

SARD: Thousands of Reference Programs for Software Assurance

Paul E. Black, National Institute of Standards and Technology, Gaithersburg, MD

One way to understand the strengths and limitations of software assurance tools is to use a corpus of programs with known bugs. The software developer can run a candidate tool on programs in the corpus to get an idea of the kinds of bugs that the tool finds (and does not find) and the false positive rate. The Software Assurance Reference Dataset (SARD) [16] at the National Institute of Standards and Technology (NIST) is a public repository of hundreds of thousands of programs with known bugs. This article describes the content of SARD, how to find specific material, and ways to use it.

SARD has over 170 000 programs in C, C++, Java, PHP, and C# covering more than 150 classes of weaknesses. Most of the test cases are synthetic programs of a page or two of code, but there are over 7000 full size applications derived from a dozen base applications. Although not every vulnerability is indicated in every program, the vast majority of weaknesses are noted in metadata, which can be processed automatically. Users can search for test cases by language, weakness type, and many other criteria and can then browse, select, and download them.

The term “bug” is ambiguous. “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.” [14, page 4] In isolation, a piece of code may have a buffer overflow or command injection weakness, but because the input is filtered or only comes from a trusted source, it may not constitute a vulnerability, which is exploitable. In fact, it may be difficult to determine if a particular piece of code may be reachable at all. It may, in practice, even be dead code. Hence, we usually talk about weaknesses and leave larger, system level concerns for another discussion.

We first explain the goals and organization of SARD, then describe the very diverse content. After that, we give advice on how to find and use SARD cases, related work and collections, and future plans for SARD.

SARD Philosophy and Organization

The SARD consists of test cases, which are individual programs. Each test case has “metadata” to label and describe it. Many test cases are organized into test suites. Some test cases share common files with other cases.

The code is typical quality. It is not necessarily pristine or exemplary, nor is it horrible. SARD is not a compiler test. For now, we ignore the question of language version, e.g., C99 vs. C11.

Users can search for test cases by programming language, weakness type, size, and several other criteria and can then browse, select, and download them. Users can access test suites, which are collections of test cases. We explain more in the section explaining how to use SARD content.

Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor does it imply that the products are necessarily the best available for the purpose.

Many synthetic programs represent thousands of variations for different weakness classes. 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, determine conditions when the piece of code may be executed, etc. Many sets of synthetic programs have the same base weakness wrapped in different *code complexities*. For instance, an uninitialized variable may be declared in one function and a reference passed to another function, where it is used. Other code complexities are when the weakness is wrapped in various types of loops or conditionals or uses different data types.

Test cases are labeled “good,” “bad,” or “mixed.” A “bad” test case contains one or more specific weaknesses. A “good” test case is associated with bad cases, but the weaknesses are fixed. Good cases can be used to check false positives. A “mixed” test case has both, for instance, code with a weakness and the same code with the weakness fixed. Weaknesses are classified using the Common Weakness Enumeration (CWE) [7] ID and name. We plan to also list their Bugs Framework (BF) [3] class. Fig. 1 shows the result of searching for test cases. Clicking on 199265 displays that case, as shown in Fig. 2.

Test Case ID ▼	Submission Date	Language	Type of Artifact	Status	Description	Weakness	Bad ✖ Good ✔ Mixed ✖✔
148725	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: getParameter_Servlet Read ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	✖✔
148762	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: getQueryString_Servlet Parse ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	✖✔
148799	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: listen_tcp Read data using a ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	✖✔
148820	2014-08-01	C	Source Code	C	CVE-2010-1772	CWE-416: Use After Free	✖
148936	2014-08-01	C	Source Code	C	CVE-2013-1572	CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')	✖
149039	2014-08-01	PHP	Source Code	C	CVE-2013-7233	CWE-352: Cross Site Request Forgery	✖
199265	2016-07-22	C	Source Code	C	Defect Type: Pointer related defects. Defect Sub-type: Bad cast of af	CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak') CWE-465: Pointer Issues CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-704: Incorrect Type Conversion or Cast	✖
199266	2016-07-22	C	Source Code	C	Defect Type: Pointer related defects. Defect Sub-type: Bad cast of af	CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak') CWE-465: Pointer Issues CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-704: Incorrect Type Conversion or Cast	✔
199267	2016-07-22	C	Source Code	C	Defect Type: Inappropriate code. Defect Sub-type: Return value ...	CWE-252: Unchecked Return Value CWE-561: Dead Code CWE-562: Return of Stack Variable Address CWE-672: Operation on a Resource after Expiration or Release	✖
199268	2016-07-22	C	Source Code	C	Defect Type: Inappropriate code. Defect Sub-type: Return value ...	CWE-252: Unchecked Return Value CWE-561: Dead Code CWE-562: Return of Stack Variable Address CWE-672: Operation on a Resource after Expiration or Release	✔
199275	2016-07-22	C	Source Code	C	Defect Type: Resource management defects. Defect Sub-type: ...	CWE-416: Use After Free CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-824: Access of Uninitialized Pointer	✖

Figure 1. A screen shot of test cases found for a search. It shows that cases have “metadata” or information such as, test case ID, source code language, status, description, weaknesses included, and an indication of whether it has weaknesses (bad), no weaknesses (good), or mixed.

CWE-476: NULL Pointer Dereference on line(s): 101, 103, 151, 157, 177, 179, 180, 182, 210, 216, 229, 240, 241, 262, 338, 352, 395, 397, 438, 443, 560, 561, 601
 CWE-704: Incorrect Type Conversion or Cast on line(s): 41, 60, 80, 122, 171, 255, 257, 265, 284, 307, 351, 373, 417, 453, 484, 528, 591
 CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak') on line(s): 172, 419
 CWE-561: Dead Code on line(s): 314
 CWE-465: Pointer Issues on line(s): 179

```

85  /*
86  * Type of defect: bad function pointer casting - Wrong return type
87  * Complexity: different return type function :char * and function pointer: int (one char * a
88  * function pointer declared and used inside for loop
89  */
90  static char * func_pointer_004_func_001 (char *str1)
91  {
92      int i = 0;
93      int j;
94      char * str_rev = NULL;
95      if (str1 != NULL)
96      {
97          i = strlen(str1);
98          str_rev = (char *) malloc(i+1);
99          for (j = 0; j < i; j++)
100          {
101              str_rev[j] = str1[i-j-1];
102          }
103          str_rev[i] = '\0';
104          return str_rev;
105      }
106      else
107      {
108          return NULL;
109      }

```

Figure 2. Screen shot of code from case 199625. The NULL pointer dereference weakness shows up on lines 101, 103, and other lines. Each case has such “metadata” available for for automatic processing.

SARD is archival. That is, once a test case is deposited, it will not be removed or changed. That way, if research uses a test case, later researchers can access that exact test case, byte for byte, to replicate the results. This is important to determine if, say, a new technique is more powerful than a previous technique.

If problems are later found with a test case, a new version may be added. For instance, if an extraneous error is found in a test case or the test case uses an obsolete language feature, an alternate may be added that corrects the problem. The original can still be accessed, but its status is *deprecated*.

A test case is deprecated if it should not be used for new work. If a test case does not yet meet our documentation, correctness, and quality requirements, its status is *candidate*. When it does, its status is *accepted*. A user can expect that the documentation of an accepted test case contains:

- A description of the purpose of the test case.
- An indication that it is good (false alarm), bad (true positive), or mixed.
- Links to any associated test cases, e.g. the other half of a bad/good test case pair.
- The source code language, e.g. C, Java, or PHP.
- Instructions to compile, analyze, or execute the test, if needed. This may include compiler name/version, compiler directives, environment variable definitions, execution instructions, or other test context information.
- The weakness(es) class(es).
- If this is a “bad” or “mixed” test, the location of known weaknesses, e.g. file name and line number.

Source code for an accepted test case will:

- Compile (for compilable languages).
- Run without fatal error, other than those expected for an incomplete program.
- Not generate any warnings, unless the warnings are expected as part of the test.
- Contain the documented weakness if the test case is a bad or mixed test case.
- Contain no weaknesses at all if it is a good case.

We have permission to publicly furnish the SARD test cases. In fact, many test cases are in the public domain. We are working to attach explicit usage rights to each case and each suite.

SARD was designed to offer as many as one billion test cases. Cases are organized in directories of one thousand. For instance, when downloaded, the path to case 1320984 is `testcases/001/320/984`. That subdirectory may contain a single file, or it may contain many files and subdirectories for a large, complex case.

SARD Content

Since it is not clear what the perfect test suite would be (or if there is one!), we gathered many different test case collections from many sources. This section describes the collections to provide an idea of the kinds of cases that are currently available. First we describe the large collections of synthetic cases generated by programs. Next we describe the collections of cases written by hand. Finally we describe cases from production code. Table 1 gives a very general idea of all SARD cases listing the number of cases in each language. (The table does not count deprecated cases.) Fig. 3 gives a better idea of the quantity, size, and source of cases in each language.

Language	Number of Cases
C	46 846
C++	21 138
Java	28 828
PHP	42 248
C#	32 018

Table 1. Number of test cases in each programming language in SARD as of 10 March 2017.

By far the largest number of test cases are synthetic. One of our first collections came from MIT Lincoln Laboratory. They 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 [11] and are designated test suite 89.

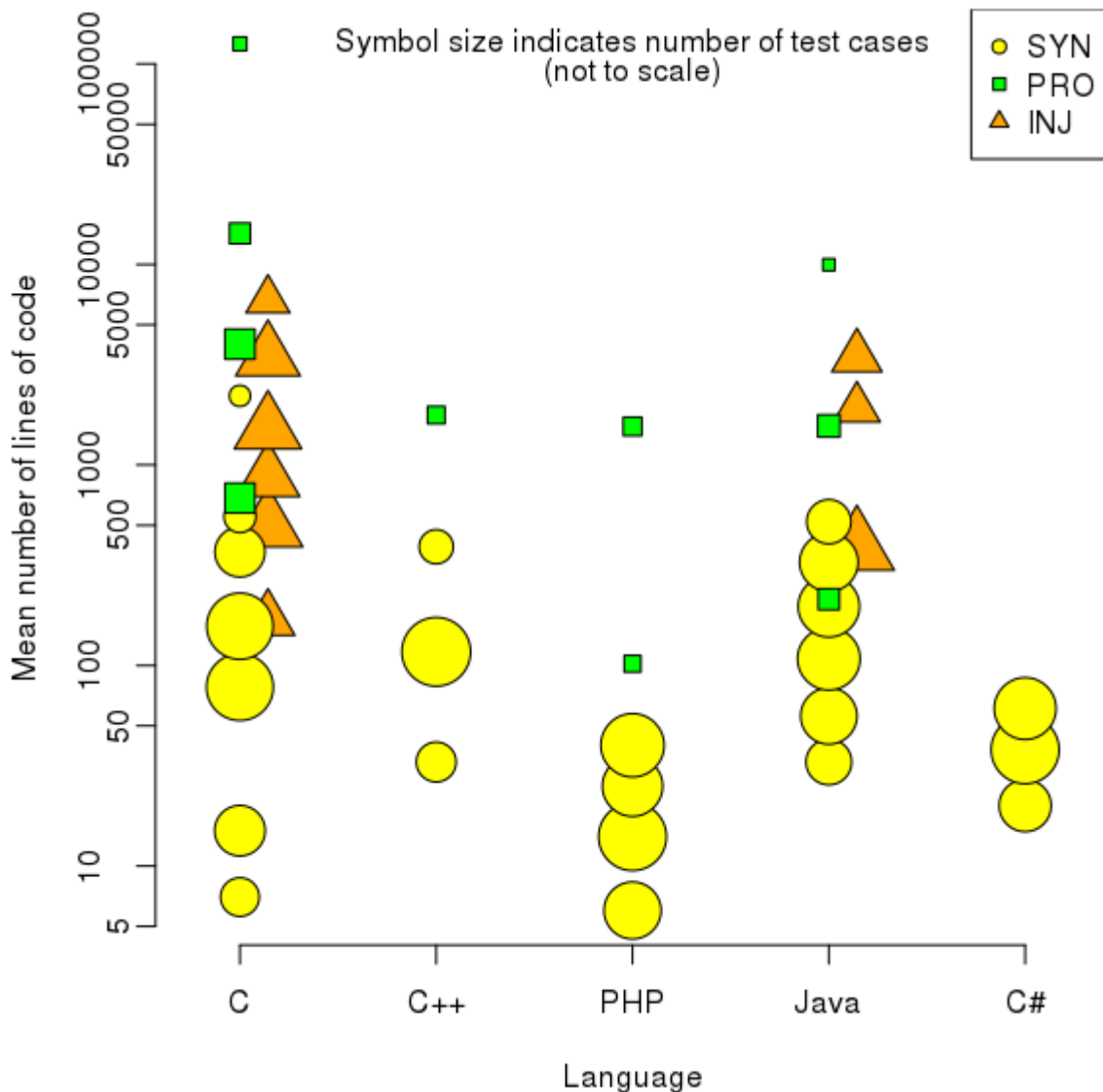


Figure 3. Number of test cases by language, clustered by lines of code. The Y axis is the mean lines of code of test cases in the cluster. Synthetic cases (SYN), in which all weaknesses are known, are yellow circles. Cases with weaknesses injected (INJ) into production code are orange triangles. Production code (PRO), which have some weaknesses identified, are green squares. The size of circles, triangles, and squares is the logarithm of number of test cases of that cluster; larger is more test cases.

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. (This was the tenth major SARD contribution and was named for the tenth letter of the International Radiotelephony Spelling Alphabet, which is "Juliet.") 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, using versions of printf() or different data types, are elaborated with up to 30 variants having complexities added. The following year they extended the collection with version 1.1, described in Boland and Black [4]. The latest version is Juliet 1.2, which comprises 61 387 C/C++ programs and 25 477 Java programs for almost two hundred weakness classes. They are test suites 86 (C/C++) and 87 (Java). The Juliet 1.0 and 1.2 suites are further described in documents at <https://samate.nist.gov/SARD/around.php>.

Following a generator architecture developed by NIST personnel and under their direction, a team of students at Telecom Nancy implemented a generator that created some PHP cases. After that, other students rewrote the generator to be more modular and extensible, under guidance of members of the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) team. They created a suite of 42 212 test cases in PHP covering the most common security weakness categories, including XSS, SQL injection, URL redirection, etc. These are suite 103 and are documented in Stivalet and Fong [20]. 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.

Manually Written Cases

Many companies donated the synthetic benchmarks that they developed manually. Fortify Software Inc., now HP Fortify, contributed a collection of 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. 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 [18]. The test cases are © 2012-2014 Toyota InfoTechnology Center, U.S.A. Inc., distributed under the “BSD License,” and added to SARD by permission. The SAMATE team noted coincidental weaknesses.

SARD also includes 329 cases from our static analyzer test suites [1]. These have suites for weaknesses, false positives, and weakness suppression in C (test suites 100 and 101), C++ (57, 58, and 59), and Java (63, 64, and 65).

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” [22] as a training reference for the Comprehensive, Lightweight Application Security Process (CLASP) of Secure Software, Inc. 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 15 cases in C#, including ASP.NET, with XSS, SQL injection, command injection, and hard coded password weaknesses.

Cases From Production Software

All the cases described to this point were a few pages of code at most and were written specifically to serve as focused tests. Small synthetic cases may not show if a technique scales or if an algorithm can handle production code with complicated, interconnected data structures over thousands of files and variables. To fill this gap, SARD has cases that came from operational code.

MIT Lincoln Laboratory extracted 14 program slices from popular Internet applications (BIND, Sendmail, WU-FTP, etc.) with known, exploitable buffer overflows [23]. That is, they chopped out all

but a relatively few functions, data structures, files, etc. so the remaining code (“the slice”) has the overflow. They also made “good” (patched) versions of each slice. These 28 test cases are in SARD as 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. The goal of STONESOUP was to fuse static analysis, dynamic analysis, execution monitoring, and other techniques to achieve orders of magnitude greater assurance. For Phase 1 they developed five collections of small C and Java programs covering five vulnerabilities: memory corruption and null pointer dereference for C, and injection, numeric handling, and tainted data for Java. Each collection may be downloaded from the SARD Test Suites page 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.

For Phase 3, STONESOUP injected thousands of weakness variants into 16 widely-used web applications, resulting in 3188 Java cases and 4582 C cases. The weaknesses covered 25 classes such as integer overflow, tainted data, command injection, buffer overflow, and null pointer. Each case is accompanied with inputs triggering the vulnerability, as well as “safe” inputs. Because the cases represent thousands of copies of full-sized applications, IARPA STONESOUP Phase 3 is distributed as a virtual machine with a complete testing environment: the base applications, all libraries needed to compile them, difference (delta) files with flaws, and a Test and Evaluation eXecution and Analysis System (TEXAS) to compile an executable from a difference file and the base app, binaries to monitor the execution, triggering and safe inputs, and expected outputs. This material, as well as results of STONESOUP, are described in documents available at <https://samate.nist.gov/SARD/around.php>.

The STONESOUP base applications are significant enough by themselves that we describe them here. They are available from the Test Suite page as Standalone apps. These 15 apps are GNU grep, OpenSSL, PostgreSQL, Tree (a directory listing command), Wireshark, 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), and Gimp (GNU Image Manipulation editor). Application ID 16, JTree, is different from the others. It is a smaller form of STONESOUP, that is, a single base case injected with weaknesses. The base case is a Java version of Tree. When processed with unzip, ID 16 produces 34 subdirectories, each with difference files to create a version of JTree with an injected weakness. Running the included `generate_application_testcases.py` creates different versions of JTree, along with test material.

For Static Analysis Tool Expositions (SATE) [14], SAMATE members tracked vulnerabilities reported through Common Vulnerability and Exposures (CVE) [6] to source code changes. This resulted in 228 CVEs in WordPress, Openfire, JSPWiki, Jetty, Apache Tomcat, Wireshark (1.2 and 1.8), Dovecot, Chrome, and Asterisk. Each of these programs has its own test suite. Each CVE has one test case that contains the file or files with the vulnerabilities. The first test case in each suite has all the CVEs, files, and identified vulnerabilities for that program. These 10 test suites represent hundreds of reported, known vulnerabilities and the corresponding source code.

SARD Home **Browse** **Search** **Resources** **Test Suites**

Search/Download Test Cases

[Search](#) | [File Search](#)

Test case ID: ?

Description contains:

Author:

Contributor:

Bad / Good / Mixed: ▼

Language: ▼

Type of Artifact: ▼

Status: ☒ **Candidate** ☒ **Accepted** ☐ **Deprecated**

Weakness:

Submission Date:
(Format: M/d/Y) ☒ **Any** ☐ **Before** ☐ **On** ☐ **After**

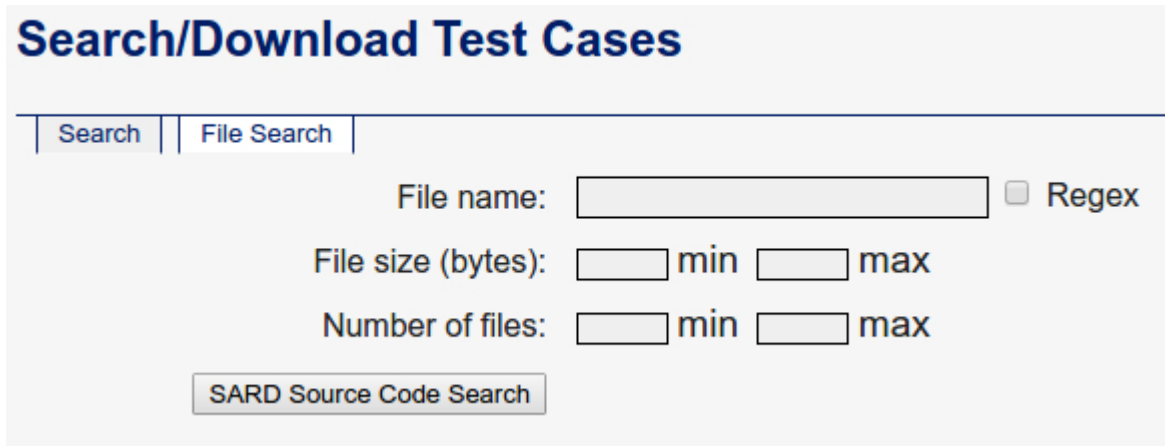
Figure 4. SARD search page. User may search for test cases meeting any combination of these criteria.

How to Use SARD Test Cases and Test Suites

The first step is to decide what test case properties are important to your situation. Programming language is the most obvious characteristic. Clicking on the “Search” tab, you may search SARD by many criteria, such as programming language, type of weakness, words in the description, type (bad, good, or mixed), status, and test case IDs, as shown in Fig. 4. Test case IDs may be ranges or lists. The type of weakness is matched to CWE descriptions as you type. You can search for and select those weaknesses that are most crucial in your situation.

Matching test cases are displayed as in Fig. 1. You may browse, select, and download any or all of the resulting cases. The download is a zip file containing a manifest (an XML listing of test cases and weakness locations) and the cases in a hierarchical directory structure of thousands described earlier.

In the File Search page, you can search for cases having files with certain names, sizes, or numbers of files, as shown in Fig. 5. This kind of search is most useful if you are trying to find, say, very large test cases. We search by file name to find where test files come from or to find related cases, which often have files with similar names.



Search/Download Test Cases

Search | **File Search**

File name: ☐ Regex

File size (bytes): min max

Number of files: min max

SARD Source Code Search

Figure 5. SARD file search page. User may search for test cases having files with particular names, files of particular sizes (minimum, maximum, or both), and particular numbers of files. User may give a regular expression to match file names, but a regex search is *far* slower.

The SARD Test Suites page lists stand-alone suites, which are very large, test suites that are collections of test cases, and web and mobile applications. The web and mobile sections are for large (full-sized) applications that we will host in those domains. Standalone apps currently consists of STONESOUP base cases, described previously. The test suites page also has links to old collections that have been superseded.

Paraphrasing Boland [4], many test suites, such as Juliet, are structured so that all the small test cases can be analyzed or compiled as a single, large program. This helps assess how a software-assurance tool performs on larger programs. Because of the number of files and size of code, some tools might not be able to analyze all these test cases as a single program. Another use is to analyze separate test cases individually or in groups.

Because the manifest indicates where flaws occur, users can evaluate tool reports semiautomatically. When users run a source code analysis tool on a test case, the desired result is for the tool to report one or more flaws of the target type. A report of this type might be considered a true positive. If the tool doesn't report a flaw of the target type in a bad method, it might be considered a false negative. Ideally, the tool won't report flaws of the target type in a "good" test case or function; a report of this type might be considered a false positive. Because flawed and similar unflawed code might be in infeasible or "dead" code, users' policies on warnings about infeasible code must be taken into account.

As an illustration of using SARD, we offer our development of a small number of cases to show that a tool is effective at finding stack-based buffer overflows. First, we downloaded all buffer overflow test cases from SARD and transformed them to expedite analysis. This resulted in 7338 test cases. We ran five tools on those cases. We compared, discussed, and grouped results until we came up with seven principles for selecting test cases [2]. This would have been far harder without the resources of SARD.

Related Work

We know of several other collections of software assurance test cases. Some include tools to run experiments and compute results. An alternative to collecting static sets is to generate sets as needed. We mention generator efforts that we know of later.

The Software-artifact Infrastructure Repository (SIR) is “meant to support rigorous controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation.” [19] It provides Java, C, C++, and C# programs in multiple versions, along with testing tools, documentation, and other material. We found 85 objects, the most recent updated in 2015.

FaultBench “is a set of real, subject Java programs for comparison and evaluation of actionable alert identification techniques (AAITs) that supplement automated static analysis.” [10] FaultBench has 780 bugs across six programs.

The OWASP Benchmark for Security Automation “is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services” [13]. It has 2740 small test cases, both with weaknesses and without weaknesses, in Java. It includes programs and scripts to run a tool and compute some results.

The Software Assurance Marketplace (SWAMP) provides more than 270 packages, in addition to the Juliet test suite, which is described above. “Packages are collections of files containing code to be assessed along with information about how to build the software package, if necessary.” [21] There are packages in Java, Python, Ruby, C, C++, and web scripting languages. Each package may have multiple versions.

SARD approaches the problem of test cases by collecting a static set. An alternative is to generate sets of cases on demand. In theory, one could specify the language, weaknesses, code complexities, and other facets, and get a—potentially unique—set as needed. Generating sets on demand would be one way to address the concern that tool makers might add bits of code just to get a high score on a static benchmark. Generated cases could be automatically obfuscated, too. The disadvantage is that each generated set would have to be examined to be sure that the cases serve their purpose (or else the generator itself would have to be qualified, which is much harder). In practice, code generators or bug injectors are enormously difficult. Nevertheless, there is some work.

Large-Scale Automated Vulnerability Addition (LAVA) creates corpora by injecting large numbers of bugs into existing source code [9]. EvilCoder also injects bugs into existing code, although it locates guard or checking code and selectively disables it [15].

The test generators implemented by Telecom Nancy students are the source of large SARD test suites in PHP and C# [20]. The Department of Homeland Security’s Static Tool Analysis Modernization Project (STAMP) recently awarded a contract to GrammaTech that includes development of a test case generator, Bug Injector [8]. KDM Analytics Inc. is enhancing their test case generator, TCG, for CAS. The latest version, TCG 3.2, produces both “bad” (flawed) and “good” (false positive) cases in C, C++, Java, and C# with control, data, and scope complexities [5]. TCG can generate millions of cases covering some three dozen Software Fault Pattern (SFP) clusters [12] and many CWEs for either or both Linux and Microsoft Windows platforms. Generated cases don’t have a `main()` function, which allows cases to be compiled individually or as one large program.

Future of SARD

SARD began in 2006 in order to collect test cases for the NIST SAMATE. We had planned to collect artifacts from all phases of the software development life cycle, including designs, source code, and binaries, in order to evaluate assurance tools for all of those. We still leave that option open, but have not yet found many tools and pressing needs for other phases.

We plan to add cases in more languages, such as JavaScript, Ruby, Swift, Objective-C, Python, or Haskell. Which language depends on the availability of test cases and the need. We are also adding thousands of mobile app test cases, since their architecture, implementation languages, and major threats are so different from typical applications.

We invite developers and researchers to donate their collections to SARD. It is a loss to the community when someone puts a lot of effort into developing a collection, then, after several years and project changes, the collection is lost.

Currently weaknesses are labeled by CWE. We will add labels of Bugs Framework (BF) classes when it is more complete [3].

Conclusion

Analysts, users, and developers have cut months off the time needed to evaluate a tool or technique using test cases from SARD. SARD has been used by tool implementers, software testers, security analysts, and four SATEs to expand awareness of static analysis tools. Educators can refer students to SARD to find examples of weaknesses. Having a reliable and growing body of code with known weaknesses helps the community improve software assurance tools themselves and encourage their appropriate use.

Acknowledgements

We thank David Flater for making the chart in Fig. 3 and Charles de Oliveira for elucidating much of the SARD content.

References

- [1] Paul E. Black, Michael Kass, Hsiao-Ming (Michael) Koo, and Elizabeth Fong, "Source Code Security Analysis Tool Functional Specification Version 1.1," NIST Special Publication 500-268 v1.1, February 2011. DOI 10.6028/NIST.SP.500-268v1.1
- [2] Paul E. Black, Hsiao-Ming (Michael) Koo, and Thomas Irish, "A Basic CWE-121 Buffer Overflow Effectiveness Test Suite," Proc. Sixth Latin-America Symposium on Dependable Computing (LADC 2013), April 2013.
- [3] Irena Bojanova, Paul E. Black, Yaacov Yesha, and Yan Wu, "The Bugs Framework (BF): A Structured Approach to Express Bugs," August 2016, 2016 IEEE Int'l Conference on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria. DOI 10.1109/QRS.2016.29

- [4] Tim Boland and Paul E. Black, “Juliet 1.1 C/C++ and Java Test Suite,” October 2012, IEEE Computer, 45(10):88-90. DOI 10.1109/MC.2012.345
- [5] Djenana Campara, private communication, 20 March 2017.
- [6] “Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names,” <https://cve.mitre.org/>, accessed 15 March 2017.
- [7] “Common Weakness Enumeration: A Community-Developed List of Software Weakness Types,” <https://cwe.mitre.org/>, accessed 10 March 2017.
- [8] “DHS S&T Awards ITHACA, NY, Company \$8M to Modernize Open-Source Software Static Analysis Tools,” <https://www.dhs.gov/science-and-technology/news/2017/03/07/st-awards-ithaca-ny-company-8m>, accessed 15 March 2017.
- [9] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan, “LAVA: Large-Scale Automated Vulnerability Addition,” 2016 IEEE Symposium on Security and Privacy, San Jose, CA, 2016, pp. 110-121. DOI: 10.1109/SP.2016.15
- [10] “FaultBench,” <http://www.researchgroup.org/faultbench/>, accessed 16 March 2017.
- [11] Kendra Kratkiewicz and Richard Lippmann, “A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools,” Proc. Workshop on Software Security Assurance Tools, Techniques, and Metrics, Elizabeth Fong, ed., NIST Special Publication 500-265, February 2006. DOI 10.6028/NIST.SP.500-265
- [12] Nikolai Mansourov and Djenana Campara, “System Assurance: Beyond Detecting Vulnerabilities,” Morgan Kaufmann, 2010, pp. 176–188.
- [13] “OWASP Benchmark Project,” <https://www.owasp.org/index.php/Benchmark>, accessed 16 March 2017.
- [14] Vadim Okun, Aurelien Delaitre, and Paul E. Black, “Report on the Static Analysis Tool Exposition (SATE) IV,” NIST Special Publication 500-297, January 2013. DOI 10.6028/NIST.SP.500-297
- [15] Jannik Pewny and Thorsten Holz, “EvilCoder: Automated Bug Insertion,” Proc. 32nd Annual Conference on Computer Security Applications (ACSAC '16), Los Angeles, CA, December, 2016, Pages 214-225. DOI: 10.1145/2991079.2991103
- [16] “Software Assurance Reference Dataset,” <https://samate.nist.gov/SARD/>, accessed 3 March 2017.
- [17] Robert C. Seacord, “Secure Coding in C and C++,” Addison-Wesley, 2005.
- [18] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu, “Test Suites for Benchmarks of Static Analysis Tools,” IEEE Int'l Symposium on Software Reliability Engineering (ISSRE '15), DOI: 10.1109/ISSREW.2015.7392027

- [19] “Software-artifact Infrastructure Repository,” <http://sir.unl.edu/>, accessed 16 March 2017.
- [20] Bertrand Stivalet and Elizabeth Fong, “Large Scale Generation of Complex and Faulty PHP Test Cases,” April 2016, 2016 IEEE Int'l Conference on Software Testing, Verification and Validation (ICST), Chicago, IL. DOI: 10.1109/ICST.2016.43
- [21] “SWAMP Packages,” <https://www.mir-swamp.org/#packages/public>, accessed 16 March 2017.
- [22] John Viega, “The CLASP Application Security Process,” Security Software, Inc., 2005.
- [23] Misha Zitser, Richard Lippmann, and Tim Leek, “Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code,” October/November 2004, Proc. SIGSOFT ‘04/12th Int'l Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12), Newport Beach, CA, pp 97-106. DOI 10.1145/1029894.1029911