ABM: A Prototype for Benchmarking Source Code Analyzers

Tim Newsham*
Waipahu, Hawaii USA
newsham@lava.net

Brian Chess
Fortify Software
Palo Alto, California USA
brian@fortifysoftware.com

ABSTRACT

We describe a prototype benchmark for source code analyzers. The prototype uses a combination of micro- and macro-benchmarking to measure the vulnerabilities a tool is capable of detecting and the degree to which it is able to distinguish between safe code and vulnerable code. We describe the design and implementation of our prototype, then discuss the effect that the our experience with the prototype has had on our future goals. Our prototype, along with sample output from a number of source code analysis tools, is available for download from http://vulncat.fortifysoftware.com.

1. INTRODUCTION

Static source code analysis provides a mechanism for reducing the amount of tedious work involved in inspecting a program for security vulnerabilities. As source code analysis grows in popularity, more potiential users of the technology are faced with the need to evalute the pros and cons of an increasing number of tools. A formal benchmark for comparing source code analyzers would provide several benefits: A benchmark would help consumers choose the best tool for their needs. It would pinpoint weaknesses in existing analyzers. It would quantify the strengths and weaknesses of competing analysis techniques and allow engineers to make measured tradeoffs. Benchmarking could also play a pivotal role in directing future research and development efforts.

We recognized the need for good benchmarking data for source code analyzers and resolved to create a benchmark. We chose to begin by constructing a prototype to test out and refine our ideas. Starting with a prototype allows us to understand what problems we know how to solve and gives us a platform to explore design decisions. An obvious goal of this work is to provide a foundation for the construction of a future benchmark.

*Under contract with Fortify Software.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SSATTM '05 Long Beach, California USA
Copyright 2005 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

We call our prototype the Analyzer BenchMark or ABM for short. The ABM benchmark is comprised of 91 micro-benchmark test cases and a single macro-benchmark test case. The purpose of each micro test case is to evaluate a tool against a very specific scenerio in a controlled way. The purpose of macro test cases is to capture properties of "real-world" programs, like size and complexity, that are absent from the micro test cases.

Our goal is to develop a framework that can be applied to any programming language or platform, but for the purposes of creating a prototype, all of the test cases target C-code source code analyzers running under Unix and Win32. We have applied the prototype to six different analyzers running in Redhat9 Linux and Windows XP. Our benchmark focuses on measuring the analysis strength of source code analyzers and does not concern itself with issues such as memory or time efficiency. Applying the ABM benchmark to source code analyzers results in quantifiable answers to the questions "What kind of vulnerabilities does this analyzer search for?" and "How effective is this analyzer at finding these vulnerabilities?".

Related work

We are not the first to attempt to measure the performance of source code analyzers. In his thesis [6] and accompanying article [7], Misha Zitser evaluates the performance of several source code analyzers for detecting buffer overflows. Zitser uses a test suite based on known vulnerabilities in real-world code. Zitser's test cases are derived from widely used applications but not comprised of the actual application code because many of the analyzers he measured were not capable of ingesting the large amount of code contained therein. Zitser's test suite is now available from the MIT Lincoln Laboratory in their publically available corpora [2].

Zitser describes the construction of "approximately 50 pairs" of small test cases. Each pair is made up of a small program with a single instance of a buffer overflow and a second matched program that is similar but does not contain the defect. To construct his test cases, Zitser first created a fairly detailed taxonomy of test case characteristics. He then hand-constructed the test pairs and classified them using his taxonomy. Although he describes the construction of his micro-benchmark test suite in detail, Zitser did not publish any results obtained from using the suite.

Zitser's micro-benchmark work was continued and extended by Kendra Kratkiewicz. In her thesis [1] Kratkiewicz extends the taxonomy created by Zitser and uses her taxonomy to guide the automatic generation of a micro-benchmark

suite of 591 quadruplets. Each quadruplet consists of four programs: one with no buffer overflow and three with successively larger buffer overflows. Kratkiewicz's use of quadruplets allows the effect of the magnitude of a buffer overflow to be analyzed. Her thesis describes the results of using this benchmark to compare the performance of several analyzers in detecting buffer overflows.

The SAMATE project at NIST [3] was created with the objective of "identification, enhancement and development of software assurance tools." [3] Part of the project's mandate is to support tool evaluation and, towards this end, they plan to create a system for benchmarking software analysis tools. This work is still in the planning stages and we are not aware of any results at this time.

Contributions

We believe our work makes several important contributions to source code analyzer benchmarking. First, we are constructing a standard benchmark. Prior efforts focused on constructing a benchmark for the author's use in measuring particular tools. We intend our benchmark to be used by others including end-users and research and development teams from academia and industry.

Second, we wish to benchmark a large number of analyzers. This goal forces us to pay attention to engineering issues such as ease of retargetting. Our benchmark is carefully automated with a build environment that has small, isolated analyzer-specific components. We use a normalization mechanism to reduce the amount of analyzer-specific code in our benchmarking process.

Third, we are interested in benchmarking all vulnerability types and do not limit our focus to a single class of vulnerabilities. This choice has resulted in a classification system that is more flexible than those used previously.

Fourth, we are interested in benchmarking across a wide set of platforms and languages.

Fifth, we see weakness in the grading mechanisms used by Zitser and Krakiewicz, which expect that each test case can have at most one reported vulnerability. Our benchmark uses a more sophisticated grading process that does not make such an assumption.

Finally, we see value in both micro-benchmarking and macro-benchmarking and use both to measure the performance of source code analyzers.

With the remainder of this paper we discuss the composition of the ABM prototype benchmark, why we made certain choices, and our plans for expanding from a prototype to a full benchmark proposal. Examples of benchmark results are provided to clarify discussion, but, due to space constraints, complete results are not given. We direct curious readers to http://vulncat.fortifysoftware.com for more results.

2. DESIGN GOALS

If a benchmark is going to gain wide adoption, it must meet a number of requirements:

First, the benchmark must be fair, objective and transparent. A benchmark that is not fair and objective will be rejected by the source code analysis community. Our benchmark should generate transparent results that can be independently reproduced, scrutinized and verified so that matters of fairness can be publicly decided.

Second, the benchmark must be able to accomodate change. The relevance of different vulnerabilities, platforms and languages will evolve over time. The widespread adoption of source code analysis tools is in its very infancy, virtually ensuring future change. In order to be successful our benchmark must be flexible and extensible. The framework should allow for the introduction of new languages, platforms and metrics.

The benchmark must also be applicable to a wide range of analysis techniques. Without good coverage of important platforms or languages, people will look elsewhere in order to find a becnhmark that is relevant to their needs and intrests. This means that, at a minimum, the benchmark must support the most popular languages and operating environments.

The benchmark must generate an easy-to-understand score. The majority of consumers of benchmark results will not be experts in source code analysis, and they will look to the benchmark for a simple way to compare competing tools. Some will use the scores to make important purchasing decisions. It is important that these scores be interpreted correctly and not be prone to marketing spin. In order to be relevant, the scores must also be based on measurements that are important in real-world programs.

Finally, the benchmark should generate a wealth of data about each tool measured. The need for this information is two-fold. First, the data will provide transparency by allowing the results of the benchmark to be scrutinized and independently verified. Second, the data will allow for detailed analysis of the strengths and weaknesses of each tool. Consumers will be able to use the data to focus on details that are most relevant to them. This data will also be useful in focusing future source code analysis research and development efforts.

Beyond these requirements we have some pragmatic goals for our benchmark. We would like the benchmark to be easy to use. We wish to create a benchmark that is easy to retarget for new source code analysis tools. To the greatest extent possible we want to automate the application of the benchmark, making it easy to carry out the benchmarking process. We believe that automation has the additional advantage of increasing the objectivity of a benchmark. We also want the benchmarking results to be easy to view, interpret and consume.

3. THE BENCHMARK

We chose to begin our project by creating a prototype. Our goal in starting with a prototype is to allow us to figure out what aspects of the design we actually understood and provide a platform to experiment with the aspects that we did not. For this purpose we chose to restrict the scope of our project significantly, focussing on static analyzers for Java and C code running under Linux and Windows XP. We decided that we should build a modestly sized suite of micro-benchmark test cases, with the understanding that the coverage of the suite would suffer for it. We also chose to begin with a single macro-benchmark test case. For maximum flexibility, all of our micro test cases would be constructed manually, and our macro test case would come from a widely adopted open source program.

The decision to use both micro- and macro-benchmarking was not an easy one. A macro-benchmark is made up of larger test cases drawn from source code in use in "real-world" applications. These test cases are large, compli-

Attribute	Parent	Keywords
Platform	General	Port Unix Win32
Program size	General	Size0...Size9
Program complexity	General	Complex0...Complex9
Vulnerability class	General	BufferOverflow Api Taint Race
Overflow location	BufferOverflow	Stack Heap
Overflow API	BufferOverflow	AdHoc AdHocDecode AdHocCopy Read Gets Strcpy Sprintf Memcpy
Overflow cause	BufferOverflow	Unbounded NoNul IntOverflow BadBound
API type	Api	MemMgmt Chroot
MemMgmt type	MemMgmt	DoubleFree Leak
Taint type	Taint	Unsafe InfoLeak FormatString
Race type	Race	Filename

Table 1: Measured attributes, their dependencies and the set of keywords used to describe them.

Keyword	Description
Port	Portable across all platforms.
Unix	Contains UNIX-specific code.
Win32	Contains Win32-specific code.
Size0...Size9	Description of the program size from small to large.
Complex0...Complex9	Description of the program complexity from simple to complex.
BufferOverflow	Contains a buffer overflow vulnerability.
Api	Contains a vulnerability caused by misusing an API.
Taint	Contains a vulnerability caused by misuse of tainted data.
Race	Contains a vulnerability cause by a race condition.
Stack	The overflow occurs in a buffer located on the stack.
Heap	The overflow occurs in a buffer located on the heap.
AdHoc	The overflow is caused by an ad hoc buffer manipulation.
AdHocDecode	The overflow is caused by an ad hoc buffer decode operation.
AdHocCopy	The overflow is caused by an ad hoc copy operation.
Read	The overflow is caused by use of the `read()` function.
Gets	The overflow is caused by use of the `gets()` function or a similar related function.
Strcpy	The overflow is caused by use of the `strcpy()` function or a similar related function.
Sprintf	The overflow is caused by use of the `sprintf()` function or a similar related function.
Memcpy	The overflow is caused by use of the `memcpy()` function or a similar related function.
Unbounded	The overflow was caused because no bounds check was made.
NoNull	The overflow was caused because a `NUL` character was expected but not found.
IntOverflow	The overflow was caused because of an integer overflow or underflow when computing bounds.
BadBound	The overflow was caused because an incorrect bounds check was performed.
MemMgmt	A memory management API was misued.
Chroot	The `chroot()` function was misued.
DoubleFree	Allocated memory was freed multiple times.
Leak	Allocated memory was never freed.
Unsafe	Tainted data was passed to an unsafe function.
InfoLeak	Sensitive data was revealed.
FormatString	Tainted data was used as a format string to a function in the `printf` family of functions.
Filename	A race was caused by accessing a file multiple times by its filename.

Table 2: Descriptions for each keyword.

cated, and provide several challenges to benchmark analysis. Macro-benchmark cases contain an entanglement of many factors that are not easily seperated for independent measure. A micro-benchmark is comprised of a set of small synthetic test cases in which each test case can be carefully designed to isolate characteristics. Tests can include control subjects to increase the realiability of any measurements made.

The precision of the micro-benchmark test cases comes at a cost: it is "real-world" applications that interest programmers, not synthetic test cases. A micro-benchmark may fail to capture some salient feature of important applications, such as size or complex interactions between features. Even when micro-benchmarks do provide useful results, their accuracy may be called into question unless they can be validated against data from important applications. For these reasons we chose to create a blend of micro-benchmarks and macro-benchmarks.

3.1 Test Case Attributes

To guide the creation of synthetic test cases we looked at a wide range of computer security vulnerabilities arising

BAD case	OK case	Keywords
ahscpy1-bad.c	ahscpy1-ok.c	Port Size0 Complex0 BufferOverflow Stack AdHocCopy Unbounded
chroot1-bad.c	chroot1-ok.c	Unix Size0 Complex0 Api Chroot
fmt1-bad.c	fmt1-ok.c	Port Size0 Complex0 Taint FormatString
fmt2-bad.c	fmt2-ok.c	Unix Size0 Complex0 Taint FormatString
fmt3-bad.c	fmt3-ok.c	Unix Size0 Complex1 Taint FormatString
	fmt4-ok.c	Port Size0 Complex0 Taint FormatString
	fmt5-ok.c	Port Size0 Complex0 Taint FormatString
into1-bad.c		Port Size0 Complex0 BufferOverflow Heap AdHoc IntOverflow
into2-bad.c	into2-ok.c	Port Size0 Complex0 BufferOverflow Heap AdHoc IntOverflow
mem1-bad.c	mem1-ok.c	Port Size0 Complex0 Api MemMgmt Leak
mem2-bad.c	mem2-ok.c	Port Size0 Complex1 Api MemMgmt Leak
race1-bad.c	race1-ok.c	Unix Size0 Complex0 Race Filename
race2-bad.c	race2-ok.c	Unix Size0 Complex0 Race Filename
snp1-bad.c	snp1-ok.c	Port Size0 Complex0 BufferOverflow Stack Sprintf BadBound
snp2-bad.c	snp2-ok.c	Port Size0 Complex0 BufferOverflow Stack Sprintf BadBound
tain1-bad.c	tain1-ok.c	Port Size0 Complex0 Taint Unsafe
tain2-bad.c	tain2-ok.c	Unix Size0 Complex0 Taint Unsafe

Table 3: A sampling of test cases and their attribute keywords.

from programming mistakes in C source code. We created a formal taxonomy of vulnerability attributes. While our taxonomy is not complete we believe it captures many of the kinds of program attributes that are important to source code analyzers. This taxonomy defines a problem space for our test cases that we were able to use for both choosing which test cases to create and to measure the coverage of the resulting test suite. The taxonomy also proved useful in automating the analysis of the results.

While creating our taxonomy, we observed that some program attributes are only relevant to some types of vulnerabilities. We support attributes in this irregular taxonomy by using a heirarchical system of keywords. For each attribute we enumerate a set of keywords that describe that attribute. Each attribute also has a parent keyword upon which it is dependent. When an attribute's parent keyword is present, exactly one of the keywords for that attribute must be specified. At the top of this dependency heirarchy is the keyword "General" which is always implicitly present. The attributes we measure are shown in Table 1 and the keywords used described in Table 2.

Each test case is described by a string of keywords for each relevant attribute. For example, the string "`Port Size0 Complex1 BufferOverflow Heap Gets Unbounded`" describes a small portable test case containing a buffer overflow on the heap using the `gets()` function, which does not perform any bounds checking.

This system of keywords is very flexible. Attributes that are specific to features of a particular language or operating system can be introduced without interfering with other unrelated attributes. Keywords and attributes can even be used to describe non-technical details about a test case such as the source of contributed material. Finally the system of keywords is well-defined (although some of the keywords currently in use are not). This simplifies the verification of well-formed test case descriptions.

3.2 Test Suite

We used our taxonomy to guide our creation of ABM micro-benchmark test cases. The intent is to have broad coverage of combinations of attributes in our taxonomy. We consciously chose to limit the number of test cases in our prototype benchmark allowing the benchmark coverage to suffer a little in order to focus on other details such as grading and analysis. We sketched out aproximately what types of test cases we wanted to include and constructed the test cases manually. Constructing a small number of test cases allowed us to explore some alternate test case designs during prototyping and allowed us to focus more quickly on other areas of the benchmark design such as grading and analysis.

The benchmark test suite is composed of 91 C test cases with a somewhat even coverage of attributes in our taxonomy. We are currently in the process of adding test cases for Java. Table 3 lists a sampling of 31 of the 91 test cases and their attributes. Whenever possible, test cases were constructed in matched pairs of `OK` and `BAD` test cases. Each `BAD` test case has code with a vulnerability in it. Corresponding `OK` cases are similar to `BAD` cases but do not share the vulnerability. `OK` cases are generally constructed by patching `BAD` cases to remove the vunlerability while retaining functionality. These `OK` cases share the same attribute keywords even though they do not contain a vulnerability.

Kratkiewicz [1] used test case quadruplets rather than pairs to measure the effects of buffer overflow magnitude. We did not take this approach because the magnitude attribute is specific to buffer overflow vulnerabilities. The effect of magnitude can still be measured by providing additional test case pairs that cover buffer overflows of varying mangitude.

Each test case contains annotations specifying where a bug may be present. Test cases with vulnerabilities are tagged with a comment "`/* BAD */`" at any line that may be considered to contribute to the vulnerability described by the test case's keywords. Those cases without vulnerabilities are tagged with a comment "`/* OK */`" at any line that could have contributed to the vulnerability if it had been present.

Each test case is also annotated with information about valid and invalid inputs. These annotations allow for automated testing of the resulting binaries. Some of the test cases we constructed were dependent on implementation details, such as compiler layout and padding, and were not

```
 1 /*
 2 Description: Printf is called with a user supplied
   format string.
 3 Keywords: Port Size0 Complex0 Taint FormatString
 4 ValidArg: "'NormalString\n'"
 5 InvalidArg: "%s"*100
 6 */
 7
 8 #include <stdio.h>
 9
10 void
11 test(char *str)
12 {
13     printf(str);                /* BAD */
14 }
15
16 int
17 main(int argc, char **argv)
18 {
19     char *userstr;
20
21     if(argc > 1) {
22         userstr = argv[1];
23         test(userstr);
24     }
25     return 0;
26 }
```

Figure 1: The test case `fmt1-bad.c`.

vulnerable when using our compiler of choice. We felt these test cases were still important and did not want to simply discard them. As a result, some of our test cases fail automatic validation. It is also important to note that an incorrect test case may still pass validation if the validation inputs are not picked properly. Thus, we were able to use automated testing as an aid in validating our test suite but could not rely on it completely.

An example of a test case in the suite is shown in Figure 1. Annotations in the comments at the head of the file give a formal and informal description of the test case. They also provide test strings for automated verification of the test case. An annotation at line thirteen denotes the occurance of a vulnerability.

For our macro-benchmark test case we chose the Apache web server [4]. Our choice was influenced by several factors. Apache is well known, accepted, and mature. It is freely available in source form and runs on a wide range of platforms. Typical deployments of Apache require that it be exposed to security threats from the entire internet. Apache provides us with a relatively large and complex program that is representative of the types of programs that people would want to analyze with a source code analyzer.

3.3 Grading

To benchmark a tool with the ABM suite, we run the tool being benchmarked against each test case in the suite and gather the results for grading. The entire process is automated with a system of makefiles. The tool is invoked once for each test case. The result of each analysis is normalized into a standard format, and a grading program compares the normalized tool output against the annotations in the

```
N: fmt1-bad
L: fmt1-bad.c 13 FormatString format printf If
  format strings can be influenced by an
  attacker, they can be exploited. Use a
  constant for the format specification.
```

Figure 3: The normalized results of Flawfinder's analysis of `fmt1-bad.c`.

test case. Finally, the graded results are combined into a benchmark result. This process is illustrated in Figure 2.

In order to benchmark a tool, the benchmark framework must be able to invoke the analyzer, and there must be a method for converting the results of the analysis into a standardized format. Most of the benchmark process is dictated by makefile rules shared by all tools. Tool-specific rules are contained in a separate makefile for each analyzer. Because some test cases are platform specific, each analyzer-specific makefile must specify which set of test cases to analyze. They must also specify how to invoke the analyzer and what file extension to use when saving the results. These tool-specific makefiles are typically less than 40 lines long.

We use a normalized output format to avoid putting analyzer-specific code in the grading process. The normalizer emits a line for each vulnerability reported by the analyzer. Each vulnerability is first mapped to the most specific matching keyword in our taxonomy. In some cases a vulnerability may be mapped to several keywords. The normalizer emits a single line for each combination of keyword, file name, and line number. To make it easier to verify that our normalizer is behaving as expected, we include the analyzer's original description of each vulnerability in the normalized output. The amount of code necessary to normalize an analyzer's output is highly dependent on the output format of an analyzer. So far we have constructed normalizers for six analyzers ranging in size from 59 to 137 lines of python code.

Figure 3 shows an exerpt of the normalized results from the Flawfinder [5] analyzer. The line starting with "`N:`" describes the test case source (`fmt1-bad`). The line starting with "`L:`" describes an instance of a `FormatString` vulnerability at line 13. The text following the `FormatString` keyword on this line is not used and is provided to aid manual review of the results. The remainder of the file contains results for other test cases.

We use an automated system to grade the normalized analysis results. Grading results manually would be tedious and error-prone and would place an artificially low bound on the practical size of our test suite. Manual grading would likely be less objective or at least less transparent than an automated process. Fortunately grading a test case's analysis results is a straightforward process of matching up the normalized results with the source code if the source code is properly annotated.

To grade the normalized results, each reported vulnerability is matched against the corresponding line in the annotated test case. Any reported vulnerability which does not appear as an attribute keyword for the test case is noted and ignored. Likewise, vulnerabilities matching attribute keywords but at lines that are not annotated as `BAD` or `OK` are noted and ignored. When a matching vulnerability occurs on a line with a `BAD` or `OK` annotation, it is noted as

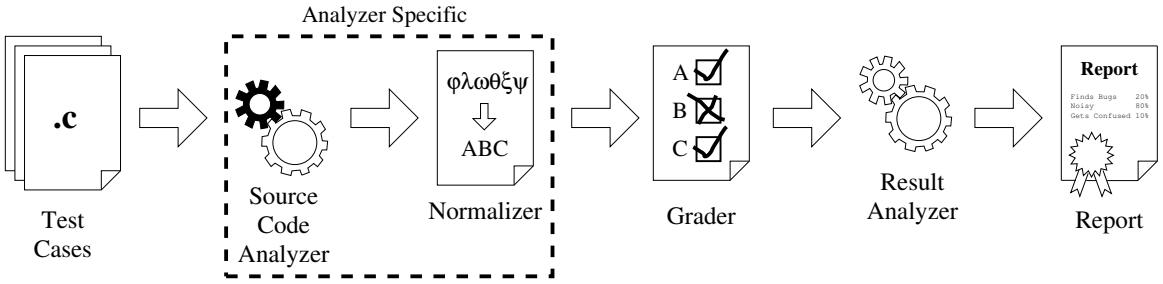

Figure 2: The ABM benchmarking process. Test cases are analyzed by the source code analyzer to be measured. The results are then normalized and graded before a report is generated. The invocation of the source code analyzer and the normalization of its results are the only analyzer-specific steps in this process.

```
fmt1-bad
Printf is called with a user supplied format string.
Port Size0 Complex0 Taint FormatString
13 BAD FormatString *MATCH* format printf If format
  strings can be influenced by an attacker, they can
  be exploited.  Use a constant for the format
  specification.

Raw results:
PASS fmt1-bad Port Size0 Complex0 Taint FormatString
```

Figure 4: The graded results of Flawfinder's analysis of `fmt1-bad.c`.

having matched. An analyzer is given a passing grade on a test case if it matches any of the `BAD` lines and does not match any of the `OK` lines. Upon completion, the grader emits a list of passing and failing test cases and their associated annotation keywords.

Notice that our grading process can ignore many reported vulnerabilities. Each test case is constructed to measure the analysis of a single type of vulnerability and the grader ignores any information about other reported vulnerabilities. Although it may appear that valuable information is discarded, this is not necessarily the case. Discarded information is not lost as long as there is another test case in the suite to measure the behavior. We believe that this approach of carefully focused measurment increases the reliability and leads to better analysis.

Figure 4 shows an exerpt of the graded results from the Flawfinder [5] analyzer. The group of lines at the top of the figure describe the grading process, starting with the test case name and the formal and informal descriptions of the test case. Following the description is a line representing the reported `FormatString` vulnerability at line 13. This line is indicated as a match since the test case is a `FormatString` test case. The Flawfinder tool passed this test case since there was a match for a "`BAD`" line. Had it failed, failure would be indicated in the output. These first lines are not used directly but are provided to ease human review of the grading process. The grader emits a table of graded results at the end of its output with a line for each graded test case. This is illustrated in the figure with a line that indicates that Flawfinder passed the `fmt1-bad` case.

Grading the macro-benchmark test cases is even simpler than grading micro-benchmark cases. As for the micro-benchmark test cases, the results from the macro-benchmark are first normalized. The benchmark cases were chosen in part for their maturity and it is assumed that there are relatively few vulnerabilities left in the code. For the purpose of grading, we assume that all reported vulnerabilities are false-positives. This assumption may introduce some error, but the approximation should be quite good and the error relatively small. Over time it is expected that a few bonafide vulnerabilities will be discovered in macro-benchmark test cases. We intend to maintain our current test cases and augment them with annotations as this occurs.

3.4 Analysis

The analysis phase makes use of the graded benchmark results to generate a report of meaningful measurements about an analyzer. The goal of the analysis is to measure the coverage and strength of an analyzer. An analyzer's coverage is a measure of the relevance of an analyzer to a variety of code defects. An analyzer with broad coverage is designed to detect a broad range of vulnerabilities whereas an analyzer with narrow coverage can only detect a small class of vulnerabilities. An analyzer's strength is a measure of the quality of analysis over diverse and sometimes difficult coding constructs. An analyzer with high strength can detect vulnerabilities in both simple and complicated code. Equally important, an analyzer with high strength is able to differentiate between vulnerable and non-vulnerable instances of similar code. An analyzer with low strength may not be able to detect a vulnerability in complex code or may incorrectly identify vulnerabilities where they do not exist.

In order to understand analysis of the results, it is necessary to first understand the meaning of passed and failed test cases. For "`BAD`" cases, a success indicates a "true positive" detection of a vulnerability while a failure indicates a "false negative" or that the analyzer incorrectly indicated that the vulnerability was not present. For "`OK`" cases, a success indicates a "true negative" or that the analyzer correctly indicated that no vulnerability was present, while a failure indicates a "false positive" detection of a vulnerability.

The simplest measure of an analyzer is given by a tally of the passing test cases. This measure imparts a rough measure of the analyzer but does not provide much insight into the analyzer's strengths or weaknesses. A slightly better measure is derived by partitioning the test cases into "`OK`" and "`BAD`" cases. This provides a measure of true positives and negatives, or, conversely, false positives and false

	BAD tests			OK tests			Total			Discriminates		
	pass	total	perc	pass	total	perc	pass	total	perc	pass	total	perc
All	26	48	54%	23	43	53%	49	91	54%	4	22	18%
`Unsafe`	2	2	100%	0	2	0%	2	4	50%	0	2	0%
`InfoLeak`	0	2	0%	0	0	- %	0	2	0%	0	0	- %
`FormatString`	3	3	100%	3	5	60%	6	8	75%	3	3	100%
`fmt1`	1	1	100%	1	1	100%	2	2	100%	1	1	100%
`fmt2`	1	1	100%	1	1	100%	2	2	100%	1	1	100%
`fmt3`	1	1	100%	1	1	100%	2	2	100%	1	1	100%
`fmt4`	0	0	- %	0	1	0%	0	1	0%	0	0	- %
`fmt5`	0	0	- %	0	1	0%	0	1	0%	0	0	- %

Table 4: Exerpts of benchmark results for the Flawfinder scanner.

negatives. The number of true positives gives some indication of how well the analyzer does the job advertised while the number of false positives gives a measure of how much additional noise it produces.

By partitioning the test cases according to vulnerability classes the coverage of an analyzer can easily be measured. An analyzer is said to cover a vulnerability class if it can report vulnerabilities in that class. If there are any true positives in the class (ie. there is at least one "`BAD`" test case that passed) then clearly the vulnerability class is covered.

Measuring an analyzer's strength is a little more complex. Some indication of an analyzer's strength is given by the number of false positives and false negatives that are reported. We can gain further insight into an analyzer's performance by partitioning a set of test cases based on a particular attribute. For example, by partitioning the set of `BufferOverflow` test cases according to program size, the effects of size on buffer overflow detection can be isolated.

One effect that is not easily isolated in this way is the ability of an analyzer to discriminate between vulnerable and non-vulnerable code. We introduce a new measure to quantify this component of an analyzer's strength. The *discrimination* of an analyzer is a tally of how often an analyzer passed an "`OK`" test case when it also passed a matching "`BAD`" test case. Together with the true negative and true positive tallies, discrimination gives a good indication of an analyzer's strength. An analyzer that finds many instances of a vulnerability but falsely reports the presence of this vulnerability where it is not present will score well when true-positives are measured but will not get a good true-negative or discrimination score.

Besides isolating the effect of defect variations, analyzing partitions based on keywords has an additional advantage – it allows us to make level comparisons of diverse analyzers. For example, the results of benchmarking an analyzer that runs only on Win32 platforms cannot directly be compared with the results from an analyzer that runs only in UNIX. However, by isolating the portable test cases (those described by the `Port` keyword) some amount of comparison can be made.

The ABM analyzer generates a report by generating tables of successively more detailed partitions of the data set. This process is straightforward because every test case is described by a sequence of keywords. Subsets of test cases are made by matching selected attribute keywords. These subsets are then scored for tallies of passed test cases, true positives, true negatives, and discrimination. Each tally is reported as an absolute count and as a percentage.

Table 4 shows an exerpt of the analyzed results from benchmarking the Flawfinder [5] source code analyzer. These results can be viewed in their entirety at http://vulncat.fortifysoftware.com. The line labelled "All" shows the accumulated results for all the micro-benchmark test cases. It shows that Flawfinder found 54% of the vulnerabilities, and properly did not report any vulnerabilities for 53% of the non-vulnerable test cases. When it was able to detect a vulnerability, it was able to discriminate it from non-vulenerable code 18% of the time. The next three lines show Flawfinder's performance for three classes of `Taint` vulnerabilities. Finally the last five lines show the individual test case pairs used to measure `FormatString` vulnerabilities. The `fmt1-bad.c` test case presented earlier is represented by the "BAD tests" column of the "`fmt1`" row.

Because of their nature, the macro-benchmark test cases are not as easy to analyze and do not provide as much information. The only analysis we perform is a counting of false-positives by attribute keyword. There is one subtlety in this process: we accumulate attribute counts up to their parent keywords. For example if there are five reported `FormatString` vulnerabilities and two reported `InfoLeak` vulnerabilities these counts are accumulated and reported as seven `Taint` vulnerabilities. The reason for this accumulation is to ease comparison of the results from different analyzers: some analyzers may report vulnerabilities deeper in the attribute taxonomy than other analyzers.

4. FUTURE WORK

We have built a prototype benchmark for source code analyzers, but our work is not yet done. A primary goal of this project has been to guide the development of our full benchmark. While what we have implemented is important, we consider what we have learned about what we must now implement an equally important contribution of our work.

We are currently in the process of adding Java test cases to the ABM suite. Although unexciting from a technical point of view these new test cases are critical to our goal of providing a standard cross-platform benchmark. Details about the Java test cases are available at http://vulncat.fortifysoftware.com.

The most glaring deficiency of the current benchmark is its coverage. This is partly due to our desire to keep the number of test cases manageably small in our prototype. A next-generation benchmark will require a micro-benchmark test suite one or two orders of magnitude larger. A hand-written test suite would clearly not be practical and we anticipate

generating test cases programatically as was done in [1].

A larger macro-benchmark test suite will also be needed. The process of incorporating more macro-benchmarks is tedious but fairly straightforward.

Test case generation will be guided by a classification system. Our initial taxonomy was successful but somewhat simplistic. Its classification of complexity and size lacks formal definition. The ABM test suite has particularly poor coverage of large or complex programs. A formal classification of size and complexity will give us a better foundation for addressing this deficiency. To ensure consistent and unbiased coverage of the taxonomy's attribute space we intend to formalize the process by which we pick test cases. The process of constructing matched `OK` cases also suffers from a lack of formal structure which we hope to address by augmenting the taxonomy with alternate patch strategies.

The taxonomies created and employed by Zitser [6] and Kratkiewicz [1] to describe programs with buffer overflows are considerably more detailed than ours. In the future we plan to incorporate attributes from their work into our taxonomy and expand our taxonomy to cover details particular to vulnerabilities other than buffer overflows.

The area of result analysis is ripe for future research. As with any benchmark, we anticipate that the availability of raw data will stimulate others to find new ways of extracting important information. There are two areas that we would like to pursue further in the future. Currently the result analysis places equal importance on each test case. This artificially weights the aggregated results according to the number of test cases in each category. We hope to address this by investigating weightings that more properly reflect the importance of test case properties in real-world situations. We hope that comparisons with macro-benchmark results will prove useful in this effort.

A second area of future interest is to provide better synergy between the micro- and macro-benchmarking components. As currently implemented our micro-benchmark and macro-benchmark cases are used to measure very different things. The relation between their results is not clearly apparent in the results. We hope that future analysis will allow the two suites to complement each other and corroborate each other's results.

A subtle issue that has been glossed over earlier in this paper is the handling of macro-benchmark results across disparate platforms. Even though the test case we chose compiles on a wide range of platforms, the source code used in the build process is not identical for all platforms. The build environment selects certain platform specific files appropriate for the platform. Conditional compilation selects certain segments of code within files that are used by all platforms. This makes comparisons of results obtained on different platforms troublesome. We are currently investigating stronger analysis techniques to remedy this.

Beyond the technical, there is a lot of work remaining in getting our benchmark adopted. We hope to work with the community to get feedback on our methodologies and address any early concerns. We plan to support the benchmark's adoption by promoting its fair use and the dissemination of results. We plan to continue benchmarking more analyzers, and we will make both the benchmark and results for a wide range of analzers available for download from http://vulncat.fortifysoftware.com.

5. REFERENCES

[1] K. Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Master's thesis, Harvard University, March 2005.

[2] MIT. Corpora. http://www.ll.mit.edu/IST/corpora.html, August 2005.

[3] NIST. Samate (nist software assurance metrics and tool evaluation). http://samate.nist.gov/, August 2005.

[4] The Apache Software Foundation. The apache http server project. http://httpd.apache.org/, August 2005.

[5] D. A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/, August 2005.

[6] M. Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, August 2003.

[7] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press.