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

NIST
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

March 2023

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, 49 pages (March 2023)

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, adding new complexities or flaws or vulnerability, or 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

List of Tables

List of Figures

1. Introduction

The Vulnerability Test Suite Generator (VTSG) generates collections of vulnerable and non-vulnerable synthetic programs expressing specific flaws. The programs can be test cases to evaluate static analyzers. Each test case targets one flaw. There are two types of test cases:

- cases with flawed code (unsafe), leading to a vulnerability, and
- safe cases with similar behavior, but without the flaws, that is, no vulnerability.

Exactly corresponding vulnerable and non-vulnerable cases could, in theory, be generated in pairs. However, since each test case is generated separately, there is no exact correspondence between cases.

Figure 1 suggests that just three sources of input, two ways of filtering the input, four sinks, and two ways to execute the query already yields 3 × 2 × 4 x 2 = 48 combinations. Multiply that by many different kinds of bugs and all flow control complexities with various conditions, then include "scaffolding" such as initializing variables, importing libraries, and declaring functions to get an idea of the work that VTSG may save.

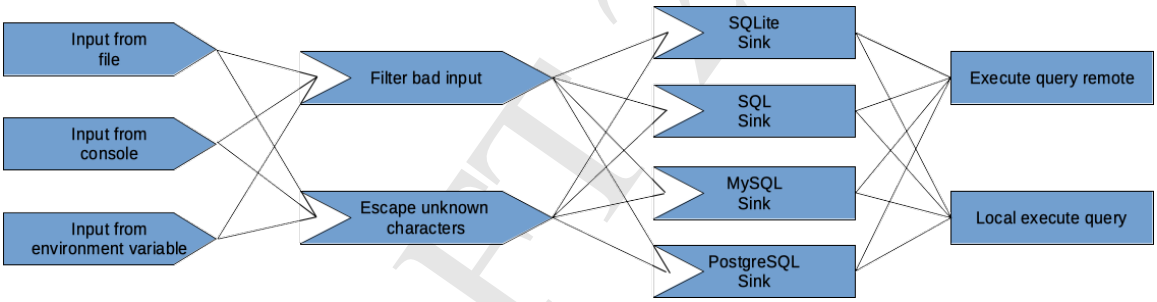

Fig. 1. Just three sources of input, two ways of filtering, four sinks, and two query executions yield 48 possible test cases.

NIST's Software Assurance Reference Dataset (SARD) has suites of paired test cases in the Juliet test suites for C/C++ and Java, which are available as SARD test suites 108 and 109. These and many other test suites are available at https://samate.nist.gov/SARD/testsuite.php.

The generator is written in Python 3. The VTSG git repository is at https://github.com/usnistgov/VTSG. The list of files and the README.md file are given in App. D.

1.1 History

Originally named Vulnerability Test Suite (VTS) generator, version 1 only generated C# programs. VTS version 2 generated PHP programs [1] in addition. Version 2 was more customizable to generate other programming languages. VTSG version 3 systematically maintains indentation, so also generates Python programs.

Readers can download the PHP and C# test cases generated by earlier versions as SARD test suites 103 and 105 from https://samate.nist.gov/SARD/testsuite.php.

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, first install the *Git* package. Here is the command to install it:

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

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

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

To download Python source code packages, install the *pip Python* package manager. Here is the command to install it:

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


One may also have to install the *pip Python 3* package manager. Here is the command to install it:

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

Another way to install *pip* is:

```
sudo python3 -m pip install --upgrade pip
```

To validate C# test cases, install *mono* and *mcs* to run them. 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).
Here is the command to install it:

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

Here is the command to install it:

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

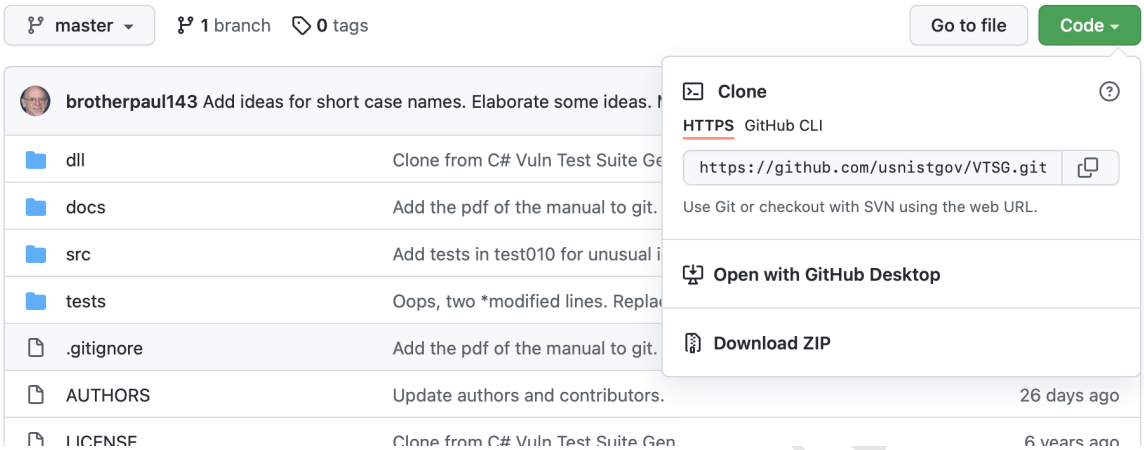

Fig. 2. usnistgov/VTSG: button to clone repository.

1.3 Install VTSG

To copy the generic VTSG from GitHub to a local Linux machine, change to a directory under which you want to install VTSG.
Looking at the GitHub website, one will see a green box labeled "Code". See Fig. 2. Click on it, then click on the "copy" icon to copy the web URL.
Here is the 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` command, one will see that the **VTSG** directory was created.
Go into that directory, using this command:

```
cd VTSG
```

To install the dependencies, use this 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 the command line interface and retrieve the appropriate sample from the generated and categorized folders.

The second targeted audience of the VTSG comprises people wishing to generate test cases using a programming language other than the languages currently supported. They must create new templates for the program using the XML tags, execute VTSG with the command line interface, and retrieve the samples from the generated and categorized file folders.

1.5 Vulnerabilities Currently Encoded in VTSG Files

Vulnerabilities are encoded in the language files, see Sec. 4. Some of the OWASP Top 10 [2] and Common Weakness Enumerations (CWEs) [3] are encoded. See App. A, B, and C for details.

2. Command Line Interface

For users who wish to generate PHP, C#, or Python 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 flaws or groups of flaws, for example, OWASP categories. The user must specify the programming language. The invocation command looks like this:

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

where `<options>` are listed in Table 1.

Table 1. Options for command line invocation

-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.
-g GROUP --group=GROUP	Only generate cases with vulnerabilities in the specified group(s). May be repeated. See Sec. 4.6.
-f Flaw --flaw=Flaw	Only generate cases with the specified flaw(s). May be repeated. See Sec. 4.6.
-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.
-n NUMBER --number-generated=NUMBER	Maximum number of sink, filter, input, and exec query combinations to generate. (Default: -1, meaning all) See below for explanation.
-t TEMPLATE_DIRECTORY --template-directory=TEMPLATE_DIRECTORY	The language templates directory. (Default: src/templates)
-d --debug	for programmer use

2.1 Explanation of Options

VTSG will generate all flaws in all flaw groups unless limited with command line options. VTSG will generate both the unsafe (buggy or vulnerable) test cases and the safe (not buggy) test cases if the user does not select either only safe (-s) or only unsafe (-u) cases. The options are mutually exclusive.

The -n (number-generated) option is not very useful. 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 yield 1 (no complexities) + 5 (each complexity, not nested) + 25 (each complexities nested within every complexity) = 31 test cases. Hence VTSG will generate

far more test cases than the number given with the -n option.

2.2 Example Invocations

Show the help message:

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

Generate all PHP test cases:

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

This takes about 25 minutes. (Generating all the C# cases takes about four minutes.)

Generate a C# (`-l cs`) test suite made of vulnerable (unsafe) test cases (`-u`) with SQL injection vulnerabilities (`--flaw=CWE_89`) and up to 3 nested levels of complexity (`-r 3`).

```
$ python3 vtsg.py -l cs -r 3 --flaw=CWE_89 -u
```

3. Overview of VTSG

Please note that this document describes two different program structures: the structure of VTSG itself and the structure of test cases that it generates.

3.1 Overview of Test Case Generation

Suppose you write an analyzer to look for bugs in programs. To test the analyzer, you need test cases. Test cases could be lots of many different programs with lots of different kinds of bugs in different places (and programs without bugs, too, to test false positives). It is tedious and error prone to construct these test cases manually. VTSG is written to construct huge numbers of test cases consistently.

In pseudocode, the structure of test cases that VTSG generates is

```
input = get_input_from_somewhere()
if is an injection attack then
    reject it (filter)
use the input to make a query (sink)
execute the query in SQL
```

You can see this structure in Fig. 3.

VTSG generates test cases from information in Template, Input, Filter, Sink, Complexity and ExecQuery files (see Fig. 3):

- Template is the overall structure of each program.
- Input is the source of untrusted data in the program, e.g., command line, variable, files, or form methods.

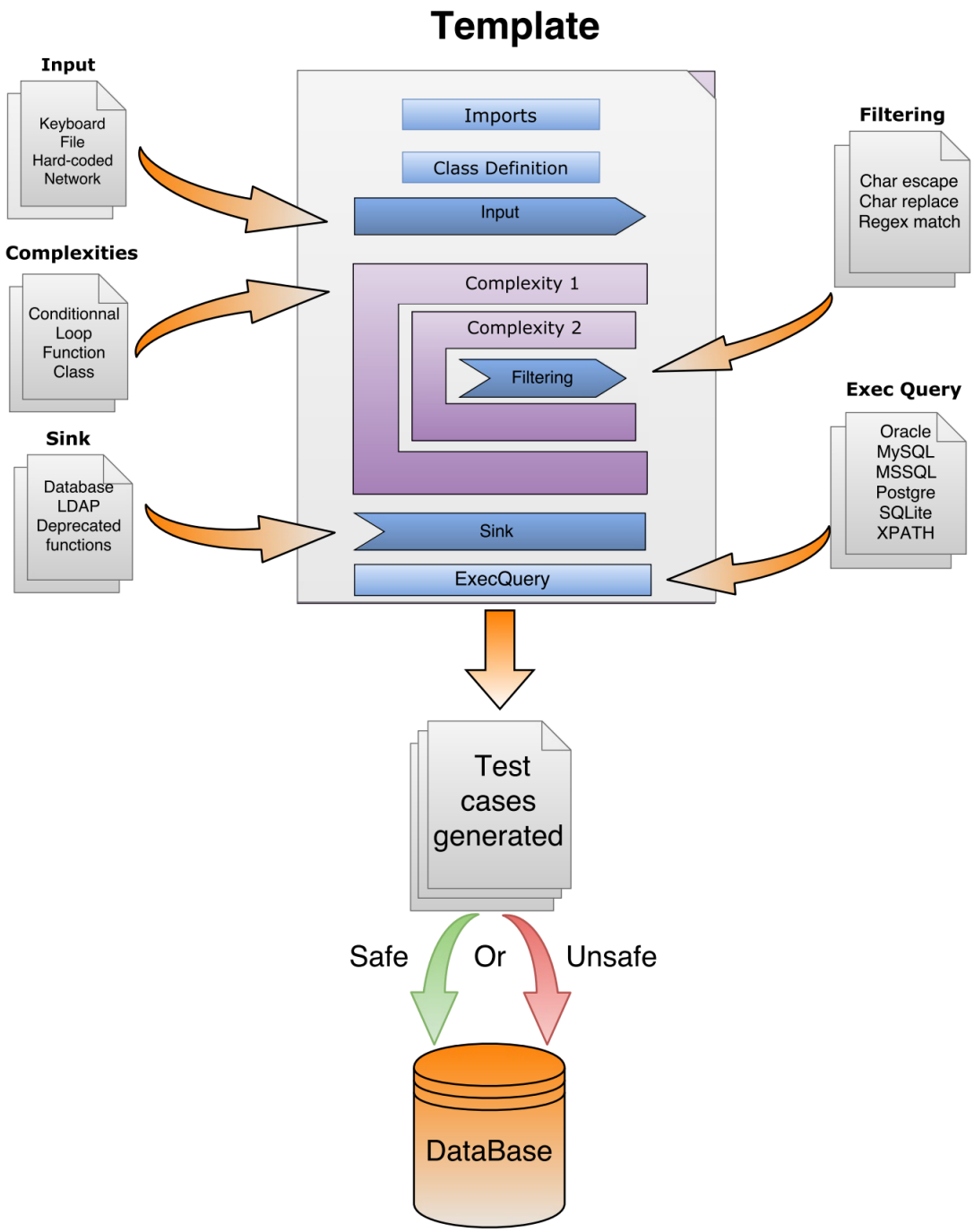

Fig. 3. Overview of VTSG test case generation process. The Template specifies how pieces are assembled. Input, Filter, Complexity, Sink, and ExecQuery modules provide alternative code. An example of a generated test case is in Figs. 14 and 15.

- Filter filters the input with functions or methods such as sanitization functions, casting, or deprecated functions.
- 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.
- Complexities are additional data flow or control flow complications that are worked into the structure of the program to exercise tools' abilities.

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

When invoked to generate test cases, VTSG creates a directory with all the results. The directory is named for the date and time created, for example, `TestSuite_03-08-2022_16h46m35`. The language directory, `PHP`, `Csharp`, or `Python` is created in this directory. This language name comes from the `name` in the `file_template.xml` file. See Sec. 4.2.

Under the language directory, VTSG creates one directory for each flaw group, for instance, `OWASP_a1` or `OWASP_a4`. These come from the `flaw_group` in the `sinks.xml` file; see Sec. 4.6. Under each flaw group directory is a subdirectory for each specific flaw, for instance, `CWE_78` or `CWE_89`. These come from the `flaw_type` entries, which are also in the `sinks.xml` file. VTSG also creates a manifest file of all the test cases generated for each flaw group, named `manifest.xml`.

If the `flaw_group` is missing or empty, subdirectories for flaw types are created immediately under the language directory.

Under the flaw directory are directories for safe or unsafe (vulnerable) test cases, depending on which are generated.

3.2 VTSG Directory Structure

All the language information files are accessed from a single subdirectory. The default is `src/templates`. Another directory may be given with the -t command line option, see Table 1.

Under this is one subdirectory for each language. VTSG chooses the subdirectory based on the language given in the `-l` command line option. Each language subdirectory has six files for that language. The files are `file_template.xml`, `inputs.xml`, `complexities.xml`, `filters.xml`, `sinks.xml`, and `exec_queries.xml`. The files are described in Sec. 4. To add another language, simply create a subdirectory for that language with the six description files.

The `src/templates` directory has `file_rights.txt`, which is copied into each generated test case to declare license rights and authorship, see Sec. 4.2, and a `dtd` subdirectory, which has document type definitions (DTDs) for all of the XML files.

3.3 Details of Test Case Generation

Each test case is constructed based on the file template, as shown in Fig. 3. Test cases are programs in a specific programming 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 and a Sink. The Filter code may be embedded in data and control flow Complexity code.

VTSG generates test cases with two broad steps. First, VTSG selects sink, filter, input, exec query, and complexities that are compatible with each other and consistent with any flaw group or flaw constraints the user gives on the command line. Second, VTSG composes test case source code from the selected modules, synthesizing variable and function 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. 4.6. 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, which are in `generator.py`:

```
for each sink:
    if this sink is the type specified:
        use this sink
        if input is needed:
            select_filtering()
        else:
            select_exec_queries()

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

def select_input():
    for each input:
        if input is compatible with filter and sink:
            use this input
```

```
            select_exec_queries()

def select_exec_queries():
    if sink needs exec_query:
        for each exec_query:
            if exec_query is compatible with sink:
                use this exec_query
                recursion_or_compose()
    else:
        recursion_or_compose()

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

... and so forth
```

The *vtsg.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 a compatible filter. It then goes through all inputs. The `<input_type>` and `<output_type>` must be consistent with the "Filter" and "Sink" XML tags. Then an exec query and complexities are selected that are compatible with the currently selected sink module.

When VTSG has selected a set of modules, it composes their code 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 "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 chosen for the file is described in Sec. 3.1. Section 6 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 flaws specified in the invocation. If VTSG is invoked with particular flaw groups (`-g`) or flaws (`-f`), only sinks satisfying those specified are used. If no flaw group is specified, all flaw groups are used. If no flaws are specified, all flaws 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 much 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. Tracing execution through structures like `while` loops, `if` statements, and function calls may confuse static analysis tools. We refer to these constructs as code complexities. VTSG can nest code complexities to create slightly more realistic source code.

The complexities currently defined are listed in the appendixes for their languages.

Here is an example of code complexities from `cwe_89__I_shell_commands__F_no_filtering__S_select_from-concatenation_simple_quote__EQ_mysql__3-2.5-9-21a.cs`:

```
if((Math.Pow(4, 2)<=42)){
    switch(6){
        case(6):
            Class_8 var_8 = new Class_8(tainted_5);
            tainted_6 = var_8.get_var_8();
            tainted_7 = tainted_6;
            break;
        default:
            break;
    }
}else{
    {}
}
```

The above has three nested control structures:

Level 1 is the “if” statement.

Level 2 is the “switch” statement.

Level 3 is the call of a method from a Class.

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

These XML files are required for each language. 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.

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

Character	Replacement
<	<
>	>
"	"
’	'
&	&

VTSG uses Jinja to compose code. In addition to the above, Jinja recognizes double-curly-brackets ({{ and }}) as introducing Jinja-specific variables and controls. Do not use pairs of curly brackets in your files, except for VTSG-related variables.

Characteristics of modules and their information are stored in XML files. This section of the document 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.

Each language directory has one file of each name. That is, one `file_template.xml` file, one `inputs.xml` file, one `filters.xml` file, one `complexities.xml` file, one `sinks.xml` file, and one `exec_queries.xml` file.

All the files, except `file_template.xml`, may have many modules, that is alternate chucks of code, in them. For example, `inputs.xml`, Sec. 4.4, typically has many input modules. Each module in `inputs.xml` provides a method to get input from a different source, such as command line options or hard-coded values.

4.1 Maintain Indentation with INDENT ... DEDENT

VTSG using Jinja does a haphazard job of producing proper indentation. Indentation does not matter for many languages. It is critical for Python, however. This section explains how to use INDENT and DEDENT to ensure correct indentation in the final source code.

4.1.1 Using INDENT ... DEDENT

INDENT ... DEDENT sections may appear in any of the above files. They most often occur in `file_template.xml` and `complexities.xml` files.

Use INDENT and DEDENT lines in code chunks to indicate that any code between those lines should be indented consistently. For example,

```
def main():
INDENT
    {{local_var}}
    {{input_content}}
    {{filtering_content}}
    {{sink_content}}
    {{exec_queries_content}}
DEDENT
```

All code produced from the statements between the INDENT/DEDENT lines is consistently indented with the string defined in `<indent> </indent>`, which appears in `file_template.xml`, see Sec. 4.2. That string is typically four spaces.

INDENT sections may be nested. For example, here is a sink module with code that needs additional indentation.

```
        print(f'file "{ {{in_var_name}} }" ', end='')
        {{flaw}}
        if os.path.exists({{in_var_name}}):
INDENT
            print('exists')
DEDENT
        else:
INDENT
            print('does not exist')
DEDENT
```


If the INDENT lines were not included, VTSG produces the following code (slightly edited for presentation).

```
def main():
    tainted_0 = input()
    tainted_1 = tainted_0

        # No filter (sanitization)
        tainted_1 = tainted_0


        print(f'file "{ tainted_1 }" ', end='')
        #flaw
        if os.path.exists(tainted_1):
            print('exists')
else:
            print('does not exist')
```

Notice that the indentation is not consistent. This is not valid Python code. With INDENT lines, VTSG produces the following, which is valid Python.

```
def main():
    tainted_0 = input()
    tainted_1 = tainted_0

    # No filter (sanitization)
    tainted_1 = tainted_0


    print(f'file "{ tainted_1 }" ', end='')
    #flaw
    if os.path.exists(tainted_1):
        print('exists')
    else:
        print('does not exist')
```


4.1.2 Details of INDENT ... DEDENT

Here are details of using INDENT and DEDENT.

VTSG processes code within INDENT sections line by line. No semantic parsing or analysis is done.

A section to be fixed is indicated by a line beginning with `INDENT`, possibly with leading whitespace. The end of the section is indicated by a line beginning with `DEDENT`, again possibly with leading whitespace. Any text after INDENT or DEDENT to the end of the line is ignored.

Indent sections may be nested.

INDENT and DEDENT lines are removed. For lines within an INDENT ... DEDENT section,

- first, any leading whitespace is removed, and
- second, if the line is not empty, indentation is added for each nested INDENT ... DEDENT section this is in.

Here is a convoluted example to illustrate the fine points. Suppose this is the code generated by composing the modules.

```
      if Condition:
INDENT            text after INDENT is ignored
           line 1
               while not True:
          INDENT
           line 3 - INDENT not at the beginning is ignored
     DEDENT
```

```
                line above is empty
        DEDENT
        line 5
```

If the indent string is specified as `<indent>..,</indent>`, the following is the result. (Note: typically, the indent is four spaces. The preceding string with periods and a comma is only for example clarity.)

```
      if Condition:
..,line 1
..,while not True:
..,..,line 3 - INDENT not at the beginning is ignored

..,line above is empty
        line 5
```

Note: because *all* leading whitespace is removed from lines in indent sections, using INDENT ... DEDENT anywhere means that every indentation must be indicated with INDENT ... DEDENT lines.

We chose "DEDENT" because it is used in Python's grammar description.

4.2 File Template

```
<template type="" name="">
    <file_extension></file_extension>
    <comment>
        <open></open>
        <close></close>
        <inline></inline>
    </comment>
    <syntax>
        <statement_terminator></statement_terminator>
        <indent></indent>
    </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 name of 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>`.
  - indent: string used to indent code, see Sec. 4.1. This is typically four spaces, but can be any string.
- namespace: Namespace name, if applicable
- variables: Information about variable names and types and how to include libraries.
  - prefix: Any prefix required for variable names. $ for PHP. Leave it empty 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 to initialize it. (optional)
    - 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 to declare the type of the variable. For some languages, such as PHP and Python, this field can be blank. This value gives the variable type when being declared, for example, `string var_0;`. In this case, "string" is the value put in this attribute.
    - init: Value assigned when this type of variable is initialized.

    If variables do not need to be declared in this language, do not include any `<variable ... />` statements or the `{{local_var}}` placeholder in the code.
- code: the template code. It should contain the following placeholders:

- comments: This is replaced by comments in the selected input, filter, sink, and exec query 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 copyrights, contact information, etc.
- stdlib_imports: This 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 to declare local variables (optional). If variables do not need to be declared in this language, do not include this placeholder or any `<variable ... />` statements in the variables.
- input_content: Location for the Input (required)
- filtering_content: Location of the Filter, which will be wrapped with complexities, if any.
- sink_content: Location for the Sink
- exec_queries_content: Location for the ExecQuery
- static_methods: Location for the static functions.

4.3 Attributes Shared By Modules

Many kinds of modules use the same attributes. Instead of repeating explanation of these attributes, they are here.

4.3.1 Module Description in Path and Dir Tags

Within the `<path>` keywords, modules may have one or more `<dir>` tags. These tags provide the descriptions of the module that is used in the file name, see Sec. 6. For example, when the key word in a selected input module is "file", the file name will contain `..._I_file_...`, where "I" indicates the input module selected.

If a module has more than one `<dir>` tag, the strings are joined with dashes. For example, if a sink has

```
<path>
    <dir>select_from</dir>
    <dir>concatenation_simple_quote</dir>
</path>
```

then cases using that module will have file names containing `_S_select_from-concatenation_simple_quote_`.

Note: we cannot think of any reason why it is better to give multiple description strings instead of just one string. But the functionality is provided in VTSG, and some modules use it.

4.3.2 Module Comment

If a sample module has a comment string, it is added to the `{{comments}}` area given in the file template, Sec. 4.2. This informs the user about the purpose or structure of the input, filter, sink, and exec query modules included. Below is an example comment string.

```
<comment>sink: check if a file exists</comment>
```

Any case using that module will have

```
sink: check if a file exists
```

in the comments area.

4.3.3 Needed Imports

Sometimes the use of code requires some library to be imported or used. This is indicated with names given in `<import></import>` directives within `<imports></imports>` sections.

Code statements are synthesized from the `import_code` in the file template and the name or names given here.

4.3.4 Marking Modules as Safe and Unsafe

Using some modules in a program for certain flaws may make them safe or may make them unsafe. For instance, prepared SQL statements are always safe from SQL injection vulnerabilities. In contrast using a broken cryptographic algorithm is always unsafe, regardless of how any user input is filtered. Similarly certain hard-coded inputs may always make a program safe from certain flaws, and some filters may make a program safe from certain flaws for any user input.

Input, filter, and sink modules can be marked as always safe or always unsafe using `safe="1"` or `unsafe="1"`. Modules may be always safe or always unsafe (or neither) for some flaws and have different safety attributes for other flaws.

Exec query modules may be marked as always safe. (No exec query module can make the program unsafe.)

A generated program is not safe if any of the selected input, filter, or sink modules are always unsafe, that is `unsafe="1"`. A program is safe if any of the selected input, filter, sink, or exec query modules is always safe, that is `safe="1"`, *and* none are unsafe. If no module is safe, the generated program is unsafe. The filter module must be executed to be considered. In other words, if a complexity never executes the filter, then the filter's safe or unsafe marking is ignored. Table 3 expresses this as a table.

Table 3. Decision table for whether a set of modules is safe or unsafe.

	Any module has safe="1"	No module is always safe
Any module has unsafe="1"	not safe	not safe
No module is always unsafe	safe	not safe

The Code Complexity Modules, Sec. 4.8.2, explains when a filter may never be executed.

4.4 Input Modules

The `inputs.xml` file has one or more "sample" input modules. Each module provides one way for the generated program to get input.

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

- dir and path, comment, and imports: see Sec. 4.3.
- flaw: indicate whether this input is always safe, always unsafe, or neither for a particular flaw type. See Sec. 4.3.4 for more details. If this input has the same safe/unsafe value for most types of vulnerabilities, put "default" as the `flaw_type`.
- input_type: this string is placed in the manifest. It has no other function in VTSG.
- output_type: the type of output. The variable generated with the placeholder `{{out_var_name}}` in the code will be that type. An input module is selected if it matches the `input_type` of the Filter (or of the Sink, if the Filter `input_type` is "nofilter").

```
<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. 4. Example Input module. Instantiated at line 14 of Fig. 14.

- 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 Filter and Sink. Do not declare this variable.

The case generated from the example Input in Fig. 4 takes an argument from the command line as Input. The input string can be either safe or unsafe, depending on user input.

4.5 Filter Modules

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

```
<sample>
    <path>
        <dir></dir>
    </path>
    <comment></comment>
    <flaws>
        <flaw flaw_type="" [safe=""] [unsafe=""]/>
    </flaws>
    <imports></imports>
```

```
    <code></code>
    <input_type></input_type>
    <output_type></output_type>
    <options need_complexity=""/>
</sample>
```

- dir and path, comment, and imports: see Sec. 4.3.
- 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. 4.2.
- output_type: the output type of the filter. The variable generated with the placeholder `{{out_var_name}}` will be that type.
- flaw: indicate whether this filter is always safe, always unsafe, or neither for a particular flaw type. See Sec. 4.3.4 for more details. If this filter has the same safe/unsafe value for most types, put "default" as the `flaw_type`.
- 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 that will be used in the Input and in Sink. Do not declare these variables. `{{out_var_name}}` must receive a value in all possible executions of the filter.
  Tip: To generate a test without functional filtering, just assign `out_var_name` the value of `in_var_name`, e.g.,

  ```
  {{out_var_name}} = {{in_var_name}};
  ```

  and make the input_type and output_type `nofilter`. This passes the value from the Input directly to the Sink.
- options: By default filters are wrapped in complexities. Complexities may be disabled by adding `options` with `need_complexity="0"`. In other words, if `need_complexity` is in `<options />` with anything other than "1", VTSG does not generate any variations with complexities.

The example Filter file in Fig. 5 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.

```
<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. 5. Example Filter module. Instantiated in lines 19–27 of Fig. 15.

4.6 Sink Modules

All sink modules are in the language's `sinks.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>
```

- dir and path, comment, and imports: see Sec. 4.3.
- 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. If the flaw_group is missing or empty, flaw type subdirectories are created immediately under the language directory. The user can limit cases generated to certain flaw groups with `-g` command line options or certain flaws with `-f` options.
- 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. 4.2.

  The input type specifies the kind of data this sink needs from the filter (or from the input). VTSG only selects filters whose output types are the same as this input type. If the filter input type is "nofilter", then VTSG selects input modules whose output types are the same as this input type.
- 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 always safe or always unsafe. For instance, a deprecated function may be marked (always) unsafe. See Sec. 4.3.4 for more details.
- 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 line following this. In generated unsafe cases, {{flaw}} is replaced with the one-line comment string, see Sec. 4.2, and "flaw". It does not appear in generated safe cases.

```
<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. 6. Example Sink module. Instantiated at line 23 of Fig. 14.

The Sink example in Fig. 6 concatenates the filtered string with an SQL query. This block of code can only be used for SQL Injection. Whether or not it is vulnerable depends on the input string.

4.7 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>
```

- type: the type of the ExecQuery. This is used in the `exec_type` tag of the Sink to link them together during generation process. The type should only contain letters, numerals, and underscore ("_").
  Languages currently available for VTSG support many database management systems, including ORACLE, MySQL, MSSQL, PostgreSQL, SQLite, and XPATH. The syntax of each ExecQuery must be compatible with its associated database system language.

- safe: whether the ExecQuery always makes the case safe. See Sec. 4.3.4 for more details.

- dir and path, comment, and imports: see Sec. 4.3.

- 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.

```
<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. 7. Example Exec_Query module. Instantiated in lines 25–39 of Fig. 14.

The block of code in the Exec_Query example, Fig. 7, executes the SQL query, used for database management systems, including MySQL, Oracle, PostgreSQL, and SQLite. This example is vulnerable. If a non-vulnerable execution of an SQL query is required, use an SQL prepared statement.

4.8 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>` part and one `<complexities>` part. All condition modules are inside `<conditions>`. All complexity modules are inside `<complexities>`.

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


4.8.1 Test Condition Modules

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

- id: string indicating this condition. Appears in the test case file name. Typically this is a number.
- code: the source code of the conditional test.
- value: either `<value>True</value>` or `<value>False</value>` depending on whether the code always evaluates to true or false.

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

Fig. 8. Example test condition module. Instantiated in Fig. 14, line 16.

4.8.2 Code Complexity Modules

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

- id: string indicating this complexity. Appears in the test case file name. Typically this is a number.
- type: Used types are: `if`, `switch`, `goto`, `for`, `foreach`, `while`, `function`, and `class`. If the type is `class`, source code in the `<body></body>` is placed in an additional file that is created for this case. Invocation statements are generated for `function` and `class` types.

  An extra variable is created for `foreach` type complexities (with group `loops`). No other type has any effect on VTSG.
- group: Used groups are: `conditionals`, `jumps`, `loops`, `functions`, and `classes`. No group, other than `loops`, has any effect on VTSG.
- executed: whether the placeholder will be executed or not. Four values are allowed:
  - 0: Never executed
  - 1: Always executed
  - condition: Executed if the condition is true
  - not_condition: Executed if the condition is false

  Table 4 gives example code for each value.
- 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}}`.

- need_condition: “1” if this complexity needs a condition. This complexity is also combined with conditions (see Sec. 4.8.1) if `executed` is `condition` or `not_condition`. (optional)
- 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.

Table 4. An example of code for each value of "executed" showing whether "placeholder" code will be executed.

Value of executed	example code

0	`switch(6) {`
	`  case(6):`
	`  break;`
	`  default:`
	`  {{ placeholder }}`
	`  break;`
	`}`

1	`switch(6) {`
	`  case(6):`
	`  {{ placeholder }}`
	`  break;`
	`  default:`
	`  break;`
	`}`

condition	`if ({{ condition }}) {`
	`  {{ placeholder }}`
	`} else {`
	`  {}`
	`}`

not_condition	`if ({{ condition }}) {`
	`  {}`
	`} else {`
	`  {{ placeholder }}`
	`}`

- 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

{{placeholder}} where the Filter is inserted. It may also contain {{condition}} where the Condition is inserted.

- body: additional source code not in the main execution flow, e.g., functions or classes. This code is placed in a separate file if the type is class. (optional)

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

Fig. 9. Example Complexity module with a while loop. Instantiated in Fig. 14, lines 16–21.

VTSG can put several complexities 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. 9. It is instantiated in lines 16–21 of Fig. 14. Line 16 is the instantiation of the control flow condition specified in Fig. 8.

The data flow complexity is a method call within the while loop. The specification is in Fig. 10. The <code> part is instantiated in lines 18 and 19 of Fig. 14. The <body> part is instantiated in Fig. 15.

5. Adding to VTSG

The next subsection is a brief tutorial giving advice on how to add a new flaw to an existing language in VTSG. The subsection after that gives some advice on how to add a new language. Finally a subsection offers some guidance on adding new capabilities or features.

5.1 How to Add a Flaw

The easiest way to add a new flaw is modify a similar existing flaw to present the new flaw. If nothing is suitable, here is a tutorial for writing a brand new flaw.

There are so many interacting specifications for a flaw, it is best to write it in three steps: first, write the flaw in regular source code; second, decide how to divide the code among modules, and third, specify the pieces in VTSG.

As a first step, choose a generated test case and modify the source code to present the target flaw. Here is the core code of a potential divide by zero guarded by a filter:

```
data = console_input()

if data == 0:
```

```
<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. 10. Example Complexity module with a method invocation. The ⟨code⟩ part is instantiated in Fig. 14, lines 18 and 19. The ⟨body⟩ part is instantiated in Fig. 15.

```
        print('Invalid input')
        sys.exit(1)

    print(f'The reciprocal of {data} is {1/data}')
```

You may want to try different variants to find which fits your needs best. For instance, one variant of this flaw is for the filter to provide a safe value instead of aborting, like this: (Since the the console input line is always the same, we will not show it further.)

```
if data == 0:
    print('Invalid input')
    data = 1

print(f'The reciprocal of {data} is {1/data}')
```

Another variant is for the weakness itself to be guarded by the filter:

```
if data != 0:
    print(f'The reciprocal of {data} is {1/data}')
else:
    print('Invalid input')
```


It helps later checking to write the code to demonstrate the failure, instead of failing silently. In the above code if division by zero is ever attempting, Python aborts with failure messages. In contrast consider a path traversal weakness. Path traversal is when the user might access private directories. Here is a simple version. It intends to open a file named by the user in the `/home` directory:

```
tainted_1 = input() # read one line

file = os.path.join('/home', tainted_1)
f = open(file, 'r')
```

Running this with an input that exploits the weakness, like `../etc/passwd`, doesn't give any indication that the exploit succeeded. The file is opened and then the program ends. The following variant is better. It prints a line from the password file to demonstrate the exploit:

```
tainted_1 = input() # read one line

file = os.path.join('/home', tainted_1)
with open(file, 'r') as f:
    print(f.readline(), end='')
```

The second step is to decide what parts of the code are inputs, what parts are filters (which will be wrapped in complexities), and what parts are sinks.

If you want to generate cases where some piece of code may or may not be executed, put that code in Filter modules. The second example above might generate a case like this. (Comments show the origin of each piece of source code.)

```
# Complexity
if True:
    # Filter
    if data == 0:
        print('Invalid input')
        data = 1

# Sink
print(f'The reciprocal of {data} is {1/data}')
```

If the sink is guarded, as in the third example, it may need to be a Filter module. In this case, the "Sink" module may be empty.

```
# Complexity
if True:
    # Filter
    if data != 0:
        print(f'The reciprocal of {data} is {1/data}')
    else:
        print('Invalid input')
```

Now that you have an idea of how to split up the code, the third step is to write a new sink (or filter) module in VTSG for the flaw. Edit the sinks.xml (or filters.xml) file to add it, then run VTSG. It's likely that nothing is generated the very first time you try because VTSG cannot find a compatible input and filter.

Figure 3 suggests that the result of the Input module connects to and must match the input of the Filter module, and its output connects to and must match the input of the Sink. These modules are shown together in Fig. 11. Compatible inputs, filters, sinks, and exec

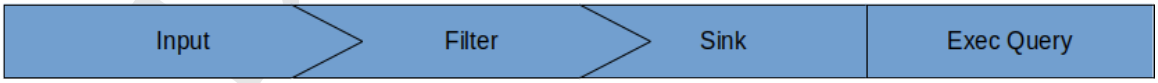

Fig. 11. The result of the Input module connects to the input of the Filter module, its output connects to the input of the Sink module, and its output connects to the input of the Exec Query module.

queries are "connected" various ways. Remember that VTSG chooses a sink module first then chooses compatible filter, input, and exec query modules to use with it. A filter is used with a sink if

1. the filter `output_type` is “nofilter” or the same as the sink `input_type` *and*
2. the filter has a `flaw_type` that is “default” or is the same as the sink `flaw_type`.

Figure 12 suggests that filters A, B, and C will be used with the sink. Filter B is compatible

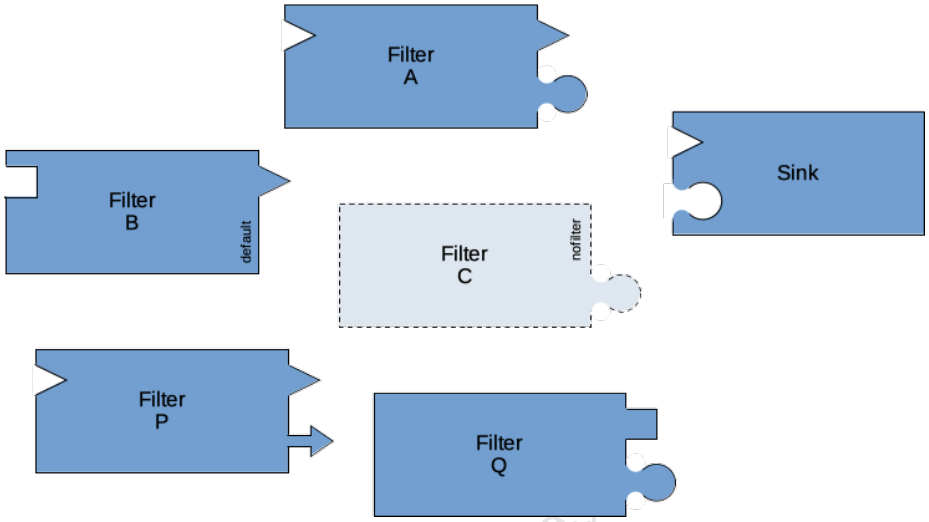

Fig. 12. How filters are chosen for a sink. Filters A, B, and C will be used. Filter P is not used because the output_type differs. Filter Q is not used because the flaw_type differs.

because the `flaw_type` is “default”; it is compatible with any type of flaw. Filter C is “nofilter” and just passes the value through. Filter P is never used with that sink because the `output_type` is not the same as the sink `input_type`. Filter Q is never used because the `flaw_type` is not the same.

An input is used with a sink and a filter if

1. the filter `input_type` is “nofilter” and the `output_type` of the input is the same as the sink `input_type` *or*
2. the filter `input_type` is something other than “nofilter” and the `output_type` of the input is the same as the filter `input_type`.

Fig. 13 illustrates this matching rule. Input 1 is used with Filter A, and Input 2 is used with Filter B. Input 1 is also used with Filter C because Filter C is “nofilter” and Input 1 matches the sink.

Be aware that any string is allowed in the `input_type` or `output_type`. This allows you to use a kind of “extended types” to connect specific inputs, filters, and sinks. An example is in Python. I wanted the input to be wrapped in complexities, like the following:

```
# Input
tainted_1 = None

# Complexity
if 5 == 5:
    # Filter
```

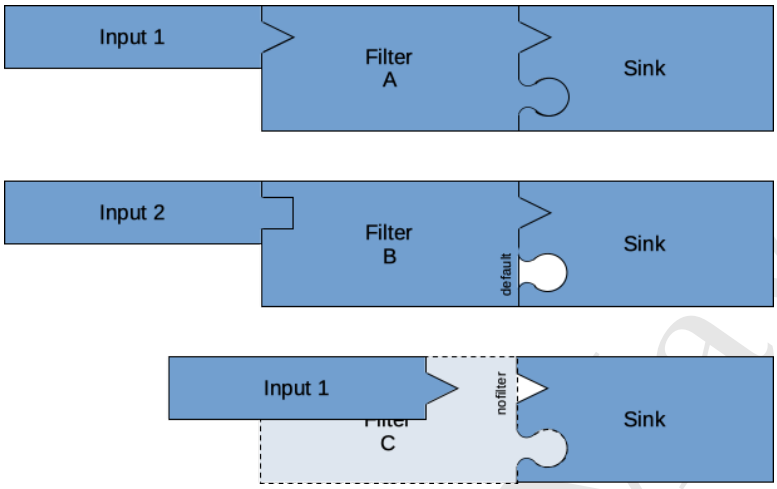

Fig. 13. How inputs are chosen for a filter and sink. Input 1 is used with Filter A. Input 2 is used with Filter B. Input 1 is also used with Filter C because Filter C is "nofilter" and Input 1 matches the sink.

```
    tainted_1 = input()

# Sink
if tainted_1 is not None:
    # flaw # no validation - allows arbitrary execution
    sys.path += [tainted_1]
    print(f'added { tainted_1 } to Python module search path')
```

Variables aren't declared in Python, but I had to initialize the variable to `None` in case the input is not executed. I declared the variable in the input module and put the input code in the filter module, so that the input code would be wrapped in complexities. I used custom types `string,filter_input` to connect the sink to the "filter" (really, input) and `InitToNone` to connect the "filter" to the "input" (really, initialization).

Figuring out how to mark the modules as safe or unsafe can be tricky. The following code may allow a large *negative* index access, and thus cause an exception. I thought this is always unsafe.

```
if index < len(array):
    print(array[index]
```

I discovered my mistake when I executed the cases and "unsafe" cases did *not* produce the exploit evidence. If the "input" was a safe hardcoded value, the code was safe!

```
index = 0

if index < len(array):
    print(array[index]
```

The proper marking is that the sink is neither always safe nor always unsafe. A proper filter or guard can make it safe. See Sec. 4.3.4 for details.

5.2 How to Add a Language

The easiest way is to

1. decide the short name of the language, e.g., py for Python or php for PHP. Using the extension of those programs is a good idea.
2. create a directory by that name under src/template
3. copy the six files of an existing language
4. change the name attribute in file_template.xml to the language name.

Change language characteristics in all the files. Generate a few cases and execute them to make sure syntax and other details are correct. It's best to start by converting just one or two sinks, inputs, and filters for quick turn around. Comment out all the rest of the modules to start. As you convert or add flaws, you may find that some language specifications need adjustment.

5.3 Adding Capabilities

New VTSG capabilities often allow the user who is describing programming languages or flaws to make new mistakes. Suppose input modules now have a `parallel` attribute, which must have one or more types of processing, such as Single Instruction, Single Data (SISD); Multiple Instruction, Single Data (MISD); Single Instruction, Multiple Data (SIMD); Multiple Instruction, Multiple Data (MIMD); Single Program, Multiple Data (SPMD), or Massively Parallel Processing (MPP). What should VTSG do when the user gives an input module a `parallel` attribute, but no type of processing? VTSG code could just crash with a traceback about `NoneType` object.

If the user can make a mistake, such as mismatched attributes or invalid fields, VTSG should explain what the problem is, where it arose, why it is invalid (e.g. where the information will be used), and what are correct approaches or alternatives. For example, for an empty `<dir></dir>` in an input module, VTSG reports

```
[ERROR] Invalid empty <dir></dir> in the inputs file.
A dir string is required; it is used in the name of the generated file.
```

then exits.

In case of an error due to a bug in VTSG itself, it may crash. In theory, VTSG developers will find and fix all bugs before the user runs it.

6. Generated Test Case File Names

This section describes how the names of test case files are created.

VTSG creates directories and subdirectories for the test cases that it generates. The directory structure is described in Sec. 3.1.

VTSG names test case files as `FLAW__I_INPUT__F_FILTER__S_SINK__EQ_EXEC_QUERY__NMBCPLX-CMPLX1[.COND]-CMPLX2[.COND]x.EXT`

- FLAW: Flaw type, e.g. CWE_89, BF, or STR30-PL, see Sec. 4.6.
- INPUT: Input description (dirs), see Sec. 4.3.1 (optional)
- FILTER: Filter description (dirs) (optional)
- SINK: Description (dirs) of the critical function
- EXEC_QUERY: ExecQuery description (dirs) (optional)
- NMBCPLX: The number of complexities.
- CMPLX1[.COND], CMPLX2[.COND], . . . : List of complexities. CMPLX is the id in the code complexity module, see Sec. 4.8.2. Tables 5, 7, and 9 in the language appendixes list complexity ids defined in C# , PHP, and Python. If the complexity has a condition, COND is the id in test condition modules, see Sec. 4.8.1. Tables 6, 8, and 10 list condition ids defined in C# , PHP, and Python. (optional)
- x: Sequence of the file within the test. If the test consists of just one file, there is no sequence letter. If the test consists of more than one file, that is, when the complexity `type` is `class`, see Sec. 4.8.2, the main file is "a", and other files, such as classes, are "b", "c", "d", etc. (optional)
- EXT: file extension, given in the `file_template.xml` file, see Sec. 4.2.

File names reflect the entire case, not just the code in a particular file. If a case consists of more than one file, as in the example used in this manual, all files have identical names, except for the final sequence letter.

As an example of a file name, consider the test case used in this manual. The case consists of two files. Fig. 14 is the main file of the example. Fig. 15 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__2-11.7-20a.cs`. The name of the class file, Fig. 15, is identical, except for the file letter "b" instead of "a" at the end.

The extension, cs, shows that it is a C# file. We understand the file name as follows:
CWE 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') [4]
The input comes from shell_commands, see specification in Fig. 4.
The filter is func_preg_match-only_numbers, Fig. 5.
The sink is select_from-concatenation_simple_quote__sql_server, Fig. 6.
The last part, 2-11.7-20, describes the complexities. The first number, 2, means this has two complexities. Tables 5 and 6 help us decode them. The first, outer complexity is 11 with condition 7. 11 is `while`, Fig. 9, with condition 7 meaning `Math.Sqrt(42)<=42`, which always evaluates to true, Fig. 8. The second, inner complexity is 20 meaning the sink code is executed in the class body, Fig. 10.
"a" means this is the main file.

```
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. 14. Main file of example. Line 14 instantiates input code from Fig. 4. Lines 16–19 instantiates complexity code from Fig. 9. Line 16 instantiates condition code from Fig. 8. Lines 18 and 19 instantiate code from the ⟨code⟩ part of Fig. 10. Line 23 instantiates critical preparation code from Fig. 6. Lines 25–39 instantiate query execution code from Fig. 7.

```
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. 15. Auxiliary file of example. It instantiates code from the ⟨body⟩ part of Fig. 10. Lines 19–27 instantiate filter code from Fig. 5.

7. Acknowledgments

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] (2017) OWASP top 10 web application security risks, https://owasp.org/www-project-top-ten/. Accessed: 17 June 2020.

[3] (2020) Common weakness enumeration, https://cwe.mitre.org/. Accessed: 17 June 2020.

[4] (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.

A. C# Language

This appendix documents the flaws, flaw groups, complexities, and conditions currently in the C# language files.

The following flaws are currently defined for this language:

- SQL Injection (CWE-89)
- XPath Injection (CWE-91)
- LDAP Injection (CWE-90)
- OS Command Injection (CWE-78)
- Path traversal (CWE-22)
- Information Leak Through Error Message (CWE-209)
- Storing Password in Plain Text (CWE-256)
- Use of Insecure Cryptographic Algorithm (CWE-327)
- NULL Pointer Dereference (CWE-476)

The flaws are in the following groups:

- OWASP_a1 has CWE_78, CWE_89, CWE_89, CWE_89, CWE_90, CWE_91, CWE_91, and CWE_91.
- OWASP_a2 has CWE_256.
- OWASP_a4 has CWE_22.
- OWASP_a5 has CWE_209.
- OWASP_a6 has CWE_327
- OWASP_a9 has CWE_476.

Here are the complexities currently available in C#. We explain the concept of code complexities in Sec. 3.4 and the format of complexity modules in Sec. 4.8.2. The brief descriptive pseudocode reminds the reader of the complexity. See the `complexity.xml` file for the specific code.

Table 5. Complexity IDs and pseudocode defined for C#

ID	Pseudocode
1	if condition code
2	if condition code else
3	if condition else code
4	if condition code else if not condition
5	if condition else if not condition code
6	if condition code else if not condition else
7	if condition else if not condition code else
8	if condition else if not condition else code
9	switch code executed
10	switch code not executed
11	while code
12	do code while
13	for code
14	foreach code
15	goto code not executed
16	goto code executed
17	function body executes code
18	input passed via function then code
19	code then output passed via function
20	class body executes code
21	input passed via class then code
22	code then output passed via class

Here are the conditions currently available to be used in code complexities. We explain condition modules in Sec. 4.8.1. Table 6 shows the ID, the code, and whether it always evaluates to true or false.

Table 6. Condition IDs, code, and value to which it evaluates defined for C#

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

B. PHP Language

This documents the flaw, flaw group, complexities, and conditions currently in the PHP language files.

The following flaw is currently defined for this language. The flaw group is in parentheses.

- SQL Injection (CWE-89) (OWASP_injection)

Here are the complexities currently available in PHP. We explain the concept of code complexities in Sec. 3.4 and the format of complexity modules in Sec. 4.8.2. The brief descriptive pseudocode reminds the reader of the complexity. See the `complexity.xml` file for the specific code.

Table 7. Complexity IDs and pseudocode defined for PHP

ID	Pseudocode
1	if condition code
2	if condition code else
3	if condition else code
4	if condition code else if not condition
5	if condition else if not condition code
6	if condition code else if not condition else
7	if condition else if not condition code else
8	if condition else if not condition else code
9	switch code executed
10	switch code not executed
11	while code
12	do code while
13	for code
14	foreach code
15	goto code not executed
16	goto code executed
17	function body executes code
18	input passed via function then code
19	code then output passed via function
20	class body executes code
21	input passed via class then code
22	code then output passed via class

Here are the conditions currently available to be used in code complexities. We explain condition modules in Sec. 4.8.1. Table 8 shows the ID, the code, and whether it always evaluates to true or false.

Table 8. Condition IDs, code, and value to which it evaluates defined for PHP

ID	Code	Value
1	`1==1`	True
2	`1==0`	False

C. Python Language

This documents the flaws, flaw groups, complexities, and conditions currently in the Python language files.

The following flaws are currently defined for Python. The flaw group is in parentheses. Flaws in the "Exception" flaw group are caught by the Python runtime. These are unlikely to be vulnerabilities (except denial of service).

- OS Command Injection (CWE-78) (OWASP_a1)
- SQL Injection (CWE-89) (OWASP_a1)
- Path traversal (CWE-22) (OWASP_a4)
- Information Leak Through Error Message (CWE-209) (OWASP_a5)
- Loop on Unchecked Input (CWE-606) (Other)
- Relative Path Traversal (CWE-23) (Other)
- Divide by Zero (CWE-369) (Exception)

Here are the complexities currently defined. We explain the concept of code complexities in Sec. 3.4 and the format of complexity modules in Sec. 4.8.2. The brief descriptive pseudocode reminds the reader of the complexity. See the `complexity.xml` file for the specific code.

Table 9. Complexity IDs and pseudocode defined for Python

ID	Pseudocode
1	if condition code
2	if condition code else
3	if condition else code
4	if condition code elif not condition
5	if condition elif not condition code
6	if condition code elif not condition else
7	if condition elif not condition code else
8	if condition elif not condition else code
11	match case code not executed
12	match case code executed
13	no match case code executed
14	no match case code not executed
20	while condition code break
20n	while condition break code
22	for range(0, 1) code
22	for array if value code
50	function body executes code
51	input passed via function then code
52	code then output passed via function
70	class body executes code
71	input passed via class then code
72	code then output passed via class

Here are the conditions currently available to be used in code complexities. We explain condition modules in Sec. 4.8.1. This table shows the ID, the code, and whether it always evaluates to true or false.

Table 10. Condition IDs, code, and value to which it evaluates defined for Python

ID	Code	Value
1	`1==1`	True
2	`1==0`	False
2u	True	True
2v	False	False
3	`4+2<=42`	True
3u	`5==5`	True
3v	`5!=5`	False
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

D. Contents of git Repository

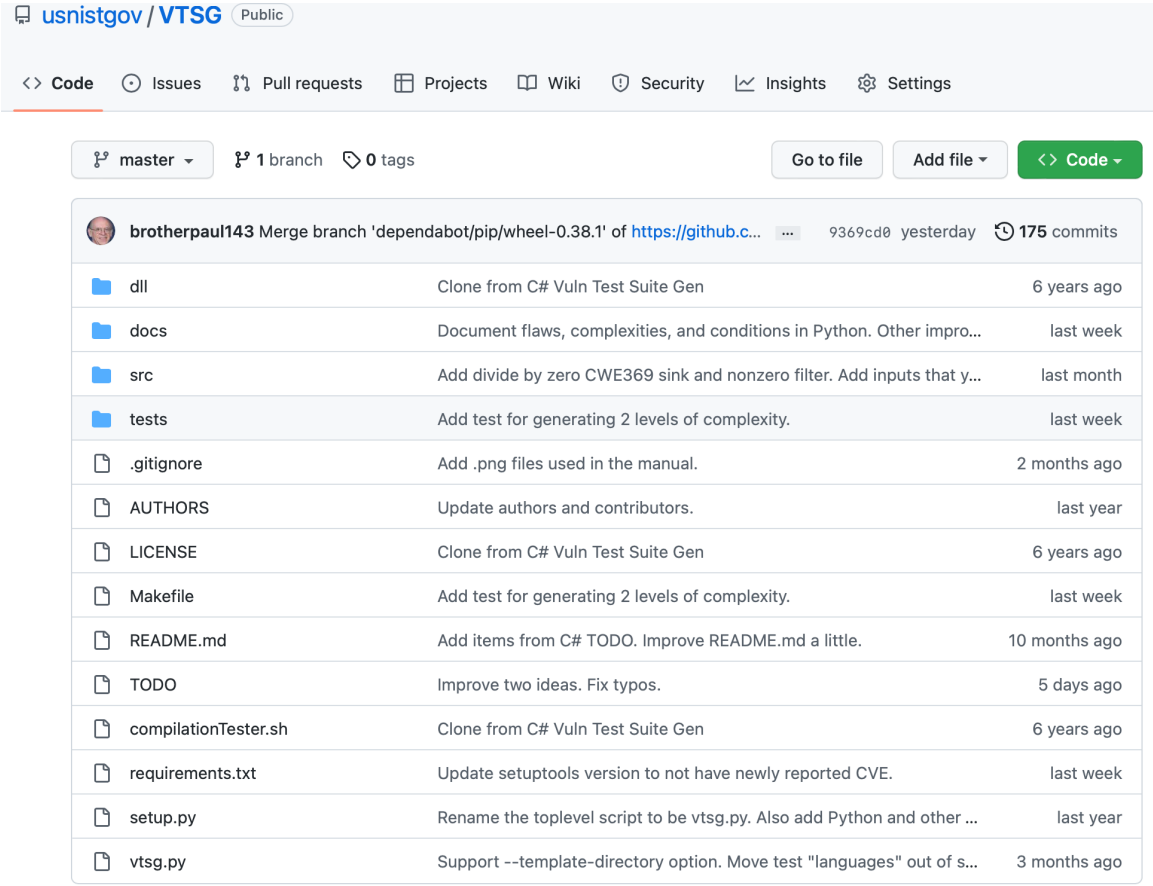

Fig. 16. Snapshot of files in the VTSG git repository, which is at https://github.com/usnistgov/VTSG, as of 12 January 2023.

README.md

Vulnerability Test Suite Generator

Program to generate vulnerable and fixed synthetic test cases expressing specific flaws.

Written in Python 3

Dependencies

- Jinja2 (depends on MarkupSafe)
- Docopt
- Setuptools (for setup.py)
- Sphinx (for generating the doc)

You have three ways to install these dependencies

Using PIP

We encourage you to use pip (installation instructions) to install these dependencies (choose one):

```
- [sudo] pip3 install -r requirements.txt (as root, system-wide)
- pip3 install --user -r requirements.txt (only for your user)
```

Using a Package Manager

You can also install this dependency with your package manager (if such a package exists in your distribution) :

```
- [sudo] aptitude install python3-jinja2 (for GNU/Linux Debian for exampl
- [sudo] aptitude install python3-docopt
- [sudo] aptitude install python3-setuptools
- [sudo] aptitude install python3-sphinx
```

Manually Installation

Jinja2 installation instructions here

Docopt installation instructions here

Setuptools site

Execute it

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

Need help ?

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

Fig. 17. README.md file, which is at https://github.com/usnistgov/VTSG/blob/master/README.md, as of 8 March 2022.