NISTIR 8165

Impact of Code Complexity On Software Analysis

Charles D. De Oliveira
Elizabeth Fong
Paul E. Black

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8165

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

NISTIR 8165

Impact of Code Complexity On Software Analysis

Charles D. De Oliveira
Elizabeth Fong
Paul E. Black
Software and Systems Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8165

February 2017

National Institute of Standards and Technology
Kent Rochford, Acting NIST Director and Under Secretary of Commerce for Standards and Technology

Abstract

The Software Assurance Metrics and Tool Evaluation (SAMATE) team studied thousands of warnings from static analyzers. Tools have difficulty distinguishing between the absence of a weakness and the presence of a weakness that is buried in otherwise-irrelevant code elements. This paper presents classes of these code elements, which we call "code complexities."

They have been present in software assurance as part of test cases generation strategy when evaluating static analyzers. Benefits of using code complexity include the development of coding guidelines, boosting diversification of test cases.

Keywords: *code complexity; test cases; static source code scanner; vulnerability; software assurance.*

DISCLAIMER

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

Table of Contents

1. Introduction

How can one gain assurance that software is free from vulnerabilities? One step is to use static and dynamic analysis tools. Many of the common weakness and bug classes, such as buffer overflows, cross-site scripting and failures to validate input values, can be found by such automated tools.

There are many types of static analysis tools today, both proprietary and open source. These tools analyze software and detect vulnerabilities. Although a tool may be able to detect vulnerabilities in many cases, a vulnerability may not be reported if it is surrounded by extraneous elements. The reasons vary. In order to be responsive and powerful, many tools employ sophisticated heuristics. These heuristics may be engineered to summarize or ignore data in order to minimize storage or to limit the depth and breadth of analysis to save time. Thus, these extraneous elements, which we call "code complexities," may confuse static code analyzers, making bugs harder to find. This paper presents classes of code complexity.

1.1. Background

The Software Assurance Metrics and Tool Evaluation (SAMATE) project at the National Institute of Standards and Technology (NIST) [8] team studied a broad class of software assurance tools by performing a series of experiments, known as the Static Analysis Tool Exposition (SATE) [12][9][10][11][4]. The five SATEs utilized test cases comprised of millions of lines of code. Static analysis tool makers ran their tools on test cases and gave us their outputs, covering a total of almost 4 million warnings. Although we did not have the resources to analyze all warnings, we spent many person-years determining the accuracy of thousands of warnings. By analyzing these warnings, we learned that elements that we call "code complexities" make the detection of warnings more difficult for tools.

For the SATEs, we chose test cases in C/C++, Java, and PHP. Some cases involved production software, e.g., Chrome and Wireshark. Other test cases had been generated: thousands of synthetic programs, consisting of a page or two of code, e.g., Juliet [2]. We knew the location of all of the vulnerabilities in the synthetic programs and some of the production vulnerabilities from Common Vulnerabilities and Exposures (CVE) [7] reports. Many of these test cases are publicly available in our repository, known as the Software Assurance Reference Dataset (SARD) [14]. It hosts more than 200 000 test cases in different programming languages.

1.2. Purpose of Code Complexity

In white-box assurance using static analysis tools, defect detection is limited by the capabilities and intelligence of a given tool. The Heartbleed vulnerability (CVE-2014-0160) and the GNU/libc DNS resolver vulnerability (CVE-2015-7547) revealed that static analysis tools failed to find some bugs. To improve their capabilities and increase their use, one needs a better understanding of how tools discover vulnerabilities.

1.3. Using Code Complexity to Characterize Vulnerabilities

Black et. al. created The Bugs Framework (BF): A Structured Approach to express bugs [1]. The approach seeks to better express software bugs enclosing in four main areas: *cause*, *attributes*, *consequences* and *sites*. The *attributes* refer to well defined characteristics of a specific bug, e.g. source code elements. The BF is intended to characterize general software bugs classes. In particular, the buffer overflow class contains code complexities described in this paper such as **access type** and **magnitude**.

1.4. Using Code Complexity to Generate Test Cases

Code complexities can be manually coded into test cases. However, the most beneficial use of code complexities is in the design of a test case generator capable of creating a large number of test cases. A test case generator needs to produce not only valid and vulnerable code, but also various complexities. This could be very time-consuming for a human to implement systematically. Using code complexity factors, each class of complexity can be systematically constructed and the results can be methodically reviewed.

2. Code Complexity Factors

Code complexity is a feature of the programming language that, in theory, has no impact on whether there is a vulnerability or not. They are combined and nested to create real source code. The most commonly used complexities occur in expression, control or data flow, loop structure, or memory access. Each code complexity can spawn many sub categories; some of which are language specific (e.g., use of pointers in C and C++). Categories of code complexities were explained by Wu and Boland [18].

```
1. char data;

2. data = 'C';




3. data = 'Z';
4. printLine(data);
```

```
1. char data;
2. if (1) {
3.     data = 'C';
4. } else {
5.     printLine(data);
6. }
7.
8. if (1) {
9.     data = 'Z';
10.    printLine(data);
11.}
```

Figure 1: Program fragments illustrating code complexities

To illustrate, consider the fragments of code in Figure 1. The code on the left side has a problem in line 2: the character "C" is assigned to the variable *data*, but that value is never used. It is overwritten by the assignment in line 3. Programmers should be warned about such problems. At best, it is a little extra code to understand, document, and compile. In the worst case, it may indicate a bug: that the value was supposed to be used (e.g., a new call to *printLine()* after line 2), that the value at line 4 should be "C," but the programmer

did not notice that it was overwritten at line 3, or that some other variable should have been set at line 2.

The fragment on the right hand side is functionally exactly the same, but it contains extra code that makes it more difficult for a static analysis tool (or a human!) to recognize the problem. As an extreme example, a very simplistic tool might ignore the control flow caused by the *if-else* statements altogether, notice the use of *data* on line 5, and decide that the value was (or, may have been) used.

Although this is a relatively simple case, other code complexities impose significant burdens on analysis. For example, the definition and use of a variable may be in separate files or the value may be transferred between several intermediate variables and structures before being used (or not). In theory, what we term a code complexity does not impact our reasoning regarding the presence or absence of a flaw. In both fragments, an analyst (human or automated program) must note that *data* was given a value, which was overwritten before it was ever used.

Note that code complexity differs from cyclomatic complexity, which is a measure of the number of paths through the code [17]. As used in this paper, the term refers to pieces of code that must be handled correctly to establish control and data flow which are frequently encountered. However, once flows and values are determined, it should not affect the detection of a flaw.

Code complexities can be minimal, such as the one in Figure 1. However, production code involves many code complexities which are intricately interwoven. Each code complexity can occur inside a loop or function call chains. Some complexities are specific to programming languages, such as pointers in C and C++.

2.1. Classes of Code Complexities

We describe some classes of code complexity, using the C programming language. Some code complexities in other programming languages may not be represented here. For clarity, we will illustrate each class of code complexity with an example of a classic stack buffer overflow, adapted from the baseline snippet below. In this example, the array *buffer* can store up to 5 integer numbers. The C standard defines arrays as starting at index 0. Thus, the greatest index of an array is *n - 1*, where n is the length of the array. However, our example accesses position 5 of *buffer*, which has a greatest index of 4 (5 - 1), causing an access outside its boundaries.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    ret = buffer[5];
}
```

- **Separate files:** This code complexity is based on the premise that programs may be composed of many files. In this example, the program uses File 1 and File 2.

Data flow through different functions and scopes in several files requires some tracking, similar to production code.

```
// File 1
int index() {
    return 5;
}
```

```
// File 2
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    ret = buffer[index()];
}
```

- **Control flow and loop structure:** These can be enumerated using the following code: *if*, *switch-case*, *goto*, *setjmp*, *longjmp*, function pointer, function call, *for*, *do-while*, and *while*. Our example was elaborated to add a basic *if* statement around the flawed line.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    if(0 == 0) {
        ret = buffer[5];
    }
}
```

- **Duplicate variable or function name:** This code complexity explores the same identifier referring to different objects. For example, a global variable named *buffer* may conflict with a local variable named *buffer*. In the example below *buffer* is declared twice. The first time buffer is declared with global scope as an array of 5 integers. The second time *buffer* is declared, it has local scope and ends its life at the closing curly brace following the semicolon. It might be confusing due to the fact *buffer* is being overflowed right after the local *buffer* was declared.

```
int buffer[] = {1,2,3,4,5};
void buffer_overflow() {
    int ret;
    { int buffer[] = {1,2,3,4,5,6}; }
    ret = buffer[5];
}
```

- **Container:** The data may be wrapped in arrays, *structs*, or *unions*. Our example shows *buffer* wrapped within a structure declaration (*struct wrap*).

```
struct wrap {
    int buffer[5];
};
void buffer_overflow() {
    struct wrap obj = {{1,2,3,4,5}};
    int ret;
    ret = obj.buffer[5];
}
```

- **Access by array or pointer:** A value may be accessed through an array-like construct, i.e., using square braces, or a pointer-like construct, i.e., using an asterisk. This example uses a pointer arithmetic approach to trigger a buffer overflow. This is because *buffer* + 5 indicates the fifth address of the array, and the asterisk is an access to that address.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    ret = *(buffer + 5);
}
```

- **Read or write access:** This code complexity explores whether access was simply using the value or assigning the value. The *buffer[5]* statement assigns a new value to the 5th element of *buffer*, which does not exist.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5};
    buffer[5] = 10;
}
```

- **Magnitude:** This term refers to how far outside the boundary the violation extends, in other words, how far from the first and last indexes of the allocated memory space. Our example below depicts an access far from regular off-by-one buffer overflows. This is because *buffer* is being requested to access its 5000th element, which is far from what it is able to store.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    ret = buffer[5000];
}
```

- **Alias (passed through other variables):** Aliases are values that originate in one place, but are copied or referenced through other variables. In our example, *buffer* is aliased to *ptr*, which in turn has its 5th element accessed, still causing the exact overflow as other examples presented.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret, *ptr;
    ptr = buffer;
    ret = ptr[5];
}
```

- **Unreachable and dead code:** This refers to a piece of code that logically can never be run because it is not reachable. In the case below, the *if* statement will always fail, since in C the value 0 is equivalent to false. Code in this state can be distracting or confusing to reviewers (human or automated).

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    if(0) {
        ret = buffer[5];
    }
}
```

- **Unusual syntax:** Test cases used to evaluate static analysis tools should exclude compiler test suites. Nonetheless, it may be useful to have some cases to understand how well unusual syntax is handled. Our example uses digraphs, trigraphs, and a pointer to an array (*(*)[]*), all valid statements specified by the C standard. Trigraphs and digraphs are features from earlier versions of the C language that allowed programmers to represent ASCII characters using non-ASCII keyboards. Our example presents two digraphs and two trigraphs. The first digraph, *<:*, is equivalent to the left square brace, *[*, and the second, *:>*, to the right square brace, *]*. The first trigraph, *??<*, is equivalent to the left curly brace, *{*, and the second , *??>*, to the right curly brace, *}*. The prefixes, *di-* and *tri-*, mean the number of characters each, 2 and 3, respectively. Lexical analyzers in C compilers consider these as their equivalent tokens. Perhaps, compilers may require extra options for trigraphs and digraphs to be properly evaluated.

```
void buffer_overflow() {
    int  buffer<::> = ??<1,2,3,4,5??>, ret, (*ptr)[5];
    ptr = buffer;
    ret = ptr[5];
}
```

- **Data type:** This is one of the simplest complexities, establishing the type of the main variable in the test case. The example below presents a variation of the baseline simple code, now using float as data type instead of an integer (*int*) data type.

```
void buffer_overflow() {
    float buffer[]={1.0,2.0,3.0,4.0,5.0}, ret;
    ret = buffer[5];
}
```

2.2. Other Complementary Code Complexities

The following are not code complexities as we have defined them. However, they are useful attributes to consider when thinking about test cases.

- **Fixed (unflawed) cases:** These cases are corrected versions of flawed test cases, used to identify false positive results. A trivial "analysis tool" might incorrectly flag SQL statements as sites of SQL injection. If only flawed test cases are presented,

then their lack of discrimination would go undetected. Because flaws can be fixed in a number of ways, there may be several "fixed" cases for a given "flawed" case. Our example below presents one of many possible ways to fix the overflow. Now *buffer* is being accessed in its 4th element, which is expected.

```
void buffer_overflow() {
    int buffer[] = {1,2,3,4,5}, ret;
    ret = buffer[4];
}
```

- **Undefined, unspecified, or implementation-dependent behavior:** This behavior refers to code that may yield divergent results depending on compilers and platforms. Strictly speaking, "unspecified" may cause absolutely any behavior. Test cases with such code are useful for checking a tool's capability to detect and report them. In our example, *func()* is defined as a function that does not receive parameters and does not return a value. However, further down in the code *func* is cast to *func ptr*, which is a pointer to a function that does not receive parameters but returns an integer. The C standard allows such features, so that when it comes to the line where *func ptr()* is called, the actual function being called is *func()*, and its return value is used as index to access an element from *buffer*. But as defined earlier, *func()* does not return anything, how can it return an integer? This question has multiple answers; so does the value this function returns. The C standard does not define behavior for such construction, letting it be implemented by compilers/platforms. We ran some tests using Linux 3.16, running Debian 8 x86 64, using two different compilers, GCC-4.9.2 and Clang-4.0.0. On the one hand, GCC uses a register to store a function's return value, then sets it to zero before a function call. So the call to *func ptr()* returned 0. On the other hand, Clang does not share that strategy, thus the call to *func ptr()* returns a different value every time the program runs.

```
void func(){}
int main() {
    int (*func_ptr)(), ret, buffer[]  = {1,2,3,4};
    func_ptr = (int (*)())func;
    ret = buffer[func_ptr()];
    return 0;
}
```

- **Minimal "fake" parsing:** Some tools perform complete syntactic analysis, parsing, or semantic analysis. It is easier to build tools that search ("grep") for a regular expression, such as an apparent call of the function *strcpy()* (string copy, which is susceptible to buffer overflow). Such a search would flag *strcpy()* in a comment. Examples with invalid code inform an evaluator of less-than-strict analysis.

```
void buffer_overflow() {
```

```
    char buffer[] = "hello";
    // strcpy(buffer, "world");
}
```

3. Code Complexities Applied in Academia and Industry

Section 2 depicted classes of code complexities by straightforward examples by a single factor. When applied in academic and industrial applications, code complexities are blended along weaknesses and bugs in order to create test cases for static analyzers. We collected 5 such test case generations, 4 by the government or a university, and 1 by researchers at Toyota InfoTechnology Center (ITC). We name each of these as a test suite.

Next subsections describe the test suites individually, and the methods for incorporating code complexity. Thus, the examples below contain detailed information about complexities identifiers, sometimes in file names or numeric codes. It is also important to note that all test suites described in the following subsections contain both flawed and fixed versions of test cases, except the one in Subsec. 3.3.

3.1. Juliet

The Center for Assured Software at the U.S. National Security Agency developed [2] the Juliet test suite to evaluate static analysis tools. The test suite is a collection of 86 864 C/C++ and Java programs with well characterized weaknesses (i.e., flaws or defects) of 181 different kinds. Each flaw occurs in simple code and embedded in three dozen code complexities, involving different control flow, data flow, and data types. The weakness dictionary for Juliet is the Common Weaknesses Enumeration (CWE).

Code complexities and weaknesses are injected in each test case in a manner that it is uncomplicated to interpret its file name. The weakness is represented with a CWE identifier at the beginning of every file name. There will be at least one test case serving as baseline per weakness type, free of code complexity (files ending with 01). **Control flow** complexities, e.g. *if* 's, *switch*'s, *while*'s, *for*'s, *goto*'s and functions change program flow appear in tests with endings from 2 through 22. These are intermixed with global constants and static variables. **Data flow** complexities, such as values being passed through other variables multiple times or even through a function in different files are present in tests from 31 through 84. Note that several code complexity variants are language-specific to C++ and Java and were not described in this paper.

Following is an example of a Juliet test case file name: **CWE121_Stack_Based_Buffer_Overflow_fgets_05**. It indicates a test case containing a stack-based buffer overflow (CWE-121), surrounded by an if using a static variable within its Boolean expression (ending with 05).

3.2. Taxonomy for Buffer Overflows

Kratkiewicz and Lippmann designed [6] a taxonomy of C code complexities consisted of twenty-two attributes, able to properly characterize a buffer access or overflow. Many complexities described in Sec. 2 appear in this test suite, including **read or write** access, **data type** and **magnitude**. They created four version of 291 test cases. First containing no weakness: buffer access does not cause overflow; the other three are structurally identical to the first except that they contain a buffer violation in different magnitudes: 1, 8 and 4096 bytes out-of-range, under or overflow. The four variations resulted in 1164 test cases.

Test cases have a twenty-two-digit numeric code describing complexities enclosing the weakness. The following is an example: 0000000100000153000110. In this particular case, it represents a write of a constant value to the upper boundary of a character buffer allocated in the stack; the trigger point occurs in the same scope where its memory was allocated; it was accessed directly through a primitive variable, and the site is wrapped by a do-while loop and an if statement. We do not intend to explain every detail of the test case since that is not the purpose of this paper.

3.3. STONESOUP

The Intelligence Advanced Research Projects Activity (IARPA) developed the Securely Taking On Software Of Uncertain Provenance (STONESOUP) program [3], igniting security research on software binaries from unknown provenance, or third-party software. For the final phase of the program, they created a test suite consisting of C and Java test cases. A test is an extension of an open source software (e.g., GNU/Gimp), spawning variants by intermixing a code complexity class, an injection point, and a weakness type. The code complexities were organized in four groups: taint source, data type, data flow and control flow. **Taint source** indicates an input type, choosing one from: environment variables, file contents, sockets, or shared memory. **Data type** indicates whether the main variable that triggers the vulnerability is: simple (primitive), array, void pointer, heap pointer, *struct/union*, or user defined types (e.g., *typedef* or *class*). **Data flow** indicates the variable type used to transport input data to a trigger point, some of the options are aliases, index aliases, constant addresses, variable number of arguments, buffer addresses, or Java generics (templates). **Control flow** defines language constructions that change the flow of a program: call back/recursive functions, infinite/long loops, inter class/procedures, *try-catch*'s, function pointers, macros, *goto*'s, and jumps. The complexities map to unique identifiers, later used to characterize test cases. All four groups are present in test cases, differing on the chosen code elements.

Complexities identifiers are stamped in file names, along with information regarding weakness and injection point. The following code name illustrates an instance of a STONESOUP test case: C-C120D-GIMP-06-ST02-DT04-DF05-CF12-01. The tainted source comes from a file (ST02). Data type is a heap pointer (DT04). Data flow is a constant address (DF05), passed to another function (CF12), where a buffer copy results in overflow. The first letter indicates a test case written in C, forked from a GNU/Gimp (GIMP) source tree, containing a buffer copy without checking the size of input (CWE-120). Other pieces of information from the example aforementioned are not relevant for the purpose of the paper.

3.4. PHP Test Suite

Stivalet and Delaitre [16] designed a test case generator and created 42 212 safe and unsafe PHP programs. The test cases cover the Open Web Application Security Project (OWASP) top 10 weaknesses [13], i.e. cross-site scripting, insecure direct object reference, injection, URL redirects, security misconfiguration and sensitive data exposure. The design of code complexities within a test case starts with input coming through a taint source; passed to a sanitizing step, filtering received input; and ends with the construction of the output. Basically these are small programs that receive input, process it then generate content based on inputs.

Similarly, to other suites presented in this Section, complexities of PHP test cases are present in file names. Following is a sample test case name extracted from the test suite: *CWE_601_fopen_no_sanitizing_header_file_name-concatenation_simple_quote*. The input comes from a file containing a malicious URL, read with the function *fopen*. No sanitization is performed. The tainted source was concatenated through the *header* function. Note that complexities explored in this test suite are language-specific and web-oriented.

3.5. ITC Benchmarks

The Toyota InfoTechnology Center developed [15] a test suite for evaluating static analysis tool, incorporating characteristics of automotive software. It contains approximately 30 000 lines of code, including 39 different weakness categories, mostly written in C. The test suite explores in depth undefined behaviors such as bit shifts, function casting, memory allocation on stack, and integer overflows. It is comprised of fifty different types of test cases, each type in its own file. There are eight major defect types, including memory and pointer issues, numerical defects, and race conditions. Each file contains many instances of a targeted defect in various levels of complexity. Although file names only identify weakness categories, the defects are well described in comments throughout the source code.

Complexities appear in the test cases uniformly, including **data type**, **alias**, and **undefined behavior**. Each test is enclosed in a function, receiving an identifier; exploring a complexity. Further in the file, tests pack multiple complexities combined. Although not clearly documented, ITC implements similar code complexities found in other test suites, such as Juliet and STONESOUP.

4. Conclusion

Applying characteristics of code complexity in test suites can greatly improve the evaluation of static analyzer products. Understanding code complexity can assist in the development of coding guidelines for assuring that software is fully analyzable by static analyzers.

In this paper, we sketched many of the factors making programs less analyzable. Our five SATEs showed that heuristics and other engineering constraints may prevent tools from identifying "obvious" weaknesses. A tool may find a weakness in one context, but not another context. Hence, test cases containing many occurrences of the same weakness type, surrounded by many code complexities [5], broadens the working range of a static analyzer, exploring more of its functionalities.

Some complexities were detailed in this paper, and some were referenced from earlier publications. The majority of available test suites used C, C++, C#, Java, and PHP. Both are attempts of creating small versions of programs as well as combining the realism of production code to improve the mechanical analysis of tools, representing the needs of organizations willing to make investments in tools.

Future research involves refining and incorporating this code complexity element in the design and selection of test cases for the next static analysis tool exposition.

5. References

[1] I. Bojanova, P. E. Black, Y. Yesha, and Y. Wu. The Bugs Framework (BF): A Structured Approach to Express Bugs. 2016 IEEE International Conference on Software Quality, Reliability and Security, 2016. https://doi.org/10.1109/QRS.2016.29

[2] T. Boland and P. E. Black. The Juliet 1.1 C/C++ and Java Test Suite. IEEE Computer society, Computer, pages 82–84, 2012. http://doi.ieeecomputersociety.org/10.1109/MC.2012.345

[3] C. De Oliveira and F. Boland. Real World Software Assurance Test Suite: STONESOUP. In Proceedings of software Technology Conference, STC '15, October 2015.

[4] A. Delaitre and et al. Report on Static Analysis Tool Exposition (SATE) V. To be published. Available at https://samate.nist.gov/SATE.html.

[5] A. Delaitre, B. Stivalent, E. Fong, and V. Okun. Evaluating Bug Finders, Test and Measurement of Static Code Analyzers. In Proceedings of the 1st International Workshop on Complex Faults and failures in Large Software Systems, COUFLESS' 15, pages 14–20. IEEE, May 2015.

[6] K. Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's thesis, Harvard University, Cambridge, MA, USA, 2005.

[7] MITRE. Common Vulnerabilities and Exposures. https://cve.mitre.org/, 2016.

[8] NIST. Software Assurance Metrics and Tool Evaluation (SAMATE). http://samate.nist.gov, 2016.

[9] V. Okun, A. Delaitre, and P. Black. The Second Static Analysis Tool Exposition (SATE). NIST Special Publication 500-287, 2010. Available at https://samate.nist.gov/SATE2009.html.

[10] V. Okun, A. Delaitre, and P. Black. Report on the Third Static Analysis Tool Exposition (SATE). NIST Special Publication 500-283, 2011. Available at https://samate.nist.gov/SATE2010.html.

[11] V. Okun, A. Delaitre, and P. Black. Report on Static Analysis Tool Exposition (SATE) IV. NIST Special Publication 500-297, 2013. Available at https://samate.nist.gov/SATE4.html.

[12] V. Okun, R. Gaucher, and P. Black. Static Analysis Tool Exposition (SATE). NIST Special Publication 500-279, 2009. Available at https://samate.nist.gov/SATE2008.html.

[13] OWASP. The ten most critical web application security risk. https://www.owasp.org/index.php/Top 10 2013-Top 10, 2013.

[14] SAMATE. Software Assurance Reference Dataset. http://samate.nist.gov/SARD, 2016.

[15] S. Shiraishi, V. Mohan, and H. Marimuthu. Quantitative Evaluation of Static Analysis Tools. In Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW '14, pages 96–99, Washington, DC, USA, 2014. https://doi.org/10.1109/ISSREW.2014.62

[16] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), ICST '16, pages 409–415. (IEEE Computer Society, Washington, DC), April 2016. https://doi.org/10.1109/ICST.2016.43

[17] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235, 1996. Available at http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

[18] Y. Wu and F. Boland. Categorizing Code Complexities in support of analysis. In Proceedings of the 11th International Conference on Cyber Warfare and Security, ICCWS 2016, pages 17–18, Boston, MA, USA, 2016.