

AsmRegex Documentation

Jordy Gennissen

Royal Holloway, University of London
jordy.gennissen@rhul.ac.uk

1 Background

This is on the why and not the how, so feel free to skip this if you're convinced.

A lot can be concluded by looking at code, whether it's source code or a compiled, binary representation. However, when compiling, higher level semantics get lost in the translation from source code towards binary code (or rather assembly code). Moreover, binary code changes drastically when using a different cpu-family or operating system, a different compiler or even a different version of the same compiler. In spite of this, the resulting executable should produce the same result (and usually does).

Hence, automatically analysing this assembly code is a very complex and error-prone task, because one approach might not work on the next version of the same compiler. Yet, the source code might not always be available when analysing an executable. On top of this, source code is easy to understand but is not the code that is executed in the end. In other words, it is very useful to be able to reason about the executed code and its assembly representation.

To do this, we chose to use a computer science approach that is well known: regular expressions. Regular expressions are a powerful tool and widely used to match all kinds of patterns: email address format, file format checks, matching an IP address or grabbing html tags to name a few. Besides, regular expressions are well-studied and so are their limitations. Matching assembly code patterns using regular expressions almost becomes natural.

In conclusion, `AsmRegex` is an approach to create a more robust detection of patterns in assembly code, by using the power of regular expressions combined with a solid and versatile framework that can be extended to compare any requirements one may need on assembly code. However, the real strength for this tool is to write `AsmRegexes` yourself, as the entire tool relies on the strength of the expression.

2 Assembly RegEx

As mentioned, the regular expressions to write resemble regular expressions, with the main difference to use “assembly instruction placeholders” instead of a normal RegEx. These placeholders are described in more detail later in this document. A typical assembly instruction placeholder looks as follows:

```
<mov,>
```

From here onwards, matching is done like your regular regular expression. For example, to match a potential if-statement like this

```
if (var == 10) {  
    i++;  
}
```

A good candidate expression would be

```
<mov,>+<cmp,><jne,><inc,>
```

It might just do the trick. However this pattern is not very versatile yet and can be heavily improved. Take a look at the next attempt:

```
<mov.lea,>*<mov,DR,0xa,><any,>,5<cmp,DR,DR,><jne,PV,><inc.add.,>
```

Due to limitations of regular expressions, matching the exact destination register of the MOV instruction to one of the arguments of CMP is (theoretically) impossible, but having a maximum of 5 instructions in between makes it likely to be the if-statement.

2.1 Design choices

The AsmRegex system differs from the “regular regular expressions” in a couple of ways.

2.1.1 Non Greedy Matching

By default, it will match every (sub)pattern in a lazy way (as opposed to greedy). This will mean that `<a>*` will try to match `` first, will try to match `<a>` next, and so on (to be changed by setting the static variable `RepetitionTracker.laziness` to `False`). In case a greedy match is preferred, one can add a capital G in front of the repetition statement of the pattern string. Thus, `<a>G*` will try to match as many `<a>`’s as possible before matching a ``. Similarly, when the default is reset to be greedy, a (sub)pattern can be set to lazy by adding the capital L.

The greediness of (sub)patterns is particularly important in the design when using the current `AssemblyMatcher.match()`, because of the following design.

2.1.2 Match Starts

The `AsmRegex` system will try to match from the first assembly instruction onwards. When an instruction while matching turns out to be a `JMP`, the next instruction to match will be the instruction where the unconditional jump points to.

In the usual case that nothing matches from the given pointer, it will change its start for matching towards the next assembly instruction and so on. This does not jump on a `JMP` instruction. However, when a match is found:

- The matcher will stop its current search for this start pointer, deleting all unexplored states for matching, and,
- the matcher will continue at the assembly instruction after the last-matched instruction

These will filter out many duplicates and partial duplicates when matching complex expressions. However, if the match includes any backwards “jump” instruction, *this can generate an infinite loop*.

2.1.3 Non Greedy Parsing

As the name suggest, parsing in general is supposed to be non greedy or lazy within the `AsmRegex`. This currently means that repetition matching is done as in most `Regex` systems (“`ab*`” is equivalent to “`a(b*)`” and not “`(ab)*`”). However, this differs from most `Regex` OR statements, where “`ab|c`” in `AsmRegex` is equivalent to “`a(b|c)`” and not to “`(ab)|c`”. This is in contrast to most general `Regex` systems.

The next section will go into more depth on the assembly placeholder syntax and power.

3 Assembly placeholder syntax

The assembly tag or placeholder defines a pattern to match to one single assembly instruction. It always starts with a “less than” sign and always ends with a “greater than” sign, analogous to html tags. Inside the tag, various conditions reside, delimited with a comma.

A full tag is defined as follows:

```
<opcode,arg1,arg2,options>
```

`ARG1` and `ARG2` are optional (and may even be non-existent in the assembly instruction). `OPCODE` and `OPTIONS` are however mandatory. Hence, every tag must contain at least one comma. As of yet, no additional options are implemented yet. This means that every tag **must** end with “`>`”.

Table 1: Prewritten opcode ranges

OPCODE	Resulting range
any	Any instruction
JC	Any conditional jump (regex('j+') but not jmp)
JS	Jumps on single conditions (je, jne, jz)
ALU	Any ALU calculation instruction (add, sub, inc, dec, and, (x)or, not, (i)mul, (i)div)
SS	Any ALU shift (shl, sal, hsr, sar)
RR	Any ALU rotate (rol, rcl, ror, rcr)
FL	Any unconditional flag set (stc, cls, cmc, std, cld, sti, cli)
PU	Any push instruction (push, pusha, pushf)
PO	Any pop instruction (pop, popa, popf)
PP	Any push or pop instruction (PU PO)

3.1 Opcodes

The OPCODE placeholder should currently be a list of opcodes, delimited with a dot. Some additional and non-existent opcodes have been added in capitals, defining a standard range of opcodes. These special ones are defined in table 1.

Apart from these additional ranges of instructions, only concrete and exact opcodes can be matched. If you feel anything is missing in this list, feel free to add it or to let me know.

3.2 Arguments

ARG1 and ARG2 are analogous apart from the position in the assembly instruction. If no comma is set to denote the argument condition or the argument condition is empty, it will match any argument. In other words, this:

```
<mov,><mov,,>
```

will match on any two consecutive moves. If you want to specify a condition on argument 2 but not on argument one, you can leave argument 1 empty:

```
<mov,,CC,>
```

Note that the last comma is always necessary because specifying an (even empty) additional condition is mandatory.

The argument condition is by default a **Regular expression** on its own. However, like in the opcodes, certain standard patterns have been precompiled to match against the usual arguments of instructions. These can be found in table 2.

Table 2: Prewritten Argument patterns

ARGUMENT	meaning
RR	Referenced Register (i.e. [rax - 0xa])
DR	Direct Register (i.e. ebx)
CC	Concrete Constant (i.e. 1 or 0x1234)
PV	(likely) Pointer Value: 0x400000 - 0x40fff
RC	(likely) Random Constant: $0x9 < x < 0xffffffe$, but not a pointer (\neg PV)

3.3 Options

Additional options are currently not implemented yet, so there's not much to document at this stage.

3.4 Inverting

Both opcodes and arguments have an “invert” option, meaning they will match on anything except the expression given. This is done by prepending a capital *I* to the assembly pattern. The following example will match any jump instruction to a location that is not a constant (i.e. register or referenced register):

```
<JC.jump,ICC,>
```

Identically, this can be used for opcodes. Adding an *I* at the beginning of the opcode specification in the example above, will match on any non-jump instruction. Note that this will automatically match opcodes too that do not have any argument at that position¹.

4 Regular expression operations

As this is a relatively simple regular expression matcher, not all options are implemented with respect to regular expressions. For example, any lookahead matching or non-matching expressions are not implemented.

Currently, only repetitions and OR statements are implemented. Repetitions can be any range $\{a,b\}$ where *a* or *b* can be left out to specify there is no minimum or maximum respectively. The usual regular expression notation is used (i.e. *?*, ***, *+*). As mentioned before, this matcher is lazy by default.

The OR is equally straightforward, where it will match the instruction or subpattern before the OR delimiter (“|”) first, skipping the second half. On a failed match, it will try to match the instruction or subpattern after the delimiter instead.

¹This was an arbitrary design decision without a particular reason and may be changed in the future if this simplifies our goals.

```

[example_pattern]
(
  <mov.lea,>*
  <mov,DR,0xa,> # move value 10 into a register
  <any,>{,5}
  <cmp,DR,DR,>
  <jne.je,PV,>?
)G+
<inc.add.,>

[example2]
(<SS.RR.ALU.mov.lea,>G+
<PP,>{1,2} # push or pop
){2,}

```

Figure 1: Example file contents

5 Loading a file

AsmRegex has the capability of loading a single file containing one or more expressions. Every pattern has to have a unique name, given in straight brackets ('[]') at the top of the pattern. Inline comments are done with a sharp ('#'), also commonly known as a hashtag. An example pattern is shown in Figure 1.