NISTIR XXXX

Vulnerability Test Suite Generator (VTSG) Version 3

Paul E. Black
William Mentzer
Elizabeth Fong
Bertrand Stivalet

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.XXXX

National Institute of
Standards and Technology
U.S. Department of Commerce

NISTIR XXXX

Vulnerability Test Suite Generator (VTSG) Version 3

Paul E. Black
Software and Systems Division
Information Technology Laboratory

William Mentzer
California State University
San Bernardino, California

Elizabeth Fong
affiliation
location

Bertrand Stivalet
affiliation
location

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.XXXX

October 2020

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology

National Institute of Standards and Technology
Internal Report XXXX
Natl. Inst. Stand. Technol. Intern. Rep. XXXX, 28 pages (October 2020)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.XXXX

Abstract

The Vulnerability Test Suite Generator (VTSG) can create vast numbers of synthetic programs with and without specific flaws or vulnerabilities. It was designed by the Software Assurance Metrics and Tool Evaluation (SAMATE) team and originally implemented by students from TELECOM Nancy. The latest version is structured to be able to generate vulnerable and nonvulnerable synthetic programs expressing specific flaws in *any* programming language. It has libraries to generate PHP, C#, and Python programs. This document may help if you are trying to generate test cases written in PHP, C#, or Python or if you are modifying VTSG Version 3 to generate test cases in other programming languages.

Key words

Software assurance; static analyzer; test case generator; software vulnerabilities.

Table of Contents

1. Introduction

We will use vulnerability, instead of flaw or weakness. other SAMATE documents use “weakness”.

The Vulnerability Test Suite Generator (VTSG) generates collections of vulnerable and non-vulnerable synthetic programs expressing specific flaws. The programs can be used as test cases to evaluate static analyzers. Each test case targets one flaw. There are two types of test cases: cases with flawed code, leading to a vulnerability, and cases that have similar behavior, but have the flaw corrected. That is, no vulnerability. Exactly corresponding vulnerable and non-vulnerable cases could, in theory, be generated. However, since each test case is generated separately, there is no exact correspondence between cases.

The generator is written in Python 3.

1.1 History

VTS version 1 only generated C# programs. VTS version 2 generated PHP programs [1] in addition. Version 2 is more customizable to generate other programming languages. VTSG version 3 (V3) systematically maintains indentation, so also generates Python programs. VTSG V3 produces manifest files in the Static Analysis Results Interchange Format (SARIF) Version 2.1.0 [2] format.

Readers can download the PHP and C# test cases generated by earlier versions from NIST’s Software Assurance Reference Dataset (SARD): http://samate.nist.gov/SARD

The global project can be found at: https://github.com/usnistgov/VTSG

1.2 Install Supporting Packages

The following instructions are provided for users, who may not have these packages already installed on their Linux machines. Users who already have these packages may skip this section.
To download files from Github, one must install the *Git* package. Use the following command to install git:

```
sudo apt-get install git
```

To execute Python source code, one must install the *pip Python* package manager. Use the following command to install it:

```
sudo apt-get install python-pip
```

VTSG is written in Python 3, so Python 3 must be installed, too. Use the following command to install Python 3:

```
sudo apt-get install python3
```

In addition, the *pip Python 3* package manager must be installed, using the following command:

```
sudo apt-get install python3-pip
```

To validate C# test cases, *mono* and *mcs* must be installed. The Mono project created *mono* as an open source platform, which implements the .NET Framework. Class libraries and C# compilation are enabled by *mcs* (http://github.com/mono/mono).
Use the following command to obtain mono-complete:

```
sudo apt-get install mono-complete
```

Type the following command to obtain *mcs*:

```
sudo apt-get install mcs
```

1.3 Install VTSG

To copy the generic VTSG from GitHub to a local Linux machine, change to a directory under which you want VTSG installed.
Returning to the GitHub website, one will see a green box, labelled "Clone or download". Click on it and use the clipboard tab to copy the web URL.
Type the following command to copy the source code and other material to the local directory:

```
git clone https://github.com/usnistgov/VTSG.git
```

If one types the `ls -a` command, one will see that the **VTSG** directory was created.
Go to that directory, using the following command:

```
cd VTSG
```

To install the dependencies, use the following command:

```
pip3 install --user -r requirements.txt
```

1.4 Users

There are two groups who will typically use VTSG. The first group comprises people requiring test cases written in PHP, C#, or Python to evaluate a static analyzer. These users must know how to invoke VTSG with command line interface and retrieve the appropriate sample from the generated and categorized folders.

The second targeted audience of the VTSG is comprised of people wishing to generate test cases using a programming language other than the languages currently supported. The second group of users must use the template to construct the program using the XML tags, execute VTSG with command line interface or another input method, and retrieve the samples from the generated and categorized file folders.

1.5 Vulnerabilities Supported by VTSG

Vulnerabilities are encoded in the language files, see Sec. 4. Some of the OWASP Top 10 [3] and Common Weakness Enumerations (CWEs) [4] are encoded. The following list shows the vulnerabilities currently in each language:

- SQL Injection (C#, PHP)
- XPath Injection (C#)
- LDAP Injection (C#)
- OS Command Injection (C#)
- Path traversal (C#)

2. Command Line Interface

For users who wish to generate PHP or C# test suites, a command line interface can generate all test cases or a specific group of test cases based on several options. For example, the user can generate vulnerable or non-vulnerable test cases based on selected OWASP categories or CWE weaknesses. The user must specify the programming language. The invocation command looks like this:

```
$ python3 vtsg.py -l {php,cs,py} <options>
```

Where ⟨options⟩ can be selected from Table 1.

-h –help	Show help and quit
–version	Show version number and quit
-l LANGUAGE –language=LANGUAGE	Language of generated cases. Currently one of php, for PHP cases, cs, for C# cases, or py, for Python cases. See Sec. 3.2.
-f FLAW_GROUP[,FLAW_GROUP]* –flaw-group=FLAW_GROUP[,FLAW_GROUP]*	Only generate cases with vulnerabilities from the specified group(s). See Sec. 4.4
-c CWE[,CWE]* –cwe=CWE[,CWE]*	Only generate cases with vulnerabilities from the specified CWE(s). See Sec. 4.4
-s –safe	Only generate non-vulnerable cases
-u –unsafe	Only generate vulnerable cases
-r DEPTH –depth=DEPTH	Maximum nested depth of complexities (Default: 1) See Sec. 3.3
-g NUMBER –number-generated=NUMBER	Maximum number of sink, filter, input (and exec query) combinations to generate. (Default: -1, meaning all) See below for explanation.
-d –debug	Not Used – Remove from code

Table 1. Options for Command Line Invocation

2.1 Explanation of Options

The default is to generate both the unsafe (buggy or vulnerable) test cases and the safe (not buggy) test cases. You can select either only safe (-s) or only unsafe (-u) cases. The options are mutually exclusive.

The -g (number-generated) option has limited utility. When the specified number of sink, filter, input (and exec query, if needed) combinations are generated, VTSG terminates. The default, -1, means generate all combinations.

Each combination of sink, filter, input, and exec query is elaborated with DEPTH nested complexities. Suppose there are 5 complexities and VTSG is invoked with `-r 2`. Each combination will have 1 (no complexities) + 5 (each complexity, not nested) + 25 (each complexities nested within every complexity) = 31 test cases. Hence VTSG may generate *far* more test cases than the number given with the -g option.

2.2 Example Invocations

Show the help message:

```
$ python3 vtsg.py --help
```

Generate all PHP test cases:

```
$ python3 vtsg.py -l php
```

Generate a C# (`-l cs`) test suite made of non-vulerable (unsafe) test cases (`-u`) with SQL injection vulnerabilities (`--cwe=89`) with up to 3 nested levels of complexity (`-r 3`).

```
$ python3 vtsg.py -l cs -r 3 --cwe=89 -u
```

3. Overview of VTSG

Explain that there are TWO structures. The structure of test cases generated and the structure of VTSG. Make it clear when each is being explained.

3.1 Overview of Test Case Generation

VTSG V3 generates test cases from information in Template, Input, Filter, Sink, Complexity and ExecQuery files (see Fig. 1):

- Template is the overall structure of each program.
- Input is the source of untrusted data in the program, e.g., command line, variable, files, and form methods.
- Filtering filters the input with sanitization functions, casting, deprecated functions, or simply no filtering.
- Sink is where a sensitive operation, such as a database query, is executed with potentially untrusted input and where the vulnerability is triggered.
- ExecQuery is an additional piece of code that is mandatory to trigger the vulnerability.
- Complexity is how the data flow and control flow are implemented in the structure of the program.

The content of these files is detailed in Sec. 4. Details of the generation process are explained in Sec. 3.3.

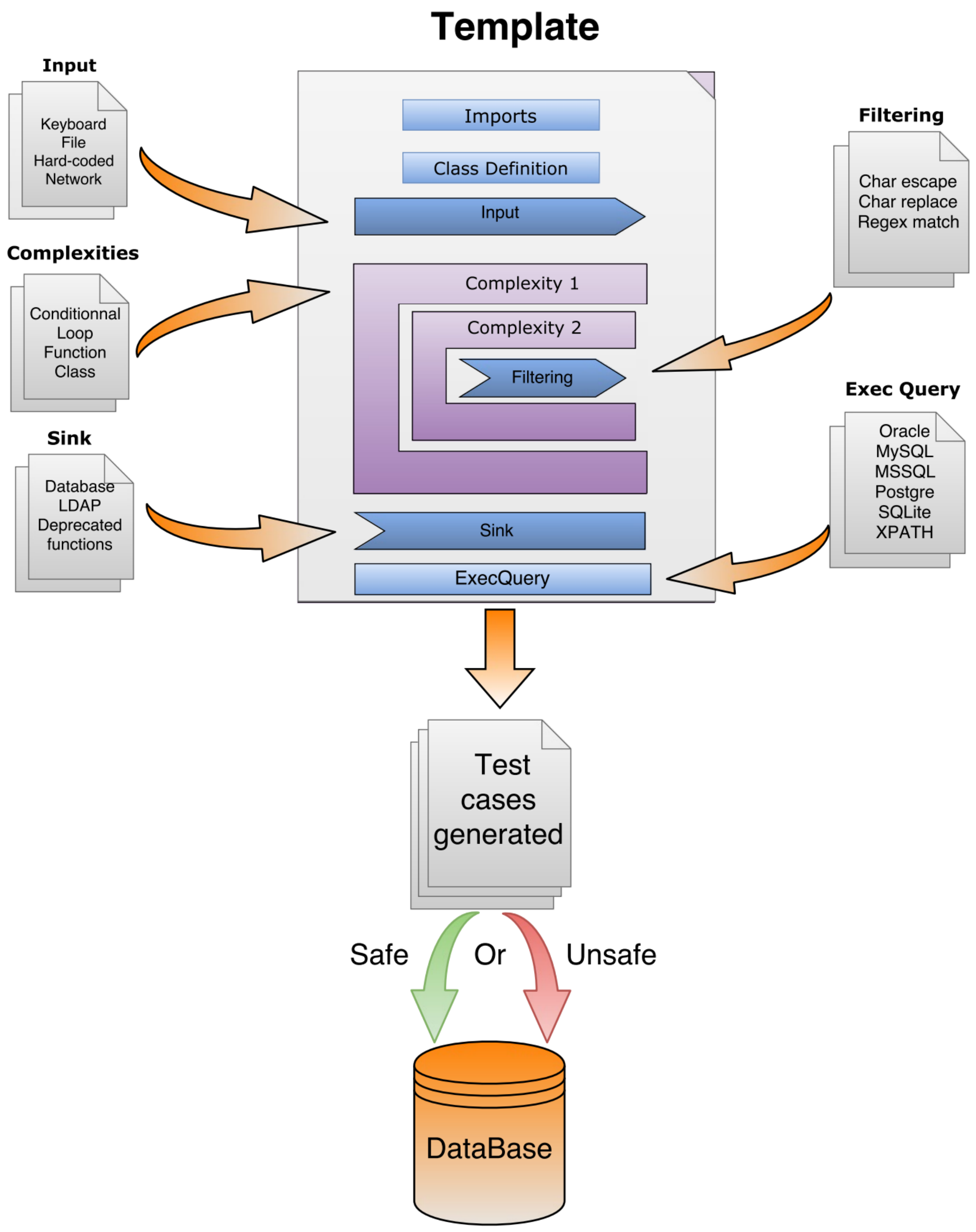


Fig. 1. Overview of VTSG test case generation process. The Template specifies how pieces are assembled. The Input, Filter, Complexities, Sink, and ExecQuery modules provide alternative code. An example of a generated test case is in Figs. 9 and 10.

3.2 VTSG Directory Structure

All these types of files are kept in a `templates` subdirectory. Under this is one subdirectory for each language. VTSG chooses the directory based on the language selected in the `-l` command line option. Each language subdirectory has six files for that language. The files are `file_template.xml`, `input.xml`, `complexities.xml`, `filtering.xml`, `sink.xml`, and `exec_queries.xml`. To add another language, simply create a subdirectory for that language with the six description files.

The `templates` directory has `file_rights.txt`, which is copied into each generated test case to declare license rights and authorship, see Sec. 4.1, and a `dtd` subdirectory, which has a document type definition (DTD) for Template, Input, Filter, and all other XML files.

3.3 Generation Process

Each test case is constructed based on the file Template, as shown in Fig. 1. Test cases are programs in a specific program language. Each test case is generated by assembling the modules according to the Template. The template may direct construction of a simple test case with just an Input, a Filter, and a Sink. The Filtering code may be embedded in data and control flow Complexity code.

VTSG generates test cases with two broad steps. First, VTSG selects Input, Filter, Complexities, Sink, and ExecQuery modules that are compatible with each other and consistent with any flaw group or CWE constraints the user gives on the command line. Second, VTSG composes test case source code from the selected modules, synthesizing variable and functions names, and writes the file(s).

The code structure is roughly

```
for each specified sink
    for each filter
        for each input
            for each exec query
                for up to DEPTH combinations of each complexity
                    compose a test case with these modules
```

The code is more complicated because only compatible modules are selected. In addition, some sinks do not need any input or filtering at all, see Sec. 4.4. The code is structured as a series of function calls to allow types of modules to be skipped. Here is a slightly more detailed overview of those steps:

```
for each sink:
    if sink is specified:
        if input is needed:
            select_filtering()
        else:
            select_exec_query()

def select_filtering():
    for each filter:
        if filter is compatible with sink:
            select_input()

def select_input()
    for each input:
        if input is compatible with filter:
            select_exec_query()

def select_query()
    if sink needs exec_query:
        for each exec_query:
            if exec_query is compatible with sink:
                select_complexity_or_compose()
    else:
        select_complexity_or_compose()

def select_complexity_or_compose():
    if input_type is not none:
        recursive_select_complexity()
    else:
        compose()

... and so forth
```

The *test_suite_gen.py* script creates a new object of the *Generator* class. The program iterates through all sink modules, selecting those specified by the user, see Sec. 2, or all of them if the user does not specify. It subsequently selects filters then inputs then exec queries and complexities that are compatible with the currently selected sink module.

When VTSG has selected a set of modules, it begins composing the code in them to generate the source code for a test case. The process of composing modules to generate source code is based on XML metadata tags. After the imports and class definition declaration for the specific program language, the "Input" metadata ⟨code⟩ portion is added to the test case. The ⟨input_type⟩ and ⟨output_type⟩ must be consistent with the "Filtering" and "Sink" XML tags. The "Filtering" metadata ⟨code⟩ portion, plus its ⟨flaw type⟩ and safety indicator, are added to the test case. Next, the "Sink" metadata ⟨code⟩ portion is added to the test case. Finally, the "ExecQuery" type is noted and the ⟨code⟩ portion of the "ExecQuery" is added to the test case. The test case is written to a file. The location of the file is described in Sec. 5.1. Section 5.2 describes how VTSG names the file.

VTSG generates many different test cases, both with and without flaws, with various control flow complexities. After VTSG finishes generating each vulnerability category, it displays how many safe (non-vulnerable) and unsafe (vulnerable) test cases it produced. VTSG generates hundreds of test cases in minutes.

VTSG is built to generate test cases with all consistent combinations of modules for the flaw groups and CWEs specified in the invocation. If VTSG is invoked with flaw groups (`-f`) or CWEs (`-c`), only sinks satisfying those specified are used. If no flaw groups are specified, all flaw groups are used. If no CWEs are specified, all CWEs are used.

The depth command line option, `-r` or `--depth`, specifies the most nested flow control complexities produced. VTSG generates test cases with all complexities up to the depth indicated. For example, the default depth, 1, leads VTSG to generate all test cases with no flow complexities and all test cases with one complexity. The option `-r 2` leads VTSG to generate all cases with no complexities, all cases with one complexity, and all cases with two nested complexities. See Sec. 3.4 for an example of three nested control flow complexities.

3.4 Code Complexities

In theory, a static analysis tool only needs to process a few lines of code that embody the vulnerability. In practice, a tool must analyze most of the program, noting its control and data flows, to accurately track data and determine the conditions when the code with weaknesses may be executed. Code complexities are constructs that may confuse static analysis tools. Each code complexity element can have many different attributes associated with it. They are combined and nested to create real source code. For example, the value of an expression may come from a constant, a single variable, some arithmetic combination, or the return value of a function call. Flow of control may be influenced by loops, conditionals, and functions calls. Also, there could be many layers or depths of such nesting structures.

The Complexity identifications currently available for PHP and C# are listed in Table 3 in the Appendix.

Here is an example code complexity from `cwe_89__I_shell_commands__F_no_filtering__S_select_from-concatenation_simple_quote__EQ_mysql__3-2.5-9-21_File1.cs`:

```
if((Math.Pow(4, 2)<=42)){
    switch(6){
        case(6):
            Class_489618 var_489618 = new Class_489618(tainted_5);
            tainted_6 = var_489618.get_var_489618();
            tainted_7 = tainted_6;
            break;
        default:
            break;
    }
}else{
    {}
}
```

The above has complexity depth 3, note __3 near the end of the file name. They correspond to:

Level 1 is the conditional id = 2.5
    Level 2 is the switch statement id = 9
        Level 3 is the call of a method from a Class defined in a different file, id = 21.

4. Template, Input, Filter, Sink, Exec_Query, and Complexity Files

These XML files are required when adding new functionality to VTSG. There are a few XML-specific caveats that must be paid attention to when creating these files. Table 2 lists the symbols that may cause errors during the process and the XML equivalent replacement necessary to complete the task without error.

Character	Replacement
<	&lt;
>	&gt;
"	&quot;
'	&apos;
&	&amp;

Table 2. Replacement sequences for characters that are treated in a special way in XML files.

Characteristics of modules and their information are stored in XML files. This section describes the structure of each type of module and the meaning of each element and its tags.

Most of the file types have an example followed by an explanation of what it does and what it generates at the end.

Each language directory has one file for each type. That is, one `file_template.xml` file, one `input.xml` file, one `filtering.xml` file, one `complexities.xml` file, one `sink.xml` file, and one `exec_queries.xml` file.

The `input.xml`, `filtering.xml`, `complexities.xml`, `exec_queries.xml`, and `sink.xml` files may have many modules in them. For example, `input.xml` typically has many input modules, Sec. 4.2, each getting input from a different source.

4.1 File Template

```
<template type="" name="">
    <file_extension></file_extension>
    <comment>
```

```
        <open></open>
        <close></close>
        <inline></inline>
    </comment>
    <syntax>
        <statement_terminator></statement_terminator>
    </syntax>
    <namespace></namespace>
    <variables prefix="" import_code="using {{import_file}};">
        <variable type="" code="" init=""/>
    </variables>
    <imports>
        <import></import>
    </imports>
    <code></code>
</template>
```

- name: Programming language name, e.g., PHP, CSharp, or Python. This appears in the manifest. It is also the subdirectory under the TestSuite directory where all the generated test cases are placed.
- file_extension: Extension of the generated files.
- comment: Strings indicating comments.
  - open: string to begin a comment, which may span many lines
  - close: string to end a comment, which may span many lines
  - inline: string to begin a one-line comment
- syntax: Other language-specific syntax.
  - statement_terminator: string to show the end of a statement. This is semicolon `<statement_terminator>;</statement_terminator>` in PHP, C, Java, and C#. Python does not have a terminator, so this is the empty string: `<statement_terminator></statement_terminator>`.
- namespace: Namespace name, if applicable
- variables: Information variable names and types and how to include libraries.
  - prefix: Any prefix required for variable name, $ for PHP. Leave it blank if not required.
  - import_code: Code to include a library. The code should have the placeholder `{{import_file}}`. For example, `#include <{{import_name}}>`).

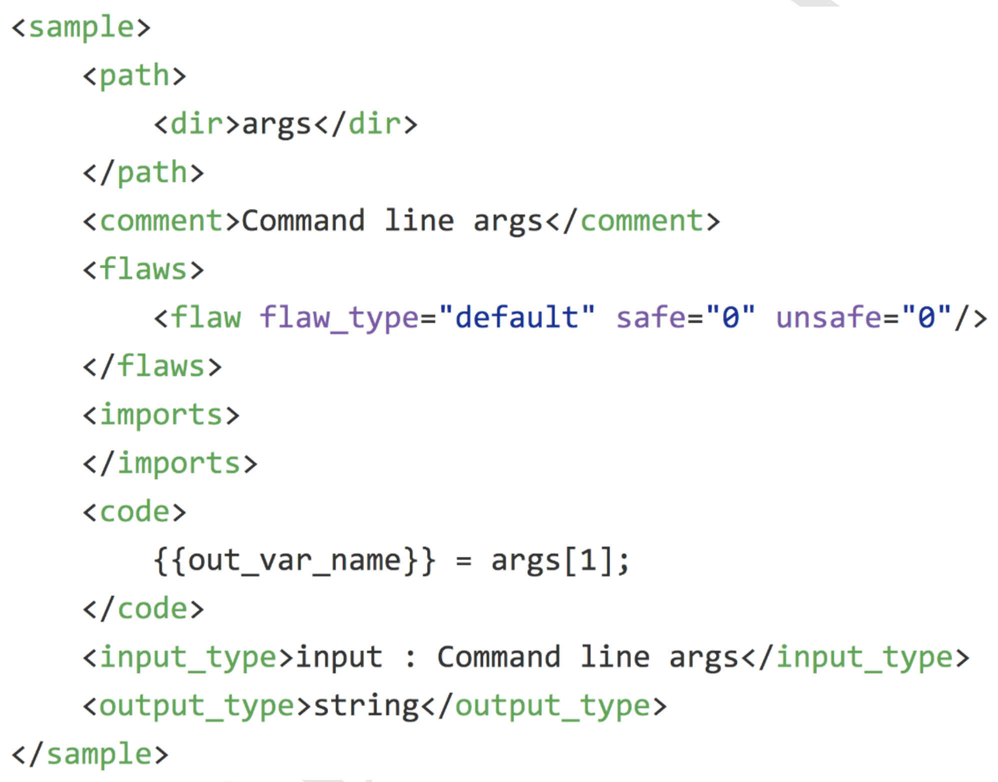
  - variable: Defines each variable type and how it will be used.
    * type: Names the type. This string does not appear in the test case code. It tells VTSG the type of variable that is being used. The input_type and output_type in Input, Filter, and Sink modules use this string.
    * code: A piece of code declaring the type of the variable. For some languages, such as PHP and Python, this field can be blank. This value takes the variable type when being declared (Ex: `string myString;`). In this case, ”string” is the value stored in this attribute.
    * init: Value assigned at the initialization of the variable. This value is used when declaring all global variables in the code.

- code: the template code. It should contain the following placeholders:
  - comments: This is replaced by the comments from Input, Filtering, Sink, and ExecQuery modules. This is intended to describe the variants, options, and use of this test case.
  - license: This is replaced by the contents of the `file_rights.txt` file. This is intended to hold authors’ names, usage and copy rights, contact information, etc.
  - stdlib_imports: This tag is a placeholder for *all* imports for the generated program
  - namespace_name: Used if the language requires it
  - main_name: Name of the main class
  - local_var: Location for local variables
  - input_content: Location for the Input
  - filtering_content: Location for the Complexity, along with the Filter
  - sink_content: Location for the Sink
  - exec_queries_content: Location for the ExecQuery
  - static_methods: Location for the static functions.

4.2 Input Modules

All input modules for a language are in that language’s `input.xml` file.

```
<sample>
    <path>
        <dir></dir>
    </path>
    <comment></comment>
    <flaws>
        <flaw flaw_type="" safe=""/>
    </flaws>
    <imports>
        <import></import>
    </imports>
    <code></code>
    <input_type></input_type>
    <output_type></output_type>
</sample>
```



```
<sample>
    <path>
        <dir>args</dir>
    </path>
    <comment>Command line args</comment>
    <flaws>
        <flaw flaw_type="default" safe="0" unsafe="0"/>
    </flaws>
    <imports>
    </imports>
    <code>
        {{out_var_name}} = args[1];
    </code>
    <input_type>input : Command line args</input_type>
    <output_type>string</output_type>
</sample>
```

Fig. 2. Example Input module. Instantiated at line 14 of Fig. 9.

- path: used in the file name. Each ⟨dir⟩ tag contains key words of the input method used. It is included in the file name. For example, when the key word in the ⟨dir⟩ tag is ”none”, the file name will contain ...I_none_...., where ”I” is used to state the input selected

- flaw: the vulnerability categories where the sample can be used, e.g., if the input is compatible with a CWE and if it is safe for this CWE. For example,

`<flaw flaw_type="CWE_89" safe="0"/>` means that the input is for the CWE 89 and that it is not safe. If an input is generic and compatible with all CWEs, put "default" in the "flaw_type" attribute.

- input_type: the type of input and its source. Declarations of `variable` in the File Template give available types, see Sec. 4.1.
- output_type: the type of output. The variable generated with the placeholder `{{out_var_name}}` in the code will be that type.
- code: The source code of an input. It should contain the placeholder `{{out_var_name}}`. That placeholder will be replaced by the variable name used in the Filtering and Sink. Do not declare this variable.

The case generated from the example Input in Fig. 2 takes an argument from the command line as Input. The input string can be either safe or unsafe, depending on user input.

4.3 Filter Modules

All filter modules are in the `filtering.xml` file.

```
<sample>
    <path>
    <dir></dir>
    </path>
    <comment></comment>
    <flaws>
    <flaw flaw_type="" safe=""/>
    </flaws>
    <imports></imports>
    <code></code>
    <input_type></input_type>
    <output_type></output_type>
</sample>
```

- path: used in the file name. Each ⟨dir⟩ tag has key words of the filtering method used. It is used in the filename.
- input_type: the input type of the filter. The variable generated with the placeholder `{{in_var_name}}` will be that type. Declarations of `variable` in the File Template give available types, see Sec. 4.1.

```xml
<sample>
    <path>
        <dir>func_preg_match</dir>
        <dir>only_numbers</dir>
    </path>
    <comment>filtering : check if there is only numbers</comment>
    <flaws>
        <flaw flaw_type="CWE_89" safe="1" unsafe="0"/>
        <flaw flaw_type="CWE_78" safe="0" unsafe="0"/>
        <flaw flaw_type="CWE_91" safe="0" unsafe="0"/>
        <flaw flaw_type="CWE_90" safe="0" unsafe="0"/>
    </flaws>
    <imports>
        <import>System.Text.RegularExpressions</import>
    </imports>
    <code>
        string pattern = @"/^[0-9]*$/";
        Regex r = new Regex(pattern);
        Match m = r.Match({{in_var_name}});
        if(!m.Success){
            {{out_var_name}} = "";
        }else{
            {{out_var_name}} = {{in_var_name}};
        }
    </code>
    <input_type>string</input_type>
    <output_type>string</output_type>
</sample>
```

Fig. 3. Example Filter module. Instantiated in lines 19–27 of Fig. 10.

- output_type: the output type of the filter. The variable generated with the placeholder `{{out_var_name}}` will be that type.
  Tip: To generate a test without Filtering, assign `in_var_name` to `out_var_name` and make the input_type and output_type `nofilter`. This passes the variable from the Input directly to the Sink.

- flaw: whether the Filter is compatible with a CWE and if it is safe for this CWE. For example,
  `<flaw flaw_type="CWE_89" safe="0"/>` means that the Filter is for CWE 89 and that it is not safe. Whereas `safe="1"` means that it is safe. It could be safe for another CWE. If a filter is compatible with all CWEs, put `ALL` in the "flaw_type" attribute.

- code: The source code of an filter. It should contain the placeholders `{{in_var_name}}` and `{{out_var_name}}`. Those placeholders will be replaced by the variable names used in the Input and Sink. Do not declare these variables.

The example Filter file in Fig. 3 makes sure the Input contains only a number. The flag safe is 1, because you cannot cause an SQL Injection (CWE 89) with only numbers.

4.4 Sink Modules

All sink modules are in the language's `sink.xml` file.

```
<sample>
    <path>
    <dir></dir>
    </path>
    <flaw_type flaw_group=""></flaw_type>
    <safety safe="" unsafe=""/>
    <comment></comment>
    <imports>
        <import></import>
    </imports>
    <code></code>
    <input_type></input_type>
    <exec_type></exec_type>
</sample>
```

- flaw_type: the flaw_group is a general category of vulnerability. Generated test cases are placed under the flaw group subdirectory, then in the flaw type subdirectory under that. The user can limit cases generated to certain flaw groups with the `-f` command line option or certain flaw types with the `-c` option.

- input_type: the input type of the sink. The variable generated with the placeholder `{{in_var_name}}` will be that type. If the sink does not require an input, this type should be `none`. The code should not contain the placeholder `{{in_var_name}}`. Declarations of `variable` in the File Template give available types, see Sec. 4.1.

- exec_type: link a sink to the exec queries. It must have the type of an ExecQuery. If it does not require an ExecQuery, exec_type should be `none`.

- safety: whether the sink is safe or not. If this sink is always "safe", assign a value of 1. If always "unsafe", assign a value of 1, so the file will be considered unsafe. This is useful for a deprecated function. If the Sink may be safe or unsafe depending on the input and filtering, assign a value of 0 to both attributes. The safety of the generated case will depend on the Input and Filtering.

- code: The source code of a sink. It should contain the placeholder `{{in_var_name}}`. The placeholder will be replaced by the variable name used in the Filter. Do not declare this variable.

  The placeholder `{{flaw}}` indicates that the next line is the location of the flaw. In other words, if this case is unsafe, the manifest reports a flaw at the number of the line following this. In generated unsafe cases, `{{flaw}}` is replaced with the one-line comment string, see Sec. 4.1, and "flaw". It does not appear in generated safe cases. Note: the generated code shown in Fig. 9 is incorrect: it does not show the `//flaw` line that would be generated.

The Sink example in Fig. 4 concatenates the filtered string with a SQL query. This block of code can only be used for SQL Injection. Whether or not it is vulnerable depends on the input string.

4.5 Exec_Query Modules

All query execution(??) modules are in the language's `exec_query.xml` file.

```
<exec_query type="" safe="">
    <path>
    <dir></dir>
    </path>
    <comment></comment>
    <imports>
        <import></import>
    </imports>
    <code></code>
</exec_query>
```

```
<sample>
    <path>
        <dir>select_from</dir>
        <dir>concatenation_simple_quote</dir>
    </path>
    <flaw_type flaw_group="A1">CWE_89</flaw_type>
    <comment>construction : concatenation with simple quote</comment>
    <imports></imports>
    <code>
        {{flaw}}
        string query = "SELECT * FROM '" + {{in_var_name}} + "'";
    </code>
    <safety safe="0" unsafe="0"/>
    <input_type>string</input_type>
    <exec_type>SQL</exec_type>
</sample>
```

Fig. 4. Example Sink module. Instantiated at line 23 of Fig. 9.

- type: the type of the ExecQuery. This type is used in the `exec_type` tag of the Sink in order to link them together during generation process. The type should only contain letters, numerals, and underscore ("_").
  VTSG V3 supports many database management systems, including ORACLE, MySQL, MSSQL, Postgre, SQLite, and XPATH. The syntax of each ExexQuery must be compatible with its associated database system language.
- safe: whether the ExecQuery makes the case safe or not.
- code: The source code of a query. It does not contain placeholders. It should be linked to the corresponding variable from the Sink. The linking is done through the "exec_type" attributes within the XML files.

The block of code in the Exec_Query example, Fig. 5, executes the SQL query, used for database management systems, including MySQL, Oracle, Postgre, and SQLite. This example is vulnerable. If a non-vulnerable execution of a SQL query is required, use a SQL prepared statement.

4.6 Test Condition and Code Complexity Modules

All test condition and code complexity modules are in the language's `complexities.xml` file. This file has a `<root>` with one `<conditions>` and one `<complexities>`. All con-

```
<exec_query type="SQL"  safe="0">
    <path>
        <dir>sql_server</dir>
    </path>
    <comment></comment>
    <imports>
        <import>System.Data.SqlClient</import>
    </imports>
    <code>
        string connectionString =  @"server=localhost;uid=sql_user;password=sql_password;database=dbname";
        SqlConnection dbConnection = null;
        try {
            dbConnection = new SqlConnection(connectionString);
            dbConnection.Open();
            SqlCommand cmd = dbConnection.CreateCommand();
            cmd.CommandText = query;
            SqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read()){
                Console.WriteLine(reader.ToString());
            }
            dbConnection.Close();
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    </code>
</exec_query>
```

Fig. 5. Example Exec_Query module. Instantiated in lines 25–39 of Fig. 9.

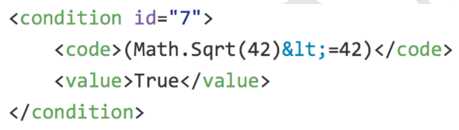
dition modules are inside `<conditions>`. All complexity modules are inside `<complexities>`.

```
<root>
    <conditions>
        <condition ...>
            ...
        </condition>
        ....
    </conditions>
    <complexities>
        <complexity ...>
            ...
        </complexity>
        ....
    <complexities>
```

4.6.1 Test Condition Modules

```
<condition id="">
    <code></code>
    <value></value>
</condition>
```

- id: Unique string(?) for the Condition. Appears in the test case file name.
- code: the source code of the conditional test.
- value: either `<value>True</value>` or `<value>False</value>` depending on whether the code evaluates to true or false.



```
<condition id="7">
    <code>(Math.Sqrt(42)&lt;=42)</code>
    <value>True</value>
</condition>
```

Fig. 6. Example test condition module. Instantiated in Fig. 9, line 16.

4.6.2 Code Complexity Modules

```
<complexity id="" type="" group="" executed="" in_out_var="i"
                                        indirection="" need_id="">
    <code></code>
<body></body>
</complexity>
```

- id: Unique string(?) for the Complexity. This is at the end of the name of the generated file.
- type: Supported types are: `if`, `switch`, `goto`, `for`, `foreach`, `while`, `function`, and `class`.
- group: Supported groups are: `conditionals`, `jumps`, `loops`, `functions`, and `classes`.
- executed: whether the placeholder will be executed or not. Four values are allowed:
  - 0: Not executed
  - 1: Executed
  - condition: Executed if the condition is true
  - not condition: Executed if the condition is false
- in_out_var: whether the variable (from the Input) will be used or transformed in the Complexity before being used in the Filter. If the variable is neither used nor transformed, do not use this attribute. Three values are allowed:
  - in: the variable is used before the placeholder
  - out: the variable is used after the placeholder
  - traversal: the variable is used in the placeholder

  If this attribute is used, the code should contain the following placeholders: `{{in_var_name}}`, `{{out_var_name}}`, and `{{var_type}}`.
- indirection: "1" if the code is split into two chunks (call and declaration) or calls a function. The body tag should be present when calling a function.
- need_id: "1" if the code has a placeholder, `{{id}}`, to generate a unique ID for the Complexity. This ID to generate a label, a parameter, or a function name in a nested context.
- code: the source code of the Complexity. Code or body should contain the placeholder, `{{placeholder}}`, where the Filter is inserted. It may contain the placeholder, `{{condition}}`, where the Condition is inserted.
- body: Contains the source code not in the main execution flow, e.g., functions or classes. (optional)

When adding Complexity, VTSG can use several complexity in one test case. The example in Sec. 6 has two types of Complexity: a control flow complexity and a data flow complexity. The control flow complexity specification is in Fig. 7. It is instantiated in lines 16–21 of Fig. 9. Line 16 is the instantiation of the control flow condition specified in Fig. 6.

```
<complexity id="11" type="while" group="loops" executed="condition">
    <code>
        while({{ condition }}){
            {{ placeholder }}
            break;
        }
    </code>
</complexity>
```

Fig. 7. Example Complexity module with a while loop. Instantiated in Fig. 9, lines 16–21.

```
<complexity id="20" type="class" group="classes" executed="1" in_out_var="traversal" need_id="1" indirection="1">
    <code>
        {{call_name}} var_{{id}} = new {{call_name}}({{in_var_name}});
        {{out_var_name}} = var_{{id}}.get_var_{{id}}();
    </code>
    <body>
        /*
        {{comments}}
        */
        /*
        {{license}}
        */
        {{ imports }}
        namespace default_namespace{
            class {{call_name}}{

                {{in_var_type}} var_{{id}};

                public {{call_name}}({{in_var_type}} {{in_var_name}}_{{id}}){
                    var_{{id}} = {{in_var_name}}_{{id}};
                }

                public {{out_var_type}} get_var_{{id}}(){
                    {{local_var}}
                    {{in_var_name}} = var_{{id}};
                    {{ placeholder}}
                    return {{out_var_name}};
                }

                {{static_methods}}
            }
        }
    </body>
</complexity>
```

Fig. 8. Example Complexity module with a method invocation. The ⟨code⟩ part is instantiated in Fig. 9, lines 18 and 19. The ⟨body⟩ part is instantiated in Fig. 10.

The data flow complexity is a method call within the `while` loop. The specification is in Fig. 8. The ⟨code⟩ part is instantiated in lines 18 and 19 of Fig. 9. The ⟨body⟩ part is instantiated in Fig. 10.

5. Generated Test Cases

This section describes where VTSG places test cases and what the names of test case files mean.

5.1 Directory Structure of Generated Test Cases

a description of the directory structure where test cases are placed.

5.2 Test Case File Names

VTSG names test case files as `cwe_NB__I_INPUT__F_FILTER__S_SINK__EQ_EXEC_QUERY__NBCPLX-CPLX1-CPLX2.COND_FileX.EXT`

- NB: CWE number (two or three digits)
- INPUT: Input name (optional)
- FILTER: Filter name (optional)
- SINK: Name of the critical function
- EXEC_QUERY: ExecQuery name (optional)
- NBCPLX: The number of complexities. Each complexity has the tags
  - CPLX1, CPLX2, . . . : ID given in code complexity modules, see Sec. 4.6.2. Table 3 in the appendix lists complexitie IDs currently used with PHP or C#. (optional)
  - COND: ID given in test condition modules, see Sec. 4.6.1. Table 4 lists condition IDs currently used. (optional)
- X: Number of the file within the test. 1 is the main file. 2, 3, . . . are other files, such as classes. Juliet uses a, b, c, d
- EXT: file extension

Example `cwe_89__I_args__F_func_preg_match-only_numbers__S_select_from-concatenation_simple_quote__EQ_mysql__1-2.4_File1.cs`

The Complexity part near the end, 1-2.4, means there is one Complexity. It is complexity 2 with condition 4.

File names reflect the entire case, not just the code in a particular file. The example in Sec. 6 shows that both files of the case have identical names, except for the final FileX part.

6. Example Test Case

This section has an example test case. The case consists of two files. Fig. 9 is the main file of the example. Fig. 10 is an auxiliary class file. The code in the main file invokes the class at line 18.

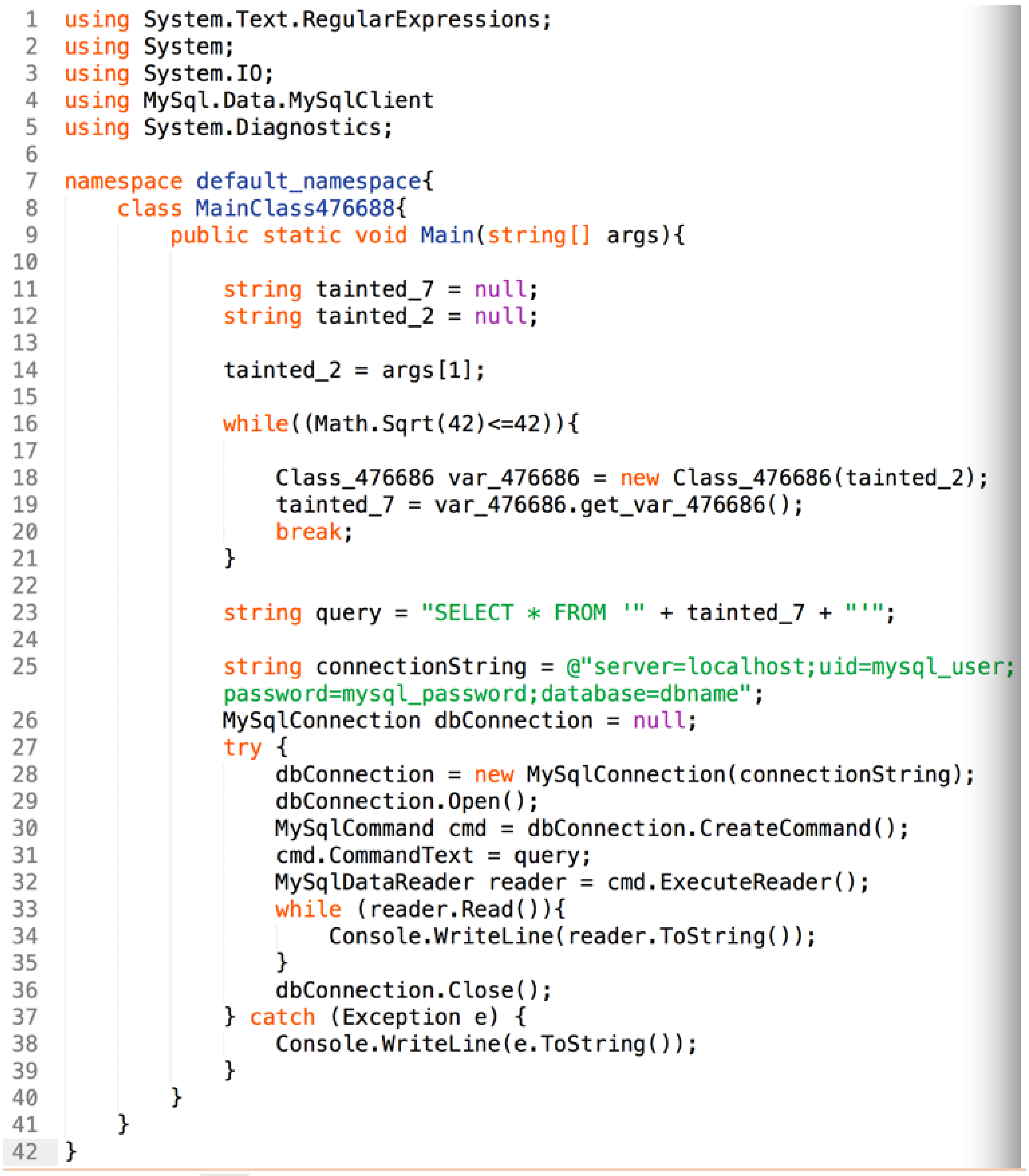
The name of the main file is `cwe_89__I_shell_commands__F_func_preg_match-only_numbers__S_select_from-concatenation_simple_quote__sql_server__2-11.7-20_File1.cs`. The name of the class file, Fig. 10, is identical, except for the file number, "2".

Using Sec. 5.2, Table 3, and Table 4, the file name can be understood as follows:
CWE 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')[5]
Input shell_commands, see specification in Fig. 2
Filter func_preg_match-only_numbers, Fig. 3
Sink select_from-concatenation_simple_quote__sql_server, Fig. 4.
Complexities 2-11.7-20 two complexities; 11 = `while`, Fig. 7, with condition 7 = `Math.Sqrt(42)<=42`: True, Fig. 6; and 20 = class traversal executed, Fig. 8.
File1 main file
The extension, cs, shows that it is a C# file.

7. Acknowledgements

Bertrand Stivalet and Aurelien Delaitre, from the NIST SAMATE team, designed the architecture of the VTSG project and managed its implementation. The project was implemented by students from TELECOM Nancy: Jean-Philippe Eisenbarth, Valentin Giannini, and Vincent Noyalet. We thank Terry Cohen for her comments and suggestions, which improved this report. We also thank Elizabeth Fong and Charles de Oliveira for their contributions to this report.

[1] Stivalet B, Fong E (2016) Large scale generation of complex and faulty PHP test cases. *2016 IEEE Intern'l Conf. on Software Testing, Verification and Validation (ICST)*, , pp 409–415. https://doi.org/10.1109/ICST.2016.43. https://doi.org/10.1109/ICST.2016.43
[2] (2020) Static analysis results interchange format (SARIF) version 2.1.0, https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. Accessed: 8 February 2022.
[3] (2017) OWASP top 10 web application security risks, https://owasp.org/www-project-top-ten/. Accessed: 17 June 2020.
[4] (2020) Common weakness enumeration, https://cwe.mitre.org/. Accessed: 17 June 2020.
[5] (2020) CWE-89: Improper neutralization of special elements used in an SQL command ('SQL injection'), https://cwe.mitre.org/data/definitions/89.html. Accessed: 18 June 2020.



```
 1 using System.Text.RegularExpressions;
 2 using System;
 3 using System.IO;
 4 using MySql.Data.MySqlClient
 5 using System.Diagnostics;
 6
 7 namespace default_namespace{
 8     class MainClass476688{
 9         public static void Main(string[] args){
10
11             string tainted_7 = null;
12             string tainted_2 = null;
13
14             tainted_2 = args[1];
15
16             while((Math.Sqrt(42)<=42)){
17
18                 Class_476686 var_476686 = new Class_476686(tainted_2);
19                 tainted_7 = var_476686.get_var_476686();
20                 break;
21             }
22
23             string query = "SELECT * FROM '" + tainted_7 + "'";
24
25             string connectionString = @"server=localhost;uid=mysql_user;
               password=mysql_password;database=dbname";
26             MySqlConnection dbConnection = null;
27             try {
28                 dbConnection = new MySqlConnection(connectionString);
29                 dbConnection.Open();
30                 MySqlCommand cmd = dbConnection.CreateCommand();
31                 cmd.CommandText = query;
32                 MySqlDataReader reader = cmd.ExecuteReader();
33                 while (reader.Read()){
34                     Console.WriteLine(reader.ToString());
35                 }
36                 dbConnection.Close();
37             } catch (Exception e) {
38                 Console.WriteLine(e.ToString());
39             }
40         }
41     }
42 }
```

Fig. 9. Main file of example. Line 14 instantiates input code from Fig. 2. Lines 16–19 instantiates complexity code from Fig. 7. Line 16 instantiates condition code from Fig. 6. Lines 18 and 19 instantiate code from the ⟨code⟩ part of Fig. 8. Line 23 instantiates critical preparation code from Fig. 4. Lines 25–39 instantiate query execution code from Fig. 5. Note: a //flaw line would be generated before line 23.

```
1  using System.Text.RegularExpressions;
2
3  namespace default_namespace{
4      class Class_476686{
5
6          string var_476686;
7
8          public Class_476686(string tainted_4_476686){
9              var_476686 = tainted_4_476686;
10         }
11
12
13         public string get_var_476686(){
14             string tainted_4 = null;
15             string tainted_5 = null;
16
17             tainted_4 = var_476686;
18
19             string pattern = @"/^[0-9]*$/";
20             Regex r = new Regex(pattern);
21             Match m = r.Match(tainted_4);
22
23             if(!m.Success){
24                 tainted_5 = "";
25             }else{
26                 tainted_5 = tainted_4;
27             }
28
29             return tainted_5;
30         }
31     }
32 }
```

Fig. 10. Auxiliary file of example. It instantiates code from the ⟨body⟩ part of Fig. 8. Lines 19–27 instantiate filter code from Fig. 3.

A. Complexity and Condition IDs

This section lists complexities and conditions currently available for C# and PHP. We explain code complexities in Sec. 3.4.

ID	Complexity
1	if condition
2	if condition
3	if not condition
4	if condition
5	if not condition
6	if condition
7	if not condition
8	if not executed
9	switch executed
10	switch not executed
11	while condition
12	do while executed
13	for executed
14	foreach executed
15	goto not executed
16	goto executed
17	function traversal executed
18	function in executed
19	function out executed
20	class traversal executed
21	class in executed
22	class out executed

Table 3. Complexity IDs Defined for C# and PHP

Why are they four “if condition”s? (and three “if not condition”s)

What does “condition” mean vs. “executed”?

Explain connection between table and code and body.

Complexities are defined in Code Complexity Modules, see Sec. 4.6.2.

ID	Code	Value
1	`1==1`	True
2	`1==0`	False
3	`4+2<=42`	True
4	`4+2>=42`	False
5	`Math.Pow(4, 2)<=42`	True
6	`Math.Pow(4, 2)>=42`	False
7	`Math.Sqrt(42)<=42`	True
8	`Math.Sqrt(42)>=42`	False

Table 4. Condition IDs Associated with Complexities in Table 3

The value indicates that the code always evaluates to true or false, respectively. Conditions are defined in Test Condition Modules, see Sec. 4.6.1.

Can any condition go with any complexity?