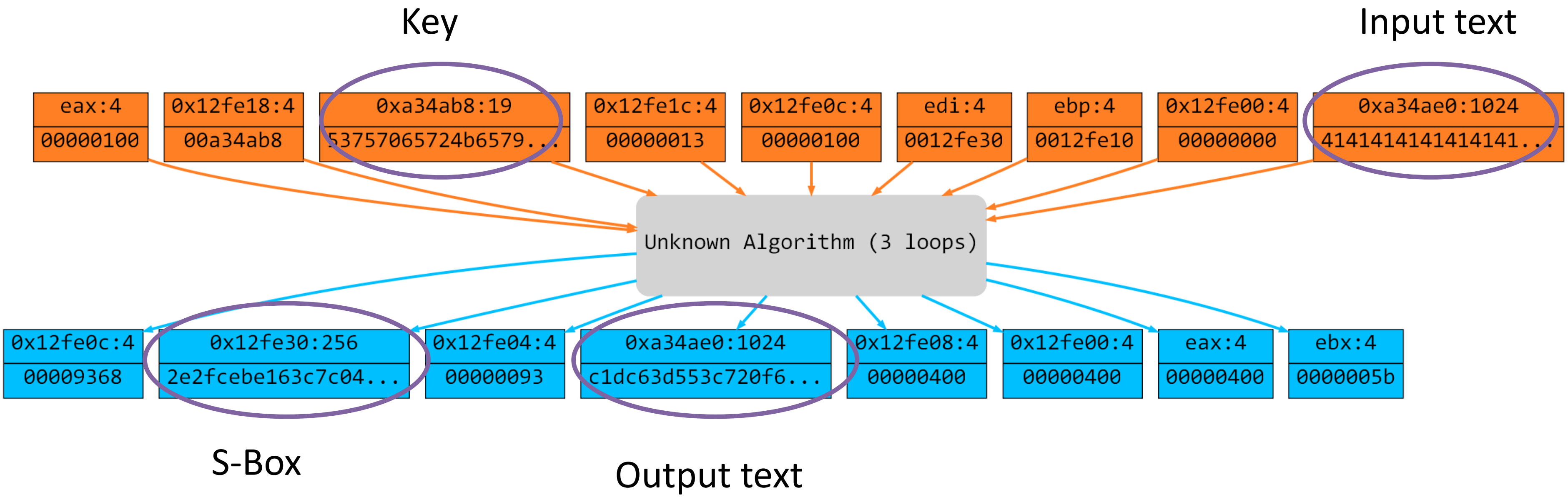
0x12fe00:4
00000400

eax:4
00000400

ebx:4
0000005b

Output text

- We extracted from our toy program:



Key

Input text

eax:4 00000100

0x12fe18:4 00a34ab8

0xa34ab8:19 53757065724b6579...

0x12fe1c:4 00000013

0x12fe0c:4 00000100

edi:4 0012fe30

ebp:4 0012fe10

0x12fe00:4 00000000

0xa34ae0:1024 4141414141414141...

Unknown Algorithm (3 loops)

0x12fe0c:4 00009368

0x12fe30:256 2e2fcebe163c7c04...

0x12fe04:4 00000093

0xa34ae0:1024 c1dc63d553c720f6...

0x12fe08:4 00000400

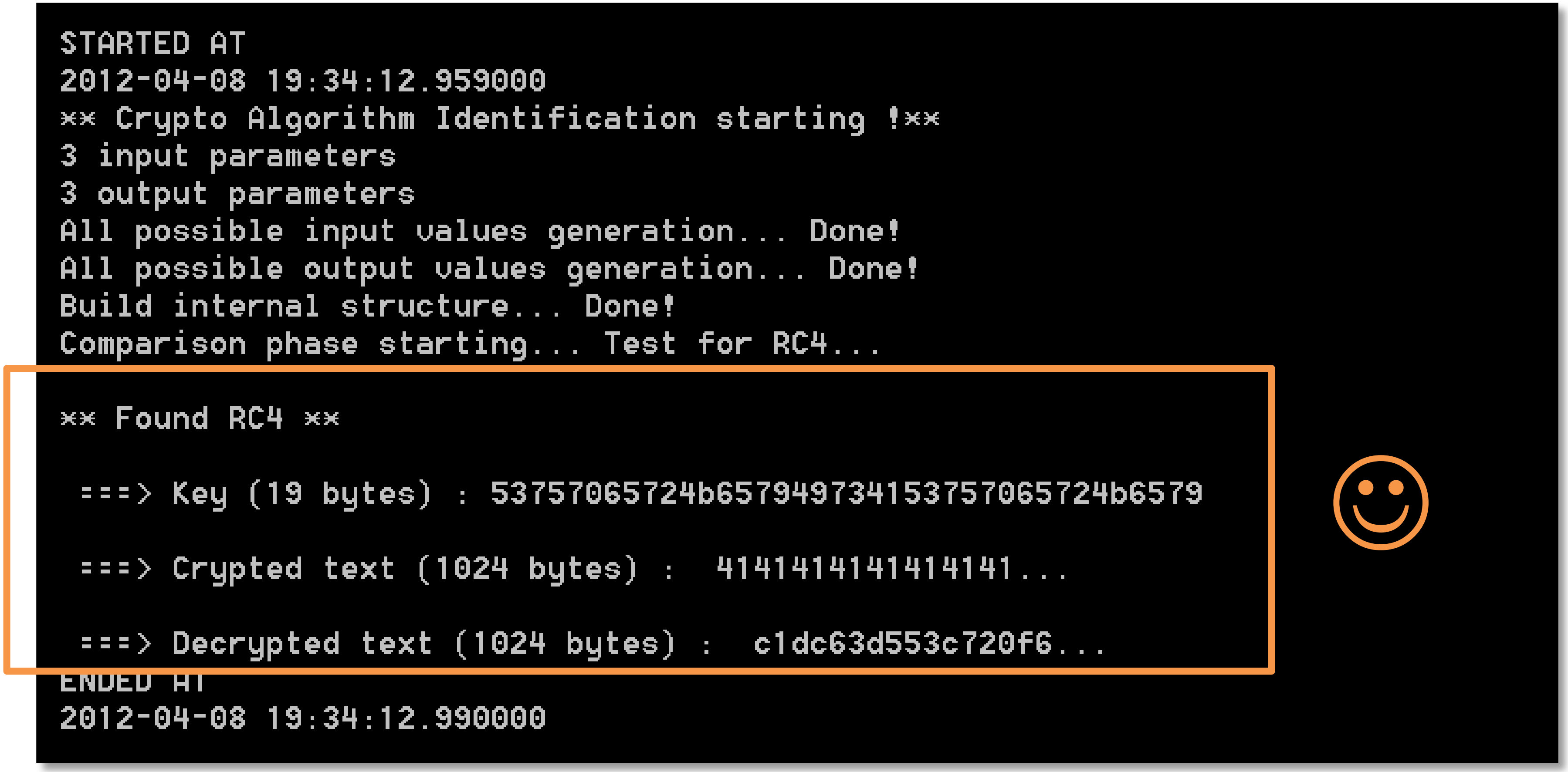
0x12fe00:4 00000400

eax:4 00000400

ebx:4 0000005b

S-Box

Output text



```
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
```

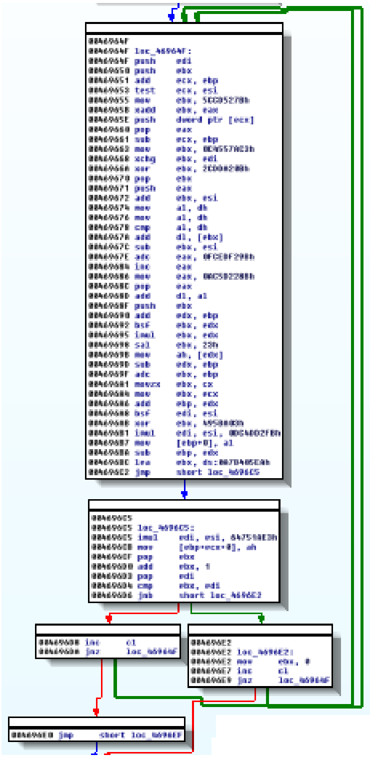
Sality

- Several internet references about the use of RC4 in the Sality packer.

- Let's take a look...

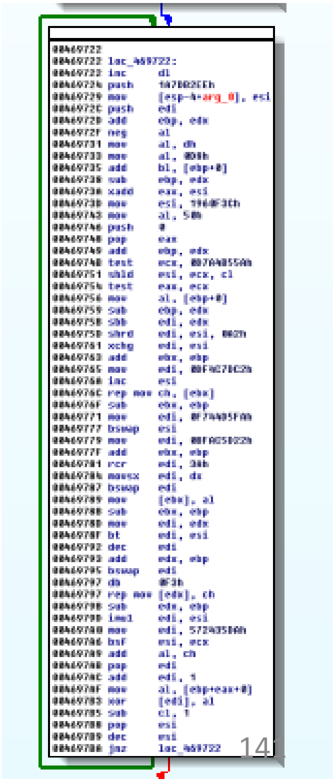
(suspense...)

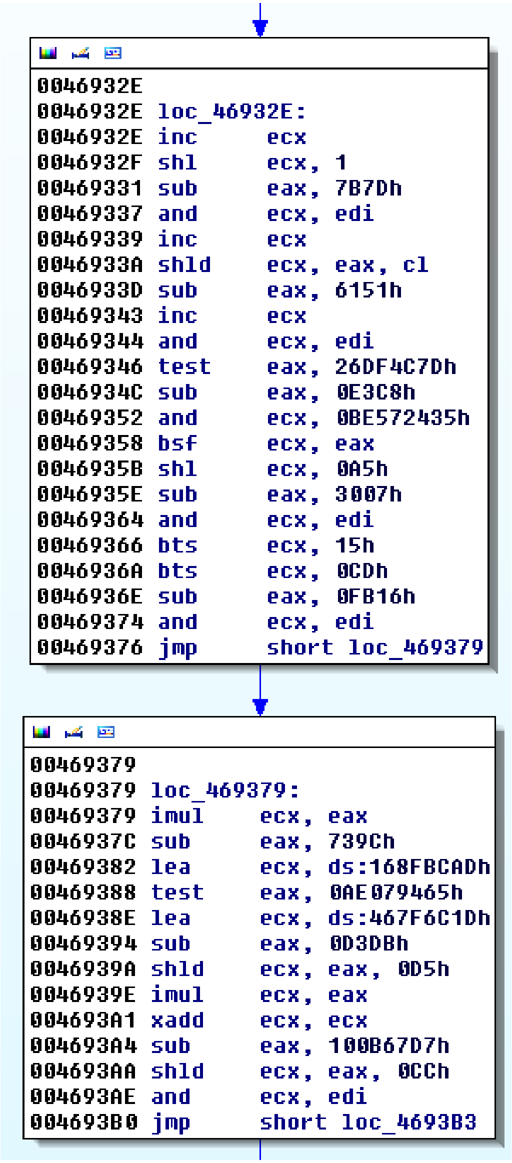
Loop 1



Loop 2

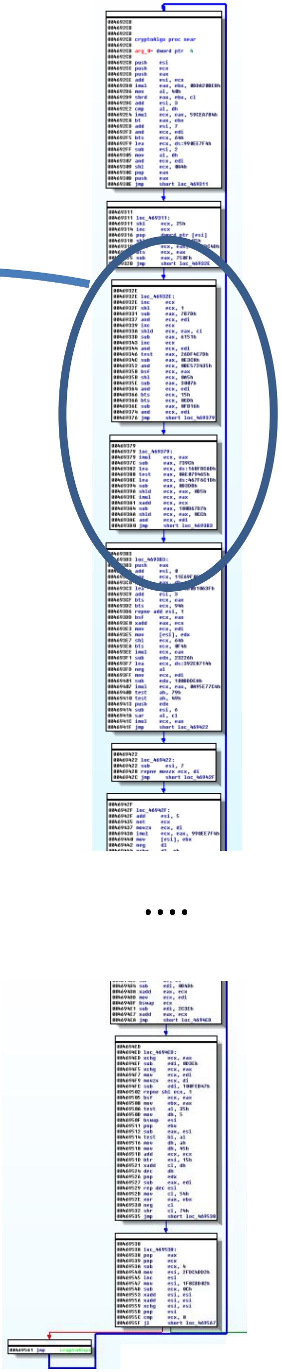
Loop 3





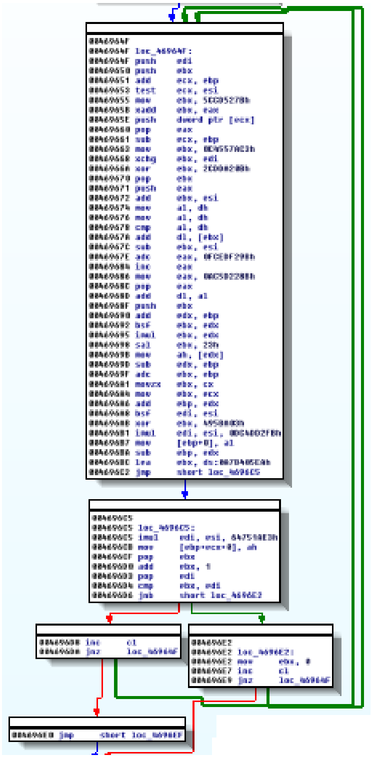
```
0046932E
0046932E loc_46932E:
0046932E inc     ecx
0046932F shl     ecx, 1
00469331 sub     eax, 7B7Dh
00469337 and     ecx, edi
00469339 inc     ecx
0046933A shld    ecx, eax, cl
0046933D sub     eax, 6151h
00469343 inc     ecx
00469344 and     ecx, edi
00469346 test    eax, 26DF4C7Dh
0046934C sub     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
```



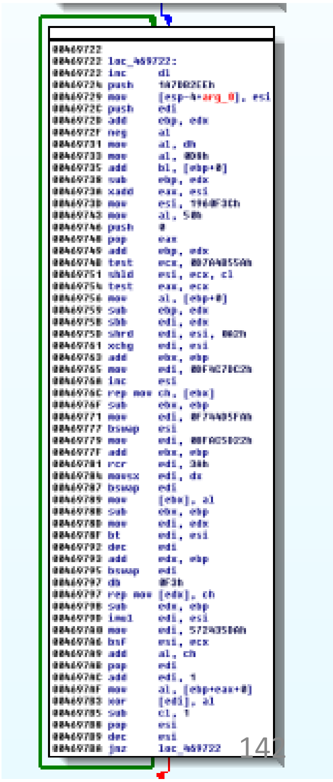
Loop 1

....

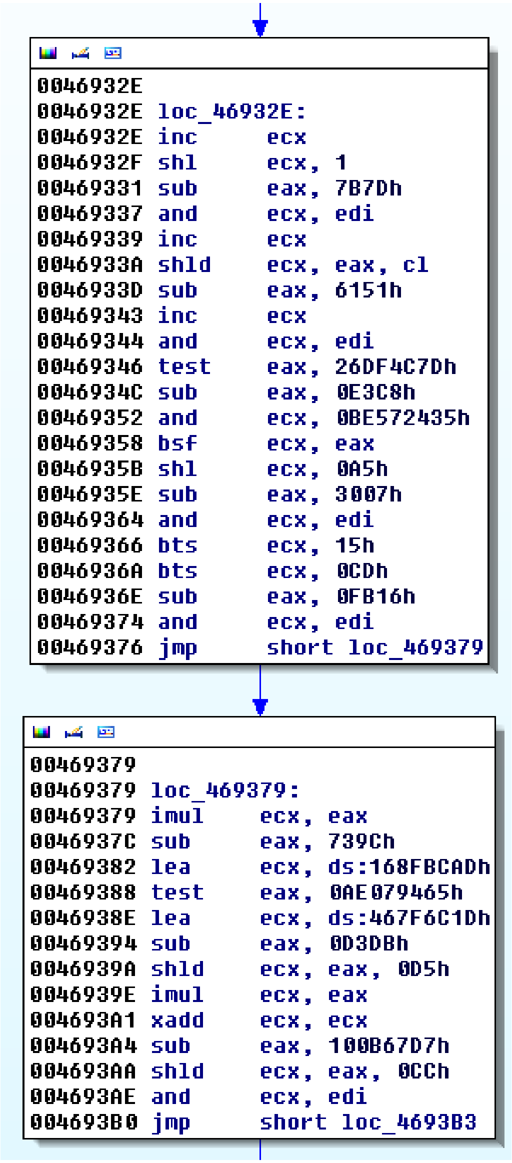


Loop 2

Loop 3

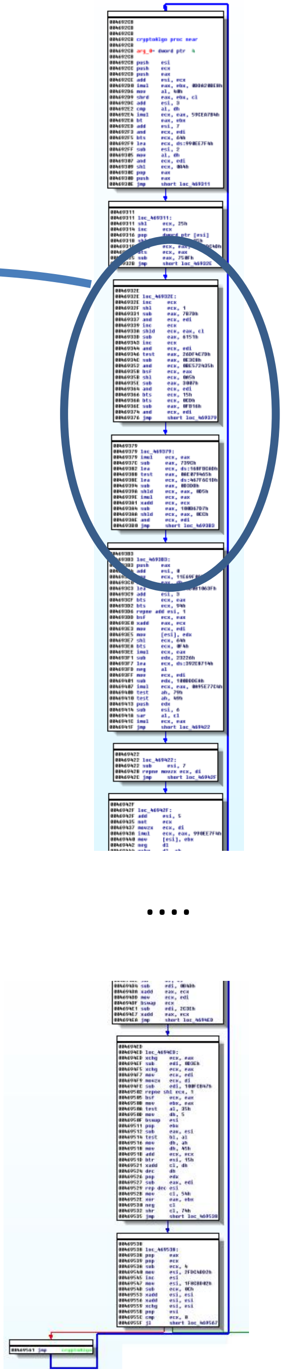


Hmpf.. Let’s try!



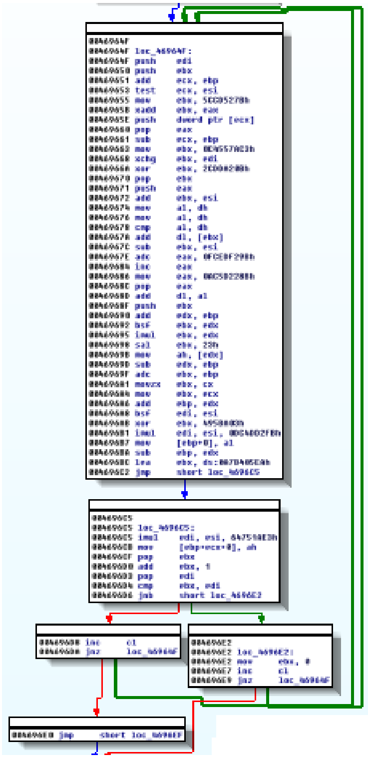
```
0046932E
0046932E loc_46932E:
0046932E inc     ecx
0046932F shl     ecx, 1
00469331 sub     eax, 7B7Dh
00469337 and     ecx, edi
00469339 inc     ecx
0046933A shld    ecx, eax, cl
0046933D sub     eax, 6151h
00469343 inc     ecx
00469344 and     ecx, edi
00469346 test    eax, 26DF4C7Dh
0046934C sub     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
```

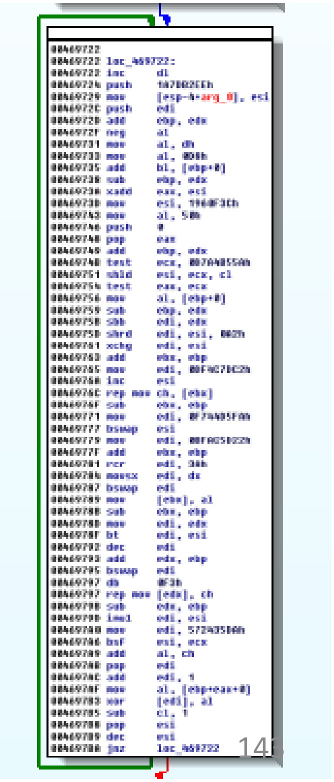


....

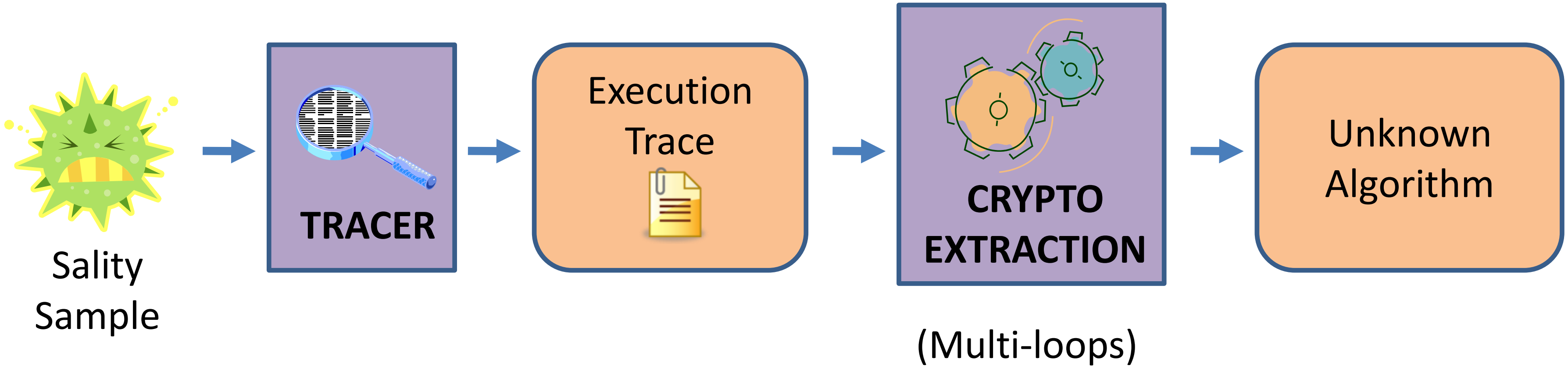
Loop 1



Loop 2



Loop 3



Sality Sample

TRACER

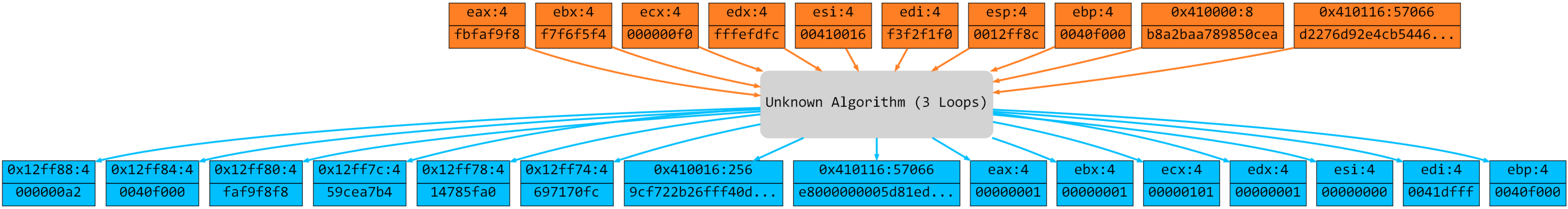
Execution Trace

CRYPTO EXTRACTION

(Multi-loops)

Unknown Algorithm

For the previous 3 loops, we extracted one algorithm:

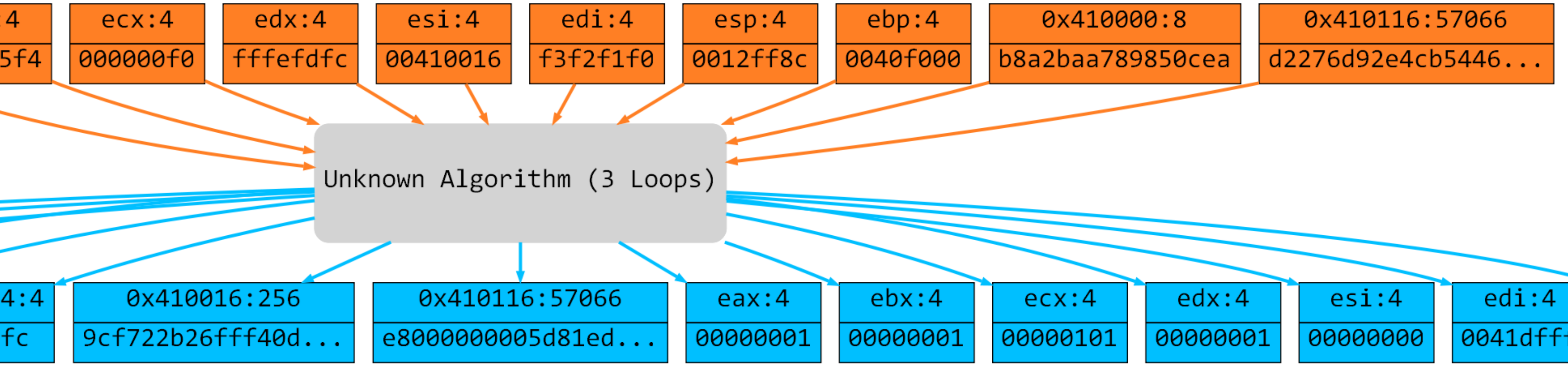


eax:4 fbfaf9f8
ebx:4 f7f6f5f4
ecx:4 000000f0
edx:4 fffefdfc
esi:4 00410016
edi:4 f3f2f1f0
esp:4 0012ff8c
ebp:4 0040f000
0x410000:8 b8a2baa789850cea
0x410116:57066 d2276d92e4cb5446...

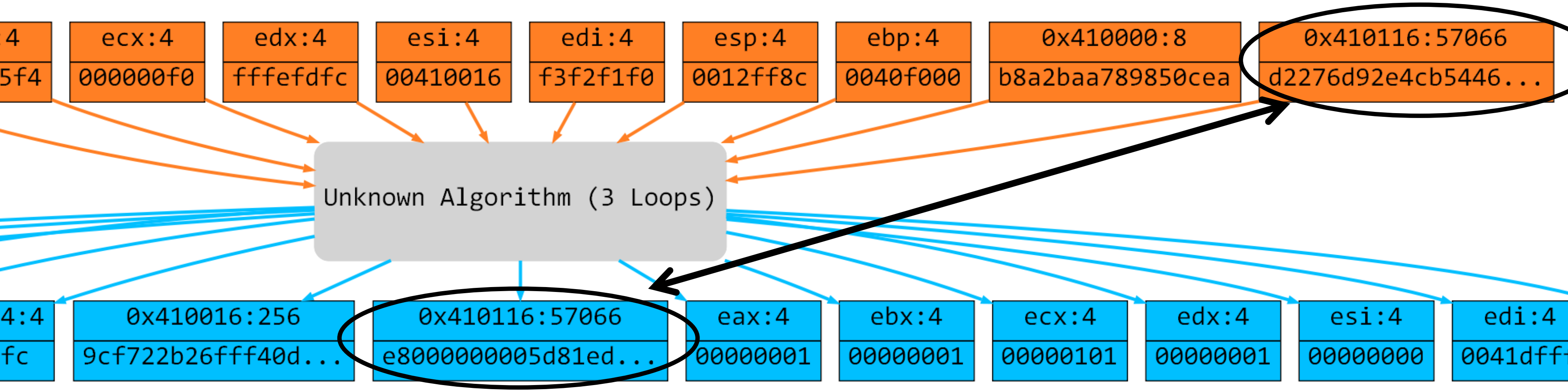
Unknown Algorithm (3 Loops)

0x12ff88:4 000000a2
0x12ff84:4 0040f000
0x12ff80:4 faf9f8f8
0x12ff7c:4 59cea7b4
0x12ff78:4 14785fa0
0x12ff74:4 697170fc
0x410016:256 9cf722b26fff40d...
0x410116:57066 e8000000005d81ed...
eax:4 00000001
ebx:4 00000001
ecx:4 00000101
edx:4 00000001
esi:4 00000000
edi:4 0041dfff
ebp:4 0040f000

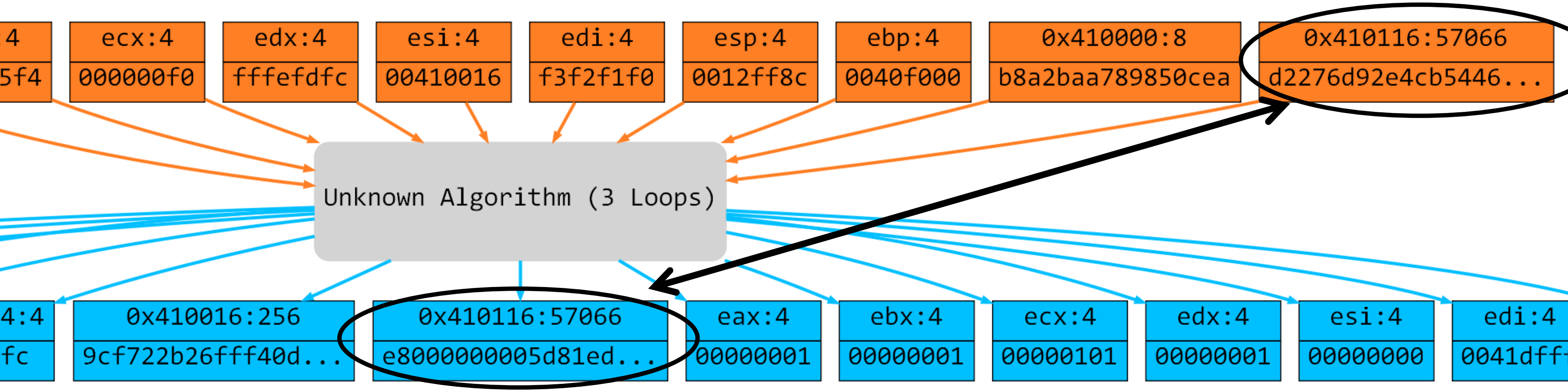
For the previous 3 loops, we extracted one algorithm:



For the previous 3 loops, we extracted one algorithm:

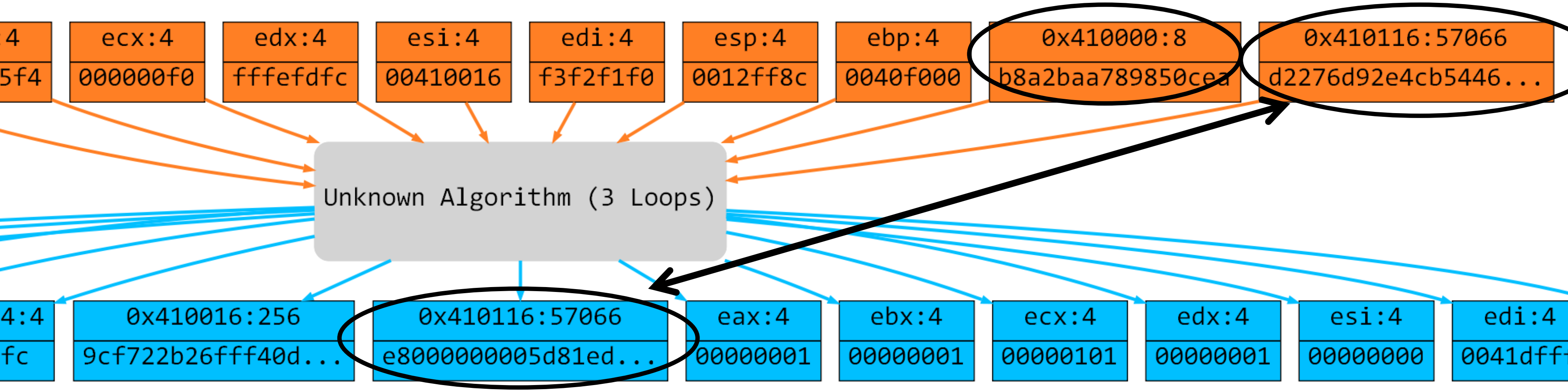


For the previous 3 loops, we extracted one algorithm:



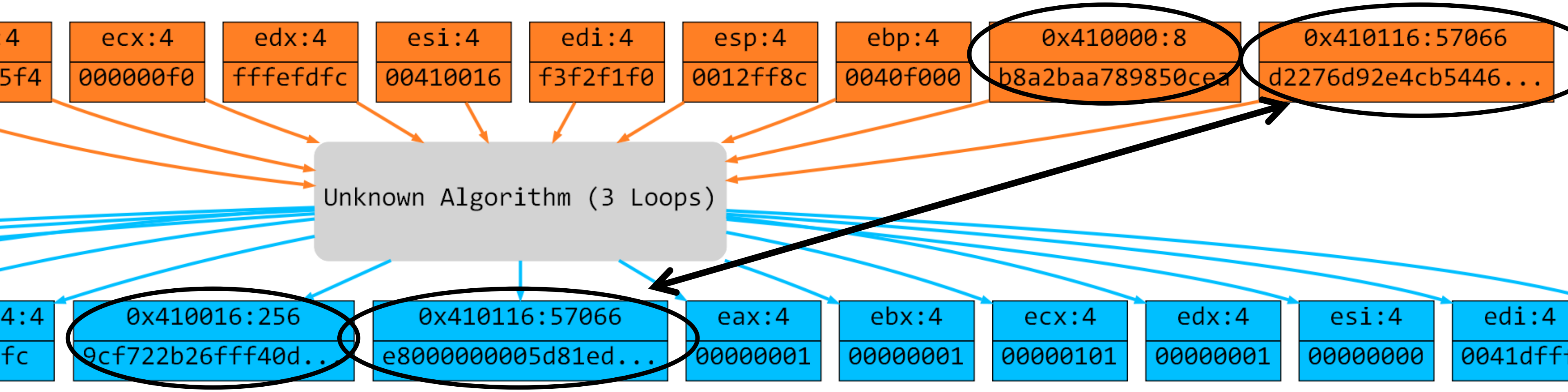
X86 ExecutableCode!

For the previous 3 loops, we extracted one algorithm:

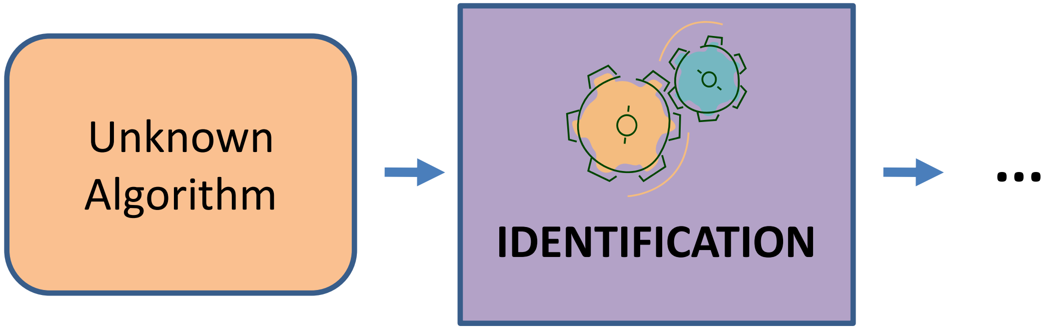


X86 ExecutableCode!

For the previous 3 loops, we extracted one algorithm:



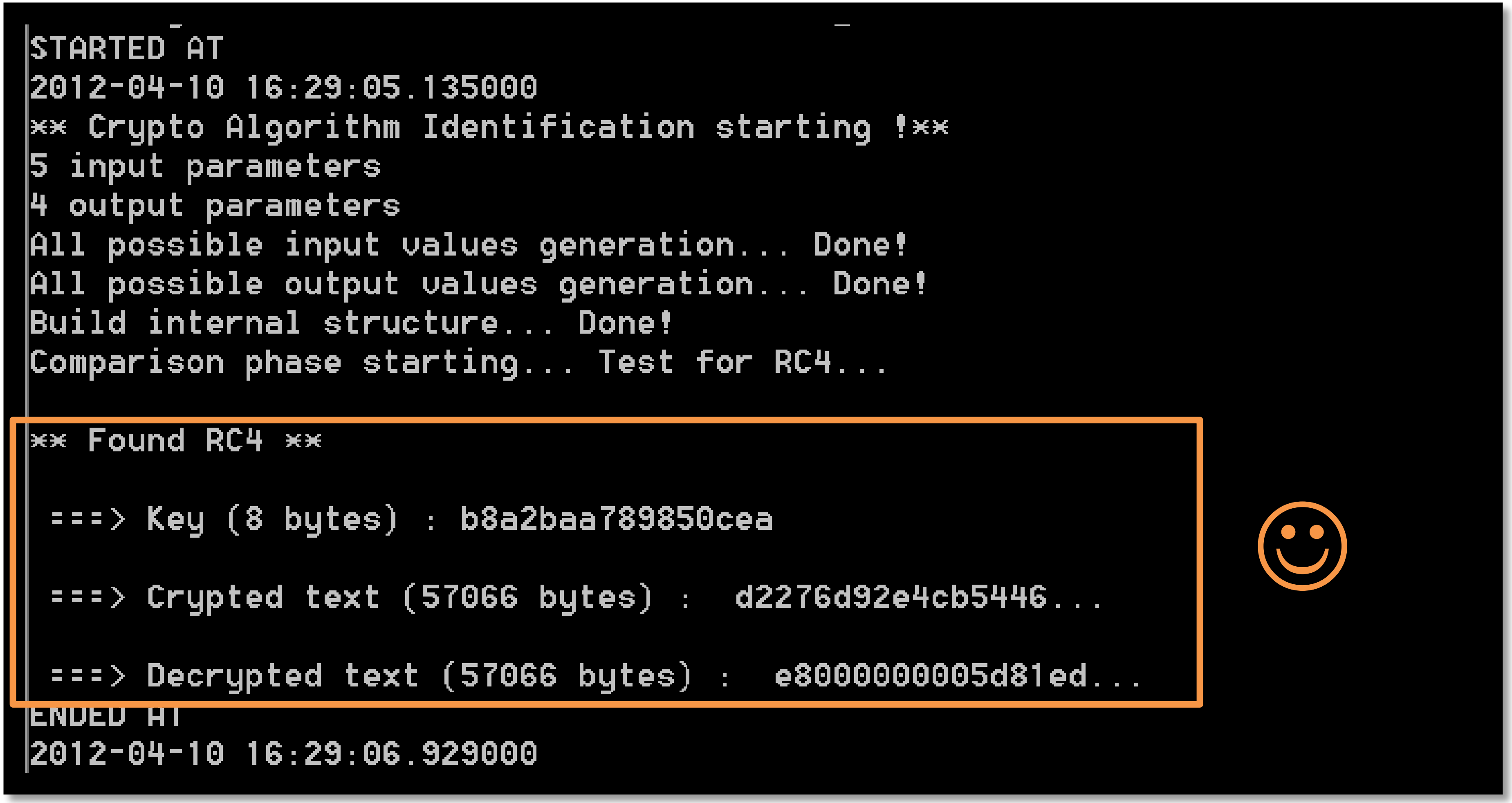
X86 ExecutableCode!



Unknown Algorithm

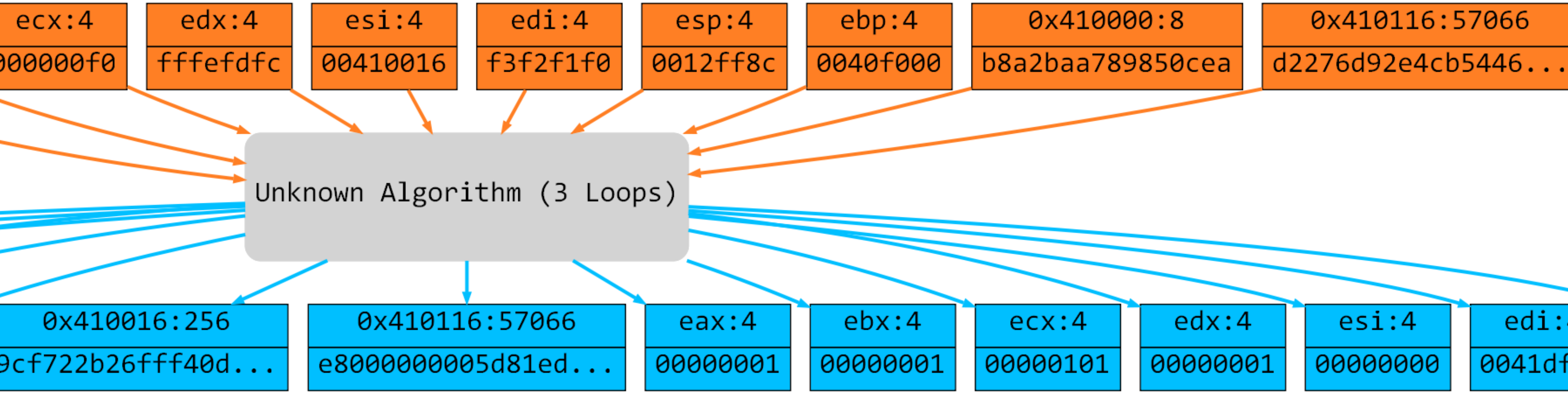
IDENTIFICATION

…



```
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
```

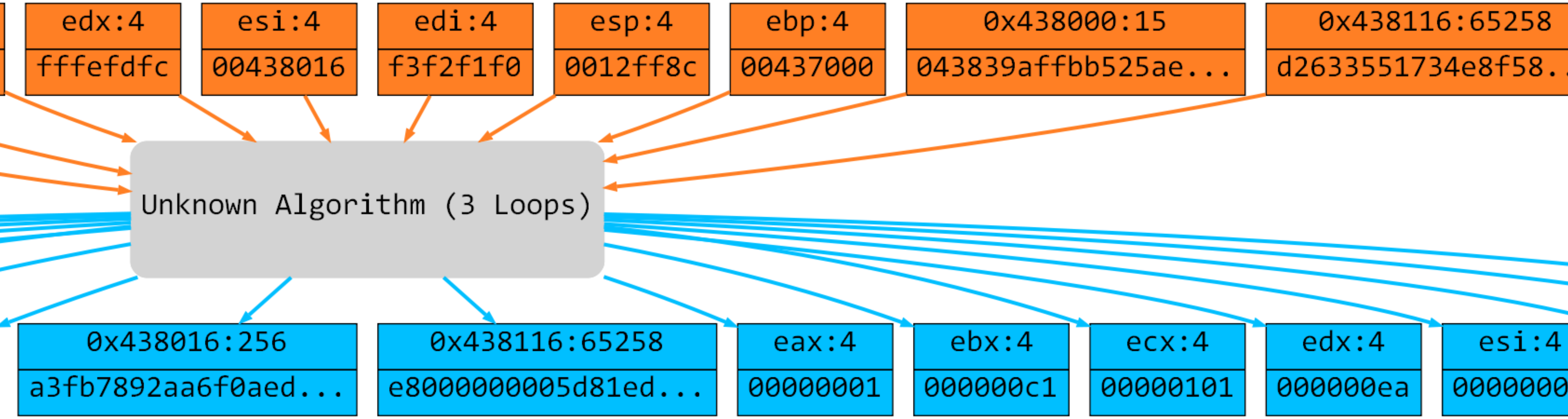
RC4 extracted from two Sality binaries



ecx:4 00000f0
edx:4 fffefdfc
esi:4 00410016
edi:4 f3f2f1f0
esp:4 0012ff8c
ebp:4 0040f000
0x410000:8 b8a2baa789850cea
0x410116:57066 d2276d92e4cb5446...

Unknown Algorithm (3 Loops)

0x410016:256 9cf722b26fff40d...
0x410116:57066 e8000000005d81ed...
eax:4 00000001
ebx:4 00000001
ecx:4 00000101
edx:4 00000001
esi:4 00000000
edi: 0041df

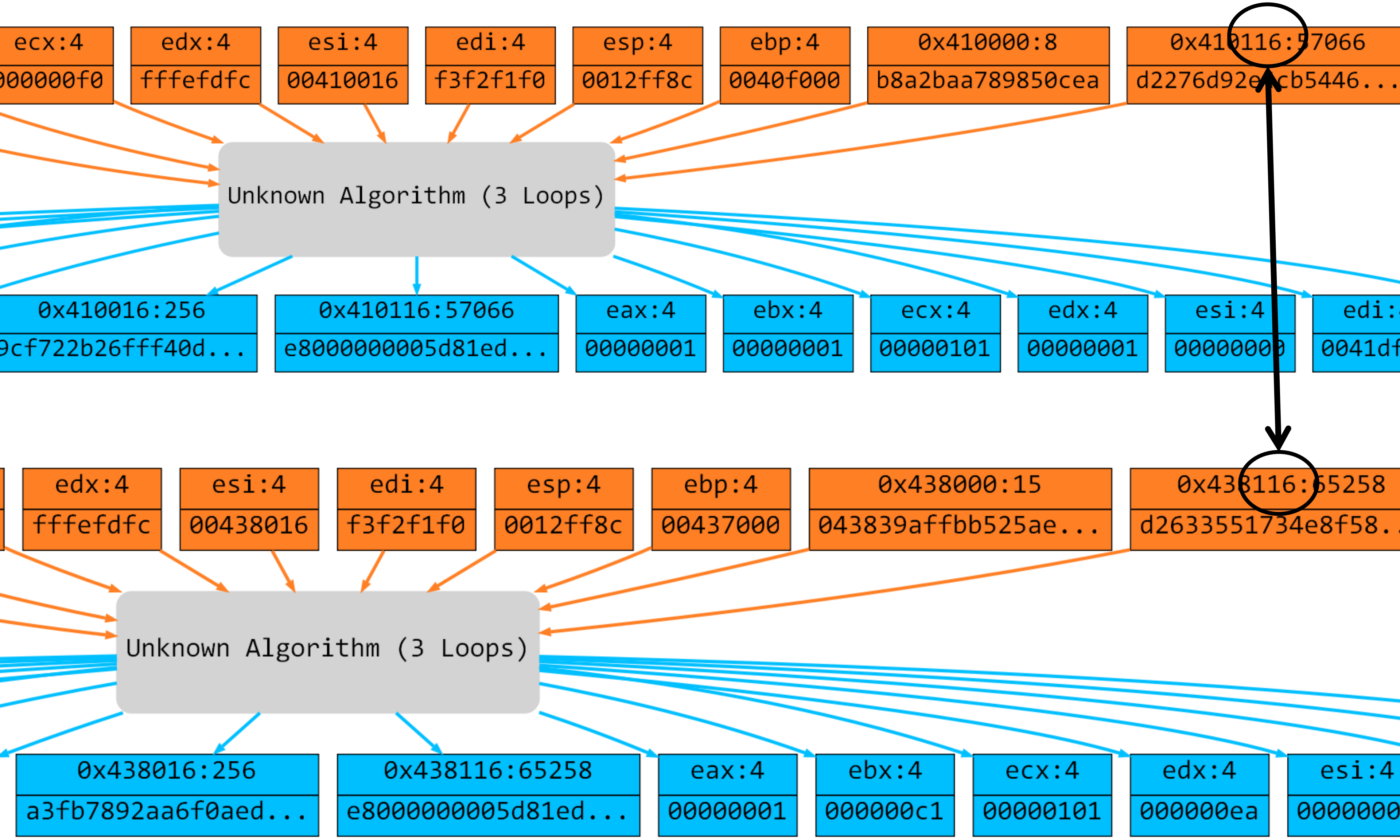


edx:4 fffefdfc
esi:4 00438016
edi:4 f3f2f1f0
esp:4 0012ff8c
ebp:4 00437000
0x438000:15 043839affbb525ae...
0x438116:65258 d263551734e8f58..

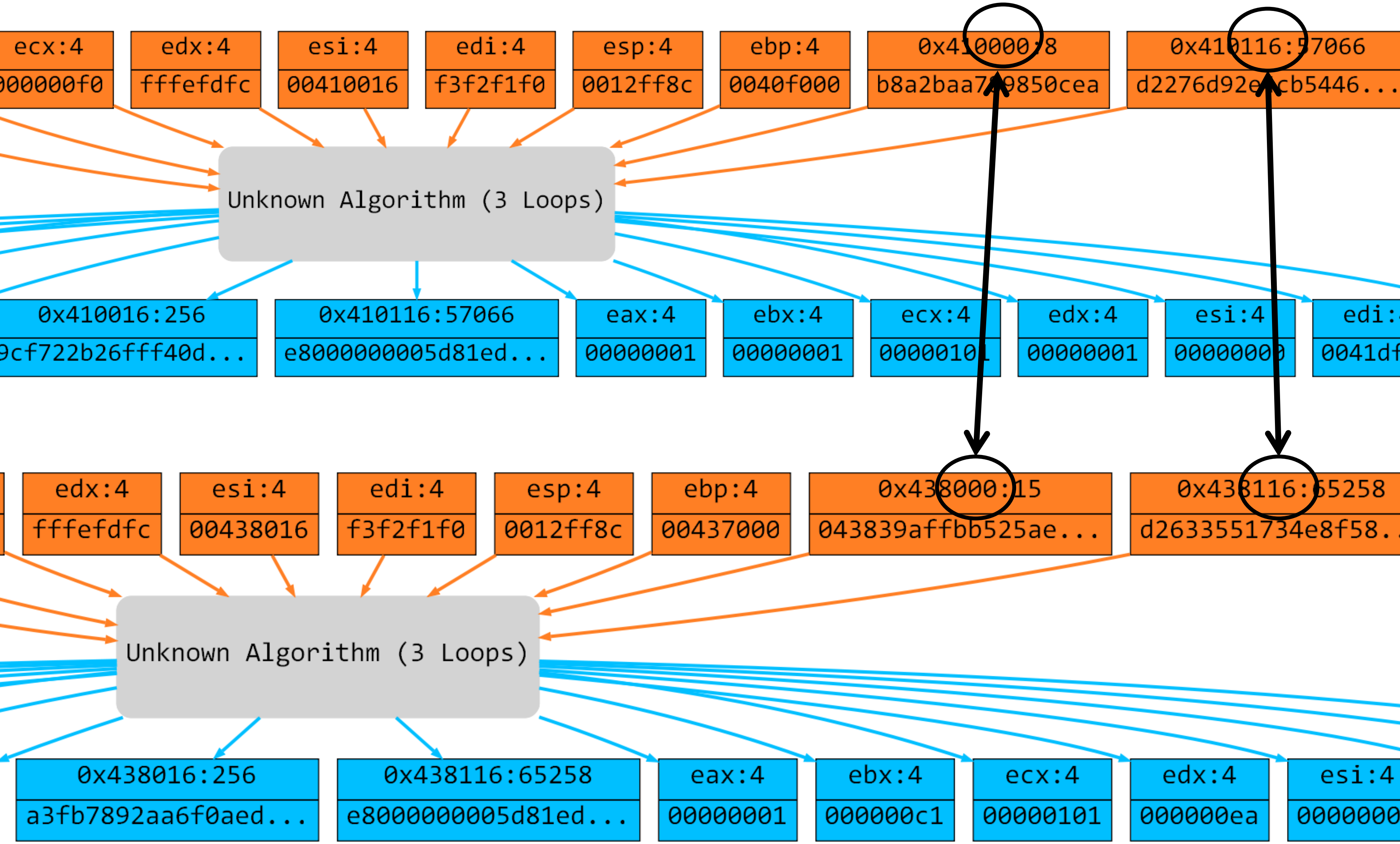
Unknown Algorithm (3 Loops)

0x438016:256 a3fb7892aa6f0aed...
0x438116:65258 e8000000005d81ed...
eax:4 00000001
ebx:4 000000c1
ecx:4 00000101
edx:4 000000ea
esi:4 0000000

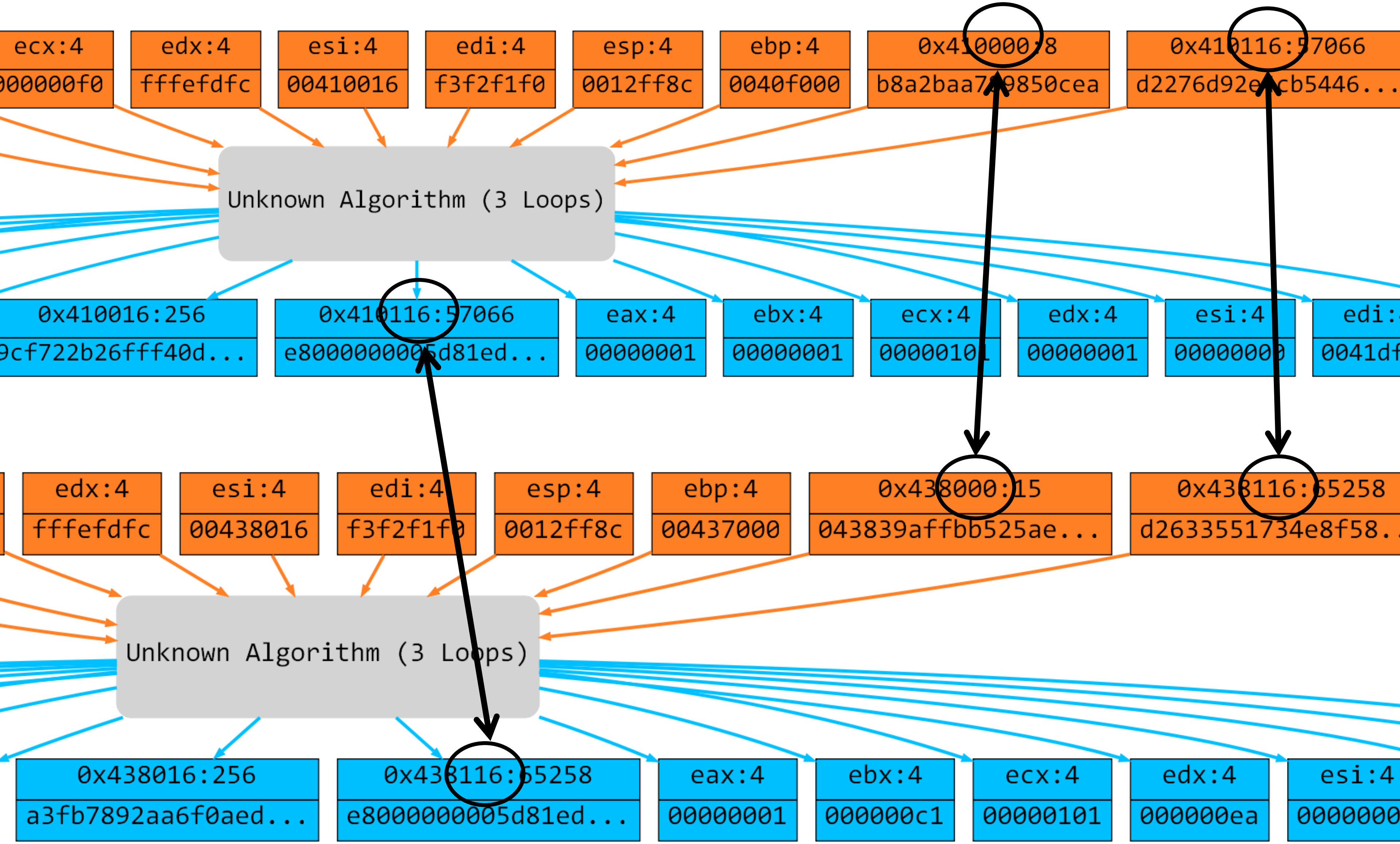
RC4 extracted from two Sality binaries



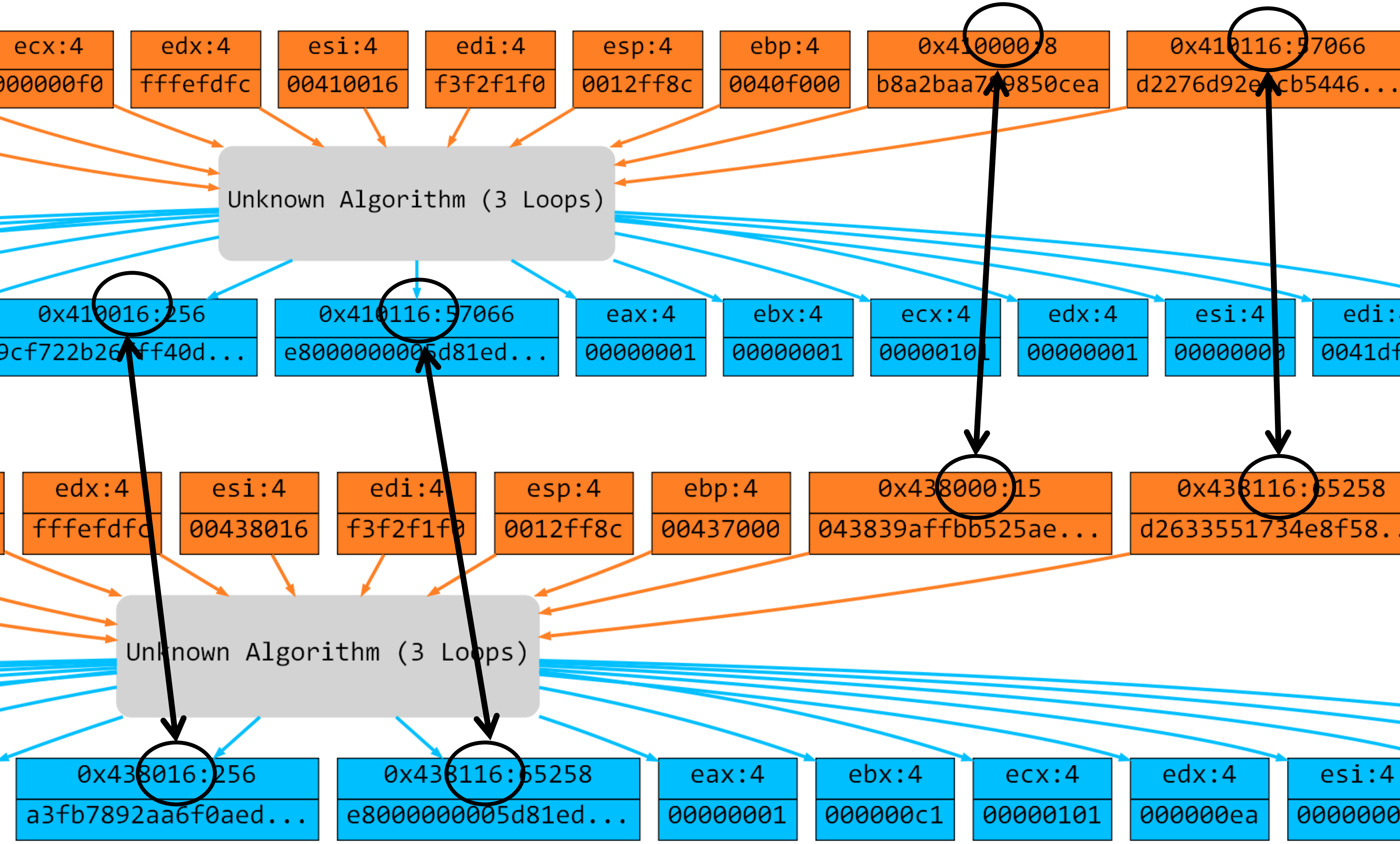
RC4 extracted from two Sality binaries



RC4 extracted from two Sality binaries



RC4 extracted from two Sality binaries



ecx:4
00000f0

edx:4
fffefdfc

esi:4
00410016

edi:4
f3f2f1f0

esp:4
0012ff8c

ebp:4
0040f000

0x410000:8
b8a2baa7[illegible]9850cea

0x410116:57066
d2276d92e[illegible]cb5446...

Unknown Algorithm (3 Loops)

0x410016:256
9cf722b26[illegible]ff40d...

0x410116:57066
e8000000005d81ed...

eax:4
00000001

ebx:4
00000001

ecx:4
00000101

edx:4
00000001

esi:4
00000000

edi:
0041df

edx:4
fffefdfc

esi:4
00438016

edi:4
f3f2f1f0

esp:4
0012ff8c

ebp:4
00437000

0x438000:15
043839affbb525ae...

0x438116:65258
d2633551734e8f58..

Unknown Algorithm (3 Loops)

0x438016:256
a3fb7892aa6f0aed...

0x438116:65258
e8000000005d81ed...

eax:4
00000001

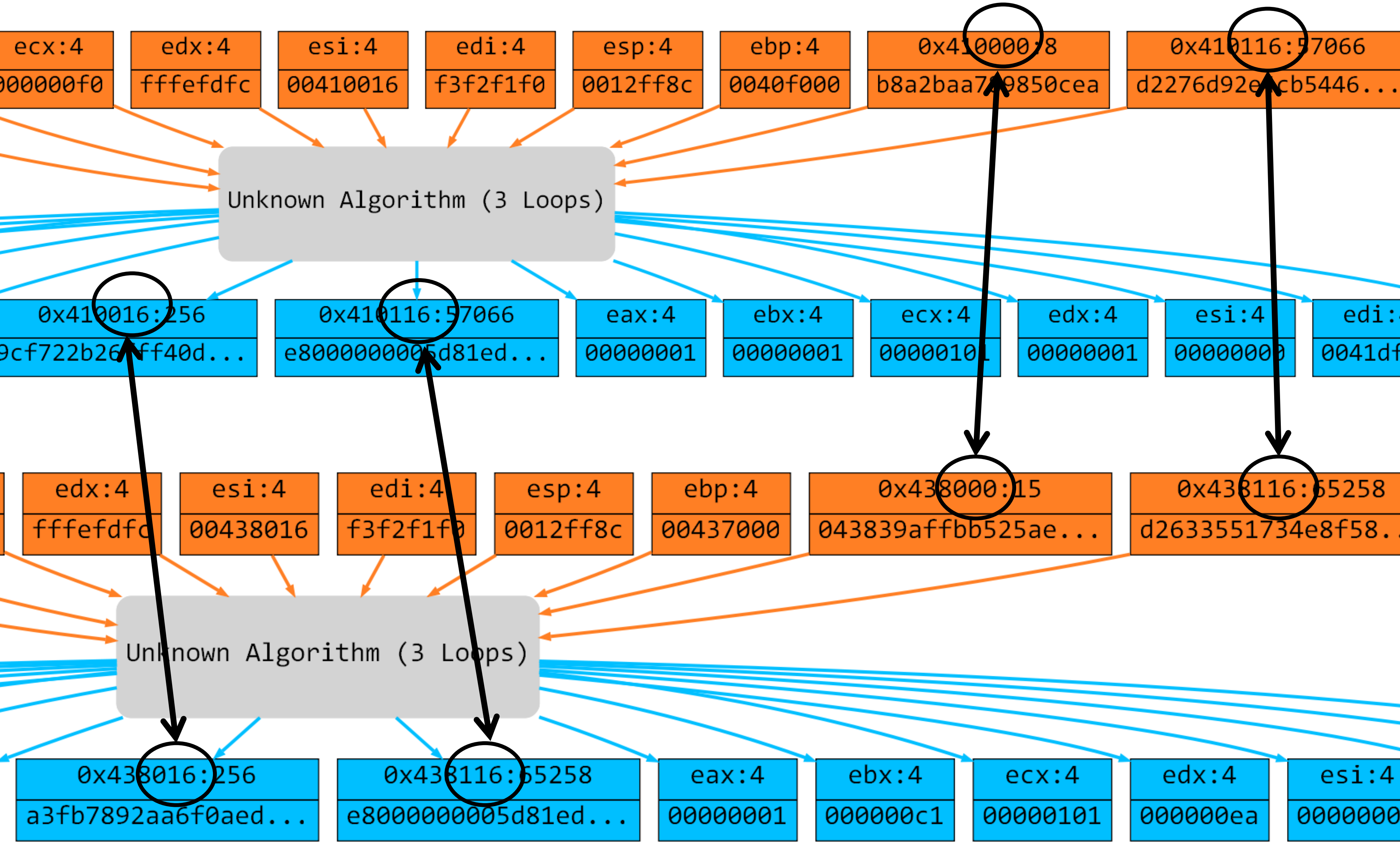
ebx:4
000000c1

ecx:4
00000101

edx:4
000000ea

esi:4
0000000

RC4 extracted from two Sality binaries



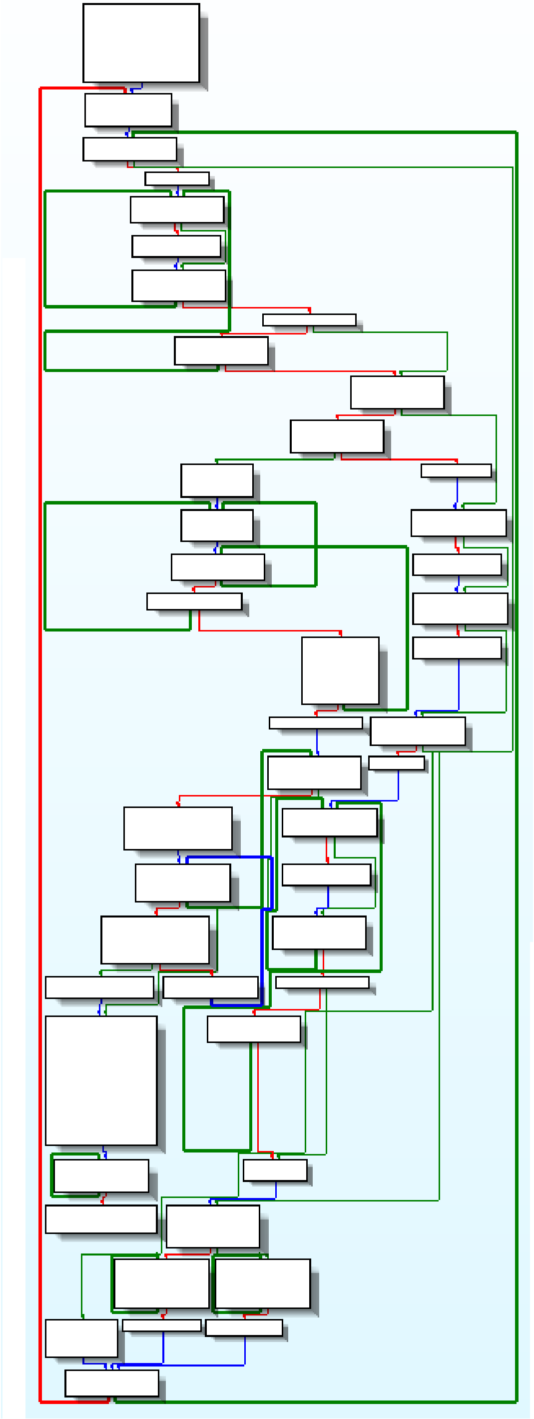
Crypto parameters always at the same offsets!

EXAMPLE WHERE IT ~~DOESN’T~~ *COULD* WORK

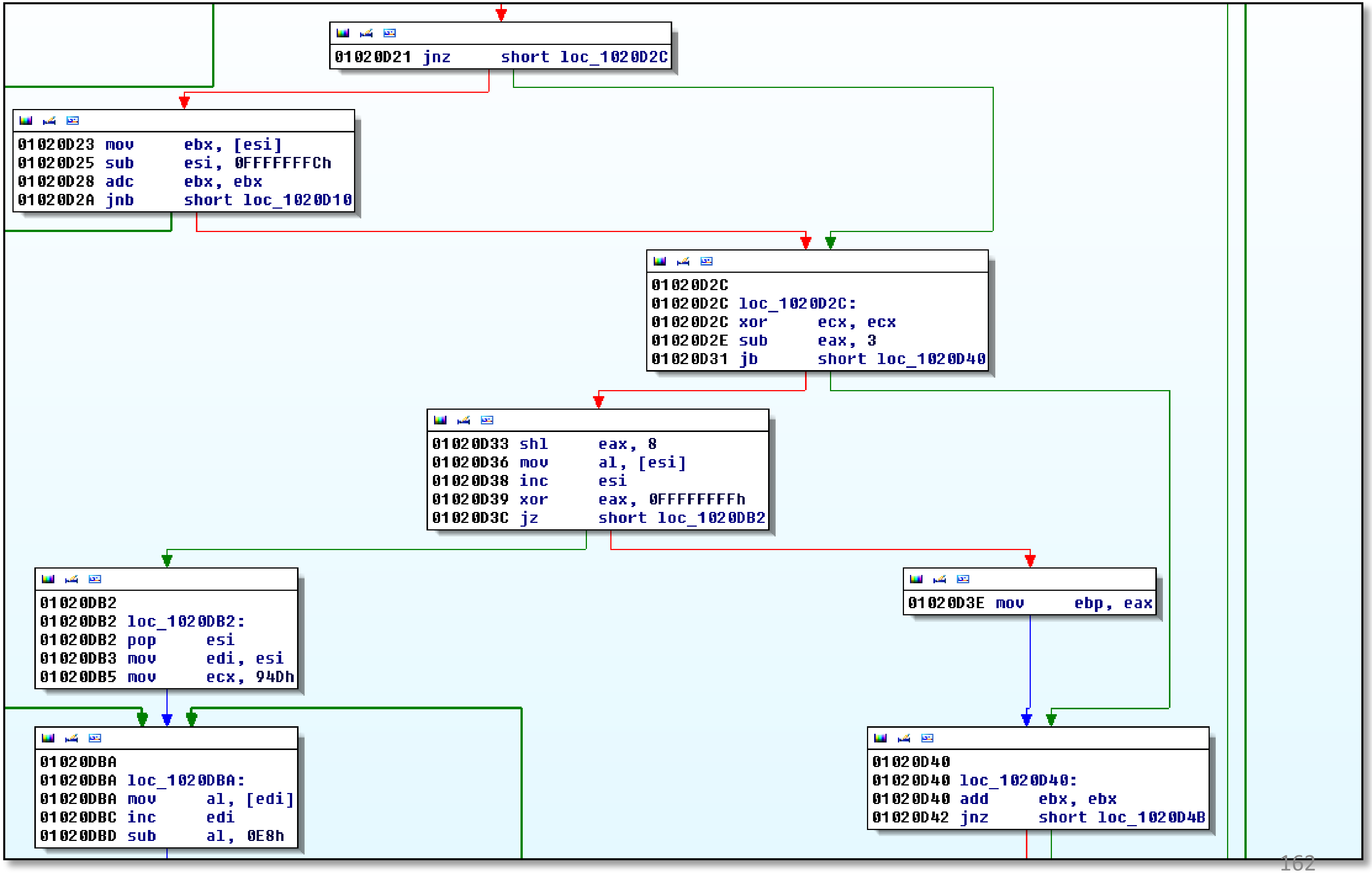
Compression Algorithms

- Compression algorithms have also a very particular input-output relationship.
- That should work too!

UPX LZMA



Fail... Why ?



```
01020D21 jnz     short loc_1020D2C
```

```
01020D23 mov     ebx, [esi]
01020D25 sub     esi, 0FFFFFFFCh
01020D28 adc     ebx, ebx
01020D2A jnb     short loc_1020D10
```

```
01020D2C
01020D2C loc_1020D2C:
01020D2C xor     ecx, ecx
01020D2E sub     eax, 3
01020D31 jb      short loc_1020D40
```

```
01020D33 shl     eax, 8
01020D36 mov     al, [esi]
01020D38 inc     esi
01020D39 xor     eax, 0FFFFFFFFh
01020D3C jz      short loc_1020DB2
```

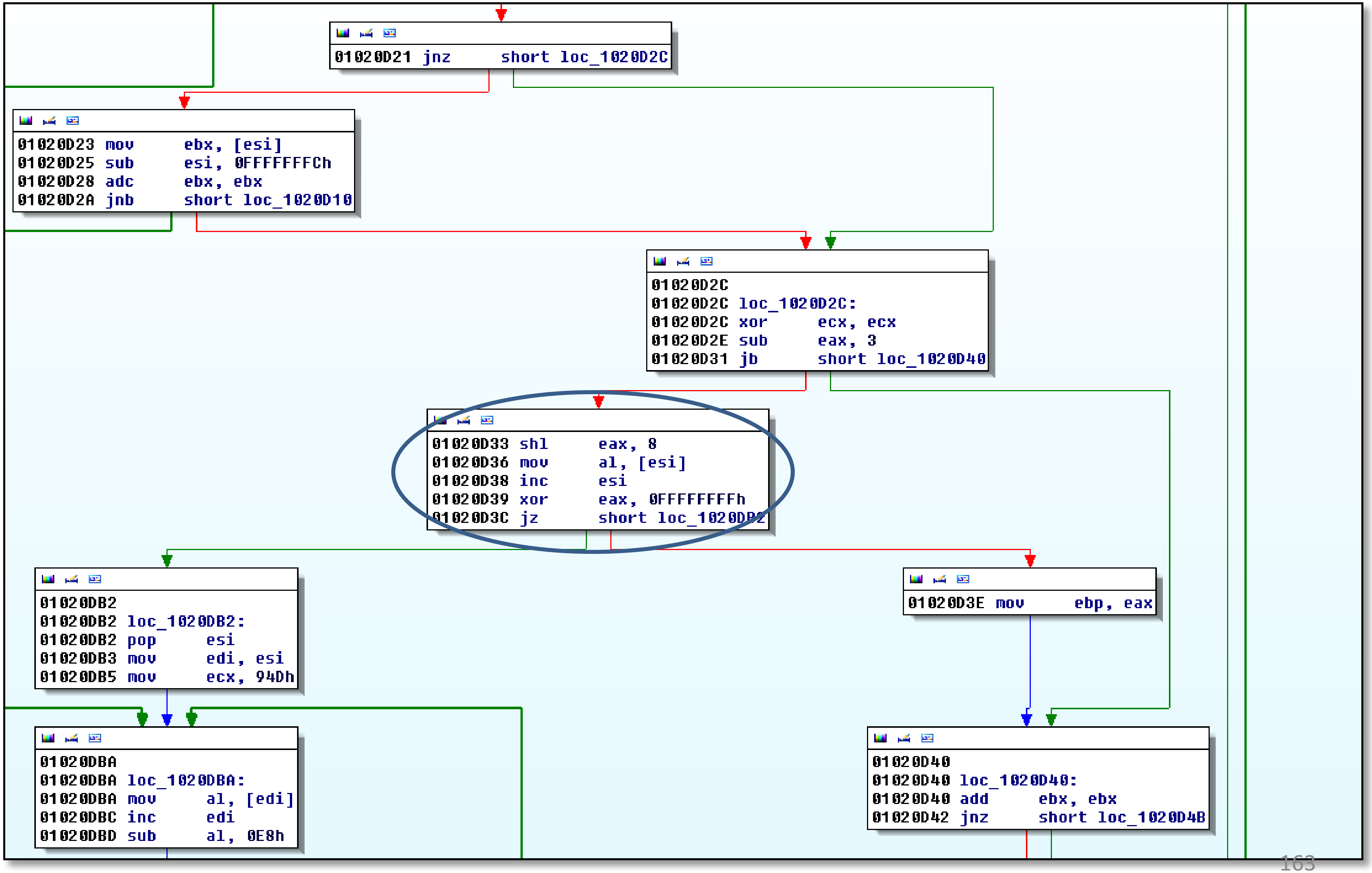
```
01020DB2
01020DB2 loc_1020DB2:
01020DB2 pop     esi
01020DB3 mov     edi, esi
01020DB5 mov     ecx, 94Dh
```

```
01020D3E mov     ebp, eax
```

```
01020DBA
01020DBA loc_1020DBA:
01020DBA mov     al, [edi]
01020DBC inc     edi
01020DBD sub     al, 0E8h
```

```
01020D40
01020D40 loc_1020D40:
01020D40 add     ebx, ebx
01020D42 jnz     short loc_1020D4B
```

Fail... Why ?



```
01020D21 jnz     short loc_1020D2C
```

```
01020D23 mov     ebx, [esi]
01020D25 sub     esi, 0FFFFFFFCh
01020D28 adc     ebx, ebx
01020D2A jnb     short loc_1020D10
```

```
01020D2C
01020D2C loc_1020D2C:
01020D2C xor     ecx, ecx
01020D2E sub     eax, 3
01020D31 jb      short loc_1020D40
```

```
01020D33 shl     eax, 8
01020D36 mov     al, [esi]
01020D38 inc     esi
01020D39 xor     eax, 0FFFFFFFFh
01020D3C jz      short loc_1020DB2
```

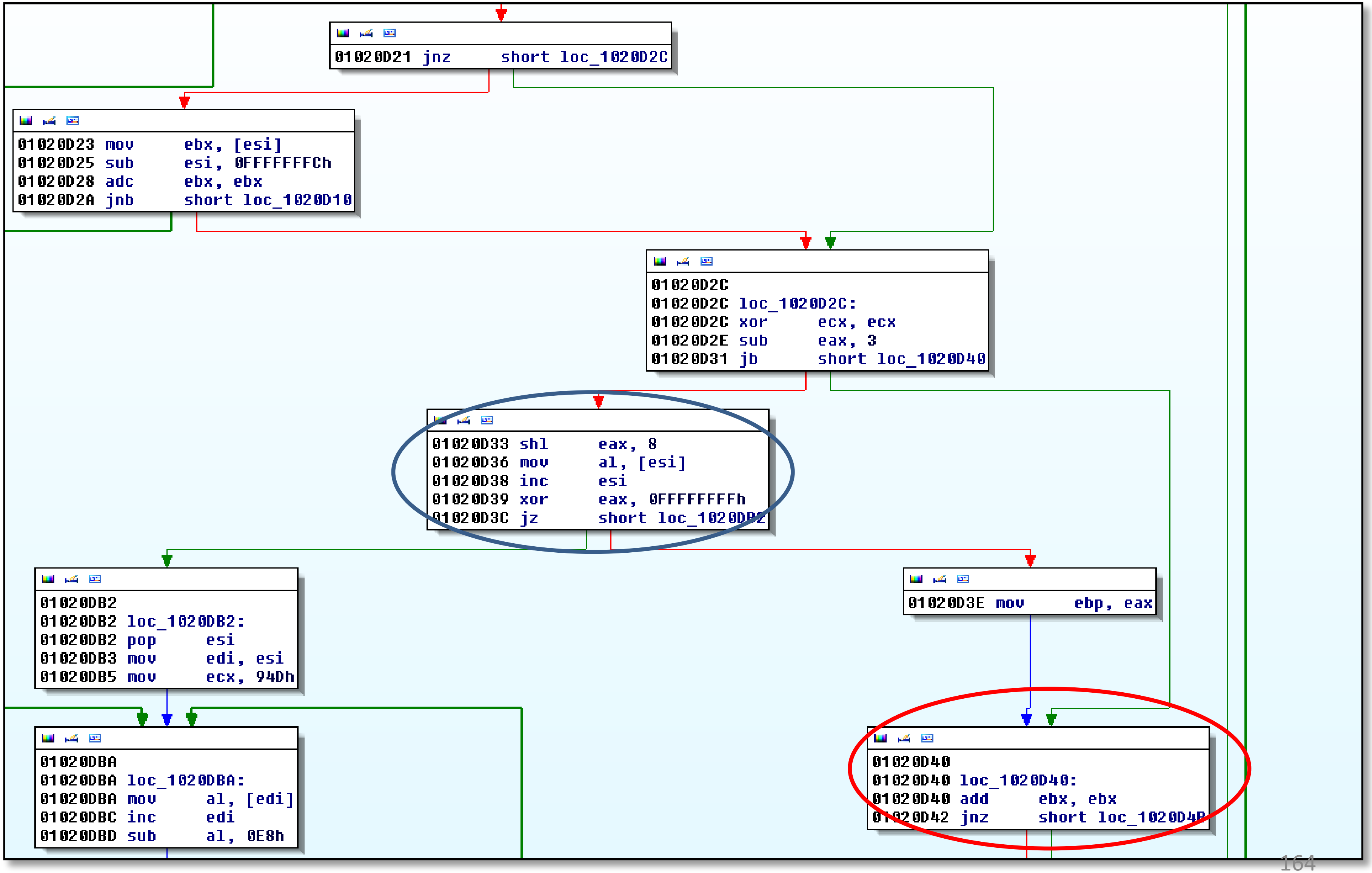
```
01020DB2
01020DB2 loc_1020DB2:
01020DB2 pop     esi
01020DB3 mov     edi, esi
01020DB5 mov     ecx, 94Dh
```

```
01020D3E mov     ebp, eax
```

```
01020DBA
01020DBA loc_1020DBA:
01020DBA mov     al, [edi]
01020DBC inc     edi
01020DBD sub     al, 0E8h
```

```
01020D40
01020D40 loc_1020D40:
01020D40 add     ebx, ebx
01020D42 jnz     short loc_1020D4B
```

Fail... Why ?



```
01020D21 jnz     short loc_1020D2C
```

```
01020D23 mov     ebx, [esi]
01020D25 sub     esi, 0FFFFFFFCh
01020D28 adc     ebx, ebx
01020D2A jnb     short loc_1020D10
```

```
01020D2C
01020D2C loc_1020D2C:
01020D2C xor     ecx, ecx
01020D2E sub     eax, 3
01020D31 jb      short loc_1020D40
```

```
01020D33 shl     eax, 8
01020D36 mov     al, [esi]
01020D38 inc     esi
01020D39 xor     eax, 0FFFFFFFFh
01020D3C jz      short loc_1020DB2
```

```
01020DB2
01020DB2 loc_1020DB2:
01020DB2 pop     esi
01020DB3 mov     edi, esi
01020DB5 mov     ecx, 94Dh
```

```
01020D3E mov     ebp, eax
```

```
01020DBA
01020DBA loc_1020DBA:
01020DBA mov     al, [edi]
01020DBC inc     edi
01020DBD sub     al, 0E8h
```

```
01020D40
01020D40 loc_1020D40:
01020D40 add     ebx, ebx
01020D42 jnz     short loc_1020D4B
```

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 the program behaves like a known crypto algorithm during one particular execution path.

Conclusion (1)

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

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

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

What’s Next ?

- That’s only the beginning! Just wanted to show that it is feasible and useful.
- Extension to other crypto algorithms.
- Compression algorithms ? What should be the correct criteria to extract compression code ?

  (my 2 cts: natural loops, that is back-edges on CFG)
- If we use several criteria to extract interesting code, how to combine them ?
- How to use the analyst knowledge ?
- Make a real tool. This one is just a PoC.

Thank you for your attention ;-)

Performances

	Sality 1	Sality 2
Trace Size (instructions)	~1M	~4M
Time To Trace	5mn	10mn
Time To Extract Crypto Algoritm	4h	15h
Time To Identify	3mn	4mn

- The tool is just a PoC, no optimization **at all**.

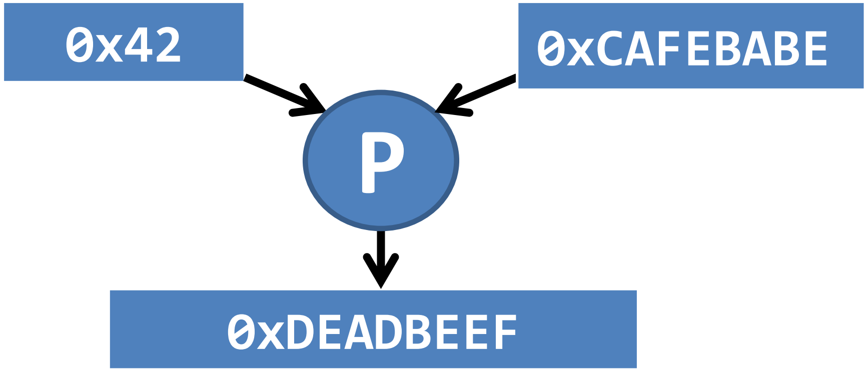
- When the analysts knows where the algorithm is, it will reduce the trace size.

Answer: Brute-Force

- We generate any possible input values by appending together **A** inputs, it gives a set **IN**.
- Same thing for output values, in the set **OUT**.

- <u>For each reference implementation **F**:</u>
  - We select the **IN** values that can fit into **F** parameters.
  - We execute **F** on each possible input value combination.
  - If **F** output is something in **OUT**, it is a success.

For example:



0x42

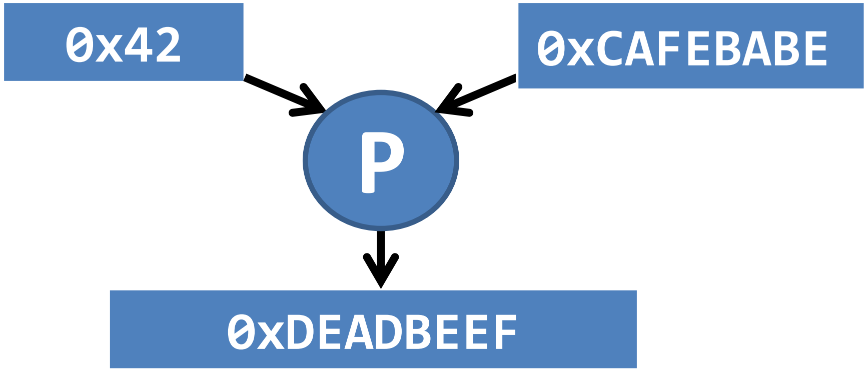
0xCAFEBABE

P

0xDEADBEEF

- For each known cryptographic algorithm **A**:

For example:



0x42

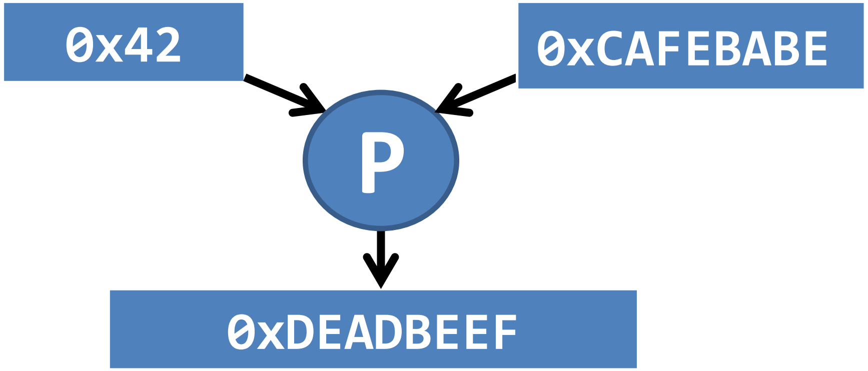
0xCAFEBABE

P

0xDEADBEEF

- For each known cryptographic algorithm **A**:
    - We execute a reference implementation of **A** on 0x42 and 0xCAFEBABE.

For example:


0x42

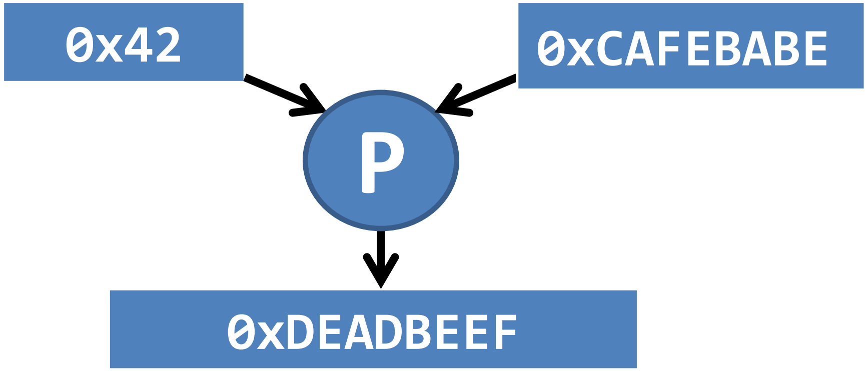
0xCAFEBABE

P

0xDEADBEEF

- For each known cryptographic algorithm **A**:
  - We execute a reference implementation of **A** on 0x42 and 0xCAFEBABE.
  - We check if the output is 0xDEADBEEF.

For example:



- For each known cryptographic algorithm **A**:
  - We execute a reference implementation of **A** on 0x42 and 0xCAFEBABE.
  - We check if the output is 0xDEADBEEF.
  - If so, **we have proved that P implements A <u>on these particular input values.</u>**