Detection of cryptographic algorithms with `grap`

Léonard Benedetti[a], Aurélien Thierry[a] and Julien Francq[a]

[a]Airbus CyberSecurity

MetaPole, 1 bd Jean Moulin, CS 40001, 78996 Élancourt Cedex, France

`benedetti@mlpo.fr`

`{aurelien.thierry,julien.francq}@airbus.com`

Abstract

The disassembled code of an executable program can be seen as a graph representing the possible sequence of instructions (Control Flow Graph). `grap` is a YARA-like tool, completely open-source, and able to detect graph patterns, defined by the analyst, within an executable program.

We used `grap` to detect cryptographic algorithms: we created patterns for AES and ChaCha20 that are based on parts of the assembly code produced by compiling popular implementations (available in LibreSSL and libsodium). Our approach is thus based on the algorithms and their structure and does not rely on constant detection.

Identifying cryptographic algorithms used by an executable has multiple applications. It can be used to detect features implemented within the binary ("this program uses AES", "this binary can verify cryptographic signatures"). Within a platform performing automated analysis one aim can be to extract cryptographic material (the AES key used, a non-standard S-Box). Finally, integrated with existing tools (IDA plugin) it can help a reverse-engineer focus on found areas ("this subroutine looks like a cryptographic function") or avoid wasting time on known algorithms ("this function implements ChaCha20").

We used `grap` [TT17a; TT17b] to create detection patterns that are based on the control flow graph of the binaries in order to focus on instruction and flow matching and offer an alternative to constant detection.

The paper is organized as follows. First there is an overview of `grap` with simple examples and a dive into its capabilities and the matching algorithm. Then we explain how we created patterns for AES and ChaCha20, and give insights on advantages and disadvantages of a detection based on CFG matching.

1 `grap` overview

`grap` takes as input patterns and binary files (PE, ELF or raw binary code), uses a Capstone-based [QDNV] disassembler to determine the CFGs of binaries (only x86 and x86_64 are supported) and detects the patterns in these CFGs. The patterns are graphs, defined by the user, composed of conditions on the instructions ("opcode is `xor` and *arg1* is `eax`") and their repetitions (3 identical instructions, one basic block, etc.).

Graph patterns and root node. The patterns are written as text files (.dot) in a sub-language of the DOT language [Dot]. Due to the algorithm used (see section 1.2) there needs to be a unique way to traverse each pattern graph, so we use directed graphs that have a root node, this is a node from which every other node is reachable.

Ordered children. On x86 assembly instructions may have 0, 1 or 2 children. Graphs obtained through `grap` disassembly have numbered children, the child numbered 1 is the following instruction (next address) and the child numbered 2 is a remote instruction. Due to this there might be a child numbered 2 and no child numbered 1. The following table shows a few instructions and the children that will be defined by disassembly.

Instruction	Child 1	Child 2
`ret`		
`mov eax, 0x1`	following instruction	
`call 0x405ed2`	following instruction	instruction at address 0x405ed2
`jne 0x4060ca`	following instruction	instruction at address 0x4060ca
`jmp 0x4043bf`		instruction at address 0x4043bf

1.1 Usage

Prototype patterns. The command line interface allows for quick prototyping of patterns without writing a DOT file. For instance, looking for a simple encryption pattern used by Backspace [TT17a] consisting of a `mov` instruction followed by a `sub`, followed by a `xor` then another `sub` can be done with the following command.

```
grap "mov->sub->xor->sub" malware.exe
```

It infers a correct pattern file and feeds it to the matching engine. It is intended as a way to create small patterns quickly and does not support all options of pattern files. Adding the `-v` option will output the path of the temporary pattern file.

Write patterns. Figure 1 shows a typical **xor** decryption loop in assembly and its CFG. It has two `mov` instructions that set the size (0x15) and the address (0x80490c5) of an encrypted string, followed by a loop doing a `xor` decryption with value 0x7f. In this implementation `eax` points to the encrypted string while `ecx` is used as a counter.

As explained previously, the children of each node are numbered: only `jne` has a child numbered 2 (the first instruction of the loop), the other instructions only have a child numbered 1.

```
mov ecx, 0x15
mov eax, 0x80490c5

loop:
xor byte [eax], 0x7f
inc eax
dec ecx
jne loop
...
```

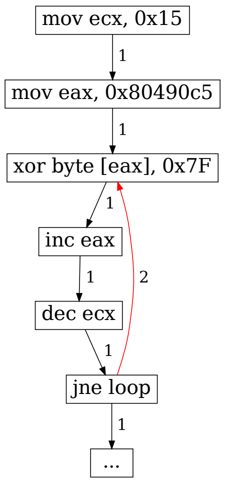


Figure 1: `xor` decryption loop and its CFG

Let us say we want to detect variants that:

- use various methods to set the size (**ecx**);
- use a different value for the `xor` decryption;
- do not use `ecx` as a counter;
- have some more instructions in the loop.

We also need to get the `xor` value used for decryption. A pattern able to detect such variants and perform extraction is given on Figure 2.

The pattern first defines nodes. Node A will match a `mov` instruction. The **root=true** option specifies that node A is the root node. Node B matches a `xor` instruction whose first argument (**arg1**, which

```
digraph xor_loop_pattern {
A [cond="opcode is mov", root=true]
B [cond="opcode is xor and arg1 contains [eax]", getid="xor"]
C [cond="opcode is inc and arg1 is eax"]
D [cond=true, minrepeat=1, maxrepeat=4, lazyrepeat=true]
E [cond="opcode beginswith j", minchildren=2]
F [cond=true]

A -> B
B -> C
C -> D
D -> E
E -> F [childnumber=1]
E -> B [childnumber=2]
}
```

Figure 2: `xor` loop detection pattern

is seen as a string) contains "`[eax]`", which means the instruction overwrites the content of `[eax]`. The **getid** option of node B flags the matched instruction for extraction: the extracted element will be named "xor". Node C matches `inc eax`. Node D matches any instruction ("**cond**=true" is always verified) repeated 1 to 4 times. The **lazyrepeat=true** option of node D indicates that the repetition shall stop once any of the child nodes' condition is fulfilled: node D has only one child (node E) so node D will be repeated until a node with an opcode beginning with "j" and with at least two children is reached. Node E matches a conditional jump: opcode begins with "j" and at least two children. Node F matches any instruction.

Note that repetitions can only be matched within a basic block: the first matched instruction must have exactly 1 child (numbered 1), the middle instructions exactly 1 parent and 1 child (numbered 1), and the last instruction exactly 1 parent.

Then the edges are defined. The first five edges specify that nodes A, B, C, D, E and F shall follow each other sequentially: A has one child numbered 1 which is B; B has one child numbered 1 which is C, etc. When the **childnumber** option is not set, it is inferred: the first defined child is numbered 1, then 2. The last line specifies that node E has a child numbered 2 which is node B; this defines the loop structure.

Node and edge options, condition syntax. In addition to the options explained in the previous example, more node options (Figure 3) are available. One edge option is available (**childnumber**), its value can be 1, 2 or, for pattern graphs, the wildcard `*`.

The syntax used to write conditions (defined by the "**cond=**" node option) distinguishes fields (Figure 4) based on their type: boolean, number or string. Note that **basicblockend** is a property and does not take a value. Supported operators on numbers are `==`, `>=`, `>`, `<=` and `<`. Boolean operators are **true**, **not**, **and**, **or**. String operators are **is** (exact match), **substring**, **beginswith** and **regex**.

More details on options and examples can be found in the document **grap_graphs.pdf** in the download section of the repository [Gra].

Field name	Possible values	Default	Description
root	**true**	*None*	Specifies which node is the pattern's root
cond	String	*None*	Condition to match against (see fields in Figure 4)
minrepeat	Number	**1**	Minimum repeat number
maxrepeat	Number	**1**	Maximum repeat number
repeat	Number, *****, **+** or **?**	**1**	Repeat number (***** is minimum 0 with no maximum, **+** is minimum 1 with no maximum and **?** is minimum 0 with maximum 1)
lazyrepeat	**true** or **false**	**false**	Aims to repeat a minimum (**true**) or a maximum (**false**) number of times
minfathers	Number	**0**	Minimum number of incoming edges
maxfathers	Number	*None*	Maximum number of incoming edges
minchildren	Number	**0**	Minimum number of outgoing edges
maxchildren	Number	*None*	Maximum number of outgoing edges
getid	String	*None*	Marks the instruction(s) for extraction with a specified identifier

Figure 3: Node options for pattern graphs

Field name	Value type	Description
inst	String	Full disassembled instruction (text)
opcode	String	Mnemonics of the disassembled opcode
nargs	Number	Number of arguments
arg1	String	First argument (if any)
arg2	String	Second argument (if any)
addr	Number	Address (VA) of the instruction
basicblockend	(No value)	**true** when the instruction is the end of a basic block: • it is an unconditional jump, or • it does not have exactly one child (e.g. *ret*, or *jne* for instance), or • its child has two or more parents.

Figure 4: Condition fields

grap options. The following command launches the detection of the previous pattern (Figure 2), saved in a file “xor_loop.dot”, on a small binary “./xor32”. The output of the command is written below.

```
grap xor_loop.dot ./xor32
```

```
./xor32.dot - 115 instructions
1 match in ./xor32.dot:  xor_loop_pattern (1)

xor_loop_pattern, match 1
xor:  0x804808a, xor byte ptr [eax], 0x7f
```

Note that by default `grap` will:

- Create a .dot file in the same directory as the binary to cache the disassembled CFG
- Output information on the analyzed file and the matched patterns
- Output the extracted instructions

The quiet (**-q**) option limits the output to one line per matching binary and shows only the number of matches per pattern, it does not output the extracted nodes.

```
grap -q xor_loop.dot *
```

```
xor32.dot - xor_loop_pattern (1)
xor32_small.dot - xor_loop_pattern (1)
loops.dot - xor_loop_pattern (3)
```

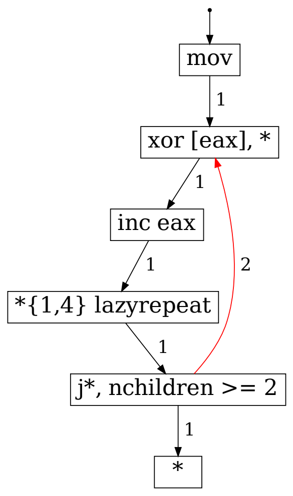
IDA plugin. We have developed an IDA plugin that allows the detection and visualization of results directly from the IDA interface. The IDA plugin also has a feature for interactive creation of patterns which, from instructions chosen graphically by the analyst, creates a pattern including these instructions. Our goal is to make pattern creation as intuitive as possible by allowing users to choose the level of accuracy and genericity (instruction, opcode, arguments, etc.) they want in their patterns.

Python bindings. Python bindings are available to create `grap`-based scripts to extract relevant information (*e.g.*, addresses, instructions) from the detected parts of graph.

1.2 Matching algorithm

1.2.1 Pattern and traversals

From a pattern to a traversal. A graph with a root node and ordered children and can be encoded with a unique traversal description. Figure 5 shows the graph structure from the previous pattern (Figure 2) with the root node at the top and its associated traversal.



$$1 : mov \xrightarrow{1} 2 : xor\ [eax], * \xrightarrow{1} 3 : inc\ eax \xrightarrow{1} 4 : (*)^{\{1,4\}}_{lazyrepeat} \xrightarrow{1} 5 : j*, nchildren \geq 2 \xrightarrow{1} 6 : * \xrightarrow{R} 5 \xrightarrow{2} 2$$

Figure 5: Pattern and its traversal

The traversal is obtained through a depth-first search from the root node:

- Number the root **1**, the root node is a `mov`, write $1 : mov$
- Follow the first child (1) and find a `xor` instruction, number it **2**, write $\xrightarrow{1} 2 : xor\ [eax], *$
- Follow the first child (1) and find an `inc` instruction, number it **3**, write $\xrightarrow{1} 3 : inc\ eax$
- Follow the first child (1) and find a repetition of 1 to 4 (any) instruction with **lazyrepeat=true** option, number it **4**, write $\xrightarrow{1}\ 4 :\ (*)^{\{1,\ 4\}}_{lazyrepeat}$
- Follow the first child (1) and find a jump instruction, number it **5**, write $\xrightarrow{1} 5 : j*, nchildren \geq 2$
- Follow the first child (1) and find any instruction, number it **6**, write $\xrightarrow{1} 6 : *$

- Jump back to node 5 (`j*`) because it had a second child, write $\xrightarrow{R} 5$
- Follow the second child (2) and find an already defined node (**2**), write $\xrightarrow{2} 2$

Thus, with these conditions, graphs have a unique representation as flat objects (traversals). This makes a huge difference with graphs in the general case and will allow our subgraph isomorphism algorithm to run in polynomial time instead of exponential time.

Traversal and isomorphism. We have now encoded our pattern into a traversal. Finding a graph isomorphism between the pattern and a subgraph of the binary's CFG is equivalent to finding a node of the CFG from which it is possible to perform the traversal of the pattern within the CFG. This property holds true for traversals generated with a depth-first search because the children are numbered and the pattern graphs have a root node [Thi15].

Thus a match is a node of the CFG fulfilling all the following conditions; once a node or block is reached, it becomes the new current node.

- Its opcode is `mov`, let us number it **1**.
- The current node has a child numbered 1 that is a `xor [eax], *`, let us number it **2**.
- The current node has a child numbered 1 that is `inc eax`, let us number it **3**.
- The current node has a child numbered 1 that is block of 1 to 4 any instructions, except for instructions whose opcode begins with j and has 2 children, let us number this block **4**.
- The current block has a child numbered 1 that is an instruction whose opcode begins with j and has 2 children, let us number it **5**.
- The current node has a child numbered 1 that is any instruction, let us number it `6`.
- Let us go back to the block numbered **5**.
- The current node has a child numbered 2 that is the block numbered **2**.

1.2.2 Handling multiple patterns

Traversal tree. For performance purposes, the patterns are sorted into a traversal tree. Figure 6 shows a traversal tree with two patterns. Each leaf of the tree is a unique pattern.

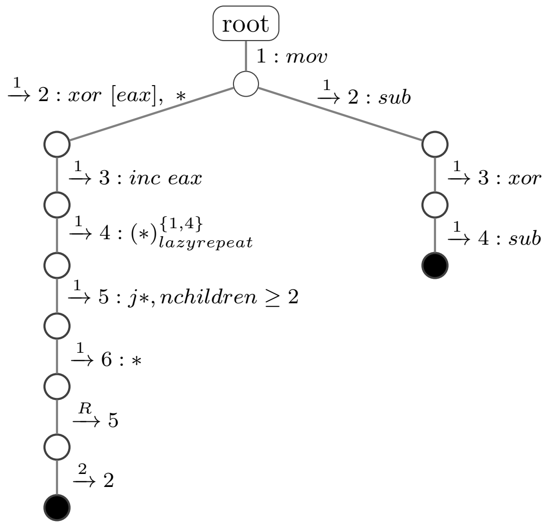

root
$1 : mov$
$\xrightarrow{1} 2 : xor\ [eax],\ *$
$\xrightarrow{1} 2 : sub$
$\xrightarrow{1} 3 : inc\ eax$
$\xrightarrow{1} 3 : xor$
$\xrightarrow{1} 4 : (*)^{\{1,4\}}_{lazyrepeat}$
$\xrightarrow{1} 4 : sub$
$\xrightarrow{1} 5 : j*, nchildren \geq 2$
$\xrightarrow{1} 6 : *$
$\xrightarrow{R} 5$
$\xrightarrow{2} 2$

Figure 6: Traversal tree with xor loop pattern (Figure 5) and `mov->sub->xor->sub`

Unknown child numbers. Patterns can specify that the child number of an edge is unknown (* instead of 1 or 2). When creating the traversal tree, those children will be duplicated (two patterns will be created: one with **childnumber=1** and one with **childnumber=2**). Thus if a pattern contains n wildcards, 2^n patterns will be added to the traversal tree.

Wildcards should be used sparsely since they will affect performance; this is an expected issue because the speed of the algorithm comes from ordered children.

Matching algorithm. Algorithm 1 matches a CFG from one specified node against a traversal tree. `grap` uses this algorithm from every node of the CFG to find pattern matches. Note that the traversal tree's nodes are labeled and the label satisfies the regex $[\xrightarrow{\alpha}]i[: C]$ where α can be either a number or R, i is an integer and C is a condition, as seen on Figure 6.

The function `leavesListRec` traverses recursively the traversal tree to find possible moves within the CFG, calling each time the `step` function to determine is a move is possible based on the current node `n`, its condition `C`, the traversal node's label `w` and the CFG `G`. Numbering a node n with number i is noted $G[i] \leftarrow n$. At the beginning of the search (when calling `leavesList`) no node has been numbered in `G`: $\forall i$, $G[i]$ does not exist.

Node repetition. In order to handle node repetition, the actual algorithm is a bit different when repeat options are used. Each node numbering (for instance $G[i] \leftarrow n$) will iterate as many instructions as possible that verify :

- they are in the same basic block: they have exactly one child (numbered 1) and one parent, and
- they fulfill the condition C, and
- the maximum number of repetition has not been reached yet.

If the node should have been numbered i, the j instructions within the block are actually numbered $i.1$, $i.2$, ..., $i.j$.

Besides, a return edge ($\xrightarrow{R} i$) will set the new current node to the last matched node of the block ($i.j$). Thus a block can only be defined once and will not be traversed twice. This is consistent with the behavior on a single instruction block whose condition cannot be defined twice.

Lazyrepeat. When building a traversal from a pattern, the condition assigned to a node with the **lazyrepeat=true** option will be computed: if the node's condition is n_{cond} and its first child's condition is c_{cond}, then the updated condition will be $n'_{cond} = n_{cond}$ *and not* c_{cond}. This step is performed prior to Algorithm 1.

This allows the repeat options to work with the **lazyrepeat=true** option: repetition will stop as soon as the condition of the child node is fulfilled.

Algorithm 1: Lists the patterns matched in a CFG, within a traversal tree, from one of the CFG's nodes

Data: A traversal tree (T) with a root node (R), a control flow graph (G) and a node (N) from G

Result: A list of the leaves from the traversal tree, each leaf being a matched pattern

```
leavesList(T, R, G, N)
    return leavesListRec(T, R, G, N)

leavesListRec(T, t, G, N)
    $(leaves, n) \leftarrow ([], N)$
    if $t$ is a leaf of $T$ then
        $leaves \leftarrow [t]$
    end
    foreach $u$ child node of $t$ in $T$, labeled $w$ do
        $(possible, n', G') \leftarrow$ step$(w, n, G)$
        if $possible$ then
            $leaves \leftarrow leaves$ @ leavesListRec$(T, u, G', n')$
        end
    end
    return $leaves$

step(w, n, G) // Label w looks like regex $[\xrightarrow{\alpha}]i[: C]$
    if $w$ looks like $1 : C$ then
        if $n$ fulfills condition $C$ then
            G[1] $\leftarrow$ n
            return (true, $n, G$)
        end
        return (false, $n, G$)
    else
        if $\alpha$ is an integer $k$ then
            if $n$ has a child $n'$ numbered $k$, there is no $j$ in $G$ such that $G[j] == n'$, $n'$ fulfills
             condition $C$, and $G[i]$ does not exist then
                $G[i] \leftarrow n'$
                return (true, $n', G$)
            else if $n$ has a child $c$ numbered $k$ and $G[i]$ exists with $c == G[i]$ then
                return (true, $n', G$)
            else
                // If another node is already $G[i]$, or
                // if the node to be numbered in $G$ has already another number in $G$,
                   or
                // if the node does node fulfill $C$, or
                // if there is no child numbered $k$
                return (false, $n, G$)
            end
        else $w$ looks like $\xrightarrow{R} i$
            if $G[i]$ exists then
                return (true, $G[i], G$)
            end
            return (false, $n, G$)
        end
    end
```

2 Detection of cryptographic algorithms

2.1 State of the art

There are already several tools for detecting cryptographic algorithms in binary programs. We present here some of them by briefly describing their mode of operation, then we will explain how the `grap` approach differs.

YARA is a tool that allows to describe rules, in the form of binary or textual patterns, which can be checked on binary programs [BMBAS]. Even though this tool is mainly used for the purpose of classifying and detecting malware, it is possible to write rules to detect particular constants or instructions relating to cryptographic operations.

Signsrch is a tool for searching for known signatures in binary files [Aur]. It is thus able to detect several algorithms using constants (encryption, compression, etc.).

FindCrypt2 is an IDA plugin that searches for known constants of cryptographic algorithms [Gui]. For example, for several symmetric-key encryption algorithms, it uses the constants of the associated S-Box (*substitution box*).

IDAScope is an IDA plugin that uses this very same technique of searching for constants and searches the binary areas which present a large number of arithmetic and logical instructions [Plo]. These areas are then returned to the analyst as being likely to contain cryptographic code (such a technique is based on the idea that a cryptographic algorithm will necessarily use many instructions of this type).

Aligot is a dynamic tool for detecting cryptographic algorithms, even if the binary is obfuscated [Cal]. Its mode of operation consists in detecting dynamically the loops and then in sending as parameters of these loops some input values in order to test whether the output values are the same as those of a known algorithm for those input values [CFM12].

All of these techniques have advantages and disadvantages: conventional static methods strongly depend on the constants being directly readable and unchanged. Thus, in the case where these constants are encrypted or obfuscated (for example dynamically calculated), these methods do not detect the desired algorithms.

Aligot does not have this disadvantage, but its technique is not compatible with binaries of large size and is difficult to use to test a large number of binaries because loop detection can take a long time. Moreover, very slight modifications of the algorithm (*e.g.* change of the constants of the S-Box) make detection with this technique impossible.

2.2 Use of `grap` for detection of cryptographic algorithms

`grap` uses a different technique, based on the analysis of control flow graphs. Since our approach consists in writing patterns for the assembly code of cryptographic functions, we do not rely on constants: detection is therefore not affected if they are not readable or have been slightly modified. This is also a fairly fast technique once the disassembly has been performed.

But on the other hand, it is necessary to create accurately designed patterns (which requires to understand and study the algorithm to be detected). Furthermore this technique depends on the quality of the disassembly; and since detection is done at the assembly level, the pattern is sensitive to the chosen compilation options (optimizations, compiler choice, etc.).

Several reasons can make different implementations of the same algorithm remarkably different, depending on the purpose and the constraints. Thus, performance, security or even modularity reasons may lead to the same algorithm to be implemented with very different source codes which, once they are compiled, will produce disparate assembler codes. This makes designing a generic pattern more complicated: it is necessary to look for characteristic invariants common to different implementations.

This analysis is done on the CFGs of several compiled implementations. It deals with the semantics of the operations of those graphs, but also with their structure. One can indeed observe with several algorithms strongly structuring elements (notably the round functions, used in many cryptographic algorithms) which are found in the form of the graphs after compilation.

In the following we will detail a few practical cases that show how we proceeded and the limits we faced.

2.3 AES pattern

In order to write a pattern for AES, we started from several reference implementations available in LibreSSL [The] and searched for invariants in the code generated by different compilers with different optimizations in different contexts.

To this end, the designed patterns seek to detect the characteristic shifts made during the AES `ShiftRows` step and the multiplication of polynomials within the Galois field (which corresponds to the linear transformation used in AES) that occurs during the `MixColumns` step. Finally, we also detect trivially the use of specialized x86 instructions relating to AES (*AES-NI*).

Figure 7 shows a simplified graphical representation of a pattern for the detection of AES. It can be noted that it is mainly arithmetic and logical instructions that are used to match this implementation.

It can also be seen that the logic of AES operation can be observed directly: it starts with an initialization phase (`InitialRound`), then the rounds are processed within a loop (the right branch), until arriving at the final round (the left branch) that differs in the absence of the `MixColumns` step.

It is interesting to note that thereby what is matched will be similar in its control flow to that of the algorithm studied. So even if the algorithm has been modified or is a particular variant, this similarity could facilitate the research of the analyst.

The representation of Figure 7 is simplified; in order to make the detection more accurate and to reduce the number of potential false positives, we use other elements in the pattern, such as the value of certain operands (when they are generic) or possible variants of the opcode (`shr`/`ror` for example).

The code of the pattern that matches a portion of the `MixColumns` step in one of the studied AES implementation is reproduced below. On this part of the pattern, one can see that opcodes are used but also the value of one of the operands (here, it is to match binary masks). This illustrates the possibilities offered by `grap` with regard to the genericity of the patterns. It is indeed possible to use different options (RegEx for instructions, variable number of repeats, non-constant operands, etc.) that make a pattern more or less granular.

```
L [cond="opcode is 'and' and arg2 is 0x80808080", repeat=2]
M [cond=true, lazyrepeat=true, repeat=*]
N [cond="opcode is 'and' and arg2 is 0xfefefefe", repeat=2]
O [cond=true, lazyrepeat=true, repeat=*]
P [cond="opcode is 'and' and arg2 is 0x1b1b1b1b", repeat=2]

L -> M
M -> N
N -> O
O -> P
```

Our presentation will give feedback on the detection of patterns at assembly level and on the use of `grap` for this purpose. We will also discuss in detail how to write rules to efficiently detect cryptographic algorithms and the advantages and disadvantages of `grap` compared to existing alternatives.

2.4 ChaCha20 pattern

ChaCha20 is a symmetric encryption algorithm and a stream cipher [Ber08]. This encryption algorithm is less common than AES, but its use is increasing. According to the best cryptanalyses published to this day [AFKMR08], it offers a high level of security, it is also remarkably fast [Ber] and its implementation is relatively simpler than most other encryption algorithms usually used.

It is an ARX algorithm (*add-rotate-xor*): its round function uses only these three operations, addition, rotation and XOR.

On the structural aspect, it does not involve very specific logic; it uses two nested loops that contain just the block generation with the round function. The writing of a pattern cannot therefore be done on the basis of this very generic arrangement.

However, concerning the semantics attached to the round function, it can be seen that it is quite specific (first an addition, then a XOR, then a rotation) and its operations are interdependent: the operation XOR involves the result of the addition, and the rotation operation involves the result of the XOR. Hence we can conclude that it is difficult for a developer or a compiler to rearrange the order of

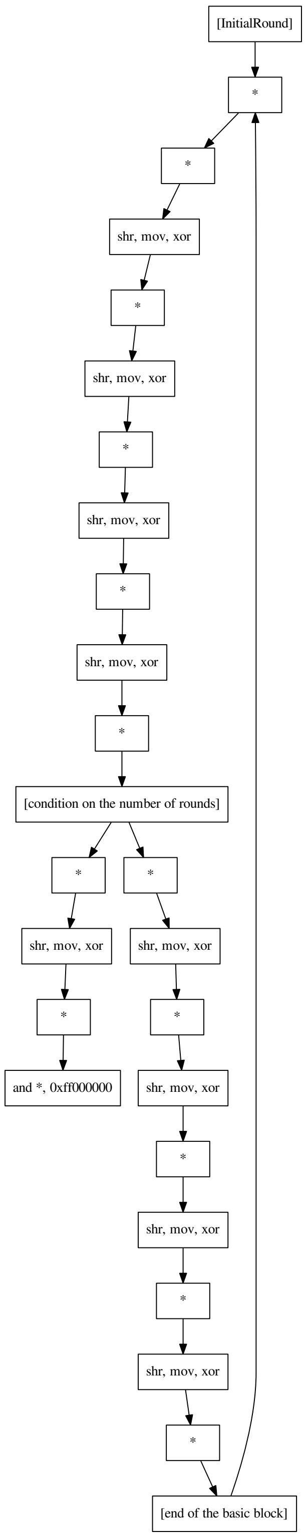


Figure 7: A simplified graphical representation of a pattern that allows the detection of an AES implementation. The general structure of the algorithm is preserved.

instructions. Since these very same operations are fairly simple, there is no opportunity to carry out arithmetic simplifications.

We can therefore propose a pattern to detect this ARX logic, by looking for the places where these three instructions are sequentially used a large number of times (a block of `add` instructions, then a block of `xor`, then a block of `rol`), possibly with instructions for memory handling (`mov` or `lea`) inserted between them.

A summary of this pattern is given below.

```
digraph ARX_crypto{
 Op1 [cond="opcode is add", repeat=+, getid="ARX_crypto"]
 Op1M [cond="opcode is mov or opcode is lea", repeat=*]
 Op2 [cond="opcode is xor", repeat=+]
 Op2M [cond="opcode is mov or opcode is lea", repeat=*]
 Op3 [cond="opcode is rol", repeat=+]
 Op3M [cond="opcode is mov or opcode is lea", repeat=*]
 [...]

 Op1 -> Op1M
 Op1M -> Op2
 Op2 -> Op2M
 Op2M -> Op3
 Op3 -> Op3M
 [...]
}
```

This example illustrates that in some cases, assembly-level detection allows interesting behaviors to be characterized fairly simply with relatively concise patterns.

2.5 Pattern usage

Performance. We tested our patterns on five compiled files from libsodium [Den] and LibreSSL [The] with different compiler options. The following performance tests were performed on a laptop with an Intel i5-2430M dual-core processor (2.40 GHz).

Library	Compiler	Disassembly time	CFG size	Matching time
libsodium 1.0.12 (18.2.0)	GCC	2.1 seconds	51,866 instructions	0.6 second
LibreSSL 2.5.4 (41.1.0) x64	Clang -O3	8.0 seconds	172,293 instructions	1.5 seconds
LibreSSL 2.3.4 (37.0.0) x64	GCC -O3	7.2 seconds	191,307 instructions	1.6 seconds
LibreSSL 2.3.4 (37.0.0) x64	GCC -O0	10 seconds	318,160 instructions	2.6 seconds
LibreSSL 2.3.4 (37.0.0) x86	GCC -O0	10 seconds	346,416 instructions	2.9 seconds

The following command launches detection of the AES and ARX patterns (*pattern.dot* combines *patterns/crypto/aes_libressl_v0.1.dot* and *patterns/crypto/arx_crypto.dot*). Thanks to multithreading, the overall computing time for disassembly and detection is around 23 seconds.

```
grap -q ../pattern.dot *
```

```
libsodium.so.18.2.0.dot - AES_NI (106), ARX_crypto (3)
x64_libcrypto.so.41.1.0_clang_O3.dot - ARX_crypto (64), LibreSSL_AES_compact (1)
x64_libcrypto.so.37.0.0_O3.dot - ARX_crypto (12), LibreSSL_AES_common (1)
x64_libcrypto.so.37.0.0_O0.dot - ARX_crypto (58), LibreSSL_AES_common (2)
x86_libcrypto.so.37.0.0_O0.dot - ARX_crypto (58), LibreSSL_AES_common (2)
```

Discussion. Those observed times seem very reasonable compared to the relatively large size of the binaries and taking into account the fact that the disassembly is a one-time operation.

One can see on libsodium that the detection of AES relies exclusively on the specialized instructions (*AES-NI*), which is consistent insofar as libsodium only uses hardware-accelerated implementations of AES.

It can also be noted that the number of matches on the `ARX_crypto` pattern is sometimes significant. This is due to the fact that this pattern aims to detect the generic ARX construction found in ChaCha20, but which can also be found in other cryptographic algorithms. It is a desirable behavior, thereby several variations of ChaCha20 and other cryptographic algorithms are detected, the analyst then being able to study these zones in particular in order to identify the algorithm or the variant.

2.6 Discussion, limits and perspectives

Creating patterns that are both effective and generic is not always feasible. One is sometimes confronted with algorithms that do not have invariant elements, semantic or structural, that can be used to precisely characterize different implementations.

For instance, we were not able to produce a generic pattern for SHA-2 (SHA-256, etc.) without relying on its constants because of the absence of invariants on different implementations and compilations. The same problem was encountered on RC4 in a previous study [TT17a].

Another important limitation we have identified is the optimization of implementations using SIMD instructions (Single Instruction, Multiple Data). These instructions, which are processor-dependent, can be used by the compiler or written directly by the developer for performance purposes. This involves using a single instruction, but which performs an operation on multiple data at once.

Such a transformation (called vectorization) profoundly modifies the semantics (and potentially the structure) of an algorithm. To properly handle this case with our approach, it is necessary to design specific patterns for this category of implementations.

The tests carried out on different versions of the same implementations make it possible to conclude on the efficiency and the genericity of the patterns. Nevertheless, it would be interesting to take this study further by focusing on the accuracy and robustness of detection on a dataset in order to have a more precise idea of the genericity of the pattern. For this purpose, the confusion matrix (true and false positives and negatives) could be studied in order to quantify more precisely the performance of our approach for detection.

That being said, the question of the dataset to use for such a study remains open for now. Indeed, the context and the algorithms to be detected profoundly alter the results obtained, it is therefore necessary to use a dataset that is relevant to these constraints. One might suggest using a ransomware database that use encryption algorithms for which a `grap` pattern has been designed, but it should be noted that they often use the CryptoAPI provided by the operating system, and do not necessarily represent the most interesting use case of this approach.

Finally, in order to improve the `grap` detection capabilities of algorithms and to be able to treat some cases more efficiently, we have considered making `grap` able to detect some kind of “metapatterns”.

This would allow a pattern to be defined with other patterns (hence subgraphs could be used inside a pattern). It would ideally be possible to add simple logic rules on them (optionality, repetition, etc.).

3 Install and use `grap`

The tool is free (MIT license) and available online [Gra]. The build process (`cmake` then `make`) and the installation process (`sudo make install`) target Ubuntu 16.04 (latest LTS) but should be applicable to any recent GNU/Linux distribution. The installation and use of the IDA plugin have been tested with Windows 7 with IDA Pro 6.8 and 6.9.

The following command (once the installation has been completed) could be used to test the detection pattern of AES on LibreSSL and other binaries:

```
grap patterns/crypto/aes_libressl_v0.1.dot <binary to test>
```

The files mentioned in this paper, which can be used for testing, are freely available here: `https://bitbucket.org/cybertools/grap/downloads/`.

Conclusion

We have detailed the operation of `grap` and how it is possible to use it to detect and classify (cryptographic) algorithms within executable programs. This can be accomplished by analyzing the structure (i.e. the topology of the graph) and the semantics (the meaning of the instructions) of the CFGs.

This approach differs from conventional methods used, which generally rely on constant detection. This makes possible, in particular, to take into account the structural aspect of the studied algorithms, and consequently it is able to detect variants which present only slight semantic modifications.

It should be noted that such a technique is not robust against an attacker who actively seeks to avoid detection (by obfuscation or deliberate modifications for this purpose). However, it is a complementary approach to existing ones with its own advantages and mode of operation.

The results obtained on cryptographic algorithms particularly used nowadays (AES, ChaCha20) show the possibilities offered by `grap` to recognize interesting behaviors with graph-based patterns. Beyond the detections described in this article, it is possible to use `grap` in different contexts, in an automated or punctual way. For example, to look for patterns within unknown binaries at the assembly level, classify them based on matched algorithms (cryptographic or not), or identify malware families.

References

[AFKMR08] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. “New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba”. In: *Fast Software Encryption: 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers.* Edited by Kaisa Nyberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 470–488. DOI: `10.1007/978-3-540-71039-4_30`.

[Aur] Luigi Auriemma. *Signsrch.* `http://aluigi.altervista.org/mytoolz.htm`.

[Ber] Daniel J. Bernstein. *Salsa20 speed.* `https://cr.yp.to/snuffle/speed.pdf`.

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20.* `https://cr.yp.to/chacha/chacha-20080128.pdf`. 2008.

[BMBAS] Hilko Bengen, Joachim Metz, Stefan Buehlmann, Victor M. Alvarez, and Wesley Shields. *YARA: The pattern matching swiss knife for malware researchers.* `https://plusvic.github.io/yara/`.

[Cal] Joan Calvet. *Aligot.* `https://code.google.com/archive/p/aligot/`.

[CFM12] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. “Aligot: Cryptographic Function Identification in Obfuscated Binary Programs”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pages 169–182. DOI: `10.1145/2382196.2382217`.

[Den] Frank Denis. *The Sodium crypto library (libsodium).* `https://libsodium.org/`.

[Dot] *The DOT Language.* `http://www.graphviz.org/content/dot-language`.

[Gra] *grap.* `https://bitbucket.org/cybertools/grap`.

[Gui] Ilfak Guilfanov. *FindCrypt2.* `http://www.hexblog.com/?p=28`.

[Plo] Daniel Plohmann. *IDAscope.* `https://bitbucket.org/daniel_plohmann/simplifire.idascope/`.

[QDNV] Nguyen Anh Quynh, Tan Sheng Di, Ben Nagy, and Dang Hoang Vu. *Capstone, the Ultimate Disassembly Framework.* `http://www.capstone-engine.org/`.

[Thi15] Aurélien Thierry. “Désassemblage et détection de logiciels malveillants auto-modifiants”. PhD thesis. Université de Lorraine, 2015.

[TT17a] Aurélien Thierry and Jonathan Thieuleux. *GRAP: define and match graph patterns within binaries.* `https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-GRAP.pdf`. REcon Brussels, 2017.

[TT17b] Aurélien Thierry and Jonathan Thieuleux. *Recherche de motifs de graphes dans un exécutable avec GRAP.* Sthack, 2017.

[The] The OpenBSD project. *LibreSSL.* `https://www.libressl.org/`.