NIST Series IR
NIST IR 8493

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.8493

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Series IR
NIST IR 8493

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

Bertrand Stivalet
International Committee of the Red Cross
Geneva, Switzerland

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8493

October 2023

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

NIST Technical Series Policies
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History
Approved by the NIST Editorial Review Board on 2023-10-12

How to cite this NIST Technical Series Publication:
Paul E. Black, William Mentzer, Elizabeth Fong, and Bertrand Stivalet (2023) Vulnerability Test Suite Generator (VTSG) Version 3. (National Institute of Standards and Technology, Gaithersburg, MD), NIST IR 8493. https://doi.org/10.6028/NIST.IR.8493

NIST Author ORCID ID
Paul E. Black: 0000-0002-7561-6614

Abstract

The Vulnerability Test Suite Generator (VTSG) Version 3 can create vast numbers of synthetic programs with and without specific flaws or vulnerabilities. Such programs are usefull for measuring static analysis tools. VTSG was designed by the Software Assurance Metrics and Tool Evaluation (SAMATE) team and originally implemented by students at TELECOM Nancy. The latest version is structured to be able to generate vulnerable and nonvulnerable synthetic programs expressing specific flaws in typical programming languages. It has libraries to generate programs in PHP: Hypertext Preprocessor (PHP), C#, and Python. This document may help if you are trying to generate test cases in PHP, C#, or Python, adding new complexities or flaws or vulnerability, or modifying VTSG to have new capabilities or to generate test cases in other programming languages.

Keywords

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

Table of Contents

List of Tables

List of Figures

Acknowledgments

Bertrand Stivalet and Aurelien Delaitre, from the National Institute of Standards and Technology SAMATE team, designed the architecture of the VTSG 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 Charles de Oliveira and Sheryl Taylor for their contributions to this manual.

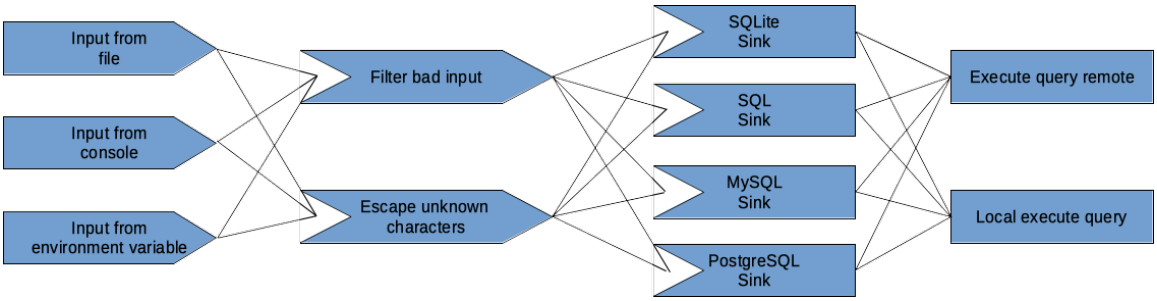

Fig. 1. Just three sources of input, two ways of filtering, four sinks, and two query executions yield 48 possible test cases.

1. Introduction

Good software must be specified and built well from the beginning; quality cannot be "tested into" software. But a critical part of essentially all software development is checking the software produced. This checking has two main dimensions: testing, or execution of the software, and static analysis, or examination of the source code or executable. Static analysis has the theoretical advantage of considering *all* possible executions. It "has the potential to efficiently preclude several classes of errors . . . " [1, p. 7]. Current static analysis tools check many issues, such as uninitialized variables, buffer overflows, Structured Query Language (SQL) injections, noncompliance with an organization's coding standards, and calling a function with incorrect arguments. Although static analyzers will not catch all problems, NIST's "Guidelines on Minimum Standards for Developer Verification of Software" [2] recommends that all software development include appropriate static analysis tools (p. 6).

We can measure how well static analyzers perform using programs having known flaws. With more than 100 classes of flaws and many popular programming languages, the task of producing measurement test suites is already daunting. For thorough evaluation each flaw class should be represented by programs with inputs from different source, various valid and invalid methods to filter the input data, different ways of conveying data, and varying sinks. 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 $3 \times 2 \times 4$ x $2 = 48$ combinations. Multiply that by distinct flow control complexities with various conditions, then include "scaffolding" such as initializing variables, importing libraries, and declaring functions and the need to produce programs without weaknesses to test false positives to get an idea of the work needed to provide a comprehensive static analyzer test suite. It would be tedious and error prone to construct these test cases manually.

The purpose of the Vulnerability Test Suite Generator (VTSG) is to generate thousands of example programs in many languages exhibiting various flaws, which are expressed in different ways and wrapped in an assortment of programming constructs.

All example programs, or test cases, have a similar structure. In pseudocode, the high-level structure of test cases that VTSG generates is

```
imports, definitions, and declarations

input = get_input_from_somewhere()

if input is an attack then
    reject it

use the input to construct a query

execute the query
```

The *input* is the source of data in the program, e.g., command line, variable, file, or form method.

In VTSG we refer to the code that checks the input as the *filter*. That is, it filters the input with functions or code such as rejecting unacceptable values, sanitization functions, or substituting a safe default value. In this example, the `if input ... reject it` code is the filter.

VTSG refers to the code that uses the input, in this example to construct a query, as the *sink*. Information flows from the input source to the sink. The sink is where a sensitive operation, such as an array access, is executed with potentially unsafe input and where a vulnerability is triggered.

You can see this high-level program structure within the center gray rectangle of Fig. 2.

A static analysis tool must analyze much of a program, noting its control and data flows, to accurately track data and determine the conditions when the code in question may be executed. Only then can it process the few lines of code that embody a problematic piece of code to determine whether it is actually a weakness. Tracing execution through structures like `while` loops, `if` statements, and function calls exercises static analysis tools' abilities. VTSG refers to these data flow and control constructs as *complexities*. You can see in Fig. 2 that the filter code is wrapped within two control flow complexities. Various *conditions* are inserted into complexities to further examine tools' capabilities. See Sec. 2.4 for an example.

Finally, additional code may be needed to trigger the vulnerability, for example executing the query constructing in the sink code. VTSG refers to this as *ExecQuery*.

The information in this manual is divided into several sections. The next section, Sect. 2, is a user manual. It explains how to invoke various options in VTSG with currently defined languages and flaws and the resulting cases. Section 3 explains how to install VTSG and required supporting packages, add new flaw classes, sinks, inputs, etc., to existing languages

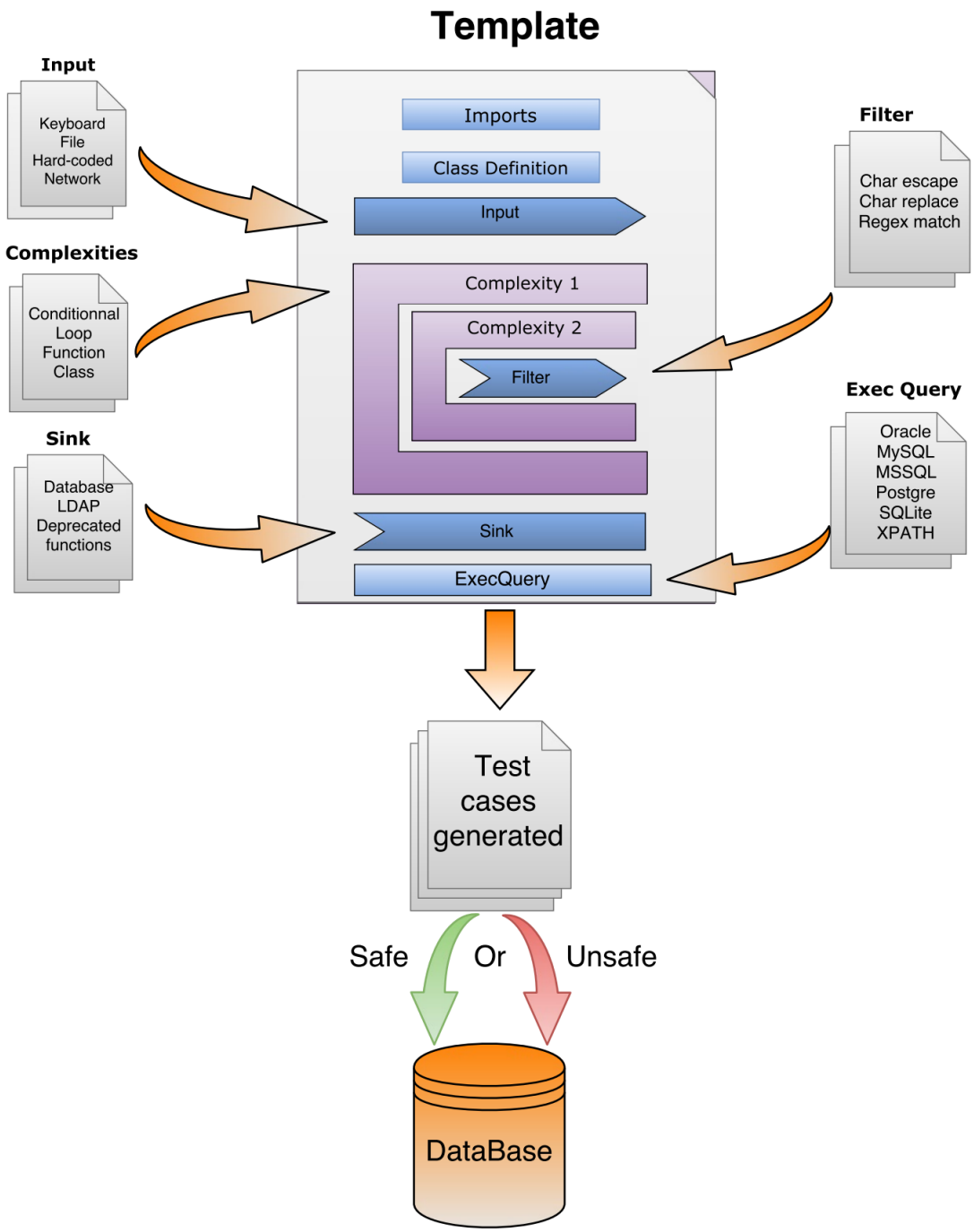

Fig. 2. Overview of VTSG test case generation. The File Template specifies how pieces are assembled. Input, Filter, Complexity, Sink, and Execute Query modules provide code. The filter is wrapped with two complexities. An example test case is in Figs. 14 and 15.

and how to add a completely new language. Section 4 documents the files to which new flaws, sinks, or languages would be added. Finally, Sect. 5 explains the internal architecture of VTSG and offers suggestions about how to enhance or extend VTSG's capabilities.

1.1. Strengths and Limitations of VTSG

VTSG generates test cases starting with the File Template. It selects Input, Filter, Complexity, Sink, and ExecQuery modules and inserts them into the Template. In the most general sense, VTSG works like a macro processor. Why, then, bother with VTSG?

- VTSG synthesizes import statements, variable initializations, additional assignments, and other code as needed.
- VTSG wraps filter code in arbitrary layers of data and control flow complexity.
- VTSG determines if a test case is unsafe (vulnerable) or safe (not vulnerable) depending on whether the filter is executed and saves them in corresponding collections.
- VTSG matches data types of inputs, filters, and sinks, for example avoiding passing a string input to a numeric sink.
- VTSG can produce a manageable fraction of all possible combinations, choosing 1 of every N cases or enough cases to try every pair or triple of code.

VTSG is not nearly as powerful as we could wish for. Here are some limitations and possible future enhancements.

- Have a method of variants. For example, data types are given once, then a "generic" case is elaborated with each data type. This would yield test cases that are structurally identical but handle, say, ints, floats, doubles, etc.
- Enhance support for multi-file cases. In particular, be able to handle Python classes. Also create cases with more than two files, like Juliet cases.
- Support more general structure of generated test cases. For example, put both safe and unsafe code in the same test case, like Juliet cases have, or have module types instead of or in addition to input, filter, sink, and execQuery.
- Mark "flaw" lines in any module, not just sinks.
- Allow the user to specify the file names of generated cases. Currently the names are long. It may be useful to structure the names differently or even just number them.
- Write manifests in Static Analysis Results Interchange Format (SARIF).
- Enhance the Automated Combinatorial Testing for Software (ACTS) interface code to handle arbitrary depths of nested complexities.
- Have a new test condition value such as "maybe", "random", or "indeterminate" for when the analyzer would not be able to determine. For example, Juliet has

```
if (random()) {
    code
}
```

1.2. History

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

The PHP and C# test suites are available in our Software Assurance Reference Dataset (SARD) [4], URL https://samate.nist.gov/SARD as test suites 103 and 105. The SARD has additional collections of test cases, such as the C/C++ and Java Juliet test suites and many other test suites.

VTSG is further work by the SAMATE team to measure software assurance. The test cases that VTSG generates help us determine how much assurance static analyzers, and other software assurance tools, can bring. This complements our work collecting test cases in the SARD. VTSG also supports our Static Analysis Tool Exposition (SATE) [5], which requires large, varied sets of programs as targets of analysis. We foresee that VTSG will be able to generate test cases structured as needed in many languages, for most weakness classes.

2. VTSG User Manual

2.1. Command Line Interface

For users who wish to generate PHP, C#, or Python test suites, a command line interface can generate all test cases, a specific group of test cases, or a subset of 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, Open Web Application Security Project (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. 3.5.
-g GROUP --group=GROUP	Only generate cases in the specified group. May be repeated. See Sec. 4.6.
-f Flaw --flaw=Flaw	Only generate cases with the specified flaw. 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. 2.4.
-n NUMBER --number-sampled=NUMBER	Only write one of every NUMBER cases generated. See Sec. 2.5.
--ACTS [doi]	Select cases using ACTS. (Default doi: 2) See Sec. 2.5. Note: no short option.
-t TEMPLATE_DIRECTORY --template-directory=TEMPLATE_DIRECTORY	The language templates directory. (Default: src/templates) See Sec. 3.5.
-d --debug	for programmer use

By default, VTSG generates all consistent combinations of test cases with all flaws in all flaw groups. If VTSG is invoked with particular flaws (`-f`) or flaw groups (`-g`), only sinks matching those are used. If no flaw group is specified, all flaw groups are used. If no flaws are specified, all flaws (in specified flaw groups) are used.

VTSG generates both the unsafe (vulnerable) test cases and the safe (not vulnerable) test cases unless the user selects either only safe (-s) or only unsafe (-u) cases. The options are mutually exclusive.

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 consisting 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
```

Generate a Python test suite consisting of every 42nd test case based on Kendra Kratkiewicz's work (KK) or Loop on Unchecked Input (CWE606).

```
$ python3 vtsg.py -l py -n 42 -f KK -f CWE606
```

Some of the OWASP Top 10 [6] and Common Weakness Enumerations (CWEs) [7] are encoded. See Apps. A, B, and C for details of what vulnerabilities are encoded in which languages.

2.3. Results and Output

When invoked to generate test cases, VTSG creates a directory for all the resulting test cases. The directory is named for the date and time created, for example, `TestSuite_03-08-2022_16h46m35`. A language directory, `PHP`, `Csharp`, or `Python`, is created in this directory. This 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 `Exception`. 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, `CWE78` 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 `safe` and `unsafe` for test cases that are not vulnerable or that are vulnerable, respectively.

After VTSG finishes generating and writing cases, it displays how many safe (non-vulnerable) and unsafe (vulnerable) test cases it generated in each group and flaw. If a subset was selected with -n or --ACTS (Sec. 2.5), VTSG also reports the number of test cases selected.

2.4. Code Complexities

VTSG can generate code with the filter nested inside control flow complexities to create slightly more realistic source code. Each language defines its own complexities. See

Apps. A, B, and C for currently defined complexities.

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.

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/else" statements.

Level 2 is the "switch/case/default" statements.

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

2.5. Using ACTS or -n to Select a Subset of Cases

By default, VTSG generates all compatible combinations of inputs, filters, sinks, exec queries, complexities, and conditions. The result can easily be tens of thousands of cases. Yet, a small, carefully chosen subset of cases may suffice [8]. VTSG has two command line options to select a subset of cases to be written: --ACTS and -n.

ACTS is a combinatorial method to choose cases such that all consistent pairs or triples or D-way combinations of modules are represented. For an explanation of combinatorial testing, see [9] especially Sec. 2.1.

Degree of interaction (doi) is the coverage guaranteed. Two-way means every pair of pa-

rameter values is represented, like every (allowed) input with every complexity. If doi is 3, then every triple is covered, e.g., every input, sink, and condition combination. Again only allowed combinations[1] are created. The default is for ACTS to generate pairwise coverage, that is, the default degree of interaction is two.

For example,

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

generates a small set of PHP test cases that has all pairs of compatible modules.

```
$ python3 vtsg.py -l py -g Exception --ACTS 4
```

generates a small set of Python test cases that has all 4-way combinations of compatible modules in the `Exception` flaw group.

VTSG first generates all consistent combinations of modules, Sec. 5.1. If `ACTS` is indicated, VTSG writes the list of modules and constraints on compatible modules to `/tmp/VTSG_ACTS_input.xml`, runs ACTS, reads the selected sets of modules from `/tmp/VTSG_ACTS_output.txt`, and creates a list of matching module combinations. VTSG then composes the code from modules and writes the test cases.

Currently, the code to write input for ACTS and read its output does *not* handle more than one complexity in a case. Therefore, `-r` (depth) greater than 1 and --ACTS may not be used together. It should be straight forward to enhance the code, which is in `select_by_acts.py`, to handle any number of nested complexities.

The -n (number-sampled) option is another way to select which cases VTSG writes. It specifies that VTSG should write one of every N cases that are generated. For example, -n 3 directs VTSG to write one of every three cases, that is, case 1, skip 2 and 3, write case 4, skip 5 and 6, write case 7, and so forth. This option approximates random case selection (but is repeatable), combinatorial testing (in case ACTS is not installed), random testing (to spot check generation), and percentage of cases (that is, only write p % of cases). Note that there is no way to generate mixed fractions such as exactly $2/5$ of all cases.

The -n option and --ACTS are mutually exclusive.

3. Advanced User Manual

This section begins by explaining how to install VTSG, including required and optional supporting packages. Section 3.4 is a brief tutorial on how to add a new flaw class. The final section explains how to add a language in addition to PHP, C#, and Python.

[1] The interface code checks which combinations are actually generated and writes ACTS constraints to *not* create combinations that never occur. For example, one filter, named "none", just passes the input value; it doesn't filter anything. The filter is marked as need_complexity="0" (See Sec. 4.5), to avoid wrapping it in any complexity: it would be useless to wrap an "if" or "while" around it. The interface code sees that filter "none" never has complexities, so constrains ACTS to only match that filter with "no complexity".

3.1. Installing Supporting Packages

The following instructions are provided for users who do not have these packages installed on their Linux machines that use apt for package management. Users who already have these packages may skip this section.

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

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

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

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

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

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

Another way to install *pip* is:

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

3.2. Installing VTSG

To download VTSG from Github, the *Git* package must be installed. Here is a command to install it:

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

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, you will see a green box labeled "Code". See Fig. 3. Click on it, then click on the "copy" icon to copy the web uniform resource locator (URL).

Here is a command to copy the source code and other material to the local directory:

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

If you type the `ls` command, you will see that the **VTSG** directory was created.

Go into that directory and install the required dependencies with these commands:

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

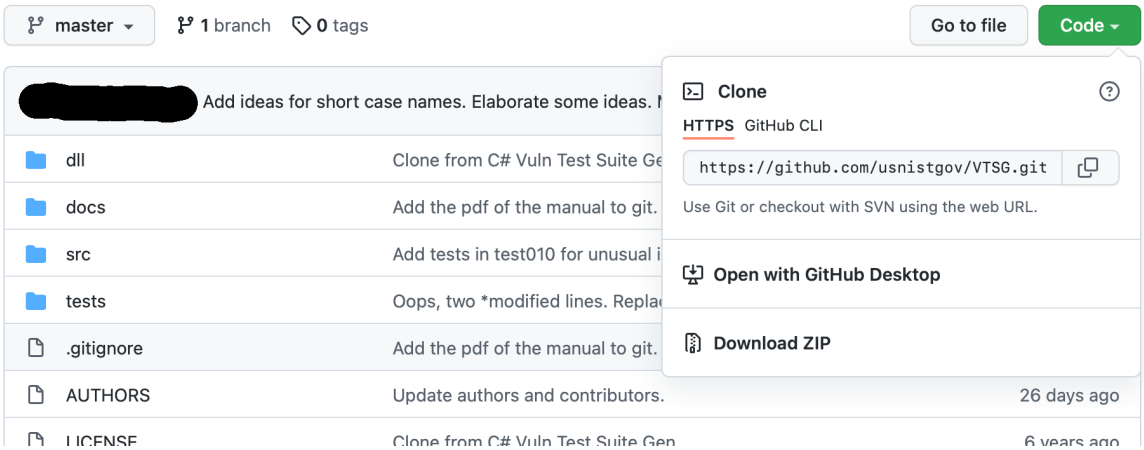

Fig. 3. usnistgov/VTSG: button to clone repository.

3.3. Installing Optional Packages

VTSG can use combinatorial testing, specifically ACTS [10], to select cases instead of producing all possible cases. The ACTS software is freely available. To get ACTS, send email to Rick Kuhn kuhn@nist.gov [11].

See App. A for packages needed to run C# programs.

3.4. Adding a New Flaw

This subsection is a brief tutorial giving suggestions on how to add a new flaw to an existing language in VTSG. The subsection after this has suggestions on how to add a new language. Some guidance on adding new capabilities or features to VTSG itself is in Sec. 5.3.

The easiest way to add a new flaw is to 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 requirements for a flaw, it is best to write it in three steps: first, write an example of the flaw in regular source code; second, decide how to divide the code among modules, and third, specify the pieces in VTSG.

3.4.1. Write an Example With the Flaw

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:

```
if data == 0:
    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:

```
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 so that it demonstrates the failure, instead of failing silently. In the above code if division by zero is ever attempted, 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='')
```

3.4.2. Decide What is Input, What is Filter, What is Sink, etc.

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.)

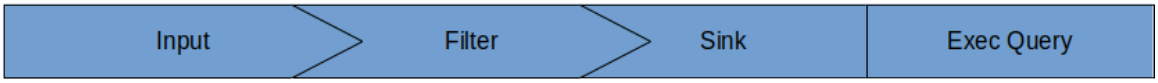

Fig. 4. The result of the Input module connects to the input of the Filter module, its output connects to the input of the Sink module, and its output connects to the input of the Exec Query module.

```
# 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')
```

3.4.3. Write the Modules

Now that you have an idea of how to split up the code, the third step is to write new modules for VTSG for the flaw. You probably need to write new sinks and filters, but you should be able to use existing input and exec query modules. Edit the sinks.xml (or filters.xml or other) file to add them, then run VTSG. It's likely that nothing is generated the very first time you try because VTSG cannot find an input and filter compatible with your new sink.

Figure 2 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. 4. Compatible inputs, filters, sinks, and exec queries are "connected" in various ways. Remember that VTSG first chooses a sink module 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`.

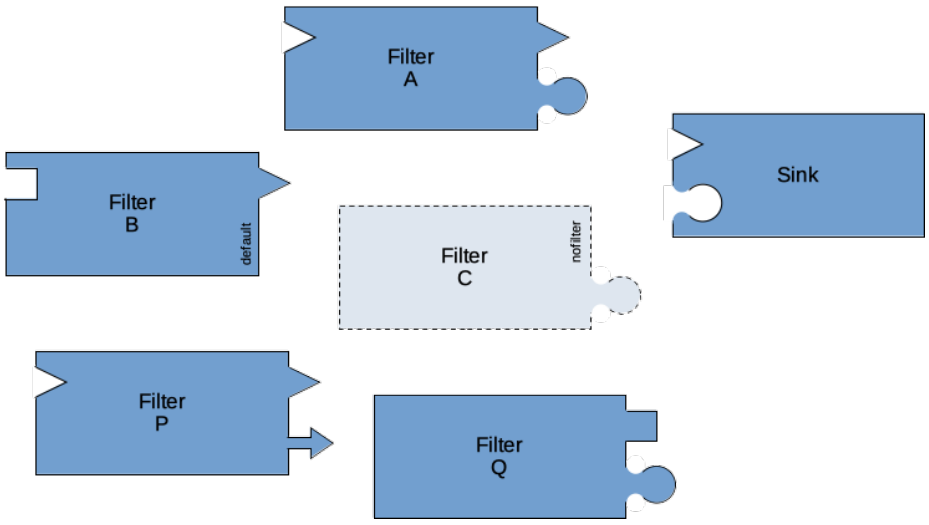

Fig. 5. How filters are chosen for a sink. Filters A, B, and C will be used. Filter P is not used because the flaw_type differs. Filter Q is not used because the output_type differs.

Figure 5 suggests that filters A, B, and C will be used with the sink. Filter B is compatible 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 `flaw_type` is not the same. Filter Q is never used because the `output_type` is not the same as the sink `input_type`.

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. 6 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 may be in the `input_type` or `output_type`. This allows you to use a kind of "extended types" to connect specific inputs, filters, and sinks. Following is an example, which is in Python. We wanted the input to be wrapped in complexities, like the following:

```
# Input
tainted_1 = None

# Complexity
if 5 == 5:
```

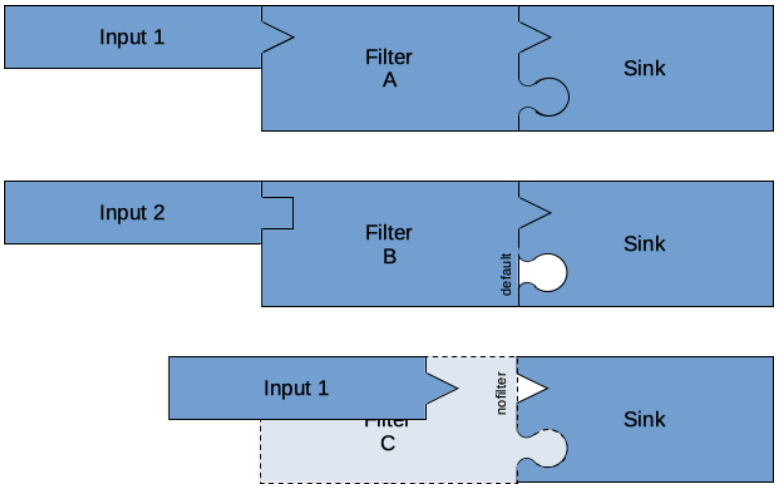

Fig. 6. How inputs are chosen for a filter and sink. Input 1 is used with Filter A. Input 2 is used with Filter B. Input 1 is also used with Filter C because Filter C is "nofilter" and Input 1 matches the sink.

```
        # Filter
        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 we had to initialize the variable to `None` in case the input is not executed. We 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. We used custom types `string,filter_input` to connect the sink to the "filter" (i.e., input) and `InitToNone` to connect the "filter" to the "input" (i.e., initialization).

3.4.4. Making Sure Case Safety is Computed Correctly

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. We thought this is always unsafe.

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

We discovered the mistake when we executed the cases and "unsafe" cases did *not* produce any evidence of a successful exploit. We realized that, 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.

3.5. Adding a New Language

VTSG accesses all the language information files through a single subdirectory. The default is `src/templates`. Under this is one subdirectory for each language. Each language subdirectory has six files: `file_template.xml`, `inputs.xml`, `complexities.xml`, `filters.xml`, `sinks.xml`, and `exec_queries.xml`. The files are described in Sec. 4.

Here are the steps to add another language.

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 turnaround. 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.

In addition to subdirectories for languages, 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 Extensible Markup Language (XML) files.

You can tell VTSG to look for a language subdirectory someplace else with the -t command line option, see Table 1.

4. File 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 XML-special characters, 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 chunks 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, VTSG adds indentation 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 in code that might be nested within it must be indicated with INDENT ... DEDENT lines.

We chose “DEDENT” because Python’s grammar description uses it.

4.2. File Template

```
<template name="">
    <file_extension></file_extension>
    <comment>
        <open></open>
        <close></close>
        <inline></inline>
    </comment>
    <syntax>
        <statement_terminator>;</statement_terminator>
        <indent>    </indent>
        <import_code>using {{import_file}};</import_code>
    </syntax>
    <variables prefix="">
        <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 that may span many lines
  - close: string to end a comment that 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.
  - import_code: Code to include a library. The code must have the placeholder `{{import_file}}`. For example, `#include <{{import_name}}>`. Note: include any needed statement terminator. None is added.
- 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 no prefix is required.
  - variable: Define each variable type and how to initialize it. (optional)
    - type: Names the type. This string does not appear in the generated 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.
- imports: libraries that are always imported. See Sec. 4.3.3.
- 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

- 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 module types use the same attributes. Instead of repeating explanation of these attributes in each module, we explain them 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. 5.2. 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_`.

In the future this should be changed to have one `<path>` string, instead of multiple `<dir>` description strings.

4.3.2. Module Comment

If a sample module has a comment string, it is added to the `{{comments}}` area 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 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 example, 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"	unsafe	unsafe
No module is always unsafe	safe	unsafe

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

```
<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. 7. Example Input module. Instantiated at line 14 of Fig. 14.

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: may indicate whether this input is always safe or always unsafe. 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").
- 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. 7 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`.

```
<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. 8. Example Filter module. Instantiated in lines 19–27 of Fig. 15.

- code: The source code of a 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. 8 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.

4.6. 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

```
<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. 9. Example Sink module. Instantiated at line 23 of Fig. 14.

`{{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 example, 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 `{{in_var_name}}`. It 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 unsafe cases, `{{flaw}}` is replaced with the one-line comment string, see Sec. 4.2, and "flaw". Nothing appears in safe cases.

The Sink example in Fig. 9 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 ("_").
  The C# language currently supports 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.

The block of code in the Exec_Query example, Fig. 10, executes the SQL query, used for database management systems, including MySQL, Oracle, PostgreSQL, and SQLite. This example is vulnerable.

4.8. Code Complexity and Test Condition 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>
        ....
```

```
<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. 10. Example Exec_Query module. Instantiated in lines 25–39 of Fig. 14.

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

Fig. 11. Example test condition module. Instantiated in Fig. 14, line 16.

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

Fig. 12. Example Complexity module with a while loop. Instantiated in Fig. 14, lines 16–21.

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

```
<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. 13. 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.

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.

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.

- need_id: "1" if the code has a placeholder, `{{id}}`, to generate a unique identifier (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)

VTSG can put several complexities in one test case. The example in Sec. 5.2 has two instances of Complexity: a control flow complexity and a data flow complexity. The control flow complexity specification is in Fig. 12. It is instantiated in lines 16–21 of Fig. 14. Line 16 is the instantiation of the control flow condition specified in Fig. 11.

The data flow complexity is a method call within the `while` loop. The specification is in Fig. 13. The `<code>` part is instantiated in lines 18 and 19 of Fig. 14. The `<body>` part is instantiated in Fig. 15.

5. VTSG Software Documentation

This section has high-level documentation of VTSG and directions to help add extensions or new features.

5.1. Details of Test Case Generation

Each test case is constructed based on the file template, as shown in Fig. 2. 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 source code from the selected modules, synthesizing variable and function names, and writes the file(s). The user may specify two ways that VTSG selects a subset of cases to write; see Sec. 2.5.

The structure of the VTSG code itself is broadly

```
for each specified sink
    for each filter
        for each input
            for each exec query
                for up to DEPTH combinations of each complexity
```

```
                    save this set of modules
for each set of saved modules
    compose this set of modules into code for a test case
```

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 actually structured as a series of function calls to allow types of modules to be skipped.

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

Value of "executed"	When executed	example code

0	never	`switch(6) {`
		`  case(6):`
		`  break;`
		`  default:`
		`  {{ placeholder }}`
		`  break;`
		`}`

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

condition	when condition is true	`if ({{ condition }}) {`
		`  {{ placeholder }}`
		`} else {`
		`  {}`
		`}`

not_condition	when condition is false	`if ({{ condition }}) {`
		`  {}`
		`} else {`
		`  {{ placeholder }}`
		`}`

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_save()
    else:
        recursion_or_save()

def recursion_or_save():
    if input_type is not none:
        recursive_select_complexity()
    else:
        save_test_case()

... and so forth
```

The `vtsg.py` script creates a new object of the *Generator* class. The program iterates through defined sink modules, selecting those specified by the user, see Sec. 2.1, or all of them if the user does not specify. It subsequently selects all compatible filters. 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. See 3.4.3 for details on module compatibility. When VTSG has selected a set of modules, it saves them as one test case.

When VTSG finishes generating all possible test cases, it then selects the cases to compose or selects all of them, as directed by the user; see Sec. 2.5. Then VTSG goes through the selected test cases one by one. For each one it composes the code of their modules 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, VTSG adds the "Input" `<code>` to the test case. Then it adds the "Filter" `<code>`, plus its `<flaw type>` and safety indicator, and the "Sink" `<code>`. Finally, VTSG notes the "ExecQuery" type and adds its `<code>` to the test case. VTSG then writes the test to one or more files. The location of the files is described in Sec. 2.3. Section 5.2 describes how VTSG names the files.

5.2. 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. 2.3.

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. If the complexity has a condition, COND is the id in the test condition module, see Sec. 4.8.1. Tables 6, 8, and 10 in the language appendixes list complexity IDs defined in C#, PHP, and Python. Tables 5, 7, and 9 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 `file_template.xml`, 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.

```
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. 7. Lines 16–19 instantiates complexity code from Fig. 12. Line 16 instantiates condition code from Fig. 11. Lines 18 and 19 instantiate code from the ⟨code⟩ part of Fig. 13. Line 23 instantiates critical preparation code from Fig. 9. Lines 25–39 instantiate query execution code from Fig. 10.

```
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. 13. Lines 19–27 instantiate filter code from Fig. 8.

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') [12]

The input comes from shell_commands, see specification in Fig. 7.

The filter is func_preg_match-only_numbers, Fig. 8.

The sink is select_from-concatenation_simple_quote__sql_server, Fig. 9.

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 means a `while` loop, Fig. 12, with condition 7 meaning `Math.Sqrt(42)<=42`, which always evaluates to true, Fig. 11. The second, inner complexity is 20, meaning the sink code is executed in the class body, Fig. 13.

"a" means this is the main file.

5.3. Adding New Capabilities to VTSG

VTSG 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.

New VTSG capabilities often allow the user who is describing programming languages or adding flaws to make new mistakes. For example, 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. This does not help the user.

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 the 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.
```

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.

References

[1] Black PE, Badger L, Guttman B, Fong E (2016) Dramatically reducing software vulnerabilities: Report to the White House Office of Science and Technology Policy (National Institute of Standards and Technology), NIST-IR 8151. https://doi.org/10.6028/NIST.NIST.IR.8151

[2] Black PE, Guttman B, Okun V (2021) Guidelines on minimum standards for developer verification of software (National Institute of Standards and Technology), NIST-IR 8397. https://doi.org/10.6028/NIST.NIST.IR.8397

[3] 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

[4] Black PE (2018) A software assurance reference dataset: Thousands of programs with known bugs. *Journal of Research of NIST* 123(123005):1–3. https://doi.org/10.6028/jres.123.005

[5] Delaitre A, Black PE, Cupif D, Haben G, Loembe AK, Okun V, Prono Y (2023) SATE VI report: Bug injection and collection (National Institute of Standards and Technology), NIST-SP 500-341. https://doi.org/10.6028/NIST.NIST.SP.500-341

[6] (2017) OWASP top 10 web application security risks, https://owasp.org/www-project-top-ten/. Accessed: 17 June 2020.

[7] (2020) Common weakness enumeration, https://cwe.mitre.org/. Accessed: 17 June 2020.

[8] Kuhn DR, Wallace DR, Gallo Jr AM (2004) Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, Vol. 30, pp 418–421. https://doi.org/10.1109/TSE.2004.24

[9] Kuhn DR, Kacker RN, Lei Y (2010) Practical combinatorial testing (National Institute of Standards and Technology), NIST-SP 800-142. https://doi.org/10.6028/NIST.SP.800-142

[10] Yu L, Lei Y, Kacker RN, Kuhn DR (2013) ACTS: A combinatorial test generation tool. *2013 IEEE Intern'l Conf. on Software Testing, Verification and Validation (ICST)*, pp 370–375. https://doi.org/10.1109/ICST.2013.52

[11] (2023) Combinatorial testing, https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software. Accessed: 13 June 2023.

[12] (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.

[13] Kratkiewicz KJ (2005) *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code* Master's thesis Harvard University.

Appendix A. Summary of Information for Generating C# Cases

This appendix documents the flaws, flaw groups, conditions, and complexities currently in the C# language files. It also gives instructions how to compile and run the test cases.

The following flaws are currently defined for C#. The flaw group is in parentheses.

- SQL Injection (CWE-89) (OWASP_a1)
- XPath Injection (CWE-91) (OWASP_a1)
- LDAP Injection (CWE-90) (OWASP_a1)
- OS Command Injection (CWE-78) (OWASP_a1)
- Path traversal (CWE-22) (OWASP_a4)
- Information Leak Through Error Message (CWE-209) (OWASP_a5)
- Storing Password in Plain Text (CWE-256) (OWASP_a2)
- Use of Insecure Cryptographic Algorithm (CWE-327) (OWASP_a6)
- NULL Pointer Dereference (CWE-476) (OWASP_a9)

Here are the conditions currently available to be used in code complexities. We explain condition modules in Sec. 4.8.1. Table 5 shows the ID, the code, and whether it always evaluates to true or false.

Table 5. 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

Here are the complexities currently available in C#. We explain the concept of code complexities in Sec. 2.4 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 6. 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

To compile and run C# test cases, install *mono* and *mcs*. The Mono project created *mono* as an open source platform that implements the .NET Framework. Class libraries and C# compilation are enabled by *mcs*. See the mono github repository.

Here is a command to install mono:

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

Here is a command to install mcs:

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

On a Mac, install Homebrew first. See the Homebrew home page. Then use homebrew to install mono. See directions to install mono using Homebrew.

The script `compilationTester.sh` uses mcs to compile all the C# cases that are generated.

Appendix B. Summary of Information for Generating PHP Cases

This documents the flaw, flaw group, conditions, and complexities 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 conditions currently available to be used in code complexities. We explain condition modules in Sec. 4.8.1. Table 7 shows the ID, the code, and whether it always evaluates to true or false.

Table 7. Condition IDs, code, and value to which it evaluates defined for PHP

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

Here are the complexities currently available in PHP. We explain the concept of code complexities in Sec. 2.4 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 8. 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

Appendix C. Summary of Information for Generating Python Cases

This documents the flaws, flaw groups, conditions, and complexities 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). "KK" flaws are adapted from Kendra Kratkiewicz's work [13].

- Operating system (OS) Command Injection (CWE78) (OWASP_a1)
- Path traversal (CWE22) (OWASP_a4)
- Information Leak Through Error Message (CWE209) (OWASP_a5)
- Loop on Unchecked Input (CWE606) (Other)
- Relative Path Traversal (CWE23) (Other)
- External Control of System or Configuration Setting (CWE15) (Other)
- Divide by Zero (CWE369) (Exception)
- Improper Validation of Array Index (CWE129) (Exception)
- Buffer Overflow (KK) (Exception)

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 9. 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

Here are the complexities currently defined. We explain the concept of code complexities in Sec. 2.4 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 10. 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
23	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

Appendix D. Contents of git Repository

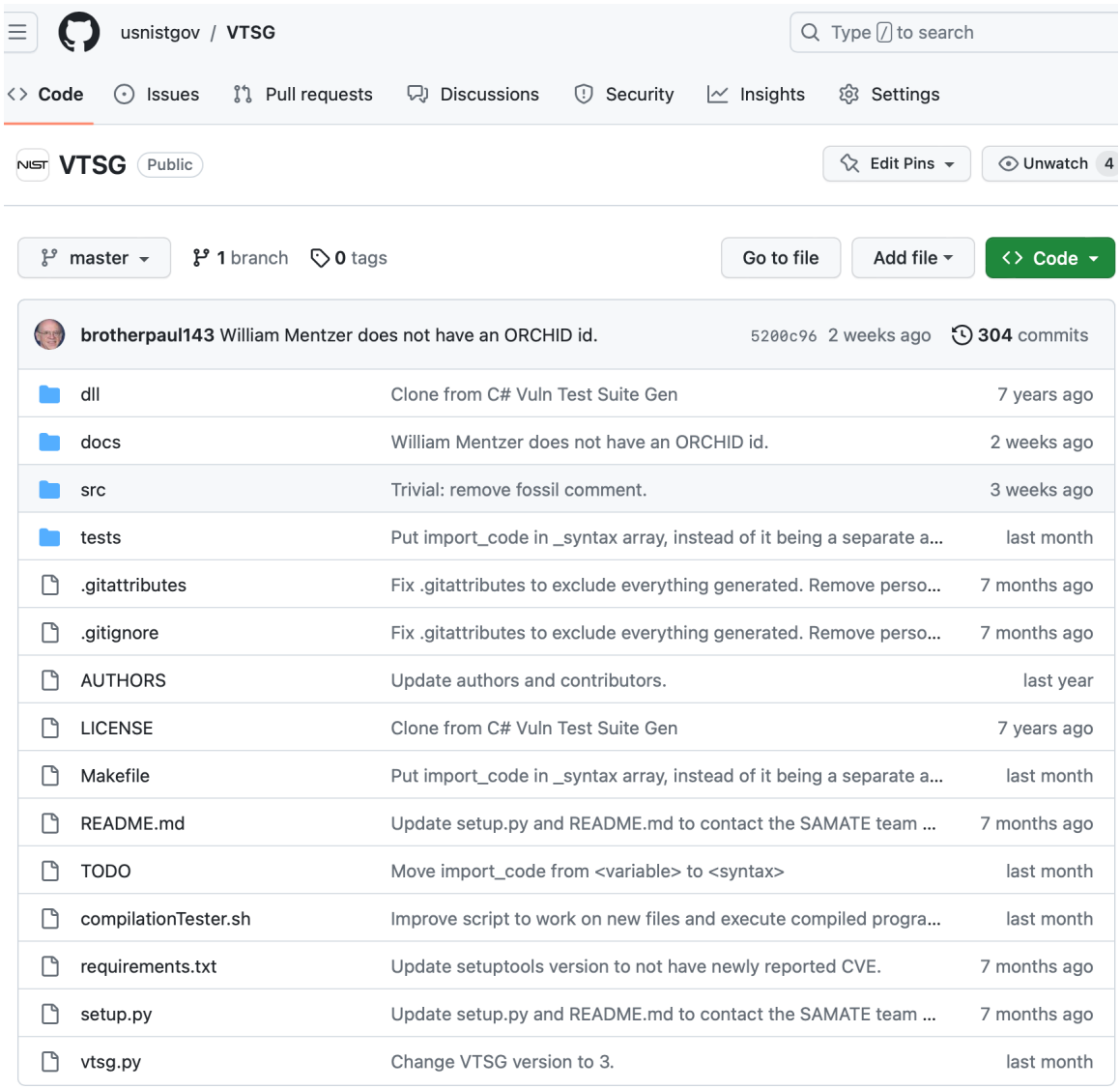

Fig. 16. Snapshot of files in the VTSG git repository, which is at https://github.com/usnistgov/VTSG, as of 31 August 2023.

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.