

NOSUCHCON 2013 – WRITE UP

31 MAY 2013

TWITTER: @KUTIOO


KUTIOO@GMAIL.COM

Table of contents

Nosuchcon 2013 – Write up	1
Introduction	2
Bad ideas and Pin tracing.....	3
Step by step	3
Hardware breakpoints on the serial and on the output.....	3
Generic unobfuscators	3
Reverse by hand	4
Pin tracing.....	4
Automated deobfuscation.....	6
Simple table (TS)	6
Handlers table (THS)	8
16-bit table (T16S)	9
Full state machine.....	11
Extract tables	12
Cryptanalysis on the AES Whitebox.....	13
Static Single Assignment form	13
Graph	13
Rounds.....	15
Cryptanalysis.....	15
Automated attack.....	16
Conclusion	19

INTRODUCTION

This year, The NoSuchCon2013 challenge was created by **Eloi Vanderbeken** from **Oppida**. The challenge is a simple **keygen-me**:



NSC #1

NSC #1

kutio	
F342A4F91D49EFB9C39AED1AE2974626	
Check	
About	Exit

The goal of the challenge is to find a pair of a (nickname, serial) that follows this equation:

$$\text{MD5}(\text{nickname}) == \text{SHUFFLE}(\text{serial})$$

SHUFFLE is the algorithm that we have to reverse to break the challenge. To do that we can follow two approaches:

- ✓ inverse the **SHUFFLE** algorithm: $\text{SHUFFLE}^{-1}(\text{MD5}(\text{nickname})) == \text{serial}$
- ✗ find a MD5 pre-image: $\text{nickname} == \text{MD5}^{-1}(\text{SHUFFLE}(\text{serial}))$

Of course it's easier to inverse the algorithm compared to find a pre-image for **MD5**. In the first section of this paper, we will see the methodology to break the challenge, even the bad ideas. Then we will see how to automate the deobfuscation. In the third section, we will see how to attack the algorithm and we will end up with a small conclusion. The aim of this write-up is more focus on the methodology than the result, that's why I'm really interested to have feedback, comments and ideas about the process.

BAD IDEAS AND PIN TRACING

The difficulty with this challenge is to isolate the useful code and try to understand the logic. The code is highly obfuscated and we don't really know if some parts of the code are junk or necessary. In this section, I will present different technics that I tried to understand the code and to bypass the obfuscation.

STEP BY STEP

When you analyze the **SHUFFLE** function with IDA, most of the code is defined as **data** and with this amount of code, it's not possible to define all the code by hand. So the first thing I try when I have to reverse a function is to browse the code **step-by-step**. The analysis is dynamic and the goal is to identify the useful code from the obfuscated code. With this approach, we can try to find the logic of the code to speed up the reverse of the function.

Unfortunately, the amount of code is really huge and it's practically unfeasible to browse all the code of the **SHUFFLE** function. Moreover, with this technic we can see that all the code is organized in the same way and we can identify repetitive snippet of code.

HARDWARE BREAKPOINTS ON THE SERIAL AND ON THE OUTPUT

The second approach is to put hardware breakpoints on the serial and on the output, and try to understand where they are used. The goal is to identify the junk code from the useful code and try to understand how the serial is used in the **SHUFFLE** function. This technic was not really suitable because the amount of code to handle the serial is also huge, but we can identify that the code is intrinsically linked and dependent. And in fact there is no junk code.

GENERIC UNOBFUSCATORS

Use a generic unobfuscator seems to be a good idea. We can for example use **coreopt** or **optimice**. There is a protection in the obfuscated code to break the graph flow. Some executed code depending on the serial:

```

0x005025C0 mov esi, 0x61700d
# 005025C5: R 00617000, size = 1, value = 0xaa
0x005025C5 mov al, byte ptr [esi]
0x005025C7 lea ebx, ptr [eax*4+0x43e740]
# 005025CE: R 0043E9E8, size = 4, value = 0x491cda
0x005025CE push dword ptr [ebx]
# 005025CE: W 0018FA20, size = 4, value = 0x491cda
# 005025D0: R 0018FA20, size = 4, value = 0x491cda
0x005025D0 ret
0x00491CDA rol dh, 0x67
0x00491CDD mov eax, 0x43ace2
0x00491CE2 mov esi, eax
0x00491CE4 lea ebp, ptr [esi]
0x00491CE6 mov byte ptr [ebp], 0x5e
# 00491CE6: W 0043ACE2, size = 1, value = 0x5e

```

This code is explained on the **Automated deobfuscation** section, but we can see that the code executed at **0x00491CDA** depends on **serial[0xD] (0x61700D)**. Most of the generic unobfuscators are not really convenient against this kind of obfuscation, and it's difficult to handle this generically.

REVERSE BY HAND

As the title suggests, the approach is to reverse the code by hand. It can be useful to identify the logic, repetitive snippets, and interdependent code. But due to the amount of code, this technic should be paired with another more effective approach. It's quite stupid and infeasible to try to reverse all the code by hand.

PIN TRACING

The most effective technic to understand this challenge is to trace the **SHUFFLE** function. Furthermore, we will see on the next section that the trace would be useful to do some automated operations.

To trace the function we can use **Pin** (from Intel). The idea is simple, we just need to record all instructions executed, all read operations, and all write operations. We can based our tools on the **pinatrace.cpp** sample of **Pin** and write 3 tracers:

- **Tracer.cpp**, records:
 - executed instructions
 - read operations (address + value)
 - write operations (address + value)
- **TracerReadKey.cpp**, records:
 - read operations on the serial (address + value)
- **TracerWriteOutput.cpp**, records:
 - write operations on the output (address + value)

Tracer.cpp allows to generate the trace **trace-full.asm**:

```
# 0059D170: R    00617016, size = 1, value =      0x14
0x0059D170 mov ah, byte ptr [0x617016]
0x0059D176 mov cl, ah
0x0059D178 shl ecx, 0x8
0x0059D17B mov esi, 0x5F7f82
# 0059D180: R    005F7F82, size = 1, value =      0xa7
0x0059D180 mov dh, byte ptr [esi]
0x0059D182 jmp 0x59d14e
0x0059D14E mov cl, dh
# 0059D150: R    004F06E5, size = 4, value = 0xac2e7051
0x0059D150 mov edi, dword ptr [ecx+0x4ef23e]
0x0059D156 jmp 0x59d15b
0x0059D15B mov ecx, edi
0x0059D15D jmp 0x59d187
0x0059D187 mov esi, 0x617016
0x0059D18C mov byte ptr [esi], cl
# 0059D18C: W    00617016, size = 1, value =      0x51
```

This trace is really huge, and now we understand why it's impossible to reverse the **SHUFFLE** function by hand. With this trace it's quite easy to identify the repetitive snippets of code.

TracerReadKey.cpp allows to generate the trace **trace-read.asm**:

```
55 00463FBA: R    0061700E 1      aa
56 005A71CD: R    0061700D 1      aa
57 0042F4AA: R    00617006 1      aa
58 0053214D: R    00617005 1      aa
59 00590363: R    00617005 1      aa
60 004CBB57: R    00617005 1      aa
61 0045D390: R    00617005 1      aa
62 004D6CF8: R    0061700B 1      aa
63 00472EA0: R    0061700A 1      aa
```

This trace shows that the serial is mainly used at the beginning of the **SHUFFLE** function (3000 first lines). We can also see that the read operations on the serial's bytes are not distributed uniformly, it's quite heterogeneous.

TracerWriteOutput.cpp allows to generate the trace **trace-write.asm**:

```
134 00446215: W    0061701F 1      c7
135 0057C3D1: W    0061701C 1      e9
136 005FB5E2: W    0061701C 1      cf
137 00564DF0: W    0061701A 1      49
138 0040383B: W    0061701E 1      ef
```

This trace shows that the output is used on the whole **SHUFFLE** function. We can distinguish 148 write operations on the output. It's quite weird because the output table is just 16 bytes long. It means that the output is used to store intermediate values during the processing of the algorithm and not only the output of the **SHUFFLE** function.

To resume, these traces put in conspicuous position that the algorithm consists essentially in 17000 lines of repetitive snippets of code. We can also see that the code is intrinsically linked and inter-dependent. It seems that there is no junk code and all code is necessary to compute the output. We can distinguish 592 different tables of 256 bytes and 3 tables of 65536 bytes. It's really unsettling to see that read and write operations are totally disordered and we can't find out a logic.

AUTOMATED DEOBFUSCATION

According to the first section, we saw that the **SHUFFLE** function consists mainly in repetitive snippets of code. The obfuscated code seems to be generated automatically, so it should be possible to unobfuscate in an automated way. In this section we will see how to abstract each snippet of code. The goal is to create a parser that implements a state machine to unobfuscate automatically the trace previously generated (**trace-full.asm**).

SIMPLE TABLE (TS)

A typical snippet of code that corresponding to what we call a **Simple Table (TS)** is the following:

```
0x004068ED mov edx, 0x5b46f9
# 004068F2: R 005B46F9, size = 1, value = 0xd4
0x004068F2 mov al, byte ptr [edx]
0x004068F4 mov esi, eax
0x004068F6 add esi, 0x5afc32
# 004068FC: R 005AFD06, size = 4, value = 0x49dad91c
0x004068FC mov ebx, dword ptr [esi]
0x004068FE lea esi, ptr [0x4386bf]
0x00406904 mov byte ptr [esi], bl
# 00406904: W 004386BF, size = 1, value = 0x1c
```

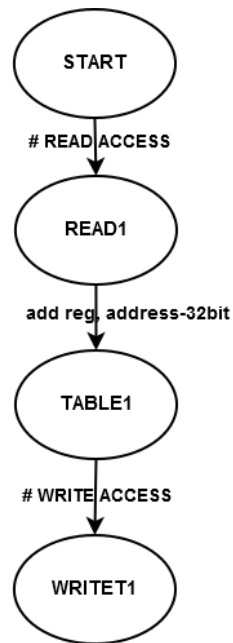
This code can be translated on this pseudo-code:

```
var_0x004386BF = TS_0x005AFC32[var_0x005B46F9] # // TS
```

It's easy to abstract this model and it can be modeled in this way:

- **first step:** READ operation
- **second step:** add register with a 32 bit address
- **last step:** WRITE operation

If the parser encounters these steps in this order, it's a **TS** snippet of code. Because we want to implement a state machine, we can model this with this state machine:



There also exists a second case for a **Simple Table (TS)**, a typical sample of code is this one:

```

0x0043DF98 mov edi, 0x617013
0x0043DF9D jmp 0x43dfa3
# 0043DFA3: R    00617013, size = 1, value = 0x12
0x0043DFA3 mov dl, byte ptr [edi]
# 0043DFA5: R    0042FFCD, size = 4, value = 0xe94cb531
0x0043DFA5 mov edi, dword ptr [edx+0x42ffbb]
0x0043DFAB mov eax, edi
0x0043DFAD mov edi, 0x617017
0x0043DFB2 mov esi, edi
0x0043DFB4 mov byte ptr [esi], al
# 0043DFB4: W    00617017, size = 1, value = 0x31
  
```

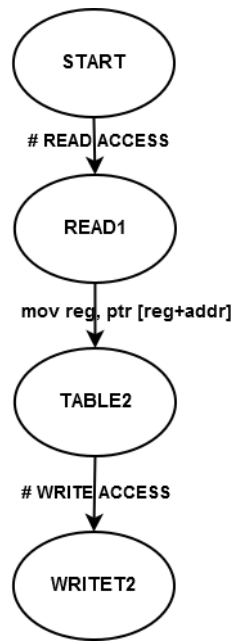
This second case can be translated on this pseudo-code:

```
output[0x7] = TS_0x0042FFBB[output[0x3]] # // TS
```

We can abstract this case in this way:

- **first step:** READ operation
- **second step:** mov register, dword ptr [register + 32-bit address]
- **last step:** WRITE operation

The state machine corresponding to this case:



HANDLERS TABLE (THS)

A typical snippet of code that corresponding to what we call a **Handlers table (THS)** is the following:

```

0x005025C0 mov esi, 0x61700d
# 005025C5: R 00617000, size = 1, value = 0xaa
0x005025C5 mov al, byte ptr [esi]
0x005025C7 lea ebx, ptr [eax*4+0x43e740]
# 005025CE: R 0043E9E8, size = 4, value = 0x491cda
0x005025CE push dword ptr [ebx]
# 005025CE: W 0018FA20, size = 4, value = 0x491cda
# 005025D0: R 0018FA20, size = 4, value = 0x491cda
0x005025D0 ret
0x00491CDA rol dh, 0x67
0x00491CDD mov eax, 0x43ace2
0x00491CE2 mov esi, eax
0x00491CE4 lea ebp, ptr [esi]
0x00491CE6 mov byte ptr [ebp], 0x5e
# 00491CE6: W 0043ACE2, size = 1, value = 0x5e
  
```

This snippet of code is really interesting because instead of containing 256 values, the table contains 256 code pointers (handlers). The idea is in fact pretty simple, instead of:

```
var_0x0043ACE2 = Table[serial[0xD]]
```

We have:

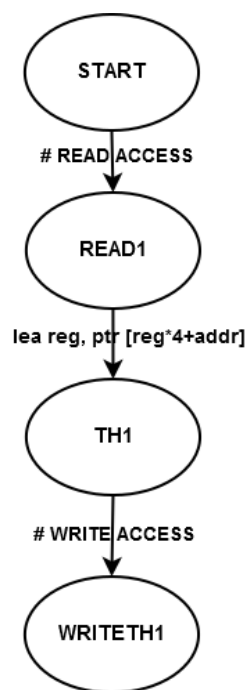
```
var_0x0043ACE2 = Table[serial[0xD]].handler.val
```


For each entry in the table, we have a specific handler that writes a unique value in a specific destination (in our example **var_0x0043ACE2**). This model is in fact a table of values and the goal is to break the graph's flow. So the pseudo-code corresponding to this case:

```
var_0x0043ACE2 = THS_0043E740[serial[0xD]] # // THS
```

We can abstract this case in this way:

- **first step:** READ operation
- **second step:** lea register, ptr [register*4 + 32-bit address]
- **last step:** WRITE operation



An important rule of our parser is that all the write and read operations on the stack (for example: **0x0018FA20**) are not taken into account and are considered as junk.

16-BIT TABLE (T16S)

A typical snippet of code that corresponding to what we call a **16-bit table (T16S)** is the following:

```

0x00450001 mov ebp, 0x58d156
# 00450006: R 0058D156, size = 1, value = 0x4d
0x00450006 mov bl, byte ptr [ebp]
0x00450009 mov cl, 0xf1
0x0045000B sub cl, 0xe9
0x0045000E jmp 0x4ffe4
0x0044FFE4 shl ebx, cl
# 0044FFE6: R 0043ACE2, size = 1, value = 0xf2
0x0044FFE6 mov bl, byte ptr [0x43ace2]
# 0044FFEC: R 005115AA, size = 4, value = 0x8454aa98
0x0044FFEC mov ecx, dword ptr [ebx+0x50c7b8]
0x0044FFF2 mov byte ptr [0x61701a], cl
# 0044FFF2: W 0061701A, size = 1, value = 0x98

```

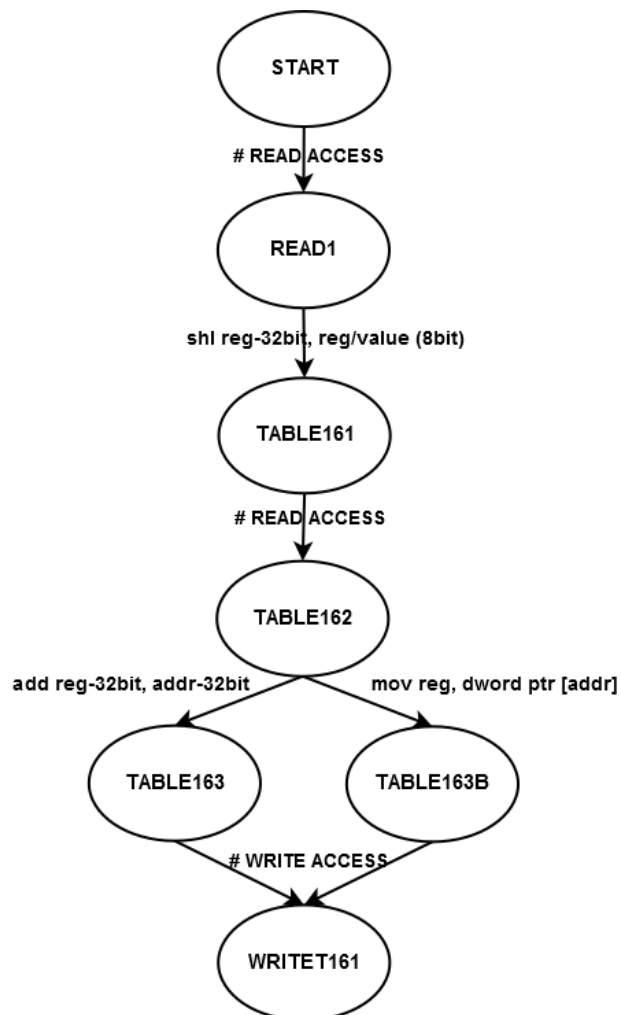
This code can be translated on this pseudo-code:

```

output[0xA] = T16S_0x0050C7B8[({var_0x0058D156 << 8}
+ var_0x0043ACE2) & 0xFFFF] # // T16S

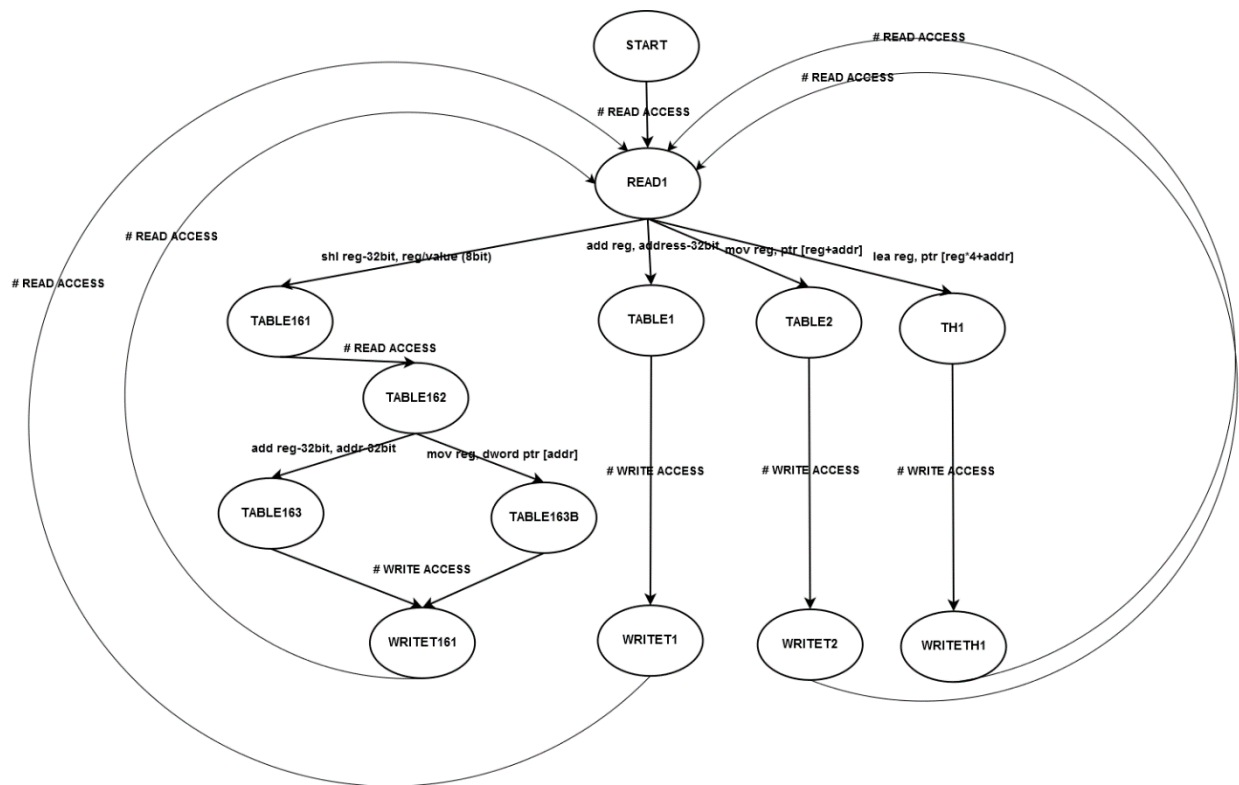
```

This one is more complicated:



FULL STATE MACHINE

The final state machine:



Now we have all the elements to implement our parser (**OppidaParser.cpp**) and to unobfuscate automatically the algorithm. It's convenient to parse directly the trace (**trace-full.asm**) because we can remove the obfuscation easily.

Example:

```
0x005C8178 mov dh, 0x9f
0x005C817A xor dh, 0x9b
0x005C817D mov byte ptr [ebx], dh
# 005C817D: W 005F7F82, size = 1, value = 0x4
```

Thanks to the memory trace, we know that the written value by the handler is **0x4**.

Our parser is able to convert the 17000 lines of obfuscated code to 1024 lines of pseudo-code (**algo.py**) in few milliseconds. Here is the output of our parser:

```

var_0x00591280 = THS_0054F184[serial[0xE]] # // THS
var_0x0049D7F9 = THS_00605BC8[serial[0xC]] # // THS
var_0x004E47E2 = TS_0x00430909[serial[0x0]] # // TS
var_0x00590A90 = TS_0x00550AA4[serial[0xE]] # // TS
var_0x0044AE8B = THS_0055ACDF[serial[0x3]] # // THS
var_0x0043ACE2 = THS_0043E740[serial[0xD]] # // THS
var_0x0055985E = TS_0x005B71A8[serial[0xB]] # // TS
var_0x00459204 = THS_0042D6CD[serial[0x9]] # // THS
var_0x005CEF77 = TS_0x0054AFF9[serial[0x8]] # // TS
var_0x005C191A = TS_0x004E19C5[serial[0xA]] # // TS
var_0x0057BE49 = TS_0x0055BF00[serial[0xF]] # // TS
var_0x005C5821 = TS_0x005CA23E[serial[0xE]] # // TS
var_0x005B9AE3 = TS_0x005C1BF2[serial[0xC]] # // TS
var_0x0058C52D = TS_0x004B765D[serial[0x1]] # // TS
var_0x0054FA8F = TS_0x005B8E5C[serial[0x4]] # // TS

```

EXTRACT TABLES

To extract completely the algorithm we need also to retrieve all the tables. There are a total of 595 different tables. To do this job, we have to parse **algo.py** automatically.

We can identify 2 cases:

- **TS and T16S**
- **THS**

For the first case it's quite easy to extract the values of each table. We can for example parse **algo.py** with an **IDAPython** script (**dump-THS.py**) and use the **Dword** function to retrieve values of the parsed table.

The second case is a little bit trickier, to retrieve the values of the handlers table we can use Pin. The idea is to execute each handler of all tables. To do that we can hijack the execution flow with **PIN_ExecuteAt** function and retrieve the first written value. In fact, the first written value corresponds to the handler's value:

```

unsigned int base_val = TH[posTH];
new_eip = (unsigned int *)(base_val+(offset*4));
value = new_eip[0];
PIN_SetContextReg(ctxt, REG_INST_PTR, value);
offset++;
PIN_ExecuteAt(ctxt);

```

The code of the Pin DLL is in **GetHandlersValue.cpp**. I think it's also possible to parse the disasm code of each handler with some regex, but this solution is not trivial.

CRYPTANALYSIS ON THE AES WHITEBOX

According to the previous section, the extracted algorithm is quite huge and disordered. We are not able to see anything and it's almost impossible to directly inverse it. In this section we will see what we can do to visualize the algorithm. The goal is to parse **algo.py** to generate a graph with **graphviz**. Then, we will see how we can do a cryptanalysis on the algorithm in order to find a serial from an output.

STATIC SINGLE ASSIGNMENT FORM

The algorithm is quite difficult to parse because this one doesn't follow the **SSA** form (Static Single Assignment). If we want to parse it directly, we have to take into account a lot of cases and it can cause a lot of bugs so it's not really convenient. To overcome this problem, we can translate our algorithm to the **SSA** form. To understand the SSA form, here is a small example:

```
var_0x0043ACE2 = THS_0043E740[serial[0xD]]  
  
var_0x005CEF77 = TS_0x0054AFF9[serial[0x8]]  
  
var_0x005CEF77 = T16S_0x00415EFE[(var_0x005CEF77 << 8) +  
                                var_0x0043ACE2]
```

In the SSA form:

```
var_1 = THS_0043E740[serial[0xD]]  
  
var_2 = TS_0x0054AFF9[serial[0x8]]  
  
var_3 = T16S_0x00415EFE[(var_2 << 8) +  
                        var_1]
```

We can parse **algo.py** to translate the algorithm to the **SSA** form (cf: **ssa-converter.py**).

GRAPH

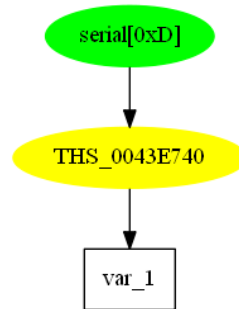
To visualize the algorithm, we can convert all the transitions of **ssa-algo-with-output.py** to a graph. For the graph, we can use **graphviz**, this tool is user friendly and the generated graphs are correct. Our goal is to produce a file for **graphviz** (.gv) from **ssa-algo-with-output.py**, so we still need to write a parser (**ssa-gengraph.py**). Our parser has two cases to handle:

- **TS** and **THS**
- **T16S**

For the **TS** and **THS** case we can represent this example:

```
var_1 = THS_0043E740[serial[0xD]]
```

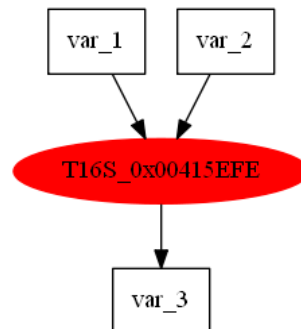
With this graph:



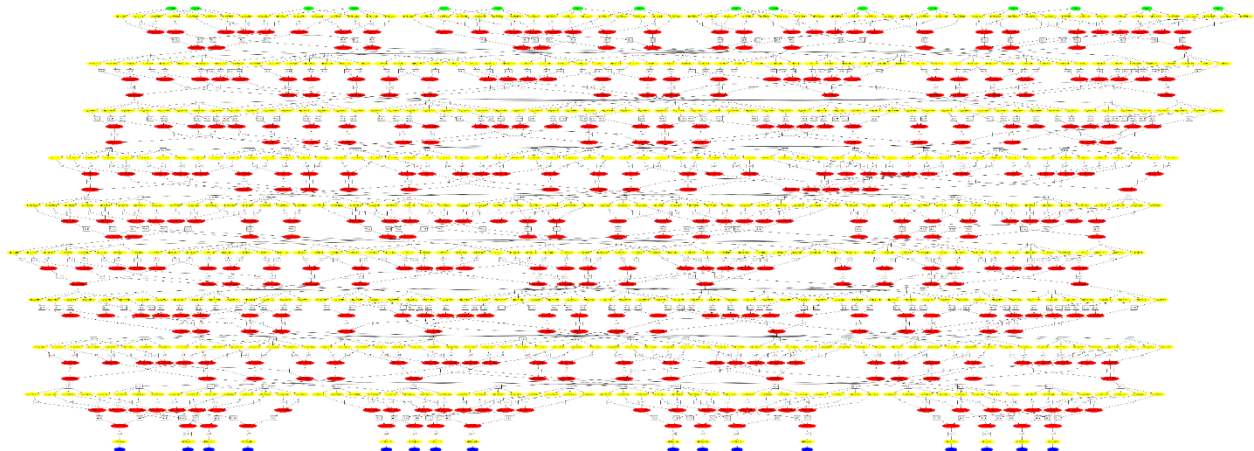
For the **T16S** case we can represent this example:

```
var_3 = T16S 0x00415EFE[(var_2 << 8) + var_1]
```

With this graph:



If all these transformations are applied for the whole algorithm, we obtain a really huge diagram ([ssa-algo.png](#)):

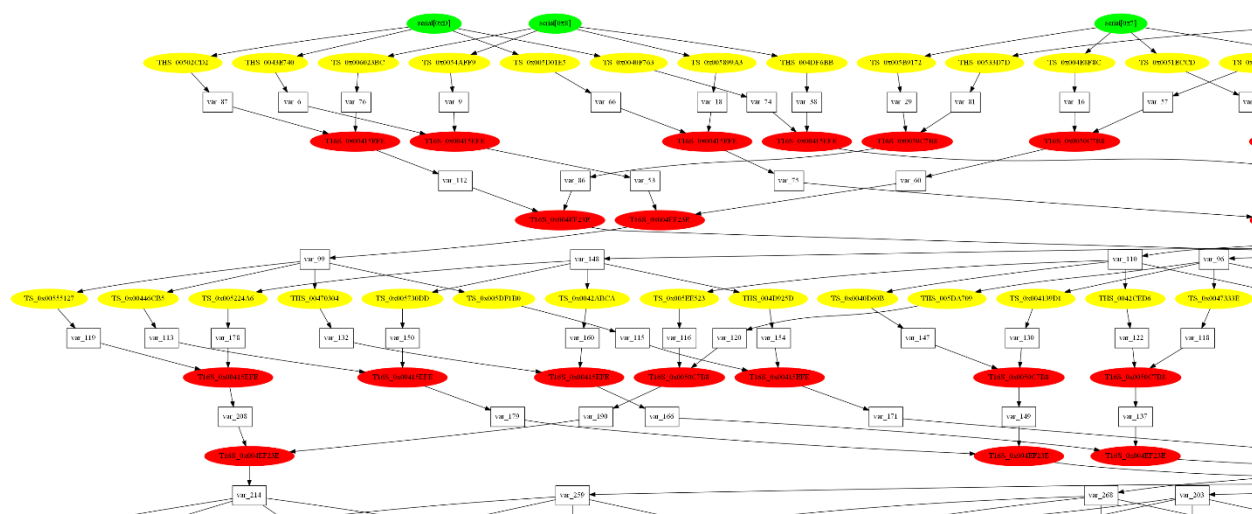


ROUNDS

The graph is useful because it provides a global view of the algorithm and it helps to get rid of the algorithm's disorder. We can clearly identify the rounds (a total of 10 rounds) and the round's operations.

Each round is composed of these operations:

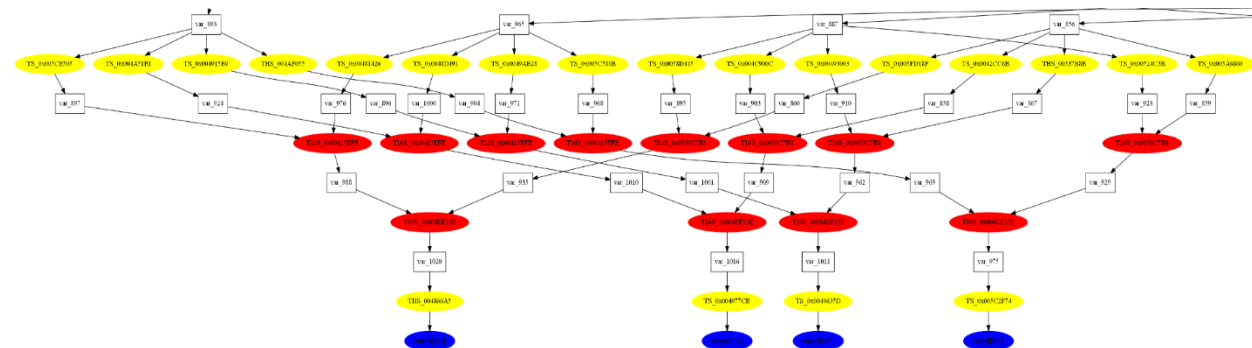
- begin with **TS** or **THS** operations
- finish with **T16S** operations (except the last round)



It's important to see that thanks to the **T16S** surjective operations, it's not that easy to inverse the algorithm. It's surjective because the input is 16-bit long and the output is 8-bit long.

CRYPTANALYSIS

When we focus on the penultimate round, we can clearly spotlight that 4 bytes of this round depend on 4 bytes of the previous round:



It's really interesting to see that because that means we can brute force 4 bytes of the third-to-last round. We brute force these bytes until the output's 4-bytes are equals to the penultimate

round's 4 bytes. It's also possible to brute force just 3 bytes but it's more complicated to automate than a cryptanalysis on the 4-bytes attack. We can retrieve this 4-bytes dependency in the AES algorithm with the **MixColumn** operation. Moreover the algorithm seems to have 10 rounds, and each states are 16 bytes long. So all these elements allow us to say that it's probably an **AES-128** Whitebox. This information was confirmed by the author. The good thing to know is that you need no knowledge in cryptography to attack the algorithm.

We can generalize the algorithm to brute-force with this pseudo-code:

```
b1 = T16S_0x004EF23E[T16S_0x0050C7B8[ T1[a1] << 8 + T2[a2] ] << 8 +  
    T16S_0x00415EFE[ T3[a3] << 8 + T4[a4] ] ]  
  
b2 = T16S_0x004EF23E[T16S_0x0050C7B8[ T5[a1] << 8 + T6[a2] ] << 8 +  
    T16S_0x00415EFE[ T7[a3] << 8 + T8[a4] ] ]  
  
b3 = T16S_0x004EF23E[T16S_0x0050C7B8[ T9[a1] << 8 + T10[a2] ] << 8 +  
    T16S_0x00415EFE[ T11[a3] << 8 + T12[a4] ] ]  
  
b4 = T16S_0x004EF23E[T16S_0x0050C7B8[ T13[a1] << 8 + T14[a2] ] << 8 +  
    T16S_0x00415EFE[ T15[a3] << 8 + T16[a4] ] ]
```

T i={1..16 } are arbitrary look-up tables corresponding to the previous **THS** and **TS** tables. In the previous pseudo-code **b1, b2, b3, b4** are known by the attacker and **a1, a2, a3, a4** are the bytes to brute-force.

AUTOMATED ATTACK

According to the previous part, we have to implement 4 brute-force per round. A brute-force lasts approximately 15 seconds. We have to rewind the 10 rounds of the algorithm to find a serial from an output. To do that we just need to brute force the first 9 rounds. The tenth round is easily invertible because there are just bijective operations (no **T16S** operations).

To resume, to find a serial from an output we need to do about 36 brute-force. It should last around 5 minutes. Furthermore it's really painful to extract a block to brute force from the whole algorithm by hand. It's also quite complicated to unroll the algorithm (in order to generalize it) because of the different **THS** and **TS** tables at each round. That's why it's difficult to write a generic brute-force and that's why we need to extract each block to attack from the algorithm.

To extract a block to brute-force we can proceed on a simple way. We reverse the lines of **algo.py** to **inv-ssa-algo.py**.

Extraction of a block to brute-force from **inv-ssa-algo.py**:


```

b1 = T16S_0x004EF23E[ var_7 << 8 + var_6]

var_7 = T16S_0x0050C7B8[ var_5 << 8 + var_4]

var_6 = T16S_0x00415EFE[ var_3 << 8 + var_2]

var_5 = T1[a1]

var_4 = T2[a2]

var_3 = T3[a3]

var_2 = T4[a4]

```

We parse the last file to extract each interesting lines with the **extract_deps** function. The arguments of this function are the output of a block (**b1, b2, b3, b4**), and because the structure of a block is always the same, our parser just needs a dictionary to extract the dependencies (**a1, a2, a3, a4**) and the needed algorithm.

Structure of an extracted block to brute-force:

```

for(int a1=0; a1 < 256; a1++) {
    for(int a2=0; a2 < 256; a2++) {
        for(int a3=0; a3 < 256; a3++) {
            for(int a4=0; a4 < 256; a4++) {

                ...

                [extracted block]

                if(b1p == b1 && b2p == b2 && b3p == b3 && b4p == b4)

                    goto ok;

            }}}

```

The script that creates the full brute-force is named **ssa-gen-bf-full.py**, it uses the **extract_deps** function that gives the inputs of a block from his outputs. The trick to automate the generation of the whole brute force is in fact simple. We extract the block to brute-force, if and only if, the

extracted dependencies (**a1, a2, a3, a4**) are equals to 4. If the number of extracted inputs is superior to 4 that means the inputs and the outputs are not linked and it's not possible to apply our attack. To automate the whole, we start from the 16 bytes of the penultimate round and we try all the combinations of 4 bytes. We extract a block to brute-force, if and only if, the number of inputs are equals to 4.

Here is the pseudo-code of the logic:

```
new_first_outputs = ["var_1020", "var_1016", "var_1011", "var_998", ..., "var_1005"]

for r in range(9):

    # extract 4 outputs from 16 outputs

    for piece in combinations(new_first_outputs, 4):

        inputs = extract_deps(piece[0], piece[1], piece[2], piece[3])

        if len(inputs) == 4:

            # 4 dependencies, we can brute-force it

            # extract the brute force

            output_bf_block()

            ...

            new_first_outputs.remove(inputs)
```

The **ssa-gen-bf-full.py** script generates around 2300 lines of brute-force corresponding to first nine rounds. We compile it and generate a serial for any digest.

wbLoop.exe:

```
var_95 = 122
var_168 = 6
var_114 = 184
var_106 = 65
serial_0x7_ = 50
serial_0x8_ = 2
serial_0x2_ = 88
serial_0xD_ = 119
serial_0x1_ = 127
serial_0xB_ = 179
serial_0xC_ = 241
serial_0x6_ = 28
serial_0x9_ = 182
serial_0x3_ = 181
serial_0x4_ = 251
serial_0xE_ = 48
serial_0x0_ = 34
serial_0xA_ = 37
serial_0x5_ = 222
serial_0xF_ = 20
serial = 22 7f 58 b5 fb de 1c 32 2 b6 25 b3 f1 77 30 14
```

To conclude, we find a way to attack the algorithm and find a serial from an output. It's almost impossible to do this approach without automation. An interested task to do is to extract the **AES key** from the algorithm with a **BGE** attack. With this approach and the **AES key**, you can write a keygen with one-line of Python (from Crypto.Cipher import AES).

CONCLUSION

To conclude, we learned from this challenge that from an obfuscation automatically generated, there is always a way to unobfuscate it automatically. We also learned that with a huge amount of data/code, we always need to find a way to automate the process (we developed a lot of parsers). Finally, we also saw how to sort a disordered algorithm with a graph and we saw how to break an **AES whitebox** without any knowledge in cryptography.

I hope this paper gave you some elements on the approach for the challenge and for reversing a cryptographic algorithm in general. I'm sure that my ideas are not the best to break the obfuscation and the **AES whitebox** and I'm really interested to have some feedback/ideas to improve the methodology.