# yaramod

***Release v3.20.1***

**Avast**

**Jul 31, 2023**

# CONTENTS

Welcome to the documentation of our project Yaramod. Parser and builder for YARA rulesets.

CHAPTER

ONE

# INSTALLATION

## 1.1 Requirements

In order to install `yaramod` you will need:

- CMake 3.6+
- C++ compiler with C++17 support

If you want to use Python bindings then you will also need:

- Python 3.6+

## 1.2 Compilation

Yaramod is primarily written in C++ so you'll have to compile it to the library which you can then link to your own project. We do not provide any prebuild binary releases so if you would like to use it from C++ code, you need to compile it yourself. We use CMake for our build system so you'll need to install it beforehand. We support 3 main platforms - Linux, Windows and macOS. The project might work also on other platforms but we do not provide any support for those nor any guarantees that it works.

In order to configure the project and start the compilation run

```
mkdir build
cmake -DCMAKE_BUILD_TYPE=<Release|Debug> ..
cmake --build .
```

If you would also like to build examples (located in src/examples/cpp) then also pass `-DYARAMOD_EXAMPLES=ON` to the second command, right after `-DCMAKE_BUILD_TYPE=...`.

It is also possible to build Python bindings directly from CMake by providing an option `-DYARAMOD_PYTHON=ON`.

## 1.3 Python Bindings

Yaramod also comes with Python 3 bindings in case you don't want to deal with linking C++ libraries or you just simply prefer Python over C++. We deploy yaramod to PyPI so you can install it directly from there using `pip`.

```
pip install yaramod
```

In case you don't want to use the latest release but would much rather use `master` branch you can run

```
pip install .
```

from the root directory of yaramod directly.

CHAPTER

TWO

# GETTING STARTED

Before we start, we would like to introduce some basic concepts we use in yaramod so you can get familiar with them and know how to use them.

Yaramod as a library serves two main purposes:

- Parsing of YARA rulesets into intermediate form which is friendly for inspection, analysis or even transformations
- Creation of new YARA rulesets through code using declarative description of how your YARA rule should look like

These two together form a strong tooling for you to build more advanced systems, perform different kind of analyses over your rulesets, etc.

We have our own custom parser for YARA, so we don't use the one in VirusTotal/yara, but we try to stay consistent as much as we can. Generally, you shouldn't run into situation where YARA does parse your rule and yaramod does not (or vice versa) but this should be considered as bug and should be reported to us using issue tracker on GitHub.

It is also important to state that yaramod in no way does any kind of matching because it does not understand your ruleset in a sense what you want to match. Yaramod just knows what is inside of your rules and gives you an option to process it. Despite that, yaramod performs semantic checks where possible so you for example need to provide correct types of parameters to module function calls or you have to import a module in order to use it etc.

## 2.1 Modules

YARA is easily extensible with modules which provide you a way to call C functions from your YARA ruleset. We realize that modules are important and yaramod therefore supports all modules in upstream YARA.

We at Avast use YARA daily and sometimes there are things we would like to match in our YARA rules something that is not accessible to the outside world so we improve existing or develop our own modules and then supply yaramod with these modules as jsons. We will now describe how to use some custom modules.

First thing that needs to be done is to write each of the custom modules in a `.json` file in a similar way the upstream modules are written in (see cuckoo module). Then you have two options on how to supply the modules to yaramod depending on your preference:

- Supply the directory, where the custom modules are stored, to `Yaramod` constructor with parameter *modules directory* as described in the `Yaramod instance` section. This means yaramod will load modules from specified directory instead of the directory, where the default upstream modules are specified.
- Supply the paths to the modules through environmental variable `YARAMOD_MODULE_SPEC_PATH` or `YARAMOD_MODULE_SPEC_PATH_EXCLUSIVE`: Setting `YARAMOD_MODULE_SPEC_PATH="<path to first module>:<path to second module>:<path to third module>"` means that you want to use the default YARA modules together with three aditional module specifications. Any number of paths separated by colon

can be specified here. When the `YARAMOD_MODULE_SPEC_PATH_EXCLUSIVE` environmental variable is set instead, yaramod will only consider the custom modules and no other modules will be available. Note, that when both variables are set yaramod throws an error.

- You can combine the two options and load modules from specified directory and additionaly some modules specified via `YARAMOD_MODULE_SPEC_PATH`.

When yaramod gets multiple different jsons describing the same module, it merges both specifications and builds module with functionality of boths specifications. We can merge structures, functions, add overloads of functions and more. An exception is thrown when there is type inconsistence or other conflict between the two specifications.

## 2.2 VirusTotal symbols

YARA allows you to define so called *external variables* which are values outside your YARA environment but they are constant across the whole YARA scan. You can reference them in your rule conditions to give an additional data to your rule so it can provide you with the matches you want.

These *external variables* are widely used on VirusTotal as you can see here. We realize this and we also realize that you might want to process rulesets that you are using in VirusTotal so we also provide these additional symbols but you can opt-out of this feature if you want.

## 2.3 Yaramod instance

The entrypoint of all yaramod is class `Yaramod` which you should instantiate and keep it alive while you are doing anything with yaramod. Accessing internal representation of YARA rules which were returned by `Yaramod` is completely unsafe and can lead to crashes of your application if you do it after `Yaramod` has been destructed. Creation of `Yaramod` object is performance heavy so you should keep the amount of instances low (ideally just one). `Yaramod` itself is not thread-safe, so in parallel environment we would suggest you to create one instance per thread or process.

`Yaramod` accepts two optional parameters when creating it and that is *import features* and *modules directory*. The first option specify in what kind of rulesets you are interested in and you can choose from:

- *All current* - This is the default option which provides you with both Avast-specific and VirusTotal-specific symbols.
- *Everything* - This also includes deprecated functions which should no longer be used.
- *Basic* - This represents that you are not interested in any additional symbols in your rules.
- *Avast* - You are interested in basic and Avast-specific symbols.
- *VirusTotal* - You are interested in basic and VirusTotal-specific symbols.

The second option enables you to supply custom modules other than YARA upstream modules. Simply enter the path to the directory where your modules are stored as *json* files.

CHAPTER

THREE

# PARSING RULESETS

Parsing a ruleset from a file is as easy as this.

Python

```python
import yaramod

y = yaramod.Yaramod(yaramod.Features.AllCurrent)
yara_file = y.parse_file('/opt/ruleset.yar')
print(yara_file.text)
```

C++

```cpp
#include <iostream>
#include <yaramod/yaramod.h>

int main() {
    auto y = yaramod::Yaramod(yaramod::Features::AllCurrent);
    auto yaraFile = y.parseFile("/opt/ruleset.yar");
    std::cout << yaraFile->getText() << std::endl;
    return 0;
}
```

You can alternatively also parse from memory.

Python

```python
import yaramod

y = yaramod.Yaramod(yaramod.Features.AllCurrent)
yara_file = y.parse_string(r'''
rule abc {
    condition:
        true
}
''')
print(yara_file.text)
```

C++

```cpp
#include <iostream>
#include <sstream>
#include <yaramod/yaramod.h>
```

(continues on next page)

(continued from previous page)

```
int main() {
    auto y = yaramod::Yaramod(yaramod::Features::AllCurrent);
    std::istringstream input(R"(
    rule abc {
        condition:
            true
    }
    )");
    auto yaraFile = y.parseStream(input);
    std::cout << yaraFile->getText() << std::endl;
    return 0;
}
```

## 3.1 Rules

You can iterate over all rules in the file.

Python

```
for rule in yara_file.rules:
    print(rule.name)
    print(f'  Global: {rule.is_global}')
    print(f'  Private: {rule.is_private}')
```

C++

```
for (const auto& rule: yaraFile->getRules()) {
    std::cout << rule->getName() << '\n'
        << "  Global: " << rule->isGlobal() << '\n'
        << "  Private: " << rule->isPrivate() << std::endl;
}
```

## 3.2 Metas

You can also access meta information of each rule

Python

```
for rule in yara_file.rules:
    for meta in rule.metas:
        if meta.value.is_string:
            print('String meta: ', end='')
        elif meta.value.is_int:
            print('Int meta: ', end='')
        elif meta.value.is_bool:
            print('Bool meta: ', end='')
        print(f'{meta.key} = {meta.value.pure_text}')
```

C++

```
for (const auto& rule : yaraFile->getRules()) {
    for (const auto& meta : rule->getMetas()) {
        if (meta->getValue()->isString())
            std::cout << "String meta: ";
        else if (meta->getValue()->isInt())
            std::cout << "Int meta: ";
        else if (meta->getValue()->isBool())
            std::cout << "Bool meta: ";
        std::cout << meta->getName() << " = " << meta->getValue()->getPureText() <<␣
↪std::endl;
    }
}
```

## 3.3 Variables

You can iterate over local variables available for each rule, see their identifier, type and value.

Python

```
for rule in yara_file.rules:
    for variable in rule.variables:
        if variable.value is str:
            print('Plain string: ', end='')
        elif variable.value is int:
            print('Integer: ', end='')
        elif variable.value is float:
            print('Double: ', end='')
        elif variable.value is bool:
            print('Boolean: ', end='')
        print(f'{variable.key} = {variable.value.text}')
```

C++

```
for (const auto& rule : yaraFile->getRules()) {
    for (const auto& variable : rule->getVariables()) {
        if (variable.getValue()->isString())
            std::cout << "String: ";
        else if (variable.getValue()->isInt())
            std::cout << "Integer: ";
        else if (variable.getValue()->isFloat())
            std::cout << "Double: ";
        else if (variable.getValue()->isBool())
            std::cout << "Boolean: ";
        std::cout << variable.getKey() << " = " << variable.getValue()->getText() << '\n
↪';
    }
}
```

## 3.4 Strings

Iterating over available strings is also possible and you can distinguish which kind of string you are dealing with.

Python

```
for rule in yara_file.rules:
    for string in rule.strings:
        if string.is_plain:
            print('Plain string: ', end='')
        elif string.is_hex:
            print('Hex string: ', end='')
        elif string.is_regexp:
            print('Regexp: ', end='')
        print(f'{string.identifier} = {string.text}')
        print(f'  ascii: {string.is_ascii}')
        print(f'  wide: {string.is_wide}')
        print(f'  nocase: {string.is_nocase}')
        print(f'  fullword: {string.is_fullword}')
        print(f'  private: {string.is_private}')
        print(f'  xor: {string.is_xor}')
        print(f'  base64: {string.is_base64}')
        print(f'  base64wide: {string.is_base64_wide}')
```

C++

```
for (const auto& rule : yaraFile->getRules()) {
    for (const auto& string : rule->getStrings()) {
        if (string->isPlain())
            std::cout << "Plain string: ";
        else if (string->isHex())
            std::cout << "Hex string: ";
        else if (string->isRegexp())
            std::cout << "Regexp: ";
        std::cout << string->getIdentifier() << " = " << string->getText() << '\n'
            << "  ascii: " << string->isAscii() << '\n'
            << "  wide: " << string->isWide() << '\n'
            << "  nocase: " << string->isNocase() << '\n'
            << "  fullword: " << string->isFullword() << '\n'
            << "  private: " << string->isPrivate() << '\n'
            << "  xor: " << string->isXor() << '\n'
            << "  base64: " << string->isBase64() << '\n'
            << "  base64wide: " << string->isBase64Wide() << std::endl;
    }
}
```

## 3.5 Condition

There are 2 ways you can look at the condition. The first one is that you just care about the textual representation of the condition and you don't care about the contents. That one is pretty straightforward.

Python

```python
for rule in yara_file.rules:
    print(rule.condition.text)
```

C++

```cpp
for (const auto& rule : yaraFile->getRules())
    std::cout << rule->getCondition()->getText() << std::endl;
```

The second way is that you care about the contents of the condition and you would like to perform some kind of analysis over the condition. This part is a bit tricky because the hierarchy of the whole condition is unknown to you so you would have to write a lot of recursive algorithms or other kinds of traversals on abstract syntax tree of your condition. To ease this all, we have adopted similar approach as LLVM and provide you with an option to use visitor design pattern to perform the traversal.

---

**Note:** If you are not faimilar with this kind of design pattern, just imagine that there are several types of expressions and statements that can be in the condition (integers, logical operations, arithmetic operations, ...). You want to perform *your operation* on all of them, taking their type into account. With visitor design pattern, you just define *your operation* for each type of expression or statement and that's it. You then *visit* each node of abstract syntax tree with *your operation* which is performed there.

---

### 3.5.1 Condition visitors

Let's say we want to print each function that is in called in the rule condition.

Python

```python
class FunctionCallDumper(yaramod.ObservingVisitor):
    def visit_FunctionCallExpression(self, expr):
        print('Function call: {}'.format(expr.function.text))
        # Visit arguments because they can contain nested function calls
        for arg in expr.arguments:
            arg.accept(self)
```

C++

```cpp
class FunctionCallDumper : public yaramod::ObservingVisitor {
public:
    void visit(FunctionCallExpression* expr) override {
        std::cout << "Function call: " << expr->getFunction()->getText() << '\n';
        // Visit arguments because they can contain nested function calls
        for (auto& param : expr->getArguments())
            param->accept(this);
    }
};
```

**Note:** As you can see, visitors depend heavily on recursion and that can represent problems sometimes with a huge rulesets where depth of AST is rather large. Python has a limit on how many stack frames you can have at the sime time in order to prevent stack overflow. This limit can be however sometimes very limiting and set too low for certain huge conditions. You might need to run sys.setrecursionlimit to process those.

### 3.5.2 Expression types

Each expression type has its own unique id (uid). These uids are unique only within scope of a single rule, this allows to identify specific node in the AST for extra processing. There are a lot of expression types that you can visit. Here is a list of them all:

**String expressions**

- `StringExpression` - reference to string in `strings` section (`$a01`, `$sa02`, `$str`, ...)
- `StringWildcardExpression` - reference to multiple strings using wildcard (`$a*`, `$*`, ...)
- `StringAtExpression` - refers to `$str at <offset>`
- `StringInRangeExpression` - refers to `$str in (<offset1> .. <offset2>)`
- `StringCountExpression` - reference to number of matched string of certain string identifier (`#a01`, `#str`)
- `StringOffsetExpression` - reference to first match offset (or Nth match offset) of string identifier (`@a01`, `@a01[N]`)
- `StringLengthExpression` - reference to length of first match (or Nth match) of string identifier (`!a01`, `!a01[N]`)

**Unary operations**

All of these provide method `getOperand()` (`operand` in Python) to return operand of an expression.

- `NotExpression` - refers to logical `not` operator (`not @str > 10`)
- `UnaryMinusExpression` - refers to unary `-` operator (`-20`)
- `PercentualExpression` - refers to unary `%` operator (`20%`)
- `BitwiseNotExpression` - refers to bitwise not (`~uint8(0x0)`)

**Binary operations**

All of these provide methods `getLeftOperand()` and `getRightOperand()` (`left_operand` and `right_operand` in Python) to return both operands of an expression.

- `AndExpression` - refers to logical `and` (`$str1 and $str2`)
- `OrExpression` - refers to logical `or` (`$str1 or $str2`)
- `LtExpression` - refers to < operator (`$str1 < $str2`)
- `GtExpression` - refers to > operator (`$str1 > $str2`)
- `LeExpression` - refers to <= operator (`@str1 <= $str2`)
- `GeExpression` - refers to >= operator (`@str1 >= @str2`)
- `EqExpression` - refers to == operator (`!str1 == !str2`)
- `NeqExpression` - refers to != operator (`!str1 != !str2`)
- `ContainsExpression` - refers to `contains` operator (`pe.sections[0].name contains "text"`)

- `MatchesExpression` - refers to `matches` operator (`pe.sections[0].name matches /(text|data)/`)
- `IequalsExpression` - refers to `iequals` operator (`pe.sections[0].name iequals "text"`)
- `IcontainsExpression` - refers to `icontains` operator (`pe.sections[0].name icontains "text"`)
- `EndsWithExpression` - refers to `endswith` operator (`pe.sections[0].name endswith "text"`)
- `IendsWithExpression` - refers to `iendswith` operator (`pe.sections[0].name iendswith "text"`)
- `StartsWithExpression` - refers to `startswith` operator (`pe.sections[0].name startswith "text"`)
- `IstartsWithExpression` - refers to `istartswith` operator (`pe.sections[0].name istartswith "text"`)
- `PlusExpression` - refers to + operator (`@str1 + 0x100`)
- `MinusExpression` - refers to - operator (`@str1 - 0x100`)
- `MultiplyExpression` - refers to * operator (`@str1 * 0x100`)
- `DivideExpression` - refers to \ operator (`@str1 \ 0x100`)
- `ModuloExpression` - refers to % operator (`@str1 % 0x100`)
- `BitwiseXorExpression` - refers to ^ operator (`uint8(0x10) ^ uint8(0x20)`)
- `BitwiseAndExpression` - refers to & operator (`pe.characteristics & pe.DLL`)
- `BitwiseOrExpression` - refers to | operator (`pe.characteristics | pe.DLL`)
- `ShiftLeftExpression` - refers to << operator (`uint8(0x10) << 2`)
- `ShiftRightExpression` - refers to >> operator (`uint8(0x10) >> 2`)

**For expressions**

All of these provide method `getVariable()` (`variable` in Python) to return variable used for iterating over the set of values (can also be `any`, `all` or `none`), `getIterable()` (`iterable` in Python) to return an iterated set (can also be `them`) and `getBody()` (`body` in Python) to return the body of a for expression. For `OfExpression`, the `getBody()` method always returns `nullptr` (`None` in Python).

- `ForDictExpression` - refers to `for` which operates on dictionary (`for all k, v in some_dict : ( ... )`)
- `ForArrayExpression` - refers to `for` which operates on array or set of integers (`for all section in pe.sections : ( ... )`)
- `ForStringExpression` - refers to `for` which operates on set of string identifiers (`for all of ($str1, $str2) : ( ... )`)
- `OfExpression` - refers to `of` (`all of ($str1, $str2)` or `all of ($str1, $str2) in (filesize-500..filesize)` or `any of ($str1, $str2) at 0`)

**Identificator expressions**

All of these provide method `getSymbol()` (`symbol` in Python) to return symbol of an associated identifier.

- `IdExpression` - refers to identifier (`rule1`, `pe`)
- `StructAccessExpression` - refers to `.` operator for accessing structure memebers (`pe.number_of_sections`)
- `ArrayAccessExpression` - refers to `[]` operator for accessing items in arrays (`pe.sections[0]`)
- `FunctionCallExpression` - refers to function call (`pe.exports("ExitProcess")`)

**Literal expressions**

- `BoolLiteralExpression` - refers to `true` or `false`
- `StringLiteralExpression` - refers to any sequence of characters enclosed in double-quotes (`"text"`)
- `IntLiteralExpression` - refers to any integer value be it decimal, hexadecimal or with multipliers (`KB`, `MB`) (`42`, `-42`, `0x100`, `100MB`)
- `DoubleLiteralExpression` - refers to any floating point value (`72.0`, `-72.0`)

**Keyword expressions**

- `FilesizeExpression` - refers to keyword `filesize`
- `EntrypointExpression` - refers to keyword `entrypoint`
- `AllExpression` - refers to keyword `all`
- `AnyExpression` - refers to keyword `any`
- `NoneExpression` - refers to keyword `none`
- `ThemExpression` - refers to keyword `them`

**Other expressions**

- `SetExpression` - refers to set of either integers or string identifiers (`(1,2,3,4,5)`, `($str*,$1,$2)`)
- `RangeExpression` - refers to range of integers (`(0x100 .. 0x200)`)
- `ParenthesesExpression` - refers to expression enclosed in parentheses (`((5 + 6) * 30)`)
- `IntFunctionExpression` - refers to special built-in functions `(u)int(8|16|32)` (`uint16(<offset>)`)
- `RegexpExpression` - refers to regular expression (`/<regexp>/<mods>`)

## 3.6 Includes

YARA language supports inclusion of other files on the filesystem. Path provided in include directive is always relative to the YARA file being parsed. Since yaramod can also parse from memory, relative paths are only allowed when parsing from the actual file.

Whenever yaramod runs into include, it takes the content of included file and starts parsing it as if it was in place of an include. Therefore, included content is merged with all other content in the file. You can distinguish where the rule comes from using a location attribute of the rule.

Python

```
for rule in yara_file.rules:
    print(f'{rule.name}: {rule.location.file_path}:{rule.location.line_number}')
```

C++

```
for (const auto& rule : yaraFile->getRules())
    std::cout << rule->getName() << ": "
        << rule->getLocation().filePath << ':'
        << rule->getLocation().lineNumber << std::endl;
```

Yaramod can also provide you with something what YARA doesn't handle well - including the same file multiple times. If you do this in YARA then you will get error that you have duplicate rules in your ruleset. This is however not something you would like to run into when doing static analyses. You can allow duplicate includes by using

Python

```
ymod = yaramod.Yaramod()
ymod.parse_file('/path/to/file', yaramod.ParserMode.IncludeGuarded)
```

C++

```
auto ymod = yaramod::Yaramod();
ymod.parse_file("/path/to/file", yaramod::ParserMode::IncludeGuarded);
```

## 3.7 Imports

Checking what modules are imported. Keep in mind that imports are merged from all included files.

Python

```
for module in yara_file.imports:
    print(f'{module.name}')
```

C++

```
for (const auto& module : yaraFile->getImports())
    std::cout << module->getName() << std::endl;
```

## 3.8 Tokens

Yaramod provides an interface to access information about the underlying tokens related to a particular object. This information can be used to determine object location within the parsed file.

```
for rule in yara_file.rules:
    for string in rule.strings:
        start = string.token_first.location.begin
        end = string.token_last.location.end
        print(f'[{start.line}, {start.column}] - [{end.line}, {end.column}]')
```

Token exposes the *Location* which consists of two *Positions*: *begin* and *end*. *Position* represents a position of charater within the parsed file given by *line* and *column*. Currently supported token getters are:

Table 1: Supported token getters

| Object | Accessor |
|---|---|
| Rule | *token_first*, *token_last* |
| Meta | *token_key*, *token_value* |
| String | *token_first*, *token_last*, *token_id*, *token_assign* |

CHAPTER

# FOUR

# CREATING RULESET

Creating of new YARA rulesets is available in yaramod through interface based on builder design pattern.

---

**Note:** Following list of functions is made for C++. Python alternatives follow the same naming but instead of `CamelCase` they use `snake_case`. Sometimes there is keyword needed to be used as a function name (like `xor`). In that case, it is followed by underscore character (so it would be `xor_`).

---

The main goal of the yaramod interface is to make the creation of new YARA rules declarative. We wanted to avoid just concatenating strings together and we also wanted to end up with the same internal representation of YARA rules as if we parsed them. We also wanted to make it composable so that you can easily build for example part of your condition in one function, the other part in the second function and join them together.

## 4.1 File

It all starts with `YaraFileBuilder` which is a main building block for creating new YARA ruleset. here is a short example on how to create new YARA file with import of `pe` module and *some* rule (will be explained later).

Python

```
new_file = yaramod.YaraFileBuilder() \
yara_file = new_file \
    .with_module('pe') \
    .with_rule(rule) \
    .get()
```

C++

```
yaramod::YaraFileBuilder newFile;
auto yaraFile = newFile
    .withModule("pe")
    .withRule(std::move(rule))
    .get();
```

## 4.2 Rules

In order to create a rule that we saw while creating new YARA file, we will need `YaraRuleBuilder`.

Python

```
rule = yaramod.YaraRuleBuilder() \
    .with_tag('tag1') \
    .with_tag('tag2') \
    .with_string_meta('author', 'foo') \
    .with_plain_string('$str', 'Hello World').ascii().wide().xor(1,255) \
    .with_condition(condition) \
    .get()
```

C++

```
auto rule = yaramod::YaraRuleBuilder{}
    .withTag("tag1")
    .withTag("tag2")
    .withStringMeta("author", "foo")
    .withPlainString("$str", "Hello World").ascii().wide().xor_(1,255)
    .withCondition(std::move(condition))
    .get();
```

## 4.3 Hex strings

Creation of plain strings is really simple as shown in the code above. When it comes to hex strings, things get complicated. Hex strings in YARA have several features which make them more like *weak regular expressions*. Those features are:

- Wildcarded nibbles
- Jumps
- Alternations

In order to fully unlock the whole potential of hex strings, they need to be created through `YaraHexStringBuilder`.

Python

```
# Simple hex string - { 10 20 30 }
yaramod.YaraHexStringBuilder([0x10, 0x20, 0x30]).get()
# Hex string with wildcard - { 10 ?? 30 }
yaramod.YaraHexStringBuilder() \
    .add(yaramod.YaraHexStringBuilder(0x10)) \
    .add(yaramod.wildcard()) \
    .add(yaramod.YaraHexStringBuilder(0x30)) \
    .get()
# Hex string with all features - { 10 ?? 30 [4] ( 40 | 50 ) }
yaramod.YaraHexStringBuilder() \
    .add(yaramod.YaraHexStringBuilder(0x10)) \
    .add(yaramod.wildcard()) \
    .add(yaramod.YaraHexStringBuilder(0x30)) \
    .add(yaramod.jump_fixed(4)) \
```

(continues on next page)

(continued from previous page)

```
    .add(yaramod.alt([
        yaramod.YaraHexStringBuilder(0x40),
        yaramod.YaraHexStringBuilder(0x50)
    ])) \
    .get()
```

C++

```
// Simple hex string - { 10 20 30 }
yaramod::YaraHexStringBuilder{std::vector<std::uint8_t>{0x10, 0x20, 0x30}}.get()
// Hex string with wildcard - { 10 ?? 30 }
yaramod::YaraHexStringBuilder{}
    .add(yaramod::YaraHexStringBuilder(0x10))
    .add(yaramod::wildcard())
    .add(yaramod::YaraHexStringBuilder(0x30))
    .get();
// Hex string with all features - { 10 ?? 30 [4] ( 40 | 50 ) }
yaramod::YaraHexStringBuilder{}
    .add(yaramod::YaraHexStringBuilder(0x10))
    .add(yaramod::wildcard())
    .add(yaramod::YaraHexStringBuilder(0x30))
    .add(yaramod::jumpFixed(4))
    .add(yaramod::alt(
        yaramod.YaraHexStringBuilder(0x40),
        yaramod.YaraHexStringBuilder(0x50)
    ))
    .get();
```

## 4.4 Regular expressions

Regular expressions are the same story as hex strings, they just have more expressive power so they also require much more builder methods. Unfortunately right now we don't provide any kind of `RegexpBuilder` which would help you with it. We expect you to provide regular expression as a plain string. Building of regular expression is however something which we would like to add in the future.

## 4.5 Conditions

Building of conditions is heavily based on `YaraExpressionBuilder` which provides you with many functions that make it easy to express the condition. It makes use of operator overloading in both C++ and Python to make your builder code readable as much as possible.

---

**Note:** Python does not allow to override logical operators such as `and`, `or` or `not` therefore there are special function which you can used instead which are `conjunction`, `disjunction` and `not_`.

---

Python

```
# all of them
of(all(), them()).get()
# $1 and ($2 or $3)
(conjunction([
    string_ref('$1'),
    paren(disjunction([
        string_ref('$2'),
        string_ref('$3')
    ]))
])).get()
# pe.sections[0].name matches /\.(text|data)/i
(id('pe').access('sections')[int_val(0)].access('name').matches(regexp(r'(/\.(text|data)/
↪)', 'i'))).get()
# filesize <= 1MB
(filesize() <= int_val(1, IntMultiplier.Megabytes)).get()
```

C++

```
using namespace yaramod;

// all of them
of(all(), them()).get();
// $1 and ($2 or $3)
(stringRef("$1") && paren(stringRef("$2") || stringRef("$3"))).get();
// pe.sections[0].name matches /\.(text|data)/i
(id("pe").access("sections")[intVal(0)].access("name").matches(regexp(R"(/\.(text|data)/)
↪", "i"))).get();
// filesize <= 1MB
(filesize() <= intVal(1, IntMultiplier::Megabytes)).get();
```

## 4.6 List

Here is a list of everything available in builder and how it maps to YARA language. Functions listed as *basic* are basic building block for YARA expressions. You always want to start from these expressions and build upon them to form complex expressions. Each of these functions returns you an object of type *YaraExpressionBuilder*. Those functions with parameters also mostly accept object of these types, so whenever you are not sure what kind of expression to put there, just look at the list of all basic expressions and find the most suitable one.

Python

**Basic expression functions**

- `filesize()` - represents `filesize` keyword
- `entrypoint()` - represents `entrypoint` keyword
- `all()` - represents `all` keyword
- `any()` - represents `any` keyword
- `them()` - represents `them` keyword
- `int_val(val, [mult])` - represents signed integer with multiplier (default: `IntMultiplier.Empty`) (`int_val(10)`, `int_val(10, IntMultiplier.Kilobytes)`, `int_val(10, IntMultiplier.Megabytes)`)

- `uint_val(val, [mult])` - represents unsigned integer with multiplier (default: `IntMultiplier.Empty`) (`int_val(10)`, `int_val(10, IntMultiplier.Kilobytes)`, `int_val(10, IntMultiplier.Megabytes)`)
- `hex_int_val(val)` - represents hexadecimal integer (`hex_int_val(0x10)`)
- `double_val(val)` - represents double floating-point value (`double_val(3.14)`)
- `string_val(str)` - represents string literal (`string_val("Hello World!")`)
- `bool_val(bool)` - represents boolean literal (`bool_val(True)`)
- `id(id)` - represents single identifier with name `id` (`id("pe")`)
- `string_ref(ref)` - represents reference to string identifier `ref` (`string_ref("$1")`)
- `set(elements)` - represents `(item1, item2, ...)` (`set([string_ref("$1"), string_ref("$2")])`)
- `range(low, high)` - represents `(low .. high)` (`range(int_val(100), int_val(200))`)
- `match_count(ref)` - represents match count of string identifier `ref` (`match_count("$1")`)
- `match_length(ref, [n])` - represent `n``th match (default: 0) length of string identifier ``ref` (`match_length("$1", int_val(1))`)
- `match_offset(ref, [n])` - represents `n``th match (default: 0) offset of string identifier ``ref` (`match_offset("$1", int_val(1))`)
- `match_at(ref, expr)` - represents `<ref> at <expr>` (`match_at("$1", int_val(100))`)
- `match_in_range(ref, range)` - represents `<ref> in <range>` (`match_in_range("$1", range(int_val(100), int_val(200)))`)
- `regexp(regexp, mods)` - represents regular expression in form ``/<regexp>/<mods>` (`regexp("^a.*b$", "i")`)
- `for_loop(spec, var, set, body)` - represents `for` loop over array or set of integers (`for_loop(any(), "i", range(int_val(100), int_val(200)), match_at("$1", id("i")))`)
- `for_loop(spec, var1, var2, set, body)` - represents `for` loop over dictionary (`for_loop(any(), "k", "v", id("pe").access("version_info"), True)`)
- `for_loop(spec, set, body)` - represents `for` loop over set of string references (`for_loop(any(), set({string_ref("$*")}), match_at("$", int_val(100)))`)
- `of(spec, set)` - represents `<spec> of <set>` (`of(all(), them())`)
- `of_in_range(spec, set, range)` - represents `<spec> of <set> in <range>` (`of(all(), them(), range(int_val(100), int_val(200)))`)
- `of_at(spec, set, offset)` - represents `<spec> of <set> at <offset>` (`of(all(), them(), int_val(200))`)
- `of(spec, iterable)` - represents `<spec> of <iterable>` (`of(any(), iterable([bool_val(False), bool_val(True)]))`)
- `paren(expr, [newline])` - represents parentheses around expressions and `newline` indicator for putting enclosed expression on its own line (`paren(int_val(10))`)
- `conjunction(terms, [newline])` - represents conjunction of `terms` and optionally puts them on each separate line if `newline` is set (`conjunction({id("rule1"), id("rule2")})`)
- `disjunction(terms, [newline])` - represents disjunction of `terms` and optionally puts them on each separate line if `newline` is set (`disjunction({id("rule1"), id("rule2")})`)

**Complex expression methods**

- `__invert__` - represents bitwise not (`~hex_int_val(0x100)`)
- `__neg__` - represents unary operator - (`-id("i")`)
- `__lt__` - represents operator < (`match_offset("$1") < int_val(100)`)
- `__gt__` - represents operator > (`match_offset("$1") > int_val(100)`)
- `__le__` - represents operator <= (`match_offset("$1") <= int_val(100)`)
- `__ge__` - represents operator >= (`match_offset("$1") >= int_val(100)`)
- `__add__` - represents operator + (`match_offset("$1") + int_val(100)`)
- `__sub__` - represents operator - (`match_offset("$1") - int_val(100)`)
- `__mul__` - represents operator * (`match_offset("$1") * int_val(100)`)
- `__truediv__` - represents operator / (`match_offset("$1") / int_val(100)`)
- `__mod__` - represents operator % (`match_offset("$1") % int_val(100)`)
- `__xor__` - represents bitwise xor (`match_offset("$1") ^ int_val(100)`)
- `__and__` - represents bitwise and (`match_offset("$1") & int_val(100)`)
- `__or__` - represents bitwise or (`match_offset("$1") | int_val(100)`)
- `__lshift__` - represents bitwise shift left (`match_offset("$1") << int_val(10)`)
- `__rshift__` - represents bitwise shift right (`match_offset("$1") >> int_val(10)`)
- `__call__` - represent call to function (`id("func")(int_val(100), int_val(200))`)
- `call(args)` - represents call to function (`id("func").call({int_val(100), int_val(200)})`)
- `contains(rhs)` - represents operator `contains` (`id("signature").contains(string_val("hello"))`)
- `matches(rhs)` - represents operator `matches` (`id("signature").matches(regexp("^a.*b$", "i"))`)
- `iequals(rhs)` - represents operator `iequals` (`id("signature").iequals(string_val("hello"))`)
- `icontains(rhs)` - represents operator `icontains` (`id("signature").icontains(string_val("hello"))`)
- `endswith(rhs)` - represents operator `endswith` (`id("signature").endswith(string_val("hello"))`)
- `iendswith(rhs)` - represents operator `iendswith` (`id("signature").iendswith(string_val("hello"))`)
- `startswith(rhs)` - represents operator `startswith` (`id("signature").startswith(string_val("hello"))`)
- `istartswith(rhs)` - represents operator `istartswith` (`id("signature").istartswith(string_val("hello"))`)
- `access(rhs)` - represents operator `.` as access to structure (`id("pe").access("numer_of_sections")`)
- `__getitem__` - represents operator `[]` as access to array (`id("pe").access("sections")[int_val(0)]`)
- `read_int8(be)` - represents call to special function `int8(be)` (`int_val(100).read_int8()`)
- `read_int16(be)` - represents call to special function `int16(be)` (`int_val(100).read_int16()`)
- `read_int32(be)` - represents call to special function `int32(be)` (`int_val(100).read_int32()`)
- `read_uint8(be)` - represents call to special function `uint8(be)` (`int_val(100).read_uint8()`)
- `read_uint16(be)` - represents call to special function `uint16(be)` (`int_val(100).read_uint16()`)

- `read_uint32(be)` - represents call to special function `uint32(be)` (`int_val(100).read_uint32()`)

**Hex strings**

- `YaraHexStringBuilder(byte)` - creates two nibbles out of byte value.
- `wildcard()` - creates ??
- `wildcard_low(nibble)` - <nibble>?
- `wildcard_high(nibble)` - ?<nibble>
- `jump_varying()` - [-]
- `jump_fixed(offset)` - [<offset>]
- `jump_varyingRange(low)` - [<low>-]
- `jump_range(low, high)` - [<low>-<high>]
- `alt([units])` - (unit1|unit2|...)

**Rule**

- `with_name(name)` - specify rule name
- `with_modifier(mod)` - specify whether rule is private or public (`Rule::Modifier::Private` or `Rule::Modifier::Public`)
- `with_tag(tag)` - specify rule tag
- `with_string_meta(key, value)` - specify string meta
- `with_int_meta(key, value)` - specify integer meta
- `with_uint_meta(key, value)` - specify unsigned integer meta
- `with_hex_int_meta(key, value)` - specify hexadecimal integer meta
- `with_bool_meta(key, value)` - specify boolean meta
- `with_plain_string(id, value)` - specify plain string with identifier `id` and content `value`
- `with_hex_string(id, str)` - specify hex string (`str` is of type `HexString`)
- `with_regexp(id, value, mod)` - specify regular expression with identifier `id` and content `value with modifiers ``mod` (These modifiers are tied to the regular expression and come after last /.)
- `with_condition(cond)` - specify condition
- `ascii()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `ascii` modifier to the list of its modifiers
- `wide()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `wide` modifier to the list of its modifiers
- `nocase()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `nocase` modifier to the list of its modifiers
- `fullword()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `fullword` modifier to the list of its modifiers
- `private()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `private` modifier to the list of its modifiers
- `xor()` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `xor` modifier to the list of its modifiers

- `xor(key)` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `xor(key)` modifier to the list of its modifiers
- `xor(low, high)` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `xor(low-high)` modifier to the list of its modifiers
- `base64` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `base64` modifier to the list of its modifiers
- `base64(alphabet)` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `base64(alphabet)` modifier to the list of its modifiers
- `base64wide` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `base64wide` modifier to the list of its modifiers
- `base64wide(alphabet)` - ties to the latest defined `with_plain_string()`, `with_hex_string()` or `with_regexp()` and adds `base64wide(alphabet)` modifier to the list of its modifiers

**YARA file**

- `with_module(name)` - specifies `import` of module named `name`
- `with_rule(rule)` - adds the rule into file

C++

**Basic expression functions**

- `filesize()` - represents `filesize` keyword
- `entrypoint()` - represents `entrypoint` keyword
- `all()` - represents `all` keyword
- `any()` - represents `any` keyword
- `them()` - represents `them` keyword
- `intVal(val, [mult])` - represents signed integer with multiplier (default: `IntMultiplier::None`) (`intVal(10)`, `intVal(10, IntMultiplier::Kilobytes)`, `intVal(10, IntMultiplier::Megabytes)`)
- `uintVal(val, [mult])` - represents unsigned integer with multiplier (default: `IntMultiplier::None`) (`intVal(10)`, `intVal(10, IntMultiplier::Kilobytes)`, `intVal(10, IntMultiplier::Megabytes)`)
- `hexIntVal(val)` - represents hexadecimal integer (`hexIntVal(0x10)`)
- `doubleVal(val)` - represents double floating-point value (`doubleVal(3.14)`)
- `stringVal(str)` - represents string literal (`stringVal("Hello World!")`)
- `boolVal(bool)` - represents boolean literal (`boolVal(true)`)
- `id(id)` - represents single identifier with name `id` (`id("pe")`)
- `stringRef(ref)` - represents reference to string identifier `ref` (`stringRef("$1")`)
- `set(elements)` - represents `(item1, item2, ...)` (`set({stringRef("$1"), stringRef("$2")})`)
- `range(low, high)` - represents `(low .. high)` (`range(intVal(100), intVal(200))`)
- `matchCount(ref)` - represents match count of string identifier `ref` (`matchCount("$1")`)
- `matchLength(ref, [n])` - represent `n``th match (default: 0) length of string identifier ``ref` (`matchLength("$1", intVal(1))`)

- `matchOffset(ref, [n])` - represents `n``th match (default: 0) offset of string identifier ``ref` (`matchOffset("$1", intVal(1))`)
- `matchAt(ref, expr)` - represents `<ref> at <expr>` (`matchAt("$1", intVal(100))`)
- `matchInRange(ref, range)` - represents `<ref> in <range>` (`matchInRange("$1", range(intVal(100), intVal(200)))`)
- `regexp(regexp, mods)` - represents regular expression in form ``/<regexp>/<mods>` (`regexp("^a.*b$", "i")`)
- `forLoop(spec, var, set, body)` - represents `for` loop over array or set of integers (`forLoop(any(), "i", range(intVal(100), intVal(200)), matchAt("$1", id("i")))`)
- `forLoop(spec, var1, var2, set, body)` - represents `for` loop over dictionary (`for_loop(any(), "k", "v", id("pe").access("version_info"), true)`)
- `forLoop(spec, set, body)` - represents `for` loop over set of string references (`forLoop(any(), set({stringRef("$*")}), matchAt("$", intVal(100)))`)
- `of(spec, set)` - represents `<spec> of <set>` (`of(all(), them())`)
- `ofInRange(spec, set, range)` - represents `<spec> of <set> in <range>` (`of(all(), them(), range(intVal(100), intVal(200)))`)
- `ofAt(spec, set, offset)` - represents `<spec> of <set> at <offset>` (`of(all(), them(), intVal(200))`)
- `paren(expr, [newline])` - represents parentheses around expressions and `newline` indicator for putting enclosed expression on its own line (`paren(intVal(10))`)
- `conjunction(terms, [newline])` - represents conjunction of `terms` and optionally puts them on each separate line if `newline` is set (`conjunction({id("rule1"), id("rule2")})`). The `terms` parameter can be an array containing other expressions to be put together in the conjunction. But also `terms` can be an array of pairs, where each pair contains a term to be put in the conjunction and a comment, which will be associated with the term and printed on the same line
- `disjunction(terms, [newline])` - represents disjunction of `terms` and optionally puts them on each separate line if `newline` is set (`disjunction({id("rule1"), id("rule2")})`). The `terms` parameter can be an array containing other expressions to be put together in the disjunction. But also `terms` can be an array of pairs, where each pair contains a term to be put in the disjunction and a comment, which will be associated with the term and printed on the same line

**Complex expression methods**

- `operator!` - represents logical `not` (`!boolVal(true)`)
- `operator~` - represents bitwise not (`~hexIntVal(0x100)`)
- `operator-` - represents unary operator - (`-id("i")`)
- `operator&&` - represents logical `and` (`id("rule1") && id("rule2")`)
- `operator||` - represents logical `or` (`id("rule1") || id("rule2")`)
- `operator<` - represents operator < (`matchOffset("$1") < intVal(100)`)
- `operator>` - represents operator > (`matchOffset("$1") > intVal(100)`)
- `operator<=` - represents operator <= (`matchOffset("$1") <= intVal(100)`)
- `operator>=` - represents operator >= (`matchOffset("$1") >= intVal(100)`)
- `operator+` - represents operator + (`matchOffset("$1") + intVal(100)`)
- `operator-` - represents operator - (`matchOffset("$1") - intVal(100)`)

- `operator*` - represents operator * (`matchOffset("$1") * intVal(100)`)
- `operator/` - represents operator / (`matchOffset("$1") / intVal(100)`)
- `operator%` - represents operator % (`matchOffset("$1") % intVal(100)`)
- `operator^` - represents bitwise xor (`matchOffset("$1") ^ intVal(100)`)
- `operator&` - represents bitwise and (`matchOffset("$1") & intVal(100)`)
- `operator|` - represents bitwise or (`matchOffset("$1") | intVal(100)`)
- `operator<<` - represents bitwise shift left (`matchOffset("$1") << intVal(10)`)
- `operator>>` - represents bitwise shift right (`matchOffset("$1") >> intVal(10)`)
- `operator()` - represent call to function (`id("func")(intVal(100), intVal(200))`)
- `call(args)` - represents call to function (`id("func").call({intVal(100), intVal(200)})`)
- `contains(rhs)` - represents operator `contains` (`id("signature").contains(stringVal("hello"))`)
- `matches(rhs)` - represents operator `matches` (`id("signature").matches(regexp("^a.*b$", "i"))`)
- `iequals(rhs)` - represents operator `iequals` (`id("signature").iequals(stringVal("hello"))`)
- `icontains(rhs)` - represents operator `icontains` (`id("signature").icontains(stringVal("hello"))`)
- `endswith(rhs)` - represents operator `endswith` (`id("signature").endswith(stringVal("hello"))`)
- `iendswith(rhs)` - represents operator `iendswith` (`id("signature").iendswith(stringVal("hello"))`)
- `startswith(rhs)` - represents operator `startswith` (`id("signature").startswith(stringVal("hello"))`)
- `istartswith(rhs)` - represents operator `istartswith` (`id("signature").istartswith(stringVal("hello"))`)
- `access(rhs)` - represents operator `.` as access to structure (`id("pe").access("numer_of_sections")`)
- `operator[]` - represents operator `[]` as access to array (`id("pe").access("sections")[intVal(0)]`)
- `readInt8(be)` - represents call to special function `int8(be)` (`intVal(100).readInt8()`)
- `readInt16(be)` - represents call to special function `int16(be)` (`intVal(100).readInt16()`)
- `readInt32(be)` - represents call to special function `int32(be)` (`intVal(100).readInt32()`)
- `readUInt8(be)` - represents call to special function `uint8(be)` (`intVal(100).readUInt8()`)
- `readUInt16(be)` - represents call to special function `uint16(be)` (`intVal(100).readUInt16()`)
- `readUInt32(be)` - represents call to special function `uint32(be)` (`intVal(100).readUInt32()`)
- `comment(message, [multiline], [indent], [linebreak])` - adds a comment `message` to the expression which then appears in the formatted text before the expression. Only the `message` parameter is required, the `multiline` (default `false`), `indent` (default "") and `linebreak` (default `true`) parameters are optional
- `commentBehind(message, [multiline], [indent], [linebreak])` - adds a comment `message` to the expression which then appears in the formatted text after the expression. Only the `message` parameter is required, the `multiline` (default `false`), `indent` (default "") and `linebreak` (default `true`) parameters are optional

**Hex strings**

- `YaraHexStringBuilder(byte)` - creates two nibbles out of byte value.
- `wildcard()` - creates ??

- `wildcardLow(nibble)` - `<nibble>?`
- `wildcardHigh(nibble)` - `?<nibble>`
- `jumpVarying()` - `[-]`
- `jumpFixed(offset)` - `[<offset>]`
- `jumpVaryingRange(low)` - `[<low>-]`
- `jumpRange(low, high)` - `[<low>-<high>]`
- `alt(units)` - `(unit1|unit2|...)`

**Rule**

- `withName(name)` - specify rule name
- `withModifier(mod)` - specify whether rule is private or public (`Rule::Modifier::Private` or `Rule::Modifier::Public`)
- `withTag(tag)` - specify rule tag
- `withStringMeta(key, value)` - specify string meta
- `withIntMeta(key, value)` - specify integer meta
- `withUIntMeta(key, value)` - specify unsigned integer meta
- `withHexIntMeta(key, value)` - specify hexadecimal integer meta
- `withBoolMeta(key, value)` - specify boolean meta
- `withPlainString(id, value, mod)` - specify plain string with identifier `id` and content `value` with modifiers `mod` (`String::Modifiers::Ascii`, `String::Modifiers::Wide`, `String::Modifiers::Nocase`, `String::Modifiers::Fullword`)
- `withHexString(id, str)` - specify hex string (`str` is of type `std::shared_ptr<HexString>`)
- `withRegexp(id, value, mod)` - specify regular expression with identifier `id` and content `value with modifiers ``mod` (Modifiers here are different than modifiers in plain string. These modifiers are tied to the regular expression and come after last `/`.)
- `withCondition(cond)` - specify condition
- `ascii()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `ascii` modifier to the list of its modifiers
- `wide()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `wide` modifier to the list of its modifiers
- `nocase()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `nocase` modifier to the list of its modifiers
- `fullword()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `fullword` modifier to the list of its modifiers
- `private()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `private` modifier to the list of its modifiers
- `xor()` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `xor` modifier to the list of its modifiers
- `xor(key)` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `xor(key)` modifier to the list of its modifiers

- `xor(low, high)` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `xor(low-high)` modifier to the list of its modifiers
- `base64` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `base64` modifier to the list of its modifiers
- `base64(alphabet)` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `base64(alphabet)` modifier to the list of its modifiers
- `base64wide` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `base64wide` modifier to the list of its modifiers
- `base64wide(alphabet)` - ties to the latest defined `withPlainString()`, `withHexString()` or `withRegexp()` and adds `base64wide(alphabet)` modifier to the list of its modifiers

**YARA file**

- `withModule(name)` - specifies `import` of module named `name`
- `withRule(rule)` - adds the rule into file

CHAPTER

FIVE

# FORMATTING RULESETS

Our recent addition to the things yaramod should do is automatic formatting of YARA rules into form which is mostly used for our YARA rules and in which we expect YARA rules when they come from other sources. Having formatted YARA rules has same importance as having properly formatted code. It is readable even if you are not the original author of the ruleset and also formatting gives additional semantics without explicitly stating it if you are used to it. Formatting that is designed well can also help you catch problems in your ruleset even before you actually submit it and start using it. Therefore we came up with this formatting best to suite our needs which incorporates:

- Use of TABs
- `{` and `}` on their own lines
- `meta`, `strings` and `condition` indented once
- Their contents idented twice
- New lines when using `and`, `or`, `(` and `)` in condition
- Comments that are on the consecutive lines are properly aligned
- Single space in every other place wher you can put multiple whitespaces

Here is an example of the rule before and after formatting:

```
rule abc {
strings:
  $s01  = "Hello"
  $s02  = "World"
condition:
  $s01 and $s02
}
```

```
// This is my rule
rule abc
{
        strings:
                $s01 = "Hello" // String 1
                $s02 = "World" // String 2
        condition:
                $s01 and
                $s02
}
```

In order to use it access `getTextFormatted()` (`text_formatted` in Python) on YARA file. It is important to state that our formatter **does not remove comments**.

This feature is brand new and is still evolving. We would really appreciate your feedback here so we can improve it further. There is a staging area on our wiki where we collect ideas on where to go with autoformatting. Just file an issue if you have any remarks.

CHAPTER

SIX

# MODIFYING RULESETS

## 6.1 Modifying Metas

In Yaramod, we are able to modify existing YARA rules through many methods. For example, we can add new metas with method `Rule::addMeta(const std::string& name, const Literal& value)` which inserts a new meta also into the `TokenStream`. This method can be used through python as `add_meta` as follows:

Python

```
yara_file = yaramod.Yaramod().parse_string(
'''
rule rule_with_added_metas {
    condition:
        true
}'''
)

rule = yara_file.rules[0]
rule.add_meta('int_meta', yaramod.Literal(42))
rule.add_meta('bool_meta', yaramod.Literal(False))
```

C++

```
yaramod::Yaramod ymod;

std::stringstream input;
input << R"(
rule rule_with_added_metas {
    condition:
        true
})";
auto yarafile = ymod.parseStream(input);
const auto& rule = yarafile->getRules()[0];

uint64_t u = 42;
rule->addMeta("int_meta", Literal(u));
rule->addMeta("bool_meta", Literal(false));
```

This will add both `meta` and `:` tokens and create two metas and the formatted text will look as follows:

```
rule rule_with_added_metas
{
    meta:
        int_meta = 42
        bool_meta = false
    condition:
        true
}
```

We can also modify existing metas using python bindings `Rule::get_meta_with_name` and `Meta::value`. The following code will change the integer value `42` of `int_meta` into string `forty two`:

Python

```
meta = rule.get_meta_with_name('int_meta')
meta.value = yaramod.Literal('forty two')
```

C++

```
auto meta = rule->getMetaWithName("int_meta");
meta->setValue(Literal("forty two"));
```

With the following result:

```
rule rule_with_added_metas
{
    meta:
        int_meta = "forty two"
        bool_meta = false
    condition:
        true
}
```

## 6.2 Modifying Visitors

### 6.2.1 The Conditions in Yaramod

To demonstrate how to alter conditions of YARA rules we first need to make sure we understand their structure.

Conditions in Yaramod are tree-like structures where each node is a derived class of the `Expression` class. Based on the arity of each expression is the number of nodes under it. The leaves of the tree correspond to the expressions of arity 0 such as `EntrypointExpression` or `StringExpression`. The expressions like `NotExpression` or other derived classes of `UnaryOpExpression` always have one other expression under them. And then we also have `BinaryOpExpression` with arity 2 or even more, because the `ForExpression` has another 3 expressions referenced under it.

### 6.2.2 Visiting Expressions

Yaramod provides two kinds of visitors enabling the user to observe or modify existing expressions. This page is about more interesting modifying visitors:

Both `ObservingVisitor` and `ModifyingVisitor` classes define a visit method for each derived `Expression` class as a parameter. These methods are pre-defined not to change/do anything in the expression they are called with. The only thing these pre-defined methods do is they trigger visiting of all subexpressions. This means, that after calling a visit on an expression, the visit methods are recursively called upon each of the subnodes in the expression tree structure. Until we modify the visit methods, each such call performs no actions on the nodes.

We will now focus on the `ModifyingVisitor` class which supplies also the `modify` method with an `Expression* expr` as a parameter. This method arranges that a correct `visit` method is called with `expr` as the parameter. Let us now describe three types of modifying visitors we can write with three examples.

### 6.2.3 Custom Visitor Examples

Following visitor provides specification of the `visit` method for `StringExpression`. It 'to uppers' the `id` of the `StringExpression`. It is modifying existing `StringExpression` instance:

Python

```
class StringExpressionUpper(yaramod.ModifyingVisitor):
    def process(self, yara_file: yaramod.YaraFile):
        for rule in yara_file.rules:
            self.modify(rule.condition)
    def visit_StringExpression(self, expr: yaramod.Expression):
        expr.id = expr.id.upper()
```

C++

```
class StringExpressionUpper : public yaramod::ModifyingVisitor
{
public:
    void process(const YaraFile& file)
    {
        for (const std::shared_ptr<Rule>& rule : file.getRules())
            modify(rule->getCondition());
    }
    virtual yaramod::VisitResult visit(StringExpression* expr) override
    {
        std::string id = expr->getId();
        std::string upper;
        for (char c : id)
            upper += std::toupper(c);
        expr->setId(upper);
        return {};
    }
};
```

We can now use this visitors instance `visitor` to alter all conditions of rules present in a given yara file simply by calling `visitor.process(yara_file)`. The next example will show a case when we replace existing visited node in the expression syntax tree by another new node:

Python

```
class RegexpVisitor(yaramod.ModifyingVisitor):
    def add(self, yara_file: yaramod.YaraFile):
        for rule in yara_file.rules:
            self.modify(rule.condition)

    def visit_RegexpExpression(self, expr: yaramod.Expression):
        output = yaramod.regexp('abc', 'i').get()
        expr.exchange_tokens(output)
        return output
```

C++

```
class RegexpVisitor : public yaramod::ModifyingVisitor
{
public:
    void process(const YaraFile& file)
    {
        for (const std::shared_ptr<Rule>& rule : file.getRules())
            modify(rule->getCondition());
    }
    virtual yaramod::VisitResult visit(RegexpExpression* expr) override
    {
        auto new_condition = regexp("abc", "i").get();
        expr->exchangeTokens(new_condition.get());
        return new_condition;
    }
};
```

This `visit` methods requires calling of a `exchange_tokens` method which deletes all tokens that the original expression refered to. Then it extracts all tokens from the supplied new expression and moves them to place where the original expression had its tokens stored.

In the third example we will show how to deal with a situation where we need to modify existing expression while keeping part of it's subexpressions. The following approach will let us use Yaramod expression builders to create new expressions from existing expressions that are already used in our rule.

Let's now assume that we need to modify each `EqExpression` in the expression syntax tree. We can do it by writing our own derived class of `ModifyingVisitor`. The new class will override the `visit(EqExpression* expr)` method in the following manner:

Python

```
class EqModifier(yaramod.ModifyingVisitor):
    def add(self, yara_file: yaramod.YaraFile):
        for rule in yara_file.rules:
            rule.condition = self.modify(rule.condition)

    def visit_EqExpression(self, expr: yaramod.Expression):
        context = yaramod.TokenStreamContext(expr)
        expr.left_operand.accept(self)
        expr.right_operand.accept(self)
        output = (yaramod.YaraExpressionBuilder(expr.right_operand) != yaramod.
↪YaraExpressionBuilder(expr.left_operand)).get()
```

(continues on next page)

(continued from previous page)

```
        self.cleanUpTokenStreams(context, output)
        return output
```

C++

```
class EqModifier : public yaramod::ModifyingVisitor
{
public:
    void process_rule(const std::shared_ptr<Rule>& rule)
    {
        auto modified = modify(rule->getCondition());
        rule->setCondition(std::move(modified));
    }
    virtual VisitResult visit(EqExpression* expr) override
    {
        TokenStreamContext context(expr);
        auto leftResult = expr->getLeftOperand()->accept(this);
        if (resultIsModified(leftResult))
            expr->setLeftOperand(std::get<std::shared_ptr<Expression>>(leftResult));
        auto rightResult = expr->getRightOperand()->accept(this);
        if (resultIsModified(rightResult))
            expr->setRightOperand(std::get<std::shared_ptr<Expression>>(rightResult));

        auto output = ((YaraExpressionBuilder(expr->getRightOperand())) !=␣
↪(YaraExpressionBuilder(expr->getLeftOperand()))).get();

        cleanUpTokenStreams(context, output.get());
        return output;
    }
};
```

The first line in the `visit` method is simply creating a snapshot `context` of the `TokenStream` and first and last `Token` of the processed expression. Because here we deal with an expression of non-zero arity, we have to trigger the Visitor also on it's subnodes. This happens on the next two lines in Python. Then a new expression `output` is created. The `cleanUpTokenStreams` method makes sure, that all remaining tokens of the old version of the expression, that have not been used by the builder, are deleted. Then all tokens maintained by the builder are moved back to the original `TokenStream` on the right place.

CHAPTER

SEVEN

# EXAMPLES

There are 2 examples available in the repository as of now. They are both implemented in C++ and Python so that you can check out the language which you prefer.

## 7.1 Dump rule AST

Dump rule AST example shows you how you can implement traversal of the whole AST of the YARA rule condition. It prints out each node it visits and dumps some information to the output.

## 7.2 Boolean simplifier

Boolean simplifier shows you how you can implement visitor which also changes the condition based on some kind of analysis. In this example, it tries to evaluate the logical operators `and`, `or` and `not` and simplify the condition. It is built on the fact that `<anything> and false = false` and `<anything> or true = true`. Then it just uses typical truth tables for `and`, `or` and `not`.

CHAPTER

EIGHT

# DEVELOPMENT

## 8.1 Architecture

Yaramod is a C++ library with Python bindings capable of parsing, building, formatting and also modifying YARA rules; hence the main four parts of Yaramod are:

**Parser of YARA rules**
: The main parser class is the `ParserDriver` class declared in the header file `parser_driver.h` and defined in source file `parser_driver.cpp`. The parser is based on POG and its grammar and tokens are defined in methods `defineTokens` and `defineGrammar` of the `ParserDriver` class. Detailed wiki page on how to use yaramod to parse YARA rules can be found in this section.

**Builder of YARA rules**
: The builder machinery is declared within the builder folder. The `YaraExpressionBuilder` creates expressions so that YARA rules conditions can be created. The `YaraHexStringBuilder` is a tool for easy creation of hexadecimal strings. The `YaraRuleBuilder` helps to create YARA rules and the `YaraFileBuilder` is there to construct YARA files from rules and module imports. More on the construction of YARA files is written here.

**YARA rules formatting**
: The main component taking care of proper formatting of YARA files is the `TokenStream` defined in file `tokenstream.h`. Each `YaraFile` instance holds a `TokenStream` instance in which all Tokens that the `YaraFile` refers to are stored. The `TokenStream::getText` method prints the tokens formatted in the desired format. The `YaraFile::getTextFormatted` method simply calls the `getText` method of the TokenStream that it owns. Please see wiki page for more on formatting.

**YARA rules modifying visitor**
: The class `ModifyingVisitor` is defined in *modifying_visitor.h* and serves as the base class for our custom visitors designed to modify specific parts of visited conditions. See section Modifying Rulesets for more information and examples.

## 8.2 Run it locally

See Installation section.

CHAPTER

NINE

# DEPLOYMENT

A new version of Yaramod is released performing the following steps:

- Open `include/yaramod/yaramod.h` and update `YARAMOD_VERSION_PATCH` (with big changes we also increment `YARAMOD_VERSION_MINOR` and set `YARAMOD_VERSION_PATCH` to 0).
- Open `docs/rtd/conf.py` and update version in `release =`.
- In `CHANGELOG.md` add entry for the new version. List all important changes with links to issues and PRs.
- Commit the changes with message "Release v<?>.<?>.<?>".
- Create a git tag by running `git tag -a v<?>.<?>.<?> -m "Release v<?>.<?>.<?>"`.
- Push the new tag with `git push origin v<?>.<?>.<?>`.
- Push the commit after the release to master with `git push origin master`.

CHAPTER

TEN

# TROUBLESHOOTING

If you encounter a problem using Yaramod, it is usually easy to determine which of the four components the problem is related to:

- A `ParserError` indicates parsing problem and the message should include the first problematic token of the imput.
- A `BuilderError` probably means that your service uses the Builder incorrectly.
- If you use the `getTextFormatted()` (`text_formatted` in Python) method of a YARA file and the output seems wrong, the problem will be probably in the the `getText` method of the `TokenStream` class. Please create a ticket and supply the input and the wrong output with some explanation so we can look into it.
- There can also be some problems when using modifying visitors. In the case your modifying visitor modifies the unformatted text very well, but the formatted text is wrong (usually missing tokens or bad ordering of tokens) it may indicate unused method `cleanUpTokenStreams` of the `ModifyingVisitor` class. Or the method could be used in a wrong matter. Please consult our examples in the section Modifying Rulesets and if that does not help consider contacting the authors.

CHAPTER

ELEVEN

# FUTURE OF YARAMOD

This page is intented to be a collection of ideas for the future of yaramod.