Parfait – Designing a Scalable Bug Checker

Cristina Cifuentes
Sun Microsystems Laboratories
Brisbane, Australia
cristina.cifuentes@sun.com

Bernhard Scholz
Sun Microsystems Laboratories
Brisbane, Australia
and The University of Sydney
Sydney, Australia
scholz@it.usyd.edu.au

ABSTRACT

We present the design of Parfait, a static layered program analysis framework for bug checking, designed for scalability and precision by improving false positive rates and scale to millions of lines of code. The Parfait framework is inherently parallelizable and makes use of demand driven analyses.

In this paper we provide an example of several layers of analyses for buffer overflow, summarize our initial implementation for C, and provide preliminary results. Results are quantified in terms of correctly-reported, false positive and false negative rates against the NIST SAMATE synthetic benchmarks for C code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors

General Terms

static analysis

1. INTRODUCTION

This project started in response to an internal need within Sun Microsystems to be able to find bugs and security vulnerabilities in internal systems code. There are many tools and techniques available in the market and the literature, however, it is hard to find a tool to suit one's needs or to justify the acquisition of such a tool to management.

As a systems company, Sun develops systems software using languages like C and C++ that provide good performance but do not have type safety. For example, the Solaris™ Operating System (OS) is written in C, the Java HotSpot™ virtual machine is written in C++, and the Sun™ Studio compilers and support tools (e.g., dbx) are written in C and C++. The code bases range in sizes from 50 KLOC for library code to 6 MLOC for the core of the Solaris OS and the compilers.

In talking to these Sun organizations it became clear that a set of requirements had not been addressed for those teams to be using existing off-the-shelf bug checking tools for C/C++ code. Such requirements are:

- scalability: few tools can run over large (millions of lines of code) code bases in an efficient way. Several tools cannot parse the code or easily integrate with existing build environments, others can but may take too long (> 3 days) to run.
- rate of false positives: the tools that support millions of lines of code tend to report many bugs that are not bugs, leading to dissatisfaction and lack of use of the tool.
- rate of false negatives: tool documentation does not give an indication of the false negative rate for the tool. In small code bases, where bugs reported by the tool are fixed, this leads to a false sense of security, making developers think they have fixed all problems in the code.
- customization of the tool is missing in many instances: customization relates to various aspects; being able to check a subset of the code, being able to have a way to specify what parts of the system are more prone to security vulnerabilities, being able to obtain results of the bugs in a variety of levels of verbosity (from line number to a full trace of how the tool determined there was a bug at a particular line), and being able to specify priorities of bugs for a given organization.

These requirements compounded with the high cost of most off-the-shelf tools makes it hard for developers to make a case for licensing of the tool for a given project.

Based on these requirements, we have designed a new bug-checking framework for C/C++ code. The framework is designed for scalability (to be able to process millions of lines of code) and precision (to produce few false positives). We provide a demonstration of the framework in the form of a worked example for buffer overflow bugs, and show results from our initial implementation using our benchmarking infrastructure.

SAW'08, June 12, 2008, Tucson, Arizona, USA.

ACM 978-1-59593-924-1/08/0006

2. STATE OF THE ART

There are several approaches used to find bugs in programs. Dynamic analysis tools find bugs at runtime based on instrumentation of statements that can potentially cause bugs. The runtime check introduced by the instrumentation imposes a significant slowdown of factors between 2 to 350. Dynamic tools find bugs in paths that are exercised by the program, normally 10% of the paths of the program. To overcome this slowdown, hybrid tools employ static analysis to minimize the number of redundant checks at runtime, reducing the slowdown by a factor between 10 and 100. In contrast, static tools analyze the program without executing it, following all paths of the program, without incurring runtime overheads. In this paper we survey static bug checking tools because they provide better code coverage and feedback for auditors and developers before the program is deployed.

Bug checkers were introduced three decades ago. The `lint` [19] tool was developed in the late 1970's and aimed at detecting "bugs and obscurities" in C source programs, as well as enforcing the type rules of C more strictly than the C compilers did at the time. Tools like `lint` check for extra bugs that a compiler does not report when parsing programs. These tools are mainly based on syntactic analysis; limiting their analysis scope. `lint`'s main drawback is the large number of false positives reported by the tool: some users report up to 90% false positives, making it hard for developers to use the tool effectively for bug checking.

In the last decade, various new bug checking tools have been developed. These tools make use of advances in static analysis to reduce the number of false positives and utilize the increased processing power of modern computers. Despite these advances, many developers are hesitant to use bug checking tools. The reported number of false positives ranges from 50% down to 10-15%.

We classify bug checking tools based on the type of bugs that they can find, following the higher level Common Weakness Enumeration (CWE) [6] classes. These classes are also available in the Seven Pernicious Kingdoms taxonomy [29]. The list is by no means exhaustive but gives an idea of the work reported in the literature in the last decade.

The Timing and State category looks into bugs that are due to distributed computation via the sharing of state across time. Examples of bugs in this category are deadlocks and race conditions. Tools that support this category of bugs include: JPF [16], a Java™ programming language checker that model-checks annotated Java code; PREfix [5], a C/C++ checker based on inter-procedural data flow analysis; ESP [7], a C checker that focuses on scalability of analysis and simulation; and Goanna [13], a C/C++ checker that model-checks static properties of a program.

The Input Validation and Representation category looks into bugs that are caused by metacharacters, alternate encodings and numeric representations, and security problems resulting from trusting input. Examples of bugs in this category are buffer overflows, command injection, cross-site scripting, format string, integer overflow, SQL injection, etc. This category includes several of the bugs normally reported as security vulnerabilities by tool vendors. Tools that support both timing and state, and input validation and representation bugs include: ESC [8], a Modula-3 and Java checker that uses a theorem prover (Simplify) to reason about the semantics of language constructs, driven by annotations in the code; Coverity [10, 11], a C, C++ and Java checker based on "may belief" analysis; Jlint [1, 2], a checker of Java classfiles that is based on data flow and abstract interpretation; PREfast [26], a C, C++ checker based on intra-procedural analysis and statistics; Splint [12], a C lint prototype for security vulnerability analysis based on taint annotations; Archer [32], a C array checker that uses symbolic analysis; PolySpace [9], an Ada, C and C++ checker based on abstract interpretation that is used in the embedded systems market; FindBugs [17], a Java checker that uses bug-patterns and data flow analysis on Java classfiles; KlocWork [31], a C, C++ and Java checker based on inter-procedural data flow analysis; CQual++ [24], a C++ checker that makes use of taint annotations to determine security-related bugs; and GrammaTech's CodeSonar [15], a C, C++ checker that performs whole-program, interprocedural analysis.

The Security Features category is concerned with authentication, access control, confidentiality, cryptography and privilege management. Examples of bugs in this category are insecure randomness, least privilege violation, missing access control, password management and privacy violation. A tool that supports timing and state, input validation, and security features is Veracode [30], a binary/executable code checker based on data flow analysis that performs penetration testing on the binary code.

The API Abuse category is concerned with the violation of the (API) contract between a caller and a callee. Examples of bugs in this category are dangerous functions that cannot be used safely, directory restrictions, heap inspection, and various often misused language or operating system features. Tools that support timing and state, as well as API abuse bugs include: SLAM [3, 4], a C/C++ device driver checker that model-checks and verifies code against a specification of a device driver; and Blast [18], a C device driver checker that model-checks and verifies behavioral properties of the interfaces the code uses. A tools that support timing and state, input validation, security features and API abuse bugs is Fortify's Static Code Analysis [14], a checker that supports 12 different programming languages that finds over 200 different security issues.

3. PARFAIT FRAMEWORK DESIGN

For better precision in bug checking, our design is to employ an ensemble of static program analyses that range in complexity and expense. Analyses are ordered from least to more (time) expensive, ensuring that each buggy statement is detected with the cheapest possible program analysis capable of detecting it, effectively achieving better precision with smaller runtime overheads.

Figure 1 shows our framework, our algorithm works as follows: First, a worklist for a specific bug (e.g. buffer overflow) is set up and populated with all statements that potentially can cause the bug. Second, we iterate over the program analyses in the ensemble in ascending order. With the selected

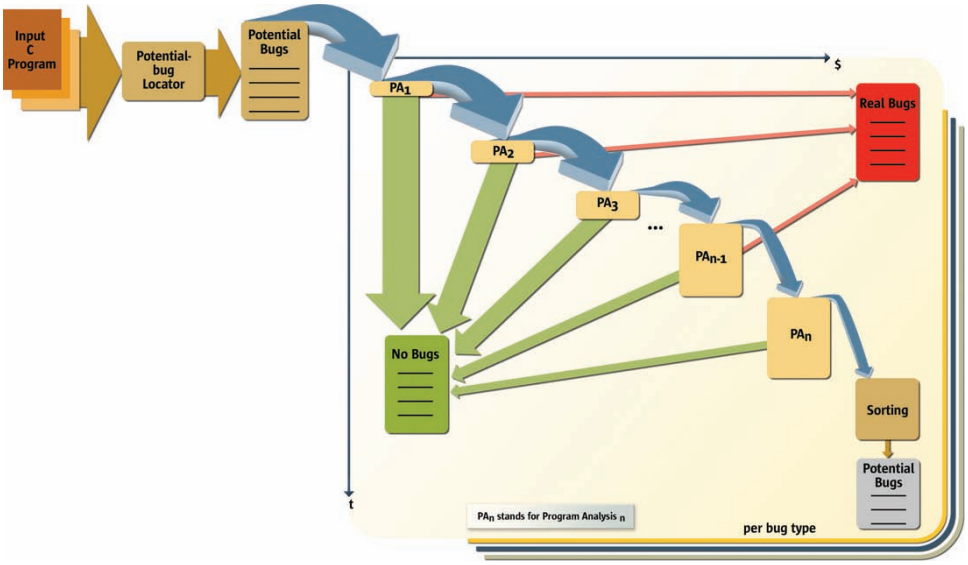

Figure 1: The Parfait Framework

program analysis we analyze the statements in the worklist. For each statement, the analysis will either confirm the presence of a real bug, reject the potential bug as a non-bug, or retain the potential bug in the worklist for further analysis by a later analysis. Third, the remaining statements in the worklist are reported as potential bugs and need to be verified by the auditor. Heuristics should be applied to order the importance of the reported bugs. Note that all analyses $PA_1, \ldots, PA_{n-1}$ need to be sound for this "waterfall" model to work.

To overcome the computational program analysis bottleneck, we parallelize the algorithm by employing demand driven program analysis instead of traditional forward analyses for the whole program. Demand driven analysis generates a backward slice of a program starting at a particular statement of interest (in our case, a potential buggy statement). The algorithm has two levels of embarrassingly parallel problems. The first level is the various worklists for specific bugs (e.g. buffer overflows, string vulnerabilities, integer overflows, etc.) and the second level is for statements in a specific worklist. Independent to us, Le and Soffa have also proposed the use of demand driven program analysis in their buffer overflow checking tool [22].

To summarize, the design of the framework features:

- scalability by reducing the problem space with fast analyses applied first,
- precision by use of sound analyses and refinements of the worklists, and
- parallelizability by employing demand driven analysis.

4. AN EXAMPLE

Figure 2 gives a simple example in C code that takes as arguments the length of some data and the data itself (as a string). The program initializes two buffers: the stack buffer `buf` is initialized to the "AAA...A" string with a trailing C end-of-string character, and the heap buffer `buf2` is initialized to the input data provided as the third parameter to the program (`argv[2]`), after allocating data from the heap of size equal to the second parameter (`argv[1]`). This program is an adaptation of Seacord's sample buggy programs [28]. This example has 5 bugs, 3 of which are buffer overflows. A buffer overflow occurs when data is copied to a location in memory that exceeds the size of the reserved destination area (i.e., a write is done outside the bounds of the array).

```
 0 #include <stdlib.h>
 1 #define BUFF_SIZE 100
 2
 3 int main (int argc, char *argv[])
 4 { char buf[BUFF_SIZE], *buf2;
 5    int n = BUFF_SIZE, i;
 6
 7    if (argc != 3){
 8       printf("Usage: prog_name length_of_data data\n");
 9       exit(-1);
10    }
11
12    for (i = 1; i <= n; i++) {
13       buf[i] = 'A';
14    }
15    buf[n] = '\0';
16
17    n = atoi(argv[1]);
18    buf2 = (char*)malloc(n);
19    for (i = 0; i <= n; i++) {
20      buf2[i] = argv[2][i];
21    }
22
23    return 0;
24 }
```

Figure 2: Sample C source code with 3 buffer overflow bugs

The bugs in the example program are:

- the index computed in line 12 is out of bounds for the last iteration of the loop, causing a stack buffer overflow on line 13,
- the terminating string character in line 15 is assigned to the wrong index location, causing a stack buffer overflow,
- the memory allocation in line 18 treats the signed integer value in `n` as an unsigned value of type `size_t`. For negative values of `n` this leads to a large, positive value being used in the memory allocation,
- the index computed in line 19 is out of bounds on the last iteration of the loop, causing an off-by-one heap buffer overflow on line 20, and
- the index computed in line 19 may be out of bounds for the read array access in line 20.

In this section we show sample layers of analyses of the framework that aid in detecting buffer overflow bugs. The program analyses and their ordering are: constant propagation and folding < partial evaluation < symbolic analysis using affine constraints.

4.1 Constant Propagation and Folding

Constant propagation and folding is one of the cheapest inner analyses that can be implemented. Constant propagation is the process of substituting the values of known constants in expressions. Constant folding is the process of simplifying constant expressions. These data flow analyses can be used to check whether constant array indices are out of bounds. In the example, variable `n` is initialized in line 5 with a constant value that can be propagated into its uses at lines 12 and 15, leading to the following snippet of code

```
 3 int main (int argc, char *argv[])
 4 { char buf[100], *buf2;
 ...
12    for (i = 1; i <= 100; i++) {
13       buf[i] = 'A';
14    }
15    buf[100] = '\0';
```

The constant array access on line 15 can be checked to be out of bounds as the array `buf` was defined in line 4 to have 100 elements indexed from 0 to 99. This analysis is partially complete as not all array indices become constant.

4.2 Partial Evaluation

Partial evaluation is a program transformation technique which operates via specialization: any function can be specialized by fixing one or more of its inputs to a particular value. For buffer overflow checking, partial evaluation can be used when analyzing a loop that accesses an array and has a constant number of iterations. A slice of the loop that contains the statements that are relevant for the array access is computed. This small slice of code can be augmented with a test for out of bounds access, then partial evaluation is performed on the augmented slice of code.

This analysis, a more expensive yet cheap layer, is relevant to the code in lines 12-14, where there is a write buffer access on line 13. The slice of this loop contains lines 12 and 14, and the execution of the augmented slice would return a true value on overflow and false otherwise, finding the second buffer overflow bug in the program. Pseudo-code for the augmented slice is as follows

```
12    for (i = 1; i <= 100; i++) {
         if (i < 0 || i > 99)
            return (true);
14    }
      return (false);
```

4.3 Symbolic Analysis using Affine Constraints

Symbolic analysis uses symbolic expressions to describe computations of a program as algebraic formulas over the program's input. In symbolic analysis we can use linear constraints to determine properties of an array. This program analysis is useful in determining whether non-constant indices of an array are out of bounds or not. In our example, we would like to check lines 19-20, as non-constant indices are used. The slice for the write array access at line 20 is the following

```
 3 int main (int argc, char *argv[])
17    n = atoi(argv[1]);
18    buf2 = (char*)malloc(n);
19    for (i = 0; i <= n; i++) {
20      buf2[i] = argv[2][i];
21    }
```

In this slice, index variable `i` needs to be within the range of 0 and `n-1`. However, variable i has the range of $0 \leq i \leq n$, violating the array bounds. A linear constraint solver detects the violation and finds this third buffer overflow in the program.

5. CURRENT IMPLEMENTATION

The initial implementation of Parfait is built on top of the LLVM framework [21], a low-level virtual machine for various languages including C and C++. Its instruction set has been designed for a virtual architecture that avoids machine specific constraints, and its instruction set is strictly typed. Every value or memory location has an associated type and all instructions obey strict type rules. LLVM code is represented in SSA form.

The code can have an unlimited number of typed virtual registers, which hold values of primitive types (integral, floating point, or pointer values). The instructions are encoded in three address code, i.e., most LLVM operations take one or two operands and produce a single result. For example, the instruction `%tmp.85 = add uint %indvar, 1` adds value 1 to virtual register `%indvar` and stores the result in `%tmp.85`.

Each variable has a single assignment such that the instruction that defines the variable and the variable's value can be synonymously used. At confluence points phi-nodes are

introduced to select a value from a set of values that are defined on different paths meeting at the confluence point.

LLVM provides type-safe pointer arithmetic with the instruction `getelementptr`, that calculates the address of a subelement in an aggregate data structure. For example

```
%tmp.16 = getelementptr sbyte** %argv, int 2
```

calculates the third element of the pointer array of `argv`. Instruction `getelementptr` either returns a pointer to a field or an element in an array. LLVM has a type conversion mechanism that is implemented via the cast instruction. The cast instruction converts a value from one type to another. The instruction `%tmp.4 = cast int %tmp.0 to uint` converts the value of `%tmp.0` that is an integer to an unsigned integer and stores the result in `%tmp.4`. LLVM provides support for alias analysis and has a couple of interprocedural, context-insensitive alias analyses in place (Andersen's and Steensgaard's), as well as a Data Structure Analysis context-sensitive analysis.

The current implementation of Parfait focuses on arrays reads and writes and is based on sound analyses in the absence of the standard C libraries:

- a potential-bug-locator that finds all locations in a C program where a buffer overflow (array write) or a read outside the bounds of the array (array read) is performed,
- a constant propagation analysis for buffer overflow and read outside the bounds of the array, and
- a partial evaluation analysis for buffer overflow and read outside the bounds of the array. The partial evaluation code is run on the LLVM interpreter using bytecodes

Parfait runs on the x64/Solaris OS, x86/Linux and x86/OSX platforms.

Security Vulnerabilities

If a developer is interested in concentrating on security vulnerabilities, an optional pre-processing analysis can be used before running the core engine (i.e., the framework), to determine which statements in the program are prone to security attacks. Using a simple definition of security vulnerability, where a bug is a security vulnerability if it can be exploited by an attacker using malicious input, we have developed a new user-input dependence analysis and have implemented both context-insensitive and context-sensitive approaches [27]. The algorithm keeps track of both, data and control dependencies, and is fast (linear in the number of statements and dependencies). A may-function alias analysis was added to LLVM to better support accuracy of the dependence analysis.

To illustrate, if analyzing the example in Figure 2 from a security vulnerability point of view, we need to determine which statements of the program are user-input dependent and only consider those for bug checking purposes. In the example there are two user inputs: the length of the data (`argv[1]`) and the string of data (`argv[2]`). The statements that are user-input dependent yield the following program fragment:

```
 0 #include <stdlib.h>
 1 #define BUFF_SIZE 100
 2
 3 int main (int argc, char *argv[])
 4 { char *buf2;
 5    int n = BUFF_SIZE, i;

17    n = atoi(argv[1]);
18    buf2 = (char*)malloc(n);
19    for (i = 0; i <= n; i++) {
20      buf2[i] = argv[2][i];
21    }
22
23    return 0;
24 }
```

This resulting program can then be fed into the core engine. For the analyses explained in Section 4, performing constant propagation in this case is of no use, neither is the partial evaluation of code, because constant values cannot stem from user-input dependencies. The only analysis that is of relevance in this case is the symbolic analysis, which will determine that there is a heap buffer overflow on line 20 for one index of the loop. This buffer overflow is the only security vulnerability in the program due to write array accesses.

For security analysis all APIs used in the programs need to be documented, including functions in the C library. For example, the `atoi` library function has the following signature:

```
int atoi (const char *nptr);
```

The output of the `atoi` function is dependent on the input to it. Hence, if the input is tainted data, the output is a tainted value. This type of information can be automatically generated by analyzing the libraries standalone. In the example, lines 18 and 19 become tainted in our analysis because the variable `n` becomes tainted. Line 20 is also tainted because the predicate of the loop contains tainted data (i.e., it is control dependent on tainted data).

6. PRELIMINARY RESULTS

We report on preliminary results using our initial implementation of two of the layers of analyses for buffer overflow. We use the NIST SAMATE benchmarks to report accuracy of the framework.

Evaluation Methodology

As far as we are aware of, there is no established evaluation methodology for bug checking tools. In the research community, a few first attempts have been made to consolidate this problem [33, 20, 23, 25], however, the reported literature does not seem to use these benchmarking results. Quite often results such as *"Tool/Technique X found Y number of bugs"* are presented in the literature. This statement lacks of precision because the following questions are not answered:

- Which kind of bugs were found? (there are many bugs in any large software project, but which ones are considered relevant to the developer or auditor?)
- How many bugs were not found? (i.e., false negative rate). Note that Y could be a small number in comparison with the real total number of bugs in the code.
- How many bugs were reported as bugs and were not actual bugs? (i.e., false positive rate)
- Does the tool understand when a reported bug has been fixed? (i.e., when the tool is re-run with a fixed bug, is the bug reported again or not?)
- How long did it take to run the tool to find the bugs?

We use a simple scheme that gives a "bug specific" view. For a specific bug class (e.g. buffer overflow bugs, signed/unsigned bugs), we conduct an automated evaluation. The evaluation contains the measurement of functional metrics for each benchmark including false positives and false negatives.

Besides functional metrics we are also interested in the execution time of the tool, i.e., how many lines of code it can process per minute, and how much memory is needed for the execution of the tool. Speed and memory consumption are important to be able to extrapolate the scalability of the system.

We are using the synthetic benchmarks that have been contributed to the SAMATE project [25], and we report on those benchmarks herein. As a side effect of our testing process we are also collecting sample proprietary and open source buggy source code and annotating it with the bugs that are known to exist in that code.

Results

Table 1 shows the results of the Parfait framework on a subset of the SAMATE benchmarks, namely, those that are for C code and that relate to buffer overflow and read outside the bounds of an array. Parfait currently implements two of the three analyses mentioned in this paper: constant propagation and partial evaluation. As such, Parfait can analyze statically-allocated buffers but not dynamically-allocated ones. The dataset had 1182 benchmarks with 896 bugs. Some benchmarks do not contain any bugs as they represent the fixed version of a buggy benchmark. The results show that 85% of the array bugs are correctly identified with a 0% false positive rate and a 6.5% false negative rate. The false negatives are due to the lack of support of the standard C libraries at this point in time; the semantic meaning of library functions like `strcpy` needs to be taken into account. In `strcpy`, the destination buffer is overwritten by the source buffer, and if the destination buffer is smaller than the source buffer, then a buffer overflow will happen, e.g., the SAMATE program basic-00046-med.c:

```
1    char src[18];
2    char buf[10];
3
4    memset(src, 'A', 18);
5    src[18 - 1] = '\0';
6    strcpy(buf, src);
```

Parfait will currently not report an error at line 6, leading to a false negative, because the error was missed.

Of the remaining 150 potential bug locations in the potential-bug list, 76 of them are real bugs yet to be determined (using symbolic analysis). The 896 bugs can be accounted for: 762 were correctly reported, 58 are not reported due to false negatives, and 76 are yet-to-be-reported (i.e., are in the potential-bug list).

7. CONCLUSIONS

In this paper we described the design of Parfait, a static layered program analysis framework for bug checking designed for scalability and precision.

Scalability is addressed by an ensemble of program analyses such that easy-to-check bugs are detected with cheap and simple analyses, and more complex bugs are detected with more expensive analyses. Scalability is also addressed by use of demand driven analyses, analyzing small slices of code around a potential buggy statement. Parallelization is easily enabled by the framework. Precision is addressed by using sound analyses in the ensemble of analyses, and keeping track of three different lists per bug-type being checked: real bugs, no bugs, and potential bugs.

We also reported on our initial implementation of buffer overflow detection and provided preliminary results using benchmarks from the NIST SAMATE project.

Acknowledgments

We would like to thank Nathan Keynes and Erica Mealy for comments to improve the presentation of this paper.

8. REFERENCES

[1] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 68–75, Canberra, ACT, Australia, August 2001. IEEE Press.

[2] C. Artho and K. Havelund. Applying Jlint to space exploration software. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937/2003 of *Lecture Notes in Computer Science*, pages 297–308. Springer Berling / Heidelberg, 2004.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Workshop on Model Checking of Software (SPIN)*, LNCS 2057, pages 103–122, May 2001.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the Principles of Programming Languages (POPL)*, pages 1–3. ACM Press, January 2002.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.

[6] CWE list (draft 5). `http://cwe.mitre.org/data/index.html`, December 2006.

[7] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In

Type of Data	Raw Data	Percentage
Number of benchmarks	1182	
Number of buffer overflows	893	
Number of read outside array bounds	3	
Number of reported buffer overflows	759	85%
Number of correctly-reported buffer overflows	759	100%
Number of false positives	0	0%
Number of false negatives	58	6.5%
Number of potential bugs in potential-bug list	150	
Number of buffer overflows in potential-bug list	76	8.5%
Number of reported read outside array bounds	3	100%
Number of correctly-reported read outside array bounds	3	100%
Number of false positives	0	0%
Number of false negatives	0	0%
Average time taken per benchmark	0.2068 sec	

Table 1: Current Results on Array Reads and Writes Using Parfait on the SAMATE C Benchmarks for Array Reads and Writes

Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pages 57–68, Berlin, Germany, June 2002. ACM Press.

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report SRC-RR-159, COMPAQ Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1998.

[9] A. Deutsch. Static verification of dynamic properties. PolySpace White Paper, February 2004.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 23–25, San Diego, CA, Oct. 2000. USENIX.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72, Alberta, Canada, Oct. 2001. ACM Press.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, January/February 2002.

[13] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna—a static model checker. In L. Brim, B. Haverkort, M. Leucker, and J. Pol, editors, *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems*, number 4346 in Lecture Notes in Computer Science, Bonn, Germany, Aug. 2006.

[14] Fortify Static Code Analysis (SCA). `http://www.fortify.com/products/sca/`. Last accessed: 1 April 2008.

[15] GrammaTech CodeSonar. `http://www.grammatech.com/products/codesonar/overview.html`. Last accessed: 1 April 2008.

[16] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), April 1999.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 92–106, Vancouver, BC, Canada, Oct. 2004. ACM Press.

[18] R. Jhala. *Lazy Abstraction*. PhD thesis, University of California, Berkeley, Fall 2004.

[19] S. Johnson. Lint, a C program checker. Unix Programmer's Manual, AT&T Bell Laboratories, 1978.

[20] K. Kratkiewicz and R. Lippmann. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[22] W. Le and M. L. Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 63–68, San Diego, CA, June 2007.

[23] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: A benchmark for evaluating bug detection tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[24] S. McPeak. Elsa/Oink/Cqual+. Talk at CodeCon, February 2006.

[25] NIST SAMATE – software assurance metrics and tool evaluation. `http://samate.nist.gov`. Last accessed: January 2007.

[26] J. Pincus. Steering the pyramids: Tools, technology, and process in engineering at microsoft. Keynote at the International Conference on Software Maintenance (ICSM). Slides at `http://research.microsoft.com/users/jpincus/icsm.ppt`, 2002.

[27] B. Scholz, C. Zhang, and C. Cifuentes. User-input

dependence analysis via graph reachability. Technical Report SMLI TR-2008-117, Sun Microsystems Laboratories, 16 Network Circle, Menlo Park, CA 94025, March 2008.

[28] R. C. Seacord. *Secure Coding in C and C++*. SEI Series, A CERT Book. Addison-Wesley, Sept. 2005.

[29] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, November/December 2005.

[30] Veracode website. `http://www.veracode.com/`. Last accessed: January 2007.

[31] M. Webster. Leveraging static analysis for a multidimensional view of software quality and security: Klocwork's solution. White paper, IDC, Framingham, MA, Sept. 2005.

[32] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336, New York, NY, USA, 2003. ACM Press.

[33] 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.