`grap`: write pattern (and test) graphs

Abstract

We describe the syntax used for pattern and test graphs and how to write them. For quick reference on patterns and conditions you can use Figures 1, 2, 3, 4, 5, 6 and 7 or jump to section 2.3 for examples.

1 Graph files

In order to represent graphs, `grap` uses the DOT format[1] with specific keywords (options) and restrictions. The DOT format allows easy visualization of the created graph with **xdot**[2].

Restrictions. The format accepted by `grap` is a sub-language of DOT, thus all valid DOT files may not be parsed. Such restrictions include:

- All nodes must be defined before the edges
- Graph options applying to the whole graph are not supported
- Subgraphs are not supported
- Only directed graphs (digraphs) are supported

Comments. Comments may be added with a line beginning by a double forward slash (//).

Quotes. Double quotes shall be used to isolate values when special characters such as spaces and commas are used (for instance: **label**="xor eax, eax").

Multiple graphs in one file. `grap` can parse multiple graphs within a single file, this is especially useful for defining multiple patterns. Note that this is not accepted by the DOT language so you won't be able to visualize those patterns with **xdot**.

2 Patterns

2.1 Graph grammar

2.1.1 Root node

Due to the matching algorithm the pattern needs to be traversable. A root node, from which all other nodes can be reached, must be chosen by the user for the given pattern.

There shall only be one root node, which is either specified with the **root** option, or is the first defined node in the pattern file.

2.1.2 Node options

Figure 1 gives the accepted options for nodes. We previously described the `root` option. The **condition** field is a boolean expression depending on the instruction's properties, it is extensively described in section 2.2. We recommend you at least fill the **condition** option. The value **true** specifies that the node could match any instruction.

[1]The DOT Language (Graphviz): `http://www.graphviz.org/content/dot-language`
[2]**xdot** is a part of Graphviz that you can typically install with your packet manager

Match multiple instructions. The matching algorithm allows pattern nodes to represent multiple instructions. All the instructions must match the **condition** field and you can control the number of matched instructions. By default exactly one instruction is allowed (no repetition) and some options (**minrepeat**, **maxrepeat** and **repeat**) modify this behavior.

Only instructions within a basic block can be repeated: the first matched instruction shall have exactly one child instruction, the last matched instruction shall have exactly one father, and the intermediate instructions have exactly one father and one child.

The **lazyrepeat** option controls whether the algorithm will attempt to fit a minimum or a maximum number of instructions into the node. By default (**false**), the matching will fit the most instructions (within the **maxrepeat** limit) it can into the node. When set to **true**, the matching will stop once the condition of the node's child is fulfilled.

Number of fathers and children. You can specify the number of fathers and children of the instruction you're seeking with four options (**minfathers**, **maxfathers**, **minchildren**, **maxchildren**).

Note that if your are matching repeated instructions the constraint on the number of fathers is checked against the first instruction while the constraint on the number of children is checked against the last instruction. Thus for repeated instructions it is recommended to use these options for such constraints and avoid the similar fields of the **condition** option.

Node extraction. By default `grap` will count the number of matches but not output information on the matched instructions. The **getid** option marks the node for extraction with a given identifier that will be printed when a match is found.

Field name	Possible values	Default	Description
root	**true**	*None*	Specifies which node is the pattern's root
condition (or **cond**)	String	*None*	Condition to match against
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 1: Node options for pattern graphs

2.1.3 Edge options

Figure 2 shows the specific `grap` options that can apply to an edge. They only consist of the child number of the specified edge. For instance "A -> B [**childnumber**=2]" means that B should be the child of A numbered 2.

Note that children can only be numbered 1 or 2 and it is possible to have a child numbered 2 and no child numbered 1. Usually number 1 is used for the following instruction (their addresses follow each other) and number 2 for a distant instruction.

Depending on the instruction type you will thus find 0, 1 or 2 children:

- Sequential instructions (`mov, xor, add...`): the following instruction numbered 1
- Unconditional jumps (`jmp B`): the target instruction (B) numbered 2

- Conditional jumps (`jne B`): the following instruction numbered 1 and the target instruction (B) numbered 2
- Calls (`call B`): the following instruction numbered 1 and the target instruction (B) numbered 2
- Returns (`ret`): no child

When the option is not specified, it is inferred depending on the number of existing children.

Field name	Possible values	Description
childnumber	**1** or **2**	Sets child number

Figure 2: Edge options for pattern graphs

2.1.4 Implicit declarations

As stated previously you may want to always declare the **condition** field, but if it is missing:

- If a field **instruction** (or **inst**) with a value **v** is filled, the condition is equivalent to "instruction beginswith **v**",
- otherwise, if the field **label** is filled, the condition is the same as previously with the value given to this field,
- otherwise, the name of the node is taken for the condition.

2.2 Condition grammar

A valid condition is either

- a single property (for instance "*basicblockend*", see list on Figure 3), or
- a single test with an immediate value: *field* **operator** value (for instance "*nargs* >= 2"), or
- a single test with a self-reference: *field1* **operator** _field2 (for instance "*arg1* **is** _arg2"), or
- a boolean operation on one or multiple conditions (see Figure 4).

Property	Description
basicblockend	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 3: Supported properties within conditions

Quotes. Single quotes may be used to isolate values when special characters are used (for instance: instruction **regex** 'xor e(a|b)x ,.*').

Operator	Description
true	Always true
not ...	Negation
... **and** ...	Conjunction
... **or** ...	Disjunction
(...)	Separation: **not** (a **and** b)

Figure 4: Supported boolean operators

2.2.1 Fields and operators

Figure 5 shows the fields on which you can create your conditions, their type and whether they are available for self-reference (such as in "instruction **contains** _arg1"). Each field has a type (String or Number) and the operators available are not the same for each type (Figures 6 and 7).

Number of fathers and children. **nfathers** and **nchildren** can be used to control the number of incoming and outgoing edges but, when applied with a repetition option, all the matched instructions must fulfill the condition and you might run into problems. Thus, as stated in section 2.1.2, when one of the **repeat** option is used, the node options should be prefered to the condition fields.

Field name	Value type	Description	Self-reference
instruction (or **inst**)	String	Full disassembled instruction (text)	No
opcode	String	Mnemonics of the disassembled opcode	No
nargs	Number	Number of arguments	No
arg1	String	First argument (if any)	Yes (_arg1)
arg2	String	Second argument (if any)	Yes (_arg2)
arg3	String	Third argument (if any)	Yes (_arg3)
address (or **addr**)	Number	Address (VA) of the instruction	No
nfathers	Number	Number of incoming edges	No
nchildren	Number	Number of outgoing edges	No

Figure 5: Condition fields

The following table shows a few examples of x86 instructions and the values of each field. Note that this describes how `grap` behaves by default but it depends on how the test binary is disassembled (see section 3).

Instruction	opcode	nargs	arg1	arg2	arg3
xor eax, ebx	xor	2	eax	ebx	
mov eax, dword ptr [0x41bc0c]	mov	2	eax	dword ptr [0x41bc0c]	
ret	ret	0			

Text-based fields. The current version of `grap` treats most instruction related fields (such as opcode and arguments) as text. Thus it is only possible to discriminate on the type of instruction (for instance: is it a conditional jump ?) or on the type of the argument (register or immediate value ?) with string-based rules (for instance: "opcode **beginswith** j", "arg1 **regex** 'e(a|b|c)x'").

Regular expressions. The boost library provides the regular expression matching with the Perl syntax[3]. The **regex** operator requires that the given field matches the whole expression, so you may need to add wildcard repetition (such as: .*).

[3]`http://www.boost.org/doc/libs/release/libs/regex/doc/html/boost_regex/syntax/perl_syntax.html`

Operator	Applies to	Description
is	String	Exact string
contains	String	Substring
beginswith	String	Matching with beginning of string
regex	String	Regular expression

Figure 6: Strings operators

Operator	Applies to
==	Number
>=	Number
>	Number
<=	Number
<	Number

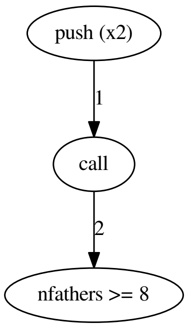
Figure 7: Numbers operators

2.3 Examples

We describe a few pattern examples with the DOT file along with their **xdot** representation. Note that colored (purple) fields are specific to `grap` while the black ones (**label**, **shape**) are only used for the graphical representation.

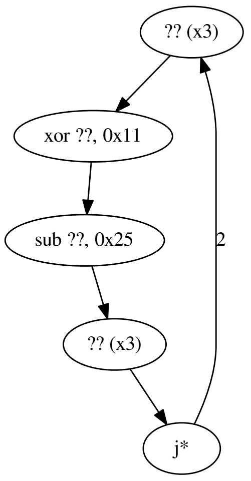
Call and incoming edges. The following example look for functions that are called at least 8 times in the binary and have at least two arguments passed with `push` instructions preceding the `call`.

```
digraph function_call{
push [label="pushes", cond="opcode is push", repeat=2, getid=push]
call [label="call", cond="opcode is call"]
function [label="nfathers >=8", cond="nfathers >=8", getid=f]

push -> call [label=1]
call -> function [label=2, childnumber=2]
}
```



Specific function. The next example detects the decryption function used in a Backspace sample (with MD5=4ee00c46da143ba70f7e6270960823be). It focuses on the algorithm consisting of a `xor 0x11` followed by a `sub 0x25` to help find very similar samples. It specifically looks for a loop consisting of any 3 instructions then the `xor`, then the `sub` then any 3 instructions and a conditional jump to loop.

```
digraph decrypt_sample_4ee00c46da143ba70f7e6270960823be {
A [cond=true, repeat=3]
B [cond="opcode is xor and arg2 is 0x11"]
C [cond="opcode is sub and arg2 is 0x25"]
D [cond=true, repeat=3]
E [cond="opcode beginswith j and nchildren == 2"]

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

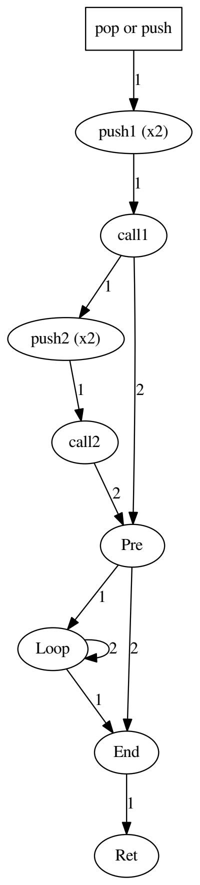


Multiple patterns. After examining many Backspace samples, we found different version of simple decryption algorithms and created small patterns to discriminate against them. This example shows three of them, including the previous `xor` then `sub`. Note that you can put multiple patterns in a single file and they will be parsed by `grap`, but you won't be able to view it with **xdot** (it is not a valid DOT file).

```
digraph decrypt_xor_sub {
A [cond="opcode is xor and arg2 is 0x11"]
B [cond="opcode is sub and arg2 is 0x25"]

A -> B
}

digraph decrypt_sub_xor_sub {
A [cond="opcode is sub"]
B [cond="opcode is xor and arg2 is 0xb"]
C [cond="opcode is sub and arg2 is 0x12"]

A -> B
B -> C
}

digraph decrypt_xor_sub_sub {
A [cond="opcode is xor and arg2 is 0x17"]
B [cond="opcode is sub"]
C [cond="opcode is add and arg2 is 0x13"]

A -> B
B -> C
}
```

Successive calls. The following example looks for a specific pattern found in Backspace samples and consisting of a `pop` or `push` instruction followed by two calls, each with two pushed arguments, to the same function. This function consists of 0 to 4 instructions (prolog), then a loop with 3 to 12 instructions, then 0 to 2 instructions followed by a `ret`.

```
digraph push_calls{
0 [label="pop or push", shape=box, cond="inst regex '^p(op|ush).*'"]
A [label="push1 (x2)", cond="opcode is push", repeat=2]
B [label="call1", cond="opcode is call"]
C [label="push2 (x2)", condition="opcode is push", repeat=2]
D [label="call2", cond="instruction beginswith call"]

Pre [label="Pre", cond=true, maxrepeat=4]
Loop [label="Loop", cond=true, minrepeat=3, maxrepeat=12, getid="loop"]
End [label="End", cond=true, minrepeat=0, maxrepeat=2, lazyrepeat=true]
Ret[label="Ret", cond="opcode beginswith ret"]

0 -> A [label=1, childnumber=1]
A -> B [label=1, childnumber=1]
B -> C [label=1, childnumber=1]
C -> D [label=1, childnumber=1]

B -> Pre [label=2, childnumber=2]
D -> Pre [label=2, childnumber=2]

Pre -> Loop [label=1, childnumber=1]
Pre -> End [label=2, childnumber=2]
Loop -> End [label=1, childnumber=1]
Loop -> Loop [label=2, childnumber=2]
End -> Ret [label=1, childnumber=1]
}
```



2.3.1 Condition examples

A few condition examples are given below.

```
cond=true
cond="opcode is xor"
cond="nfathers >=5"
cond="inst contains eax and addr >= 0x401500 and addr <= 0x401505"
cond="inst regex '.*(x)?or.*|^j?c.*'"
cond="nargs == 2"
```

Conditional jumps. The next example is useful for detecting conditional jumps.

```
cond="opcode beginswith j and nchildren == 2"
```

Zero assignment. The following example uses a self-reference (_arg2) to find assignment to zero: it is either `mov ??, 0` or `xor arg1, arg2` with arg1 = arg2.

```
cond="(opcode is mov and arg2 is 0) or (opcode is xor and arg1 is _arg2)"
```

Memory access. The next condition can be used to detect a `xor` overwriting a value in memory (such as: `xor [eax], 0x11`).

```
cond="opcode is xor and arg1 contains ["
```

Xor ending a basic block. This example looks for a `xor` condition that would be the end of a basic block.

cond="**opcode** **is** xor **and** **basicblockend**"

3 Test graphs

This section describes how to write test graphs from binaries. It may be useful if you wish to write test graphs manually or write your own disassembler.

3.1 Node options

Figure 8 shows the fields that you can fill for them to be matched against pattern graphs. Some fields will be implicitly filled by `grap` while parsing but you can explicitly define them. It is recommended to fill at least the **instruction** and the **address** fields since they cannot be guessed.

Field name	Value type	Default	Implicit example
instruction (or **inst**)	String	`label` value, or name	
opcode	String	*implicit*	mov
nargs	Number	*implicit*	2
arg1	String	*implicit*	eax
arg2	String	*implicit*	0x3
arg3	String	*implicit*	
address (or **addr**)	Number	0	

Figure 8: Test fields, implicit values assume **instruction**="mov eax, 0x3"

3.2 Edge options

As with pattern graphs, test graphs only have one edge option (Figure 9) which specifies the number of the child. It will be implicitly inferred based on the number of children already declared for the given node.

Field name	Possible values	Default
childnumber	**1** or **2**	*implicit*

Figure 9: Edge options for test graphs

3.3 Example

Below is a small example of a test graph with the **instruction** and **address** fields always filled. The other fields may be filled but otherwise they will be implicitly inferred.

```
digraph G {
401000 [inst="mov eax, 0x407f94", address="0x401000"]
401005 [inst="call 0x407D90", address="0x401005"]
407D90 [inst="push ebp", nargs=1, arg1=ebp, address="0x401005"]
40100A [inst="mov ecx, 0", opcode=mov, nargs=2, arg1=ecx, arg2=0, address="0x40100A"]
40100B [inst="push ebx", address="0x40100B"]
40100C [inst="push esi", address="0x40100C"]
40100D [inst="push edi", address="0x40100D"]
401000 -> "401005" [childnumber=1]
401005 -> "40100A" [childnumber=1]
401005 -> "407D90" [childnumber=2]
40100A -> "40100B" [childnumber=1]
40100B -> "40100C" [childnumber=1]
40100C -> "40100D" [childnumber=1]
}
```