NIST Internal Report
NIST IR 8561

The Software Assurance Reference Dataset (SARD)

Paul E. Black

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8561

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

NIST Internal Report
NIST IR 8561

The Software Assurance Reference Dataset (SARD)

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.8561

January 2025

U.S. Department of Commerce
Jeremy Pelter, Acting Secretary of Commerce

National Institute of Standards and Technology
Craig Burkhardt, Acting Under Secretary of Commerce for Standards and Technology and Acting NIST Director

Publication History
Approved by the NIST Editorial Review Board on 2025-01-29

How to cite this NIST Technical Series Publication:
Paul E. Black (2025) The Software Assurance Reference Dataset (SARD). (National Institute of Standards and Technology, Gaithersburg, MD), NIST IR 8561. https://doi.org/10.6028/NIST.IR.8561

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

Abstract

The Software Assurance Reference Dataset (SARD) has over 450 000 buggy programs in five languages covering more than 150 classes of weaknesses. We describe the principles of the collection and the very diverse content. We explain how to search and access the programs and the rich documentation and web pages associated with the SARD.

Keywords

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

Table of Contents

1. Introduction 1
1.1. Attributes of a Good Test Suite 3
1.2. Attributes of SARD Test Cases 4
1.2.1. Review Status 5
1.2.2. Case Versions 6
2. SARD Contents 7
2.1. Cases From Production Software 7
2.2. Manually Written Test Cases 9
2.3. Generated Cases 10
2.4. Documentation 11
3. Using the SARD 11
3.1. Searching for Test Cases 11
3.2. Downloading Test Cases and Test Suites 12
3.3. Using the Application Programming Interface (API) 14
References 14
Appendix A. Web Pages Associated with the SARD 17
A.1. SARD Acknowledgments and Test Suites Descriptions 17
A.2. Non-NIST Publications on SARD Content 19
A.3. Other Assurance Tool Test Collections 21
A.4. Other SARD Web Pages 22
Appendix B. Details of Figure 1 22

Acknowledgments

We thank Yann Prono and Bapiste Chocot for maintaining the SARD, both the content and web service software, for many years.

1. Introduction

One can assess some of the strengths and limitations of software assurance tools using a corpus of programs with known attributes and bugs. The software engineer can run the assurance tool on a selection of cases from the corpus to estimate the kinds of bugs the tool finds and does not find, false positive rates, and the software constructs it handles. As of April 2024 the Software Assurance Reference Dataset [1] has over 450 000 programs in C, C++, Java, PHP, and C# covering more than 150 classes of weaknesses. Most of these are synthetic programs of a page or two of code, but there are over 170 full-sized production programs and 7770 production variations derived by injecting bugs into a dozen full-sized applications.

We use "SARD" to mean two different things. First, the collection of programs that can be used as test material. Second, the service program and website that searches for and displays test cases and displays test suites and other material. The SARD website is https://samate.nist.gov/SARD.

We explain goals and organization, then in Sec. 2 describe the very diverse content of the dataset. Section 3 explains a little about accessing SARD test cases. Appendix A documents the current content of web pages associated with the SARD, which are acknowledgments and collection descriptions, citations, other test collections, and other documentation.

The SARD collection can be viewed as test cases, most grouped into test suites, and some related documentation. Test cases are programs with three kinds of origins. First are production programs such as the Apache Hypertext Transfer Protocol (HTTP) Server, GNU Image Manipulation Program (GIMP), grep, and Wireshark. Next are production programs with bugs injected into them. Last are synthetic programs written specifically for testing purposes. Most synthetic programs are generated by programs, but some are written by hand.

Figure 1 depicts the quantity, size, and origin of cases in each language. Details of the figure are in App. B. It is an updated version of Fig. 1 in [2]. Note that production cases, in green, are plotted double height and extend above and below the language center line. More details of the content, organization, and use of the SARD are in [3].

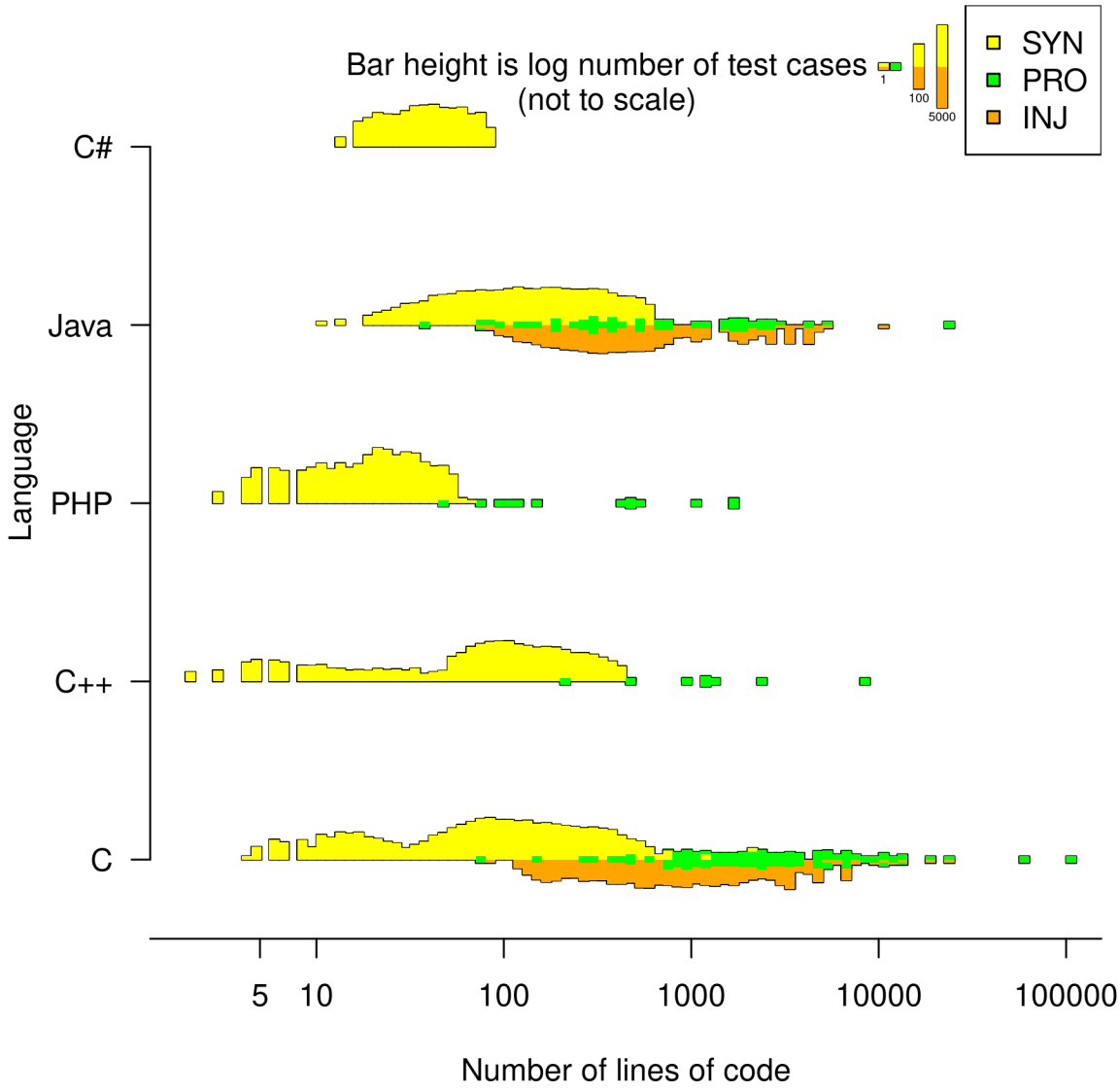

Fig. 1. Number of test cases in the SARD by language and size as of March 2024. The horizontal (X) axis is the number of lines of code plotted on a logarithmic scale. The height of each bar is proportional to the logarithm of the number of test cases with that many lines of code. Synthetic cases (SYN), which are created to be test cases, are yellow. Cases of production code (PRO) are green and are double height above and below the language center line. Cases with weaknesses injected (INJ) into production code are orange.

Some Terminology

Terms such as "bug" or "flaw" are ambiguous. When examining a few lines of code, we may see a potential buffer overflow or command injection. However, taking a broader view, we may find that the relevant variables are filtered or that the input only comes from

a reliable source, therefore those lines of code are not exploitable. It may be difficult to even determine if the code is reachable [3]. Therefore, we use weakness, flaw, or bug to mean a "defect in a system that may (or may not) lead to a vulnerability." [4, pg. 12] "A *vulnerability* is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure. ... A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation." [5, Sec. 1.4] Since few of these definitions are universally accepted, we do not slavishly require any one definition of error, fault, flaw, weakness, failure, and so forth in the SARD.

Background

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) group created this collection in 2005 and originally called it the Standard Reference Dataset, abbreviated SRD [6]. The name was quickly changed to Software Reference Dataset to be more specific, then to SAMATE Reference Dataset to be less presumptuous.

In 2013, we learned that the abbreviation SRD conflicted with the federally legislated Standard Reference Data (SRD). About the same time, we came to desire a more distinct acronym so that web searches were easier. Several rounds of soliciting suggestions, brain storming, polling, and discussion produced over two dozen possibilities. The new name, Software Assurance Reference Dataset, was suggested by Bertrand Stivalet and was announced at the Static Analysis Tool Exposition (SATE) V Experience Workshop on 14 March 2014.

1.1. Attributes of a Good Test Suite

The purpose of the SARD is to provide researchers, software assurance tool developers, and end users a set of artifacts with known software weaknesses. Figure 2 illustrates the three primary attributes that make the artifacts useful. First, the code is realistic, that is, similar to production code. Second, we know the location and type of all bugs. Third, the collection has a large enough volume of a variety of representative bugs in various code constructs and contexts to infer method performance.

Software whose source code that anyone can inspect, so-called "open source" software, is a resource for production software with enough realistic code, but we don't know all of the bugs. We know what each kind of bug is in some synthetic, generated suites, like Juliet (see Sec. 2.3) and there are mountains of cases, but the code is far from what is usually written and used. Finally, production code with Common Vulnerabilities and Exposures (CVE) reports has known bugs. However, we find there are relatively few cases, which also lack variety. To approach an ideal suite, we have experimented with injecting bugs into production code to create many cases of realistic code with a variety of known bugs [4, Sec. 2].

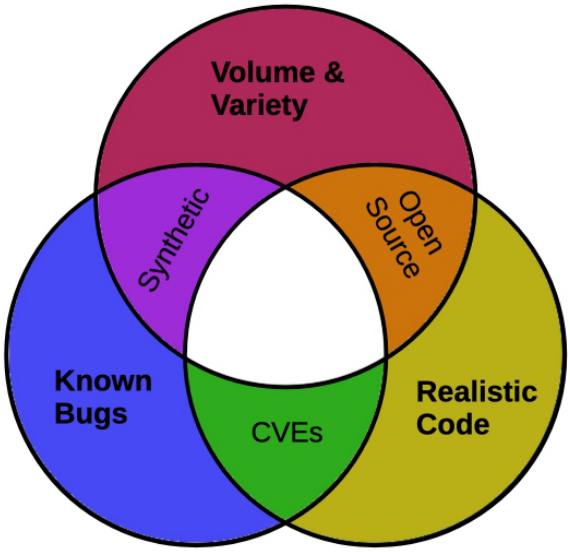

Fig. 2. The best test suite has three attributes: realistic code, known bugs, and large volume and variety. Production software, synthetic cases, and software with bugs identified through Common Vulnerability and Exposures (CVE) reports each have only two attributes. An ideal test suite has all three attributes, that is, located in the central white bulging triangular region. After [7].

1.2. Attributes of SARD Test Cases

A test case consists of one or more files, which express the weakness, and metadata about the file(s), such as weakness types and locations, language, etc.

Code is typical quality. It is not necessarily pristine or exemplary. Nor is it horrible. The SARD is not a compiler test. Test cases that are absolutely the cleanest, most excellent code with great style, except for the weakness, minimizes confounding concerns. These may be useful for basic research and instruction. But SARD test cases may have style faults, non-portable code, "code smells", and extraneous weaknesses.

Some test cases are as small as a few lines of generated code, which clearly represent a vulnerability. SARD cases range in size up to entire applications, to represent production code and to allow investigation of scaling.

The SARD software can handle binary (executable) test cases, in addition to source code. Currently, all cases are source code.

Cases are never removed from the SARD or changed, although newer versions may be added. This permanence allows research work to refer to, say SARD test case 1552, knowing that that exact code can always be retrieved. Later work may thus begin by replicating previous work.

Many cases with intentionally-flawed code have corresponding cases with that code fixed.

The corresponding cases are important to test false-positive rates. Sometimes there is more than one way to fix a flaw, so multiple fixed cases are possible. All cases have a *state*. The state is "bad", that is, it contains code intentionally having flaws, "good", it contains no (intentionally) flawed code, or "mixed", meaning it contains both flawed code and matching fixed code.

Each test case can have a variety of metadata. The metadata includes the kinds and locations of weaknesses, programming language, who contributed it and when, hints as to its quality or suitability, and complementary test cases. Although not every case has metadata indicating every vulnerability, the vast majority of weaknesses are noted in the metadata, which can be processed automatically.

Some synthetic test cases occur in complementary groups. For example, a "bad" test case, that is, a test case with a weakness, may have a complementary "good" test case in which the weakness is corrected. Pairs of good and bad test cases may help determine false positive rates of tools. One good case may be related to several bad cases. An example is in Kratkiewicz's suite in which each good case has three bad cases with overflows just outside, moderately outside, and far outside the buffer boundary. See Sec. 2.3 for more details. Some cases in the Juliet suites have more than one good function for each bad function.

All test case metadata is available in a *manifest* file, which uses Static Analysis Results Interchange Format (SARIF) [8] format. The manifest has information such as the Uniform Resource Locator (URL) to download the test case, test case number and version, kinds and locations of weaknesses, programming language, state, status, creation date, and author. The manifest for the entire SARD is over one gigabyte. It is available at https://samate.nist.gov/SARD/downloads/sarifs.json. Its schema is available through the link `sard-schema.json` at https://samate.nist.gov/SARD/documentation#resources.

1.2.1. Review Status

Every case has a SARD *status* giving some information about how well it has been reviewed: candidate (C), accepted (A), or deprecated (D). Test cases are "candidates" when they are added to the SARD. This means they have not been thoroughly reviewed, yet. Cases are marked "deprecated" when they are found not to be suitable for new work. This may happen if an unintended significant weakness is found. Often deprecated cases are replaced by new versions.

An "accepted" test case is one that meets our documentation, correctness, and quality requirements. If a case has a status of "accepted", you can expect that the source code will

- compile (for compilable languages) or run without fatal errors (for complete interpreted languages),
- run without fatal error messages, other than those expected for an incomplete program (if it is just functions or modules),

- not produce any warnings, unless they are expected as part of the test,
- contain the documented weakness if the test case is a bad or mixed test case, and
- contain no known weaknesses at all if it is a good test case.

In addition, an accepted case will have the following documentation:

- a description of the purpose of the case,
- the author's name,
- if it is submitted by someone other than the author, the contributor's name,
- its language, e.g. C, C++, Java, or PHP,
- its type, e.g., "source code" or "binary".
- if it is "bad or "mixed", the locations of flaw(s), and
- any instructions to compile, analyze, or execute the test. This may include compiler name/version, compiler directives, environment variable definitions, inputs that prove the vulnerability exists, or other test context information. Any drivers, stubs, include files, declarations, "make" files, and so forth are readily available, too.

The test case description may include the kind of weakness(es), contributing weaknesses in case of a chain or composite, and code complexities. Typical code complexities are loops, inter-procedural data flow, buffer aliasing, and use of structures. "In theory only the code pertaining to the weakness need be examined to determine that it is, indeed, a weakness. However, analysis tools must handle an unbounded amount of surrounding code to find sources of sinks [or] determine conditions when the code is executed" [3]. Some cases are a copy of a base application with an injected weakness, while many synthetic programs have a base weakness wrapped in different code complexities or using different data types.

Originally we planned to thoroughly assess every test case for suitability and quickly accept candidate cases. We found that resources are scarce and that the added work to certify a case as accepted was rarely worth the effort.

1.2.2. Case Versions

Cases have versions. The initial version number of a test case is 1.0.0. The highest numbered version is the default.

If the source code of a test case is changed significantly, we increase the major number (e.g., 1.0.0 $\rightarrow$ 2.0.0). Typical changes are to fix an extraneous error or update for a revised programming language standard, e.g., C99 $\rightarrow$ C11. Even correcting a function name should change the major number, because a small code change can result in a big change for software analysis tool reports. The major number does *not* increase if the Abstract Syntax Tree (AST) remains the same. That is, changes in formatting, white space, or comments only change the minor number.

Increase the minor number (e.g., 1.0.0 $\rightarrow$ 1.1.0) if there are significant changes to the metadata, that is, something that appears in the manifest file. Significant information includes

weakness identification (Common Weakness Enumeration [9] or CWE number), line numbers, or file name. When a new minor version is released, SARD users should compare the new SARIF file with what they are working on.

Increase the patch number (e.g., 1.0.0 → 1.0.1) for all other kinds of changes, for example, documentation, README.md, Dockerfile, or compilation instruction. A new patch version should have no impact on research results based on SARD test cases.

2. SARD Contents

This section describes many of the collections that constitute the SARD. Much of this section is adapted from or quotes "SARD: Thousands of Reference Programs for Software Assurance" [3]. This article was originally published in the CSIAC Journal of Cyber Security and Information Systems Vol. 5, No 3.

The SARD has benefited from many contributors. We appreciate them and acknowledge their work and generosity. These test cases represent considerable intellectual effort to reduce reported vulnerabilities to examples, classify them, generate elaborations of particular flaws, come up with corresponding correct examples, etc. Many of the contributions are credited on the Acknowledgments and Test Case Descriptions page [10]. The acknowledgments also have more details about the test cases, their sources, links to paper explaining them, and other information. The content of the Acknowledgment page as of August 2024 is in App. A.1.

In this section we first describe cases from production software. This includes cases created by injecting flaws into production software. Next in Sec. 2.2 we describe test suites written for test purposes. Then we describe suites generated by programs in Sec. 2.3. Each description includes how to access just that suite or collection of cases. The last subsection explains some special documentation provided with the SARD.

2.1. Cases From Production Software

Collecting production code allows us to present actual bugs that are found in realistic software. We located bugs in several ways:

- Inject bugs into code,
- Use information from bug reports, such as Common Vulnerability and Exposures (CVE) or the National Vulnerability Database (NVD), https://nvd.nist.gov/,
- Compare earlier versions of the software with later versions to see what code changed to fix a bug, or
- Debug — test the program, then scan and analyze the code.

For Static Analysis Tool Exposition (SATE) IV [11], SAMATE team members tracked vulnerabilities reported as Common Vulnerability and Exposures (CVE) [12] back to source code changes. This resulted in 228 CVEs located in Asterisk, Chrome, Dovecot, Wireshark

(1.2 and 1.8), Apache Tomcat, Jetty, JSPWiki, Openfire, and WordPress. Each of these has its own test suite, 90 through 99. Each CVE has one test case containing the file or files with the vulnerability. The first case in each suite has all the CVEs, files, and identified vulnerabilities for that program. These 10 test suites represent hundreds of reported vulnerabilities and the corresponding source code.

MIT Lincoln Laboratory extracted 14 program slices from popular Internet applications, BIND, Sendmail, and WU-FTP, with exploitable buffer overflows [13]. That is, they removed all but a relatively few functions, data structures, behaviors, and so forth making sure the remaining code still has the overflow. They also made "good" (patched) versions of each slice. These 28 test cases are test suite 88.

The Intelligence Advanced Research Projects Activity (IARPA) Securely Taking On New Executable Software Of Uncertain Provenance (STONESOUP) program created test suites in three phases. STONESOUP documents are available at https://samate.nist.gov/SARD/documentation#iarpa. For Phase 1 they developed five collections of small C and Java programs covering five vulnerabilities. Each collection may be downloaded from the SARD Test Suites page [14] and includes directions on how to compile and execute them and inputs that trigger the vulnerability. The test cases for Phase 2 were not particularly different from the Phase 1 cases, so were not saved.

For Phase 3, STONESOUP injected thousands of weakness variants into 16 widely-used web applications, resulting in 3188 Java cases and 4582 C cases. "Although the base programs were real-world software, the inserted code snippets, or cysts, were unrelated to the control and data flow of the base programs. The resulting weaknesses were not representative of bugs made by real programmers." [4] The weaknesses covered 25 classes. Each case is accompanied with inputs triggering the vulnerability, that is proof of vulnerability or POV, as well as "safe" inputs. Because the cases represent thousands of copies of full-sized applications, STONESOUP Phase 3 is distributed as a virtual machine with a complete testing environment, test suite 113.

The STONESOUP base applications are significant enough by themselves that we describe them here. They are available from the Test Suite page as Standalone applications, test suites 3 through 18 and 21. Fifteen applications are GNU grep, OpenSSL, PostgreSQL, Tree (a directory listing command), Wireshark 1.10.2, Coffee MUD (Multi-User Dungeon game), Elastic Search, Apache Subversion, Apache Jena, Apache JMeter, Apache Lucene, POI (Apache Java libraries for reading and writing files in Microsoft Office formats), FFmpeg (a program to record, convert, and stream audio and video), Gimp (GNU Image Manipulation editor), and Open Fire. Two additional collections from STONESOUP are Wireshark 1.8.0, suite 22, and Asterisk, suite 19. The 16th application is JTree, which is different from the others. It is a single base case injected with weaknesses. When processed with unzip, it produces 34 subdirectories, each with difference files to create a version of JTree with an injected weakness.

For SATE VI, SAMATE team members began with widely-used production software. In

most instances they began by tracking down and documenting reported bugs. They also injected bugs into some of the test cases. Bug injection seeks to place many known vulnerabilities in production software, thus getting closer to the best test cases illustrated in Fig. 2. A thorough treatise on bug injection, including its potential benefits, reviews of bug injection efforts, and its use in SATE VI is in Sec. 2 of [4]. Section 2.3 details bug injection and the collections used in SATE VI. As it explains, "Bugs can be injected with different degrees of automation. ... bug quality tends to degrade with an increase in automation. In SATE VI, we used different degrees of automation with successes and pitfalls." Section 5.2 details issues with automatically injected bugs.

Following is a summary of the SATE VI collections and the approaches used to prepare them. In WireShark 1.2.0 and DSpace they tracked down and documented reported bugs then injected others. They injected bugs into Sakai and SQLite. The final case came from the 2016 Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge (CGC). It consists of "custom-made programs specifically designed to contain vulnerabilities that represent a wide variety of crashing software flaws. ... they approximate real software with enough complexity to stress both manual and automated vulnerability discovery." [4, Sec. 2.3.6]. The CGC case was ported by Trail of Bits. These collections are not yet available through the SARD. Until they are, the SATE VI Classic Track page has links to download them. https://www.nist.gov/itl/ssd/software-quality-group/sate-vi-classic-track. We are working on these collections to add them to the SARD.

2.2. Manually Written Test Cases

Many companies donated synthetic benchmarks that they developed manually. Fortify Software, Inc., contributed C programs that manifest various software security flaws. They updated the collection as ABM 1.0.1. These 112 cases cover various software security flaws, along with associated "good" versions. These are test suite 6. In 2006, Klocwork Inc. shared 41 C and C++ cases from their regression suite. These are all a few lines of code to demonstrate use after free, memory leak, use of uninitialized variables, etc. They are suite 106. Toyota InfoTechnology Center (ITC), U.S.A. Inc. created a benchmark in C and C++ for undefined behavior and concurrency weaknesses. The test suite, 104, has 100 test cases containing a total of 685 pairs of weaknesses. Each pair has a version of a function with a weakness and a fixed version of the function. For more details see [15]. The test cases are ©2012–2014 Toyota InfoTechnology Center, U.S.A. Inc., distributed under the "BSD License," and added to the SARD by permission. The SAMATE team noted coincidental weaknesses.

The SARD also includes 329 cases from our static analyzer test suites [5]. These have suites for weaknesses, false positives, and weakness suppression in C++ (test suites 57, 58, and 59), Java (63, 64, and 65), and C (100 and 101). The development of suites 100 and 101 from suites 45 and 46 and their validation is thoroughly detailed in [16].

The SARD includes many small collections of synthetic test cases from various sources.

Frédéric Michaud and Frédéric Painchaud, Defence R&D Canada, created and shared 25 C++ test cases. These test cases cover string and allocation problems, memory leaks, divide by zero, infinite loop, incorrect use of iterator, etc. These are test suite 62. Robert C. Seacord contributed 69 examples from "Secure Coding in C and C++" [17]. John Viega wrote "The CLASP Application Security Process" [18] as a training reference for the Comprehensive, Lightweight Application Security Process (CLASP) of Secure Software, Inc. The SARD initially included 36 cases with examples of software vulnerabilities from use of hard-coded password and unchecked error condition to race conditions and buffer overflow. Many of the original cases have been improved and replaced.

Hamda Hasan contributed a total of 18 cases in C#, including ASP.NET, with XSS, SQL injection, command injection, and hard coded password weaknesses.

2.3. Generated Cases

By far the largest number of test cases are synthetic generated by special programs.

One of the first SARD collections came from Massachusetts Institute of Technology Lincoln Laboratory. Kendra Kratkiewicz developed a taxonomy of code complexities and 291 basic C programs representing this taxonomy to investigate static analysis and dynamic detection methods for buffer overflows. Each program has four versions: a "good" version, accessing within bounds, and three "bad" versions, accessing just outside, moderately outside, and far outside the boundary of the buffer. These 1164 cases are explained in Kratkiewicz and Lippmann [19] and are test suite 89.

In 2011, the National Security Agency's Center for Assured Software (CAS) generated thousands of test cases in C/C++ and Java covering over 100 CWEs, called Juliet 1.0. They can be compiled individually, in groups, or all together. Each case is one or two pages of code. They are grouped by language, then by CWE. In each CWE, base programs are elaborated with up to 30 variants having complexities added. They soon updated to version 1.1 [20, 21]. The next year, they extended the collection to Juliet 1.2 [22, 23], which comprises 61 387 C/C++ programs and 25 477 Java programs in almost two hundred weakness classes.

The latest version of Juliet is 1.3. We corrected about a dozen systematic problems and added pre- and postfix increment overflow and decrement underflow cases to C/C++ and Java suites [24]. That also details known problems remaining. The updated suites are 111 for Java and 112 for C/C++. Both collections are available with extra support, like Makefiles, as test suites 109, Java, and 116 C/C++. CAS added 28 881 test cases in C# covering 105 CWEs to Juliet 1.3. They are available as test suite 110.

It is tempting to use the Juliet suites to train machine learning systems. If this is attempted, it should only be done with the greatest care. Nascimento points out [25, Sec. 5.1.1.1] several limitations of Juliet, such as

1. High correlation between function names and important code attributes. "... the model may rely on [names] to classify the methods rather than effectively learning the underlying vulnerabilities."
2. Effectively duplicate instances. All Juliet cases are strikingly similar in structure, and the variations in data types and conditions may only give the illusion of diversity.

Following an architecture developed by NIST personnel and under their direction, a team of students at TELECOM Nancy, a computer engineering school of the Université de Lorraine, Nancy, France, implemented a generator for PHP cases. They created a suite of 42 212 test cases in PHP covering the most common security weakness categories, including XSS, SQL injection, and URL redirection. These are suite 103 and are documented in Stivalet and Fong [26]. In 2016, SAMATE members oversaw additional work, again by TELECOM Nancy students, who created a suite of 32 003 cases in C#. These cases are suite 105. The Vulnerability Test Suite Generator (VTSG) has been enhanced to version 3, which produces test cases in Python [27] in addition to other improvements.

Felix Schuckert et. al. wrote a program that constructs cases structurally similar to those produced by VTSG. The program uses Abstract Syntax Trees (AST) and decision trees internally. Details are in [28]. They generated two test suites addressing data flow, test suite 114, and soundness, test suite 115, for XSS and SQLight. The dataflow test suite has 248 592 cases.

2.4. Documentation

Some suites have extended documentation associated with them. The SARD also makes this available through a special page. The Documentation page, at https://samate.nist.gov/SARD/documentation, begins with information about the SARD, including a link to a SARD manual, a link to a SARIF validator, our theory of test case versioning, and explanation of search filters.

The next part of the page is a repository for all the documents we host related to the IARPA STONESOUP program. The last part has Juliet documents.

3. Using the SARD

We hope that using the SARD web interface is intuitive. This section has some points that may be helpful.

3.1. Searching for Test Cases

The search finds test cases with specific features. Click on "Test Cases" in the top navigation bar. This takes you to a test cases display page with default result. You can search by criteria such as test case id, test case description, language, weakness type, string in the source file, etc. See Table 1 for details.

Enter the search criteria in the "Filters" box, then click "Search". Each filter is a key, a colon (:), and a value.

Putting the cursor in the "Filters" text box causes a drop-down selection box to appear that has all possible key/value pairs (except "q" for arbitrary text search of metadata).

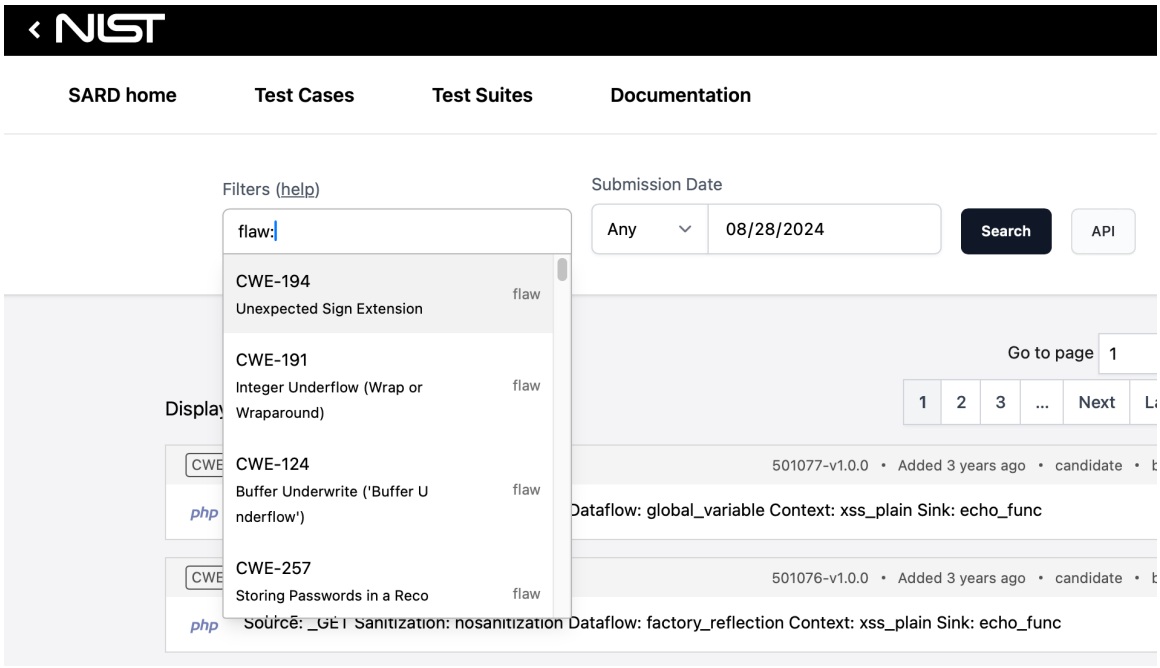

Fig. 3. The SARD "search bar" showing the filter options drop-down.

Search for a test case just by entering its number, for example "2079". (You will see the software use q:, the string search filter.) To get a certain version of a test case, search for that test case, click on the test case to display it, then select the version from the drop-down menu box.

Selected cases are the intersection ("AND") of all filters. Values of the same criteria are in a union ("OR"). Different criteria are in an intersection ("AND"). For example,

```
flaw: CWE-120  flaw:CWE-191  language:c
```

returns test cases that are in the C language *and* either CWE-120 or CWE-191.

To search by the date of submission, select "Before", "On", or "After", then the date. Select month, day, or year by clicking on each one and entering the value or by clicking in the calendar pop up. See Fig. 4.

3.2. Downloading Test Cases and Test Suites

Clicking a test case ID displays that test case. (You can select a previous version of the test case in the drop-down menu.) Click the down arrow ( ).

Table 1. All search criteria with examples.

Criterion	Example
q: ANY STRING	`q: sql injection` matches cases whose metadata contains "sql injection".
author: AUTHOR	`author: MIT` matches cases created by MIT.
cve: CVE	`cve: CVE-2014-0160` matches cases related to that CVE.
language: LANG	`language: java` matches cases written in Java.
state: STATE	`state: good` matches good cases; details in Sec. 1.2.
status: STATUS	`status: deprecated` matches deprecated cases; details in Sec. 1.2.1.
flaw: FLAWNAME	`flaw: CWE-121` matches cases labeled CWE-121.
file: BASENAME	`file: CWE121_Stack_Based_Buffer_Overflow__CWE129_large_14.c` matches cases containing the given file. The file name must be the basename, that is, without any leading directory components.
operating system: CPE	`operating system: cpe:2.3:o:microsoft:windows:*:*:*:*:*:*:*:*` matches cases created for the Windows platform. The value must be a Common Platform Enumeration [29]
application: CPE	`application: cpe:2.3:a:ffmpeg:ffmpeg:1.2.2:*:*:*:*:*:*:*` matches cases using FFmpeg version 1.2.2. The value must be a Common Platform Enumeration [29]

You can download any particular test case by accessing the URL https://samate-internal.nist.gov/SARD/downloads/versions/NNNN-vV.V.V.zip, where NNNN is the test case ID and V.V.V is the version number. For example, the latest version of test case 248320, which is 2.0.0, can be downloaded with https://samate-internal.nist.gov/SARD/downloads/versions/248320-v2.0.0.zip.

There is no method in the SARD for users to download sets of test cases.

It takes effort to find and download many cases. One possibility is to use the Application Programming Interface (API), as indicated in Sec. 3.3. Another approach is to download all the information about test cases, then select desired test cases and download them.

All test case metadata information is available in one huge *manifest*. For more information, see Sec. 1.2.

Test suites can be downloaded by clicking on the download arrow on the right of each test suite in the Test Suites page https://samate.nist.gov/SARD/test-suites. Each test suite also has a Download button on its own page.

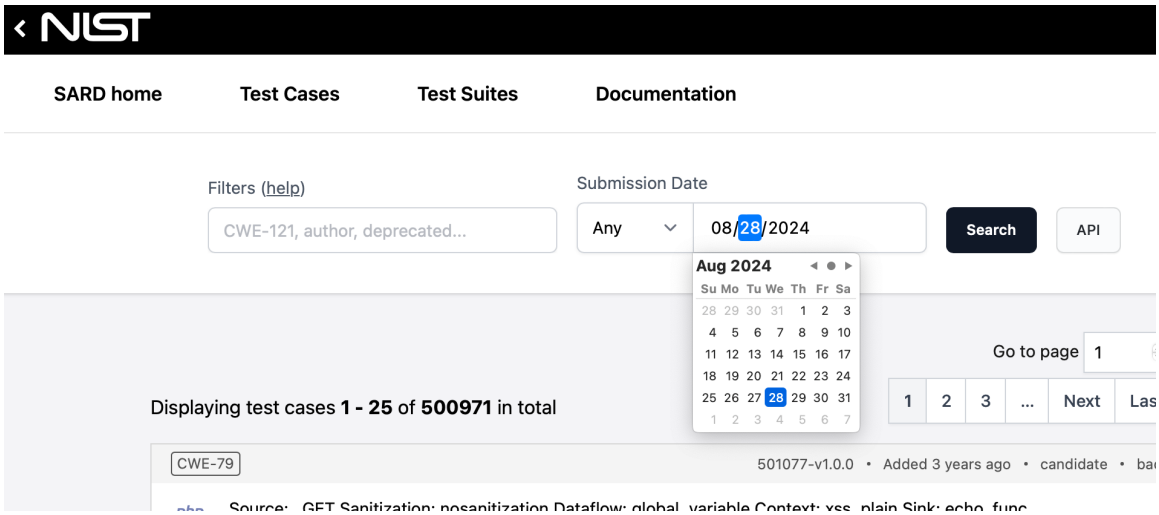

Fig. 4. The SARD "search bar" showing the "Filters" box, the link to search "help" the "Submission Date" filter with the date select drop-down, and the "Search" button.

3.3. Using the Application Programming Interface (API)

Instead of trying to anticipate every possible use of the SARD, we provide an API. The URL of the API is https://samate.nist.gov/SARD/api/test-cases/search. Pass parameters as GET parameters. Parameters are described at https://samate.nist.gov/SARD/documentation#search. The API URL, or the API button on the search page, returns search results in a JavaScript Object Notation (JSON) format. With this API, anyone can write other tools by submitting curl commands, for example a page that displays statistics about test cases.

The following example retrieves the second page of 50 test cases with CWE-194:

https://samate.nist.gov/SARD/api/test-cases/search?flaw%5B%5D=CWE-194&page=2&limit=50

The following example retrieves the first 25 test cases with both CWE-191 and CWE-194:

https://samate.nist.gov/SARD/api/test-cases/search?flaw%5B%5D=CWE-191&flaw%5B%5D=CWE-194&limit=25

References

[1] Software Assurance Reference Dataset (SARD). Accessed 25 April 2024. Available at https://samate.nist.gov/SARD/.

[2] 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

[3] Black PE (2017) SARD: Thousands of reference programs for software assurance. *CSIAC Journal of Cyber Security and Information Systems* 5(3):6–13. Special Software Assurance edition – DoD Software Assurance (SwA) Community of Practice: Design and Development Process for Assured Software – Volume 2: Tools & Testing Techniques.

[4] 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.SP.500-341

[5] Black PE, Kass M, Koo M, Fong E (2011) Source code security analysis tool functional specification version 1.1 (NIST), NIST-SP 500-268 v1.1. https://doi.org/10.6028/NIST.SP.500-268v1.1

[6] Black PE (2005) Software assurance metrics and tool evaluation. *Proc. Intern'l Conf. on Software Engineering Research and Practice (SERP '05)*, eds Arabnia H, Reza H (CSREA), Vol. II.

[7] Delaitre A, Stivalet B, Black PE, Okun V, Ribeiro A, Cohen TS (2018) SATE V report: Ten years of static analysis tool expositions (National Institute of Standards and Technology), NIST-SP 500-326. https://doi.org/10.6028/NIST.SP.500-326

[8] (2020) Static analysis results interchange format (SARIF) version 2.1.0, https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. Accessed 8 February 2022.

[9] Common weakness enumeration (CWE). Accessed 29 April 2024. Available at https://cwe.mitre.org/.

[10] SARD acknowledgments and test suites descriptions, https://www.nist.gov/itl/ssd/software-quality-group/sard-acknowledgments-and-test-suites-descriptions. Accessed 25 July 2024.

[11] Okun V, Delaitre A, Black PE (2013) Report on the static analysis tool exposition (SATE) IV (National Institute of Standards and Technology), NIST-SP 500-297. https://doi.org/10.6028/NIST.SP.500-297

[12] Common vulnerabilities and exposures (CVE), https://www.cve.org/. Accessed July 2024.

[13] Zitser M, Lippmann RP, Leek T (2004) Testing static analysis tools using exploitable buffer overflows from open source code. *Proceedings 12th International Symposium on Foundations of Software Engineering* (ACM SIGSOFT), pp 97–106. https://doi.org/10.1145/1029894.1029911

[14] Test suites, https://samate.nist.gov/SARD/test-suites. Accessed 31 July 2024.

[15] Shiraishi S, Mohan V, Marimuthu H (2015) Test suites for benchmarks of static analysis tools. *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* ISSREW '15 (IEEE Computer Society), pp 12–15. https://doi.org/10.1109/ISSREW.2015.7392027

[16] Hoole AM, Traore I, Delaitre A, de Oliveira C (2016) Improving vulnerability detection measurement: [test suites and software security assurance]. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Association for Computing Machinery). https://doi.org/10.1145/2915970.2915994

[17] Seacord RC (2005) *Secure Coding in C and C++* (Addison-Wesley).

[18] Viega J (2005) *The CLASP Application Security Process* (Security Software, Inc.).

[19] Kratkiewicz K, Lippmann R (2006) A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. *Proc. Workshop on Software Security Assurance Tools, Techniques, and Metrics*, ed Fong E NIST-SP 500-265, pp 44–51. https://doi.org/10.6028/NIST.SP.500-265

[20] (2011) Juliet test suite v1.1 for Java user guide (Center for Assured Software), Available at https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.1_for_Java_-_User_Guide.pdf.

[21] (2011) Juliet test suite v1.1 for C/C++ user guide (Center for Assured Software), Available at

https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.1_for_C_Cpp_-_User_Guide.pdf.

[22] (2012) Juliet test suite v1.2 for Java user guide (Center for Assured Software), Available at https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf.

[23] (2012) Juliet test suite v1.2 for C/C++ user guide (Center for Assured Software), Available at https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf.

[24] Black PE (2018) Juliet 1.3 test suite: Changes from 1.2 (National Institute of Standards and Technology), NIST-TN 1995. https://doi.org/10.6028/NIST.TN.1995

[25] do Nascimento AFM (2023) *A Transformer-based GitHub Action for Vulnerability Detection* Master's thesis Faculdade de Engenharia da Universidade do Porto. https://repositorio-aberto.up.pt/bitstream/10216/151941/2/636675.pdf.

[26] 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

[27] Black PE, Mentzer W, Fong E, Stivalet B (2023) Vulnerability test suite generator (VTSG) version 3 (National Institute of Standards and Technology), NIST-IR 8493. https://doi.org/10.6028/NIST.IR.8493

[28] Schuckert F, Langweg H, Katt B (2022) Systematic generation of XSS and SQLi vulnerabilities in PHP as test cases for static code analysis. *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp 261–268. https://doi.org/10.1109/ICSTW55395.2022.00053

[29] Common platform enumeration (CPE). Available at https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe.

[30] Boland T, Black PE (2012) Juliet 1.1 C/C++ and Java test suite. *IEEE Computer* 45(10):88–90. https://doi.org/10.1109/MC.2012.345

[31] Danial A, Snel S, jolkdarr, Beckmann C, MichaelDimmitt, Roman, Wilk J, et al (2021) cloc: v1.90. https://doi.org/10.5281/zenodo.5760077.

Appendix A. Web Pages Associated with the SARD

Along with the SARD, we host other web pages. For convenience, here is the content of those pages as of August 2024.

Appendix A.1. SARD Acknowledgments and Test Suites Descriptions

https://www.nist.gov/itl/ssd/software-quality-group/sard-acknowledgments-and-test-case-descriptions

The first part of this page credits those who contributed test suites or test cases to the SARD and includes a short description for each suite. We appreciate them. These represent considerable intellectual effort to reduce reported vulnerabilities to examples, classify them, generate elaborations of particular flaws, come up with corresponding correct examples, etc. This page acknowledges those people, groups, companies, and entities who have generously shared with everyone.

Contributors are listed in alphabetical order:

ABM06 Fortify Software, Inc., now HP Fortify, contributed ABM 1.0.1, which is a collection of small, synthetic C programs in flawed and flaw-free forms. The test cases cover various software security flaws, along with good or fixed versions. This is an update of FSI05. These test cases are in test suite 6.

BCS15 Bertrand C. Stivalet and Aurelien Delaitre designed the architecture and oversaw development of a test generator by TELECOM Nancy students to create 42 212 test cases in PHP. See [26]. Test suite 103.

BCS16 Bertrand C. Stivalet and Aurelien Delaitre designed the architecture and oversaw development of a more modular and extensible test generator based on [BCS15] by TELECOM Nancy students to create 32 003 test cases in C#. Test suite 105.

CAS10 National Security Agency's Center for Assured Software created over 45 000 test cases in C/C++ and 14 000 in Java covering over 100 CWEs, called the Juliet test suite. They can be compiled individually, in groups, or all together. The C/C++ or Java cases and supporting files can be downloaded from the Test Suites page. Individually they are in test suites 68 (C/C++) and 69 (Java).
This was superseded by Juliet 1.1 [CAS12].

CAS12 National Security Agency's Center for Assured Software released Juliet 1.1, which is extended to over 57 000 test cases in C/C++ and almost 24 000 in Java. They can be compiled individually, in groups, or all together. It is described in [30]. The C/C++ or Java cases and supporting files are in test suites 26 (C/C++) and 28 (Java).
This was superseded by Juliet 1.2 [CAS13].

CAS13 National Security Agency's Center for Assured Software updated its Juliet test suite to version 1.2. The new suite contains over 61 000 test cases in C/C++ and 25 000 in Java. The C/C++ or Java cases and supporting files can be downloaded from the Test Suites page. Individually they are in test suites 86 (C/C++) and 87 (Java).
This was superseded by Juliet 1.3 [NIST17].

CAS20 National Security Agency's Center for Assured Software created almost 29 000 test cases in C# covering 105 CWEs, called the Juliet test suite for C# version 1.3. The C# test cases and supporting files can be downloaded from the Test Suites page.

DRDC06 Frédéric Michaud and Frédéric Painchaud, Defence R&D Canada http://www.drdc-rddc.gc.ca/, created 25 C++ test cases. These test cases, plus a 26th with a main() including them all, are in test suite 62. Jeffrey Meister, NIST, entered them.

FSI05 Fortify Software Inc., now HP Fortify, contributed a collection of C programs that manifest various software security flaws. Those flaws include (1) Buffer Overflow (2) Format String Vulnerability (3) Untrusted Search Path (4) Memory Leak (5) Double Free Vulnerability (6) Race Condition (7) Direct Dynamic Code Evaluation (8) Information Leak, etc.

HH11 Hamda Hasan contributed C#, including ASP.NET, test cases with XSS, SQL injection, command injection, and hard coded password weaknesses. You can find these cases by searching for "Hamda".

IARPA12 The Intelligence Advanced Research Projects Activity (IARPA) created a test suite for Phase 1 of the Securely Taking On New Executable Software Of Uncertain Provenance (STONESOUP) program. The test suite consists of small C and Java programs, along with inputs triggering the vulnerability and build and execute directions. It comprises five collections of test cases: memory corruption for C, null pointer dereference for C, injection for Java, numeric handling for Java, and tainted data for Java. The five collections, about 450 test cases, may be downloaded as test suites 29, 30, 32, 33, and 34.

IARPA14 The Intelligence Advanced Research Projects Activity (IARPA) created a test suite for Phase 3 of the Securely Taking On New Executable Software Of Uncertain Provenance (STONESOUP) program. The test suite is a collection of 7770 C and Java test cases based on 16 widely-used open source programs in which vulnerabilities have been seeded. IARPA STONESOUP documents are available at the SARD Documentation page. It may be downloaded in a virtual machine as test suite 113. Alternatively, the test cases can be viewed and downloaded individually as test suite 102.

ITC14 Toyota InfoTechnology Center (ITC), U.S.A. static analysis benchmarks for undefined behavior and concurrency weaknesses. 100 test cases in C and C++ containing a total of 685 pairs of intended weaknesses. The test cases are © 2012–2014 and distributed under the "BSD License." See [15]. Test suite 104.

KLOC06 Klocwork Inc. shared 41 cases in C and C++ from their regression test suite. The test cases are © 2000–2005 Klocwork Inc. All rights reserved. See the test cases for details. Since then, some cases have been deprecated and replaced. Test suite 106.

MLL05a MIT Lincoln Laboratory developed a comprehensive taxonomy of C program buffer overflows and 291 diagnostic C code test cases representing this taxonomy. Each test case has three flawed versions (with buffer overflows just outside, moderately outside, and far outside the buffer boundary) and a patched version (without buffer overflow). Examples of using these test cases are in [19]. Test suite 89.

MLL05b MIT Lincoln Laboratory extracted 14 model programs from popular internet applications (BIND, Sendmail, WU-FTP) with known, exploitable buffer overflows. These programs have the portion of code with the overflows. Patched versions are also available. Examples of using these model programs are in [13]. These 28 test cases are test suite 88 and test cases 1283 to 1310.

MS06 Michael Sindelar, UMass-Amherst and NIST, wrote test cases for threading.

NIST17 Paul E. Black, Charles de Oliveira, and Eric Trapnell updated the Juliet 1.2 test suite [CAS13], originally from National Security Agency's Center for Assured Software, to version 1.3. They fixed more than a dozen problems affecting thousands of test cases, including getting

all but Windows-specific cases to compile in Linux, and added 6140 cases of pre- and post-increment and decrement over- or underflow. The C/C++ or Java cases and supporting files can be downloaded from the Test Suites page. Individually, they are in test suites 108 (C/C++) and 109 (Java).

RC06 Roderick Chapman, Altran Praxis, contributed an array access out-of-bounds case (1484) that occurs if the compiler generates code one way but not if it generates code another way. The C language does not specify which way.

RCS06 Robert C. Seacord contributed 69 examples from "Secure Coding in C and C++" [17]. Romain Gaucher, NIST, wrote the descriptions and entered the examples.

SSW05 Secure Software Inc. published CLASP [18] (Comprehensive, Lightweight Application Security Process) in 2005. Volume 1.1 Training Manual, Chapter 5, Vulnerability Root-Causes, has coding examples of software vulnerabilities.

Appendix A.2. Non-NIST Publications on SARD Content

The second part of the Acknowledgments page lists publications about work that uses SARD cases or work that comments directly on them. They are listed newest first.

Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin, "VulCNN: An image-inspired scalable vulnerability detection system," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 2365–2376, https://doi.org/10.1145/3510003.3510229

> "...we focus on detecting vulnerability in C/C++, therefore, we only select functions written in C/C++ in SARD. Data obtained from SARD consists of 12,303 vulnerable functions and 21,057 non-vulnerable functions."

Matteo Mauro, "Regole di Programmazione per la Safety e Security: Analisi, Strumenti e Relazioni" ("Programming Rules for Safety and Security: Analysis, Tools and Relations"), Bachelor Thesis, Università Degli Studi Firenze, 2018, unpublished.

> Mauro ran several static analyzers for MISRA rules on some Juliet test cases. The goal was to study which MISRA rules can also be helpful for security.

Gabriel Díaza and Juan Ramón Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests", Information and Software Technology, 55(8):1462–1476, August 2013. https://doi.org/10.1016/j.infsof.2013.02.005

> "The study compares the performance of nine tools (CBMC, K8-Insight, PC-lint, Prevent, Satabs, SCA, Goanna, Cx-enterprise, Codesonar) ... against SAMATE Reference Dataset test suites 45 and 46 for C language."

Anne Rawland Gabriel, "NIST Tool Boosts Software Security, FedTech", 8 February 2013. https://fedtechmagazine.com/article/2013/02/nist-tool-boosts-software-security

> "Using the SARD test suites for internal testing and evaluation allows our researchers to gain insight into how their technology fares against a wide range of vulnerabilities . . . "

Robet Auger, "NIST publishes 50kish vulnerable code samples in Java/C/C++, is officially krad", cgisecurity.com blog, 31 March 2011.

> He calls the Juliet test suite "a fantastic project" and says, "If you're new to software security and wish to learn what vulnerabilities in code look like, this is a great central repository . . . "

Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz, "BegBunch: benchmarking for C bug detection tools", Proc. 2nd International Workshop on Defects in Large Software Systems; held in conjunction with Int'l Symposium on Software Testing and Analysis (ISSTA 2009), Chicago, Illinois, July 2009.

> Describes BegBunch. Compares BegBunch with the SARD and other collections.

Henny Sipma, "SAMATE Case Analysis Report", Kestrel Technology, April 2008.

> This is "An application of CodeHawk to a NIST benchmark suite." The first page reads "CodeHawk Buffer-overflow Analysis Report: Benchmarks 115-1278". CodeHawk found a previously-unrecognized buffer underflow vulnerability in case 834.

John Anton, Eric Bush, Allen Goldberg, Klaus Haveland, Doug Smith, and Arnaud Venet, "Towards the Industrial Scale Development of Custom Static Analyzers", Kestrel Technology, 2006.

> "The SAMATE database will provide the basis for studying the specification language." Specifically uses cases 1314 and 54.

The following mention the SARD in passing.

Redge Barthomew, "Evaluation of Static Source Code Analyzers for Safety-Critical Software Development", 1st International Workshop on Aerospace Software Engineering (AeroSE 07), 21–22 May 2007.

Robert C. Seacord and Jason A. Rafail, "Secure Coding Standards", Third Annual Cyber Security and Information Infrastructure Research Workshop (CSIIRW 2007), eds Frederick Sheldon, Axel Krings, Seong-Moo Yoo, and Ali Mili, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 14–15 May 2007, pp. 70–72. https://www.lulu.com/shop/frederick-sheldon-and-seong-moo-yoo-and-ali-mili-and-axel-krings/proceedings-of-2007-cyber-security-and-information-infrastructure-workshop/ebook/product-1447877.html

Appendix A.3. Other Assurance Tool Test Collections

https://www.nist.gov/itl/ssd/software-quality-group/other-assurance-tool-test-collections

These are other assurance tool collections and benchmarks that we are aware of. Test collections in this list must be designed to assess the capabilities of assurance tools. Tool test collections may include requirements analysis tool tests, design model analysis tool tests, source code analysis tool tests, static and dynamic binary analysis tool tests. Test collections listed may include a harness or a test framework.

BUGZOO and ManyBugs.
Christopher Steven Timperley, Susan Stepney, and Claire Le Goues, (2018), "Poster: BUGZOO – A Platform for Studying Software Bugs", 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 446–447, DOI: 10.1145/3183440.3195050.
Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer, (2015), "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs", IEEE Transactions on Software Engineering (TSE), vol. 41, no. 12, pp. 1236–1256, December 2015, DOI: 10.1109/TSE.2015.2454513.

Software-artifact Infrastructure Repository (SIR): This provides programs for experimenting with testing and analysis techniques and materials facilitating that use. The primary purpose of the tests and testing framework is generation of experiments in software fault testing. That said, extension of the test suites and test objects is encouraged by the developers. http://sir.unl.edu/

- Test Suite Type: Source Code for Specific Versions of Open Source Applications
- Number of Programs: 85
- Number of Bugs: 572+
- Average Number of Lines of Code: 38,825
- Languages: Java, C, C++, C#
- Supports Multiple Versions of Test Cases: Yes
- Test Harness: Yes
- Documentation: Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact", Empirical Software Engineering, vol. 10, pp. 405–435, 22 July 2005. https://doi.org/10.1007/s10664-005-3861-2

FaultBench: is a set of real Java programs for comparison and evaluation of actionable alert identification techniques (AAITs) that supplement automated static analysis.

- Test Suite Type: Source Code (via CVS) for Multiple Versions of Open Source Applications
- Number of Programs: 6
- Number of Bugs: 780
- Average Number of Lines of Code: 4973
- Language: Java
- Supports Multiple Versions of Test Cases: Yes
- Test Harness: No
- Documentation: Sarah Heckman and Laurie Williams, "On Establishing a Benchmark for

Evaluating Static Analysis Alert Prioritization and Classification Techniques", Proc. Second Int'l Symposium on Empirical Software Engineering and Measurement (ESEM 2008), October 9–10, 2008, Kaiserslautern, Germany. https://doi.org/10.1145/1414004.1414013

OWASP Benchmark Project is a suite of synthetic test cases designed to evaluate the speed, coverage, and accuracy of vulnerability detection tools. It includes both test cases with weaknesses and without weaknesses (to test for false positives). The test suite is updated periodically. https://owasp.org/www-project-benchmark/

- Test Suite Type: Source Code
- Number of Programs: Over 2500
- Number of Bugs: Over 1300
- Average Number of Lines of Code: 53 (As of version 1.2)
- Language: Java
- Supports Multiple Versions of Test Cases: Yes
- Test Harness: Yes

Appendix A.4. Other SARD Web Pages

More important parts of these pages have been incorporated into this report. We list there pages here for completeness.

"Software Assurance Reference Dataset (SARD) Manual"

https://www.nist.gov/itl/ssd/software-quality-group/software-assurance-reference-dataset-sard-manual

This page has sections on the purpose of the SARD, test cases, test suites, future enhancements, and some history.

"SARD Design Issues"

https://www.nist.gov/itl/ssd/software-quality-group/sard-design-issues

This page discusses many of the design issues and decisions of the SARD.

"SARD Manual: What the Test Case Status Means"

https://www.nist.gov/itl/ssd/software-quality-group/sard-manual-what-test-case-status-means

This page explains what the SARD Candidate, Accepted, and Deprecated statuses mean.

Appendix B. Details of Figure 1

The figure showing the language, size, origin, and quantity of cases in the SARD required complicated processing heuristics. There are undoubtedly inaccuracies in the heuristics, but they should not invalidate the general appearance of Fig. 1.

Classification of test cases as synthetic, production, or production with injected weaknesses may not be correct for all cases. This information is not part of metadata for test cases. We derived this

information during processing for this figure based on the author's knowledge of the source of cases and on examination of text and code in the cases. The designations are likely sufficiently correct. Following is the actual code.

```
if   149249 <= testcase_id_int <= 157018:
    kind = 'INJ'
elif 148804 <= testcase_id_int <= 149041:
    kind = 'PRO'
elif 1 <= testcase_id_int <= 231333:
    # the rest of 2017 SARD is synthetic
    kind = "SYN"
elif 231334 <= testcase_id_int <= 231338:
    # STONESOUP Digium - new versions
    kind = 'INJ'
elif 231339 <= testcase_id_int <= 231380:
    # STONESOUP Wireshark
    kind = 'INJ'
elif 501108 <= testcase_id_int <= 501377:
    # SATE VI injected cases
    kind = 'INJ'
elif author in ("Felix Schuckert", "NSA/Center for Assured Software")
                            or author.startswith("Daniel Marjam"):
    # synthetic PHP or Java cases
    kind = "SYN"
elif author == "WordPress Foundation":
    kind = "PRO"
else:
    print(f'Do not know the kind of case {testcase_id_int}')
    sys.exit(1)
```

It is infeasible to assign a measure of number of lines of code in *every* case. Some cases are only test or build material. Other cases include configuration files or setup scripts. Many test cases duplicate support files for thousands of files; we don't count such repeated "library" files. Specifically, if there is a `src/testcases` or `src/main/java/testcases` directory, the program only counts files there, not in a `src/testcasesupport` directory.

We compared test case classification and lines of code with data used in the 2017 paper [3]. Slight differences remain that we have not accounted for. The 2017 paper and our notes for it didn't have enough details for us to understand all the differences. We do know that we counted lines of code slightly different; we believe the current count to be more accurate. In 2017, we counted files with extension .jsp in *all* test cases; now we only count them in some cases. In 2017 C files in `scripts` subdirectories were counted, e.g., case 153829; we do not include them here. We do not include cases that have been deprecated since 2017.

We document here our concerns about inaccuracies in assignment to kind and lines of code because we did not perform exhaustive cross checks, audits, and sanity checks. Some checks we performed turn up some errors, which we corrected. Further examination did not seem to be worth the effort.

We used cloc 1.90 [31] to count the source lines of code. We invoked it as

```
cloc --sum-one --include-ext=c,cc,cpp,java,php,phps,cs,aspx,jsp
```

This only counts lines in files ending in one of those extensions, ignoring comment and blank lines.

Plotting the Data

We grouped the logarithm of test case sizes into buckets for plotting. Values of groups range from 0 to 5.5 inclusive in steps of 0.05. Here's the pertinent Perl code, where `$LoC` is the lines of code.

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}

$precision = 20;

    $LogLoC = log10($LoC);
    $bucket = int($LogLoC * $precision);
    $numbTCs[$bucket]++;
```

Each bucket is plotted with a rectangle. The left edge of the rectangle aligns with the X axis position. For synthetic, yellow, and injected, orange, cases, the rectangle height is the logarithm of the number of test cases plus one. We added one so buckets with count of one are visible[1]. Synthetic case rectangles go from the language center line upward. Injected rectangles go downward from the center line. Since there are so few production cases, we decided to make production rectangles extend from where the top of a matching synthetic rectangle would be to where the bottom of a matching injected rectangle would be. This makes them twice the height of synthetic or injected rectangles. Here's the R code:

```
data$size  <- log10(data$numTestCases + 1) # +1 so log 1 case is not 0
# some arbitrary number to scale above size to rect sizes
scaleSize   <- 1/8
# SYN and PRO extend upward
data$ytop    <- ifelse(data$kind=='SYN',
    data$y + scaleSize*data$size/2, data$y)
data$ytop    <- ifelse(data$kind=='PRO',
    data$y + scaleSize*data$size/2, data$ytop)
# INJ and PRO extend downward
data$ybottom <- ifelse(data$kind=='INJ',
    data$y - scaleSize*data$size/2, data$y)
data$ybottom <- ifelse(data$kind=='PRO',
    data$y - scaleSize*data$size/2, data$ybottom)
```

The bars at the top showing the size for 1, 100, and 5000 cases follow the same sizing computation.

[1] The logarithm of 1 is 0, which would be no height.