Scanstud: A Methodology for Systematic, Fine-grained Evaluation of Static Analysis Tools

Martin Johns
SAP Research
martin.johns@sap.com

Moritz Jodeit
n.runs AG
moritz.jodeit@nruns.com

***Abstract*—Static analysis of source code is considered to be a powerful tool for detecting potential security vulnerabilities. However, only limited information regarding the current quality of static analysis tools exist. A public assessment of the capabilities of the competing approaches and products is not available. Also, neither a common benchmark nor a standard evaluation procedure has yet been defined. In this paper, we propose a general methodology for systematically evaluating static analysis tools. We document the design of an automatic execution and evaluation framework to support iterative testcase design and reliable result analysis. Furthermore, we propose a methodology for creating testcases which can assess the specific capabilities of static analysis tools on a very fine level of detail. We conclude the paper with a brief discussion of our experiences which we collected through a practical evaluation study of six commercial static analysis products.**

I. Introduction

A. Motivation

The majority of today's security problem arise because of code-based vulnerabilities such as buffer overflows, SQL injection, or cross-site scripting (XSS) [4]. Consequently, the discipline of secure programming has been gradually shifted into the focus of many software producing entities. To help developers to find security bugs in their code, static analysis approaches which examine source code for vulnerabilities have been proposed. Furthermore, due to increasing demand in this area, several commercial products have been successfully introduced in the last years. However, the actual quality of the provided analysis is in large parts unknown. Especially, commercial tools are black boxes. Up to this day, no common benchmark has been defined and only very few public evaluation efforts have been undertaken – with NIST's SATE [20] project being one of the first steps in this direction.

B. Technical background

The BCS SIGIST defines **static analysis** of source code as the "analysis of a program carried out without executing the program" [24]. This definition emphasizes the contrast to dynamic analysis, where the behaviour of program is observed while it is executed. Analogously, a **static analysis for security** describes a class of programs which take source code as input and aim to find potential security problems in this code. Static analysis is not exclusively used for finding security problems. Applications can also be found, for instance, in compiler optimization or improvement of general code quality.

In general, static analysis problems are undecidable [11]. Thus, no analysis can be sound and complete. This means that a static analysis program is either unable to reliably detect all targeted problems or is prone to *false positives*, i.e., it reports findings which turn out to be wrong on closer examination. A further discussion about the implications and necessary trade-offs is given in [28] and [2]. In practice, most tools expose both variants of erroneous behaviour.

Available static analysis for security programs exist in the form of freeware [26] [25] [22], academic prototypes [5] [2] [12] [23], company-internal tools (e.g., Microsoft's Prefast [7] and Prefix [13]), and commercial products (see [18] for an overview).

II. Evaluating static analysis tools

A. Evaluation criteria

Before discussing existing and future approaches towards evaluating static analysis tools, it has to be decided which attributes and characteristics should be considered. Chess and West [3] list the following four criteria in this context:

- {**C.1**} **Quality of the analysis.** The quality of a given tool's analysis depends on the capabilities and precision of a tool's engine in respect to finding the targeted insecurities. This is determined by factors, such as language feature coverage, false negative rate (number off missed cases), false positive rate (number of false alerts per true finding), or comprehension of non-trivial control- and data-flows.
- {**C.2**} **Implemented trade-offs between precision and scalability.** Such trade-offs directly affect analysis time, false positive rate, and detection capabilities. For instance, depending on time- or memory constraints security flaws which in theory could be detected by a tool's engine are missed because the analysis was terminated prematurely.
- {**C.3**} **Set of known vulnerability types.** It has to be measured, if the tool is aware of all relevant types of potential, code-based vulnerability types.
- {**C.4**} **Usability of the tool.** This is a mostly non-technical requirement. In the context of this paper, we do not consider usability aspects.

For the remainder of this paper, we will reference these four criteria to assess discussed evaluation approaches.

B. Existing approaches towards benchmarking

In general there are three approaches towards practically evaluating the quality of a static analysis tool for security. They mainly differ in the choice of source code which is utilized within the evaluation.

Real-world, vulnerable software: In general, applying a static anylsis tool to a non-trivial, real-world software project which contains at least one previously known security vulnerability is the first approach towards assessing its quality. Typical candidates for such an evaluation are large-scale open source applications, such as Sendmail or WU-FTPd. For example, Livshit's SecuriBench [14] follows this approach.

This evaluation method has several shortcomings. For one, the only available success criteria is to verify that the previously known vulnerability was detected by the tool. However, in cases in which the vulnerability was not found, there is no indication why the test failed. Was the tool simply not capable to comprehend the causing controlflow (criterion C.1)? Was the vulnerability hidden too deep in the call-stack, so that the analysis terminated prematurely (criterion C.2)? Or is the given vulnerability type simply not contained in the tool's set of targeted bug classes (criterion C.3)? Analogously, even if the vulnerability was found, there is also only little evidence how or why the test succeeded. Consequently, the amount of knowledge about a tool's capabilities that can be gained using this approach is limited.

Besides checking for alerts which may stand in direct relation to the previously known insecurity, the further option is to investigate the additional warnings that the tool reported in respect to the source code. For each of these warnings, it can be examined, if the finding indeed constitutes a vulnerability or a mere false positive. While this process would provide insight concerning the tool's false positive rate, to do so is very time consuming and potentially error-prone.

Furthermore, assessing the tool's coverage of known vulnerability types (criterion C.3) is difficult to achieve. It is unlikely to find a real-world application that contains instances of all applicable vulnerability classes. Consequently, it is necessary utilize several applications. This in turn multiplies the already significant benchmarking effort.

Finally, this approach offers no indication on false negatives in respect to yet unknown vulnerabilities in the source code.

Nonetheless, this approach is well suited to assess the runtime behaviour of the tool (criterion C.2) in respect to realistic application complexity.

Educational applications: Several testing applications exist which have been written to contain security problems on purpose, e.g., OWASP's WebGoat [21], or Foundstone's Hacme series [17]. In general, they are designed as self-contained applications which often mimic a well-known use case, e.g., an online banking site. These applications are mainly used for educational practices, teaching developers about software security practices by exposing them to vulnerable code. The developer can interact with the application and, this way, try to find and exploit the crafted vulnerabilities.

Compared to real-word applications, using the source code of such programs for evaluation purposes can offer some slight advantages. For one, educational applications provide a fairly comprehensive coverage of vulnerability types (criterion C.3). For example, WebGoats's webpage lists more then 12 distinct bug classes [21]. Consequently, applying a static analysis tool to WebGoat's code might offer insight, if the tool masters these individual vulnerability types.

Furthermore, the ratio between true vulnerabilities and lines-of-code is usually rather high while the software's overall complexity is comparatively low. In addition, the source code's documentation probably focusses on the included vulnerabilities. All of this helps to assess the manual inspection of the analysis results.

Micro benchmarking suites: Finally, the third option is the usage of a specifically designed micro benchmarking suite. Such test suites consist of a set of mini-tests which contain one or more security vulnerabilities each. In contrast to educational applications, these suites are written specifically to evaluate static analysis tools. Hence, they do not represent complete applications. They are not even necessarily executable [27].

Micro benchmarks allow fine-grained probing for specific individual analysis capabilities (criterion C.1). Furthermore, testing for coverage of vulnerability classes (criterion C.3) is not an issue. This level of control over the tested features provides the designer of the suite with the capability to execute an investigation of a tool's analysis engine. E.g., if a test fails, the test can be iteratively modified to examine the reason which caused the failure.

Also, the only source code contained in such suites either represents a testcase or is necessary to host the test. No further user interface or application logic exists. This leads to a small lines-of-code total and a very high vulnerability ratio. These characteristics carry the significant benefit that manual inspection of a tool's report is feasible with moderate effort.

However, in consequence, the overall complexity of the resulting test programs is lower compared to real-life applications. This limits obtaining evaluation results in respect to runtime behaviour and scalability/precision trade-off (criterion C.2) which may be encountered under realistic circumstances.

For the reasons listed above, we chose to design a testing methodology which is based on micro benchmarking suites.

C. Objectives

We regard the following characteristics to be essential for a systematic evaluation system:

1) **Targeted testing of individual capabilities:** Our goal is to learn as much as possible about a tool's strengths and weaknesses. Hence, a targeted testing for individual capabilities of a given tool has to be feasible. We propose to achieve this through designing single purpose testcases which have a very narrow, clearly defined scope. A single testcase should ideally test only for one specific characteristic. Consequently, the proposed benchmarking system should support the creation and evaluation of a large set of individual testcases.

We will discus our approach towards testcase design in depth in Section IV of this paper.

2) **Easy, correct, and iterative testcase creation:** The quality of the utilized testcases determines the overall quality of the complete evaluation. For this reason, it is of great importance that the testcode is correct. I.e., it has to be verified that testcode which was designed to contain a specific vulnerability is indeed vulnerable. Otherwise, the validity of the complete benchmarking effort is questionable if the correctness of the tests is in doubt.

   More precisely, our evaluation system should allow the creation of testcases that are short and as human readable as possible. Furthermore, the system should support manual verification of the existence/absence of the targeted vulnerability. Only manual verification provides a suitable level of confidence for our purpose.

3) **Automatic test execution and result evaluation:** Repeatedly running a given tool can be a complex, time consuming, and tedious task. However, during testcode development, it is advisable to iteratively run the benchmarked tools against the originating test suite. This way, early mistakes and pitfalls, that might occur during testcode development, can be detected and avoided. Also, an extension and modification of the benchmarking suite should be easily possible. This way, the test code can be adapted and refined according to earlier results. Thus, the tools can be benchmarked in depth through follow-up tests.

   For this reason, we regard it as essential, that the actual test execution and the evaluation of the resulting reports is automated. As such an approach makes running the benchmarking system cheap for the developer, it allows convenient, iterative testcode development. Furthermore, such an approach provides an evaluation of the tool's reports which is neutral and free of human error.

In the following sections, we discuss how our approach satisfies the here stated objectives.

III. Scanstud: Test methodology

In this section we discuss how we designed our Scanstud framework to fulfill the in Sec. II-C stated objectives.

A. Reliable result evaluation through dedicated applications

Automating the evaluation of a tool's reports is not trivial. The main challenge is to correctly map the tool's individual findings to the crafted vulnerabilities which are contained in the test code.

Simply matching the line number of the reported finding with the line number of the vulnerability is not sufficient: A specific vulnerability's line number in the source code is not always obvious. Take for example Listing 1 which exemplifies a common cause for off-by-one vulnerabilities in C. Which line should be reported? Line 3 in which the off-bounds writing operation is located? Or line 2 which hosts the causing, faulty bounds-check? Both choices would be reasonable. However, if the creator of the tool decides that line 2 should be reported while the benchmarking system checks if line 3 is flagged, the benchmarking system is unable to notice whether the tool in fact correctly detected the vulnerability.

```
1 [...]
2 for (i = 0; src[i] && (i <= sizeof(dst)); i++) {
3     dst[i] = src[i];
4 }
5 [...]
```

Listing 1. Off-by-one vulnerability caused by insecure loop condition

Therefore, a mapping mechanism that does not rely on line-numbers is crucial. We achieve this as follows: Every testcase is hosted in a separate, dedicated application. This application contains only one single testcase, either a true vulnerability or a crafted false positive. Within this application's source code we differentiate between the *testcode* and the *host program*. The *host program* is a stub which contains all necessary components to promote the *testcode* to a parsable, compilable and executable application.

The host program is designed to be completely error free. This should result in an absence of warnings in respect to source code portions that are not part of the testcode. Consequently, any finding that a tool alerts has to be directly related to the testcode. Hence, if a tool detects a problem, the crafted vulnerability was correctly found with a very high probability (or the intended false positive was alerted). Obviously, sanity checking of tool's warnings has to be applied, e.g., for cases in which a tool falsely detects a completely wrong bug type or more than one vulnerability is reported. Applied to the example of Listing 1, as long as the tool reports an off-by-one error for the tested application, the testcase is considered to be detected correctly.

B. Removing false positives caused by the host program

In theory, the evaluation methodology outlined above provides an elegant way to assess a tool's output. However, in practise, there is a common problem: No matter how much effort is dedicated for removing all possible suspicious code portions from the host program's code, the tendency to report false positives of certain tools may still result in unnecessary warnings which are not related to the testcase.

To address this problem, we implemented a simple, yet effective pre-processing step, called *Scanstud Diff*, to isolate the false positives and other noise caused by the host program. We create two versions of the test application, one containing the testcode and one in which the testcode is omitted or replaced by something harmless. Both applications are processed by the to-be-evaluated static analysis tool. The resulting reports are compared. All findings which occur in both reports are necessarily caused by the host program's code which is identical for both applications. Thus, these warnings are removed from the report. Consequently, the remaining findings are reliably caused by the testcode.

C. Automatic creation of test applications

The separation between testcode and host program has an additional benefit. The actual testcode is comparatively small

and clearly defined. As motivated in Section II-C, it is crucial to ensure that the testcode correctly contains the targeted vulnerability. As all support code, which is necessary to create a complete application is placed in the host program, the designer of the source code can fully focus on creating the testcase.

To take full advantage of this, our approach proposes the usage of an universal host program which is utilized by all test applications. Such a host program contains all the support infrastructure which is required by the testcases. For instance, in the case of our C-language test suite (see Section V-A2), the host program provides a simple TCP server that reads untrusted data from an open socket and passes this data to the testcases.

The testcases are kept separately in small template files. These files only contain the code which is directly associated with the testcase. The host program contains clearly defined markers in which the testcode is inserted.A small assembly script combines the template files and the host program into the test applications. This way, a large number of dedicated test applications can be created effortlessly. Furthermore, the actual testcase's code is small and of manageable complexity, enabling the testcase designer to achieve a high level of confidence regarding the test's correctness.

D. Support for manual verification

We emphasised in Section II-C the importance of practical verification of a testcase's vulnerability. For this reason, the host program should be executable and provide means to interact with the testcode. This way, the testcase designer can practically trigger the crafted vulnerability to ensure that he indeed wrote insecure code. Especially, with complex vulnerabilities which utilize non-trivial data- and control-flows, such a manual verification step is crucial.

For instance, in our Java-language test suite (see Sec. V-A1), every test application is indeed a full J2EE application, which exposes the testcode in the form of an servlet. Furthermore, the host program provides a suiting HTML frontend which allows the designer to directly access the vulnerability.

E. Summary: The Scanstud testcode assembly architecture

Each testcase is assembled by combining the testcode with the host program. The resulting application is passed on to the to-be-evaluated static analysis tool, which is integrated into the automatic testing infrastructure by a tool wrapper. The analysis' result is diffed against the scan result of the benign host program in order to remove potential noise from the analysis. Then the result is recorded for the final evaluation.

This infrastructure allows unobserved, automatic execution and evaluation of the benchmarking effort. Furthermore, focused testcase development is fostered through the separation between test and support code.

IV. Scanstud: Testcode

A. Testcase design

Within our approach we differentiate between the terms *test* and *testcase*:

- A *testcase* is the smallest unit in our approach. It is designed to check for a single, specific capability of the tested tool, e.g., "does the tool detect unsanitized data which is routed through an array?" (see Listing 2). A testcase is passed if the vulnerability was correctly found or, respectively, if the intended false positive was correctly ignored.
- A *test* consists of one or more testcases. These testcases are compiled to determine if the tested tool does master a specific concept, e.g., "does the tool understand the semantics of dataflows which are routed through an array?" A test is passed when all belonging testcases were passed.

To illustrate this distinction, consider the array example of Listing 2: To determine if the tested tool actually correctly comprehends the given dataflow, the testcase sketched in Listing 2 alone is not sufficient. For instance, a tool that simply alerts all `println`-calls with non-constant parameters to be potentially vulnerable[1] would also alert this case without correctly analysing the array semantics. Therefore, to establish whether a tool correctly understands the given dataflow, we also have to test if properly sanitized data entered in the same dataflow remains unannounced (see Listing 3). A "dumb" tool that alerts all `println`-calls would falsely flag this snippet to be potentially vulnerable, thus, showing that the array dataflow indeed was ignored. In this example, the two exemplified testcases together constitute one single *test*. The test in only passed, if both testcases are interpreted correctly, i.e., the vulnerable code is alerted and the false positive is ignored.

```
1 String[] arr = new String[2];
2 arr[0] = request.getParameter("data");  // unsanitized
3 response.println(arr[0]);               // XSS vuln.
```

Listing 2. Exemplified testcase: Vulnerable dataflow through an array

```
1 String[] arr = new String[2];
2 arr[0] = HTMLEncode(request.getParameter("data"));
3 response.println(arr[0]);  // safe
```

Listing 3. Exemplified testcase: Safe dataflow through an array

Whenever applicable, our *test*s consist of two *testcases* both checking the same capability. One represents a true vulnerability, the other one a false positive. This way it can be ensured, that the true finding was not found by accident but indeed because the tool interpreted the source codes semantics correctly.

B. Testcase categories

In the context of a tool evaluation with our testing framework, a given test falls into one of the following categories:

- It tests either for the completeness of the tool's set of known vulnerability types,
- for the degree of the provided language coverage,
- or for the capability to comprehend the semantics of specific control- and dataflows.

In the remainder of this section we briefly explore these three categories. Please note: These categories are not mutually

[1]For instance, tools such as Rats [22], Flawfinder [26], or ITS4 [25] check for suspicious API calls and, e.g., flag every single occurrence of `strcpy`.

exclusive. A given test may include characteristics from more than one class. However, as our framework aims to provide fine-grained insight concerning a given tool's functioning, it is important that the designer decided what the focus of the given testcase is.

1) Vulnerability class coverage: This is the most basic test category. Using simple testcases, it is verified that the tool indeed checks for the probed class of security vulnerabilities. I.e., we check whether the tool alerts bug classes such as buffer overflows, off-by-one errors, format string vulnerabilities in C programs, or XSS, SQL Injection, etc. in Java applications.

For this purpose, the testcases should contain a representative of the bug class in its "purest form". This means, the testcode should contain the minimal set of instructions which would cause such a vulnerability, omitting all sophisticated language features which might cause the tool to fail the analysis.

Example: See Listing 4 for our testcase concerning buffer overflows in C. Please note, that even this simple testcase requires the tool to master two concepts which are not directly related to the probed bug class. For one, the tool has to comprehend the semantics of the `strcpy`-method. Furthermore, the tool has to be capable to do basic interprocedual reasoning, as the untrusted data is passed through the argument `*bad`. In the context of our benchmarking effort, we consider it to be reasonable to assume that a mildly advanced static analysis tool for security possesses both capabilities.

```
1 void buffer_overflow_test(char *bad) {
2         char buf[1024];
3         strcpy(buf, bad);
4 }
```

Listing 4. Simple buffer overflow coverage testcase

2) Language feature coverage: Testcases in this category actively examine if a tool understands basic features of the targeted programming language. Such features include, for instance, the semantics of native datatypes, the functioning of methods provided by the standard library, or the effects of advanced language features, e.g., inheritance or scoping.

A test in this category is designed as follows: To test if a tool understands a given language feature, the corresponding testcase utilizes this language feature within a source code that contains a vulnerability. Inclusion of a vulnerability in the testcase is necessary as the evaluated tools are specifically designed to find security problems.

Hence, a basic vulnerability type which is contained in the tool's set of know bugs is chosen. Using our methodology explained in Sec. IV-A, two testcases are created using an instance of this bug: The first testcase is vulnerable. The second testcase, in most parts, consist of exactly the same source code as the first testcase, with the only difference that the vulnerability is fixed. Both testcases contain the targeted language feature as a necessary component in their code. If the tool finds the vulnerability and ignores the safe variant, it is reasonable to assume that the semantics of the examined language feature are understood by the tool. See Listings 2 and 3 for an example which probes if the tool comprehends arrays.

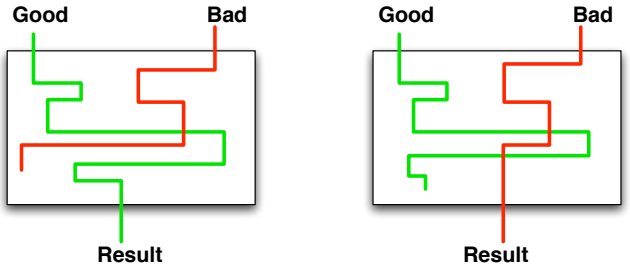
This category of tests is very important to assess potential, unexpected side effects: If a tool does not understand language feature *X* then it is not able to correctly interpret all further testcases which utilize *X*. This is especially problematic with testcases that were designed to probe for a tool characteristic (for instance *Y*) which is not related to language feature *X*. If the testcode uses *X*, the testcase fails even though the tool may actually understand *Y*.

Example: During our practical evaluation of our framework (see Sec. V) we tested a commercially available static analysis tool which claimed to find web application related security problems in Java source code. However, during our tests the tool was not able to correctly alert a single instance of our XSS testcases. This puzzled us, especially, as the tool otherwise exposed decent results. After further investigation, the source of the problem was isolated: Our XSS testcases used J2EE's `response.getWriter()`-method for writing data to the HTML output. However, the tool did not comprehend this specific call. Fortunately, due to the compactness of our actual test code and our framework's support for repeated and automated evaluation rounds, we were able to adapt our testcode quickly. After we replaced all occurrences of `getWriter()` with `getOutputStream()`, several of our XSS testcases were alerted correctly. While the failure to understand the `getWriter()`-method is a serious flaw of the tool, the overshadowing and falsifying of the other testcases which was caused by this was not desirable.

Consequently, if possible, such language coverage testcases should be written and evaluated early, before the rest of the benchmarking suite is designed. This way unwanted side effect because of incomplete coverage of language features can be avoided.

3) Control and dataflows: Tests contained in this category are closely related to the former class of testcases which check for language features. The here summarized tests aim to assess a tool's ability to comprehend non-obvious control- and dataflows. Again, as it was the case with testing for language features, the only way to test for such capabilities is to create insecure source code which incorporates the targeted control- or dataflow in its execution path. Therefore, it is necessary to ensure that the evaluated tool is able detect the utilized vulnerability class in the first place. We illustrate this point (which also applies the language feature testcases) with a further experience from our practical experiments: Within our Java suite, we used XSS vulnerabilities in testcases which probed for the ability to assess non-trivial dataflows. However, one of the evaluated tools did not check for web application specific vulnerabilities, such as XSS. For this reason, a subset of our testcases was not applicable for this specific tool.

We propose the construction of tests within this category to be conducted as follows: First a basic vulnerability class is chosen, which is triggered when an untrusted, attacker-controlled value is used in a security sensitive context. For instance, XSS bugs for web applications and formatstring vulnerabilities for C programs are suiting candidates. The constructed testcase contains two data sources, i.e., variables



A. Safe dataflow B. Vulnerable dataflow

Fig. 1. Utilizing dataflows for testcase design.

which are filled with dynamic values. One of the sources is *safe*, this means the dynamic value is properly sanitized and the other source is *unsafe*, i.e., controlled by the attacker. Furthermore, the code has one sink, a variable that ends up being used within the potentially security sensitive construct. Then, a control- or dataflow is constructed containing the targeted characteristic. Both sources enter the execution flow. Within this flow, one of the sources is assigned to the sink, while the other is either discarded or used in a security insensitive context. Depending on which of the both sources gets assigned to the sink, the testcase is a true vulnerability or a potential false positive (see Listing 5 and see Fig. 1 for examples). Only if the tool correctly comprehends the execution flow which is used within the test, it is able to pass both testcases.

```
String bad    = req.getParameter("testpar");
String good   = HTMLEncode(req.getParameter("testpar"));
String result = "";

int i1 = 0;
for (int i2 = 0 ; i2 < 10 ; i2++){
    i1 += 1;
}
if (i1 == 9) {
    result = bad;
} else {
    result = good;
}
writer.println("<h3>" + result + "</h3>"); // XSS?
```

Listing 5. Testcase to evaluate if loop invariants are calculated

Such tests include for instance:

- **Controlflow:** Calculation of loop invariants, cascading conditionals, dead code detection, interprocedual execution flows, handling of side effect of object state, or exception handling.
- **Dataflow:** Flows through local and global buffers, pointer arithmetic, aliasing, data flow through local classes (e.g., custom linked lists), or second order code injection through temporary, local storage.

We consider these tests to be most significant when it comes to assess the actual analysis capabilities of a given tool. While shortcomings in bug class and language feature coverage probably can be easily corrected in future versions of the tool, the ability to correctly analyse sophisticated flows is a direct characteristic of the underlying analysis engine. Furthermore, a non-trivial program very likely contains numerous advanced execution paths.

V. Implementation and Practical Evaluation

We implemented our benchmarking framework according to the design considerations outlined in Section III. More specifically, we created scripts for automatic testcode assembly, tool execution, result diffing, and result evaluation. Furthermore, we designed two test suites [8], one targeting web application specific vulnerability classes in Java programs and one containing C language specific vulnerabilities.

A. Language specific testing suites

1) Java suite: Our suite of tests in the Java programming language was designed as follows: The host program consists of a J2EE web application. This application contains a single servlet. Within this servlet a small number of placeholders exist which function as insertion points for source code from the testcase templates. Within these templates, the actual testcases is encapsulated in Java classes which export the method `doTest()`. During the evaluation process, the framework iteratively uses the individual testcase templates and merges them with the host program into a complete and compilable application source code.

Most of web application specific bug classes, such as XSS or SQL injection, occur because of insecure dataflows. Consequently, the majority of the testcases contained in this suite target non-trivial dataflows. In total, the suite consists of 85 testcases (or 49 tests respectively).

2) C suite: In general, the C suite was designed similar to the Java suite. The host program is a simple TCP server which reads data from a socket and passes pointer to test code. It consists of less then 100 lines of code.

Within C programs, the majority of vulnerabilities arise due to memory mismanagement problems. For this reason, the C language test suite emphasises on such problems. Thus, we designed testcases concerning bug types such as buffer overflows, unlimited/Off-by-one pointer loop overflows, integer overflows/underflows, signedness bugs, or null pointer dereferences. In total, the suite consists of 116 testcases (or 57 tests respectively).

B. Practical experiences

To verify the practical applicability of our approach, we conducted an evaluation study of several competing tools.

The majority of the freely available tools either only test for a very narrow set of bug classes (e.g., [5], [23], [2]) or do not possess sufficient capabilities for interprocedual analysis (e.g., [26], [25], [22], [5]) which is a necessary prerequisite for our testing methodology (see Sec. IV-B). Consequently, these tools were not suitable to test-ride our framework.

Therefore, we chose a set of six commercial static analysis products as candidates for the project. Unfortunately, due to the various agreements which we had to sign with the tool's vendors, we are not at liberty to publish the results of this study. However, we do not consider the exact results to be terribly interesting. They only document a snapshot of the current capabilities which is outdated with the next releases of the individual tools. Instead, in the remainder of this section

we briefly document our general experiences in respect to the evaluation process. Furthermore, as we have published the source code of our testing framework and our testcases [8], interested parties can run our benchmarks themselves.

Some brief observations on analysis quality: In the remainder of this section we list selected observations, to exemplify potential insights which can be gained using the proposed methodology[2]:

As expected, the tools exposed individual strength and weaknesses in respect to analysis quality. In general, a tendency to favour false positives over false negatives could be observed. It appeared in some cases, that when in doubt certain tools opted for a warning as default behaviour, which is a reasonable policy from a security point of view.

Furthermore, we could make the following observations: Within our C suite, tests containing double-frees and null-pointer dereferences were passed by most tools, However, most tools had significant problems with non-trivial integer overflow vulnerabilities. Concerning Java testcases, a general area of strength was the tracking of dataflows within a single function. However, dataflows that left the local scope often caused problems. For instance, non of the tools was able to pass tests involving a custom-written linked list object (see Listing 6).

```
1 class Node {
2    public    String value;
3    public    Node   next;
4 }
```

Listing 6. Custom linked list object used in testcases

VI. Related work

In this section we list related benchmarking efforts. With the exception of NIST's SATE (see below), we focus on approaches that utilize micro benchmarking suites.

To our best knowledge, the first systematic benchmarking study concerning static analysis for security was conducted by Wilander and Kamkar in 2002 [27]. Within their study, they utilized a micro benchmark which consisted of a single C file containing a total of 23 insecure and 21 safe API calls. No tests for language feature coverage or advanced analysis capabilities were included.

Conceptually closest to our approach are the micro benchmarking suites by Livshits [15] and Kratkiewicz [9]. Both are well designed suites, consisting of numerous small testfiles. [15] targets analysis tools for web application vulnerabilities by providing a total of 96 testcases encapsulated in Java classes. The majority of the provided tests aim to asses the evaluated tool's capabilities for language feature coverage. [9] exclusively focuses on buffer overflows in C applications. For this purpose, the suite consists of 291 tests, modeling buffer overflows in all imaginable variations. Compared to our approach, neither [15] nor [9] provide support for automatic test execution or result evaluation.

[2]NB: These observations were initially made early 2008. As we shared our results with the vendors, it is likely that the identified shortcomings have been addressed in the mean time.

Zitser et al. document in [29] an evaluation of four open source and one commercial static analysis tool. Initially, the authors aimed to utilize three large-scale real-world applications with disclosed vulnerabilities (BIND, WU-FTPD, and Sendmail) for their project. However, early test-runs exposed that some of the tools were not able to analyse the chosen test applications due to non-trivial code constructs, such as custom type definitions. For this reason, Zitser et al. implemented 14 small test applications with sizes ranging between 90 to 800 lines of code. These applications were designed to closely model existing vulnerabilities from the initially chosen set of test applications. This was done by extracting just as much code as it was required to reproduce the respective vulnerability. Analogous to our approach, each test application was implemented both in a vulnerable and in a safe variant. Their proposed approach towards testcode assembly is compelling. It is situated in between evaluation using micro benchmarks and real-life software. However, it still has to be shown that all relevant characteristics of static analysis tools can be assessed this way. Furthermore, compared to our approach, creating testcases requires significantly more effort: Real-life vulnerabilities have to be found, analysed, and extracted. Finally, it is debatable if modeling the test after existing vulnerabilities has actual advantages over completely crafted testcases.

In [10] Ku et al. briefly document a benchmarking suite which was designed following Zitser et al.'s approach. The suite consists of a total of 298 test based on 22 analysed vulnerabilities from 12 programs. The suite was designed to help the development of a novel analysis tools. Hence, no information of general evaluation methodology or results have been published.

In 2008 the NIST initiated the first Static Analysis Tool Exposition (SATE) [1], a public evaluation project which invited authors and vendors of static analysis tools to participate. The stated goals of the evaluation were to enable empirical research based on large test sets, to encourage improvement of tools, and to speed adoption of the tools by objectively demonstrating their use on real software. For this purpose, SATE chose six real-life open source applications with expected security vulnerabilities and at least several thousand lines of code, as analysis targets – three written in C, three written in Java. Furthermore, a common XML dialect for the participating tool's reports was defined.

SATE ended in May 2008 with the sending of the tool's reports to the organisers. However, it took more than a year to publish the final assessment of these reports [20]. The reason for this delay was that the preparation of the final evaluation took longer than anticipated. This way, the SATE project indirectly supports our claim that manual inspection and evaluation of non-trivial result-sets produced by static analysis tools demands significant resources and, thus, is not suitable for benchmarking efforts. In comparison, our evaluation mechanism allows unobserved, automatic report assessment. As our testcases are small and well defined, manual investigation is only necessary in limited, conspicuous

cases. However, subjective criteria can better be solved through manual report evaluation, e.g., estimating if the utilized warning levels concerning the findings are appropriate.

In 2009, NIST undertook a second public tool evaluation (SATE2009) [19]. Unlike the first attempt, this time only a subsets of a tool's warnings was chosen for further investigation. This way, a timely analysis of the result set could be achieved. However, the outcome's meaningfulness is subject to debate as not all warnings have been examined. In contrast, our approach guarantees a full assessment of all warnings both in respect to true findings and false positives.

Besides SATE, several further benchmarking efforts utilizing pre-existing, real-life software projects as evaluation targets (e.g. [16] or [6]).

VII. Conclusion

In this paper we presented Scanstud, a methodology for systematic evaluation of static analysis tools for security. The Scanstud methodology is twofold:

For one, it defines an automatic test execution and evaluation framework. In this framework, we achieve reliable correlation between tool report and expected testcase answer. This is done through encapsulating individual testcases in dedicated applications and through clean separation of test and support code.

Furthermore, Scanstud proposes a general approach towards benchmarking suite design which relies on fine-grained, targeted tests. A given test within Scanstud probes for an clearly defined, singled-out characteristic of the evaluated tool. Such characteristics include knowledge of vulnerability classes, comprehension of language features, or possession of analysis capabilities in respect to defined types of control/dataflows. By combining two testcases within one test, in-depth insight regarding the evaluated tool's capabilities can be gained.

In conclusion, Scanstud offers sophisticated and flexible capacities for tool evaluation combined with a high degree of confidence in the obtained results. An wide adoption of our methodology would benefit identifying existing limitations in the current state of the art in static analysis and, hence, foster further improvements in this area.

VIII. Acknowledgments

We would like to thank the Siemens CERT (esp. W. Koeppl and M. Wimmer) for supporting the effort. This work was in parts funded by the EU project WebSand (FP7-256964).

References

[1] Paul E. Black. SAMATE and Evaluating Static Analysis Tools. *ADA User Journal*, 28(3):184 – 189, September 2007.

[2] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, 2004.

[3] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[4] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE, Version 1.1. [online], http://cwe.mitre.org/documents/vuln-trends/index.html, (09/11/07), May 2007.

[5] D.Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS 2000*, 2000.

[6] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 41–50, New York, NY, USA, 2008. ACM.

[7] Michael Howard. Inside the Secure Windows Initiative. [online], http://technet.microsoft.com/en-us/library/cc723542.aspx, January 2000.

[8] Martin Johns and Moritz Jodeit. Scanstud: Framework and testcases. [software], http://web.sec.uni-passau.de/research/softwaresecurity/scanstud/, 2009.

[9] Kendra June Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's thesis, Harvard University, 2005.

[10] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392, 2007.

[11] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323 – 337, December 1992.

[12] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilites. In *USENIX Security Symposium*, 2001.

[13] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Faehndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[14] Benjamin Livshits. Defining a Set of Common Benchmarks for Web Application Security. Workshop on Defining the State of the Art in Software Security Tools (position paper), August 2005.

[15] Benjamin Livshits. Stanford SecuriBench Micro. [software], Version 1.08, http://suif.stanford.edu/~livshits/work/securibench-micro/index.html, May 2006.

[16] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[17] McAfee, Inc. Hacme bank v2.0. [software], http://www.foundstone.com/us/resources/proddesc/hacmebank.htm, May 2006.

[18] National Institute of Standards and Technology (NIST). List of Source Code Security Analyzers. [online], Software Assurance Metrics And Tool Evaluation Project, http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html, July 2005.

[19] Vadim Okun, Aurelien Delaitre, and Paul E. Black. The Second Static Analysis Tool Exposition (SATE) 2009. National Institute of Standards and Technology (NIST) Special Publication (SP) 500-287, June 2010.

[20] Vadim Okun, Romain Gaucher, and Paul E. Black. Static Analysis Tool Exposition (SATE) 2008. NIST Special Publication 500-279, http://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf, June 2009.

[21] Open Web Application Project. OWASP WebGoat Project. [software], Version 5.2, http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, July 2008.

[22] Secure Software, Inc. RATS - Rough Auditing Tool for Security. [software], http://www.securesoftware.com/resources/download_rats.html, 2001.

[23] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[24] Specialist Interest Group in Software Testing (BCS SIGIST). Glossary of terms used in software testing. BSC Working Draft, BS 7925-1, Version 6.3, http://www.testingstandards.co.uk/bs_7925-1_online.htm, 2006.

[25] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, 2000.

[26] David A. Wheeler. Flawfinder. [software], http://www.dwheeler.com/flawfinder, 2001.

[27] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Static Intrusion Prevention. In *7th Nordic Workshop on Secure IT Systems (Nordsec 2002)*, 2002.

[28] Y. Xie, M. Naik, B. Hackett, and A. Aiken. Soundness and its role in bug detection systems (position paper). In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[29] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, 29(6):97 – 106, 2004.