Improving Vulnerability Detection Measurement

[Test Suites and Software Security Assurance]

Alexander M. Hoole, Issa Traore
Dept. of Electrical and Computer Engineering
University of Victoria
BC, Canada
{alex.hoole,itraore}@ece.uvic.ca

Aurelien Delaître, Charles de Oliveira *
Software and Systems Division, IT Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{aurelien.delaître, charles.deoliveira}@nist.gov

ABSTRACT
The Software Assurance Metrics and Tool Evaluation (SAMATE) project at the National Institute of Standards and Technology (NIST) has created the Software Assurance Reference Dataset (SARD) to provide researchers and software security assurance tool developers with a set of known security flaws. As part of an empirical evaluation of a runtime monitoring framework, two test suites were executed and monitored, revealing deficiencies which led to a collaboration with the NIST SAMATE team to provide replacements. Test Suites 45 and 46 are analyzed, discussed, and updated to improve accuracy, consistency, preciseness, and automation. Empirical results show metrics such as recall, precision, and F-Measure are all impacted by invalid base assumptions regarding the test suites.

Categories and Subject Descriptors
K.6.5 [Management of Computing and Information Systems]: Security and Protection; D.2.4 [Software Engineering]: Software/Program Verification; D.2.8 [Software Engineering]: Metrics—Product metrics

Keywords
Static Analysis, Dynamic Analysis, Weakness, Vulnerability, Security Metrics, Test Suites

1. INTRODUCTION
Vulnerabilities are the result of security or quality flaws that have been introduced into the code base during implementation, potentially by architectural oversight, poor quality control or deficiencies in requirements, which violate security policies. As a security assurance community we must continue to improve our ability to identify, verify, and fix software weaknesses as part of our software security assurance measures.

Security vulnerabilities, and their underlying weaknesses, can be identified using a variety of methods including static analysis, dynamic analysis, penetration testing, fuzzing, and code review/inspection. Static analysis is conducted on some version of the source code (e.g. source, object, or binary) without actually executing the program. Dynamic analysis is conducted against a program under execution. Penetration testing has a much broader scope and can employ the use of static (where the target is a white box with all information provided) or dynamic analysis (where the target is a black box with only minimal information available); however, penetration testing is a process which uses all available means to determine if a system is vulnerable to attack [12]. Fuzzing, or fuzz testing, is another software testing technique in which invalid, unexpected, or random data is provided to the inputs of a program in order to identify problems in software [8]. Code review is the oldest of these methods where human auditors are responsible for identifying defects in software using numerous approaches such as Fagan’s Software Inspection, Formal Technical Reviews, Humphrey’s Inspection Model, and Structured Walkthroughs [5, 13, 6, 15].

Improving software security assurance is not a trivial task and it requires careful thought, planning, and execution to improve the state of the industry over time. To reduce the amount of individual effort required to measure the effectiveness of a security assurance product, developers and researchers must have access to test suites designed for evaluating software security assurance products. The Software Assurance Metrics and Tool Evaluation (SAMATE) project at the National Institute of Standards and Technology (NIST) created the Software Assurance Reference Dataset (SARD) to provide users, researchers, and software security assurance tool developers with a set of known security flaws. These known security flaws take the form of applications which contain intentional security weaknesses, grouped into test suites with specific purposes, which can then be employed to measure different security attributes.

Building test suites or datasets for evaluating software security assurance tools is challenging due to the complexity of test subjects and the problem domain. Common sense is to rely on public datasets when adequate datasets are available. In this case, the integrity of the test results depends on
the reliability of the test suites. Despite enormous efforts invested by designers of the test suites, many test sets include mistakes and oversights that potentially flaw test results. To ensure accuracy and validity of empirical evaluation, verification of test suites is essential to ensure that they satisfy specified requirements.

We describe in this paper our analysis of test suite 45 (TS45) and test suite 46 (TS46), discuss various deficiencies and associated improvements, evaluate how invalid assumptions lead to imprecise measurement, and introduce replacements: test suite 100 (TS100) and test suite 101 (TS101). Finally, to measure the impact of the replacement test suites when evaluating security assurance products, we compare the results of a commercial static code analysis product using recall, precision, and F-Measure.

2. CHALLENGES: TEST SUITES 45 & 46

NIST has accumulated many useful test suites which provide researchers and software security assurance tool developers with mechanisms to compare and improve methods for detecting security vulnerabilities. In 2006, NIST produced a pair of draft special publications (SP) specifying the minimum capabilities of a source code security analyzer against a specific set of security weaknesses. The two documents consisted of a functional specification and a related test plan which included test suites for several programming languages. The initial version was released in 2007 and after being revised, released again as version 1.1 in 2011 [3, 7].

We initially selected, from the above test plan, test suite 45 (TS45) to test the capability of a runtime monitoring software security assurance tool’s handling of certain weaknesses in the C language and test suite 46 (TS46) to assess the false positive ratio of the tool under test[7].

2.1 Purpose of Test Suites

Test suites 45 and 46 specify sets of test cases, in the C-language, intended to measure the ability to successfully identify weaknesses and determine the false positive ratio when comparing source code security analysis products.

We use Test Suites 45\(^1\) and 46\(^2\) which were both created by Michael Koo et al. based on the SCSSA-RM-1 through SCSSA-RM-6 requirements specified in the "Source Code Security Analysis Tool Functional Specification" as part of the NIST SAMATE project[3]. The NIST SAMATE project, beginning in 2004, has been focused on improving software assurance by developing methods to enable software tool evaluations in support of the Department of Homeland Security’s Software Assurance Tools and R&D Requirements Identification Program. The Source Code Security Analyzers effort is defined as part of a specification under NIST Special Publication 500-268 v1.1 [3] and test plan under NIST Special Publication 500-270 [7]. These publications are intended to assist in understanding capabilities of source code security analysis tools and a method for evaluation. Black et. al. define the following terms in [2] and [3]:

- **weakness**: A defect in a system that may (or may not) lead to a vulnerability.
- **security vulnerability**: A property of system requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited resulting in a security failure.

\(^1\) http://samate.nist.gov/SRD/view.php?tsID=45

false negative (FN): When a tool does not report a weakness where one is present. [...]omitted...

true positive (TP): When a tool reports a weakness where one is present.

false positive (FP): When a tool reports a weakness where no weakness is present.

false positive rate: The number of false positives divided by the sum of the number of false positives and the number of true positives.

As such, every security vulnerability depends upon the existence of one or more security weaknesses within a target software artifact. These weaknesses create opportunities for an attacker to cause the software system to behave in an unintended fashion resulting in the access/modification of important data, interruption of normal execution, performing of actions outside of the allowed sphere of permissions, or violation of any number of additional system requirements. The terms above, combined with true negative (TN) which specifies a tool correctly does not report a non-existent weakness, combine to form a contingency table which will later be used as a basis of comparison (see Table 1). From the table, the columns for *Actual Weakness* and *Non-Existential Weakness* are represented by TS45 and TS46, respectively. The rows for *Reported Weakness* and *Unreported Weakness* represent what is reported by a particular analysis method when attempting to identify weaknesses.

![Weakness Identification Contingency Table](http://samate.nist.gov/SRD/view.php?tsID=46)

Table 1: Weakness identification contingency table.

We agree with the overall definition for terms given by Black et al., with the exception of false negative which had a second sentence "If the tool does not claim to identify a certain class of weakness, not reporting a weakness of that class is not a false negative"[3]. We omit the second sentence of the definition for false negative since the goal of these tools is to report all relevant security vulnerabilities (i.e. satisfy mandatory requirement SCSSA-RM-1). Other areas of science such as medicine classify a false negative, or type II error, as a result that is erroneously classified in a negative category because of imperfect methods or procedures. If a security weakness is present (as in the programs under TS45) and the analysis does not report it, then the lack of reporting gives rise to a false negative (just as the reporting of a security weakness that does not exist gives rise to a false positive). The definition of a false negative remains true whether or not a specific product is designed to detect a particular weakness type. Supporting a given weakness type also does not infer coverage of that weakness type for every permutation (i.e. occurrence of weakness in every API for a given programming language), therefore, the definition of "claim support" or "designed to detect" becomes very subjective. This is especially true of certain types of
weaknesses that are tightly coupled to the APIs being used by the program. Cross-Site Scripting (XSS), for example, is specifically identified by PCI DSS 3.0 under Requirement 6.5.7 as needing to be detected if present in the code base. If XSS is not detected by an analysis tool, and it is present, then it is a false negative in accordance with what the software security assurance tool is intended to detect according to security policy. Ideally a given tool should be customizable to support additional weakness types and coverage of specific APIs in order to reduce false negatives over time.

The above terms are used to specify mandatory requirements for source code analysis tools which must be able to handle various coding complexities and specific source code weaknesses. Koo et al. then specify a set of metrics, including test suites and methods, to determine how well a particular source code security analysis tool conforms to the specification[7]. Test suites were provided for three example programming languages (C, C++, and Java) with the intent of evaluating the requirements and features set out in NIST Special Publication 500-268 v1.1. Each test case, within a test suite, then provides metadata including a test description, a weakness description, expected result, and source code. Test cases are atomic programs that demonstrate a given weakness (marked bad) in a specific code construct in order to measure a tool’s ability to detect actual security vulnerabilities. Alternatively, a test case represents a fixed version of a weakness (marked good) in a specific code construct where the weakness has been removed to measure the capability of the tool to avoid generation of false positives. We will refer to test cases and programs interchangeably.

2.2 Coverage of CWE’s

Test suites 45 and 46 cover 21 specific CWE-IDs, from the Common Weakness Enumeration (CWE) types specified by MITRE, for evaluation. These CWEs were selected as a "base set" of weaknesses spanning the category domains of input validation, range errors, API abuse, security features, time and state, code quality, and encapsulation. Test suite 45 consists of 77 programs which are intentionally seeded with specific security weaknesses spanning 21 CWE weakness types. While discussing TS45 we should note that the NIST SP 500-270 specifies 75 test case IDs for TS45, while the downloaded test suite from SARD contains 77. The extra two test cases are 1446 (duplicate replaced by 2199) and 2108 (duplicate replaced by 2208) which affect the test suite by artificially increasing both CWE-415 and CWE-416 by one test case each respectively. Test suite 46 consists of 73 programs which have had their intentionally seeded weaknesses fixed (void of weakness). Table 2 provides a summary of the specific source code weaknesses that must be covered by a given source code analysis tool[3] along with an enumeration of the test cases that have been created to evaluate the detection of specific weaknesses[7]. For each program in TS45 the expected result of running the tool is to report the associated weakness, while for each program in TS46 the expected result is to not report the associated weakness category since the vulnerabilities have intentionally been "fixed". When the expected result for each atomic program in TS45 is not reported we record a false negative and when an expected result is reported we record a true positive. For TS46, if the expected result is not achieved for a given atomic program we record a false positive. To ensure the accuracy of such measurements, a detailed review of the programs and a verification that weaknesses are exploitable is also warranted to ensure the sanity of results.

2.3 Process for Weakness Identification

While conducting a code review of the programs in TS46, and verifying weaknesses with a new runtime monitoring approach, we uncovered exploitable weaknesses in 13 out of the 73 programs designed to not contain exploitable weaknesses of specific CWE types. Work by Díaz and Bermejo comparing various static analysis products also identifies 8 of the 13 latent defects, however, the test suite was never updated to resolve these deficiencies [4]. Thirteen out of 73 programs is significant enough to have a relatively large error which would hinder any evaluation approach comparing different approaches using statistical measures such as false positive rate, recall, precision, and F-measure [1, 9, 10, 11].

![Figure 1: Process for verifying ability to exploit.](https://cwe.mitre.org/)

### 2.3.1 Verification of Weaknesses

The general approach for identifying weaknesses and verifying the ability to exploit, depicted in Figure 1, can involve the application of several analysis and verification approaches. Assuming that the set of all possible analysis methods for identifying potential vulnerabilities is represented by the set X and that \( s_j \) is a source code artifact from the set of all possible source code artifacts S, we can select a particular analysis method \( x_i \) and use it to generate a result set \( W_{ij} \) of potential vulnerabilities (i.e. weaknesses) by passing source code \( s_j \) to analysis method \( x_i \).

\[
W_{ij} = x_i(s_j) \tag{1}
\]

Thus, each result set \( W_{ij} \), as seen in equation (1), consists of zero or more unique weaknesses \( w_k \) identified by \( x_i(s_j) \).

Once potential vulnerabilities (instances of security weaknesses which may, or may not, be reachable) have been identified, we present a practical approach to evaluate and validate the effectiveness of these methods.
Table 2: Weakness categories covered by TS45 and TS46. Test cases where weaknesses found in TS46 violate the requirement to be void of targeted CWEs are marked in **ITALICS**.

<table>
<thead>
<tr>
<th>Category</th>
<th>Source Code Weakness</th>
<th>CWE ID</th>
<th>Vulnerable Programs</th>
<th>Fixed Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TS45 (SARD Test Case ID)</td>
<td>TS46 (SARD Test Case ID)</td>
</tr>
<tr>
<td>Input Validation</td>
<td>OS Command Injection</td>
<td>78</td>
<td>111 1881 1883 1885</td>
<td>2106 2113 2118 2119</td>
</tr>
<tr>
<td></td>
<td>Basic XSS</td>
<td>80</td>
<td>1781 1794 1919 1921 2198</td>
<td>1795 1920 1922 1924 2204</td>
</tr>
<tr>
<td></td>
<td>SQL Injection</td>
<td>89</td>
<td>1796 1798 1800</td>
<td>1797 1799 1801 1930</td>
</tr>
<tr>
<td></td>
<td>Resource Injection</td>
<td>99</td>
<td>1895 1897 1899 1901</td>
<td>1896 1898 1900 1902</td>
</tr>
<tr>
<td>Range Errors</td>
<td>Stack Overflow</td>
<td>121</td>
<td>1544 1548 1563 1565 1571</td>
<td>1545 1547 1549 1566 1602</td>
</tr>
<tr>
<td></td>
<td>Heap Overflow</td>
<td>122</td>
<td>1611 1612 1843 1845</td>
<td>1574 1613 1615 1844 1848 2134</td>
</tr>
<tr>
<td></td>
<td>Format String Vulnerability</td>
<td>134</td>
<td>10 92 93 1831 1833</td>
<td>1556 1560 1562 1832 1834</td>
</tr>
<tr>
<td></td>
<td>Improper Null Termination</td>
<td>170</td>
<td>1849 1850 1854 1857 2010</td>
<td>1855 1856 1858 2012</td>
</tr>
<tr>
<td>API Abuse</td>
<td>Heap Inspection</td>
<td>244</td>
<td>1737</td>
<td>244 1865 1867 1869 1871 1973</td>
</tr>
<tr>
<td></td>
<td>Often Misused String Management</td>
<td>251</td>
<td>1865 1867 1869 1871 1973</td>
<td>1866 1868 1870 1872 1874</td>
</tr>
<tr>
<td>Security Features</td>
<td>Hard-coded Password</td>
<td>259</td>
<td>1810 1835 1837 1839 1841</td>
<td>2130 2131 2132 2133</td>
</tr>
<tr>
<td>Time and State</td>
<td>Time-of-check Time-of-use Race Condition</td>
<td>367</td>
<td>102 1806 1808</td>
<td>1892 1894</td>
</tr>
<tr>
<td></td>
<td>Unchecked Error Condition</td>
<td>391</td>
<td>1928</td>
<td>1929</td>
</tr>
<tr>
<td>Code Quality</td>
<td>Memory Leak</td>
<td>401</td>
<td>1585 1588</td>
<td>1556 1589 1926 1935</td>
</tr>
<tr>
<td></td>
<td>Unrestricted Critical Resource Lock</td>
<td>472</td>
<td>2109</td>
<td>2206</td>
</tr>
<tr>
<td></td>
<td>Double Free</td>
<td>415</td>
<td>29 1827 1829 1909 2199</td>
<td>1584 1828 1830 2271</td>
</tr>
<tr>
<td></td>
<td>Use After Free</td>
<td>416</td>
<td>2380 2381 2382 2383</td>
<td>1911 2135 2206 2270</td>
</tr>
<tr>
<td></td>
<td>Uninitialized Variable</td>
<td>457</td>
<td>1707 2003 2019</td>
<td>2186</td>
</tr>
<tr>
<td></td>
<td>Unintentional Pointer Scaling</td>
<td>468</td>
<td>1782</td>
<td>1927</td>
</tr>
<tr>
<td></td>
<td>Null Dereference</td>
<td>476</td>
<td>1875 1877 1879 2193</td>
<td>1876 1880 2194 2195</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Leftover Debug Code</td>
<td>489</td>
<td>1861</td>
<td>1862</td>
</tr>
</tbody>
</table>

2.3.2 Results of Verification

Now that we have introduced the approach to finding and verifying weaknesses, we return to the test suites themselves. According to the mandatory requirements in section 3.2.2 of the test plan, TS46 was designed to measure the ability of a software security analysis tool to produce reasonably low numbers of false positives. Each program is meant to have a "fixed" version of a given security weakness, as can be confirmed by reviewing the metadata of each test case in SARD; however, during a manual code review and verification experiment 13 programs that were supposed false positives were still exploitable for their specific weakness type.

Equation (1) represents the set of weaknesses detected using a particular analysis approach for a given software artifact. This result set consists of a set of zero or more true positives (TP) and zero or more false positives (FP). In addition, \( W_{ij} \) will not include any false negatives (FN) nor true negatives (TN). Measuring weakness detection rates as percentages will require suitable formulas. Accuracy is not a suitable metric to determine the efficiency of detecting weaknesses, see equation (2), since the total population of true negatives could far exceed the true positives and false negatives (combining to form the total number of actual weaknesses) leading to potentially large misleading results.

\[
A = \frac{TP + TN}{TP + FP + FN + TN} \tag{2}
\]

Recall, as seen in equation (3), is arguably one of the better metrics for comparing different approaches for detecting security weaknesses as it reports the number of correctly reported weaknesses from the total population of known weaknesses. In order to maximize the percentage recall a product will need to minimize false negatives which reduces the risk of having latent vulnerabilities in software artifacts.

\[
R = \frac{TP}{TP + FN} \tag{3}
\]

In addition to recall, precision also provides a useful metric to evaluate efficiency of weakness detection methods which reports the number of correctly reported weaknesses from the total population of reported weaknesses. The primary advantage of having high precision, caused by reducing the number of false positives in the denominator, is that it can help reduce the amount of human intervention required to audit detected weaknesses.

\[
P = \frac{TP}{TP + FP} \tag{4}
\]

Finally, F-Measure has been widely used in natural language processing for binary classification and provides a reasonable measure of the accuracy of a given test reported as a percentage [9, 14]. F-Measure combines both recall and precision...
into a weighted harmonic mean in the form of equation (5). The \( \alpha \) and \( \beta \) variables are control parameters which specify how much emphasis is placed upon precision versus recall. Equation (5) is then simplified into the most common form, known as the balanced form or F1 measure, where \( \beta = 1 \) and \( \alpha = 1/2 \) (see equation (6)). This provides equal weight between precision and recall.

\[
F = \frac{1}{\frac{1}{P} + \frac{(1-\alpha)}{R}} = \frac{\beta^2 + 1}{\beta^2 P + R} 
\]

F1 = \frac{2PR}{P + R} \quad (6)

These metrics will be used in the evaluation section to compare results, under different assumptions, since inaccurate values for FP and TP affect measurements.

It is very useful to have datasets which assist us in measuring the ability of software security analysis tools and we are grateful to those who expend effort in creating them. We must continue to expend tremendous effort to ensure that datasets used to measure software security assurance approaches provide concise and consistent measurements.

3. BUILD TEST SUITES 100 & 101

Discovery of persisting weaknesses in TS46 led to an extensive review, update, and replacement of both TS45 and TS46. The goals during the review were to improve consistency between the test suites, accuracy of obtained results, precision of test cases, and to improve the ability of researchers to automate their testing. Furthermore, the metadata around the test cases themselves were also improved.

3.1 Accurate: Fix 13 Incorrect Test Cases

Test suite 46 contains 13 test cases, spanning five of 21 vulnerability categories, that do not fix the weakness they were designed to correct. Table 2 marks the specific programs by test case ID (ITALICS). The University of Victoria and NIST collaborated to correct the inconsistencies.

Four CWE-78: OS Command Injection test cases (2136-2139) let dangerous characters pass through a filtering function which does not handle all the different command separators which exist for the command shell. This blacklist approach prevents one type of attack, but remains oblivious to many others. The replacement test cases (149152, 149154, 149156, 149242) use a whitelist filter allowing only safe characters, thus blocking possible attacks.

Two CWE-80: Basic XSS test cases (1920, 1924) misuse an API function designed to securely display HTML content. Instead of using it for its intended purpose, the program expects the function to sanitize tainted data, so it can display them. However data are not sanitized and can trigger a XSS attack when displayed. The replacements (149174, 149178) properly use the function to securely display data.

All four CWE-99: Resource Injection test cases (1896, 1898, 1900, 1902) implement a whitelisting function to limit access to a set of files, but the function returns inverse results, granting access to all files but the few intentionally allowed. The replacement test cases (149158, 149160, 149162, 149164) restore the proper operation.

Test case 1848 for CWE-122: Heap Overflow generates a string of random size and content. It allocates memory on the heap and fills the buffer with characters. But it adds a null terminator outside the buffer leading to a buffer overflow. Replacement test case 149126 allocated the proper amount of memory to accommodate the null terminator and corrects the location where it is written.

Test cases 1892 and 1894, examples of CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (‘Race Condition’), are used instead to express CWE-367: TOCTOU Race Condition. These test cases check file permissions before opening one, however, no file locking is implemented between the time of check and the time of file opening, leading to a race condition. After correction, the test cases where identical to others in the same category, so we modified their logic to increase diversity and comply with the targeted CWE. The resulting test cases (149232, 149234) check the file size instead of permissions.

3.2 Precise: Remove ExTRANeous Weaknesses

Test cases from TS45/TS46 contain numerous extraneous weaknesses (defects of a different type than test case intended) related to unchecked return values, buffer overflows, and other weaknesses. These weaknesses, along with minor code enhancements to fix business logic, have been removed from the replacement test cases resulting in reduced noise generated by tools while analyzing the test suites. Table 3 maps test cases in TS45/TS46 to replacements in TS100/TS101 which correct the extraneous weaknesses detailed below.

3.2.1 Unchecked Return / Buffer Weaknesses

Test cases 11/2139 for CWE-78 calculate a buffer size incorrectly that leads to a buffer overflow (CWE-680: Integer Overflow to Buffer Overflow); they also fail to check the return value of an important system function (CWE-252) creating a potential memory leak (CWE-401: Memory Leak). Test cases 111/1646 for CWE-78 disregard an important function’s return value (CWE-252: Unchecked Return Value). Test cases 1881/2136, 1883/2137 and 1885/2138, implementing the same weakness (CWE-78), could trigger a buffer overflow when concatenating a string to an uninitialized buffer (CWE-457: Uninitialized Variable as well as CWE-121). These test cases also fail to check the return value of an important function (CWE-252).

In test cases 1796/1801 and 1930 for CWE-89: SQL Injection, the database initialization function might fail, but the return value is not checked (CWE-252), leading to unexpected behavior; in addition, good test cases 1797, 1799, 1801 and 1930 use an encoding function to prevent injection, however, they are not allocating enough memory to store the string, leading to CWE-121: Stack-based Buffer Overflow. Test cases 1895/1896 and 1899/1900 for CWE-99 do not ensure proper buffer null-termination (CWE-170: Improper Null Termination).

Test cases 2009 (CWE-121) and 2134 (CWE-122) access program arguments without verifying existence, potentially causing CWE-476: NULL Pointer Dereference; additionally, three sites triggering a buffer overflow in test case 2009 have been reduced to one. Test cases 1573/1574, for CWE-122, do not check the return value of a number processing function (CWE-252), leading to unexpected behavior. Test case 10 for CWE-134: Uncontrolled Format String reads a large amount of memory from a parameter regardless of the latter’s size leading to CWE-126: Buffer Over-read and CWE-170; also, this test case uses the program’s arguments with-
out checking their number, potentially leading to CWE-476. Test cases 1831/1832, for the same CWE, do not always null-terminate a string (CWE-170) leading to CWE-126.

Test cases 1849/1856, 1850/1851, 1854/1855 and 2010/2012 for CWE-170 do not check the return value of reading functions properly (CWE-253: Incorrect Check of Function Return Value). Test case 1737 for CWE-244: Heap Inspection does not check if a memory re-allocation function fails, leading to CWE-690: Unchecked Return Value to NULL Pointer Dereference; also, one buffer is not freed on all paths (CWE-401). Test cases 1806-1809 for CWE-367: TOCTOU Race Condition fail to check the return value of writing functions (CWE-252). Test case 1926 for CWE-401 fails to check for successful memory allocation (CWE-252). This defect can also lead to CWE-415: Double Free. Test cases 1914 and 2270 also fail to check the return value of memory allocation.

3.2.2 Other Issues
Test cases 1843/1844 implementing CWE-122 allocate but do not free memory on all paths, leading to CWE-401. Test cases 93/1558 and 1559/1560 for CWE-134 both contain CWE-117: Improper Output Neutralization for Logs vulnerabilities. Test cases 1866, 1868 and 1870 for CWE-251: Misused String Management misuse a string copy function by failing to set a null terminator at the end of the destination buffer, causing CWE-170. Test cases 1810, 1835/2130, 1837/2131, 1839/2132 as well as 1841/2133 for CWE-259: Hard-coded Password prompt the user for a password but fail to erase it before release, leading to CWE-226: Unchecked Sensitive Information.

Test case 1588 for CWE-401 assigns a value to a variable that is never read, leading to CWE-563: Unused Variable. Test case 2200 for CWE-416: Use After Free contains two sites for this weakness. Test case 1757 for CWE-457 bears two sites for the weakness. In addition, 1875/1876, 1877/2194 and 2195 for CWE-476 and test cases 1751 and 1907-1910 for CWE-121 also contain CWE-563. Test case 2195 for CWE-476 also references a potentially uninitialized argument CWE-457.

Finally, test cases 1905/1906, 1845/1848 and 1828-1830 use a weak random number generator (CWE-392: Insufficient Entropy in PRNG).

3.3 Consistent: Complete GOOD/BAD Pairs
To improve consistent measurement we needed to ensure that every bad test case had an equivalent good. We discovered 16 pairs could be identified as indirect good/bad pairs, 15 test cases were identified as existing as pairs in SARD but not in the test suite, 27 test cases were created to establish good/bad pairs where none existed, 37 test cases had existing pairs, and 2 test case pairs were created from scratch. Resulting in 96 GOOD/BAD pairs in the new test suites.

3.4 Automation and Metadata Enhancements
While improving the accuracy, precision, and consistency were a primary focus, we also endeavored to make the test suites more consumable by researchers by providing consistent naming conventions, labeling, and metadata.

For automation, file names were updated with a suffix marking each test case as either "-good" or "-bad". Additionally, each test case was marked with comments to indicate the line where either "Flaw" or "Fix" was located.

<table>
<thead>
<tr>
<th>Extraneous Weakness</th>
<th>Incorrect Programs</th>
<th>Correct Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS45 / TS46</td>
<td>TS100 / TS101</td>
<td></td>
</tr>
<tr>
<td>10 1737 1914 1926 1930 2270</td>
<td>237 085 240 182 188 248</td>
<td></td>
</tr>
<tr>
<td>2009 2134</td>
<td>193/194 201/202</td>
<td></td>
</tr>
<tr>
<td>11^/2139 111/1640</td>
<td>241/242 053/054</td>
<td></td>
</tr>
<tr>
<td>1573/1574 1796/1797</td>
<td>069/070 069/096</td>
<td></td>
</tr>
<tr>
<td>1998/1999 1800/1801</td>
<td>097/098 099/100</td>
<td></td>
</tr>
<tr>
<td>1806/1807^ 1808/1809^</td>
<td>101/102 103/104</td>
<td></td>
</tr>
<tr>
<td>1831/1832 1849/1849</td>
<td>111/112 127/128</td>
<td></td>
</tr>
<tr>
<td>1850/1851^ 1854/1855</td>
<td>129/130 131/132</td>
<td></td>
</tr>
<tr>
<td>1881/2136 1883/2137</td>
<td>151/152 153/154</td>
<td></td>
</tr>
<tr>
<td>1899/2138 1899/1896</td>
<td>155/156 157/158</td>
<td></td>
</tr>
<tr>
<td>1899/1900 2010/2012</td>
<td>161/162 105/106</td>
<td></td>
</tr>
<tr>
<td>1858 1751 1757 1810 1828 1866 1868 1870 2195 2200</td>
<td>073 087 089 105 108 138 140 142 246 219</td>
<td></td>
</tr>
<tr>
<td>93/1558^ 1559/* 1560</td>
<td>047/048 061/062</td>
<td></td>
</tr>
<tr>
<td>1829/1830 1835/2130</td>
<td>109/110 115/116</td>
<td></td>
</tr>
<tr>
<td>1837/2131 1839/2132</td>
<td>117/118 119/120</td>
<td></td>
</tr>
<tr>
<td>1841/2133 1843/1844</td>
<td>121/122 123/124</td>
<td></td>
</tr>
<tr>
<td>1845/1848 1857/1856</td>
<td>127/128 134/135</td>
<td></td>
</tr>
<tr>
<td>1877/2194 1905/1906</td>
<td>243/244 165/166</td>
<td></td>
</tr>
<tr>
<td>1907/1908 1909/1910</td>
<td>167/168 169/170</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Extraneous Weaknesses found in TS45 and TS46 are removed in TS100 and TS101. Prefix 149 removed from TS100/TS101 test case identifiers for readability.

Our revisions to SARD also enhance the metadata associated with individual test programs. First, the new test suites had a positive impact on the NIST SARD, through the addition of a new feature called the ‘Association’ field. The new section allows users to navigate between good/bad test case pairs, explore replaced/replacing hierarchies of deprecated test cases, and jump to suites to which test cases belong.

We also adopted a systematic approach to describing each new test case with its specific characteristics for metadata information. Using descriptions from TS45 and 46 test cases as input, we passed the description content through several revisions to ensure correctness and consistency prior to addition to TS100 and TS101. Test cases that did not have a replacement pair (good/bad) in TS45 and TS46 derive their description from the negation of their good/bad pair.

During our review process we found that scanning TS45 and TS46 produced a list of test cases with incorrect syntax and missing library dependencies. In addition to numerous test cases not specifying their necessary includes, test case 2139 had issues related to its implementation. The issues in test case 2139 included the following: i) missing type specifier for variable ‘buffLength’ on line 21; ii) typo in ‘buffLength’ on line 27; iii) wrong attribution to ‘buf’ on line 34; iv) missing semi-colon at the end of line 35; and v) the missing underscore for ‘buff’ on line 38. SQL Injection test cases depend on ‘mysqlclient’ library and ‘mysql.h’ header in order to compile correctly. Finally, Cross-Site Scripting test cases also require the CGIc library. Such compile-time details may cause inconsistent results for different researchers. To help ensure that researchers obtain comparable results, compiler commands were included in the Instructions metadata section of test cases of TS100 and TS101.

Upgrading and replacing TS45 and TS46 led to the deprecation of test cases from these suites. Individual notes were added in the Comments section for test cases which have been replaced in the SARD by TS100 and TS101. Each
note provides the main reason why the test case was replaced and directs users to navigate between replaced/new test cases via the Association metadata field.

Authorship and copyright of test cases from TS45 and TS46 is attributed to Michael Koo, Romain Gaucher, and Fortify, among others. Accurate attribution of authorship is preserved in TS100 and TS101, by importing all authors from the original test cases and duplicating authorship when a good/bad pair was created. In addition, the original authors’ names are complemented by adding the names of individuals who were responsible for the improvements to the source code. In some test cases, which originated from TS45 and TS46, NIST is also listed as an author.

Test cases 1794, 1795, 1919-1924, 2198 and 2204 from TS45 and TS46 are categorized as CWE-79: Improper Neutralization of Input During Web Page Generation (XSS), creating a statement that accepts a user input from an HTML page variable q, then printing it out without a specific sanitizing filtering. This weakness’ documentation covers different types of XSS, one of them being “Reflected XSS (or Non-Persistent)”, representing the test cases above. CWE-80: Basic XSS, specializes CWE-79 and therefore we updated the metadata in the replacement test cases 149093, 149004, 149173-149178, 149215, and 149216 respectively in TS100 and TS101. The same approach was used for test case 149054 from TS101 replacing 1646. The older version implements a CWE-020: Improper Input Validation while the new one carries similar source code, but a more specific weakness category as CWE-78.

In Subsection 3.2 we corrected test cases 1806-1809 as having extraneous weaknesses. Furthermore, test cases 1807 and 1809 were identified as CWE-362: Race Condition, while their bad pairs 1806 and 1808, respectively, implemented CWE-367: TOCTOU. The new test cases 149102 and 149104 also fix the CWE mismatch.

The resulting programs form test suites 100 and 101 as replacements for test suites 45 and 46. In Table 4 we have removed the 149 prefix from all the test case identifiers, for readability, and list all of the test case programs for each targeted CWE. Test suite 100 (TS100) contains 96 programs with intentionally seeded vulnerabilities in odd numbered test cases. Test suite 101 (TS101) contains 96 "fixed" versions of the programs from TS100 in even numbered programs. For example, test case 148053 is a test program seeded with an OS Command Injection weakness. The matching pair, test case 149054, contains the same program with the weakness fixed.

4. EVALUATION

Experiments such as those presented in [4, 11], have shown that static analysis is an efficient approach for detecting possible vulnerabilities. In contrast to section 2.3.2, where manual code review and runtime monitoring were employed, we employ static analysis to evaluate the impact on measurement under the assumptions held pre/post verification and the replacement test suites.

To evaluate impact of the changes, the following subsections compare the results of scanning for weaknesses against the four test suites to evaluate the following three scenarios:

- Base: Consider assumptions of the test plan valid
- Validated: Based upon validated test suites
- Replacement: Based upon replacement test suites

The comparison is conducted using the commercial static analysis product HP Fortify SCA v6.10, under academic license, with the default 2014.1.0.0010 rulepacks. Both the base and validated scenarios are conducted against an unmodified version of TS45 and TS46. The replacement scenario is conducted against TS100 and TS101. These scenarios compare detection measurement results from the viewpoint of a software security assurance product, HP Fortify SCA, to detect the specified weakness categories under the three different scenarios. Measurements for comparison include the number of detected true positives, detected false positives, recall, precision, and F-Measure.

4.1 Base Scenario

This scenario holds certain assumptions under the context of the test plan defined in NIST SP 500-270 v1.1 [7]. For TS45 these assumptions include that each program contains an instance of its stated CWE and that the weakness is exploitable. Test suite 46 holds the assumption that for each test case, there will be a "fixed" version of a program that contained the stated CWE and that the weakness is not exploitable. For both test suites, we also assume that all of the test cases are included in the download from SARD.

Scanning TS45 and 46, under the assumptions of the test plan, correctly identifies 74% of the weaknesses seeded in TS45. As shown in the Base column of Table 5, 100% of the command injection, SQL injection, resource injection, heap overflow, format string, improper null termination, heap inspection, often misused string management, memory leak, double free, use after free, and uninitialized variable vulnerabilities are detected. Results for CWE-251 are reported by a stack-based or heap-based buffer overflow, in addition to the misused string management since CWE-251 is dangerous because it leads to buffer overflows (true in 100% of CWE-251 test cases). Even when CWE-251 is not exploitable, it is advisable to review as it could lead to future vulnerabilities.

The expected weaknesses are not reported for twenty test programs and are treated as false negatives. Specifically, basic XSS (CWE-80) is not detected in any of the five related test programs due to HP Fortify SCA not providing specific rule support for the CGIC APIs. Issues reported as CWE-121 are only considered true positives if the analyzer correctly identified a stack-based buffer overflow, with the exception of 1563 which uses the function gets which is never safe, at the correct location of the weakness resulting in a 67% detection rate. Hard-coded password, CWE-259, is detected in 40% of the possible five test programs. Upon further analysis, two of the three undetected test programs (1839 and 1841) would be extremely difficult for any static analysis approach to find correctly since the source code does not directly imply the intended business logic. The hard-coded string in the conditional could have nothing to do with a password, and could be any string comparison leading to any number of possible operations. For example, the following code from 1841 could be logic to determine if logging is enabled:

```c
static bool logged = false;
int main(int argc, char *argv[]) {
    for (unsigned i=1;i<argc && !logged;++i) {
        if (!strcmp(argv[i], "0xDEADBEF")) {
            logged = true;
            printf("Logged in\n");
        }
    }
    return 0;
}
```
None of the three TOCTOU (CWE-367) issues are detected, however, programs 1806 and 1808 appear to be designed to demonstrate CWE-362 rather than CWE-367. For several CWEs which only had a single test case, HP Fortify SCA did not report anything (CWE-391, CWE-412, CWE-468, and CWE-489). Finally, only two of the four Null Dereferences were reported (CWE-476).

Several CWE categories, shown in the Base column of Table 5, suffer high rates of false positives in TS46 (e.g. CWE-78, CWE-89, and CWE-90). The cause of the false positives is the nature of the detection mechanism. All of these weaknesses are detected by dataflow analysis, which tracks taint from a data input source to a potentially vulnerable sink where the tainted data is used. If the mechanism which removes the risk of the weakness is a custom function which modifies the tainted data, such as a white-list, then the analyzer would need to be informed via a custom rule that the particular taint being tracked is removed. Specifically, all of the SQL Injection fixes are accomplished using the function `mysql_real_escape_string` which can still be vulnerable if incorrect quotations are used. The OS Command Injection issues all use custom white listing function `check` or `purify`. Finally, Resource Injection issues also use custom white-listing function `allowed`. Since HP Fortify SCA is a "security review" product, it is debatable whether or not these issues are false positives, or not, and should be shown. The primary question is how one ensures that the business logic of the defensive function(s) truly removes all risk? Having a manual code review and penetration test to verify the functionality could lead to custom cleanse rules which remove the taint which is being tracked by the dataflow analyzer when program logic passes through the defensive function. If a custom rule is added, and the removal of taint is truly warranted, then these "false positives" will go away. However, adding cleanse rules to remove taint should be done with care since the removal of taint can lead to false negatives if the defensive function is insufficient (e.g. several defensive functions in TS46 are vulnerable as discussed in section 3).

### 4.2 Validated Scenario

If we assume that the test suites satisfied all the assumptions in the previous scenario, we would be almost done; however, our verification of TS46 required us to (in)validate those assumptions. Interesting results were found by the scanner in TS46, under the verified test suites, including the following:

- 11 unexpected exploitable vulnerabilities (1848, 1892, 1894, 1896, 1898, 1900, 1902, 2136, 2137, 2138, 2139).
- 9 false positives {1615, 1797, 1799, 1801, 1830, 1855, 1930, 2012, 2195}.
- 55 true negatives.

This brings the grand total to 68 true positives and 9 false positives being reported. In order to represent these validated test cases, we treat TS45 and TS46 as a union (see Validated column in Table 5 which highlights above changes). In addition, while verifying suspected vulnerabilities using runtime monitoring, an additional two unexpected vulnerabilities were also detected (1920 and 1924) in CWE-80, which are counted here but not detected since they would require custom rules (for a total of 13 unexpected exploitable test cases). To summarize, for the unmodified version of TS46, HP Fortify correctly identifies exploitable vulnerabilities in 11 programs, of the 13 identified in section 2.3.2, which are not supposed to contain weaknesses.

In particular, CWE-78 and CWE-99 results vary strongly from the test plan since the programs in TS46 were correctly identified as having weaknesses. The existence of the weaknesses in TS46, results in an absolute error of 13, yielding a possible percentage error of 18%. This error in TS46 explains the almost doubling of percentage of false positives
reported, from 15% to 27%, between the test suites in the Base and Validated columns of Table 5. These inaccuracies also cause the increase in true positive detection from 74% to 77% in the validated scenario.

Four Command Injections are reported in TS46 which should be false positives (as discussed in Section 3.1). On detailed analysis, however, these appear to be true positives as the added custom function (purify) that is intended to remove "";" is not sufficient to remove command injection vulnerabilities. As such these tests are still exploitable, re-validating assumptions (Validated), and TS100/TS101 white-list (in effect turning it into a black-list).

Finally, all four SQL Injection test cases in TS46 which are reported, from 15% to 27%, between the test suites in the Base and Validated columns of Table 5. These inaccuracies also cause the increase in true positive detection from 74% to 77% in the validated scenario.

Four Command Injections are reported in TS46 which should be false positives (as discussed in Section 3.1). On detailed analysis, however, these appear to be true positives as the added custom function (purify) that is intended to remove "";" is not sufficient to remove command injection vulnerabilities. As such these tests are still exploitable, re-validating assumptions (Validated), and TS100/TS101 white-list (in effect turning it into a black-list).

In addition, during the design of an application, an architect identifies various programming languages, frameworks, and APIs which satisfy the requirements of the project. Depending upon the architectural design and implementation decisions, a particular software security assurance product may, or may not, have the ability to detect a particular weakness. For example, the use of CGIC in the test suites made it initially appear that HP Fortify was not able to detect vulnerabilities. As such these tests are still exploitable, re-validating assumptions (Validated), and TS100/TS101 white-list (in effect turning it into a black-list).

Test suite 45 and 46 had deficiencies which required a thorough analysis, discussed in Section 3, resulting in the creation of TS100 and TS101 as replacements. We assert the results in the Replacement column of Table 5 provide a more accurate assessment of coverage, reporting 78% of all true positives and a realistic 23% for false positives. The improved accuracy is a direct result of the changes described in sections 3.1 and 3.3 which resulted in the fixing of 13 vulnerable test cases and creation of equivalent test cases grouped in GOOD/BAD pairs across the two new test suites.

The final metrics, displayed in Table 6, summarize the results of scanning under the three different scenarios. The base and validated scenarios show an 8% increase in the F-Measure and a 14% increase in precision which can be attributed to incorrect classification of vulnerable programs into a not-vulnerable bucket. The metrics for the replacement scenario and comparison.


<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>TP+FP</th>
<th>TP+FN</th>
<th>R</th>
<th>P</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>57</td>
<td>20</td>
<td>20</td>
<td>77</td>
<td>77</td>
<td>.74</td>
<td>.74</td>
<td>.74</td>
</tr>
<tr>
<td>Validated</td>
<td>68</td>
<td>9</td>
<td>20</td>
<td>77</td>
<td>88</td>
<td>.74</td>
<td>.88</td>
<td>.82</td>
</tr>
<tr>
<td>Replacement</td>
<td>75</td>
<td>22</td>
<td>21</td>
<td>96</td>
<td>96</td>
<td>.78</td>
<td>.78</td>
<td>.78</td>
</tr>
</tbody>
</table>

Table 6: Metrics for targeted CWEs by HP SCA in TS45 and TS46 under base scenario.

The last decade has seen great strides towards improving the maturity of software security assurance through both techniques and the ability to measure results. Test suites 100 and 101 are now available in the SARD as of April 29th, 2015, providing numerous improvements including the removal of targeted weaknesses in 13 test cases from TS45, removal of extraneous weaknesses in test cases from TS45 and TS46, addition of missing test cases for GOOD/BAD pairings across test suites, code updates for quality and automation, removal of bugs/“incorrect logic” (e.g. forking), updating of file names to indicate “good” and “bad” test cases, and marking of “Flaw” and “Fix” locations in the code. The new test cases also provide improved metadata on the SARD website including updated “replaces”/”replaced-by” mappings to improve associations, updated “pair” mappings to improve associations, updated descriptions, inclusion of compiler instructions for consistency of research results, notes explaining deprecations, and corrected CWE mappings.

Defining multiple input strings for every test case in SARD would also be beneficial to software security assurance activities. Specifically, if a test case contains a specific weakness, then example permutations which exploit the weakness would be beneficial. For example, for command injection, input strings demonstrating an exploit for each special character that can exploit the weakness would expose the latent defect (recall that the purify function only removed ‘ ‘).

Despite the significant improvements, these revised test suites still do not meet all of the requirements listed in the appendix of the specification and test plan documents since the specification lists 17 complexities, each of which has multiple permutations, across 21 CWEs[3, 7]. Continuous improvements to software security assurance techniques and measurement are still needed. For example, ignoring the permutations of enumerations for each complexity, absolute minimal coverage would require 130 unique test cases for evaluating true positives for each complexity over the 21 CWEs. Furthermore, if we assume that all of the complexities and their enumerations are complete, and that combinations of permutations involving different enumerations from different complexities are not considered, we would require at least 610 unique test cases just to cover one test case for each complexity and their enumerations defined in the appendix of NIST SP 500-268 v1.1.

6. REFERENCES