Workshop on the Evaluation of Software Defect Detection Tools

Sunday, June 12th, 2005
Co-located with PLDI 2005

Workshop co-chairs: William Pugh (University of Maryland) and Jim Larus (Microsoft Research)

Program chair: Dawson Engler (Stanford University)
Program Committee: Andy Chou, Manuvir Das, Michael Ernst, Cormac Flanagan, Dan Grossman, Jonathan Pincus, Andreas Zeller

Time **What**

8:30 am Discussion on Soundness

- *The Soundness of Bugs is What Matters*, Patrice Godefroid, Bell Laboratories, Lucent Technologies
- *Soundness and its Role in Bug Detection Systems*, Yichen Xie, Mayur Naik, Brian Hackett, Alex Aiken, Stanford University

9:15 am break

9:30 am Research presentations

- *Locating Matching Method Calls by Mining Revision History Data*, Benjamin Livshits, Thomas Zimmermann, Stanford University
- *Evaluating a Lightweight Defect Localization Tool*, Valentin Dallmeier, Christian Lindig, Andreas Zeller, Saarland University

10:30 am break

10:45 am Invited talk on Deployment and Adoption, Manuvir Das, Microsoft

11:15 am Discussion of Deployment and Adoption

- *The Open Source Proving Grounds*, Ben Liblit, University of Wisconsin-Madison
- *Issues in deploying SW defect detection tools*, David Cok, Eastman Kodak R&D
- *False Positives Over Time: A Problem in Deploying Static Analysis Tools*, Andy Chou, Coverity

12 noon lunch

1:00 pm Research presentations

- *Model Checking x86 Executables with CodeSurfer/x86 and WPDS++*, Gogul Balakrishnan, Thomas Reps, Nick Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Yong, Chi-Hua Chen, Tim Teitelbaum, Univ. of Wisconsin
- *Empowering Software Debugging Through Architectural Support for Program Rollback*, Radu Teodorescu, Josep Torrellas, UIUC Computer Science
- *EXPLODE: A Lightweight*, General Approach to Finding Serious Errors in Storage Systems, Junfeng Yang, Paul Twohey, Ben Pfaff, Can Sar, Dawson Engler, Stanford University

Time **What**

2:30 pm break

2:45 pm Research presentations

- *Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications*, Kostis Sagonas, Uppsala University
- *Soundness by Static Analysis and False-alarm Removal by Statistical Analysis: Our Airac Experience*, Yungbum Jung, Jaehwang Kim, Jaeho Sin, Kwangkeun Yi, Seoul National University

3:45 pm break

4:00 pm Discussion of Benchmarking

- *Dynamic Buffer Overflow Detection*, Michael Zhivich, Tim Leek, Richard Lippmann, MIT Lincoln Laboratory
- *Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools*, Kendra Kratkiewicz, Richard Lippmann, MIT Lincoln Laboratory
- *BugBench: A Benchmark for Evaluating Bug Detection Tools*, Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, Yuanyuan Zhou, UIUC
- *Benchmarking Bug Detection Tools. Roger Thornton*, Fortify Software
- *A Call for a Public Bug and Tool Registry*, Jeffrey Foster, Univ. of Maryland
- *Bug Specimens are Important*, Jaime Spacco, David Hovemeyer, William Pugh, University Maryland
- *NIST Software Assurance Metrics and Tool Evaluation (SAMATE) Project*, Michael Kass, NIST

5:00 pm Discussion of New Ideas

- *Deploying Architectural Support for Software Defect Detection in Future Processors*, Yuanyuan Zhou, Josep Torrellas, UIUC
- *Using Historical Information to Improve Bug Finding Techniques*, Chadd Williams, Jeffrey Hollingsworth, Univ. of Maryland
- *Locating defects is uncertain*, Andreas Zeller, Saarland University
- *Is a Bug Avoidable and How Could It Be Found?*, Dan Grossman, Univ. of Washington

5:45 pm wrap up and discussion of future workshops

6:00 pm done

The Soundness of Bugs is What Matters (Position Statement)

Patrice Godefroid
Bell Laboratories, Lucent Technologies

There is one thing stronger than all the armies in the world; and that is an idea whose time has come. – Victor Hugo

In this short note[1], I argue that most program analysis and verification research seems confused about the ultimate goal of software defect detection.

The Goal is to Find Bugs. The main practical usefulness of software defect detection is the ability to *find bugs*, not to report that "no bugs have been found". Unfortunately, the latter is sometimes confused for a *correctness proof*. In practice, there is no such thing as a *complete* correctness proof, since even a *sound analysis* implemented flawlessly in a bug-free tool is bound to check only a specific set of properties.

So, Why May-Analysis? Yet, most defect detection tools are surprisingly based on program verification ideas and make use of *conservative abstractions*. By design, such tools detect bugs that *may* happen. The price to pay for this questionable design decision is enormous: such tools are *doomed* to report (many) false alarms, i.e., *unsound bugs*. Despite progress on limiting false alarms (e.g., by using more precise symbolic execution or alarm classification techniques), any *may* program analysis is bound to generate false alarms and hence to require (significant) human effort.

The Importance of Testing. This may explain why *testing* has been a multi-billion dollar industry for many years, while automated code-inspection is merely emerging as a multi-million dollar business. Today, several orders of magnitude more effort (people, money, time) is spent on testing than on code inspection (manual and automated). The reason is simple: *testing finds sound bugs* while *may-analysis does not*.

A Paradox. *If defect detection is the goal, why are so many defect detection tools based on may-analysis?*

Sources of Imprecision. It is important to distinguish *two distinct* sources of imprecision in program analysis: (a) since we want to analyze *open* programs, we need realistic *environment assumptions*; (b) using abstraction implies approximate reasoning. While (a) is a hard problem (i.e., often requires user assistance), (b) is simply an *engineering issue* (see below).

Alternatives. After realizing that *"The Soundness of Bugs is What Matters"*, alternatives emerge. Here are three concrete examples, drawn from my own work:
1. VeriSoft is a software model checker for languages like C and C++ which uses a run-time scheduler for systematically driving the executions of a software implementation through its state space (no abstraction is used). Since this search is typically incomplete, VeriSoft sacrifices, by design, soundness of correctness proofs ("soundness") for soundness of bugs.
2. Must abstractions are abstractions geared towards finding errors, which dualize may abstractions geared towards proving correctness. With combined may/must abstractions, *both correctness proofs and bugs found* are guaranteed to be sound.
3. DART (Directed Automated Random Testing) is a new approach and tool (see PLDI'2005) for *automatically* testing software (i.e., no test driver needed) that combines (1) *automated* interface extraction from source code, (2) *random* testing at that interface, and (3) dynamic test generation to *direct* executions along alternative program paths. Although DART uses imprecise abstraction techniques, all bugs found by DART are guaranteed to be sound, by design.

Role of "Soundness". In my opinion, "soundness" (to find *all* potential bugs of a particular type) is important not for exhaustiveness reasons but only for efficiency reasons: the ability to prove the absence of bugs can be used to stop a dual search for sound bugs.

Conclusion. The *May* and the *Must* are the Yin and the Yan of program analysis. Yet, past research in program analysis and verification has tilted the balance towards the *May* and hence prevented a wider adoption of program analysis tools. I suggest to repair this imbalance by restoring the proper role of *Must*, i.e., to put *the soundness of bugs first*.

[1] Written in a provocative style for entertainment purposes – please take this note with the grain of salt it deserves.

Soundness and its Role in Bug Detection Systems

Yichen Xie Mayur Naik Brian Hackett Alex Aiken

Computer Science Department
Stanford University
Stanford, CA 94305

The term *soundness* originated in mathematical logic: a deductive system is sound with respect to a semantics if it only proves valid arguments. This concept naturally extends to the context of optimizing compilers, where static analysis techniques were first employed. There, soundness means the preservation of program semantics, which is the principal requirement of a correct compiler.

In bug detection systems, soundness means the ability to detect *all* possible errors of a certain class. Soundness is a primary focus of many proposals for bug detection tools. Tools that do not offer such guarantees are sometimes summarily dismissed as being, well, unsound, without regard to the tool's effectiveness.

However, soundness has costs. Beyond the simplest properties, analysis problems are often statically undecidable and must be approximated conservatively. These approximations may be expensive to compute or so coarse that a substantial burden is imposed on the user in analyzing spurious warnings (so-called *false positives*) from the tool.

In our view, successful approaches to bug detection (and program analysis in general) balance three desirable, but often competing, costs: soundness, or rather, the cost of false negatives that result from being unsound; computational cost; and usability, as measured in the total time investment imposed on a user. The question "*Is soundness good or bad?*" does not make sense by itself. Rather, it can be discussed only in the context of particular applications where these costs can be estimated.

This simple, "three cost" model allows us to make a few observations and predictions about the future of bug detection tools.

First, we can expect in almost every application area that widely used unsound tools will precede sound ones, because imposing soundness as a requirement constrains the design space sufficiently that it is simply more difficult and time consuming to find design points that give acceptable usability and computational cost. A good historical example is static type systems, which can be regarded as the canonical example of an analysis where soundness is very desirable. The most widely used languages of the 1980's (C and C++) deliberately had unsound type systems because of perceived usability and performance problems with completely sound type systems. The first widely used language with a static type system intended to be sound (Java) did not appear until the mid-1990's. As another example, the first successful checkers for concurrency problems used dynamic analysis techniques. Because they are dependent on test case coverage, dynamic analyses are unsound almost by definition. Today there is still no completely satisfactory static checker for concurrency errors for any mainstream programming language. We also note that sound systems are sometimes used in an unsound way in practice by, for example, turning off global alias analysis to reduce false positives.

Second, while sound systems will be slower to appear, they will appear. One can expect widely used sound analyses in two different classes of applications. In situations where the extra cost of soundness is minimal there will be no reason not to be sound. In applications where the cost of a single missed bug is potentially catastrophic, users will be more willing to sacrifice usability and performance for soundness. Areas such as security critical, safety critical, and mission critical applications are all likely targets for sound analyses. In general, however, unless a sound analysis can approach the performance and usability of unsound bug finding tools, the sound analysis will be used mainly in applications where the potential cost of unsoundness is highest.

Third, we expect most sound systems will assume at least some user annotations. As mentioned above, most of the analysis problems of practical interest are undecidable, and it is usually impossible to compute both a sound and reasonably precise (i.e., nearly complete) analysis for such problems. However, it is a striking property of many analysis applications that a small amount of additional information dramatically lowers the cost from undecidable to, say, linear time. This phenomenon has been studied extensively in the area of type theory, where the enormous difference in many type systems between the computational complexity of type inference (i.e., analysis with no annotations) and of type checking (i.e., analysis with user annotations) is well known. A little extra information can make many very hard analysis problems quite straightforward. Despite the understandable reluctance to impose any extra work, no matter how small, on users, we expect in most cases adding annotations will be the best path to a practical sound system.

Locating Matching Method Calls by Mining Revision History Data

Benjamin Livshits
Computer Science Department
Stanford University
Stanford, USA
livshits@cs.stanford.edu

Thomas Zimmermann
Computer Science Department
Saarland University
Saarbrücken, Germany
zimmerth@cs.uni − sb.de

Abstract

Developing an appropriate fix for a software bug often requires a detailed examination of the code as well as generation of appropriate test cases. However, certain categories of bugs are usually easy to fix. In this paper we focus on bugs that can be corrected with a *one-line code change*. As it turns out, one-line source code changes very often represent bug fixes. Moreover, a significant fraction of previously known bug categories can be addressed with one-line fixes. Careless use of file manipulation routines, failing to call `free` to deallocate a data structure, failing to use `strncpy` instead of `strcpy` for safer string manipulation, and using tainted character arrays as the format argument of `fprintf` calls are all well-known types of bugs that can typically be corrected with a one-line change of the program source.

This paper proposes an analysis of software revision histories to find highly correlated pairs of method calls that naturally form application-specific useful coding patterns. Potential patterns discovered through revision history mining are passed to a runtime analysis tool that looks for pattern violations. We focus our pattern discovery efforts on *matching method pairs*. Matching pairs such as ⟨`fopen`, `fclose`⟩, ⟨`malloc`, `free`⟩, as well as ⟨`lock`, `unlock`⟩-function calls require exact matching: failing to call the second function in the pair or calling one of the two functions twice in a row is an error. We use *common bug fixes* as a heuristic that allows us to focus on patterns that caused bugs in the past. The user is presented with a choice of patterns to validate at runtime. Dynamically obtained information about which patterns were violated and which ones held at runtime is presented to the user. This combination of revision history mining and dynamic analysis techniques proves effective for both discovering new application-specific patterns *and* for finding errors when applied to very large programs with many man-years of development and debugging effort behind them.

To validate our approach, we analyzed Eclipse, a widely-used, mature Java application consisting of more than 2,900,000 lines of code. By mining revision histories, we have discovered a total of 32 previously unknown highly application-specific matching method pairs. Out of these, 10 were dynamically confirmed as valid patterns and a total of 107 previously unknown bugs were found as a result of pattern violations.

The first author was supported in part by the National Science Foundation under Grant No. 0326227. The second author was supported by the Graduiertenkolleg “Leistungsgarantien für Rechnersysteme”.

1 Introduction

Much attention has lately been given to addressing application-specific software bugs such as errors in operating system drivers [1, 4], security errors [5, 8], and errors in reliability-critical embedded software in domains such as avionics [2, 3]. These represent critical errors in widely used software and tend to get fixed relatively quickly when found. A variety of static and dynamic analysis tools have been developed to address these high-profile bugs. However, many other errors are specific to individual applications or sets of APIs.

Repeated violations of these application-specific coding rules, referred to as *error patterns*, are responsible for a multitude of errors. Error patterns tend to be re-introduced into the code over and over by multiple developers working on the project and are a common source of software defects. While each pattern may be only responsible for a few bugs, taken together, the detrimental effect of these error patterns is quite serious and they can hardly be ignored in the long term.

In this paper we propose an automatic way to extract likely error patterns by mining software revision histories. Moreover, to ensure that all the errors we find are relatively easy to confirm and fix, we limit our experiments to errors that can be corrected with a one-line change. When reporting a bug to the development team, having a bug that is easy to fix usually improves the chances that it will actually be corrected. It is worth noticing that many well-known error patterns such as memory leaks, double-`free`'s, mismatched locks, operations on operating system resources, buffer overruns, and format string errors can often be addressed with a one-line fix. Our approach uses revision history information to infer likely error patterns. We then experimentally evaluate the error patterns we extract by checking for their violations dynamically.

We have performed our experiments on Eclipse, a very large, widely-used open-source Java application with many man-years of software development behind it. By mining CVS revision histories of Eclipse, we have identified 32 high-probability patterns in Eclipse APIs, all of which were previously unknown to us. While the user of our system has a choice as to which patterns to check at runtime, in our experiments we limited ourselves to patterns that were frequently encountered in our revision history data. Out of these, 10 were dynamically confirmed as valid patterns and 107 bugs were found as a result of pattern violations.

1.1 Contributions

This paper makes the following contributions:

- We propose a *data mining strategy* that detects common matching method pairs patterns in large software systems by analyzing software revision histories. We also propose an effective ranking technique for the patterns we discover that uses one-line changes to favor previously fixed bugs.
- We propose a *dynamic analysis approach* for validating usage patterns and finding their violations. We currently utilize an off-line approach that simplifies the matching machinery. Error checkers are implemented as simple pattern-matchers on the resulting dynamic traces.
- We present an *experimental study* of our techniques as applied to finding errors in Eclipse, a large, mature Java application with many years of development behind it. We identified 32 patterns in Eclipse sources and out of these, 10 patterns were experimentally confirmed as valid. We found more that 107 bugs representing violations of these pattern with our dynamic analysis approach.

2 Revision History Mining

To motivate our approach to revision history mining, we start by presenting a series of observations.

Observation 2.1 *Given multiple software components that use the same set of APIs, it should be possible to find* common errors *specific to that API.*

In fact, much of the research done on bug detection so far can be thought of as focusing on specific classes of bugs pertaining to particular APIs: studies of operating-system bugs provide synthesized lists of API violations specific to operating system drivers resulting in rules such as "do not call the interrupt disabling function `cli()` twice in a row" [4].

Observation 2.2 *Method calls that are frequently added to source code simultaneously often represent a usage pattern.*

In order to locate common errors, we mine for frequent usage patterns in revision histories. Looking at incremental changes between revisions as opposed to full snapshots of the sources allows us to better focus our mining strategy. However, it is important to notice that not every pattern *mined* by considering revision histories is an actual *usage* pattern. Figure 1 lists sample method calls that were added to revisions of files `Foo.java`, `Bar.java`, `Baz.java`, and `Qux.java`. All these files contain a usage pattern that says that methods {`addListener`, `removeListener`} must be precisely matched. However, mining these revisions yields additional patterns like {`addListener`, `println`} and {`addListener`, `iterator`} that are definitely *not* usage patterns.

File	Revision	Added method calls
`Foo.java`	1.12	`o1.addListener` `o1.removeListener`

`Bar.java`	1.47	`o2.addListener`
		`o2.removeListener`
		`System.out.println`

`Baz.java`	1.23	`o3.addListener`
		`o3.removeListener`
		`list.iterator`
		`iter.hasNext`
		`iter.next`

`Qux.java`	1.41 1.42	`o4.addListener` `o4.removeListener`

Figure 1: Method calls added across different revisions.

```
SELECT l.Callee AS CalleeL, r.Callee AS CalleeR,
     COUNT(*) AS SupportCount
     INTO pairs FROM items l, items r
WHERE l.FileId = r.FileId
     AND l.RevisionId = r.RevisionId
     AND l.InitialCallSequence = r.InitialCallSequence
GROUP BY l.Callee, r.Callee;
```

Figure 2: Find correlated pairs ⟨`CalleeL`, `CalleeR`⟩ of methods sharing the initial call sequence, calls to which are added in the same revision of a file identified by `FileId` and `RevisionId`.

2.1 Mining Approach

In order to speed-up the mining process, we pre-process the revision history extracted from CVS and store this information in a general-purpose database; our techniques are further described in Zimmermann et al. [11]. This database contains method calls that have been inserted in each revision. To determine the calls inserted between two revisions r_1 and r_2, we build abstract syntax trees (ASTs) for both r_1 and r_2 and compute the set of all calls C_1 and C_2, respectively, by traversing the ASTs. $C_2 \setminus C_1$ is the set of calls inserted between r_1 and r_2.

After the revision history database is set-up, i.e., all calls that were added are recorded in the `items` table, mining is performed using SQL queries. The query in Figure 2 produces *support counts* for each method pair, which is the number of times the two methods were added to revision history together. We perform filtering based on support counts to only consider method pairs that have a sufficiently high support.

2.2 Pattern Ranking

Filtering the patterns based on their support count is not enough to eliminate unlikely patterns. To better target user efforts, a ranking of our results is provided. In addition to sorting patterns by their support count, a common ranking strategy in data mining is to look at the pattern's *confidence*. The confidence determines how strongly a particular pair of methods is correlated and is computed as

$$conf(\langle a, b\rangle) = \frac{support(\langle a, b\rangle)}{support(\langle a, a\rangle)}$$

Both support count and confidence are standard ranking approaches used in data mining; however, using problem-specific

```
SELECT DISTINCT Callee
INTO fixes
FROM (SELECT MIN(Callee)
        FROM items GROUP BY FileId, RevisionId
        HAVING COUNT(*) = 1) t;
```

Figure 3: Find "fixed" methods, calls to which were added in at least one *one-call* check-ins. A one-call check-in adds exactly one call, as indicated by `COUNT(*) = 1`.

```
SELECT p.CalleeL, p.calleeR, SupportCount
FROM pairs p, fixes l, fixes r
WHERE p.calleeL = l.Callee
   AND p.calleeR = r.Callee
```

Figure 4: Find pairs from the table `pairs`, where both methods had previously been fixed, i.e., are contained in the table `fixed`.

knowledge yields a significantly better ranking of results. We leverage the fact that in reality some patterns may be inserted incompletely, e.g., by mistake or to fix a previous error. In Figure 1 this occurs in file `Qux.java`, where `addListener` and `removeListener` were inserted independently in revisions 1.41 and 1.42. The observation below motivates a novel ranking strategy we use.

Observation 2.3 *Small changes to the repository such as one-line additions are often bug fixes.*

This observation is supported in part by anecdotal evidence and also by recent research into the nature of software changes. A recent study of the dynamic of small repository changes in large software systems performed by Purushothaman et al. sheds a new light on this subject [6]. Their paper points out that almost 50% of all repository changes were small, involving less than 10 lines of code. Moreover, among one-line changes, less than 4% were likely to cause a later error. Furthermore, only less than 2.5% of all one-line changes were *perfective* or adding functionality (rather than *corrective*) changes, which implies that most one-line check-ins are bug fixes.

We use this observation by marking method calls that are frequently added in one-line fixes as *corrective* and ranking patterns by the number of corrective calls they contain. The SQL query in Figure 3 creates table `fixes` with all corrective methods, calls to which were added as one-line check-ins. Finally, as shown in Figure 4, we find all method pairs where *both* methods are corrective. We favor these patterns by ranking them higher than other patterns.

3 Dynamic Analysis

Similarly to previous efforts that looked at detecting usage rules [9, 10], we use revision history mining to find common coding patterns. However, we combine revision history mining with a bug-detection approach. Moreover, our technique looks for pattern violations at runtime, as opposed to using a static analysis technique. This choice is justified by several considerations outlined below.

Scalability. Our original motivation was to be able to analyze Eclipse, which is one of the largest Java applications ever created. The code base of Eclipse contains of more than 2,900,000 lines of code and 31,500 classes. Most of the patterns we are interested in are spread across multiple methods and need a precise interprocedural approach to analyze. Given the substantial size of the application, precise whole-program flow-sensitive static analysis can be prohibitively expensive.

Validating discovered patterns. A benefit of using dynamic analysis is that we are able to "validate" the patterns we discover through CVS history mining as real usage patterns by observing how many times they occur at runtime. While dynamic analysis is unable to prove properties that hold in all executions, patterns that are matched many times with only a few violations represent likely patterns. With static analysis, validating patterns would not generally be possible unless flow-sensitive *must* information is available.

False positives. Runtime analysis does not suffer from false positives because all pattern violations detected with our system actually *do* occur. This significantly simplifies the process of error reporting and addresses the issue of false positives.

Automatic error repair. Finally, only dynamic analysis provides the opportunity to fix the problem on the fly without any user intervention. This is especially appropriate in the case of matching method pair when the second method call is missing. While we have not implemented automatic "pattern repair", we believe it to be a fruitful future research direction.

While we believe that dynamic analysis is more appropriate than static analysis for the problem at hand, dynamic analysis has a few well-known problems of its own. A serious shortcoming of dynamic analysis is its lack of coverage. In our dynamic experiments, we managed to find runtime use cases for some, but not all patterns discovered through revision history mining. Furthermore, the fact that a certain pattern appears to be a strong usage pattern based on dynamic data should be treated as a a suggestion, but not a proof.

We use an offline dynamic analysis approach that instruments all calls to methods that may be included in the method pairs of interest. We then post-process the resulting dynamic trace to find properly matched or mismatched method pairs.

4 Preliminary Experience

In this section we discuss our practical experience of applying our system to Eclipse. Figure 5 lists matching pairs of methods discovered with our mining technique.[1] A quick glance at the table reveals that many pairs follow a specific naming strategy such as `pre`–`post`, `add`–`remove`, and `begin`–`end`. These pairs could have been discovered by simply pattern matching on the method names. However, a large number of pairs have less than obvious names to look for, including ⟨`HLock`, `HUnlock`⟩, ⟨`progressStart`, `progressEnd`⟩, and ⟨`blockSignal`, `unblockSignal`⟩. Finally, some pairs are very difficult to recognize as matching method pairs and require a detailed study of the API to confirm, such as ⟨`stopMeasuring`, `commitMeasurements`⟩, ⟨`suspend`, `resume`⟩, etc. Many more potentially interesting matching pair patterns become available if we consider lower support counts; for the experiments we have only considered patterns with a support of five or more.

We also found some unexpected matching method pairs consisting of a constructor call followed by a method call that at first we thought ware caused by noise in the data. One such pair is ⟨`OpenEvent`, `fireOpen`⟩. This pattern indicates that all objects of type `OpenEvent` should be "consumed" by passing them into method `fireOpen`. Violations of this pattern may lead to resource and memory leaks, a serious problem in long-running Java program such as Eclipse, which may be open at a developer's desktop for days at a time [7].

[1] The methods in a pair are listed in the order they are supposed to be executed. For brevity, we only list unqualified method names.

METHOD PAIR $\langle a, b \rangle$		CONFIDENCE			SUPPORT
Method a	**Method** b	$conf_{ab} \times conf_{ba}$	$conf_{ab}$	$conf_{ba}$	$count$
CORRECTIVE RANKING					
NewRgn	DisposeRgn	0.75	0.92	0.22	49
kEventControlActivate	kEventControlDeactivate	0.68	0.83	0.83	5
addDebugEventListener	removeDebugEventListener	0.61	0.85	0.72	23
beginTask	done	0.59	0.74	0.81	493
beginRule	endRule	0.59	0.80	0.74	32
suspend	resume	0.58	0.83	0.71	5
NewPtr	DisposePtr	0.57	0.82	0.70	23
addListener	removeListener	0.56	0.68	0.83	90
register	deregister	0.53	0.69	0.78	40
malloc	free	0.46	0.68	0.68	28
addElementChangedListener	removeElementChangedListener	0.41	0.73	0.57	8
addResourceChangeListener	removeResourceChangeListener	0.41	0.90	0.46	26
addPropertyChangeListener	removePropertyChangeListener	0.39	0.54	0.73	140
start	stop	0.38	0.59	0.65	32
addDocumentListener	removeDocumentListener	0.35	0.64	0.56	29
addSyncSetChangedListener	removeSyncSetChangedListener	0.34	0.62	0.56	24
REGULAR RANKING					
createPropertyList	reapPropertyList	1.00	1.00	1.00	174
preReplaceChild	postReplaceChild	1.00	1.00	1.00	133
preLazyInit	postLazyInit	1.00	1.00	1.00	112
preValueChange	postValueChange	1.00	1.00	1.00	46
addWidget	removeWidget	1.00	1.00	1.00	35
stopMeasuring	commitMeasurements	1.00	1.00	1.00	15
blockSignal	unblockSignal	1.00	1.00	1.00	13
HLock	HUnlock	1.00	1.00	1.00	9
addInputChangedListener	removeInputChangedListener	1.00	1.00	1.00	9
preAddChildEvent	postRemoveChildEvent	1.00	1.00	1.00	8
preRemoveChildEvent	postAddChildEvent	1.00	1.00	1.00	8
progressStart	progressEnd	1.00	1.00	1.00	8
CGContextSaveGState	CGContextRestoreGState	1.00	1.00	1.00	7
annotationAdded	annotationRemoved	1.00	1.00	1.00	7
OpenEvent	fireOpen	1.00	1.00	1.00	7
addInsert	addDelete	1.00	1.00	1.00	7

Figure 5: Matching method pairs discovered through CVS history mining. The support count is $count$, the confidence for $\{a\} \Rightarrow \{b\}$ is $conf_{ab}$, for $\{b\} \Rightarrow \{a\}$ it is $conf_{ba}$. The pairs are ordered by $conf_{ab} \times conf_{ba}$.

4.1 Experimental Setup

In this section we describe our revision history mining and dynamic analysis setup. When we performed the pre-processing on Eclipse, it took about four days to fetch all revisions over the Internet because the complete revision data is about 6GB in size and the CVS protocol is not well-suited for retrieving large volumes of history. Computing inserted methods by analyzing the ASTs and storing this information in a database took about a day.

Figure 6 summarizes our dynamic results. Because the incremental cost of checking for additional patterns at runtime is generally low, when reviewing the patterns for inclusion in our dynamic experiments, we were fairly liberal in our selection. We usually either looked at the method names involved in the pattern or briefly examined a few usage cases. We believe our setup to be quite realistic, as we cannot expect the user to spend hours poring over the patterns. Overall, 50% of all patterns we chose turned out to be either usage or error patterns; had we been more selective, a higher percentage of patterns would have been confirmed dynamically. To obtain dynamic results, we ran each application for a few minutes, which typically resulted in a few hundred or thousand dynamic events being generated.

After we obtained the dynamic results, the issue of how to count errors arose. A single pattern violation at runtime involves one or more objects. We can obtain a *dynamic count* by counting how many different objects participated in a particular pattern violation during program execution. The dynamic error count is highly dependent on how we use the program at runtime and can be easily influenced by, for example, rebuilding a project in Eclipse multiple times. However, dynamic counts are not representative of the work a developer has to do to fix an error, as many dynamic violations can be caused by the same error in the code. To provide a better metric on the number of errors found in the application code, we also compute a *static count*. This is done by mapping each method participating in a pattern to a static call site and counting the number of unique call site combinations that are seen at runtime. Static counts are obtained for both validated and violated dynamic patterns.

Dynamic and static counts are shown in parts 2 and 3 of the table, respectively. The rightmost section of the table shows a classification of the patterns. We use information about how many times each pattern is validated and how many times it is violated to classify the patterns. Let v be the number of validated instances of a pattern and e be the number of its violations. We define an error threshold $\alpha = min(v/10, 100)$. Based on the validation and violation counts v and e, patterns can be loosely classified into the following categories:

- **Likely usage patterns**: patterns with a sufficiently high support that are mostly validated with relatively few errors ($e < \alpha \wedge v > 5$).
- **Likely error patterns**: patterns that have a significant number of validated cases as well as a large number of violations ($\alpha \leq e \leq 2v$).
- **Unlikely patterns**: patterns that do not have many vali-

Method pair $\langle a, b \rangle$		Dynamic events		Static events		Pattern Type		
Method a	**Method b**	**Validated**	**Errors**	**Validated**	**Errors**	**Usage**	**Error**	**Unlikely**
Corrective Ranking								
`addDebugEventListener`	`removeDebugEventListener`	4	1	4	1			✓
`beginTask`	`done`	334	642	42	21		✓	
`beginRule`	`endRule`	7	0	4	0	✓		
`addListener`	`removeListener`	118	106	35	26		✓	
`register`	`deregister`	1,279	313	6	7		✓	
`addResourceChangeListener`	`removeResourceChangeListener`	25	4	19	4			✓
`addPropertyChangeListener`	`removePropertyChangeListener`	1,789	478	55	25		✓	
`start`	`stop`	40	36	9	12		✓	
`addDocumentListener`	`removeDocumentListener`	39	1	14	1	✓		
Result subtotals for the corrective ranking scheme:		**3,635**	**1,581**	**188**	**97**	**2**	**5**	**2**
Regular Ranking								
`preReplaceChild`	`postReplaceChild`	40	0	26	0	✓		
`preValueChange`	`postValueChange`	63	2	11	2	✓		
`addWidget`	`removeWidget`	1,264	16	5	2	✓		
`preRemoveChildEvent`	`postAddChildEvent`	0	172	0	3			✓
`annotationAdded`	`annotationRemoved`	0	8	0	2			✓
`OpenEvent`	`fireOpen`	0	3	0	1			✓
Result subtotals for the regular ranking scheme:		**1,367**	**201**	**42**	**10**	**3**	**0**	**3**
Overall totals:		**5,002**	**1,782**	**230**	**107**	**5**	**5**	**5**

Figure 6: Result summary for the validated usage and error patterns in Eclipse.

dated cases or cause too many errors to be usage patterns ($e > 2v \vee v \leq 5$).

About a half of all method pair patterns that we selected from the filtered mined results were confirmed as likely patterns, out of those 5 were usage patterns and 5 were error patterns.

Overall, corrective ranking was significantly more effective than regular ranking schemes that are based on the product of confidence values. The top half of the table that addresses patterns obtained with corrective ranking contains 16 matching method pairs; so does the second half that deals with the patterns obtained with regular ranking. Looking at the subtotals for each ranking scheme reveals 188 static validating instances with corrective ranking vs only 42 for regular ranking; 97 static error instances are found vs only 10 for regular ranking. Finally, 7 patterns found with corrective ranking were dynamically confirmed as either error or usage patterns vs 3 for regular ranking. This confirms our belief that corrective ranking is more effective.

5 Conclusions

In this paper we presented an approach for discovering matching method pair patterns in large software systems and finding their violations at runtime. Our framework uses information obtained by mining software revision repositories in order to find good patterns to check. User input may be used to further restrict the number of checked patterns. Checking of patterns occurs during program execution, with the help of dynamic instrumentation.

We experimentally evaluated our system on Eclipse, a very large Java application totalling more than 2,900,000 lines of code shows that our approach is highly effective at finding a variety of previously unknown patterns. Overall, we discovered a total of 32 matching method pairs in our benchmarks. Out of these, 5 turned out to be dynamically confirmed usage patterns and 5 were frequently misused error patterns responsible for many of the bugs. Our ranking approach that favors corrective ranking overperformed the traditional data mining ranking strategies at identifying good patterns. In our experiments, 1,782 dynamic pattern violations were responsible for a total of 107 dynamically confirmed errors in the source code.

References

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft, 2004.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, June 7–14 2003.

[3] G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[4] 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 Implentation*, pages 1–16, 2000.

[5] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, May 2004.

[6] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 90–94, May 2004.

[7] B. A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[8] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, San Diego, California, Feb. 2000.

[9] C. Williams and J. K. Hollingsworth. Bug driven bug finders. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 70–74, May 2004.

[10] C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, May 2005.

[11] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, May 2004.

Evaluating a Lightweight Defect Localization Tool

Valentin Dallmeier Christian Lindig Andreas Zeller
Saarland University
Department of Computer Science
Saarbrücken, Germany
{dallmeier,lindig,zeller}@cs.uni-sb.de

ABSTRACT

AMPLE locates likely failure-causing classes by *comparing method call sequences* of passing and failing runs. A difference in method call sequences, such as multiple deallocation of the same resource, is likely to point to the erroneous class. In this paper, we describe the implementation of AMPLE as well as its evaluation.

1. INTRODUCTION

One of the most lightweight methods to locate a failure-causing defect is to compare the *coverage* of passing and failing program runs: A method executed only in failing runs, but never in passing runs, is correlated with failure and thus likely to point to the defect. Some failures, though, come to be only through a *sequence* of method calls, tied to a *specific object*. For instance, a failure may occur because some API is used in a specific way, which is not found in passing runs.

To detect such failure-correlated call sequences, we have developed AMPLE[1], a plugin for the development environment Eclipse that helps the programmer to locate failure causes in Java programs. AMPLE works by comparing the method call sequences of passing JUnit test cases with the sequences found in the failing test (Dallmeier et al., 2005). As a result, AMPLE presents a class ranking with those classes at the top that are likely to be responsible for the failure. A programmer looking for the bug thus is advised to inspect classes in the presented order.

Figure 1 presents a programmer's view of AMPLE as a plugin for Eclispe. The programmer is working on the source code for the AspectJ compiler for which a JUnit test case has failed, as was reported in AspectJ bug report #30168. AMPLE instruments the classes of AspectJ on the byte-code level and runs the failing test for observation again, as well a passing test case. As a result, it presents a class ranking

[1]Analyzing Method Patterns to Locate Errors

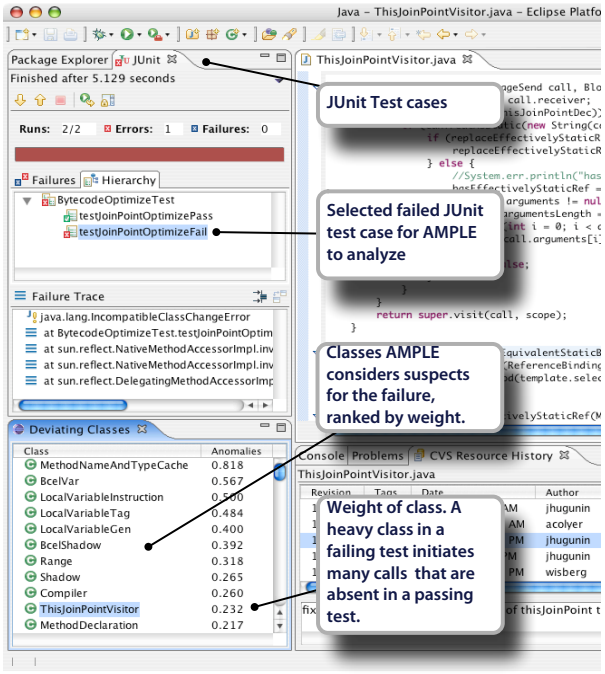


Figure 1: The AMPLE plugin in Eclipse: based on one passing and one failing JUnit test case, AMPLE presents a class ranking in the view *Deviating Classes*. High-ranking classes are suspect because their behavior deviated substantially between passing and failing runs. The AspectJ bug #30168 shown was fixed in the class at position #10, out of 2,929 classes. Our improved ranking algorithm now places the class at position #6.

in view *Deviating Classes*; each class is associated with a weight—its likeliness to be responsible for the failure. The actual bug was fixed in class `ThisJoinPointVisitor`, ranked at position 6, out of 997 classes.

While such anecdotal evidence for the predictive power of AMPLE is nice, we had to evaluate AMPLE in a more systematic way. In this paper, we briefly describe the implementation of AMPLE (Section 2), before discussing its evaluation (Section 3) as well as related work (Section 4). In Section 5, we describe our experiences from the evaluation, and make

suggestions for future similar evaluations.

2. AMPLE IN A NUTSHELL

AMPLE works on a hypothesis first stated by Reps et al. (1997) and later confirmed by Harrold et al. (1998): faults correlate with differences in traces between a correct and a faulty run. A trace is a sequence of actions observed over the lifetime of a program. AMPLE traces the control flow of a class by observing the calls invoked from within its methods. To rank classes, AMPLE compares the method call sequences from multiple passing runs and one failing run. Those classes that call substantially different methods in the failing run than in a passing run are suspect. These are ranked higher than classes that behave similarly in both runs.

AMPLE captures for each object the sequence of methods it calls. To gather such a *trace*, it instruments the program on the byte-code level using BCEL (Dahm, 1999). However, capturing the trace of calls for every object in a program is unfeasable for a number of reasons: the amount of trace data would lead to a high runtime overhead (Reiss and Renieris, 2001). While objects initiate calls they have no source-code representation, only classes do. We therefore rather need a characterization for classes. And finally, the differences between traces need to be qualified, just comparing traces for equality would be too coarse.

AMPLE's solution to these issues are *call-sequence sets*. A call-sequence set contains short sequences of consecutive calls initiated by an object. A call-sequence set is computed from a trace by sliding a window over it: given a string of calls $S = \langle m_1, \ldots, m_n \rangle$ and a window width k, the call-sequence set $P(S,k)$ holds the k-long substrings of S: $P(S,k) = \{w \mid w \text{ is a substring of } S \wedge |w| = k\}$. For example, consider a window of size $k = 2$ slid over S and the resulting set of sequences $P(S,2)$:

$$S = \langle abcabcdc \rangle \qquad P(S,2) = \{ab, bc, ca, cd, dc\}$$

Call-sequence sets have many advantages over traces: (1) they are compact because a trace typically contains the same substring many times; (2) call-sequence sets can be aggregated: we obtain a characterization of a class by aggregating the call-sequence sets of its objects; and (3), call-sequence sets are meaningful to compare, in particular the call-sequence sets from different runs of the same class.

To find classes whose behavior differs between passing and failing runs, AMPLE computes a call-sequence set for each class in the failing and passing runs. Looking at all call sequences observed for a class, it finds some call sequences common to all runs, some that occurred only in passing runs, and others that occurred only in failing runs. Each call sequence is weighted such that sequences that occur in the failing run, but never or seldom in passing runs, are assigned a larger weight than common call sequences.

Given these weights for sequences, a class is characterized by its *average call-sequence weight*. Classes with a high average sequence weight exhibit many call sequences only present in the failing run, and thus are prime suspects. As a result, classes are ranked by decreasing average call-sequence weight, as shown in Figure 1.

Version	*Classes*	*LOC*	*Faults*	*Tests*	*Drivers*
1	16	4334	7	214	79
2	19	5806	7	214	74
3	21	7185	10	216	76
5	23	7646	9	216	76
total		24971	33		

Table 1: Four versions of NanoXML, the subject of our controlled experiment.

Call-sequences sets can be computed directly, without capturing a trace first. The resulting runtime and memory overhead varies widely, depending on the number of objects that a program instantiates. We measured for the SPEC benchmark (SPEC, 1998) a typical runtime-overhead factor of 10 to 20, as well as a memory-overhead factor of two. While this may sound prohibitive, it is comparable to the overhead of a simpler coverage analysis with JCoverage (Morgan, 2004). We also believe that statistical sampling, as proposed by Liblit et al. (2003), can reduce the overhead for programs that instantiate many objects.

3. EVALUATION

For the systematic evaluation of AMPLE, we picked NanoXML as our test subject. NanoXML is a small, non-validating XML parser implemented in Java that Do et al. (2004) have pre-packed as a test subject. It is intended for the evaluation of testing methods and can be obtained[2] by other researchers directly from Do and others, which ensures reproducibility and comparability of our results.

The package provided by Do et al. includes the source code of five development versions, each comprising between 16 and 22 classes (Table 1). Part of the package are 33 defects that may be individually activated in the NanoXML source code. These defects were either found during the development of NanoXML, or seeded by Do and others.

The NanoXML package also contains over 200 test cases. Related test cases are grouped by test drivers, of which there are about 75. Test cases and defects are related by a fault matrix, which is also provided. The fault matrix indicates for any test, which of the defects it uncovers. Because development version 4 lacked a fault matrix, we could only use the other four versions listed in Table 1 for the evaluation of AMPLE.

3.1 Experimental Setup

The main question for our evaluation was: How well does AMPLE locate the defect that caused a given failure? To model this situation, we selected test cases from NanoXML that met all of the following conditions:

- We let AMPLE analyze a version of NanoXML with *one known defect*, which manifests itself in a faulty class.
- As *failing run*, we used a test case that uncovered the known defect.

[2]from `http://csce.unl.edu/~galileo/sir/`

- As *passing runs*, we selected *all* test cases that did not uncover the known defect.
- All test cases for passing and failing runs must belong to the *same driver*. This limits the number of passing runs to those that are semantically related to the failing run.

Altogether, we found 386 such situations, which therefore lead to 386 class rankings. Each ranking included one class with a known defect.

Note that AMPLE also works for test cases whose failure is caused by a combination of bugs or bugs whose fix involves more than one class. However, we did not evaluate these settings.

3.2 Evaluation Results

To evaluate a class ranking, we consider a ranking as advice for the programmer to inspect classes in the presented order until she finds the faulty class. In the experiment, the position of the known faulty class reflects the quality of the ranking: the higher the faulty class is ranked, the better the ranking. More precisely, we took the *search length* as the measure of quality: the number of classes atop of the faulty class in the ranking. This is the number of classes a programmer must inspect *before* she finds the faulty class.

Table 2 shows the average search length values over 386 rankings for various window sizes k. The numbers in row *Object* present the search length computed from sequence sets as discussed in Section 2. For example, with a window size $k = 7$, a programmer would have to inspect 1.98 classes in vain before finding the faulty class.

The numbers in row *Class* stem from an alternative way to compute sequence sets: rather than computing sequence sets per object and joining them into one sequence set per class, one sequence set per class is computed directly. This should be problematic for threaded programs (which NanoXML isn't)—details can be found in Dallmeier et al. (2005).

3.3 Discussion

A comparison with random guessing provides a partial answer for how good the numbers in Table 2 are. On average, a test run of NanoXML utilizes 19.45 classes, from which 10.56 are actually executed. Without any tool support or prior knowledge, a programmer on average would have to inspect half of these before finding the faulty class. This translates into a search length of $(10.56-1)/2 = 4.78$ when considering only executed classes, or 9.22 when considering all classes. All observed search lengths are better than random guessing, even under the assumption that the programmer could exclude non-executed classes (which is unlikely without tool support). Hence, AMPLE's recommendations are useful.

The search length in Table 2 is minimal for window sizes around 4 and 5. We have not analyzed this in detail but find the following plausible: larger window sizes in general provide more useful context than smaller window sizes, which leads to smaller search lengths. But large window sizes require objects to be long-lived in order to fill the window.

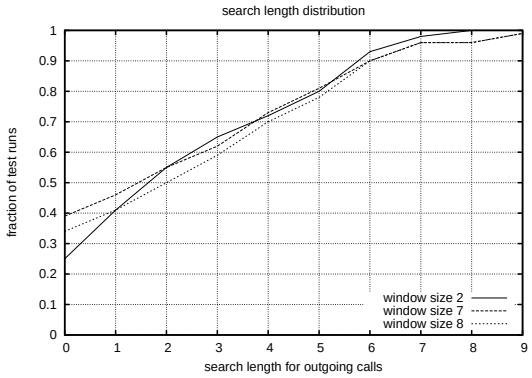


Figure 2: Distribution of search length for NanoXML. Using a window size of 7, the defective class is pinpointed (search length 0) in 38% of all test runs.

With larger window sizes fewer such objects exist, leading to a higher search length. These two opposing forces seem to balance out for window sizes around 4 and 5.

For the average search length to be meaningful, the search length must be distributed normally. Since we could not be sure of this, we examined the actual distribution of the search length. Figure 2 shows the distribution for selected window sizes and confirms the usefulness of AMPLE: with a search length of 2, the faulty class can be found in 50% to 60% of all cases, and in 38% of all cases for $k = 7$, the faulty class is right at the top of the ranking.

Does AMPLE perform better than existing techniques? This question is much harder to answer because the technique closest to ours works on the level of statements, rather than classes: defect localization based on coverage in TARANTULA compares the statement coverage of passing and failing runs (Jones et al., 2002). Statements executed more often in failing runs than in passing runs are more likely to have caused a failure. While this technique also assigns a weight to source code entities, it does it at a much finer granularity, which makes a direct comparison infeasible.

We have argued in Dallmeier et al. (2005) by analogy. A window size of one is equivalent to coverage analysis: the sequence-call set for a window size of $k = 1$ holds just the executed methods. Since the search length for all call-sequence sets with a window size $k \geq 2$ is smaller, the extra context provided by a larger window is obviously useful. This suggests that call-sequence sets outperform pure coverage analysis.

We admit that suggesting entire classes for inspection is quite coarse-grained. Individual methods could be suggested for inspection by taking into account not the class, but the method that invokes a call. This is planned for future work.

3.4 Does it Scale?

While we were satisfied with the results from our systematic evaluation of NanoXML, we expected critics to find NanoXML too small a subject. After all, AMPLE saved us only the inspection of less than three classes on average, compared to random guessing. We therefore recently complemented our evaluation with the AspectJ compiler as another test subject (Dallmeier, 2005).

		Window Size										*Random Guess*	
Subject	*Trace*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*Executed*	*All*
NanoXML	Object	2.53	2.31	2.19	2.17	2.04	2.00	1.98	2.12	2.15	2.14	4.78	9.22
	Class	2.53	2.35	2.22	2.14	2.03	2.04	2.03	2.02	2.22	2.25	4.78	9.22
AspectJ	Object	32.4	31.8	30.8	10.2	8.6	23.4	22.6	23.8	24.4	24.0	209	272
	Class	32.4	32.2	34.8	12.8	12.4	25.2	24.8	25.2	25.2	25.6	209	272

Table 2: Evaluation of class rankings. A number indicates the average *search length*: the number of classes atop of the faulty class in a ranking. The two rightmost columns indicate these numbers for a random ranking when (1) considering only executed classes, (2) all classes.

			Size (LOC)	
Bug ID	*Version*	*Defective Class*	*Class*	*Fix*
29691	1.1b4	org.aspectj.weaver.patterns.ReferencePointcut	294	4
29693	1.1b4	org.aspectj.weaver.bcel.BcelShadow	1901	8
30168	1.1b4	org.aspectj.ajdt.internal.compiler.ast.ThisJoinPointVisitor	225	20
43194	1.1.1	org.aspectj.weaver.patterns.ReferencePointcut	299	4
53981	1.1.1	org.aspectj.ajdt.internal.compiler.ast.Proceed	133	19

Table 3: Bugs in AspectJ used for the evaluation of AMPLE.

AspectJ is a compiler implemented in Java and consists in version 1.1.1 of 979 classes, representing 112,376 lines of code. Unlike NanoXML, it does not come pre-packed with a set of defects, and therefore it was not possible to use it in a systematic evaluation. But its developers have collected bug reports and provide a source code repository which documents how bugs were fixed. From these we could reconstruct passing and failing test cases for the further evaluation of AMPLE.

In order to obtain results comparable with our evaluation using NanoXML, we restricted ourself to bugs whose fixes involved only one Java class; Table 3 shows the 5 bugs that we found, and Table 2 the observed average search lengths for window sizes up to 10.

The average search lengths in Table 2 confirm that AMPLE scales to large programs and works for real-world bugs. Again, rankings for a window size of $k = 1$ perform worse than wider windows. Compared with random guessing, AMPLE saves the programmer the inspection of 177 classes.

4. RELATED WORK

Locating defects that cause a failure is a topic of active research that has proposed methods ranging from simple and approximative to complex and precise.

Comparing multiple runs. The work closest to ours is TARANTULA by Jones et al. (2002). Like us, they compare a passing and a failing run for fault localization, albeit at a finer granularity: TARANTULA computes the statement coverage of C programs over several passing and one failing run. While a direct comparison is difficult, we have argued by analogy in Section 3.3 that sequence sets as a generalization of coverage perform better.

Data anomalies. Rather than focusing on diverging control flow, one may also focus on differing data. *Dynamic invariants,* pioneered by Ernst et al. (2001), is a predicate for a variable's value that has held for all program runs during a training phase. If the predicate is later violated by a value in another program run this may signal an error. Learning dynamic invariants takes a huge machine-learning apparatus; a more lightweight technique for Java was proposed by Hangal and Lam (2002).

Isolating failure causes. To localize defects, one of the most effective approaches is isolating *cause transitions* between variables, as described by Cleve and Zeller (2005). Again, the basic idea is to compare passing and failing runs, but in addition, the delta debugging technique generates and tests *additional runs* to isolate failure-causing variables in the program state (Zeller, 2002). Due to the systematic generation of additional runs, this technique is precise, but also demanding—in particular, one needs a huge apparatus to extract and compare program states. In contrast, collecting call sequences is far easier to apply and deploy.

5. CONCLUSIONS AND CONSEQUENCES

Like other defect detection tools, AMPLE can only make an *educated guess* about where the defect in question might be located. At a very fundamental level, this is true for any kind of defect detection: If we define the defect as the part of the code that eventually was fixed, no tool can exactly locate a defect, as this would mean predicting future actions of the programmer. With AMPLE, we make that guess explicit—by *ranking* source code according to the assigned probability.

Such a ranking has the advantage of providing a straightforward measure of the tool's precision. The model behind the ranking is that there is an ideal programmer who can spot defects by looking at the code, and thus simply work her way through the list until the "official" defect is found. Thus, the smaller the search length, the better the tool.

To demonstrate the advance beyond the state of the art,

the ranking must obviously be better than a random ranking (which we showed for AMPLE), but also be better than a ranking as established by previously suggested techniques (which we also showed for AMPLE, but using an analogon rather than the real tool). This also requires that the test suites are publicly available, and that the rankings, as obtained by the tools, are fully published. This way, we can establish a number of benchmarks by which we can compare existing tools and techniques.

All this comes with a grain of salt: AMPLE starts with a given failure. Hence, we know that a defect must exist somewhere in the code; the question is where to locate it. Static analysis tools, in contrast, are geared towards the future, and help preventing failures rather than curing them. For a static analysis tool, the central issue is not so much where a defect is located, but whether a code feature should be flagged as defect or not.

Nonetheless, ranking is still a way to go here: Rather than setting a threshold for defects and non-defects, a static analysis tool could simply assign each piece of code a computed probability of being defective. If a caller and a callee do not match, for instance, both could be flagged as potential defects (although only one needs to be fixed). Again, such a probability could end in a ranking of locations, which can be matched against the "official" defects seeded into the test subject, or against a history of "official" fixes: "If the programmer examines the 10% of the ranked locations, she will catch 80% of the defects." Again, these are figures which can be compared for multiple defect detection tools.

While developing AMPLE, we found that such benchmarks help a lot in directing our research—just like test-driven development, we established a culture of "benchmark early, benchmark often". Comparing against other work is fun and gives great confidence when defending the results. We therefore look forward to the emergence of standard benchmarks which will make this young field of defect detection rock-solid and fully respected.

References

Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. 27th International Conference of Software Engineering (ICSE 2005)*, St. Louis, USA, 2005. To appear.

Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, July 07 1999. URL `http://www.inf.fu-berlin.de/~dahm/JavaClass/ftp/report.ps.gz`.

Valentin Dallmeier. Detecting failure-related anomalies in method call sequences. Diploma thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, March 2005.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, July 2005. To appear. Also available from `http://www.st.cs.uni-sb.de/papers/dlz2004/`.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *International Symposium on Empirical Software Engineering*, pages 60–70, Redondo Beach, California, August 2004.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301, New York, May 19–25 2002. ACM Press.

Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), ACM SIGPLAN Notices*, pages 83–90, Montreal, Canada, July 1998.

James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proc. International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, Florida, May 2002.

Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In Jr. James B. Fenwick and Cindy Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 141–154, New York, June 9–11 2003. ACM Press.

Peter Morgan. JCoverage 1.0.5 GPL, 2004. URL `http://www.jcoverage.com/`.

Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 221–232, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

Thomas Reps, Thomas Ball, Manuvir Das, and Jim Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

SPEC. SPEC JVM 98 benchmark suite. Standard Performance Evaluation Corporation, 1998.

Andreas Zeller. Isolating cause-effect chains from computer programs. In William G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, volume 27, 6 of *Software Engineering Notes*, pages 1–10, New York, November 18–22 2002. ACM Press.

The Open Source Proving Grounds

Ben Liblit
Computer Sciences Department
University of Wisconsin-Madison
<liblit@cs.wisc.edu>

Open source software is an attractive target when developing and evaluating software defect detection tools. This position paper discusses some of the benefits and challenges I have encountered as a researcher interacting with the open source community.

The Cooperative Bug Isolation Project (*CBI*) explores ways to identify and fix bugs in widely deployed software. We use a mix of lightweight instrumentation and statistical modeling techniques to reveal failure patterns in large numbers of end user runs. The ultimate validation of the CBI approach comes when real data from real users helps us find real bugs in real code. Thus, field deployment is a key component of this project. CBI offers instrumented binaries for several popular open source programs for anyone to download and use.

An Open Marketplace of Code and Ideas

Most open source projects expose their entire development process to the public. This includes much more than just source code. Certainly, I can grab millions of lines of code at any hour of the day or night. However, the more disciplined projects also have revision histories, design documents, regression test suites, bug tracking systems, release schedules... all the trappings of "real" software development, free for the taking. All of this comes with no nondisclosure agreements, no lawyers, and no limits on publication of potentially embarrassing defect data.

Furthermore, open source software has gained enough market- and mind-share that it is seen as realistic. Scientific progress demonstrated on open source software is assumed to apply equally well to proprietary systems and to software engineering in general. In the marketplace of research, open source is now considered legitimate.

Openness also facilitates feedback of defect findings to project developers. As CBI discovers bugs in target applications, we report them using the same bug tracking systems used by the developers themselves. Our bug reports must compete with all others for developers' attention. If our reports are clear and describe important bugs, we gain credibility and our patches are accepted gladly. Uninformative or unimportant reports languish. Thus, observing how developers respond to the information we provide is itself an important part of evaluating our tools. The transparency of open source projects makes this process much easier to observe.

At the same time, open source licenses mean we don't need developers' permission, and there's not much they could do to either help or hinder our work. As one developer put it, if there's even a tiny chance that CBI will find a single bug, he's all in favor of it, and in any case it costs him nothing to let us try. A disadvantage stemming from this openness is that open source *distributors* are only loosely connected to many open source *developers*. It has been difficult to convert developer enthusiasm into a truly large scale deployment backed by any commercial open source vendor such as Red Hat or Novell. Ultimately the challenges here are no different from the challenges faced in partnering with any large company. Even so, it can come as a surprise that the lead developer on a project has little or no influence on a third-party Linux vendor that does not pay his salary and simply downloads his source code just like everyone else.

Finding a User Community

Many open source projects feel that because they provide source, they are under no obligation to provide compiled binaries. However, these applications can be quite complex with many dependencies that make them difficult for end users to build. For research that includes dynamic analysis of deployed software, this source/binary gap creates an opportunity for barter with end users. I spend the time to build clean, tested, installable binary packages of open source applications for CBI. In exchange, the users who download these packages agree to provide me with the raw data I need to do my research. Several people have told me that they use CBI releases simply because they are the easiest way to get the desired applications installed and running on their machines.

This approach to finding users has some disadvantages as compared with piggybacking on a commercial release. It is easy to get a few users this way, but hard to get truly large numbers. All the traffic CBI accrues from download links cannot come close to what Microsoft would get by instrumenting even a small fraction of, say, shrink-

wrapped Office 2003 CD's. Anything requiring an explicit download and install naturally selects for a more technical user base whose behavior may not be representative of software usage patterns in general. In spite of these factors, the relative ease with which one can get a small user community makes binary distribution of open source applications an attractive option for research involving deployed software.

Effects of Shortened Release Cycles

Open source projects typically release new versions earlier and more often than their commercial equivalents. This can be an advantage or a disadvantage from a research perspective. Early releases make projects' "dirty laundry" more visible. When hunting for bugs, early releases from young open source projects are a target-rich environment. Feedback provided to developers can be incorporated quickly; one project posted a new release specifically in response to bugs reported by CBI. As noted above, enthusiastic developer response is strong validation for any defect detection tool.

On the other hand, it can be difficult for software quality researchers to keep up with a moving target. CBI depends on accumulating many examples of good and bad runs. If new releases come out every few weeks, there is little time to accumulate data for any single snapshot of the code. However, if we stop tracking each new release, then users eager to try the latest and greatest may wander elsewhere. With access to many binary providers, as well as the source itself, users have no strong reason to stay in one place. Commercial providers have a more captive audience. Combined with longer release cycles, this gives researchers operating in a commercial environment more time to collect data for analysis.

Conclusions

In hindsight, I have found that working with open source makes it easier to get started, but perhaps harder to get finished. The barriers to entry are low, making it very easy to try out any crazy scheme that comes to mind. A tool can sail or sink based on its technical merits, and feedback from real developers is a fast and direct validation channel. On the other hand, the decentralized nature of open source development means there is no clearly defined decision maker. There is no one who can declare by executive fiat that his engineering team will henceforth use my tools on all their millions of lines of code. Small deployments are easy to arrange, but large ones are difficult. Researchers working outside a commercial environment should keep these trade-offs in mind as they consider using open source as a proving grounds for their work.

Further Reading on Cooperative Bug Isolation

[1] Ben Liblit. The Cooperative Bug Isolation Project. `<http://www.cs.wisc.edu/cbi/>`.

[2] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In Jr. James B. Fenwick and Cindy Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 141–154. ACM Press, 2003.

[3] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Sampling user executions for bug isolation. In Alessandro Orso and Adam Porter, editors, *RAMSS '03: 1st International Workshop on Remote Analysis and Measurement of Software Systems*, pages 5–8, Portland, Oregon, May 9 2003.

[4] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Public deployment of cooperative bug isolation. In Alessandro Orso and Adam Porter, editors, *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 57–62, Edinburgh, Scotland, May 24 2004.

[5] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 11–17 2005.

[6] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

[7] Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

BUGS workshop 2005 Position Statement:

Issues in Deploying SW Defect Detection tools

David R. Cok
Kodak R&D Laboratories, Eastman Kodak Company
31 March 2005

In any technology, there is a large gap between a research prototype and a working, useful product; this is no less true with tools that embody complex, novel software technology. The various software defect detection tools available today, typically as some variation of open source software, exemplify this. The most successful are backed by a sizable research group with long-term research objectives that use the tool as a platform; the deploying of the tool requires considerable investment of time in careful coding, testing, documentation, and release procedures that does not necessarily benefit the academic research enterprise. Nevertheless, groups are often sufficiently passionate about the technology to invest the effort; however, this alone does not generate user demand for a product.

From an industrial software developer's point of view, a new tool must pass muster in two key user experience dimensions in order to be successful: usefulness and usability; furthermore a software development organization must have confidence in the tool supplier's longevity in order to adopt the tool.

Usefulness is a measure of the added value the tool supplies a user. It is not necessarily a measure of breadth of the tool, although tools such as IDEs that are meant to be working environments must be relatively complete. Indeed a tool that has all-encompassing claims may well generate suspicion rather than curiosity. Rather a defect detection tool must have these features:

It must be reasonably accurate in its output. From the user's perspective, the issue is not so much one of technical criteria like soundness and completeness; instead, the user's measure will be the fraction of false positives present and false negatives missed in the total output. False positives, in which a tool claims a code fragment to be a defect when the code is in fact valid, are noise to the user. The user must still assess (with the potential for error) each defect report and decide whether or not it is valid. Too high a fraction of useless reports will cause the user to see less benefit in the results of the tool overall. On the other hand, false negatives are defects that the tool fails to detect. The impact of false negatives can be mitigated if it can be clearly stated what types of defects the tool expects to address. False negatives within the domain of the tool, however, will reduce the trust the user has in the tool overall, since the user will need to invest duplicate manual effort in finding those errors.

Secondly*, the tool must address the bulk of the important defect areas for the user*. Importance can be measured in terms of user effort: important defect classes consume large fractions of the user's development and debugging time, because of the combination of the defect's frequency and the average effort to diagnose and correct each defect. Experimental case studies of actual programmer experience on large scale projects would be useful to determine which defect classes are the most problematic in the overall software engineering effort on typical projects.

And third, it must have time and space *performance and stability consistent with the benefit provided*.

Usability, or ease-of-use, describes the ease with which a user, particularly a novice user, can obtain useful results from a tool. There are a number of components to keep in mind:
- Out-of-the-box experience: how easy is it to install, configure and obtain results on a user example the very first time? Is the tool self-contained, or does one need to download and configure other tools first?
- Resource investment profile: how much time, money and effort must a user invest in order to obtain a small amount of initial benefit or invest incrementally to obtain increasing benefit? Can results be obtained on small portions of a large project? Can the tool be layered alongside of the user's existing working environment?
- Perspicuity: how clear are the reports provided by the tool? Do they actually save work? Can false positives, once evaluated, be annotated so that those false reports are not regenerated on subsequent invocations of the tool and the work of repeated defect report assessment avoided.

Finally, there is increasing willingness to adopt academic and open source tools that do not become part of a final commercial product. However, a software development organization has a need for **confidence** that the tool supplier will continue to support, correct, improve, and release the tool. This typically weighs against academic and open source tool suppliers and in favor of commercial tool vendors, even in the face of license fees, even though license fees place a considerable hurdle in the path of informal initial trial.

False Positives Over Time: A Problem in Deploying Static Analysis Tools

Andy Chou, Coverity Inc., andy@coverity.com

All source code analyzers generate *false positives*, or issues which are reported but are not really defects. False positives accumulate over time because developers fix real defects but tend to leave false positives in the source code. Several methods are available to mitigate this problem, some of which are shown in the following table (especially important advantages or disadvantages are labeled with (*)):

Technique	Advantages	Drawback(s)
Add annotations to the source code that indicate the location of false positives, which the tool can then use to suppress messages.	• Persistent across code changes and renamed files (*) • Seamlessly propagate between different code branches via version control systems • Also works for real bugs that users don't want to fix	• Some users are unwilling to add annotations to source code, even in comments (*) • Remain in the code even if analysis is changed to not find the false positives • Adding annotations to third party code is usually undesirable
Allow users to override analysis decisions that lead to false positives.	• Eliminates entire classes of false positives with same root cause (*) • Flexible: can be done using source annotations or tool configuration	• Difficult for users to understand (*) • Changing analysis decisions can have unexpected side-effects, such as missing other bugs
Change the source code to get the tool to "shut up."	• No changes to tool	• Not always obvious how to change the code to make the false positive go away (*) • Might not be possible to change the code in an acceptable way
Stop using the tool, or turn off specific types of checks causing the false positives.	• Simple • Minimize cost of dealing with false positive-prone analyses (*)	• Lost opportunity to discover real bugs
Rank errors using some criteria, or otherwise use statistical information to identify likely bugs vs. likely false positives.	• (mostly) Automatic • Adapts to application-specific coding conventions (*)	• Does not deal with all false positives • Larger development organizations want to distribute bugs to be inspected; each developer gets a small number of bugs to inspect, making ranking less useful (*) • Users don't like deciding when to stop; they fear missing bugs while simultaneously loathing false positives. • Users usually want "rank by severity" not "rank by false positive rate"
Annotate the output of the tool to mark false positives, then use this information in future runs to avoid re-reporting the same issues.	• No changes to code • Works for almost any type of static analysis (*)	• Need heuristics to determine when false positives are the "same" in the presence of code changes (*) • False positives may re-appear depending on the stability of the merging heuristic (*)

These are not the only ways of attacking this problem. Very little work has been done on classifying and evaluating these techniques, yet they are critical to the adoption of static analysis in industry. Some observations from Coverity customers include:

- Users are unwilling to add annotations unless the tool has already shown to be an efficient bug-finder.
- Users are usually unwilling to change source code to eliminate false positive warnings.
- Ranking errors by "likely to be a bug" can help pull real bugs to the forefront, but users have a hard time deciding where the cutoff should be. Users also dislike having no cutoff at all, because they will often query for bugs in a specific location of interest, where only a handful of results are found and ranking is not useful. Ranking by estimated severity of bug is more often useful.
- If users annotate tool output so the tool gains "memory," the heuristic used to determine that previous results are the "same" as new results must be robust. Users very much dislike false positives coming back.

Model Checking x86 Executables with CodeSurfer/x86 and WPDS++*

G. Balakrishnan[1], T. Reps[1,2], N. Kidd[1], A. Lal[1], J. Lim[1],
D. Melski[2], R. Gruian[2], S. Yong[2], C.-H. Chen, and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps,kidd,akash,junghee}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,radu,suan,chi-hua,tt}@grammatech.com

Abstract. This paper presents a tool set for model checking x86 executables. The members of the tool set are *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*. CodeSurfer/x86 is used to extract a model from an executable in the form of a *weighted pushdown system*. WPDS++ is a library for answering generalized reachability queries on weighted pushdown systems. The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++ that supports safety queries about the program's possible control configurations.

1 Introduction

This paper presents a tool set for model checking x86 executables. The tool set builds on (i) recent advances in static analysis of program executables [2], and (ii) new techniques for software model checking and dataflow analysis [5, 26, 27, 21]. In our approach, CodeSurfer/x86 is used to extract a model from an x86 executable, and the reachability algorithms of the WPDS++ library [20] are used to check properties of the model. The Path Inspector is a software model checker that automates this process for safety queries involving the program's possible control configurations (but not the data state). The tools are capable of answering more queries than are currently supported by the Path Inspector (and involve data state); we illustrate this by describing two custom analyses that analyze an executable's use of the run-time stack.

Our work has three distinguishing features:

- The program model is extracted from the executable code that is run on the machine. This means that it automatically takes into account platform-specific aspects of the code, such as memory-layout details (i.e., offsets of variables in the run-time stack's activation records and padding between fields of a struct), register usage, execution order, optimizations, and artifacts of compiler bugs. Such information is hidden from tools that work on intermediate representations (IRs) that are built directly from the source code.
- The entire program is analyzed—including libraries that are linked to the program.
- The IR-construction and model-extraction processes do not assume that they have access to symbol-table or debugging information.

Because of the first two properties, our approach provides a "higher fidelity" tool than most software model checkers that analyze source code. This can be important for certain kinds of analyses; for instance, many security exploits depend on platform-specific features, such as the structure of activation records. Vulnerabilities can escape notice when a tool does not have information about adjacency relationships among variables.

Although the present tool set is targeted to x86 executables, the techniques used [2, 27, 21] are language-independent and could be applied to other types of executables.

The remainder of the paper is organized as follows: Sect. 2 illustrates some of the advantages of analyzing executables. Sect. 3 sketches the methods used in CodeSurfer/x86 for IR recovery. Sect. 4 gives an overview of the model-checking facilities that the tool set provides. Sect. 5 discusses related work.

2 Advantages of Analyzing Executables

This section presents some examples that show why analysis of an executable can provide more accurate information than an analysis that works on source code.

One example of a mismatch between the program's intent and the compiler's generated code can be seen in the following lines of source code taken from a login program [18]:

```
memset(password, '\0', len);
free(password);
```

* Portions of this paper are based on a tool-demonstration paper of the same title that will appear at CAV 2005.

The login program (temporarily) stores the user's password—in clear text—in a dynamically allocated buffer pointed to by pointer variable `password`. To minimize the lifetime of sensitive information (in RAM, in swap space, etc.), the source-code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset` and therefore the call on `memset` can be removed, thereby leaving sensitive information exposed in the heap [18]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

A second example where analysis of an executable does better than typical source-level analyses involves pointer arithmetic and function pointers. Consider the following piece of code:

```
int (*f)(void);
int diff = (char*)&f2 - (char*)&f1; // The offset between f1 and f2
f = &f1;
f = (int (*)())(char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

Existing source-level analyses (that we know of) are ill-prepared to handle the above code. The common assumption is that pointer arithmetic on function pointers leads to undefined behavior, so either (a) they assume that the indirect function call might call any function; or (b) they simply ignore the arithmetic and assume that the indirect function call calls `f1` (on the assumption that the code is ANSI-C compliant). In contrast, the *value-set analysis* (VSA) algorithm [2] used in CodeSurfer/x86 correctly identifies `f2` as the invoked function. Furthermore, VSA can detect when pointer arithmetic results in a function pointer that does not point to the beginning of a function; the use of such a function pointer to perform a function "call" is likely to be a bug (or else a very subtle, deliberately introduced security vulnerability).

A third example involves a function call that passes fewer arguments than the procedure expects as parameters. (Many compilers accept such (unsafe) code as an easy way of implementing functions that take a variable number of parameters.) With most compilers, this effectively means that the call-site passes some parts of one or more local variables of the calling procedure as the remaining parameters (and, in effect, these are passed by reference—an assignment to such a parameter in the callee will overwrite the value of the corresponding local in the caller.) An analysis that works on executables can be created that is capable of determining what the extra parameters are [2], whereas a source-level analysis must either make a cruder over-approximation or an unsound under-approximation.

A final example is shown in Fig. 1. The C code on the left uses an uninitialized variable (which triggers a compiler warning, but compiles successfully). A source-code analyzer must assume that `local` can have any value, and therefore the value of `v` in `main` is either 1 or 2. The assembly listings on the right show how the C code could be compiled, including two variants for the prolog of function `callee`. The Microsoft compiler (cl) uses the second variant, which includes the following strength reduction:

> *The instruction* `sub esp,4` *that allocates space for* `local` *is replaced by a* `push` *instruction of an arbitrary register (in this case,* `ecx`*).*

In contrast to an analysis based on source code, an analysis of an executable can determine that this optimization results in `local` being initialized to 5, and therefore `v` in `main` can only have the value 1.

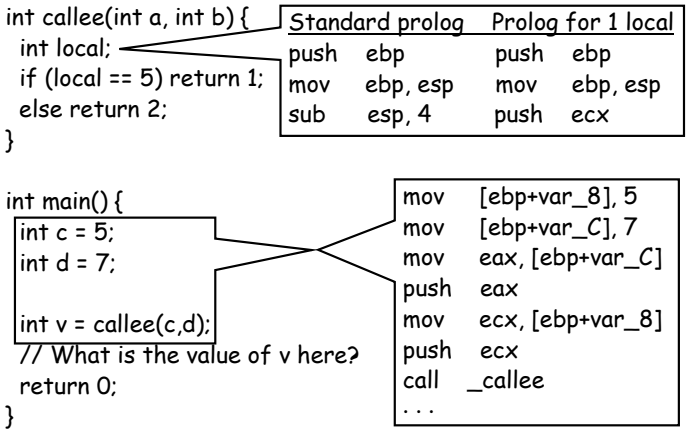


Fig. 1. Example of unexpected behavior due to the application of an optimization. The box at the top right shows two variants of code generated by an optimizing compiler for the prolog of `callee`. Analysis of the second of these reveals that the variable `local` always contains the value 5.

3 Recovering Intermediate Representations from x86 Executables

To recover IRs from x86 executables, CodeSurfer/x86 makes use of both IDAPro [19], a disassembly toolkit, and GrammaTech's CodeSurfer system [12], a toolkit for building program-analysis and inspection tools. Fig. 2 shows how the components of CodeSurfer/x86 fit together.

An x86 executable is first disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information: (1) procedure boundaries, (2) calls to library functions using an algorithm called the Fast Library Identification and Recognition Technology (FLIRT) [15], and (3) statically known memory addresses and offsets. IDAPro provides access to its internal resources via an API that allows users to create plug-ins to be executed by IDAPro. We created a plug-in to IDAPro, called the Connector, that creates data structures to represent the information that it obtains from IDAPro. The IDAPro/Connector combination is also able to create the same data structures for dynamically linked libraries, and to link them into the data structures that represent the program itself. This infrastructure permits whole-program analysis to be carried out—including analysis of the code for all library functions that are called.

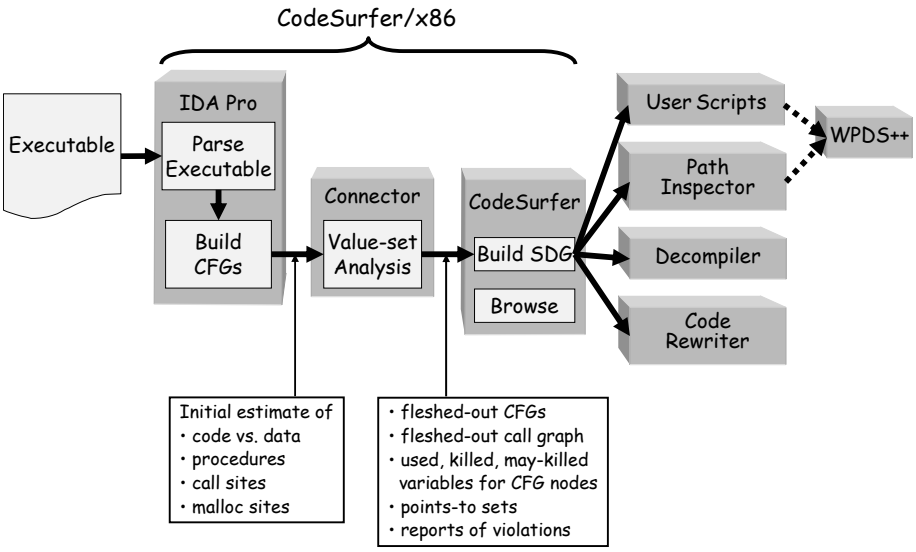


Fig. 2. Organization of CodeSurfer/x86 and companion tools.

Using the data structures in the Connector, we implemented a static-analysis algorithm called *value-set analysis* (VSA) [2]. VSA does not assume the presence of symbol-table or debugging information. Hence, as a first step, a set of data objects called a-locs (for "abstract locations") is determined based on the static memory addresses and offsets provided by IDAPro. VSA is a combined numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses (or *value-set*) that each a-loc holds at each program point. A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable at execution time.

IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. However, the information computed during VSA can be used to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

VSA also checks whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records (ARs) are pushed onto the stack on procedure entry and popped from the stack on procedure exit; a procedure does not modify the return address on stack; the program's instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program's data. If it cannot be confirmed that the executable conforms to the model, then the IR is possibly incorrect. For example, the call-graph can be incorrect if a procedure modifies the return address on the stack. Consequently, VSA issues an error report whenever it finds a possible violation of the standard compilation model; these represent possible memory-safety violations. The analyst can go over these reports and determine whether they are false alarms or real violations.

Once VSA completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer. CodeSurfer then builds a collection of IRs, consisting of abstract-syntax trees, control-flow graphs (CFGs), a call graph, a system dependence graph (SDG) [17], VSA results, the sets of used, killed, and possibly killed a-locs at each instruction, and information about the structure and layout of global memory, activation records, and dynamically allocated storage. CodeSurfer supports both a graphical user interface (GUI) and an API (as well as a scripting language) to provide access to these structures.

4 Model-Checking Facilities

Model checking [11] involves the use of sophisticated pattern-matching techniques to answer questions about the flow of execution in a program: a model of the program's possible behavior is created and checked for conformance with a model of expected behavior (as specified by a user query). In essence, model-checking algorithms explore the program's state-space and answer questions about whether a bad state can be reached during an execution of the program.

For model checking, the CodeSurfer/x86 IRs are used to build a *weighted pushdown system* (WPDS) [5, 26, 27, 21] that models possible program behaviors. Weighted pushdown systems are a model-checking technology that is similar to what is used in MOPS, a software model checker that has demonstrated the ability to find subtle security vulnerabilities in large code bases [6]. However, in contrast to the ordinary (unweighted) pushdown systems used in MOPS, the techniques available in the CodeSurfer/x86 tool set [27, 21] are capable of representing the (logically) infinite set of data valuations that may arise during program execution. This capability allows the CodeSurfer/x86 tool set to address certain kinds of security queries that cannot be answered by MOPS.

4.1 WPDS++ and Stack-Qualified Dataflow Queries

WPDS++ [20] is a library that implements the symbolic reachability algorithms from [27, 21] on weighted pushdown systems. We follow the standard approach of using a pushdown system (PDS) to model the interprocedural control-flow graph (one of CodeSurfer/x86's IRs). The stack symbols correspond to program locations; there is only a single PDS state; and PDS rules encode control flow as follows:

Rule	Control flow modeled
$q\langle u\rangle \hookrightarrow q\langle v\rangle$	Intraprocedural CFG edge $u \rightarrow v$
$q\langle c\rangle \hookrightarrow q\langle entry_P\ r\rangle$	Call to P from c that returns to r
$q\langle x\rangle \hookrightarrow q\langle\rangle$	Return from a procedure at exit node x

Given a configuration of the PDS, the top stack symbol corresponds to the current program location, and the rest of the stack holds return-site locations—much like a standard run-time execution stack.

Encoding the interprocedural control-flow as a pushdown system is sufficient for answering queries about reachable control states (as the Path Inspector does; see Sect. 4.2): the reachability algorithms of WPDS++ can determine if an undesirable PDS configuration is reachable [6]. However, WPDS++ also supports *weighted* PDSs. These are PDSs in which each rule is weighted with an element of a (user-defined) semiring. The use of weights allows WPDS++ to perform interprocedural dataflow analysis by using the semiring's *extend* operator to compute weights for sequences of rule firings and using the semiring's *combine* operator to take the meet of weights generated by different paths [27, 21]. (When the weights on rules are conservative abstract data transformers, an over-approximation to the set of reachable concrete configurations is obtained, which means that counterexamples reported by WPDS++ may actually be infeasible.)

The CodeSurfer/x86 IRs are a rich source of opportunities to check properties of interest using WPDS++. For instance, WPDS++ has been used to implement an illegal-stack-manipulation check: for each node n in procedure P, this checks whether the net change in stack height is the same along all paths from $entry_P$ to n that have perfectly matched calls and returns (i.e., along "same-level valid paths"). In this analysis, a weight is a function that represents a stack-height change. For instance, `push ecx` and `sub esp,4` both have the weight $\lambda height.height - 4$. Extend is (the reversal of) function composition; combine performs a meet of stack-height-change functions. (The analysis is similar to linear constant propagation [29].) When a memory access performed relative to r's activation record (AR) is out-of-bounds, stack-height-change values can be used to identify which a-locs could be accessed in ARs of other procedures.

VSA is an interprocedural dataflow-analysis algorithm that uses the "call-strings" approach [30] to obtain a degree of context sensitivity. Each dataflow fact is tagged with a call-stack suffix (or *call-string*) to form (call-string, dataflow-fact) pairs; the call-string is used at the exit node of each procedure to determine to which call site a (call-string, dataflow-fact) pair should be propagated. The call-strings that arise at a given node n provide an opportunity to perform stack-qualified dataflow queries [27] using WPDS++. CodeSurfer/x86 identifies induction-variable relationships by using the affine-relation domain of Müller-Olm and Seidl [24] as a weight domain. A *post** query builds an automaton that is then used to find the affine relations that hold in a given calling context—given by call-string *cs*—by querying the *post**-automaton with respect to a regular language constructed from *cs* and the program's call graph.

4.2 The Path Inspector

The Path Inspector provides a user interface for automating safety queries that are only concerned with the possible control configurations that an executable can reach. It uses an automaton-based approach to model checking: the query is specified as a finite automaton that captures forbidden sequences of program locations. This "query automaton" is combined with the program model (a WPDS) using a cross-product construction, and the reachability algorithms of WPDS++ are used to determine if an error configuration is reachable. If an error configuration is reachable, then *witnesses* (see [27]) can be used to produce a program path that drives the query automaton to an error state.

The Path Inspector includes a GUI for instantiating many common reachability queries [14], and for displaying counterexample paths in the disassembly listing.[3] In the current implementation, transitions in the query automaton are triggered by program points that the user specifies either manually, or using result sets from CodeSurfer queries. Future versions of the Path Inspector will support more sophisticated queries in which transitions are triggered by matching an AST pattern against a program location, and query states can be instantiated based on pattern bindings. Future versions will also eliminate (many) infeasible counterexamples by using transition weights to represent abstract data transformers (similar to those used for interprocedural dataflow analysis).

5 Related Work

Several others have proposed techniques to obtain information from executables by means of static analysis [22, 13, 9, 8, 10, 4, 3]. However, previous techniques deal with memory accesses very conservatively; e.g., if a register is assigned a value from memory, it is assumed to take on any value. VSA does a much better job than previous work because it tracks the integer-valued and address-valued quantities that the program's data objects can hold; in particular, VSA tracks the values of data objects *other than just the hardware registers*, and thus is not forced to give up all precision when a load from memory is encountered. This is a fundamental issue; the absence of such information places severe limitations on what previously developed tools can be applied to.

The basic goal of the algorithm proposed by Debray et al. [13] is similar to that of VSA: for them, it is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include *memory locations* in addition to registers. In their analysis, a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [9] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The work that is most closely related to VSA is the data-dependence algorithm for executables proposed by Amme et al. [1]. However, that algorithm performs only an intraprocedural analysis, and it is not clear that the algorithm fully accounts for dependences between memory locations.

Several people have developed techniques to analyze executables in the presence of additional information, such as the source code, symbol-table information, or debugging information [22, 28]. For many security-related applications, these are not appropriate assumptions.

Christodorescu and Jha used model-checking techniques to detect malicious-code variants [7]. Given a sample of malicious code, they extract a parameterized state machine that will accept variants of the code. They use CodeSurfer/x86 (with VSA turned off) to extract a model of each program procedure, and determine potential matches between the program's code and fragments of the malicious code. Their technique is intraprocedural, and does not analyze data state.

Other groups have used run-time program monitoring and checkpointing to perform a systematic search of a program's dynamic state space [16, 23, 25]. Like our approach, this allows for model checking properties of the low-level code that is actually run on the machine. However, because the dynamic state space can be unbounded, these approaches cannot perform an exhaustive search. In contrast, we use static analysis to perform a (conservative) exhaustive search of an abstract state space.

[3] We assume that source code is not available, but the techniques extend naturally if it is: one can treat the executable code as just another IR in the collection of IRs obtainable from source code. The mapping of information back to the source code would be similar to what C source-code tools already have to perform because of the use of the C preprocessor.

References

1. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. on Parallel Processing*, 2000.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
3. J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*, 2001.
4. J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, pages 184–189, 1999.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.*, pages 62–73, 2003.
6. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Dist. Syst. Security*, 2004.
7. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security Symposium*,, 2003.
8. C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, Univ. Queensland, 1997.
9. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
10. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Int. Conf. on Softw. Maint.*, pages 228–237, 1998.
11. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The M.I.T. Press, 1999.
12. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
13. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
14. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Softw. Eng.*, 1999.
15. Fast library identification and recognition technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
16. P. Godefroid. Model checking for programming languages using VeriSoft. In ACM, editor, *Princ. of Prog. Lang.*, pages 174–186. ACM Press, 1997.
17. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
18. M. Howard. Some bad news and some good news, October 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp.
19. IDAPro disassembler, http://www.datarescue.com/idabase/.
20. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems. http://www.cs.wisc.edu/wpis/wpds++/.
21. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
22. J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Prog. Lang. Design and Impl.*, pages 291–300, 1995.
23. P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Spin Workshop*, 2004.
24. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
25. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Op. Syst. Design and Impl.*, 2002.
26. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, 2003.
27. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.* To appear.
28. X. Rival. Abstract interpretation based certification of assembly code. In *Int. Conf. on Verif., Model Checking, and Abs. Int.*, 2003.
29. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.
30. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

Empowering Software Debugging Through Architectural Support for Program Rollback

Radu Teodorescu and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

This paper proposes the use of processor support for program rollback, as a key primitive to enhance software debugging in production-run environments. We discuss how hardware support for program rollback can be used to characterize bugs on-the-fly, leverage code versioning for performance or reliability, sandbox device drivers, collect monitoring information with very low overhead, support failure-oblivious computing, and perform fault injection. To test our ideas, we built an FPGA prototype. We run several buggy applications on top of a version of Linux.

1. Introduction

Dynamic bug-detection tools (e.g., [9, 13]) face major challenges when targeting *production-run environments*. In such environments, bug monitoring and detection have to be done with very low overhead. In addition, it is often desirable to provide graceful recovery from bugs, so that the system can continue to work.

One way to accomplish these goals is to provide hardware support in the processor for low-overhead software bug characterization, and for graceful recovery from bugs. For this, we propose a hardware primitive that quickly undoes (rolls back) sections of code. When a certain suspicious event that may be a bug has been detected, the hardware rolls the program thousands of instructions back with very little overhead. At that point, several options are possible. We can either choose to re-execute the same section of code or to jump off to another section where additional monitoring can be done. If we choose the former, we can re-execute the code section with the same input data set but with more instrumentation enabled, so that we can further characterize the bug. Alternatively, we can re-execute the section with a different input or algorithm, to skip the bug altogether.

To test these ideas, we have implemented such a hardware extension to a simple processor prototyped using FPGAs (Field Programmable Gate Arrays). In this paper, we describe the operation and software interface of our prototype. In addition,, we describe some of the uses that such hardware support can have in helping software debugging. Such uses are to fully characterize a bug on-the-fly, leverage code versioning, sandbox the kernel's device drivers, collect and sample information with very low overhead, support failure-oblivious computing, and perform fault injection, among other issues.

This paper also evaluates the FPGA-based prototype we built [16]. The extensions added include holding speculative data in the cache, register checkpointing, and software-controlled transitions between speculative and non-speculative execution. We experiment with several buggy applications running on top of a version of Linux. Overall, we show that this rollback primitive can be very effective in production-run environments.

2 System overview

The system we propose allows the rollback and re-execution of large sections of code (typically up to tens of thousands of instructions) with very low overhead. This is achieved through a few relatively simple changes to an existing processor.

We have implemented two main extensions: (1) the cache can hold speculative data and, on demand, quickly commit it or discard it all, and (2) the register state can be quickly checkpointed into a special storage and restored from there on demand. These two operations are done in hardware. When entering speculative execution, the hardware checkpoints the registers and the cache starts buffering speculatively written data. During speculative execution, speculative data in the cache gets marked as such and is not allowed to be displaced from the cache. When transitioning back to normal execution, any mark of speculative

data is deleted and the register checkpoint is discarded. If a rollback is necessary, the speculatively written data is invalidated and the register state is restored from the checkpoint.

2.1 Speculative Execution Control

The speculative execution can be controlled either in hardware or in software. There are benefits on both sides and deciding which is best is dependent on what speculation is used for.

2.1.1 Hardware Control

If we want the system to always execute code speculatively and be able to guarantee a minimum rollback window, the hardware control is more appropriate. As the program runs, the cache buffers the data generated and always marks them as speculative. There are always two *epochs* of speculative data buffered in the cache at a time, each one with a corresponding register checkpoint. When the cache is about to get full, the earliest epoch is committed, and a new checkpoint is created. With this support, the program can always roll back at least one of the two execution epochs (a number of instructions that filled roughly half of the L1 data cache).

2.1.2 Software Control

If, on the other hand, we need to execute speculatively only some sections of code, and the compiler or user is able to identify these sections, it is best to expose the speculation control to the software. This approach has two main benefits: more flexibility is given to the compiler and a smaller overhead is incurred since only parts of the code execute speculatively.

In this approach, the software explicitly marks the beginning and the end of the speculative section with *BEGIN_SPEC* and *END_SPEC* instructions. When a *BEGIN_SPEC* instruction executes, the hardware checkpoints the register state and the cache starts buffering data written to the cache, marking them as speculative.

If, while executing speculatively, a suspicious event that may be a bug is detected, the software can set a special *Rollback* register. Later, when *END_SPEC* is encountered, two cases are possible. If the Rollback register is clear, the cache commits the speculative data, and the hardware returns to the normal mode of execution. If, instead, the Rollback register is set, the program execution is rolled back to the checkpoint, and the code is re-executed, possibly with more instrumentation or different parameters.

If the cache runs out of space before the *END_SPEC* instruction is encountered, or the processor attempts to perform an uncacheable operation (such as an I/O access), the processor triggers an exception. The exception handler decides what to do, one possibility being to commit the current speculative data and continue executing normally.

3 Using Program Rollback for Software Debugging

The architectural support presented in this work provides a flexible environment for software debugging and system reliability. In this section, we list some its possible uses.

3.1 An Integrated Debugging System

The system described here is part of a larger debugging effort for production-run codes that includes architectural and compiler support for bug detection and characterization. In this system, program sections execute in one of three states: *normal*, *speculative* or *re-execute*. While running in speculative mode, the hardware guarantees that the code (typically up to about tens of thousands of instructions) can be rolled back with very low overhead and re-executed. This is used for thorough characterization of code sections that are suspected to be buggy.

The compiler [6] is responsible for selecting which sections of code are more error-prone and thus, should be executed in speculative mode. Potential candidates are functions that deal with user input, code with heavy pointer arithmetic, or newer, less tested functions. The programmer can also assist by indicating the functions that he or she considers less reliable.

In addition, a mechanism is needed to detect potential problems, and can be used as a starting point in the bug detection process. Such a mechanism can take many forms, from a simple crash detection system to more sophisticated anomaly detection mechanisms. Examples of the latter include software approaches like Artemis [3] or hardware approaches like iWatcher [19].

Artemis is a lightweight run-time monitoring system that uses execution contexts (values of variables and function parameters) to detect anomalous behavior. It uses training data to learn the normal behavior of an application and will detect unusual situations. These situations can act as a trigger for program rollbacks and re-executions for code characterization.

iWatcher is an architecture proposed for dynamically monitoring memory locations. The main idea of iWatcher is to associate programmer-specified monitoring functions with monitored memory objects. When a monitored object is accessed, the monitoring function associated with this object is automatically triggered and executed by the hardware without generating an exception to the operating system. The monitoring function can be used to detect a wide range of memory bugs that are otherwise difficult to catch.

We provide two functions, namely `enter_spec` to begin speculative execution and `exit_spec` to end it with commit or rollback. In addition, we have a function `proc_state()` used to probe the state of the processor. A return value of 0 means normal mode, 1 means speculative mode, and 2 means re-execute mode (which follows a rollback).

The following code shows how these functions are used. `exit_spec` takes one argument, `flag`, that controls whether speculation ends with commit or rollback. If an anomaly is detected, the software immediately sets the `flag` variable. When the execution finally reaches `exit_spec`, a rollback is triggered. The execution resumes from the `enter_spec` point.

```
num=1;
...
/* begin speculation */
enter_spec();
...
/* heavy pointer arithmetic */
p=m[a[*x]]+&y;
if (err) flag=1;
...
/* info collection */
/* only in re-execute mode */
if (proc_state()==REEXECUTE) {
 collect_info();
}
exit_spec(flag);
/* end speculation */
num++;
...
```

The compiler inserts code in the speculative section to collect relevant information about the program execution that can help characterize a potential bug. This code is only executed if the processor is in re-execute mode (proc_state() returns 2).

Figure 1 shows the three possible execution scenarios for the example given above. Case (a) represents normal execution: no error is found, the `flag` variable remains clear and, when `exit_spec(flag)` is reached, speculation ends with commit.

In case (b), an abnormal behavior that can lead to a bug is encountered. `Flag` is set when the anomaly is detected and, later, when execution reaches `exit_spec(flag)`, the program rolls back to the beginning of the speculative region and continues in re-execute mode. This can be repeated, possibly even inside a debugger, until the bug is fully characterized. `Flag` can be set as a result of a failed assertion or data integrity test.

Finally, in case (c) the speculative state can no longer fit in the cache. The overflow is detected by the cache controller and an exception is raised. The software is expected to handle this case. The example assumes that the exception handler commits the speculative data. When the execution reaches the `exit_spec(flag)` instruction, the state

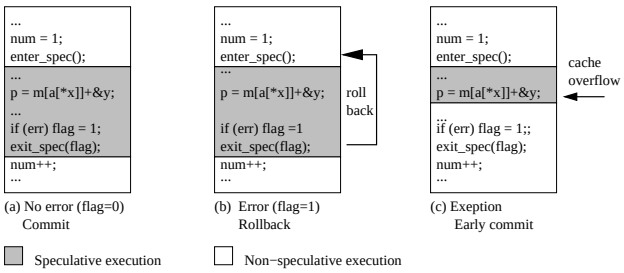


Figure 1. Speculative execution ends with commit (a), a rollback (b), or an early commit due to cache overflow (c).

of the processor is first checked. Since the processor is no longer speculative (due to the early commit), the instruction is simply ignored.

3.2 Other Uses of Program Rollback

3.2.1 Code Versioning

Code versioning, or N-version programming [8] is a technique that involves generating multiple, different versions of the same code. It can be used for performance or reliability. When targeting performance, a compiler generates a main version that is aggressively optimized, and potentially sometimes incorrect. Using our hardware, this version can be executed speculatively, with some verification code in place. If the function fails or produces an incorrect result as indicated by the verification code, the processor is rolled back, and a second, unoptimized but safe version of the code is executed.

In the same way, when targeting reliability, we can have two versions of the same function that are safe, have similar performance, but use different functional units in the processor. Each version includes some verification code that checks that the computation was correct. We can first run the first function and its verification code. If the verification code fails, we then run the second function and its verification code. Since the functions use different parts of the processor, they are unlikely to both fail.

3.2.2 OS Kernel and Driver Debugging

One of the major challenges in OS reliability is to ensure correct execution of the OS kernel in the presence of faulty drivers. In fact, in Linux, the frequency of coding errors is seven times higher for device drivers than for the rest of the kernel [1]. Several solutions have been proposed to this problem including many that involve isolating the kernel from the device drivers with some protection layer [15].

In general, these solutions require major changes to OS design and implementation and can introduce significant overheads.

We propose a simpler solution with potentially very low overhead that takes advantage of the rollback support implemented in the hardware.

In general, the kernel and driver code interact through interface functions, and maintain data structures in both kernel and driver memory. In a system like ours, function calls from kernel to driver or vice-versa could be executed speculatively. If an error is detected, the changes made to kernel memory would then be rolled back. The idea is to prevent the kernel from becoming corrupted or even crashing due to a faulty driver. A cleanup procedure could then be called to shut down the driver and either attempt to reinitialize it or report the error to the user.

The current system cannot roll back any I/O operations. This is because we currently buffer only cacheable data. However, we can still roll back the processor in case of a fault. Any communication with the faulty device is lost but the processor is restored to the state before the device access began. If the device somehow corrupted the kernel, the correct state can still be recovered from the checkpoint. The fault model for a system like this would target kernel integrity rather than guaranteeing the correct operation of individual devices.

3.2.3 Lightweight Information Collection and Sampling

Detecting bugs in production code can be challenging because it is hard to obtain substantial information about program execution. It is hard to collect relevant information without incurring a large overhead. Previous solutions to this problem have suggested using statistical sampling to obtain execution information with small overheads [7].

We propose using our system to perform lightweight collection of execution information based on anomaly detection. In this case, the processor would always execute in speculative state. When an anomaly is detected (an unusual return value, a rarely executed path, etc.), the processor is rolled back as far as its speculative window allows and then re-executed. Upon re-execution, instrumentation present in the code is turned on, and the path that led to the anomalous execution recorded. This allows more precise information about anomalous program behavior than statistical sampling would. Also, because the additional code is rarely executed, the overhead should be very low.

3.2.4 Failure-Oblivious Computing

A failure-oblivious system [12] enables programs to continue executing through memory errors. Invalid memory accesses are detected, but, instead of terminating the execution or raising an exception, the program discards the invalid writes and manufactures values for invalid reads, enabling the program to continue execution.

A failure-oblivious system can greatly benefit from our rollback support. When a read results in an invalid access, the system enters speculative mode, generates a fake value, and uses it in order to continue execution. It is unknown however, whether the new value can be used successfully or, instead, will cause further errors. Since the code that uses the fake value executes speculatively, it can roll back if a new error is detected. Then, the program can use a different predicted value and re-execute the code again, or finally raise an exception.

3.2.5 Fault Injection

Our rollback hardware can also be used as a platform for performing fault injection in *production systems*. It offers a way of testing the resilience of systems to faulty code, or test *what if* conditions, without causing system crashes. The code that is injected with faults is executed speculatively, to determine what effect it has on the overall system. Even if the fault propagates, the code can be rolled back and the system not allowed to crash. The process can be repeated multiple times, with low overhead, to determine how a system behaves in the presence of a wide array of faults.

4 Evaluation

4.1 FPGA infrastructure

As a platform for our experiments, we used a synthesizable VHDL implementation of a 32-bit processor [4] compliant with the SPARC V8 architecture.

The processor has an in-order, single-issue, five stage pipeline. This system is part of a system-on-a-chip infrastructure that includes a synthesizable SDRAM controller, PCI and Ethernet interfaces. The system was synthesized using Xilinx ISE v6.1.03. The target FPGA chip is a Xilinx Virtex II XC2V3000 running on a GR-PCI-XC2V development board [10].

On top of the hardware, we run a version of the SnapGear Embedded Linux distribution [2]. SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. A cross-compilation tool-chain for the SPARC architecture is used for the compilation of the kernel and Linux applications.

To get a sense of the hardware overhead imposed by our scheme, we synthesize the processor core with and without the support for speculative execution. We look at the

utilization of the main resources in FPGA chips, the Configurable Logic Blocks (CLBs). Virtex II CLBs are organized in an array and are used to build the combinatorial and synchronous logic components of the design. The CLB overhead of our scheme is small (less than 4.5% on average) [16].

4.2 Speculative execution of buggy applications

We run experiments, using standard Linux applications that have known (reported) bugs. For these applications, we want to determine whether we can speculatively execute a section of dynamic instructions that is large enough to contain *both the bug and the location where the bug is caught* by a detection mechanism like iWatcher [19]. Some parameters of the experimental setup are given in Table 1.

We assume that the compiler has identified the suspicious region of code that should be executed speculatively. We also assume the existence of a detection mechanism (such as iWatcher), which can tell us that a bug has occurred. We want to determine if, under these circumstances, we can roll back the buggy section of code in order to characterize the bug thoroughly by enabling additional instrumentation.

We use five buggy programs from the open-source community. The bugs were introduced by the original programmers. They represent a broad spectrum of memory-related bugs. The programs are: *gzip*, *man*, *polymorph*, *ncompress and tar*. *Gzip* is the popular compression utility, *man* is a utility used to format and display on-line manual pages, *polymorph* is a tool used to convert Windows style file names to something more portable for UNIX systems, *ncompress* is a compression and decompression utility, and *tar* is a tool to create and manipulate archives.

In the tests we use the bug-exhibiting inputs to generate the abnormal runs. All the experiments are done under realistic conditions, with the applications running on top of a version of Linux running on our hardware.

Table 1. Main parameters of the experimental setup.

Processor	LEON2, SPARC V8 compliant
Clock frequency	40MHz
Instruction cache	8KB
Data cache	32KB
RAM	64MB
Windowed register file	8 windows $\times$ 24 registers each
Global registers	8 registers

Table 2 shows that the buggy sections were successfully rolled back in most cases, as shown in column four. This means that the system executed speculatively the entire buggy section, performed a rollback when the end speculation instruction was reached, and then re-executed the entire section. On the other hand, a failed rollback (*polymorph*) means that before reaching the end speculation instruction, a condition is encountered that forces the early commit of the speculative section. Rollback is no longer possible in this case.

The fifth column shows the number of dynamic instructions that were executed speculatively. Notice that in the case of *polymorph* the large number of dynamic instructions causes the cache to overflow the speculative data, and forces an early commit.

5 Related work

Some of the hardware presented in this work builds on extensive work on Thread-Level Speculation (TLS) (e.g. [5, 14]). We employ some of the techniques first proposed for TLS to provide lightweight rollback and replay capabilities. TLS hardware has also been proposed as a mechanism to detect data races on-the-fly [11].

Previous work has also focused on various methods for collecting information about bugs. The "Flight Data Recorder" [17] enables off-line deterministic replay of applications and can be used for postmortem analysis of a bug. It has a significant overhead that could prevent its use in production codes.

There is other extensive work in the field of dynamic execution monitoring. Well-known examples include tools like Eraser [13] or Valgrind [9]. Eraser targets detection of data races in multi-threaded programs. Valgrind is a dynamic checker to detect general memory-related bugs such as memory leaks, memory corruption and buffer overflow. Most of these systems have overheads that are too large to make them acceptable in production code.

There have also been proposals for hardware support for detecting bugs, such as iWatcher [19] and AccMon [18]. These systems offer dynamic monitoring and bug detection capabilities that are sufficiently lightweight to allow their use on production software. This work is mostly complementary to ours. In fact we assume some of the detection capabilities of iWatcher when evaluating our system.

6 Conclusions and future work

This work shows that with relatively simple hardware we can provide powerful support for debugging production codes. We show it by building a hardware prototype of the envisioned system, using FPGA technology. Finally, we run experiments on top of Linux running on this system.

The hardware presented in this work is part of a comprehensive debugging infrastructure. We are working toward

Application	Bug location	Bug description	Successful rollback	Speculative instructions
ncompress-4.2.4	compress42.c: line 886	Input file name longer than 1024 bytes corrupts stack return address	Yes	10653
polymorph-0.4.0	polymorph.c: lines 193 and 200	Input file name longer than 2048 bytes corrupts stack return address	No	103838
tar-1.13.25	prepargs.c: line 92	Unexpected loop bounds causes heap object overflow	Yes	193
man-1.5h1	man.c: line 998	Wrong bounds checking causes static object corruption	Yes	54217
gzip-1.2.4	gzip.c: line 1009	Input file name longer than 1024 bytes overflows a global variable	Yes	17535

Table 2. Speculative execution in the presence of bugs.

integrating compiler support to identify vulnerable code regions as well as to instrument the code with speculation control instructions.

We have presented several uses of this hardware for debugging, including to characterize bugs on-the-fly, leverage code versioning for performance or reliability, sandbox device drivers, collect monitoring information with very low overhead, support failure-oblivious computing, and perform fault injection. We will be implementing some of these techniques in the near future.

References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. *SIGOPS Operating Systems Review*, 35(5):73–88, 2001.

[2] CyberGuard. SnapGear Embedded Linux Distribution. www.snapgear.org.

[3] L. Fei and S. P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Errors. Technical Report TR-ECE05-02, School of ECE, Purdue University, 2005.

[4] J. Gaisler. LEON2 Processor. www.gaisler.com.

[5] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, pages 866–880, September 1999.

[6] S. I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[7] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[8] M. Lyu and Y. He. Improving the N-Version Programming Process Through the Evolution of a Design Paradigm. *Proceedings of IEEE Transactions on Reliability*, 1993.

[9] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Lecture Notes in Theoretical Computer Science*, 89(2), 2003.

[10] R. Pender. Pender Electonic Design. www.pender.ch.

[11] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–121. ACM Press, 2003.

[12] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, pages 303–316, 2004.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[14] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *International Symposium on Computer Architecture (ISCA)*, pages 1–24, 2000.

[15] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[16] R. Teodorescu and J. Torrellas. Prototyping Architectural Support for Program Rollback Using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2005.

[17] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture (ISCA)*, pages 122–135. ACM Press, 2003.

[18] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 269–280. IEEE Computer Society, 2004.

[19] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 224–237, June 2004.

EXPLODE: A Lightweight, General Approach to Finding Serious Errors in Storage Systems

Junfeng Yang, Paul Twohey, Ben Pfaff, Can Sar, Dawson Engler *
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{junfeng,twohey,blp,csar,engler}@cs.stanford.edu

ABSTRACT

File systems, RAID systems, and applications that require data consistency, among others, assure data integrity by carefully forcing valuable data to stable storage. Unfortunately, verifying that a system can recover from a crash to a valid state at any program counter is very difficult. Previous techniques for finding data integrity bugs have been heavyweight, requiring extensive effort for each OS and file system to be checked. We demonstrate a lightweight, flexible, easy-to-apply technique by developing a tool called EXPLODE and show how we used it to find 25 serious bugs in eight Linux file systems, Linux software RAID 5, Linux NFS, and three version control systems.

1. INTRODUCTION

Many systems prevent the loss of valuable data by carefully forcing it to stable storage. Applications such as version control and mail handling systems ensure data integrity via file system synchronization services. The file system in turn uses synchronous writes, journaling, etc. to assure integrity. At the block device layer, RAID systems use redundancy to survive disk failure. At the application layer, software based on these systems is often trusted with the only copy of data, making data loss irrevocable and arbitrarily serious. Unfortunately, verifying that a system can recover from a crash to a valid state at any program counter is very difficult.

Our goal is to comprehensively test real systems for data persistence errors, adapting ideas from model checking. Traditional model checking [5] requires that the implementor rewrite the system in an artificial modeling language. A later technique, called implementation-level model checking, eliminates this requirement [19, 18, 21] by checking code directly. It is tailored to effectively find errors in system code, not verify correctness. To achieve this goal, it aggressively deploys unsound state abstractions to trade completeness for effectiveness. One major disadvantage of this technique is that it cannot check software without source code and requires porting the entire OS to run on top of a model checker, which necessitates many intrusive, non-portable modifications. Checking a new OS or a different version of the same OS requires a new port. Even checking new file systems often requires about a week of effort.

This paper describes our improved approach that remedies these problems. We reduce the infrastructure needed for checking a system to a single device driver, which can be run inside of a stock kernel that runs on real hardware. This lightweight approach makes it easy to check new file systems (and other storage systems): simply mount and run. Checking a new OS is just a matter of implementing a device driver.

Our approach is also very general in that it rarely limits the types of checks that can be done: if you can run a program, you can check it. We used EXPLODE to check CVS and Subversion, both open source version control systems, and BitKeeper, a commercial version control system, finding bugs in all three. At the network layer, we checked the Linux NFS client and server. At the file system layer, we checked 8 different Linux file systems. Finally, at the block device layer, we checked the Linux RAID 5 implementation. EXPLODE can find errors even in programs for which we do not have the source, as we did with BitKeeper.

EXPLODE can check almost all of the myriad ways the storage layers can be stacked on one another, as shown on the left side of Figure 1. This ability has three benefits. First, EXPLODE can reuse consistency checks for one specific layer to check all the layers below it. For example, once we implement a consistency check for a file system on top of a single disk, we can easily plug in a RAID layer and check that RAID does not compromise this guarantee. Second, testing entire stacks facilitates end-to-end checking. Data consistency is essentially end-to-end: in a multi-layer system, if one layer is broken, the entire system is broken. Simply verifying that a single layer is correct may have little practical value for end-to-end consistency, and it may require a huge amount of manual effort to separate this layer from the system and build a test harness for it. Third, we can *cross-check* different implementations of one layer and localize errors. For example, if a bug occurs with an application on top of 7 out of 8 file systems, the application is probably buggy, but if it occurs on only one file system, that file system is probably buggy.

The contributions of this paper can be summarized as follows:

*This research was supported by NSF grant CCR-0326227 and 0121481, DARPA grant F29601-03-2-0117, an NSF Career award and Stanford Graduate Fellowship.

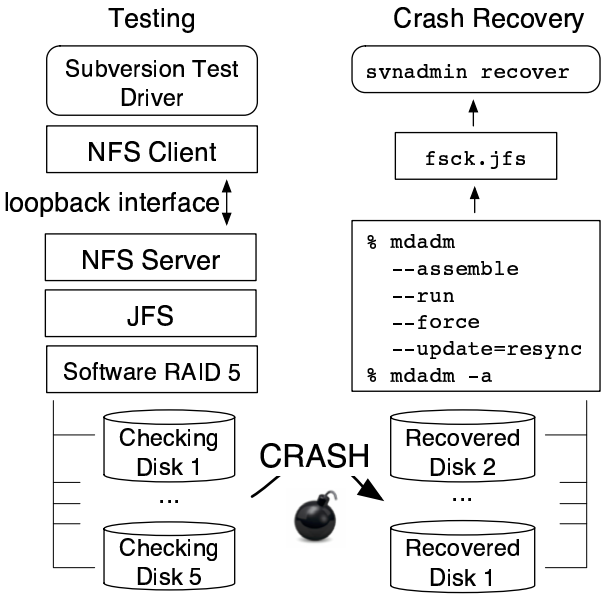


Figure 1: A snapshot of the EXPLODE system with a stack of storage systems being tested on the left and the recovery tools being run on the right after EXPLODE "crashes" the system to generate a recovery scenario.

- A lightweight, minimally invasive approach for checking storage systems.
- Model checking every layer in a complex storage stack from RAID at the bottom to version control systems running over NFS at the top.
- A series of new file system specific checks for catching bugs in the data synchronization facilities used by applications to ensure data integrity.

This paper is organized as follows. Section 2 gives an overview of the checking system. Section 3 discusses the new challenges for our approach. Section 4 explains the basic template work needed to check a given subsystem, and how to apply this template to different storage systems. Finally, Section 5 discusses related work and Section 6 concludes.

2. SYSTEM OVERVIEW

EXPLODE has three parts: a user-land model checking library (MCL), a specialized RAM disk driver (RDD), and a test driver. We briefly describe each of them, highlighting their most important functions and interactions. Figure 2 shows an overview.

MCL is similar in design to our prior implementation-level model checking work [19, 18, 21]. It provides a key mechanism to enumerate all possible choices at a given "choice point," which is a program point where abstractly a program can do multiple actions. For example, dynamic memory allocation can either succeed or fail. When such an allocator is properly instrumented, EXPLODE can explore both branches.

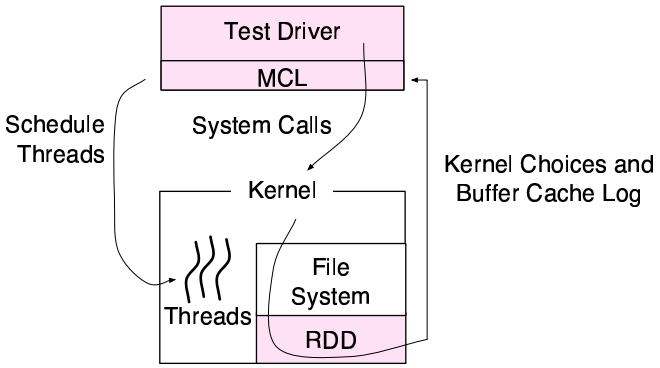


Figure 2: EXPLODE overview with a local file file system as an example storage system. Components of EXPLODE are shaded.

RDD provides the needed infrastructure so EXPLODE can explore all possible behaviors for a *running* Linux kernel, using the "choice point" mechanism provided by MCL. It exports a log of buffer cache operations, allows EXPLODE to schedule a kernel thread to run, and provides a generic interface to install and remove "choice points" inside a running kernel. We defer the discussion of why we need this functionality to §3.2.

The test driver exercises storage system operations on our specialized RAM disks to trigger potential disk writes. It also performs system-specific consistency checks once EXPLODE is done simulating crash-recovery. To simulate all possible crashes, EXPLODE clones current disks, applies all possible subsets of the potential disk writes[1] to the disk clones, and invokes system-specific utilities to recover them, as noted previously. Figure 1 shows how EXPLODE "crashes" and recovers a running stack of a storage system.

3. IMPLEMENTATION

As mentioned in the introduction, our current approach is much more lightweight and general than our previous approach. However, this approach also poses new challenges, discussed in the following sections.

3.1 State Checkpoint and Restore

One challenge in checking live storage systems is how to checkpoint and restore storage system states. State checkpointing and restoring is essential for exploring all possible behaviors of a system. Our previous work [21] runs an entire kernel on top of a model checker, which makes checkpointing and restoring states as easy as copying the entire kernel memory and simulated disks. Since extracting relevant kernel data from a live system is difficult, we no longer checkpoint the kernel memory. Instead, to checkpoint a file system state, we store the sequence of choices that led to the current state, starting from the initial pristine disk. To

[1]In practice, if the set of potential writes grows larger than a user specified limit, EXPLODE no longer tests all subsets but randomly tests a few.

restore a state, we unmount the current disk, then mount a copy of the initial pristine disk and replay all the previously made choices. Storage systems have the convenient property that unmounting a file system clears its in-memory state, and replaying the same set of operations on a pristine storage system is deterministic in terms of the content contained in the storage system. MCL also allows users to selectively checkpoint non-pristine disks, given that all the dirty blocks are flushed to disk.

This method for state checkpointing and restoration is identical to the state compression technique used in traditional model checking where each state is represented by a trace starting from the initial state to the current state. Unlike traditional model checking where state compression is simply a time/space tradeoff, it is essential to EXPLODE because it is the only practical method for a live kernel.

3.2 Exploring All Kernel Behaviors

With MCL we can explore the choice points we can see. However, kernel choice points are often not exposed to user-land. RDD provides three key mechanisms to expose these kernel choice points to user-land, so EXPLODE can explore all possible kernel behaviors.

First, RDD monitors all buffer cache operations and stores them in a temporal log. EXPLODE retrieves this log using an `ioctl` command provided by RDD and replays it to reconstruct the set of all possible disk images that could result from a crash. We use temporal logging instead of getting the set of dirty buffers directly from the kernel because Linux 2.6, the OS we tested, has a complicated unified buffer and page cache that makes the latter very difficult.

Second, our driver provides a general interface for installing and removing "choice points" within the kernel. Practically, EXPLODE uses this mechanism to induce artificial errors at critical places in the kernel. RDD provides a new `ioctl` that allows a process to request a failure for the nth kernel choice point in its next system call. Using this mechanism for each system call, EXPLODE fails the first choice point, then the second, and so on until all the choice points have been tested in turn. We could use this mechanism to fail more than one choice point at a time, but in our experience kernel developers are not interested in bugs resulting from multiple, simultaneous failures.

Third, our driver allows EXPLODE to schedule any kernel thread to run. Storage systems often have background threads to commit disk I/O. Controlling such threads allows EXPLODE to explore more system behaviors. As an added benefit, this makes our error traces deterministic. Without this feature EXPLODE would still find errors but it would be unable to present traces to the user which would reliably trigger the bug when replayed. Our driver implements this function by simply setting a thread to have a very high priority. Although in theory this method is not guaranteed to run the thread in all cases, in practice it works reliably.

4. CHECKING STORAGE SYSTEMS

EXPLODE can check almost any storage system that runs on Linux, be it a file system, a RAID system, a network file system, a user-land application that handles data, or any combination thereof.

Checking a new storage system at the top of the stack is often a matter of providing EXPLODE utilities to set up, tear down and recover the storage system, and writing a test driver to mutate the storage system and check its consistency. One nice feature is that one test driver can exercise every storage system below it in the storage hierarchy.

FS	`mount sync`	`sync`	`fsync`	`O_SYNC`
ext2	✘	✓	✘	✘
ext3	✓	✓	✓	✓
ReiserFS	✘	✓	✘	✘
JFS	✘	✓	✘	✘
MSDOS	✘	✘	n/a	n/a
VFAT	✘	✘	n/a	n/a
HFS+	✘	✘	✘	?
XFS	✘	✓	✓	?

Table 1: Sync checking results. ✓: no errors found; ✘: one or more errors found; n/a: could not complete test; ?: not run due to lack of time.

This section discusses how we checked different storage systems in detail. For each of them, we first list its setup, tear-down, and recovery utilities, then describe how the test driver mutates the storage system and what consistency checks are performed. Lastly, we show the errors we found.

4.1 Checking File Systems

To set up a file system, EXPLODE needs to create a new file system (`mkfs`) and mount it. To tear it down, EXPLODE simply unmounts it. EXPLODE uses `fsck` to repair the FS after a crash.

EXPLODE's FS test driver enumerates topologies containing less than a user-specified number of files and directories. At each step the test driver either modifies the file system topology, by creating, deleting, or moving a file or directory, or a file's contents, by writing in or truncating a file. To avoid being swamped by many different file contents, the FS test driver only writes out 5 different possible file contents chosen to require varying numbers of indirect and doubly indirect blocks.

To avoid wasting time re-checking similar topologies, we memoize file systems that have isomorphic directory structure and identical file contents, disregarding file names and most metadata.

To check FS-specific crash recovery, FS developers need to provide EXPLODE with a model of how the file system should look after crash recovery. Following the terminology in [21], we call this model the StableFS. It describes what has been committed to stable storage. We call the user-visible, in-memory state of the file system the VolatileFS, because it has not necessarily been committed to stable storage.

Without an FS-specific StableFS, EXPLODE checks that the four basic sync services available to applications that care about consistency honor their guarantees. These services are: synchronous mount, which guarantees that after an FS operation returns, the StableFS and the VolatileFS are identical; `sync`, which guarantees that after `sync` returns, the StableFS and the VolatileFS are identical; `fsync`, which guarantees that the data and metadata of a given file are identical in both the StableFS and VolatileFS; and the `O_SYNC` flag for `open`, which guarantees that the data of the opened file is identical in both the StableFS and VolatileFS.

Table 1 summarizes our results. Surprisingly, 7 of the 8 file systems we tested have synchronous mount bugs – ext3 being the only tested file system with synchronous mount correctly implemented. Most file systems implement `sync`

correctly, except MSDOS, VFAT and HFS+. Crash and `fsck` on MSDOS and VFAT causes these file systems to contain directory loops, which prevents us from checking `fsync` and `O_SYNC` on them. Intuitively, `fsync` is more complicated than `sync` because it requires the FS to carefully flush out only the data and metadata of a particular file. Our results confirm this intuition, as the `fsync` test fails on three widely used file systems: ext2, ReiserFS and JFS. The JFS `fsync` bug is quite interesting. It can only be triggered when several `mkdir` and `rmdir` operations are followed by creating and `fsync`ing a file and its enclosing directories. After a crash this causes the file data to be missing. JFS developer Dave Kleikamp quickly confirmed the bug and provided a fix. The problem resided in the journal-replay code, triggered by the reuse of a directory inode by a regular file inode, so that reproducing the bug requires the journal to contain changes to the inode both as a directory and as a file.

Note that when we say a service is "correct" we simply mean EXPLODE did not find an error before we stopped it or all the possible topologies for a given number of file system objects were enumerated. We found most bugs within minutes of starting our test runs, although some took tens of minutes to discover.

4.2 Checking RAID

We tested the Linux software implementation of RAID 5 along with its administration utility `mdadm`. To set up a test run, we assembled several of our special RAM disks into a RAID array. To tear down, we disabled the RAID array. Crash recovery for RAID was not complex: we simply `fsck` the file system running on top of RAID. To recover from a disk failure, we used the `mdadm` command to replace failed disks. Read failures in individual disks in the RAID array were simulated using the kernel choice point mechanism discussed in §3.2.

We reused our file system checker on top of the RAID block device. The consistency check we performed was that the loss of any one disk in a RAID 5 array should not lead to data loss—the disk's contents can always be reconstructed by computing the exclusive-or of the $n-1$ remaining disks [20].

EXPLODE found that the Linux RAID implementation does not reconstruct bad sectors when a read error occurs. Instead, it simply marks the disk faulty, removes it from the RAID array, and returns an I/O error. EXPLODE also found that when two sector read errors happen on different disks, requiring manual maintenance, almost all maintenance operations fail. Disk write requests also fail in this case, rendering the RAID array unusable until the machine is rebooted. Software RAID developer Neil Brown confirmed that the above behaviors were undesirable and should be fixed with high priority.

4.3 Checking NFS

We used EXPLODE to check Linux's Network File System version 3 (NFSv3) and its in-kernel NFS server. To set up an NFS partition, we export a local FS as an NFS partition over the loopback interface. To tear it down, we simply unmount it. We use the `fsck` for the local file system to repair crashed NFS partitions. Currently we do not model network failures.

As NFS is a file system built on top of other file systems we are able to leverage our existing FS test driver to test the Linux NFS implementation. We can also reuse our consistency checker for synchronously mounted file systems as NFS should have identical crash recovery guarantees [4]

We found one inconsistency in NFS, in which writing to a file, then reading the same file through a hard link in a different directory yields inconsistent data. This was due to a Linux NFS security feature called "subtree checking" that adds the inode number of the file's containing directory to the file handle. Because the two links are in different directories, their file handles differ, causing the client to cache their data separately. This bug was not known to us until EXPLODE found it, but the NFS developers pointed us to a manpage describing subtree checking. The manpage said that the client had to rename a file to trigger it, but the checker did not do so. The NFS developers then clarified that the manpage was inaccurate. It described what could be done, not what Linux NFS actually implemented (because it was too difficult).

We found additional data integrity bugs in specific file systems exported as NFS, including JFS and ext2.

4.4 Checking Version Control

We tested the popular version control systems CVS, Subversion, and BitKeeper. We found serious errors that can cause committed data to be permanently lost in all three. Our test driver checks out a repository, does a local modification, commits the changes, and simulates a crash on the block device that stores the repository. It then runs `fsck` and the version control system's recovery tool (if any), and checks the resulting repository for consistency.

The errors in both CVS and BitKeeper are similar: neither attempts to sync important version control files to disk, meaning an unfortunate crash can permanently lose committed data. On the other hand, Subversion carefully `fsync`s important files in its database directory, but forgets to sync others that are equally important. (Adding a `sync` call fixes the problem for all three systems.)

If our system was not modular enough to allow us to run application checkers on top of arbitrary file systems we would have missed bugs in both BitKeeper and Subversion. On ext3 `fsync` causes all prior operations to be committed to the journal, and by default also guarantees that data blocks are flushed to disk prior to their associated metadata, hiding bugs inside applications. This ability to *cross-check* different file systems is one of the key strengths in our system.

To demonstrate that EXPLODE works fine with software to which we do not have source, we further checked BitKeeper's atomic repository syncing operations "push" and "pull." These operations should be atomic: either the merge happens as a whole or not at all. However, EXPLODE found traces where a crash during a "pull" can badly corrupt the repository in ways that its recovery tool cannot fix.

5. RELATED WORK

In this section, we compare our approach to file system testing techniques to software model checking efforts and other generic bug finding approaches.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. These testing frameworks are non-deterministic and less comprehensive than our approach, but they are more lightweight and work "out of the box." We view stress testing as com-

plementary to our approach — there is no reason not to both test a file system and then test with EXPLODE (or vice versa). In fact, our approach can be viewed as deterministic, comprehensive testing.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [16, 15, 17, 2, 19, 18, 21, 6, 1]. Verisoft [15] is a software model checker that systematically explores the interleavings of a concurrent C program. Unlike the technique we use, Verisoft does not store states at checkpoints and thereby can potentially explore a state more than once. Verisoft relies heavily on partial order reduction techniques that identify (control and data) independent transitions to reduce the interleavings explored. Determining such independent transitions is extremely difficult in systems with tightly coupled threads sharing a large amount of global data. As a result, Verisoft would not perform well for these systems, including the storage systems checked in this paper.

Java PathFinder [2] uses related techniques to systematically check concurrent Java programs by checkpointing states. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. The techniques described in this paper are applicable to Java PathFinder as well.

Generic bug finding. There has been much recent work on bug finding, including both better type systems [9, 14, 12] and static analysis tools [8, 1, 7, 3, 10, 13]. Roughly speaking, because static analysis can examine all paths and only needs to compile code in order to check it, it is relatively better at finding errors in surface properties visible in the source ("lock is paired with unlock") [11]. In contrast, model checking requires running code, which makes it much more strenuous to apply (days or weeks instead of hours) and only lets it check executed paths. However, because it executes code it can more effectively check the properties implied by code, e.g. that the log contains valid records, that `cvs commit` will commit versioned user data to stable storage. Based on our experience with static analysis, the most serious errors in this paper would be difficult to find with that approach. But, as with testing, we view static analysis as complementary to our lightweight model checking—it is easy enough to apply that there is no reason not to apply it and then use lightweight model checking.

6. CONCLUSION

This paper demonstrated that ideas from model checking offer a promising approach to investigating crash recovery errors and that these ideas can be leveraged with much less work than previous, heavyweight implementation-level model checkers. This paper introduced a lightweight, general approach to finding such errors using a minimally invasive kernel device driver. We developed an implementation, EXPLODE, that runs on a slightly modified Linux kernel on raw hardware and applied it to a wide variety of storage systems, including eight file systems, Linux software RAID 5, Linux NFS client and server, and three version control packages, including one for which we did not have source code. We found serious data loss bugs in all of them, 25 in all. In the future we plan on extending EXPLODE in two directions: checking more comprehensively by exploring better search heuristics, and checking other mission critical storage systems such as databases.

7. REFERENCES

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[2] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[3] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: NFS version 3 protocol specification, June 1995.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*.

[7] SWAT: the Coverity software analysis toolset. `http://coverity.com`.

[8] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[9] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, Sept. 2000.

[11] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Invited paper: Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, pages 191–210, Jan. 2004.

[12] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[13] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[14] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[15] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[16] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[17] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.

[18] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[19] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[20] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Conference*, pages 109–116, June 1988.

[21] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Dec. 2004.

Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications

Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Abstract

We describe some of our experiences from developing the Dialyzer defect detection tool and overseeing its use in large-scale commercial applications of the telecommunications industry written in Erlang. In particular, we mention design choices that in our opinion have contributed to Dialyzer's acceptance in its user community, things that have so far worked quite well in its setting, the occasional few that have not, and the lessons we learned from interacting with a wide, and often quite diverse, variety of users.

1. Introduction

Programmers occasionally make mistakes, even functional programmers do. This latter species is by choice immune to some of the more typical kinds of software defects such as buffer overruns or accessing memory which has been freed, but cannot escape many other kinds of programming errors, even the more mundane ones such as simple typos. To catch some of these errors early in the development phase, many functional programmers prefer using statically typed languages such as ML or Haskell. These languages impose a static type discipline on programs and report obvious type violations during compilation. Static typing is not a panacea and does have some drawbacks. First of all, the errors that are caught are limited by the power of the currently employed type system; for example, none of the employed type systems statically catches division by zero errors or using a negative integer as an array index. Another drawback is that static typing often imposes quite stringent rules on what is considered a type-correct program (for example, type systems often require that each variable has a type which can be uniquely determined by constructors) and forces a program development model with a fair amount of constraints (e.g., ML requires that the module structure of an application is hierarchical and that there are no calls to functions with unknown type signatures).

Mainly due to reasons such as those described above, some programmers feel more at ease practicing a different religion. They adopt a more *laissez-faire* style of programming and choose to program in dynamically typed functional languages, like Lisp or Scheme, instead. Erlang [1] is such a language. In fact, it is not only dynamically typed but it also extends pattern matching by allowing type guards in function heads and in case statements. It is also a concurrent language which is used by large companies in the telecommunications industry to develop large-scale (several hundred thousand lines of code) commercial applications.

The Erlang/OTP development environment Since defect detection tools are only additional weapons in the war against software bugs, we briefly describe the surroundings of our tool. The development environment of the Erlang/OTP system from Ericsson[1] strongly encourages rapid prototyping and performing unit testing early on in the development cycle. Like many functional language implementations, the Erlang/OTP system comes with an interactive shell where Erlang modules can be loaded and the functions in them can easily be tested on an individual basis by simply issuing calls to them. If an exception occurs at any point, it is caught and presented to the user together with a stack trace which shows the sequence of calls leading to the exception. Many errors are eliminated in this way. Of course, testing of multi-thousand (and often million) LOC commercial applications such as e.g. the software of telecom switches is not limited to unit testing to catch exceptions but is much more thorough and systematic; for one such example see [7]. However, testing, no matter how thorough, cannot of course detect all software defects. Tools that complement testing, such as static analyzers, have their place in software development regardless of language. Erlang is no exception to this.

2. Dialyzer: A brief overview

The Dialyzer [5] is a lightweight static analysis tool that identifies some software defects such as obvious type errors, unreachable code, redundant tests, virtual machine bytecode which is unsafe, etc. in single Erlang modules or entire applications. Because not all of defects identified by Dialyzer are software bugs, we thereafter collectively refer to them as code *discrepancies*.[2]

Dialyzer starts its analysis either from Erlang source code or from the virtual machine bytecode that the Erlang/OTP compiler has produced and reports to its user the functions where the discrepancies occur and an indication of what each discrepancy is about.

Characteristics Notable characteristics of Dialyzer are:

- Currently Dialyzer is *a push-button technology and completely automatic*. In particular it accepts Erlang code "as is" and does not require any annotations from the user, it is very easy to customize, and supports various modes of operation (GUI vs. command-line, module-local vs. application-global analysis, using analyses of different power, focusing on certain types of discrepancies only, etc.)
- Its analysis is *complete* — though of course not guaranteed to find all errors. In particular, Dialyzer's analysis reports no false positives; more on this below.
- Its basic analysis is typically *quite fast*. On a 2GHz Pentium 4 laptop it "dialyzes" about 800 lines of Erlang code per second.

Basic functionality explained using an example The simplest way of using Dialyzer is via the command line. The command:

```
dialyzer --src -r dir
```

[1] OTP stands for Open Telecom Platform; see `www.erlang.org`.

[2] DIALYZER stands for DIscrepancy AnaLYZer of ERlang programs.

will find, recursively, all `.erl` Erlang source files under `dir` and will collectively analyze them for discrepancies. (The `--src` option is needed because, for historical reasons, analysis starts from virtual machine bytecode by default. The command with the `--src` option omitted, will analyze all `.beam` bytecode files under `dir`.)

We illustrate some of the kinds of discrepancies that Dialyzer is capable of identifying by the following, quite factitious, example. Assume that we are analyzing a bunch of modules, among them `m1` and `m2`, and that `m2` contains a function `bar` (denoted `m2:bar`) called by function `foo` in module `m1`. Because Dialyzer constructs the inter-modular function dependency graph, function `m2:bar` will be analyzed first. In doing so, let us assume that type inference determines that function `m2:bar`, when not throwing an exception, returns a result of the following type:

```
'gazonk' | {'ok', 0 | 42 | 'aaa' | [{'ok',_}]}
```

i.e., its result is either the atom `'gazonk'` or a pair (i.e., a 2-tuple) whose first element is the atom `'ok'` and its second element is either the integer `0`, or the integer `42`, or the atom `'aaa'`, or a list of pairs whose first element is the atom `'ok'`. (The use of an underscore in the second element denotes the universal type *any*.)

First of all, note that this is a type which will not be derived by the inferencers that statically typed language commonly employ. Type inferencers like those of e.g. ML would typically collapse the integers `0` and `42` to the built-in *integer* type and would not allow mixing primitive types such as integers and atoms without having them wrapped in appropriate constructors. More importantly, they would never derive an unconstrained type (such as the type *any*) at some position.

So, how come Dialyzer's type inferencer comes up with the *any* type for the second element of pairs in the list? This can happen for various reasons:

- The most common reason is that the source code of function `m2:bar` does not contain enough information to determine the type of the second element of these pairs.[3] It may indeed be the case that the function is polymorphic in this position, or more likely that this position is not constrained by information derived by or supplied to the analysis. The latter can happen if the second element of these pairs is manipulated by a function in some other module `m3` that Dialyzer knows nothing about because `m3` was not included in the analysis. (Type analyzers for statically typed languages would never tolerate this situation and simply give up here.)
- The analysis has decided to over-approximate, through *widening*, the inferred type. This can happen either to ensure termination or for performance reasons.

Given the return type for `m2:bar` shown above, when analyzing the code of function `m1:foo` — shown with numbered discrepancies as Program 1 — Dialyzer will report the following:

1. The call to the built-in function `list_to_atom`, if reached, will raise a runtime exception since it is called with an argument which is an atom rather than a list. (The programmer is obviously confused here; for example, perhaps the intention was to use the function `atom_to_list` instead.) In a similar manner, type clashes in calling other analyzed functions which are not necessarily language built-ins (e.g. `m2:bar`) will be identified. This is the kind of type errors that any static type analyzer would also be able to detect.
2. The case clause guarded by `is_integer(Num), Num < 0` will never succeed because its guard will never evaluate to true. The complete case clause is thus redundant. This is something that most static type analyzers would not be able to catch, for reasons explained above. Strictly, this is not an error but having redundant code like that scattered in the program is a strong indication for programmer confusion — programmers rarely fancy writing redundant code; see also [8]. Our experience is that such discrepancies quite often indicate places where bugs may creep, or may be the remains of obsolete interfaces; perhaps some time ago `m2:bar` was returning pairs with negative integers in their second element but not anymore. Such discrepancies indicate code that can be eliminated, which often simplifies the interface between functions. Note that in this case all the callers of `m1:foo` would also have to handle the 2-tuple where the first element is `'error'` besides `'answer'`. In any case, reporting to the user that this case clause will never succeed is a true statement prompting the user for some action. In our opinion, it cannot be considered a false positive; it is not a side-effect of inaccuracy in the analysis due to e.g. over-approximation or path-insensitivity.
3. The guard `size(List)` will fail since its argument is a list and in Erlang the `size` function only accepts tuples and binaries as its argument, not lists. (The corresponding function for lists is called `length` and this is a common programming mistake.) The problem here is that this defect will remain undetected at runtime because, for good reasons, all exceptions in guard contexts are intercepted and silenced; the semantics of `when` guards in Erlang dictates this. Testing has therefore very little chance of discovering this error. (The only method is to find out that the . . . code in the body of the case clause never executes).
4. The case clause with pattern `{'EXIT', R}` will never match. This may seem obvious given the return type of `m2:bar`, but it indicates another common Erlang programming error. The programmer probably intended to handle exceptions here, but forgot to protect the call to `m2:bar` with a `catch` construct; i.e., write the case statement as:

   ```
   case catch m2:bar(...) of
   ```

   which *would* then match the 2-tuple `{'EXIT', R}` in the case when `m2:bar` threw an exception. (Catch-based exceptions in Erlang are wrapped in a 2-tuple whose first element is the atom `'EXIT'` and the second element is a symbolic representation of the reason, which typically includes a detailed stack trace.)

Program 1 Code snippet which is full of discrepancies.

```
-module(m1).
⋮
foo(...) ->
  case (m2:bar(...) = Bar) of
    Atom when is_atom(Atom) ->
      ..., List = list_to_atom(Atom) [1], ...;
    {'ok', 42} ->
      {'answer', 42};
    {'ok', Num} when is_integer(Num), Num < 0 [2] ->
      {'error', "Negative integer - not handled yet"};
    {'ok', [H|T] = List} when size(List) [3] > 1 ->
      ...;
    {'ok', [Bar|T]} [5] ->
      ...;
    {'EXIT', R} [4] ->
      io:format("Caught exception; reason: ~p~n", [R])
  end, % end of the case statement
⋮
```

[3] Erlang programs contain no type declarations or any explicit type information.

5. This one is subtle. The pattern P = {'ok', [Bar|T]} is actually type-correct, but the pattern matching of the term returned by m2:bar and assigned to the variable Bar will never match with the pattern P, with Bar being a sub-term of it. We are not aware of any type checker that would catch this error.

On the other hand, notice that Dialyzer will *not* warn that the case statement has no catch-all clause (similar to C's default) or that the {'ok', 0} return from m2:bar is not handled. This is done so as to minimize irrelevant 'noise' from the tool. In this respect, Dialyzer differs from tools such as lint [4].

As mentioned, the example code we just presented is factitious, albeit only slightly so. Dialyzer has yet to find a single function that contains all these discrepancies in its code at the same time, but all of them, even 5, are examples of discrepancies that Dialyzer actually found in well-tested, commonly-used code of commercial products. Besides these, Dialyzer is also capable of identifying some other software defects in Erlang programs (e.g., possibly unsafe bytecode generated by older versions of the Erlang/OTP compiler and misuses of certain language constructs), but their illustration is beyond the scope of this short experience paper.

Dialyzer is of course not guaranteed to report all software defects — not even all type errors — that an Erlang application might contain. In fact, because Dialyzer's analysis is not path-sensitive, Dialyzer currently suppresses all discrepancies that might be due to losing information when joining the analysis results from different paths. This is in contrast to other defect detection tools that do report false positives due to inaccuracies or heuristics in analyses they employ.

Dialyzer comes with an extensive set of options that allow its user to focus on certain types of discrepancies only and employ analyses of varying precision vs. time complexity trade-offs. It also comes with a graphical user interface in which the user is able to inspect the information that led to the identification of some discrepancy of interest.

For more up-to-date information, see also Dialyzer's homepage: www.it.uu.se/research/group/hipe/dialyzer/

3. Dialyzer's usage so far

Even before its first public release, Dialyzer was applied to relatively large code bases, both by us and more commonly by Erlang application developers. We have been working closely with developers of the AXD301 and GPRS[4] projects at Ericsson, and with a T-Mobile team in the U.K. which also uses Erlang for some of its product development. An early account of the effectiveness of an internal and significantly less powerful version of the tool appears in [5].

The first releases of Dialyzer, versions 1.0.*, featured analysis starting from virtual machine bytecode only and the tool only had a GUI mode. We were somewhat (positively) surprised to receive numerous user requests to develop a command-line version of the tool so that Dialyzer becomes more easily integrated to the purely make-based build environment of some projects.[5] Once this happened, we even received extensions to the tool's functionality contributed by users that made it into the next release. For example, the code that supports the -r (add files recursively) option was a user contribution; before that, users were forced to manually specify all files and directories to include in the analysis.

At the time of this writing, some of the code bases analyzed by Dialyzer are open-source community programs (e.g., the code of the Wings 3D subdivision graphics modeler,[6] of the Yaws web server,[7] and of the esdl graphical user interface library[8]). However, the majority of Dialyzer's uses are large commercial applications from the telecommunications domain. Among them is the code base of the AXD301 ATM switch consisting of about 2,000,000 lines of Erlang code, where by now Dialyzer has identified a significant number (many hundreds) of software defects that have gone unnoticed after years of extensive testing. It is also continuously being used in the Erlang/OTP R10 (release 10) system to eliminate numerous bugs that previous releases contained in some of its standard libraries. We also know that Dialyzer is being used on the code of some of Nortel's products, but we do not have any further information on it.

At least in the commercial projects, Dialyzer is typically run as part of a centralized (often nightly) build. Perhaps because of this, many Dialyzer users typically complain that Dialyzer's identification of discrepancies is not as clear and concise as the messages they are used to getting from the Erlang/OTP compiler. Although it is indeed the case that currently there is plenty of room for improvement in Dialyzer's presentation of the identified discrepancies, it is clear that for many of the discrepancies simple one-line explanations of the form "line 42: unused variable X" will never be possible. Some of the discrepancies identified are complex, involve interactions of functions from various modules, and it is not always clear how to assign blame. As a simple example, note discrepancy 4 of Program 1: Dialyzer will currently complain that:

```
The clause matching involving 'EXIT' will never match;
argument is of type 'ok'
```

while the culprit is probably a missing catch construct after the case. As a more involved example, for discrepancy 2, Dialyzer will report that the guard will always fail. But perhaps function m2:bar should return negative integers in this tuple position after all.[9] Finding why inter-modular type inference determines that m2:bar only returns the numbers 0 and 42 in that tuple position might not be at all trivial — especially to users who are not familiar with type systems.

4. Experiences and lessons learned

Requests for better explanations of identified discrepancies aside, Dialyzer has been extremely successful. It has managed to identify a significant number of software defects that have remained undetected after a long period of extensive testing. For example, because of the high level of reliability required from telecom switches, the developers of AXD301, a team consisting of about 200 people, has over a period of more than eight years spent a considerable percentage of their effort on testing. Still Dialyzer managed to identify many discrepancies and often serious bugs. As another example, certain bugs in Erlang/OTP standard libraries managed to survive over many releases of the system, despite being in commonly used modules of the system. Although this may sound a bit contradictory, it has a simple explanation. Many of the bugs were in error-handling code or code paths of the library modules which were not executed frequently enough.

[4] GPRS: General Packet Radio Service.

[5] We thought that when academic projects got *real* users, it was supposed to be the other way around: the users would demand a GUI! More seriously, this is somewhat contrary to experience reported in literature. For example [2] reports that a significant portion of the effort is spent in explaining defects in a user-friendly way (presumably aided by a GUI). This probably does tell something about the habits of the Erlang programmer community, which is mostly Unix-centered, but we will not try to analyze it further.

[6] See www.wings3d.com/.

[7] Yaws: Yet Another Web Server; see yaws.hyber.org/.

[8] Erlang OpenGL/SDL API and utilities; see sourceforge.net/projects/esdl.

[9] Worse yet, there might even be a comment in its code to the effect that m2:bar possibly returns a negative integer. Some programmers currently trust comments more than output from static analyzers... but we are working on slowly changing this!

```
remote_dirty_select(Tab, [{HeadPat,_,_}] = Spec, [H|T]) when tuple(HeadPat), size(HeadPat) > 2, H =< size(Spec) ->
    ... % code for the body of this clause
remote_dirty_select(...) ->
    .
    .
    .  % code for this and other clauses below handling select queries with general specifications
```

Figure 1. Code from the `mnesia` database with a guard that will always fail making the body of the first clause unreachable.

Observations Some qualitative observations can be made:

- The vast majority of (at least non-trivial) defects identified by Dialyzer are due to the interaction between multiple functions; a significant number of them span across module boundaries. This is mainly due to the fact that Erlang is great for testing functions on an individual basis (or in small sets), but currently provides little support for specifying and ensuring proper use of functions in other modules.
- Probably due to the reason described above, we did not observe the usual inverse correlation between the age of some piece of code and the number of discrepancies in it, at least not directly. The problem is that callers of some function may be significantly older than the callee and as the callee's interface evolves, the callers possibly remain unchanged. Having redundant clauses handling return values that were perhaps returned long ago but not anymore, is an extremely common discrepancy. As mentioned, such code is typically harmless but often desperately in need for cleanups. Doing so, simplifies the code where the discrepancy occurs and exposes opportunities for further simplifications elsewhere, often significant ones.
- Even in dynamically typed languages such as Erlang, the code that is frequently executed does not have type errors. As a result, Dialyzer tends to find most discrepancies off the commonly executed paths. Commonly executed paths are often reasonably well-tested and most discrepancies have already been eliminated. On the other hand, exception- or error-handling code, code that handles timeouts in concurrent programs, etc. does not always have this property.
- Quite often, fixing even simple discrepancies in some particular piece of code exposes more serious ones in code which lies in close proximity to the code which is fixed.

Most of these observations are not very surprising and in line with those of other researchers in the area.

Myths On the other hand, our experience so far has made us seriously doubt the validity of the following common beliefs:

1. *Software defects identified by a static analysis tool are shallow.* Occasionally one might see such a statement, especially in comparisons between static analysis and model checking techniques; see e.g. [3]. It is of course very hard to dispute such a statement, as its validity depends on what one considers as a "shallow" defect, but we will try to do so anyway.

   Figure 1 shows a small code segment from the code of Mnesia [6], a database management system distributed as part of Erlang/OTP. It also shows an actual discrepancy identified by Dialyzer, which is now fixed. On the surface, the indicated discrepancy is indeed shallow; a simple misuse of a library predicate (using `size` on a list rather than `length`). Viewed from this prism, there is indeed nothing "deep" here: the programmer made a silly mistake. Because the function `remote_dirty_select` is quite commonly used, what is surprising in this case is that this mistake managed to remain unnoticed over many Erlang/OTP releases. The subtlety of the problem was actually in the other clauses for the function. This bug was not identified because this clause was there for optimization purposes (in order to handle the common case of a single-element specification list fast). The subsequent clauses also provided the functionality of the first clause, but using a more general (handling specification lists of any size) and thus more expensive mechanism. It is very difficult to identify such performance-related software defects by means of (even exhaustive) testing. It is not clear to us that such software defects, for which the correctness criterion cannot be specified using a simple formula whose validity can be checked by model-checking techniques, are of the "shallow" kind.
2. *Software defects, once identified, are soon fixed.* Coming from academia, one is a bit shocked to discover that the "real world" is somehow different. Fixing bugs, no matter how serious, is not always a developer's top-priority, because program development in the real world follows a different model than that of open-source projects managed by small teams of individuals. Software evolution in big commercial projects goes hand-in-hand with filling bug reports, sending them to the developer who is responsible for the maintainance of the piece of code containing the bug, caring about backwards compatibility even when the functionality is crippled, and often having to invest a non-trivial amount of effort in order to modify or extend existing regression test suites, which are typically not maintained by programmers but by a separate testing team. (Our experience here is actually in line with that of other researchers; see e.g. [2, 3].)

   In fact, we even seriously doubt that an automatic classification of the seriousness of defects would help here. As a concrete example, we reported 18 discrepancies that Dialyzer identified in some library of Erlang/OTP to the library's maintainer, which incidentally was not the original author. Most of them were fixed pretty soon, but one in particular — which was the most serious — was not. In fact, it remained unfixed for quite some time. The reason for this was that it would involve serious redesign of the code and this might significantly change the behaviour of the library.
3. *Only programs written in low-level languages such as C seriously benefit from defect detection tools.* Not many serious developers believe this statement anyway, but often one of the arguments used in favour of high-level languages is that these languages avoid common programmer errors. Although this is a very true statement in some contexts (for example, one does not have the possibility to free memory once, let alone twice, in a garbage-collected language), it often fails to point out the fact that any language, no matter how high a level of abstraction it offers, has silly pitfalls and traps for developers, and these are often directly connected, and difficult to separate from, the language's strengths. We hold that software defect detection tools, especially lightweight ones, have their place in sofware development independently of the programming environment and language which is employed.

Final remarks Dialyzer is a static analysis tool identifying discrepancies — out of which some of them are serious bugs — in Erlang applications. We believe that the following, perhaps unique characteristics, have played a crucial role in Dialyzer's acceptance by its user community:

- The tool is extremely lightweight and requires absolutely no code changes or user-supplied annotations in order to be useful.
- The tool has so far tried its best to keep down the level of 'noise' which it generates, often at the expense of failing to report actual bugs. For the first versions of Dialyzer, one desired feature was to never issue a warning that could be perceived as misleading or be such that the user would find it extremely difficult to interpret. For example, we noticed that when analyzing virtual machine bytecode which has been generated using aggressive inlining, it would probably be difficult for naïve users to interpret the discrepancies. The approach we took was to simply suppress all discrepancies found in inline-compiled bytecode.

Although some might no doubt find this approach a bit extreme, we felt it was important for Dialyzer to succeed in gaining the developers' trust and be integrated in a non-disruptive way in the development process (i.e., without requiring any methodological changes from the users). Of course, this is only step number one. Once the developers' attitude and expectation level has been raised sufficiently, we intend to provide options that lift some of these restrictions.

Acknowledgments

Dialyzer's implementation would not have been possible without Tobias Lindahl. Thanks also go to all Dialyzer's users for useful contributions, feedback, thought-provoking requests for extensions of its functionality, and often working code for these changes.

The research of the author has been supported in part by grant #621-2003-3442 from the Swedish Research Council and by the Vinnova ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson AB and T-Mobile U.K.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, June 2000.

[3] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation. Proceedings of the 5th International Conference*, volume 2937 of *LNCS*, pages 191–210. Springer, Jan. 2004.

[4] S. C. Johnson. Lint, a C program checker. Technical report, Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, Dec. 1977.

[5] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[6] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia - a distributed robust DBMS for telecommunications applications. In G. Gupta, editor, *Practical Applications of Declarative Languages: Proceedings of the PADL'1999 Symposium*, volume 1551 of *LNCS*, pages 152–163, Berlin, Germany, Jan. 1999. Springer.

[7] U. Wiger, G. Ask, and K. Boortz. World-class product certification using Erlang. *SIGPLAN Notices*, 37(12):25–34, Dec. 2002.

[8] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Trans. Software Eng.*, 29(10):915–928, Oct. 2003.

Soundness by Static Analysis and False alarm Removal by Statistical Analysis: Our Airac Experience*

Yungbum Jung, Jaehwang Kim, Jaeho Shin, Kwangkeun Yi
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr
Programming Research Laboratory
Seoul National University

ABSTRACT

We present our experience of combining, in a realistic setting, a static analysis for soundness and a statistical analysis for false-alarm removal. The static analyzer is Airac that we have developed in the abstract interpretation framework for detecting bu er overruns in ANSI + GNU C programs. Airac is sound (nding all bugs) but with false alarms. Airac raised, for example, 1009 bu er-overrun alarms in commercial C programs of 636K lines and 183 among the 1009 alarms were true. We addressed the false alarm problem by computing a probability of each alarm being true. We used Bayesian analysis and Monte Carlo method to estimate the probabilities and their credible sets. Depending on the user-provided ratio of the risk of silencing true alarms to that of false alarming, the system selectively present the analysis results (alarms) to the user. Though preliminary, the performance of the combination lets us not hastily trade the analysis soundness for a reduced number of false alarms.

1. Introduction

When one company's software quality assurance department started working with us to build a static analyzer that automatically detects bu er overruns[1] in their C softwares, they challenged us on three aspects: they hoped the analyzer 1) to be sound, detecting all possible bu er overruns; 2) to have a "reasonable" cost-accuracy balance; 3) not to assume a particular set of programming style about the C programs to analyze. Building a C bu er-overrun analyzer that satis es all the three requirements was a big challenge. In the literature, we have seen impressive static analyzers, but their application targets allow them to drop one of the three aspects [6, 3, 10, 8, 9].

In this article, we present our response that consists of two things: a sound analyzer named Airac and a statistical analysis engine on top of it. Airac collects all the true bu er-overrun points in C programs yet always with false alarms. The soundness is maintained, and the analysis accuracy is stretched to a point where the analysis cost remains acceptable. The statistical engine, given the analysis results (alarms), estimates the probability of each alarm being true. Only the alarms that have true-alarm probabilities higher than a threshold are reported to the user. The threshold is determined by the user-provided ratio of the risk of silencing true alarms to that of raising false alarms.

*An extended version of this paper will be presented in SAS 2005. This work was supported by Brain Korea 21 of Korea Ministry of Education and Human Resource Development, and by National Security Research Institute of Korea Ministry of Information and Communication.

[1]Bu er overruns happen when an index value is out of the target bu er size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to bu er overruns.

2. Airac, a Sound Analyzer

Automatically detecting bu er overruns in C programs is not trivial. Arbitrary expressions from simple arithmetics to values returned by function calls can be array indices. Pointers pointing bu ers can be aliased and they can be passed over as function parameters and returned from function calls. Bu ers and pointers are equivalent in C. Contents of bu ers themselves also can be used as indexes of arrays. Pointer arithmetic complicates the problem once more.

Airac's sound design is based on the abstract interpretation framework[4, 5]. To nd out all possible bu er overruns in programs, Airac has to consider all states which can occur during programs executions. Airac computes sound approximation of program state at every program point and reports all possible bu er overruns by examining the approximate program states.

For a given program, Airac computes a map from flow edges to abstract machine states. The abstract machine state consists of abstract stack, abstract memory and abstract dump. Abstract stack, abstract memory and abstract dump are maps of which range domains consist of abstract values. We use interval domain $\widehat{\mathbb{Z}}$ for abstract numeric values. $[a, b] \in \widehat{\mathbb{Z}}$ represents an integer interval that has a as minimum and b as maximum. And this interval means a set of numeric values between a and b. To represent in nite interval, we use $-\infty$ and $+\infty$. $[-\infty, +\infty]$ means all integer values. An abstract array (an abstract pointer to an array) is a triple which consists of its base location, its size interval, and an o set interval. We use allocation sites to denote abstract memory locations. An integer array which is allocated at l and has size s is represented as $\langle l, [s, s], [0, 0]\rangle$.

2.1 Striking a Cost Accuracy Balance

Airac has many features designed to decrease false alarms or to speed-up analysis and all techniques don't violate the analysis soundness.

2.1.1 Accuracy Improvement

	Software	#Lines	Time (sec)	#Airac Alarms		#Real bugs
				#Bu ers	#Accesses	
GNU S/W	tar-1.13	20,258	576.79s	24	66	1
	bison-1.875	25,907	809.35s	28	50	0
	sed-4.0.8	6,053	1154.32s	7	29	0
	gzip-1.2.4a	7,327	794.31s	9	17	0
	grep-2.5.1	9,297	603.58s	2	2	0
Linux kernel version 2.6.4	vmax302.c	246	0.28s	1	1	1
	xfrm_user.c	1,201	45.07s	2	2	1
	usb-midi.c	2,206	91.32s	2	10	4
	atkbd.c	811	1.99s	2	2	2
	keyboard.c	1,256	3.36s	2	2	1
	af_inet.c	1,273	1.17s	1	1	1
	eata_pio.c	984	7.50s	3	3	1
	cdc-acm.c	849	3.98s	1	3	3
	ip6_output.c	1,110	1.53s	0	0	0
	mptbase.c	6,158	0.79s	1	1	1
	aty128fb.c	2,466	0.32s	1	1	1
Commercial Softwares	software 1	109,878	4525.02s	16	64	1
	software 2	17,885	463.60s	8	18	9
	software 3	3,254	5.94s	17	57	0
	software 4	29,972	457.38s	10	140	112
	software 5	19,263	8912.86s	7	100	3
	software 6	36,731	43.65s	11	48	4
	software 7	138,305	38328.88s	34	147	47
	software 8	233,536	4285.13s	28	162	6
	software 9	47,268	2458.03s	25	273	1

Table 1: Analysis speed and accuracy of Airac

We use the following techniques to improve the analysis accuracy of Airac:

- **Unique Renaming** Memory locations are abstracted by allocation sites. In Airac, sites of variable declarations are represented by variable name and other sites are assigned unique labels. So to prevent interferences among variables, Airac renames all variables to have unique names.
- **Narrowing After Widening** The height of integer interval domain is in nite. Widening operator[4] is essential for the analysis termination. But this operator decreases accuracy of analysis result. Narrowing is used for recovering accuracy.
- **Flow Sensitive Analysis** Destructive assignment is always allowed except for within cyclic flow graphs.
- **Context Pruning** We can con ne interval values using conditional expressions of branch statements. Airac uses these information to prune interval values and this pruning improve analysis accuracy.
- **Polyvariant Analysis** Function-inlining e ect by labeling function-body expressions uniquely to each call-site: the number of di erent labels for an expression is bound by a value from user. This method is weakened within recursive call cycles.
- **Static Loop Unrolling** Loop-unrolling e ect by labeling loop-body expressions uniquely to each iteration: the number of di erent labels for an expression is bound by a value from the user.

2.1.2 Cost Reduction

When the xpoint iteration reaches the junction points, we have to check the partial orders of abstract machines and we also commit the join($\sqcup$) operations. These tasks take most of analysis time. The speed of the analysis highly depends on how we handle such operations e ciently.

We developed techniques to reduce time required for partial order checking and join operation.

- **Stack Obviation** We transform the original programs whose e ects on stack are reflected by the memory. And this transformation makes Airac avoid scanning abstract stacks during ordering abstract machines.
- **Selective Memory Join** Airac keeps track of information that indicates changed entries in abstract memory. Join operation is applied only to those changed values.
- **Wait-at-Join** For program points where many data flows join, Airac delays the computation for edges starting from the current point until all computations for the incoming edges are done.

3. Performance of Airac

This section presents Airac's performance. Numbers that are before the statistical engine sift out alarms that are probably false.

Airac is implemented in nML[2] and tested to analyze GNU softwares, Linux kernel sources and commercial softwares.

[2]Korean dialect of ML programming language. `http://ropas.snu.ac.kr/n`

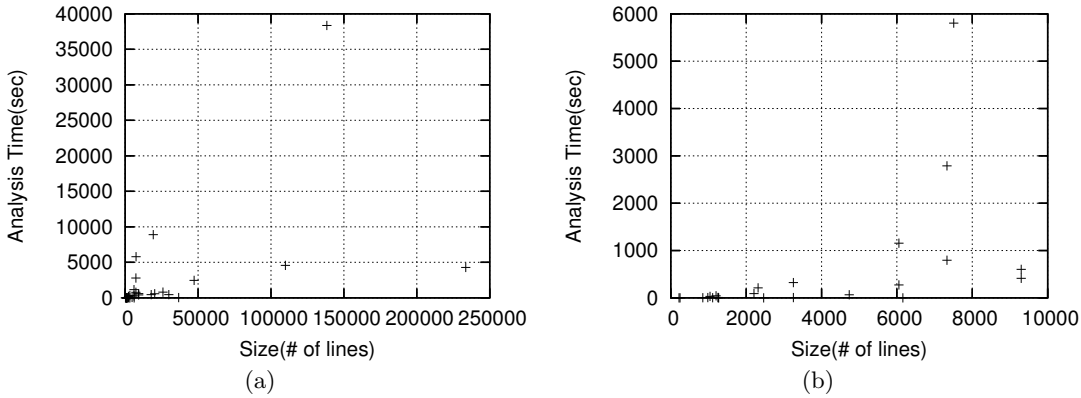


Figure 1: Airac's scalability

The commercial softwares are all embedded softwares. Airac found some fatal bugs in these softwares which were under development. Table 1 shows the result of our experiment. "#Lines" is the number of lines of the C source les before preprocessing them. "Time" is the user CPU time in seconds. "#Bu ers" is the number of bu ers those may be overrun. "#Accesses" is the number of bu er-access expressions that may overrun. "#Real Bugs" is the number of bu er accesses that are con rmed to be able to cause real overruns. Two graphs in Figure 1 show Airac's scalability behavior. X axis is the size (number of lines) of the input program to analyze and Y axis is the analysis time in seconds. (b) is a microscopic view of (a)'s lower left corner. Experiment was done in a Linux system with a Pentium4 3.2GHz CPU and 4GB of RAM.

We found some examples in real codes that Airac's accuracy and soundness shines:

- In GNU S/W tar-1.13 program rmt.c source, Airac detected the overrun point inside the `get_string` function to which a bu er pointer is passed:

```
static void
get_string (char *string)
{
  int counter;

  for (counter = 0;
       counter < STRING_SIZE;
       counter++) {
    .....
  }
  string[counter] = '\0';
      // counter == STRING_SIZE
}


int
main (int argc, char *const *argv)
{
  char device_string[STRING_SIZE];
  ......
  get_string(device_string);
  ......
}
```

- Airac catched errors in the following simple cases, for which syntactic pattern matching or unsound analyzer are likely to fail to detect.
  - Function pointer is used for calculating an index value:

```
int incr(int i) { return i+1;}
int decr(int i) { return i-1;}

main() {
  int (*farr[]) (int) = {decr, decr, incr};
  int idx = rand()%3;
  int arr[10];
  int num = farr[idx](10);
  arr[num] = 10;          //index:[9, 11]
}
```

  - Index variable is increased in an in nite loop:

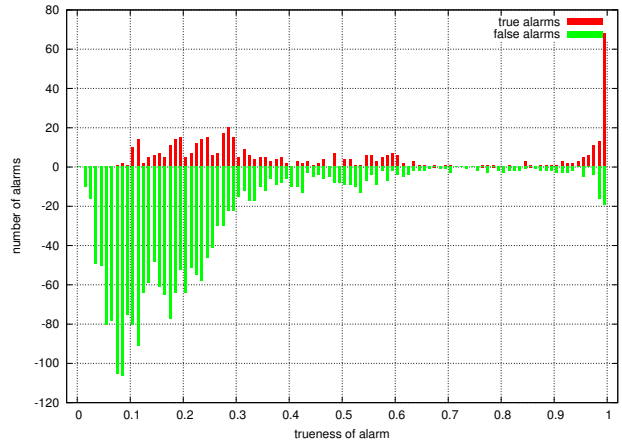
```
main() {
  int arr[10];
  int i = 0;
  while(1){
    *(arr + i) = 10;    //index:[0, +Inf]
    i++;
  }
}
```

  - Index variable is passed to a function by parameter and updated in the function:

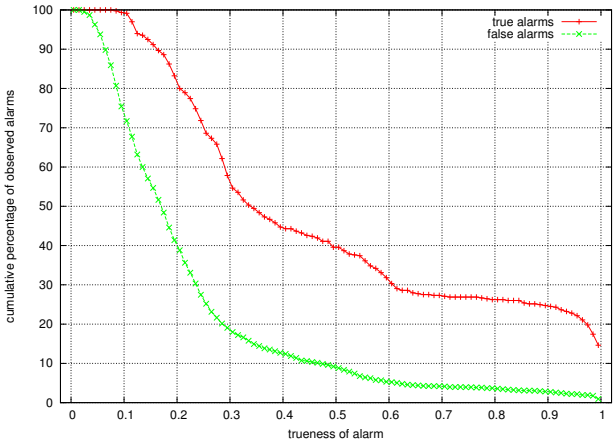
```
simpleCal(int idx) {
  int arr[10];
  idx += 5;
  idx += 10;
  arr[idx];             //index:[17, 17]
}
main() {
  simpleCal(2);
}
```

4. Sifting out False Alarms By Statistical Post Analysis

We use Bayesian approach [7] to compute the probability of alarms being true. Let $\oplus$ denote the event an alarm raised is true and $\ominus$ the event an alarm is false. S_i denotes a single symptom is observed in the raised alarm and $\vec{S}$ is a vector of such symptoms. $P(E)$ denotes the probability of an event E, and $P(A \mid B)$ is the conditional probability of A given B. Bayes' rule is used to predict the probability of a new event from prior knowledge. In our case, we accumulate the number of true and false alarms having each speci c symptom from alarms already veri ed and classi ed to be true or false by humans. From this knowledge we compute



(a) Frequency of probability being true in true and false alarms. False alarms are counted in negative numbers. 74.83% of false alarms have probability less than 0.25.



(b) Cumulative percentage of observed alarms starting from probability 1 and down.

Figure 2: Experiment results

the probability of a new alarm with some symptoms being a true one.

To compute the Bayesian probability, we need to de ne symptoms featuring alarms and gather them from already analyzed programs and classi ed alarms. We de ned symptoms both syntactically and semantically. Syntactic symptoms describe the syntactic context before the alarmed expressions. The syntactic context consists of program constructs used before the alarmed expressions. Semantic symptoms are gathered during Airac's xpoint computation phase. For such symptoms, we de ned symptoms representing whether context pruning was applied, whether narrowing was applied, whether an interval value has in nity and so forth.

From the Bayes' theorem, probability $P(\oplus \mid \vec{S})$ of an alarm being true that has symptoms $\vec{S}$ can be computed as the following:

$$P(\oplus \mid \vec{S}) = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S})} = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S} \mid \oplus)P(\oplus) + P(\vec{S} \mid \ominus)P(\ominus)}.$$

By assuming each symptom in $\vec{S}$ occurs independently under each class, we have

$$P(\vec{S} \mid c) = \prod_{S_i \in \vec{S}} P(S_i \mid c) \text{ where } c \in \{\oplus, \ominus\}.$$

Here, $P(S_i \mid c)$ is estimated by Bayesian analysis from our empirical data. We assume prior distributions are uniform on $[0, 1]$. Let p be the estimator of the probability $P(\oplus)$ of an alarm being true. $P(S_i \mid \oplus)$ and $P(S_i \mid \ominus)$ are estimated by θ_i and η_i respectively. Assuming that each S_i are independent in each class, the posterior distribution of $P(\oplus \mid \vec{S})$ taking our empirical data into account is established as following:

$$\hat{\psi_j} = \frac{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p}{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p + (\prod_{S_i \in \vec{S}} \eta_i) \cdot (1 - p)} \qquad (1)$$

where p, θ_i and η_i have beta distributions as

$$\begin{aligned} p &\sim Beta(N(\oplus) + 1, n - N(\oplus) + 1) \\ \theta_i &\sim Beta(N(\oplus, S_i) + 1, N(\oplus, \neg S_i) + 1) \\ \eta_i &\sim Beta(N(\ominus, S_i) + 1, N(\ominus, \neg S_i) + 1) \end{aligned}$$

and $N(E)$ is the number of events E counted from our empirical data.

Now the estimation of p, θ_i,η_i are done by Monte Carlo method. We randomly generate $p_i, \theta_{ij}, \eta_{ij}$ values N times from the beta distributions and compute N instances of ψ_j. Then the $100(1-2\alpha)\%$ credible set of $\hat{\psi}$ is $(\psi_{j_{\alpha \cdot N}}, \psi_{j_{(1-\alpha) \cdot N}})$ where $\psi_{j_1} < \psi_{j_2} < \cdots < \psi_{j_N}$. We take the upper bound $\psi_{j_{(1-\alpha) \cdot N}}$ for $\hat{\psi}$. After obtaining the upper bound $\hat{\psi}$ of probability being true for each alarm, we have to decide whether we should report the alarm or not. To choose a reasonable threshold, user supplies two parameters de ning the magnitude of risk: r_m for not reporting true alarms and r_f for reporting false alarms.

	$\oplus$	$\ominus$
risk of reporting	0	r_f
risk of not reporting	r_m	0

Given an alarm whose probability being true is ψ, the expectation of risk when we raise an alarm is $r_f \cdot (1 - \psi)$, and $r_m \cdot \psi$ when we don't. To minimize the risk, we must choose the smaller side. Hence, the threshold of probability to report the alarm can be chosen as:

$$r_m \cdot \psi \geq r_f \cdot (1 - \psi) \quad \Longleftrightarrow \quad \psi \geq \frac{r_f}{r_m + r_f}.$$

If the probability of an alarm being true can be greater than or equal to such threshold, i.e. if the upper bound of $\hat{\psi}$ is greater than such threshold, then the alarm should be raised with $100(1-2\alpha)\%$ credibility. For example, user can supply $a_1 = 3, a_2 = 1$ if (s)he believes that not alarming for true errors have risk 3 times greater than raising false alarms. Then the threshold for the probability being true to report becomes $1/4 = 0.25$ and whenever the estimated probability of an alarm is greater than 0.25, we should report it. For a sound analysis, to miss a true alarm is considered much riskier than to report a false alarm, so it is recommended to choose the two risk values $a_1 \gg a_2$ to keep more soundness.

We have done some experiments with our samples of programs and alarms. Some parts of the Linux kernel and programs that demonstrate classical algorithms were used for the experiment. For a single experiment, samples were rst

divided into learning set and testing set. 50% of the alarms were randomly selected as learning set, and the others for testing set. Each symptom in the learning set were counted according to whether the alarm was true or false. With these pre-calculated numbers, $\hat{\psi}$ for each alarm in the testing set was estimated using the 90% credible set constructed by Monte Carlo method. Using Equation (1), we computed 2000 ψ_j's from 2000 p's and θ_i's and η_i's, all randomly generated from their distributions. We can view alarms in the testing set as alarms from new programs, since their symptoms didn't contribute to the numbers used for the estimation of $\hat{\psi}$.

Figure 2 was constructed from the data generated by repeating the experiment 15 times. For the histogram (a) on the left, dark bars indicate true alarms and lighter ones are false. 74.83% (≈1504/2010) of false alarms have probability less than 0.25, so that they can be sifted out. For users who consider the risk of missing true error is 3 times greater than false alarming, almost three quarters of false alarms could be sifted out, or preferably just deferred.

For a sound analysis, it is considered much riskier to miss a true alarm than to report a false one, so it is recommended to choose the two risk values $r_m \gg r_f$ to keep more soundness. For the experiment result Figure 2 presents, 31.40% (≈146/465) of true alarms had probability less than or equal to 0.25, and were also sifted out with false alarms. Although we do not miss any true alarm by lowering the threshold down to 0.07 ($r_m/r_f \approx 13$) for this case, it does not guarantee any kind of soundness in general. However, to obtain a sound analysis result, one can always set $r_f = 0$, i.e. allowing none of the alarms to be sifted out.

We can rank alarms by their probability to give e ective results to user. This ranking can be used both with and without the previous sifting-out technique. By ordering alarms, we let the user handle more probable errors rst. Although the probable of true alarms are scattered over 0 through 1, we can see that most of the false alarms have small probability. Hence, sorting by probability and showing in decreasing order will e ectively give true alarms rst to the user. (b) of Figure 2 shows the cumulative percentage of observed alarms starting from probability 1 and down. Only 15.17% (=305/2010) of false alarms were mixed up until the user observes 50% of the true alarms, where the probability equals 0.3357.

5. Conclusion

Our Airac experience encourages us to conclude that in a realistic setting it is not inevitable to trade the soundness for a reduced number of false alarms. By striking a cost-accuracy balance of a sound analyzer, we can rst achieve an analyzer that is itself useful with small false-alarm rate in most cases (as the experiment numbers showed for analyzing Linux kernels). Then, by a careful design of a Bayesian analysis of the analyzer's false-alarm behaviors, we can achieve a post-processing engine that sifts out false alarms from the analysis results.

Though the Bayesian analysis phase still has the risk of sifting out true alarms, it can reduce the risk at the user's desire. Given the user-provided ratio of the risk of silencing true alarms to that of false alarming, a simple decision theory determines the threshold probability that an alarm with a lower probability is silenced as a false one. Because the underlying analyzer is sound, if the user is willing to, (s)he can receive a report that contain all the real alarms. For Airac, when the risk of missing true alarms is three times greater than that of false alarming, three quarters of false alarms could be sifted out. Moreover, if user inspects alarms having high probability rst, only 15% of the false ones get mixed up while 50% of the trues are observed.

The Bayesian analysis' competence heavily depends on how we de ne symptoms. Since the inference framework is known to work well, better symptoms and feasible size of pre-classi ed alarms is the key of this approach. We think promising symptoms are tightly coupled with analysis' weakness and/or its preciseness, and some fair insight into the analysis is required to de ne them. However, since general symptoms, such as syntactic ones, are tend to reflect the programming style, and such patterns are well practiced within organizations, we believe local construction and use of the knowledge base of such simple symptoms will still be e ective. Furthermore, we see this approach easily adaptable to possibly any kind of static analysis.

Another approach to handling false alarms is to equip the analyzer with all possible techniques for accuracy improvement and let the user choose a right combination of the techniques for her/his programs to analyze. The library of techniques must be extensive enough to specialize the analyzer for as wide spectrum of the input programs as possible. This approach lets the user decide how to control false alarms, while our Bayesian approach lets the analysis designer decide by choosing the symptoms based on the knowledge about the weakness and strength of his/her analyzer. We see no reason we cannot combine the two approaches.

Acknowledgements We thank Jaeyong Lee for helping us design our Bayesian analysis. We thank Hakjoo Oh and Yikwon Hwang for their hardwork in collecting programs and classifying alarms used in our experiments.

6. References

[1] bugtraq. `www.securityfocus.com`.

[2] CERT/CC advisories. `www.cert.org/advisories`.

[3] Bruno Blanchet, Patric Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antonie Mine, David Monnizux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a uni ed lattice model for static analysis of programs by construction or approximation of xpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[6] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all bu er overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167. ACM Press, 2003.

[7] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Texts in Statistical Science. Chapman & Hall/CRC, second edition edition, 2004.

[8] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[9] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, New York, NY, USA, 2004. ACM Press.

[10] Yichen Xie, Andy Chou, and Dawson 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. ACM Press, 2003.

Dynamic Buffer Overflow Detection*

Michael Zhivich
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
mzhivich@ll.mit.edu

Tim Leek
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
tleek@ll.mit.edu

Richard Lippmann
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
lippmann@ll.mit.edu

ABSTRACT

The capabilities of seven dynamic buffer overflow detection tools (Chaperon, Valgrind, CCured, CRED, Insure++, ProPolice and TinyCC) are evaluated in this paper. These tools employ different approaches to runtime buffer overflow detection and range from commercial products to open-source gcc-enhancements. A comprehensive testsuite was developed consisting of specifically-designed test cases and model programs containing real-world vulnerabilities. Insure++, CCured and CRED provide the highest buffer overflow detection rates, but only CRED provides an open-source, extensible and scalable solution to detecting buffer overflows. Other tools did not detect off-by-one errors, did not scale to large programs, or performed poorly on complex programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; K.4.4 [**Computers and Society**]: Electronic Commerce Security

General Terms

Measurement, Performance, Security, Verification

Keywords

Security, buffer overflow, dynamic testing, evaluation, exploit, test, detection, source code

*This work was sponsored by the Advanced Research and Development Activity under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

2005 Workshop on the Evaluation of Software Defect Detection Tools 2005, June 12, Chicago, IL.

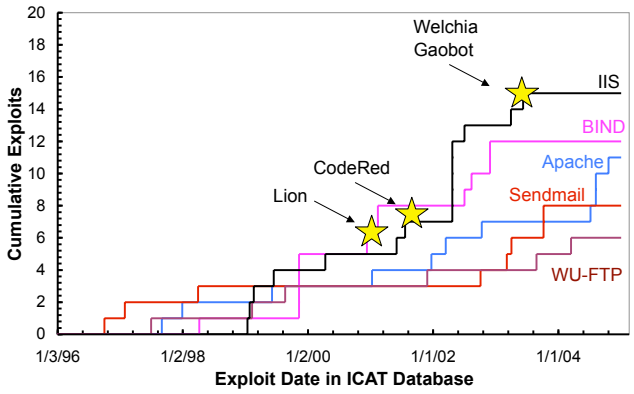


Figure 1: Cumulative exploits in commonly used server software.

1. INTRODUCTION

Today's server software is under constant scrutiny and attack, whether for fun or for profit. Figure 1 shows the cumulative number of exploits found in commonly used server software, such as `IIS`, `BIND`, `Apache`, `sendmail`, and `wu-ftpd`. The stars indicate appearances of major worms, such as `Lion`, `CodeRed` and `Welchia`. As the data demonstrates, new vulnerabilities are still found, even in code that has been used and tested for years. A recent analysis by Rescorla [18] agrees with this observation, as it shows that vulnerabilities continue to be discovered at a constant rate in many types of software.

Buffer overflows enable a large fraction of exploits targeted at today's software. Such exploits range from arbitrary code execution on the victim's computer to denial of service (DoS) attacks. For 2004, CERT lists 3,780 vulnerabilities [3], while NIST reports that 75% of vulnerabilities in its ICAT database are remotely exploitable, of which 21% are due to buffer overflows [15]. Detecting and eliminating buffer overflows would thus make existing software far more secure.

There are several different approaches for finding and preventing buffer overflows. These include enforcing secure coding practices, statically analyzing source code, halting exploits via operating system support, and detecting buffer overflows at runtime [5]. Each approach has its advantages; however, each also suffers from limitations. Code reviews, no matter how thorough, will miss bugs. Static analysis seems

like an attractive alternative, since the code is examined automatically and no test cases are required. However, current static analysis tools have unacceptably high false alarm rates and insufficient detection rates [24]. Operating system patches, such as marking stack memory non-executable, can only protect against a few types of exploits.

Dynamic buffer overflow detection and prevention is an attractive approach, because fundamentally there can be no false alarms. Tools that provide dynamic buffer overflow detection can be used for a variety of purposes, such as preventing buffer overflows at runtime, testing code for overflows, and finding the root cause of segfault behavior.

One disadvantage of using this approach to find errors in source code is that an input revealing the overflow is required, and the input space is generally very large. Therefore, dynamic buffer overflow detection makes the most sense as part of a system that can generate these revealing inputs. This evaluation is part of a project to create a grammar-based dynamic program testing system that enables buffer overflow detection in server software before deployment. Such a testing system will use the dynamic buffer overflow detection tool to find buffer overflows on a range of automatically-generated inputs. This will enable a developer to find and eliminate buffer overflows before the faults can be exploited on a production system. A similar testing approach is used in the PROTOS project at the University of Oulu [13].

This paper focuses on evaluating the effectiveness of current dynamic buffer overflow detection tools. A similar evaluation has been conducted by Wilander et al. [22], but it focused on a limited number of artificial exploits which only targeted buffers on the stack and in the bss section of the program. Our evaluation reviews a wider range of tools and approaches to dynamic buffer overflow detection and contains a more comprehensive test corpus.

The test corpus consists of two different testsuites. Section 3 presents the results for *variable-overflow* testsuite, which consists of 55 small test cases with variable amounts overflow, specifically designed to test each tool's ability to detect small and large overflows in different memory regions. Section 4 presents the results for 14 model programs containing remotely exploitable buffer overflows extracted from `bind`, `wu-ftpd` and `sendmail`.

The rest of the paper is organized as follows: Section 2 presents an overview of the tools tested in this evaluation, Sections 3 and 4 present descriptions and results for two different testsuites, Section 5 describes performance overhead incurred by the tools in this evaluation, and Section 6 summarizes and discusses our findings.

2. DYNAMIC BUFFER OVERFLOW DETECTION TOOLS

This evaluation tests modern runtime buffer overflow detection tools including those that insert instrumentation at compile-time and others that wrap the binary executable directly. This section presents a short description of each tool, focusing on its strengths and weaknesses.

A summary of tool characteristics is presented in Table 1. A tool is considered to include *fine-grained bounds checking* if it can detect small (off-by-one) overflows. A tool *compiles large programs* if it can be used as a drop-in replacement for gcc and no changes to source code are needed to build the executable; however, minimal changes to the makefile are acceptable. The *time of error reporting* specifies whether the error report is generated when the error occurs or when the program terminates. Since program state is likely to become corrupted during an overflow, continuing execution after the first error may result in incorrect errors being reported. Instrumentation may also be corrupted, causing failures in error checking and reporting. If a tool can protect the program state by intercepting out-of-bounds writes before they happen and discarding them, reporting errors at termination may provide a more complete error summary.

2.1 Executable Monitoring Tools

Chaperon [16] is part of the commercial Insure toolset from Parasoft. Chaperon works directly with binary executables and thus can be used when source code is not available. It intercepts calls to `malloc` and `free` and checks heap accesses for validity. It also detects memory leaks and *read-before-write* errors. One limitation of Chaperon is that fine-grained bounds checking is provided only for heap buffers. Monitoring of buffers on the stack is very coarse. Some overflows are reported incorrectly because instrumentation can become corrupted by overflows. Like all products in the Insure toolset, it is closed-source which makes extensions difficult.

Valgrind [12] is an open-source alternative to Chaperon. It simulates code execution on a virtual x86 processor, and like Chaperon, intercepts calls to `malloc` and `free` that allow for fine-grained buffer overflow detection on the heap. After the program in simulation crashes, the error is reported and the simulator exits gracefully. Like Chaperon, Valgrind suffers from coarse stack monitoring. Also, testing is very slow (25 – 50 times slower than running the executable compiled with gcc [12]), since the execution is simulated on a virtual processor.

2.2 Compiler-based Tools

CCured [14] works by performing static analysis to determine the type of each pointer (`SAFE`, `SEQ`, or `WILD`). `SAFE` pointers can be dereferenced, but are not subject to pointer arithmetic or type casts. `SEQ` pointers can be used in pointer arithmetic, but cannot be cast to other pointer types, while `WILD` pointers can be used in a cast. Each pointer is instrumented to carry appropriate metadata at runtime - `SEQ` pointers include upper and lower bounds of the array they reference, and `WILD` pointers carry type tags. Appropriate checks are inserted into the executable based on pointer type. `SAFE` pointers are cheapest since they require only a `NULL` check, while `WILD` pointers are the most expensive, since they require type verification at runtime.

The main disadvantage of CCured is that the programmer may be required to annotate the code to help CCured determine pointer types in complex programs. Since CCured requires pointers to carry metadata, wrappers are needed to strip metadata from pointers when they pass to uninstrumented code and create metadata when pointers are received from uninstrumented code. While wrappers for commonly-used C library functions are provided with CCured, the developer will have to create wrappers to interoperate with other uninstrumented code. These wrappers introduce another source of mistakes, as wrappers for `sscanf` and `fscanf` were incorrect in the version of CCured tested in this evaluation; however, they appear to be fixed in the currently-available version (v1.3.2).

Tool	Version	OS	Requires Source	Open Source	Fine-grained Bounds Checking	Compiles Large Programs	Time of Error Reporting
Wait for segfault	N/A	Any	No	Yes	No	Yes	Termination
gcc	3.3.2	Linux	No	Yes	No	Yes	Termination
Chaperon	2.0	Linux	No	No	No*	Yes	Occurrence
Valgrind	2.0.0	Linux	No	Yes	No*	Yes	Termination
CCured	1.2.1	Linux	Yes	Yes	Yes	No	Occurrence
CRED	3.3.2	Linux	Yes	Yes	Yes	Yes	Occurrence
Insure++	6.1.3	Linux	Yes	No	Yes	Yes	Occurrence
ProPolice	2.9.5	Linux	Yes	Yes	No	Yes	Termination
TinyCC	0.9.20	Linux	Yes	Yes	Yes	No	Termination

Table 1: Summary of Tool Characteristics (* = fine-grained bounds checking on heap only)

C Range Error Detector (CRED) [19] has been developed by Ruwase and Lam, and builds on the Jones and Kelly "referent object" approach [11]. An *object tree*, containing the memory range occupied by all objects (i.e. arrays, structs and unions) in the program, is maintained during execution. When an object is created, it is added to the tree and when it is destroyed or goes out of scope, it is removed from the tree. All operations that involve pointers first locate the "referent object" – an object in the tree to which the pointer currently refers. A pointer operation is considered illegal if it results in a pointer or references memory outside said "referent object." CRED's major improvement is adhering to a more relaxed definition of the C standard – *out-of-bounds* pointers are allowed in pointer arithmetic. That is, an *out-of-bounds* pointer can be used in a comparison or to calculate and access an *in-bounds* address. This addition fixes false alarms that were generated in several programs compiled with Jones and Kelly's compiler, as pointers are frequently tested against an *out-of-bounds* pointer to determine a termination condition. CRED does not change the representation of pointers, and thus instrumented code can interoperate with unchecked code.

Two main limitations of CRED are unchecked accesses within library functions and treatment of structs and arrays as single memory blocks. The former issue is partially mitigated through wrappers of C library functions. The latter is a fundamental issue with the C standard, as casting from a struct pointer to a char pointer is allowed. When type information is readily available at compile time (i.e. the buffer enclosed in a struct is accessed via `s.buffer[i]` or `s_ptr->buffer[i]`), CRED detects overflows that overwrite other members within the struct. However, when the buffer inside a struct is accessed via an alias or through a type cast, the overflow remains undetected until the boundary of the structure is reached. These problems are common to all compiler-based tools, and are described further in Section 2.3.

Insure++ [16] is a commercial product from Parasoft and is closed-source, so we do not know about its internal workings. Insure++ examines source code and inserts instrumentation to check for memory corruption, memory leaks, memory allocation errors and pointer errors, among other things. The resulting code is executed, and errors are reported when they occur. Insure's major fault is its performance overhead, resulting in slowdown factor of up to 250 as compared to gcc. Like all tools, Insure's other limitation stems from the C standard, as it treats structs and arrays as single memory blocks. Since the product is closed-source, extensions are difficult.

ProPolice [8] is similar to StackGuard [6], and outperformed it on artificial exploits [22]. It works by inserting a "canary" value between the local variables and the stack frame whenever a function is called. It also inserts appropriate code to check that the "canary" is unaltered upon return from this function. The "canary" value is picked randomly at compile time, and extra care is taken to reorder local variables such that pointers are stored lower in memory than stack buffers.

The "canary" approach provides protection against the classic "stack smashing attack" [1]. It does not protect against overflows on the stack that consist of a single out-of-bounds write at some offset from the buffer, or against overflows on the heap. Since ProPolice only notices the error when the "canary" has changed, it does not detect read overflows or underflows. The version of ProPolice tested during this evaluation protected only functions that contained a character buffer, thus leaving overflows in buffers of other types undetected; this problem has been fixed in later versions by including `-fstack-protector-all` flag that forces a "canary" to be inserted for each function call.

Tiny C compiler (TinyCC) [2] is a small and fast C compiler developed by Fabrice Bellard. TinyCC works by inserting code to check buffer accesses at compile time; however, the representation of pointers is unchanged, so code compiled with TinyCC can interoperate with unchecked code compiled with gcc. Like CRED, TinyCC utilizes the "referent object" approach [11], but without CRED's improvements. While TinyCC provides fine-grained bounds checking of buffer accesses, it is much more limited than gcc in its capabilities. It failed to compile large programs such as `Apache` with the default makefile. It also does not detect read overflows, and terminates with a segfault whenever an overflow is encountered, without providing an error report.

2.3 Common Limitations of Compiler-based Tools

There are two issues that appear in all of the compiler-based tools – unchecked accesses within library functions and treatment of structs and arrays as single memory blocks. The former problem is partially mitigated by creating wrappers for C library functions or completely reimplementing them. Creating these wrappers is error-prone, and many functions (such as File I/0) cannot be wrapped.

The latter problem is a fundamental issue with the C stan-

dard of addressing memory in arrays and structs. According to the C standard, a pointer to any object type can be cast to a pointer to any other object type. The result is defined by implementation, unless the original pointer is suitably aligned to use as a resultant pointer [17]. This allows the program to re-interpret the boundaries between struct members or array elements; thus, the only way to handle the situation correctly is to treat structs and arrays as single memory objects. Unfortunately, overflowing a buffer inside a struct can be exploited in a variety of attacks, as the same struct may contain a number of exploitable targets, such as a function pointer, a pointer to a `longjmp` buffer or a flag that controls some aspect of program flow.

3. VARIABLE-OVERFLOW TESTSUITE EVALUATION

The *variable-overflow* testsuite evaluation is the first of two evaluations included in this paper. This testsuite is a collection of 55 small C programs that contain buffer overflows and underflows, adapted from Misha Zitser's evaluation of static analysis tools [24]. Each test case contains either a *discrete* or a *continuous* overflow. A *discrete* buffer overflow is defined as an out-of-bounds write that results from a single buffer access, which may affect up to 8 bytes of memory, depending on buffer type. A *continuous* buffer overflow is defined as an overflow resulting from multiple consecutive writes, one or more of which is out-of-bounds. Such an overflow may affect an arbitrary amount of memory (up to 4096 bytes in this testsuite), depending on buffer type and length of overflow.

Each test case in the variable-overflow testsuite contains a 200-element buffer. The overflow amount is controlled at runtime via a command-line parameter and ranges from 0 to 4096 bytes. Many characteristics of buffer overflows vary. Buffers differ in type (`char`, `int`, `float`, `func *`, `char *`) and location (stack, heap, data, bss). Some are in containers (struct, array, union, array of structs) and elements are accessed in a variety of ways (index, pointer, function, array, linear and non-linear expression). Some test cases include runtime dependencies caused by file I/O and reading from environment variables. Several common C library functions (`(f)gets`, `(fs)scanf`, `fread`, `fwrite`, `sprintf`, `str(n)cpy`, `str(n)cat`, and `memcpy`) are also used in test cases.

3.1 Test Procedure

Each test case was compiled with each tool, when required, and then executed with overflows ranging from 0 to 4096 bytes. A 0-byte overflow is used to verify a lack of false alarms, while the others test the tool's ability to detect small and large overflows. The size of a memory page on the Linux system used for testing is 4096 bytes, so an overflow of this size ensures a read or write off the stack page, which should segfault if not caught properly. Whenever the test required it, an appropriately sized file, input stream or environment variable was provided by the testing script. There are three possible outcomes of a test. A *detection* signifies that the tool recognized the overflow and returned an error message. A *segfault* indicates an illegal read or write (or an overflow detection in TinyCC). Finally, a *miss* signifies that the program returned as if no overflow occurred.

Table 1 describes the versions of tools tested in our evaluation. All tests were performed on a Red Hat Linux release 9 (Shrike) system with dual 2.66GHz Xeon CPUs. The standard Red Hat Linux kernel was modified to ensure that the location of the stack with respect to *stacktop* address (`0xC0000000`) remained unchanged between executions. This modification was necessary to ensure consistent segfault behavior due to large overflows.

3.2 Variable-overflow Testsuite Results

This section presents a summary of the results obtained with the variable-overflow testsuite. The graph in Figure 2 shows the fraction of test cases in the variable-overflow testsuite with a non-*miss* (*detection* or *segfault*) outcome for each amount of overflow. Higher fractions represents better performance. All test cases, with the exception of the 4 underflow test cases, are included on this graph even though the proportional composition of the testsuite is not representative of existing exploits. Nonetheless, the graph gives a good indication of tool performance. Fine-grained bounds checking tools are highlighted by the "fine-grained" box at the top of the graph.

The top performing tools are Insure++, CCured and CRED, which can detect small and large overflows in different memory locations. TinyCC also performs well on both heap and stack-based overflows, while ProPolice only detects continuous overflows and small discrete overflows on the stack. Since the proportion of stack-based overflows is larger than that of heap-based overflows in our testsuite, ProPolice is shown to have a relatively high fraction of detections. Chaperon and Valgrind follow the same shape as gcc, since these tools only provide fine-grained detection of overflows on the heap. This ability accounts for their separation from gcc on the graph.

As the graph demonstrates, only tools with fine-grained bounds checking, such as Insure++, CCured and CRED are able to detect small overflows, including off-by-one overflows, which can still be exploitable. For tools with coarse stack monitoring, a large increase in detections/segfaults occurs at the overflow of 21 bytes, which corresponds to overwriting the return instruction pointer. The drop after the next 4 bytes corresponds to the discrete overflow test cases, as they no longer cause a segfault behavior. ProPolice exhibits the same behavior for overflows of 9–12 bytes due to a slightly different stack layout. Tools with fine-grained bounds checking also perform better in detecting discrete overflows and thus do not exhibit these fluctuations. For very large overflows, all tools either detect the overflow or segfault, which results in fraction of non-*miss* outcomes close to 1, as shown on the far right side of the graph.

4. REAL EXPLOIT EVALUATION

Previously, we evaluated the ability of a variety of tools employing *static analysis* to detect buffer overflows [25]. These tools ranged from simple lexical analyzers to abstract interpreters [9, 10, 20, 21, 23]. We chose to test these tools against fourteen historic vulnerabilities in the popular Internet servers `bind`, `sendmail`, and `wu-ftpd`. Many of the detectors were unable to process the entire source for these programs. We thus created *models* of a few hundred lines that reproduced most of the complexity present in the original. Further, for each model, we created a patched copy in which we verified that the overflow did not exist for a test input that triggered the error in the unpatched version. In that evaluation, we found that current static analysis tools

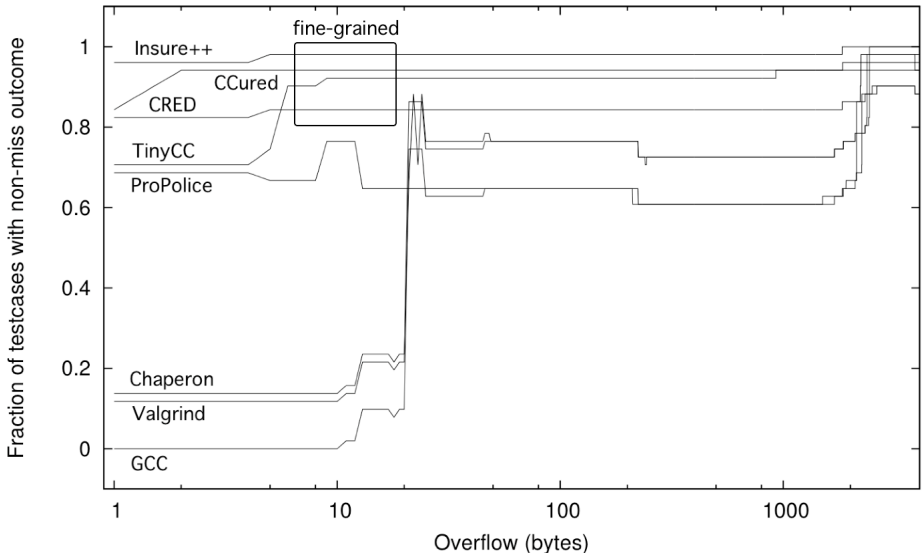


Figure 2: Combined fraction of detections and segfaults vs the amount of overflow in bytes. A box highlights tools with fine-grained bounds checking capabilities.

either missed too many of these vulnerable buffer overflows or signaled too many false alarms to be useful. Here, we report results for seven dynamic overflow detectors on that same set of fourteen models of historic vulnerabilities. This provides a prediction of their performance on real overflows that occur in open-source servers.

4.1 Test Procedure

During testing, each unpatched model program was compiled with the tool (if necessary) and executed on an input that is known to trigger the overflow. A *detection* signifies that the tool reported an overflow, while a *miss* indicates that the program executed as if no overflow occurred. A patched version of the model program was then executed on the same input. A *false alarm* was recorded if the instrumented program still reported a buffer overflow.

4.2 Real Exploit Results

Table 2 presents the results of this evaluation, which agree well with those on the variable-overflow testsuite. Three of the dynamic overflow detectors that provide fine-grained bounds checking, CCured, CRED, and TinyCC, work extremely well, detecting about 90% of the overflows whilst raising only one false alarm each. The commercial program Insure, which also checks bounds violations rigorously, fares somewhat worse with both fewer detections and more false alarms. Notice that misses and false alarms for these tools are errors in the implementation, and are in no way a fundamental limitation of dynamic approaches. For example, in the case of CRED the misses are due to an incorrect `memcpy` wrapper; there are no misses once this wrapper is corrected. The CRED false alarm is the result of overly aggressive string length checks included in the wrappers for string manipulation functions such as `strchr`. None of the tools are given credit for a segmentation fault as a signal of buffer overflow (except TinyCC and gcc as this is the only signal provided). This is why, for instance, ProPolice appears to perform *worse* than gcc. As a final comment, it is worth considering the performance of gcc alone. If provided with the right input, the program itself detects almost half of these real overflows, indicating that input generation may be a fruitful area of future research.

5. PERFORMANCE OVERHEAD

The goals of the performance overhead evaluation are twofold. One is to quantify the slowdown caused by using dynamic buffer overflow detection tools instead of gcc when executing some commonly used programs. The other is to test each tool's ability to compile and monitor a complex program. In addition, this evaluation shows whether the tool can be used as a drop-in replacement for gcc, without requiring changes to the source code. Minimal modifications to the makefile are allowed, however, to accommodate the necessary options for the compilation process.

Our evaluation tests overhead on two common utility programs (`gzip` and `tar`), an encryption library (`OpenSSL`) and a webserver (`Apache`). For `OpenSSL` and `tar`, the testsuites included in the distribution were used. The test for `gzip` consisted of compressing a tar archive of the source code package for glibc (around 17MB in size). The test for `Apache` consisted of downloading a 6MB file 1,000 times on a loopback connection. The overhead was determined by timing the execution using `time` and comparing it to executing the test when the program is compiled with gcc. The results are summarized in Table 3. Programs compiled with gcc executed the tests in 7.2s (`gzip`), 5.0s (`tar`), 16.9s (`OpenSSL`) and 38.8s (`Apache`).

Compiling and running `Apache` presented the most problems. Chaperon requires a separate license for multi-threaded programs, so we were unable to evaluate its overhead. Valgrind claims to support multi-threaded programs but failed to run due to a missing library. Insure++ failed on the configuration step of the makefile and thus was unable to

	Chaperon	Valgrind	CCured	CRED	gcc	Insure++	ProPolice	TinyCC
b1		d	d		d			d
b2		d	d		d			d
b3			d	d	d	d	d	d
b4	df	df	d	d	d	df	d	df
f1			d	d		d		d
f2			d	df		df		d
f3			d	d		d		d
s1			d	d		d		d
s2			d	d		df		d
s3			d	d				d
s4			d	d				d
s5			d	d	d	df	d	d
s6			df	d		d		
s7	d	d		d	d	d		d
$P(det)$	0.14	0.29	0.93	0.86	0.43	0.71	0.21	0.93
$P(fa)$	0.07	0.07	0.07	0.07	0	0.29	0	0.07

Table 2: Dynamic buffer overflow detection in 14 models of real vulnerabilities in open source server code. There are four `bind` models (b1–b4), three `wu-ftpd` models (f1–f3), and seven `sendmail` models (s1–s7). A 'd' indicates a tool detected a historic overflow, while an 'f' means the tool generated a false alarm on the patched version. $P(det)$ and $P(fa)$ are the fraction of model programs for which a tool signals a detection or false alarm, respectively.

Tool	gzip	tar	OpenSSL	Apache
Chaperon	75.6		61.8	
Valgrind	18.6	73.1	44.8	
CCured				
CRED	16.6	1.4	29.3	1.1
Insure++	250.4	4.7	116.6	
ProPolice	1.2	1.0	1.1	1.0
TinyCC				

Table 3: Instrumentation overhead for commonly used programs as a multiple of gcc execution time. The blank entries indicate that the program could not be compiled or executed with the corresponding tool.

compile `Apache`. CCured likewise failed at configuration, while TinyCC failed in parsing one of the source files during the compilation step.

The performance overhead results demonstrate some important limitations of dynamic buffer overflow detection tools. Insure++ is among the best performers on the variable-overflow testsuite; however, it incurs very high overhead. CCured and TinyCC, which performed well on both the variable-overflow testsuite and the model programs, cannot compile these programs without modifications to source code. CCured requires the programmer to annotate sections of the source code to resolve constraints involving what the tools considers "bad casts," while TinyCC includes a C parser that is likely incomplete or incorrect.

While CRED incurs large overhead on programs that involve many buffer manipulations, it has the smallest overhead for a fine-grained bounds checking tool. CRED can be used as a drop-in replacement for gcc, as it requires no changes to the source code in order to compile these programs. Only minimal changes to the makefile were required to enable bounds-checking and turn off optimizations. CRED's high detection rate, ease of use and relatively small overhead make it the best candidate for use in a comprehensive solution for dynamic buffer overflow detection.

6. DISCUSSION

The three top-performing tools in our evaluation are Insure++, CCured and CRED. Insure++ performs well on test cases, but not on model programs. It adds a large performance overhead, and the closed-source nature of the tool inhibits extensions. CCured shows a high detection rate and is open-source; however, it requires rewriting 1–2% of code to compile complicated programs [4]. CRED also offers a high detection rate, and it is open-source, easily extensible and has fairly low performance overhead (10% slowdown for simple Apache loopback test). Its main disadvantage is lack of overflow detection in library functions compiled without bounds-checking. Like all compiler-based tools, CRED does not detect overflows within structs in a general case; however, if the buffer enclosed in a struct is referenced directly, then CRED detects the overflow.

As this study demonstrates, several features are crucial to the success of a dynamic buffer overflow detection tool. Memory monitoring must be done on a fine-grained basis, as this is the only way to ensure that discrete writes and off-by-one overflows are caught. Buffer overflows in library functions, especially file I/O, often go undetected. Some tools solve this problem by creating wrappers for library functions, which is a difficult and tedious task. Recompiling libraries with the bounds-checking tool may be a better alternative, even if it should entail a significant slowdown. Error reporting is likewise essential in determining the cause of the problem because segfaults alone provide little information. Since instrumentation and messages can get corrupted by large overflows, the error should be reported immediately after the overflow occurs.

Of all the tools surveyed, CRED shows the most promise as a part of a comprehensive dynamic testing solution. It offers fine-grained bounds checking, provides comprehensive error reports, compiles large programs and incurs reasonable performance overhead. It is also open-source and thus easily extensible. CRED is likewise useful for regression testing to find latent buffer overflows and for determining the cause of segfault behavior.

7. REFERENCES

[1] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 7(47), 1998.

[2] F. Bellard. TCC: Tiny C compiler. http://www.tinycc.org, Oct. 2003.

[3] CERT. CERT/CC statistics. http://www.cert.org/stats/cert_stats.html, Feb. 2005.

[4] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244. ACM Press, 2003.

[5] C. Cowan. Software security for open-source systems. *IEEE Security & Privacy*, 1(1):38–45, 2003.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.

[7] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, Dec. 2004.

[8] H. Etoh. GCC extension for protecting applications from stack smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, Dec. 2003.

[9] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[10] G. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002*, Pasadena, CA, USA, June 2002.

[11] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–25, 1997.

[12] N. N. Julian Seward and J. Fitzhardinge. Valgrind: A GPL'd system for debugging and profiling x86-linux programs. http://valgrind.kde.org, 2004.

[13] R. Kaksonen. A functional method for assessing protocol implementation security. Publication 448, VTT Electronics, Telecommunication Systems, Kaitoväylä 1, PO Box 1100, FIN-90571, Oulu, Finland, Oct. 2001.

[14] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[15] NIST. ICAT vulnerability statistics. http://icat.nist.gov/icat.cfm?function=statistics, Feb. 2005.

[16] Parasoft. Insure++: Automatic runtime error detection. http://www.parasoft.com, 2004.

[17] P. Plauger and J. Brodie. *Standard C*. PTR Prentice Hall, Englewood Cliffs, NJ, 1996.

[18] E. Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[19] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of Network and Distributed System Security Symposium*, pages 159–169, 2004.

[20] P. Technologies. PolySpace C verifier. http://www.polyspace.com/c.htm, Sept. 2001.

[21] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb. 2000.

[22] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, Feb. 2003.

[23] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

[24] M. Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Aug. 2003.

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

Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools*

Kendra Kratkiewicz
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108
Phone: 781-981-2931
Email: KENDRA@LL.MIT.EDU

Richard Lippmann
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108
Phone: 781-981-2711
Email: LIPPMANN@LL.MIT.EDU

ABSTRACT
A corpus of 291 small C-program test cases was developed to evaluate static and dynamic analysis tools designed to detect buffer overflows. The corpus was designed and labeled using a new, comprehensive buffer overflow taxonomy. It provides a benchmark to measure detection, false alarm, and confusion rates of tools, and also suggests areas for tool enhancement. Experiments with five tools demonstrate that some modern static analysis tools can accurately detect overflows in simple test cases but that others have serious limitations. For example, PolySpace demonstrated a superior detection rate, missing only one detection. Its performance could be enhanced if extremely long run times were reduced, and false alarms were eliminated for some C library functions. ARCHER performed well with no false alarms whatsoever. It could be enhanced by improving inter-procedural analysis and handling of C library functions. Splint detected significantly fewer overflows and exhibited the highest false alarm rate. Improvements in loop handling and reductions in false alarm rate would make it a much more useful tool. UNO had no false alarms, but missed overflows in roughly half of all test cases. It would need improvement in many areas to become a useful tool. BOON provided the worst performance. It did not detect overflows well in string functions, even though this was a design goal.

Categories and Subject Descriptors
D.2.4 [Software Engineering] Software/Program Verification, D.2.5 [Software Engineering] Testing and Debugging, K.4.4 [Computers and Society] Electronic Commerce Security.

General Terms
Measurement, Performance, Security, Verification.

Keywords
Security, buffer overflow, static analysis, evaluation, exploit, test, detection, false alarm, source code.

1. INTRODUCTION
Ideally, developers would discover and fix errors in programs before they are released. This, however, is an extremely difficult task. Among the many approaches to finding and fixing errors, static analysis is one of the most attractive. The goal of static analysis is to automatically process source code and analyze all code paths without requiring the large numbers of test cases used in dynamic testing. Over the past few years, static analysis tools have been developed to discover buffer overflows in C code.

Buffer overflows are of particular interest as they are potentially exploitable by malicious users, and have historically accounted for a significant percentage of the software vulnerabilities published each year [18, 20], such as in NIST's ICAT Metabase [9], CERT advisories [1], Bugtraq [17], and other security forums. Buffer overflows have also been the basis for many damaging exploits, such as the Sapphire/Slammer [13] and Blaster [15] worms.

A buffer overflow vulnerability occurs when data can be written outside the memory allocated for a buffer, either past the end or before the beginning. Buffer overflows may occur on the stack, on the heap, in the data segment, or the BSS segment (the memory area a program uses for uninitialized global data), and may overwrite from one to many bytes of memory outside the buffer. Even a one-byte overflow can be enough to allow an exploit [10]. Buffer overflows have been described at length in many papers, including [20], and many descriptions of exploiting buffer overflows can be found online.

This paper focuses on understanding the capabilities of static analysis tools designed to detect buffer overflows in C code. It extends a study by Zitser [20, 21] that evaluated the ability of several static analysis tools to detect fourteen known, historical vulnerabilities (all buffer overflows) in open-source software. The Zitser study first found that only one of the tools could analyze large, open-source C programs. To permit an evaluation, short, but often complex, model programs were extracted from the C programs and used instead of the original, much longer programs. Five static analysis tools were run on model programs with and without overflows: ARCHER [19], BOON [18], Splint [6, 12], UNO [8], and PolySpace C Verifier [14]. All use static analysis techniques, including symbolic analysis, abstract interpretation, model checking, integer range analysis, and inter-procedural analysis. Results were not encouraging. Only one of the five tools performed statistically better than random guessing. Not only did the tools fail to detect a significant number of overflows, but they also produced a large number of false alarms, indicating overflows where none actually existed. Equally discouraging were the confusion rates, reflecting the number of cases where a tool reports an error in both the vulnerable and patched versions of a program.

2005 Workshop on the Evaluation of Software Defect Detection Tools
2005,June 12, Chicago, IL

*This work was sponsored by the Advanced Research and Development Activity under Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Given the small number of model programs, and the fact that buffer overflows were embedded in complex code, it is difficult to draw conclusions concerning why the tools performed poorly. This paper describes a follow-on analysis of the five tools evaluated in the previous study. It's simpler but broader, and more diagnostic test cases are designed to determine specific strengths and weaknesses of tools. Although this research evaluated only static analysis tools, it provides a taxonomy and test suite useful for evaluating dynamic analysis tools as well.

2. BUFFER OVERFLOW TAXONOMY

Using a comprehensive taxonomy makes it possible to develop test cases that cover a wide range of buffer overflows and make diagnostic tool assessments. Zitser developed a taxonomy containing thirteen attributes [20]. This taxonomy was modified and expanded to address problems encountered with its application, while still attempting to keep it small and simple enough for practical application. The new taxonomy consists of twenty-two attributes listed in Table 1.

Table 1. Buffer Overflow Taxonomy Attributes

Attribute Number	Attribute Name
1	Write/Read
2	Upper/Lower Bound
3	Data Type
4	Memory Location
5	Scope
6	Container
7	Pointer
8	Index Complexity
9	Address Complexity
10	Length/Limit Complexity
11	Alias of Buffer Address
12	Alias of Buffer Index
13	Local Control Flow
14	Secondary Control Flow
15	Loop Structure
16	Loop Complexity
17	Asynchrony
18	Taint
19	Runtime Environment Dependence
20	Magnitude
21	Continuous/Discrete
22	Signed/Unsigned Mismatch

Details on the possible values for each attribute are available in [11], and are summarized below. For each attribute, the possible values are listed in ascending order (i.e. the 0 value first).

Write/Read: describes the type of memory access (write, read). While detecting illegal writes is probably of more interest in preventing buffer overflow exploits, it is possible that illegal reads could allow unauthorized access to information or could constitute one operation in a multi-step exploit.

Upper/Lower Bound: describes which buffer bound is violated (upper, lower). While the term "buffer overflow" suggests an access beyond the upper bound of a buffer, it is equally possible to underflow a buffer, or access below its lower bound (e.g. `buf[-1]`).

Data Type: indicates the type of data stored in the buffer (character, integer, floating point, wide character, pointer, unsigned character, unsigned integer). Character buffers are often manipulated with unsafe string functions in C, and some tools may focus on detecting overflows of those buffers; buffers of all types may be overflowed, however, and should be analyzed.

Memory Location: indicates where the buffer resides (stack, heap, data region, BSS, shared memory). Non-static variables defined locally to a function are on the stack, while dynamically allocated buffers (e.g., those allocated by calling a malloc function) are on the heap. The data region holds initialized global or static variables, while the BSS region contains uninitialized global or static variables. Shared memory is typically allocated, mapped into and out of a program's address space, and released via operating system specific functions. While a typical buffer overflow exploit may strive to overwrite a function return value on the stack, buffers in other locations have been exploited and should be considered as well.

Scope: describes the difference between where the buffer is allocated and where it is overrun (same, inter-procedural, global, inter-file/inter-procedural, inter-file/global). The scope is the same if the buffer is allocated and overrun within the same function. Inter-procedural scope describes a buffer that is allocated in one function and overrun in another function within the same file. Global scope indicates that the buffer is allocated as a global variable, and is overrun in a function within the same file. The scope is inter-file/inter-procedural if the buffer is allocated in a function in one file, and overrun in a function in another file. Inter-file/global scope describes a buffer that is allocated as a global in one file, and overrun in a function in another file. Any scope other than "same" may involve passing the buffer address as an argument to another function; in this case, the *Alias of Buffer Address* attribute must also be set accordingly. Note that the test suite used in this evaluation does not contain an example for "inter-file/global."

Container: indicates whether the buffer resides in some type of container (no, array, struct, union, array of structs, array of unions). The ability of static analysis tools to detect overflows within containers (e.g., overrunning one array element into the next, or one structure field into the next) and beyond container boundaries (i.e., beyond the memory allocated for the container as a whole) may vary according to how the tools model these containers and their contents.

Pointer: indicates whether the buffer access uses a pointer dereference (no, yes). Note that it is possible to use a pointer dereference with or without an array index (e.g. `*pBuf` or `(*pBuf)[10]`); the *Index Complexity* attribute must be set accordingly. In order to know if the memory location referred to by a dereferenced pointer is within buffer bounds, a code analysis tool must keep track of what pointers point to; this points-to analysis is a significant challenge.

Index Complexity: indicates the complexity of the array index (constant, variable, linear expression, non-linear expression, function return value, array contents, N/A). This attribute applies only to the user program, and is not used to describe how buffer accesses are performed inside C library functions.

Address Complexity: describes the complexity of the address or pointer computation (constant, variable, linear expression, non-linear expression, function return value, array contents). Again, this attribute is used to describe the user program only, and is not applied to C library function internals.

Length/Limit Complexity: indicates the complexity of the length or limit passed to a C library function that overruns the buffer (N/A, none, constant, variable, linear expression, non-linear expression, function return value, array contents). "N/A" is used when the test case does not call a C library function to overflow the buffer, whereas "none" applies when a C library function overflows the buffer, but the function does not take a length or limit parameter (e.g. `strcpy`). The remaining attribute values apply to the use of C library functions that do take a length or limit parameter (e.g. `strncpy`). Note that if a C library function overflows the buffer, the overflow is by definition inter-file/inter-procedural in scope, and involves at least one alias of the buffer address. In this case, the *Scope* and *Alias of Buffer Address* attributes must be set accordingly. Code analysis tools may need to provide their own wrappers for or models of C library functions in order to perform a complete analysis.

Alias of Buffer Address: indicates if the buffer is accessed directly or through one or two levels of aliasing (no, one, two). Assigning the original buffer address to a second variable and subsequently using the second variable to access the buffer constitutes one level of aliasing, as does passing the original buffer address to a second function. Similarly, assigning the second variable to a third and accessing the buffer through the third variable would be classified as two levels of aliasing, as would passing the buffer address to a third function from the second. Keeping track of aliases and what pointers point to is a significant challenge for code analysis tools.

Alias of Buffer Index: indicates whether or not the index is aliased (no, one, two, N/A). If the index is a constant or the results of a computation or function call, or if the index is a variable to which is directly assigned a constant value or the results of a computation or function call, then there is no aliasing of the index. If, however, the index is a variable to which the value of a second variable is assigned, then there is one level of aliasing. Adding a third variable assignment increases the level of aliasing to two. If no index is used in the buffer access, then this attribute is not applicable.

Local Control Flow: describes what kind of program control flow most immediately surrounds or affects the overflow (none, if, switch, cond, goto/label, setjmp/longjmp, function pointer, recursion). For the values "if", "switch", and "cond", the buffer overflow is located within the conditional construct. "Goto/label" signifies that the overflow occurs at or after the target label of a goto statement. Similarly, "setjmp/longjmp" means that the overflow is at or after a longjmp address. Buffer overflows that occur within functions reached via function pointers are assigned the "function pointer" value, and those within recursive functions receive the value "recursion". The values "function pointer" and "recursion" necessarily imply a global or inter-procedural scope, and may involve an address alias. The *Scope* and *Alias of Buffer Address* attributes should be set accordingly.

Control flow involves either branching or jumping to another context within the program; hence, only path-sensitive code analysis can determine whether or not the overflow is actually reachable. A code analysis tool must be able to follow function pointers and have techniques for handling recursive functions in order to detect buffer overflows with the last two values for this attribute.

Secondary Control Flow: has the same values as *Local Control Flow*, the difference being the location of the control flow construct. *Secondary Control Flow* either precedes the overflow or contains nested, local control flow. Some types of secondary control flow may occur without any local control flow, but some may not. The *Local Control Flow* attribute should be set accordingly.

The following example illustrates an `if` statement that precedes the overflow and affects whether or not it occurs. Because it precedes the overflow, as opposed to directly containing the overflow, it is labeled as secondary, not local, control flow.

```
int main(int argc, char *argv[])
{
  char buf[10];
  int i = 10;

  if (i > 10)
  {
    return 0;
  }

  /*  BAD  */
  buf[i] = 'A';

  return 0;
}
```

Only control flow that affects whether or not the overflow occurs is classified. In other words, if a preceding control flow construct has no bearing on whether or not the subsequent overflow occurs, it is not considered to be secondary control flow, and this attribute would be assigned the value "none."

The following example illustrates nested control flow. The inner `if` statement directly contains the overflow, and we assign the value "if" to the *Local Control Flow* attribute. The outer `if` statement represents secondary control flow, and we assign the value "if" to the *Secondary Control Flow* attribute as well.

```
int main(int argc, char *argv[])
{
  char buf[10];
  int i = 10;

  if (sizeof buf <= 10)
  {
    if (i <= 10)
    {
      /*  BAD  */
      buf[i] = 'A';
    }
  }

  return 0;
}
```

Some code analysis tools perform path-sensitive analyses, and some do not. Even those that do often must make simplifying approximations in order to keep the problem tractable and the

solution scalable. This may mean throwing away some information, and thereby sacrificing precision, at points in the program where previous branches rejoin. Test cases containing secondary control flow may highlight the capabilities or limitations of these varying techniques.

Loop Structure: describes the type of loop construct within which the overflow occurs (none, standard for, standard do-while, standard while, non-standard for, non-standard do-while, non-standard while). A "standard" loop is one that has an initialization, a loop exit test, and an increment or decrement of a loop variable, all in typical format and locations. A "non-standard" loop deviates from the standard loop in one or more of these areas. Examples of standard `for`, `do-while`, and `while` loops are shown below, along with one non-standard `for` loop example:

Standard `for` loop:

```
for (i=0; i<11; i++)
{
    buf[i] = 'A';
}
```

Standard `do-while` loop:

```
i=0;
do
{
    buf[i] = 'A';
    i++;
} while (i<11);
```

Standard `while` loop:

```
i=0;
while (i<11)
{
    buf[i] = 'A';
     i++;
}
```

A non-standard `for` loop:

```
for (i=0; i<11; )
{
    buf[i++] = 'A';
}
```

Non-standard loops may necessitate secondary control flow (such as additional if statements). In these cases, the *Secondary Control Flow* attribute should be set accordingly. Any value other than "none" for this attribute requires that the *Loop Complexity* attribute be set to something other than "not applicable."

Loops may execute for a large number or even an infinite number of iterations, or may have exit criteria that depend on runtime conditions; therefore, it may be impossible or impractical for static analysis tools to simulate or analyze loops to completion. Different tools have different methods for handling loops; for example, some may attempt to simulate a loop for a fixed number of iterations, while others may employ heuristics to recognize and handle common loop constructs. The approach taken will likely affect a tool's capabilities to detect overflows that occur within various loop structures.

Loop Complexity: indicates how many loop components (initialization, test, increment) are more complex than the standard baseline of initializing to a constant, testing against a constant, and incrementing or decrementing by one (N/A, none, one, two, three). Of interest here is whether or not the tools handle loops with varying complexity in general, rather than which particular loop components are handled or not.

Asynchrony: indicates if the buffer overflow is potentially obfuscated by an asynchronous program construct (no, threads, forked process, signal handler). The functions that may be used to realize these constructs are often operating system specific (e.g. on Linux, `pthread` functions; `fork`, `wait`, and `exit`; and `signal`). A code analysis tool may need detailed, embedded knowledge of these constructs and the O/S-specific functions in order to properly detect overflows that occur only under these special circumstances.

Taint: describes whether a buffer overflow may be influenced externally (no, argc/argv, environment variables, file read or stdin, socket, process environment). The occurrence of a buffer overflow may depend on command line or stdin input from a user, the value of environment variables (e.g. `getenv`), file contents (e.g. `fgets`, `fread`, or `read`), data received through a socket or service (e.g. `recv`), or properties of the process environment, such as the current working directory (e.g. `getcwd`). All of these can be influenced by users external to the program, and are therefore considered "taintable." These may be the most crucial overflows to detect, as it is ultimately the ability of the external user to influence program operation that makes exploits possible. As with asynchronous constructs, code analysis tools may require detailed modeling of O/S-specific functions in order to properly detect related overflows. Note that the test suite used in this evaluation does not contain an example for "socket."

Runtime Environment Dependence: indicates whether or not the occurrence of the overrun depends on something determined at runtime (no, yes). If the overrun is certain to occur on every execution of the program, it is not dependent on the runtime environment; otherwise, it is.

Magnitude: indicates the size of the overflow (none, 1 byte, 8 bytes, 4096 bytes). "None" is used to classify the "OK" or patched versions of programs that contain overflows. One would expect static analysis tools to detect buffer overflows without regard to the size of the overflow, unless they contain an off-by-one error in their modeling of library functions. The same is not true of dynamic analysis tools that use runtime instrumentation to detect memory violations; different methods may be sensitive to different sizes of overflows, which may or may not breach page boundaries, etc. The various overflow sizes were chosen with future dynamic tool evaluations in mind. Overflows of one byte test both the accuracy of static analysis modeling, and the sensitivity of dynamic instrumentation. Eight and 4096 byte overflows are aimed more exclusively at dynamic tool testing, and are designed to cross word-aligned and page boundaries.

Continuous/Discrete: indicates whether the buffer overflow jumps directly out of the buffer (discrete) or accesses consecutive elements within the buffer before overflowing past the bounds (continuous). Loop constructs are likely candidates for containing continuous overflows. C library functions that overflow a buffer while copying memory or string contents into it demonstrate continuous overflows. An overflow labeled as continuous should have the loop-related attributes or the Length Complexity attribute (indicating the complexity of the length or limit passed to a C library function) set accordingly. Some dynamic tools rely

on “canaries” at buffer boundaries to detect continuous overflows [5], and therefore may miss discrete overflows.

Signed/Unsigned Mismatch: indicates if the buffer overflow is caused by using a signed or unsigned value where the opposite is expected (no, yes). Typically, a signed value is used where an unsigned value is expected, and gets interpreted as a very large unsigned or positive value, causing an enormous buffer overflow.

This taxonomy is specifically designed for developing simple diagnostic test cases. It may not fully characterize complex buffer overflows that occur in real code, and specifically omits complex details related to the overflow context.

For each attribute (except for Magnitude), the zero value is assigned to the simplest or “baseline” buffer overflow, shown below:

```
int main(int argc, char *argv[])
{
  char buf[10];
  /*  BAD  */
  buf[10] = 'A';
  return 0;
}
```

Each test case includes a comment line as shown with the word “BAD” or “OK.” This comment is placed on the line before the line where an overflow might occur and it indicates whether an overflow does occur. The buffer access in the baseline program is a write operation beyond the upper bound of a stack-based character buffer that is defined and overflowed within the same function. The buffer does not lie within another container, is addressed directly, and is indexed with a constant. No C library function is used to access the buffer, the overflow is not within any conditional or complicated control flows or asynchronous program constructs, and does not depend on the runtime environment. The overflow writes to a discrete location one byte beyond the buffer boundary, and cannot be manipulated by an external user. Finally, it does not involve a signed vs. unsigned type mismatch.

Appending the value digits for each of the twenty-two attributes forms a string that classifies a buffer overflow, which can be referred to during results analysis. For example, the sample program shown above is classified as “00000000000000000100.” The single “1” in this string represents a “Magnitude” attribute indicating a one-byte overflow. This classification information appears in comments at the top of each test case file, as shown in the example below:

```
/* Taxonomy Classification: 00000000000000000000 */

/*
 * WRITE/READ                0   write
 * WHICH BOUND               0   upper
 * DATA TYPE                 0   char
 * MEMORY LOCATION           0   stack
 * SCOPE                     0   same
 * CONTAINER                 0   no
 * POINTER                   0   no
 * INDEX COMPLEXITY          0   constant
 * ADDRESS COMPLEXITY        0   constant
 * LENGTH COMPLEXITY         0   N/A
 * ADDRESS ALIAS             0   none
 * INDEX ALIAS               0   none
 * LOCAL CONTROL FLOW        0   none
 * SECONDARY CONTROL FLOW    0   none
 * LOOP STRUCTURE            0   no
 * LOOP COMPLEXITY           0   N/A
 * ASYNCHRONY                0   no
 * TAINT                     0   no
 * RUNTIME ENV. DEPENDENCE   0   no
 * MAGNITUDE                 0   no overflow
 * CONTINUOUS/DISCRETE       0   discrete
 * SIGNEDNESS                0   no
 */
```

While the Zitser test cases were program pairs consisting of a bad program and a corresponding patched program, this evaluation uses program quadruplets. The four versions of each test case correspond to the four possible values of the Magnitude attribute; one of these represents the patched program (no overflow), while the remaining three indicate buffer overflows of one, eight, and 4096 bytes denoted as minimum, medium, and large overflows.

3. TEST SUITE

A full discussion of design considerations in creating test cases is provided in [11]. Goals included avoiding tool bias; providing samples that cover the taxonomy; measuring detections, false alarms, and confusions; naming and documenting test cases to facilitate automated scoring and encourage reuse; and maintaining consistency in programming style and use of programming idioms.

Ideally, the test suite would have at least one instance of each possible buffer overflow that could be described by the taxonomy. Unfortunately, this is completely impractical. Instead, a “basic” set of test cases was built by first choosing a simple, baseline example of a buffer overflow, and then varying its characteristics one at a time. This strategy results in taxonomy coverage that is heavily weighted toward the baseline attribute values. Variations were added by automated code-generation software that produces C code for the test cases to help insure consistency and make it easier to add test cases.

Four versions of 291 different test cases were generated with no overflow and with minimum, medium, and large overflows. Each test case was compiled with gcc, the GNU C compiler [7], on Linux to verify that the programs compiled without warnings or errors (with the exception of one test case that produces an unavoidable warning). Overflows were verified using CRED, a fine-grained bounds-checking extension to gcc that detects overflows at run time [16], or by verifying that the large overflow caused a segfault. A few problems with test cases that involved complex loop conditions were also corrected based on initial results produced by the PolySpace tool.

4. TEST PROCEDURES

The evaluation consisted of analyzing each test case (291 quadruplets), one at a time using the five static analysis tools (ARCHER, BOON, PolySpace, Splint, and UNO), and collecting tool outputs. Tool-specific Perl programs parsed the output and determined whether a buffer overflow was detected on the line immediately following the comment in each test case. Details of

the test procedures are provided in [11]. No annotations were added and no modifications were made to the source code for any tool.

Since BOON does not report line numbers for the errors, automated tabulation cannot validate that the reported error corresponds to the commented buffer access in the test case file. Instead, it assumes that any reported error is a valid detection. Therefore, BOON detections and false alarms were further inspected manually to verify their accuracy, and some were dismissed (two detections and two false alarms) since they did not refer to the buffer access in question.

Special handling was required for PolySpace in cases where the buffer overflow occurs in a C library function. PolySpace reports the error in the library function itself, rather than on the line in the test case file where the function is called. Therefore, the results tabulator looks for errors reported in the called library function and counts those detections irrespective of the associated line number. Additionally, one test case involving wide characters required additional command-line options to work around errors reported when processing wctype.h.

5. RESULTS AND ANALYSIS

All five static analysis tools performed the same regardless of overflow size (this would not necessarily hold for dynamic analysis). To simplify the discussion, results for the three magnitudes of overflows are thus reported as results for "bad" test cases as a whole.

Table 2 shows the performance metrics computed for each tool. The detection rate indicates how well a tool detects the known buffer overflows in the bad programs, while the false alarm rate indicates how often a tool reports a buffer overflow in the patched programs. The confusion rate indicates how well a tool can distinguish between the bad and patched programs. When a tool reports a detection in both the patched and bad versions of a test case, the tool has demonstrated "confusion." The formulas used to compute these three metrics are shown below:

$$\text{detection rate} = \frac{\text{\# test cases where tool reports overflow in bad version}}{\text{\# test cases tool evaluated}}$$

$$\text{false alarm rate} = \frac{\text{\# test cases where tool reports overflow in patched version}}{\text{\# of test cases tool evaluated}}$$

$$\text{confusion rate} = \frac{\text{\# test cases where tool reports overflow in both bad and patched version}}{\text{\# test cases where tool reports overflow in bad version}}$$

As seen in Table 2, ARCHER and PolySpace both have detection rates exceeding 90%. PolySpace's detection rate is nearly perfect, missing only one out of the 291 possible detections. PolySpace produced seven false alarms, whereas ARCHER produced none. Splint and UNO each detected roughly half of the overflows. Splint, however, produced a substantial number of false alarms, while UNO produced none. Splint also exhibited a fairly high confusion rate. In over twenty percent of the cases where it properly detected an overflow, it also reported an error in the patched program. PolySpace's confusion rate was substantially lower, while the other three tools had no confusions. BOON's detection rate across the test suite was extremely low.

Table 2. Overall Performance on Basic Test Suite (291 cases)

Tool	Detection Rate	False Alarm Rate	Confusion Rate
ARCHER	90.7%	0.0%	0.0%
BOON	0.7%	0.0%	0.0%
PolySpace	99.7%	2.4%	2.4%
Splint	56.4%	12.0%	21.3%
UNO	51.9%	0.0%	0.0%

It is important to note that it was not necessarily the design goal of each tool to detect every possible buffer overflow. BOON, for example, focuses only on the misuse of string manipulation functions, and therefore is not expected to detect other overflows. It is also important to realize that these performance rates are not necessarily predictive of how the tools would perform on buffer overflows in actual, released code. The basic test suite used in this evaluation was designed for diagnostic purposes, and the taxonomy coverage exhibited is not representative of that which would be seen in real-world buffer overflows.

Figure 1 presents a plot of detection rate vs. false alarm rate for each tool. Each tool's performance is plotted with a single data point representing detection and false alarm percentages. The diagonal line represents the hypothetical performance of a random guesser that decides with equal probability if each commented buffer access in the test programs results in an overflow or not. The difference between a tool's detection rate and the random guesser's is only statistically significant if it lies more than two standard deviations (roughly 6 percentage points when the detection rate is 50%) away from the random guesser line at the same false alarm rate. In this evaluation, every tool except BOON performs significantly better than a random guesser. In Zitser's evaluation [20], only PolySpace was significantly better. This difference in performance reflects the simplicity of the diagnostic test cases.

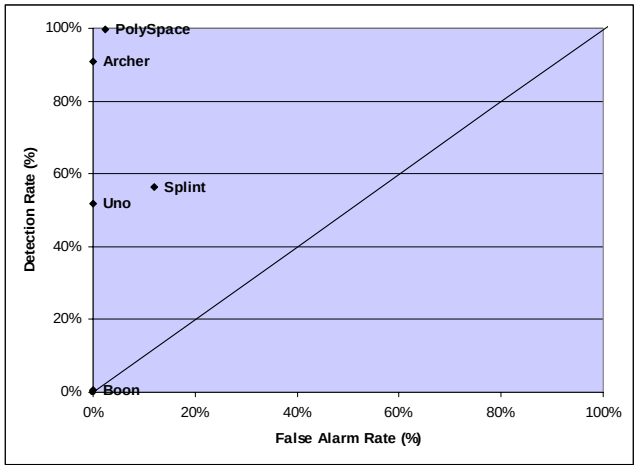


Figure 1. False Alarm and Detection Rates per Tool

Since PolySpace missed only one detection, and three of the other tools did detect the overflow in that test case, one could obtain perfect detection across the evaluation test suite by using PolySpace as the primary authority, and using one of the other tool's results only when PolySpace did not detect an overflow. ARCHER or UNO would be the best choice for this, as neither adds false alarms.

Similarly combining ARCHER and Splint would produce a detection rate of ninety-eight percent. ARCHER missed twenty-seven detections, and Splint detected all but five of those. Unfortunately, using Splint would also add thirty-five false alarms.

Table 3. Tool Execution Times

Tool	Total Time (secs)	Average Time per Test Case (secs)
ARCHER	288	0.247
BOON	73	0.063
PolySpace	200,820 (56 hrs)	172.526
Splint	24	0.021
UNO	27	0.023

Execution times for the five tools were measured as the total time to run each test case, including tool startup time, and are provided in Table 3. PolySpace's high detection rate comes at the cost of dramatically long execution times. ARCHER demonstrated both the second highest detection rate and the second highest execution time. Splint and UNO, with intermediate detection rates, had the two fastest execution times. BOON's slightly longer execution time did not result in a higher detection rate.

Some general observations can be made from inspecting the results as a whole. Missed detections and false alarms tend to group in certain attribute sets and follow logical patterns. If one tool missed a detection on a particular test case, usually some of the other tools missed it as well. For five test cases, only PolySpace did not miss detections in the bad programs. No attribute sets and no individual test cases have perfect detections across all five tools, but eight attribute sets contain no false alarms at all (Upper/Lower Bound, Data Type, Pointer, Alias of Buffer Index, Loop Structure, Loop Complexity, Asynchrony, and Signed/Unsigned Mismatch). Without the BOON results, looking exclusively at the results of the other four tools, three of the attribute sets (Write/Read, Data Type, and Alias of Buffer Index) and 108 individual test cases had perfect detections across the four tools. Complete and detailed results are presented in [11].

6. Detailed Tool Diagnostics

The following paragraphs discuss each tool's performance in detail, especially compared to the tools' design goals.

ARCHER's strategy is to detect as many bugs as possible while minimizing the number of false alarms. It is designed to be inter-procedural, path-sensitive, context-sensitive, and aware of pointer aliases. It performs a fully-symbolic, bottom-up data flow analysis, while maintaining symbolic constraints between variables (handled by a linear constraint solver). ARCHER checks array accesses, pointer dereferences, and function calls that take a pointer and size. It is hard-coded to recognize and handle a small number of memory-related functions, such as malloc [19].

The authors discuss many limitations of the current version of ARCHER. It does not handle function pointers, and imposes a five second limit on the analysis of any particular function. Furthermore, it loses precision after function calls, as it does not perform a proper inter-procedural side effects analysis and has a very simple alias analysis. It does not understand C library string functions, nor does it keep track of null pointers or the length of null-terminated strings. Its linear constraint solver is limited to handling at most two non-linearly related variables. Finally, some of the techniques it uses to reduce false alarms will necessarily result in missed detections. For instance, if no bounds information is known about a variable used as an array index, ARCHER assumes the array access is trusted and does not issue a warning. Similarly, it only performs a bounds check on the length and offset of a pointer dereference if bounds information is available; otherwise it remains quiet and issues no warning [19].

With close to a 91% detection rate and no false alarms, ARCHER performs well. Most of its twenty-seven missed detections are easily explained by its limitations. Twenty of these were inter-procedural, and this seems to be ARCHER's main weakness. The twenty inter-procedural misses include fourteen cases that call C library functions. While the authors admit to ignoring string functions, one might have expected memcpy() to be one of the few hard-coded for special handling. The other inter-procedural misses include cases involving shared memory, function pointers, recursion, and simple cases of passing a buffer address through one or two functions. Of the remaining seven misses, three involve function return values, two depend on array contents, and two involve function pointers and recursion.

While some of the missed detections occurred on cases whose features may not be widespread in real code (such as recursion), the use of C library functions and other inter-procedural mechanisms are surely prevalent. Indeed, ARCHER's poor performance in [20] is directly attributable to the preponderance of these features. ARCHER detected only one overflow in this prior evaluation, which was based on overflows in real code. Of the thirteen programs for which ARCHER reported no overflows, twelve contained buffer overflows that would be classified according to this evaluation's taxonomy as having inter-procedural scope, and nine of those involve calls to C library functions. To perform well against a body of real code, ARCHER needs to handle C library functions and other inter-procedural buffer overflows correctly.

BOON's analysis is flow-insensitive and context-insensitive for scalability and simplicity. It focuses exclusively on the misuse of string manipulation functions, and the authors intentionally sacrificed precision for scalability. BOON will not detect overflows caused by using primitive pointer operations, and ignores pointer dereferencing, pointer aliasing, arrays of pointers, function pointers, and unions. The authors expect a high false alarm rate due to the loss of precision resulting from the compromises made for scalability [18].

In this evaluation, BOON properly detected only two out of fourteen string function overflows, with no false alarms. The two detected overflows involve the use of strcpy() and fgets(). BOON

failed to detect the second case that calls strcpy(), all six cases that call strncpy(), the case that calls getcwd, and all four cases that call memcpy(). Despite the heavy use of C library string functions in [20], BOON achieved only two detections in that evaluation as well.

PolySpace is the only commercial tool included in this evaluation. Details of its methods and implementation are proprietary. We do know, however, that its approach uses techniques described in several published works, including: symbolic analysis, or abstract interpretation [2]; escape analysis, for determining inter-procedural side effects [4]; and inter-procedural alias analysis for pointers [3]. It can detect dead or unreachable code. Like other tools, it may lose precision at junctions in code where previously branched paths rejoin, a compromise necessary to keep the analysis tractable.

PolySpace missed only one detection in this evaluation, which was a case involving a signal handler. The PolySpace output for this test case labeled the signal handler function with the code "UNP," meaning "unreachable procedure." PolySpace reported seven false alarms across the test suite. These included all four of the taint cases, shared memory, using array contents for the buffer address, and one of the calls to strcpy(). The false alarm on the array contents case is not too surprising, as it is impractical for a tool to track the contents of every location in memory. PolySpace does not, however, report a false alarm on the other two cases involving array contents. The other six false alarms are on test cases that in some way involve calls to C library or O/S-specific function calls. Not all such cases produced false alarms, however. For instance, only one out of the two strcpy() cases produced a false alarm: the one that copies directly from a constant string (e.g., "AAAA"). Without more insight into the PolySpace implementation, it is difficult to explain why these particular cases produced false alarms.

PolySpace did not perform as well in Zitser's evaluation [20]. Again, without more knowledge of the tool's internals, it is difficult to know why its detection rate was lower. Presumably the additional complexity of real code led to approximations to keep the problem tractable, but at the expense of precision. The majority of the false alarms it reported in Zitser's evaluation were on overflows similar to those for which it reported false alarms in this evaluation: those involving memory contents and C library functions.

PolySpace's performance comes with additional cost in money and in time. The four other tools were open source when this evaluation was performed, and completed their analyses across the entire corpus in seconds or minutes. PolySpace is a commercial program and ran for nearly two days and eight hours, averaging close to three minutes of analysis time per test case file. This long execution time may make it difficult to incorporate into a code development cycle.

Splint employs "lightweight" static analysis and heuristics that are practical, but neither sound nor complete. Like many other tools, it trades off precision for scalability. It implements limited flow-sensitive control flow, merging possible paths at branch points. Splint uses heuristics to recognize loop idioms and determine loop bounds without resorting to more costly and accurate abstract evaluation. An annotated C library is provided, but the tool relies on the user to properly annotate all other functions to support inter-procedural analysis. Splint exhibited high false alarm rates in the developers' own tests [6, 12].

The basis test suite used in this evaluation was not annotated for Splint for two reasons. First, it is a more fair comparison of the tools to run them all against the same source code, with no special accommodations for any particular tool. Second, expecting developers to completely and correctly annotate their programs for Splint seems unrealistic.

Not surprisingly, Splint exhibited the highest false alarm rate of any tool. Many of the thirty-five false alarms are attributable to inter-procedural cases; cases involving increased complexity of the index, address, or length; and more complex containers and flow control constructs. The vast majority, 120 out of 127, of missed detections are attributable to loops. Detections were missed in all of the non-standard for() loop cases (both discrete and continuous), as well as in most of the other continuous loop cases. The only continuous loop cases handled correctly are the standard for loops, and it also produces false alarms on nearly all of those. In addition, it misses the lower bound case, the "cond" case of local flow control, the taint case that calls getcwd, and all four of the signed/unsigned mismatch cases.

While Splint's detection rate was similar in this evaluation and the Zitser evaluation [20], its false alarm rate was much higher in the latter. Again, this is presumably because code that is more complex results in more situations where precision is sacrificed in the interest of scalability, with the loss of precision leading to increased false alarms.

Splint's weakest area is loop handling. Enhancing loop heuristics to more accurately recognize and handle non-standard for loops, as well as continuous loops of all varieties, would significantly improve performance. The high confusion rate may be a source of frustration to developers, and may act as a deterrent to Splint's use. Improvements in this area are also important.

UNO is an acronym for uninitialized variables, null-pointer dereferencing, and out-of-bounds array indexing, which are the three types of problems it is designed to address. UNO implements a two-pass analysis; the first pass performs intra-procedural analysis within each function, while the second pass performs a global analysis across the entire program. It appears that the second pass focuses only on global pointer dereferencing, in order to detect null pointer usage; therefore, UNO would not seem to be inter-procedural with respect to out-of-bounds array indexing. UNO determines path infeasibility, and uses this information to suppress warnings and take shortcuts in its searches. It handles constants and scalars, but not computed indices (expressions on variables, or function calls), and easily loses precision on conservatively-computed value ranges. It does not handle function pointers, nor does it attempt to compute possible function return values. Lastly, UNO does not handle the setjmp/longjmp construct [8].

UNO produced no false alarms in the basic test suite, but did miss nearly half of the possible detections (140 out of 291), most of which would be expected based on the tool's description. This included every inter-procedural case, every container case, nearly every index complexity case (the only one it detected was the simple variable), every address and length complexity case, every address alias case, the function and recursion cases, every

signed/unsigned mismatch, nearly every continuous loop, and a small assortment of others. It performed well on the various data types, index aliasing, and discrete loops. Given the broad variety of detections missed in the basic test suite, it is not surprising that UNO exhibited the poorest performance in Zitser's evaluation [20].

7. CONCLUSIONS

A corpus of 291 small C-program test cases was developed to evaluate static and dynamic analysis tools that detect buffer overflows. The corpus was designed and labeled using a new, comprehensive buffer overflow taxonomy. It provides a benchmark to measure detection, false alarm, and confusion rates of tools, and can be used to find areas for tool enhancement. Evaluations of five tools validate the utility of this corpus and provide diagnostic results that demonstrate the strengths and weaknesses of these tools. Some tools provide very good detection rates (e.g. ARCHER and PolySpace) while others fall short of their specified design goals, even for simple uncomplicated source code. Diagnostic results provide specific suggestions to improve tool performance (e.g. for Splint, improve modeling of complex loop structures; for ARCHER, improve inter-procedural analysis). They also demonstrate that the false alarm and confusion rates of some tools (e.g. Splint) need to be reduced.

The test cases we have developed can serve as a type of litmus test for tools. Good performance on test cases that fall within the design goals of a tool is a prerequisite for good performance on actual, complex code. Additional code complexity in actual code often exposes weaknesses of the tools that result in inaccuracies, but rarely improves tool performance. This is evident when comparing test case results obtained in this study to results obtained by Zitser [20] with more complex model programs. Detection rates in these two studies are shown in Table 4. As can be seen, the two systems that provided the best detection rates on the model programs (PolySpace and Splint) also had high detection rates on test cases. The other three tools performed poorly on model programs and either poorly (BOON) or well (ARCHER and UNO) on test cases. Good performance on test cases (at least on the test cases within the tool design goals) is a necessary but not sufficient condition for good performance on actual code. Finally, poor performance on our test corpus does not indicate that a tool doesn't provide some assistance when searching for buffer overflows. Even a tool with a low detection rate will eventually detect some errors when used to analyze many thousands of lines of code.

Table 4. Comparison of detection rates with 291 test cases and with 14 more complex model programs in Zitser [20].

Tool	Test Case Detection	Model Program Detection [20]
ARCHER	90.7%	1%
BOON	0.7%	5%
PolySpace	99.7%	87%
Splint	56.4%	57%
UNO	51.9%	0.0%

The test corpus could be improved by adding test cases to cover attribute values currently underrepresented, such as string functions. It may also be used to evaluate the performance of dynamic analysis approaches. Anyone wishing to use the test corpus should send email to the authors.

8. ACKNOWLEDGMENTS

We would like to thank Rob Cunningham and Tim Leek for discussions, and Tim for help with getting tools installed and running. We also thank David Evans for his help with Splint, David Wagner for answering questions about BOON, Yichen Xie and Dawson Engler for their help with ARCHER, and Chris Hote and Vince Hopson for answering questions about C-Verifier and providing a temporary license.

9. REFERENCES

[1] CERT (2004). CERT Coordination Center Advisories, http://www.cert.org/advisories/, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA

[2] Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs, *Proceedings of the 2nd International Symposium on Programming*, Paris, France, 106--130

[3] Deutsch, A. (1994). Interprocedural may-alias analysis for pointers: beyond *k*-limiting, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, Florida, 230--241

[4] Deutsch, A. (1997). On the complexity of escape analysis, *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 358--371

[5] Etoh, H. (2004). GCC extension for protecting applications from stack smashing attacks, http://www.trl.ibm.com/projects/security/ssp/

[6] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis, *IEEE Software*, 19 (1), 42--51

[7] GCC Home Page (2004). Free Software Foundation, Boston, MA, http://gcc.gnu.org/

[8] Holzmann, G. (2002). UNO: Static source code checking for user-defined properties, Bell Labs Technical Report, Bell Laboratories, Murray Hill, NJ, 27 pages

[9] ICAT (2004). The ICAT Metabase, http://icat.nist.gov/icat.cfm, National Institute of Standards and Technology, Computer Security Division, Gaithersburg, MD

[10] klog (1999). The frame pointer overwrite, *Phrack Magazine*, 9 (55), http://www.tegatai.com/~jbl/overflow-papers/P55-08

[11] Kratkiewicz, K. (2005). Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code, Master's Thesis, Harvard University, Cambridge, MA, 285 pages

[12] Larochelle, D. and Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities, *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, 177--190

[13] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. (2003). The Spread of the Sapphire/Slammer Worm, http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html

[14] PolySpace Technologies (2003). PolySpace C Developer Edition, http://www.polyspace.com/datasheets/c_psde.htm, Paris, France

[15] PSS Security Response Team (2003). PSS Security Response Team Alert - New Worm: W32.Blaster.worm, http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/alerts/msblaster.asp, Microsoft Corporation, Redmond, WA

[16] Ruwase, O. and Lam, M. (2004). A practical dynamic buffer overflow detector, *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, 159--169

[17] Security Focus (2004). The Bugtraq mailing list, http://www.securityfocus.com/archive/1, SecurityFocus, Semantec Corporation, Cupertino, CA

[18] Wagner, D., Foster, J.S., Brewer, E.A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities, *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 3--17

[19] Xie, Y., Chou, A., and Engler, D. (2003). ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors, *Proceedings of the 9th European Software Engineering Conference/10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Helsinki, Finland, 327--336

[20] Zitser, M. (2003). Securing Software: An Evaluation of Static Source Code Analyzers, Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, 130 pages

[21] Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open-source code, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, 97--106

BugBench: Benchmarks for Evaluating Bug Detection Tools

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou and Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana Champaign, Urbana, IL 61801

ABSTRACT

Benchmarking provides an effective way to evaluate different tools. Unfortunately, so far there is no good benchmark suite to systematically evaluate software bug detection tools. As a result, it is difficult to quantitatively compare the strengths and limitations of existing or newly proposed bug detection tools.

In this paper, we share our experience of building a bug benchmark suite called *BugBench*. Specifically, we first summarize the general guidelines on the criteria for selecting representative bug benchmarks, and the metrics for evaluating a bug detection tool. Second, we present a set of buggy applications collected by us, with various types of software bugs. Third, we conduct a preliminary study on the application and bug characteristics in the context of software bug detection. Finally, we evaluate several existing bug detection tools including Purify, Valgrind, and CCured to validate the selection of our benchmarks.

1 Introduction

1.1 Motivation

Software bugs account for more than 40% system failures [20], which makes software bug detection an increasingly important research topic. Recently, many bug detection tools have been proposed, with many more expected to show up in the near future. Facing ever so many tools, programmers need a guidance to select tools that are most suitable for their programs and occurring failures; and researchers also desire a unified evaluation method to demonstrate the strength and weakness of their tools versus others. All these needs strongly motivate a representative, fair and comprehensive evaluation benchmark suite for the purpose of evaluating software bug detection tools.

Benchmark is a standard of measurement or evaluation, and an effective and affordable way of conducting experiments [28]. A good all-community accepted benchmark suite has both technique and sociality impact. In the technical aspect, evaluations with standard benchmarks are more rigorous and convincing; alternative ideas can be compared objectively; problems overlooked in previous research might be manifested in benchmarking. In the social aspect, building benchmark enforces the collaboration within community and help the community to form a common understanding of the problem they are facing [26]. There are many successful benchmark examples, such as SPEC (Standard Performance Evaluation Corporation) benchmarks [27], and TPC series (Transaction Processing Council) [29], both of which have been widely used by the corresponding research and product development communities.

However, in the software bug detection area, there is no widely-accepted benchmark suite to evaluate existing or newly proposed methods. As a result, many previous studies either use synthetic toy-applications or borrow benchmarks (such as SPEC and Siemens) from other research areas. While such evaluation might be appropriate to use for proof of concept, it hardly provides a solid demonstration of the unique strength and shortcomings of the proposed method. Being aware of this problem, some studies [5, 23, 32] use real buggy applications for evaluation, which makes the proposed tools much more convincing. Unfortunately, based on our previous experience of evaluating our own bug detection tools [18, 24, 32, 33], finding real applications with real bugs is a time-consuming process, especially since many bug report databases are not well documented for our purposes, i.e., they only report the symptoms but not the root causes. Furthermore, different tools are evaluated with different applications, making it hard to cross-compare tools with similar functionality.

Besides benchmarking, the evaluation criteria of software bug detection tools are also not standardized. Some work evaluated only the execution overhead using SPEC benchmarks, completely overlooking the bug detection functionality. In contrast, some work [16, 18] did much more thorough evaluation. They not only reported false positives and/or false negatives, but also provided the ranking of reported bugs.

As the research area of software bug detection starts booming with many innovative ideas, the urgency of a unified evaluation method with a standard benchmark suite has been recognized, as indicated by the presence of this workshop. For example, researchers at IBM Haifa [6] advocate building benchmarks for testing and debugging concurrent programs. Similarly, although not formally announced as benchmark, a Java application set, HEDC used in [31], is shared by a few laboratories to compare the effectiveness of data race detection methods.

1.2 Our Work

Benchmark suite building is a long-term, iterative process and needs the cooperation from all over the community. In this paper, we share our experience of building a bug benchmark suite as a vehicle to solicit feedbacks. We plan to release the current collection of buggy applications soon to the research community. Specifically, this paper reports our work on bug benchmark design and collection in the following aspects:

(1) General guidelines on bug benchmark selection criteria and evaluation metrics: By learning from successful benchmarks in other areas and prior unsuccessful bug benchmark trials, we summarize several criteria that we follow when selecting a buggy application into our benchmark suite. In addition, based on previous research experience and literature research in software bug detection, we also summarize a set of quantitative and qualitative metrics for evaluating bug detection tools.

(2) A C/C++ bug benchmark suite *BugBench*: By far, we have collected 17 C/C++ applications for our *BugBench* and we are still looking for more applications to enrich the suite. All of the applications are from the open source community and contain various software defects including buffer overflows, stack smashing, double frees, uninitialized reads, memory leaks, data races, atomic violations, semantic bugs, etc. Some of these buggy applications have been used by our previous work [18, 24, 32, 33], and also forwarded by us to a few other research groups at UCSD, Purdue, etc in their studies [7, 22].

(3) A preliminary study of benchmark and bug characteristics: We have studied the characteristics of several benchmarks that contain memory-related bugs, including memory access frequencies, malloc frequencies, crash latencies (the distance from the root cause to the manifestation point), etc., which would affect the overhead and bug-detection capability of a dynamic bug

detection tool. *To our best knowledge, ours is one of the first in studying buggy application characteristics in the context of software bug detection.*

(4) A preliminary evaluation of several existing tools: To validate our selection of benchmarks and characteristics, we have conducted a preliminary evaluation using several existing tools including Purify [12], Valgrind [25] and CCured [23]. Our preliminary results show that our benchmarks can effectively differentiate the strengths and limitations of these tools.

2 Lessons from Prior Work

2.1 Successful Benchmark in Other Areas

SPEC (Standard Performance Evaluation Cooperative) was founded by several major computer vendors in order to "*provide the industry with a realistic yardstick to measure the performance of advanced computer systems*" [3]. To achieve this purpose, SPEC has very strict application selection process. First, candidates are picked from those that have significant use in their fields, e.g. gcc from compiler field, and weather prediction from scientific computation field. Then, candidates are checked for their clarity and portability over different architecture platforms. Qualified candidates will be analyzed for detailed dynamic characteristics, such as instruction mix, memory usage, etc. Based on these characteristics, SPEC committee decides whether there are enough diversity and little redundancy in the benchmark suite. After several iterations of the above steps, a SPEC-benchmark is finally announced.

TPC (Transaction Processing Council) was founded in the middle 80's to satisfy the demand of comparing numerous database management systems. TPC benchmark shares some common properties as that in SPEC, i.e. representative, diverse, and portable, etc. Take TPC-C (an OLTP benchmark) as an example[17]. To be representative, TPC-C uses five real-world popular transactions: new order, payment, delivery, order status, and stock level. In terms of diversity, these transactions cover almost all important database operations. In addition, TPC-C has a comprehensive evaluation metric set. It adopts two standard metrics: new-order transaction rate and price/performance, together with additional tests for ACID properties, e.g. whether the database can recover from failure. All these contribute to the great success of TPC-C benchmark.

2.2 Prior Benchmarks in Software Engineering and Bug Detection Areas

Recently, much progress has been made on benchmarking in software engineering-related areas. *CppETS* [26] is a benchmark suite in reverse engineering for evaluating "factor extractors". It provides a collection of C++ programs, each associated with a question file. Evaluated tools will answer the questions based on their factor extracting results and get points from their answers. The final score from all test programs indicates the performance of this tool. This benchmark suite is a good vehicle to objectively evaluate and compare factor extractors.

The benchmark suites more related to bug detection are Siemens benchmark suite [11] and PEST benchmark suite [15] for software testing. In these benchmark suites, each application is associated with some buggy versions. Better testing tools can distinguish more buggy versions from correct ones. Although these benchmark suites provide a relatively large bug pool, most bugs are semantic bugs. There is almost no memory-related bugs and definitely no multi-threading bugs. Furthermore, the benchmark applications are very small (some are less than 100 line of code), hence cannot represent real bug detection scenarios and can hardly be used to measure time overhead. Therefore, they are not suitable for serving as bug detection benchmarks.

In the bug detection community, there is not much work done in benchmarking. Recently, researchers in IBM Haifa [14] propose building multithreading program benchmarks. However, their efforts are unsuccessful as also acknowledged in their following paper [6], because they rely on students to purposely generate buggy programs instead of using real ones.

3 Benchmarking Guideline

3.1 Classification of Software Bugs

In order to build good bug benchmarks, we first need to classify software bugs. There are different ways to classify bugs [1, 15], in this section we make classification based on different challenges the bug exposes to the detection tools. Since our benchmark suite cannot cover all bug types, in the following we only list the bug types that are most security critical and most common. They are also the design focus of most bug detection tools.

Memory related bugs Memory related bugs are caused by improper handling of memory objects. These bugs are often exploited to launch security attack. Based on US-CERT vulnerability Notes Database [30], they contribute the most to all reported vulnerabilities since 1991. Memory-related bugs can be further classified into: (1) Buffer overflow: Illegal access beyond the buffer boundary. (2) Stack smashing: Illegally overwrite the function return address. (3) Memory leak: Dynamically allocated memory have no reference to it, hence can never be freed. (4) Uninitialized read: Read memory data before it is initialized. The reading result is illegal. (5) Double free: One memory location freed twice.

Concurrent bugs Concurrent bugs are those that happen only in multi-threading (or multi-processes) environment. They are caused by ill-synchronized operations from multiple threads. Concurrent bugs can be further divided into following groups: (1) Data race bugs: Conflicting accesses from concurrent threads touch the shared data in arbitrary order. (2) Atomicity-related bugs: A bunch of operations from one thread is unexpectedly interrupted by conflicting operations from other threads. (3) Deadlock: In resource sharing, one or more processes permanently wait for some resources and can never proceed any more.

An important property of concurrent bugs is un-deterministic, which makes them hard to be reproduced. Such temporal sensitivity adds extra difficulty to bug detection.

Semantic bugs A big family of software bugs are semantic bugs, i.e. bugs that are inconsistent with the original design and the programmers' intention. We often need semantic information to detect these bugs.

3.2 Classification of Bug Detection Tools

Different tools detect bugs using different methods. A good benchmark suite should be able to demonstrate the strength and weakness of each tool. Therefore, in this section, we study the classification of bug detection tools, by taking a few tools as examples and classifying them by two criteria in Table 1.

	Static	**Dynamic**	**Model Checking**
Programming-rule based tools	PREfix [2] RacerX [4]	Purify [12] Valgrind [25]	VeriSoft[9] JPFinder[13]
Statistic-rule based tools	CP-Miner [18] D. Engler's [5]	DIDUCE [10] AccMon [32] Liblit's [19]	CMC[21]
Annotation-based	ESC/Java [8]		

Table 1: Classification of a few detection tools

As shown in Table 1, one way to classify tools is based on the rules they use to detect bugs. Most detection tools hold some "rules" in mind: code violating the rules is reported as bug. *Programming-rule-based* tools use rules that should be followed in programming, such as "array pointer cannot move out-of-bound". *Statistic-rule-based* approaches learn statistically correct rules (invariants) from successful runs in training phase. *Annotation-based* tools use programmer-written annotations to check semantic bugs.

We can also divide tools into *static*, *dynamic* and *model checking*. *Static* tools detect bugs by static analysis, without requiring code execution. *Dynamic* tools are used during execution, analyzing run-time information to detect bugs on-the-fly. They add runtime overhead but are more accurate. *Model checking* is a formal verification method. It was usually grouped into static detection tools. However, recently people also use model checking during program execution.

3.3 Benchmark Selection Criteria

Based on the study in section 2 and 3.1, 3.2, we summarize following bug detection benchmark selection criteria. **(1) Representative:** The applications in our benchmark suite should be able to represent real buggy applications. That means: First, the application should be real, implemented by experienced programmers instead of novices. It is also desirable if the application has significant use in practice. Second, the bug should also be real, naturally generated, not purposely injected. **(2) Diverse:** In order to cover a wide range of real cases, the applications in benchmark should be diverse in the state space of some important characteristics, including bug types; some dynamic execution characteristics, such as heap and stack usage, the frequency of dynamic allocations, memory access properties, pointer dereference frequency, etc; and the complexity of bugs and applications, including the bug's crash latency, the application's code size and data structure complexity, etc. Some of these characteristics will be discussed in detail in section 4.2. **(3) Portable:** The benchmark should be able to evaluate tools designed on different architecture platforms, so it is better to choose hardware-independent applications. **(4) Accessible:** Benchmark suites are most useful when everybody can easily access them and use them in evaluation. Obviously, proprietary applications can not meet this requirement, so we only consider open source code to build our benchmark. **(5) Fair:** The benchmark should not bias toward any detection tool. Applying above criteria, we can easily see that benchmarks like SPEC, Siemens are not suitable in our context: many SPEC applications are not buggy at all and Siemens benchmarks are not diverse enough in code size, bug types and other characteristics.

In addition to the above five criteria designed for selecting applications into the bug benchmark suite, application inputs also need careful selection. A good input set should contain both correct inputs and bug-triggering inputs. Bug-triggering inputs will expose the bug and correct inputs can be used to calculate false positives and enable the overhead measurement in both buggy runs and correct runs. Additionally, a set of correct inputs can also be used to unify the training phase of invariant-based tools.

3.4 Evaluation Metrics

The effectiveness of a bug detection tool has many aspects. A complete evaluation and comparison should base on a set of metrics that reflect the most important factors. As shown in Table 2, our metric set is composed of four groups of metrics, each representing an important aspect of bug detection.

Most metrics can be measured quantitatively. Even for some traditionally subjective metric, such as "pinpoint root cause", we can measure it quantitatively by the distance from the bug root cause to the bug detection position in terms of dynamic and/or static instruction numbers (we call it *Detection Latency*). Some metrics, such as manual effort and new hardware reliance, will be measured qualitatively.

Functionality Metrics	**Overhead Metrics**
Bug Detection False Positive	Time Overhead
Bug Detection False Negative	Space Overhead
Easy to Use Metrics	Static Analysis Time
Reliance on Manual Effort	Training Overhead
Reliance on New Hardware	Dynamic Detection Overhead
Helpful to Users Metrics	
Bug Report Ranking	
Pinpoint Root Cause?	

Table 2: Evaluation metric set

We should also notice that, the same metric may have different meanings for different types of tools. That is the reason that we list three different types of overhead together with the time and space overhead metrics. We will only measure static analysis time for static tools; measure both training and dynamic detection overhead for *statistical-rule-based* tools and measure only dynamic detection overhead for most *programming-rule-based* tools. The comparison among tools of the same category is more appropriate for some metrics. When comparing tools of different categories, we should keep the differences in mind.

4 Benchmark

4.1 Benchmark Suite

Based on the criteria in section 3.3, we have collected 17 buggy C/C++ programs from open source repositories. These programs contain various bugs including 13 memory-related bugs, 4 concurrent bugs and 2 semantic bugs [1]. We have also prepared different test cases, both bug-triggering and non-triggering ones, for each application. We are still in the process of collecting more buggy applications.

Table 3 shows that all applications are real open-source applications with real bugs and most of them have significant use in their domains. They have different code sizes and have covered most important bug types.

As we can see from the table, the benchmark suite for memory-related bugs is already semi-complete. We will conduct more detailed analysis for them in the following sections. Other types of bugs, however, are incomplete yet. Enriching *BugBench* with more applications on other types of bugs and more analysis on large applications remains as our future work.

4.2 Preliminary Characteristics Analysis

An important criterion for a good benchmark suite is its diversity on important characteristics, as we described in section 3.3. In this section, we focus on a subset of our benchmarks (memory-related bug applications) and analyze their characteristics that would affect *dynamic* memory bug detection tools.

Dynamic memory allocation and memory access behaviors are the most important characteristics that have significant impact on the overheads of dynamic memory-related bug detection tools. This is because many memory-related bug detection tools intercept memory allocation functions and monitor most memory accesses. In table 4, we use frequency and size to represent dynamic allocation properties. As we can see, in 8 applications, the memory allocation frequency ranges from 0 to 769 per Million Instructions and the size ranges from 0 to 6.0 MBytes. Such large range of memory allocation behaviors will lead to different overheads in dynamic bug detection tools, such as Valgrind and Purify. In general, the more frequent of memory allocation, the larger

[1] some applications contain more bugs than we describe in Table 3.

Name	Program	Source	Description	Line of Code	Bug Type
NCOM	ncompress-4.2.4	Red Hat Linux	file (de)compression	1.9K	Stack Smash
POLY	polymorph-0.4.0	GNU	file system "unixier" (Win32 to Unix filename converter)	0.7K	Stack Smash & Global Buffer Overflow
GZIP	gzip-1.2.4	GNU	file (de)compression	8.2K	Global Buffer Overflow
MAN	man-1.5h1	Red Hat Linux	documentation tools	4.7K	Global Buffer Overflow
GO	099.go	SPEC95	game playing (Artificial Intelligent)	29.6K	Global Buffer Overflow
COMP	129.compress	SPEC95	file compression	2.0K	Global Buffer Overflow
BC	bc-1.06	GNU	interactive algebraic language	17.0K	Heap Buffer Overflow
SQUD	squid-2.3	Squid	web proxy cache server	93.5K	Heap Buffer Overflow
CALB	cachelib	UIUC	cache management library	6.6K	Uninitialized Read
CVS	cvs-1.11.4	GNU	version control	114.5K	Double Free
YPSV	ypserv-2.2	Linux NIS	NIS server	11.4K	Memory Leak
PFTP	proftpd-1.2.9	ProFTPD	ftp server	68.9K	Memory Leak
SQUD2	squid-2.4	Squid	web proxy cache	104.6K	Memory Leak
HTPD1	httpd-2.0.49	Apache	HTTP server	224K	Data Race
MSQL1	msql-4.1.1	MySQL	database	1028K	Data Race
MSQL2	msql-3.23.56	MySQL	database	514K	Atomicity
MSQL3	msql-4.1.1	MySQL	database	1028K	Atomicity
PSQL	postgresql-7.4.2	PostgreSQL	database	559K	Semantic Bug
HTPD2	httpd2.0.49	Apache	HTTP server	224K	Semantic Bug

Table 3: Benchmark suite

Name	Malloc Freq. (# per MInst)	Allocated Memory Size	Heap Usage vs. Stack Usage	Memory Access (# per Inst)	Memory Read vs. Write	Symptom	Crash Latency (# of Inst)
NCOM	0.003	8B	0.4% vs. 99.5%	0.848	78.4% vs. 21.6%	No Crash	NA
POLY	7.14	10272B	23.9% vs. 76.0%	0.479	72.6% vs. 27.4%	Varies on Input*	9040K*
GZIP	0	0B	0.0% vs. 100%	0.688	80.1% vs. 19.9%	Crash	15K
MAN	480	175064B	85.1% vs. 14.8%	0.519	70.9% vs. 20.1%	Crash	29500K
GO	0.006	364B	1.6% vs. 98.3%	0.622	82.7% vs. 17.3%	No Crash	NA
COMP	0	0B	0.0% vs. 100%	0.653	79.1% vs. 20.9%	No Crash	NA
BC	769	58951B	76.6% vs. 23.2%	0.554	71.4% vs. 28.6%	Crash	189K
SQUD	138	5981371B	99.0% vs. 0.9%	0.504	54.2% vs. 45.8%	Crash	0

Table 4: Overview of the applications and their characteristics (*:The crash latency is based on the input that will cause the crash.)

overhead would be imposed by such tools. To reflect the memory access behavior, we use access frequency, read/write ratio and heap/stack usage ratio. Intuitively, access frequency directly influences the dynamic memory bug detection overhead: the more frequent memory accesses, the larger the checking overhead. Some tools use different policies to check read and write accesses and some tools differentiate stack and heap access, so all these ratios are important to understand the overhead. In table 4, the access frequencies of our benchmark applications change from 0.479 to 0.848 access per instruction and heap usage ratio from 0 to 99.0%. Both show a good coverage. Only the read/write ratio seems not to change much within all 8 applications, which indicates the need to further improve our benchmark suite based on this.

Obviously, the bug complexity may directly determines the false negatives of bug detection tools. In addition, the more difficult to detect, the more benefits a bug detection tool can provide the programmers. While it is possible to use many ways to measure complexity (which we will do in the future), we use the symptom and crash latency to measure this property. Crash latency is the latency from the root cause of a bug to the place where the application finally crashes due to the propagation of this bug. If the crash latency is very short, for example, right at the root cause, even without any detection tool, programmers may be able to catch the bug immediately based on the crash position. On the other hand, if the bug does not manifest until a long chain of error propagation, detecting the bug root cause would be much more challenging for both programmers and all bug detection tools. As shown in Table 4, the bugs in our benchmarks manifest in different ways: crash or silent errors. For applications that will crash, their crash latency varies from zero latency to 29 million instructions.

5 Preliminary Evaluation

In order to validate the selection of our bug benchmark suite, in this section, we use *BugBench* to evaluate 3 popular bug detection tools: Valgrind [25], Purify [12] and CCured [23]. All these three are designed to detect memory-related bugs, so we choose 8 memory-relate buggy applications from our benchmarks. The evaluation results are shown in Table 5.

In terms of time overhead, among the three tools, CCured always has the lowest overhead, because it conducted static analysis beforehand. Purify and Valgrind have similar magnitude of overhead. Since Valgrind is implemented based on instruction emulation and Purify is based on binary code instrumentation, we do not compare the overheads of these two tools. Instead, we show how an application's characteristics affect one tool's overhead. Since our bug benchmark suite shows a wide range of characteristics, the overheads imposed by these tools also vary from more than 100 times of overhead to less than 20% overhead. For example, the application BC has the largest overhead in both Valgrind and Purify, as high as 119 times. The reason is that BC has very high memory allocation frequency, as shown in Table 4. On the other hand, POLY has very small overhead due to its smallest memory access frequency as well as its small allocation frequency.

In terms of bug detection functionality, CCured successfully catches all the bugs in our applications and also successfully points out the root cause in most cases. Both Valgrind and Purify fail to catch the bugs in NCOM and COMP. The former is a stack buffer overflow and the latter is a one-byte global buffer overflow. Valgrind also fails to catch another global buffer overflow in GO and has long detection latencies in the other three global buffer over-

	Catch Bug?			False Positive			Pinpoint The Root Cause (Detection Latency(KInst)[1])			Overhead			Easy to Use		
	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured
NCOM	No	No	Yes	0	0	0	N/A	N/A	Yes	6.44X	13.5X	18.5%	Easiest	Easy	Moderate
POLY	Vary[2]	Yes	Yes	0	0	0	No(9040K)[2]	Yes	Yes	11.0X	27.5%	4.03%	Easiest	Easy	Moderate
GZIP	Yes	Yes	Yes	0	0	0	No(15K)	Yes	Yes	20.5X	46.1X	3.71X	Easiest	Easy	Moderate
MAN	Yes	Yes	Yes	0	0	0	No(29500K)	Yes	Yes	115.6X	7.36X	68.7%	Easiest	Easy	Hard
GO	No	Yes	Yes	0	0	0	N/A	Yes	Yes	87.5X	36.7X	1.69X	Easiest	Easy	Moderate
COMP	No	No	Yes	0	0	0	N/A	N/A	Yes	29.2X	40.6X	1.73X	Easiest	Easy	Moderate
BC	Yes	Yes	Yes	0	0	0	Yes	Yes	Yes	119X	76.0X	1.35X	Easiest	Easy	Hardest
SQUD	Yes	Yes	N/A[3]	0	0	N/A[3]	Yes	Yes	N/A[3]	24.21X	18.26X	N/A[3]	Easiest	Easy	Hardest

Table 5: Evaluation of memory bug detection tools. (1: Detection latency is only applicable when fail to pinpoint the root cause; 2: Valgrind's detection result varies on inputs. Here we use the input by which Valgrind fails to pinpoint root cause; 3: We fail to apply CCured on Squid)

flow applications: POLY, GZIP and MAN. The results indicate that Valgrind and Purify handle heap objects much better than they do on stack and global objects.

As for POLY, we tried different buggy inputs for Valgrind and the results are interesting: if the buffer is not overflowed significantly, Valgrind will miss it; with moderate overflow, Valgrind catches the bug after a long path of error propagation, not the root cause; only with significant overflow, Valgrind can detect the root cause. The different results are due to POLY's special bug type: first global corruption and later stack corruption.

Although CCured performs much better than Valgrind and Purify in both overhead and functionality evaluation, the tradeoff is its high reliance on manual effort in code preprocessing. As shown in the "Easy to Use" column of Table 5, among all these tools, Valgrind is the easiest to use and requires no re-compilation. Purify is also fairly easy to use, but requires re-compilation. CCured is the most difficult to use. It often requires fairly amount of source code modification. For example, in order to use CCured to check BC, we have worked about 3 to 4 days to study the CCured policy and BC's source code to make it satisfy the CCured's language requirement. Moreover, we fail to apply CCured on a more complicated server application: SQUD.

6 Current Status & Future Work

Our *BugBench* is an ongoing project. We will release these applications together with documents and input sets through our web page soon. We welcome feedbacks to refine our benchmark.

In the future, we plan to extend our work in several dimensions. First, we will enrich the benchmark suite with more applications, more types of bugs based on our selection criteria and characteristic analysis (the characteristics in Table 4 show that some important benchmark design space is not covered yet). We are also in the plan of designing tools to automatically extract bugs from bug databases (e.g. Bugzilla) maintained by programmers, so that we can not only get many real bugs but also gain deeper insight into real large buggy applications. Second, we will evaluate more bug detection tools, which will help us enhance our *BugBench*. Third, we intend to add some supplemental tools, for example, program annotation for static tools, and scheduler and record-replay tools for concurrent bug detection tools.

REFERENCES

[1] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[3] K. M. Dixit. The spec benchmarks. *Parallel Computing*, 17(10-11), 1991.

[4] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, Oct. 2003.

[5] 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 *SOSP '01*, pages 57–72, 2001.

[6] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*, 2004.

[7] L. Fei and S. Midkiff. Artemis: Practical runtime monitoring of applications for errors. Technical Report TR-ECE05-02, Purdue University, 2005.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[9] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *ISSTA*, pages 124–133, 1998.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02*, May 2002.

[11] M. J. Harrold and G. Rothermel. Siemens Programs, HR Variants. URL: http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, Jan. 1992.

[13] K. Havelund and J. U. Skakkebæk. Applying model checking in java verification. In *SPIN*, 1999.

[14] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *IPDPS*, 2003.

[15] James R. Lyle, Mary T. Laamanen, and Neva M. Carlson. PEST: Programs to evaluate software testing tools and techniques. URL: www.nist.gov/itl/div897/sqg/pest/pest.html.

[16] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12*, pages 83–93, 2004.

[17] C. Levine. TPC-C: an OLTP benchmark. URL: http://www.tpc.org/information/sessions/sigmod/sigmod97.ppt, 1997.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[20] E. Marcus and H. Stern. Blueprints for high availability. John Willey and Sons, 2000.

[21] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, Dec. 2002.

[22] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA'05*, 2005.

[23] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, Jan. 2002.

[24] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05*, 2005.

[25] J. Seward. Valgrind. URL: http://www.valgrind.org/.

[26] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *ICSE '03*, pages 74–83. IEEE Computer Society, 2003.

[27] Standard Performance Evaluation Corporation. SPEC benchmarks. URL: http://www.spec.org/.

[28] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[29] Transaction Processing Council. TPC benchmarks. URL: http://www.tpc.org/.

[30] US-CERT. US-CERT vulnerability notes database. URL: http://www.kb.cert.org/vuls.

[31] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[32] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *MICRO '04*, Dec. 2004.

[33] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA '04*, June 2004.

Benchmarking Bug Detection Tools

Barmak Meftah • Vice President, Engineering, Fortify Software •
barmak@fortifysoftware.com

The best benchmarks serve non-experts by producing simple, easy-to-compare results regardless of the complexity of the solution. I propose the following three principles for creating a benchmark for bug detection:

1) Take the user's perspective. A bug detection benchmark should measure the following properties of a tool above all else:

- Ability to detect an explicit set of bugs that the community deems important.
- Ability to produce output that can be consumed efficiently (often expressed as "give me few false alarms").
- Performance (both capacity and execution time).

2) Name the scenario. Consider the TPC-C benchmark, the most widely used standard for evaluating database performance:

"In the TPC-C business model, a wholesale parts supplier (called the Company below) operates out of a number of warehouses and their associated sales districts… Each warehouse in the TPC- C model must supply ten sales districts, and each district serves three thousand customers. An operator from a sales district can select, at any time, one of the five operations or transactions offered by the Company's order-entry system." (From http://www.tpc.org/tpcc/detail.asp)

For bug detection tools, interesting scenarios might include:

- A network server that speaks a standard protocol (http, ftp, smtp, etc.)
- A set of device drivers
- A web-based enterprise application
- A privileged system utility.

3) Start with documented bugs (and their solutions). The body of open source applications is large enough and rich enough that a bug detection benchmark should never suffer the complaint that the included bugs do not represent realistic practices or coding styles. The benchmark code need not be drawn verbatim from open source applications, but the pedigree of a each bug should be documented as part of the benchmark. Because false alarms are such a concern, the benchmark should include code that represents both a bug and the fix for the bug.

Creating a benchmark is hard. It requires making value judgments about what is important, what is less important, and what can be altogether elided. These judgments are harder still for people who value precision, since computing a benchmark result will invariably require throwing some precision away. We should not expect to get it all right the first time. Any successful benchmark will inevitably evolve in scope, content, and methodology. The best thing we can do is make a start.

Benchmarks should not be made to serve experts or connoisseurs. The target audience for a benchmark should be non-experts who need a way to take a complex topic and boil it down to a simple one. Domain experts are drawn to building benchmarks partially because they help focus the world's attention on their problem. Invariably this attention causes the experts to put a lot of thought and effort into how to come out on top in the benchmark rankings. The winner of the 2nd place trophy will always bemoan the benchmark: "it ignores the subtlety of the topic, the area where my approach truly shines." But if the benchmark is good, these complaints will be quickly forgotten. The benchmark will focus research on problems that benefit the field.

Experts appreciate benchmarks because they can draw the world's attention to problems they consider important. Invariably, the experts focus on achieving a top ranking in the benchmark, rather than on the overall usefulness of the solution that is being benchmarked. This creates a situation in which everyone except the winner of the benchmark bemoans the fact that the benchmark did not accurately measure the true robustness of their solution.

The best benchmarks serve non-experts by producing simple, easy-to-understand results regardless of the complexity of the solution. This talk discusses three principles for creating a useful and easily understood benchmark for bug detection tools.

Position Paper: A Call for a Public Bug and Tool Registry

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

The foundation of the scientific method is the experimental, repeatable validation of hypotheses. However, it is currently extremely difficult to apply this methodology to tools and techniques for finding bugs in software. As a community, we are suffering from two major limitations:

1. It is extremely difficult to quantitatively measure the results of a tool. We can easily count the number of warnings generated, but on a large code base it is extremely difficult to categorize all of the warnings as correct or as false positives, and to correlate them with underlying software bugs. One warning may correspond to multiple underlying bugs, or one bug may yield multiple warnings. Even identifying separable "bugs" can be problematic, since code in isolation can be correct in some sense but interact incorrectly with other code.

   Moreover, we have almost no objective way of measuring false negatives, except by comparing with other tools. But even this approach is problematic: In a recently published paper [1], we compared a number of different bug finding tools for Java, and we found that even when the tools looked for the same kind of error (e.g., concurrency errors or null pointer errors), they report different warnings in different places, most likely due to differing design tradeoffs between false positives and false negatives. Finally, we often have no independent information about the severity of bugs, making it difficult to decide if one error is more important than another. For example, null pointer errors are bad, but they often cause a program to crash at the dereference site, making identifying at least the crash site easy. Whereas race conditions that corrupt internal program logic are extremely difficult to track down, yet may not even violate standard type safety.

2. It is extremely difficult to reproduce others' results. If tools are run on open-source software, then it is easy to get the code for comparison. But it may be hard to duplicate the experimental configuration (did it compile with a different set of header files or different flags?). And even the most thorough description on paper omits by necessity many small details that can have a significant impact on the behavior of a tool. When tools are run to produce the results in a paper, they are often not made publicly available, and often if they are made available later, they may have undergone changes from the original description in the paper. Finally, reimplementing tools and reevaluating them is most likely not considered publication-worthy in top conferences—at least, not without a strong result, like showing that previous claims about the technique are false. Some seemingly unimportant details can also make reproducing experiments extremely difficult, even if the tools are made available. Changes in compilers and runtime environments over time can make building the tools difficult. Moreover, small changes in the tools' target can render them unusable. For example, in [1], we found that language drift over time made it difficult to apply the different tools—tools that ran correctly on Java 1.3 code did not accept Java 1.4 code, often for minor reasons. Even without an "official" change, tools sometimes need to be tweaked—we have found that handling new features of gcc required (minor) changes to a C front end.

Given this state of affairs, I propose that our community should adopt two policies. First, we should develop an (initially) small, shared registry for storing open source code, in raw and possibly preprocessed/munged form, along with detailed information about bugs. By keeping the registry small (at least for a few years), we could focus effort on a set of programs, making it feasible to do much more detailed evaluation of false negatives in particular. It would also be useful to get bug information from developers. One solution might be to develop or co-opt a freely-available bug tracking system that both provided advantages over current solutions and encouraged developers to report bugs and patches in a form amenable to automated extraction. (I do not know what such a system would look like.)

Second, we should strongly encourage researchers to make their tools publicly available for comparison purposes. For developers of open source software, this would be easy, but we should still cultivate a community in which the tool is made available when the paper is published. For developers of proprietary software, the registry would provide a standardized test bed, and the raw output of the tool would be made public at publication time.

References

[1] N. Rutar, C. B. Almazan, and J. S. Foster. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, Saint-Malo, Bretagne, France, Nov. 2004.

2005 SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools

Bug Specimens are Important

Jaime Spacco, David Hovemeyer, William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD, 20742 USA
{jspacco,daveho,pugh}@cs.umd.edu

June 10, 2005

> science *n. The observation, identification, description, experimental investigation, and theoretical explanation of phenomena.*

To perform a scientific experimental investigation of software defects, we need bugs.

Lots of them. Thousands only begins to cover it.

Pinned to the wall, embedded in blocks of clear plastic, with a whole cabinet full of note cards about them.

So, how are we going to make that happen?

Studying Software Defects

Many papers on software reliability don't even address the issue of software defects, and instead report on numbers such as the size of the points-to relation, with a hope that the numbers will be relevant to software defect detection.

But when papers do report on number of bugs found, often they only report the "number of bugs found".

Unfortunately, companies now now feel that the exact bugs they can identify with their tools is a trade secret, and no longer make such information publicly available.

Even the exact nature of the bugs identified by research papers is often unclear: many different things can be classified as a null pointer or synchronization error, and without being able to examine which bugs are being classified as which, it is very difficult to compare two different analysis techniques. The use of heuristics to eliminate false positives or unimportant bugs makes it hard even to compare bug counts.

Work on array data dependence analysis showed that many things other than proposed techniques can substantially results presented in a paper. For example, the kind of induction variable recognition used often had a far greater impact on the accuracy of the analysis than the actual array data dependence analysis technique used. I would not be surprised to find the same of software defect detection techniques.

OK, fine, so we need to come up with some benchmarks.

But I'm not sure if benchmarks will cut it.

Most benchmarks are lousy. Benchmarks can be OK for allowing you to compare two different techniques. But conventional benchmarks can be really lousy as a basis for trying to build an experimental understanding of a phenomena. If we put together a benchmark of 10 programs with a total of 14 null pointer dereferences, what does that tell us? Not much.

The problem is that we don't understand software defects enough to craft small benchmarks that characterize the types of problems we want to be able to detect with software defect detection tools. Some special cases, maybe (e.g., format string vulnerabilities).

But if we want to understand the larger universe of software defects, we need huge collections of software defects. This, of course, means a huge collection of software. And open source repositories can be a good source of software for research.

But we also need to have some kind of ground truth as to what is a bug, and how important the bug is. One possible source for this is bug databases for open source projects. These bug databases can be pretty noisy, and matching the bug reports with the actual lines of code containing the defect or the lines changed in order to fix the defect is a challenging research

problem.

To use open source projects as a basis for studying software defects, we need

- Results on specific and available versions of open-source software artifacts.
- Detailed bug reports published in a easy, machine readable format.

To be truly successful, our community also needs to choose some specific versions of specific open-source software artifacts. This would allow

- Artifacts to be examined by multiple researchers.
- Development of standard interchange format for labeling defects and warnings in the artifacts.
- Some kind of shared information about confirmed defects found in the artifact. These confirmed defects would arise from bug databases, change logs, unit tests, or manual inspections.

The Marmoset Project

In addition to using open source, widely used, software artifacts for benchmarking, at the University of Maryland we are using student code as an experimental basis for studying software defects. This work is taking place as part of the Marmoset project, which is an innovative system for student programming project submission and testing. The Marmoset system provides significant technical, motivational and pedagogical advantages over traditional project submission and grading systems, and in particular helps students strengthen their skills at developing and using testing.

But of more relevance, the Marmoset system allows us to have students participate in research studies where we collect every save of every file as they develop their programming projects. These each are added to a database, and we compile and run all of the unit tests on each compilable snapshot. We collect code coverage information from each run, and also apply static analysis tools such as FindBugs to each snapshot. Figure 1 shows some data from two semesters of our second semester OO programming course (taught in Java).

We have found the tests results to be uniquely valuable. It starts letting us have a good approximation for ground truth. But more importantly, we can look for defects we weren't expecting. We didn't expect to see so many ClassCast and StackOverflow errors. By sampling the snapshots where those errors occurred, we were able to discover new bug patterns, implement them as static analysis rules, validate them on student code and use them to find dozens of serious bugs in production code.

# snapshots	108,352
# compilable	84,950
# unique	68,226
# test outcomes	939,745
# snapshots with exception:	
NullPointer	11,527
ClassCast	4,705
IndexOutOfBounds	3,345
OutOfMemory	2,680
ArrayIndexOutOfBound	2,268
StringIndexOutOfBound	2,124
NoSuchElement	2,123
StackOverflow	2,023

Figure 1: Marmoset Data from CMSC 132

We still have lots of work to do; we've only started to collect code coverage information from our test runs, and still have to integrate it into our analysis. The correspondence between defects and exceptions under test isn't as direct as we would like, because one fault can mask other faults, or faults can occur in code not covered by any test.

Still, we are very excited about the data we are collecting via the Marmoset project and would love to talk to other researchers about sharing the data we've collected and rolling the Marmoset system out to other schools to allow them to start collecting data for the Marmoset project as well.

NIST Software Assurance Metrics and Tool Evaluation (SAMATE) Project

Michael Kass

Computer Scientist
National Institute of Standards and Technology (NIST)
Information Technology Laboratory

<michael.kass@nist.gov>

Introduction

Software metrics are a means of assuring that an application has certain attributes, such as adequate security. Some software assurance techniques are scanning source code, byte code, or binaries, penetration testing, "sandbox" testing. Source code, byte code, and and binary scanners in particular are important in evaluating a software product because they may detect accidental or intentional vulnerabilities such as "back doors". Additionally, software metrics are essential to help determine what effect, if any, a change in the software development process has on the software quality.

The U.S. Department of Homeland Security is concerned about the effectiveness of software assurance (SA) tools. When an SA tool reports software vulnerabilities, or the lack thereof, for a software product, to what degree can the user be confident that the report is accurate? Does the tool faithfully implement SA checking techniques? What is the rate of false positives and the rate of false negatives for those techniques? Standard testing methodologies with reference matter may help measure a tool's effectiveness.

A more fundamental question is, given that some technique has been used to examine a software product, how much confidence should the user have in the product? In other words, how well do techniques actually measure security, correctness, robustness, etc.? What assurance level can be assigned? How much do different techniques overlap in detecting the same problems (or lack thereof)?

NIST is tasked by DHS to help define these needs. The Software Assurance Metrics and Tool Evaluation (SAMATE) program [1] is designed to develop metrics for the effectiveness of SA techniques and tools and to identify deficiencies in software assurance methods and tools.

This paper is targeted at the community of researchers, developers and users of software defect detection tools. In particular, this paper addresses the technical areas where NIST can provide SA tool evaluation support.

Background

The performance, effectiveness, and scope of SA tools vary significantly. For instance, although a tool may be generally classed as a "source code scanner", the scanning methodology employed, the depth and rigor with which that tool identifies software flaws and potential vulnerabilities may be quite different. In addition, different vendors make different trade-offs in performance, precision, and completeness based on the needs of different industrial segments.

Today tool vendors develop and test internally against their own test material and tool metrics. The ability to find code flaws and vulnerabilities is an important metric in measuring a tool's effectiveness. However, there is no common benchmark. A set of common, publicly available programs would allow vendors to independently check their own progress. Small tool producers and researchers would especially benefit from having a collection maintained and checked. Users would be more confident in the capabilities of different tools and be able to choose the tool(s) that are most appropriate for their situation.

Finding code flaws may not be the only metric. How useful is a tool that generates a large number of false positives? Or conversely, how effective is a tool that generates few false positives, but misses many problems (many false negatives)? These concerns must be factored in, too.

Publicly available work is already being done in the area of SA tool evaluations and metrics ([TSAT] and [ASATMS]). NIST is examining these and other existing bodies of work as possible sources of contribution to the SAMATE specifications, metrics and test material.

NIST's Information Technology Laboratory has developed or helped develop specifications and test suites for numerous technologies, including XML, PHIGS, SQL, Smart Card, and computer forensic tools. The highly successful NIST Computer Forensic Tools Testing (CFTT) project [2] is a model of tool testing that can be applied to the evaluation of the effectiveness of SA tools. The CFTT framework defines a taxonomy of forensic tool functions, functional specifications of expected tool behavior, and metrics for determining the effectiveness of test procedures.

Vendors are properly concerned about reports on their product. NIST, a part of the U.S. Department of Commerce, is a "neutral" party in the development of testing specifications and test suites. Because NIST's mandate is to support U.S. commerce and business, our role in testing is to help companies improve the quality of the products they bring to market. NIST is not "consumer reports", and does not endorse one company's product over another.

The SAMATE roadmap

By serving as neutral party maintaining an open repository for SA tool testing methods and matter, NIST will provide a common resource for SA tool

vendors, researchers and users to measure the suitability and effectiveness of a particular tool or technique.

The SAMATE project will provide an open, free, publicly reviewed resource to SA tool vendors, researchers and users that will include:

- A taxonomy of classes of SA tool functions
    - NIST will help find or develop a taxonomy based on the state of the art in software assurance tools and techniques
- Workshops for SA tool developers and researchers and users to prioritize particular SA tool functions
    - Priority may be based upon commonality, criticality, cost efficiency or other factors
    - This list will determine areas of focus and specification development
- Specifications of SA tool functions
    - Based upon focus group results, detailed specifications of functions for particular classes of SA verification will be developed
- Detailed test methodologies
    - How and which reference applications to use
    - Well-defined counting and computing procedures
    - Associated scripts and auxiliary functions
- Workshops to define and study metrics for the effectiveness of SA functions
    - Follow-on workshops to critique methodologies and formalize metrics for SA tools based upon experience
    - Incorporate ongoing research and thinking
- A set of reference applications with known vulnerabilities
- Publish papers in support of the SAMATE metric
    - The methodology used to define the functional specifications, test suites, test reports and definition of SA tool metrics will be published by NIST for peer review by the community

Immediate Goals

By introducing the SAMATE project at this workshop, NIST's goals are to:

- Solicit participation in upcoming NIST workshops to prioritize functional areas of testing
- Identify existing taxonomies, surveys, metrics, etc.
- Solicit contributions of example program suites
- Discuss issues of importance to vendor, user, and researcher participation

Bibliography

[TSAT] *Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code,* Misha Zitser, Richard Lippmann, Tim Leek, Copyright © 2004, ISBN 1-58113-855-5, Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004. ACM 2004, 97-106, http://portal.acm.org/citation.cfm?doid=1029911

[ASATMS] *DRAFT Application Security Assessment Tool Market Survey Version 1.0,* U.S. Defense Information Systems Agency (DISA), Washington D.C., 2002, https://iase.disa.mil/appsec/index.html

---

[1] http://www.cftt.nist.gov , web page for the NIST Computer Forensic Tool Testing Project

[2] http://samate.nist.gov , web page for the NIST SAMATE Project

Deploying Architectural Support for Software Defect Detection in Future Processors

Yuanyuan Zhou and Josep Torrellas
Department of Computer Science, University of Illinois at Urbana-Champaign
{yyzhou,torrellas}@cs.uiuc.edu

1 Motivation

Recent impressive advances in semiconductor technology are enabling the integration of ever more transistors on a single chip. This trend provides a unique opportunity to enhance processor functionality in areas other than performance, such as in easing software development or enhancing software debugging.

Exploring architectural support for software debugging is a research direction of great promise. First, it enables new tradeoffs in the performance, accuracy, and portability of software defect detection tools, possibly allowing the detection of new types of defects under new conditions. Secondly and most importantly, it opens up a new possibility: on-the-fly software defect detection in *production runs*. This last area has been under-explored due to the typically large overheads of many software solutions.

Table 1 briefly summarizes the architectural supports for software defect detection that our research group has recently proposed.

Architectural Support	Description	Overhead
ReEnact [1]	Extend the communication monitoring mechanisms in thread-level speculation to detect and characterize data races automatically on the fly.	1% to 13%
iWatcher [4]	Associate program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead without going through the OS.	4% to 80%
AccMon [3]	Detect general memory-related bugs by extracting and monitoring the set of instruction PCs that normally access a given monitored object.	0.24X to 2.88X
SafeMem [2]	Exploit existing ECC memory technology to detect memory corruption and prune false positives in memory leak detection.	1% to 29%

Table 1: Architectural supports for software defect detection recently proposed by our research group.

In general, using architectural support to detect software defects provides several key advantages over software-only dynamic approaches:

(1) Performance. Architectural support can significantly lower the overhead of dynamic monitoring if it eliminates the need for extensive code instrumentation. Such instrumentation may even interfere with compiler optimizations. Moreover, the hardware can be used to speed up certain operations. For example, AccMon uses a hardware Bloom filter to filter 80-99% of the accesses that would go to a software monitor function. As shown in Table 1, hardware support enables dynamic detection of bugs with orders-of-magnitude less overhead than commonly existing software-only tools.

(2) Accuracy. Architectural support enables ready access to execution information that is often hard to obtain with software-only instrumentation. An example is all the accesses to a monitored memory object and only those — instrumentation-based tools need to check more memory accesses due to pointer aliasing problems. This capability is leveraged by iWatcher [4]. Another example of hard to obtain information is the exact interleaving of the accesses that caused a data race in a multithreaded program running at production speeds. This capability is leverages by ReEnact [1] to detect production-run data races.

(3) Portability. Architectural support is typically language independent, cross module (capable of checking for bugs in third-party libraries), and easy to use with low-level system code such as the operating system. Moreover, it can be designed to work with binary code without recompilation.

2 Deployment Challenges and Plans

Due to the above advantages, it is highly desirable for software developers to be able to use architectural support to assist in detecting software defects. Of course, to make this happen, we need the cooperation of processor design companies such as Intel, IBM, AMD, or Sun Microsystems. To help improve the chances that one or more of these companies finds it attractive to include architectural support for software debugging in their processors, we put forward the following suggestions:

(1) The research communities working on software defect detection tools and on computer architecture can work together to identify (i) what are important or difficult-to-catch bugs (e.g. data races) that need architectural support? and (ii) what architectural extensions can be added to exisiting processors to help detect these bugs?

The architectural supports that are more likely to succeed are those that have one or several of the following features: simplicity, generality, reconfigurability and leverage existing hardware. Simple designs are those that require modest extensions to current processor hardware, such as iWatcher or SafeMem. General designs are those that can be used for multiple purposes, such as debugging and profiling, as also exemplified by iWatcher. Reconfigurable designs can easily be disabled, enabled or reconfigured for other purposes so that hardware vendors can manufacture only one type of hardware but can configure it for different users. Finally, designs that leverage existing hardware and use it for other purposes are likely to be successful. For example, SafeMem makes novel use of ECC memory for bug detection.

(2) Software companies such as Microsoft with significant leverage on processor companies can push the latter to provide the necessary architectural hooks in their processors. Software companies can also be willing to buy some special "testing" processors (processors with sophisticated bug detection support) that are priced higher than regular processors. Similarly, software companies can motivate customers to buy such processors by offering them incentives such as lower license fees or better services.

(3) Interest in the topic of architectural support for software productivity and debuggability can be broadened through workshops, conferences, and funding efforts in this area.

REFERENCES

[1] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA*, 2003.

[2] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *HPCA*, Feb 2005.

[3] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *Micro*, Dec 2004.

[4] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA*, 2004.

Using Historical Information to Improve Bug Finding Techniques

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

Tools used to identify bugs in source code often return large numbers of false positive warnings to the user. These false positive warnings can frustrate the user and require a good deal of effort to identify. Various attempts have been made to automatically identify false positive warnings. We take the position that historical data mined from the source code revision history is useful in refining the output of a bug detector by relating code flagged by the tool to code changed in the past.

1 Introduction

Tools used to identify bugs in source code often return large numbers of false positive warnings to the user. True positive warnings are often buried among a large number of distracting false positives. By making the true positives hard to find, a high false positive rate can frustrate users and discourage them from using an otherwise helpful tool.

Prior research has focused on inspecting the code surrounding the warning producing code with the assumption that a tool may produce a large number of false positive warnings very close together in the code [2].

More recent work has added user driven feedback to refine the ranking of warnings [1]. As the user inspects a warning and classifies it as either a bug or false positive, the remaining warnings are re-ranked. The intuition is that warnings that are part of some grouping are likely to all be either bugs or false positives. This approach has the advantage of giving the user a preliminary ranking of the warnings and then refining that ranking with as close to true fact as one can get: the opinion of the user.

2 Repository Mining as a Solution

We believe we can use data mined from the source code repository to help determine the likelihood of a warning being a true bug or a false positive by relating code flagged by warnings to code that was changed in the past. With the source code repository we have a record of each source code change. We can determine when a piece of code is added and, more importantly, when code is changed. The code changes may be used to highlight bug fixes through the life of the project.

Examining the code changes and the state of the code before and after the change may allow us to match previous code changes to warnings produced by a bug finding tool. Warnings could be matched to code changes in a number of ways. The functions invoked, the location in the code (module/API/function) or the control or data flow may be used to link the flagged code to the code from the repository. Warnings that flag code similar to code snippets that have been changed in the past may be more likely to be true positives.

In [3] we show how historical data can be used to rank warnings produced by a static analysis tool with a particularly high false positive rate. We mined the source code repository to determine which functions in a software project had a particular type of bug fix applied to their invocation. We produced a ranking of the warnings where warnings involving functions flagged with a bug fix were pushed to the top of the list. Our approach produced a ranking with a higher density of likely bugs near the top as compared to a more naïve ranking scheme.

We investigate function usage patterns mined from the software repository in [4]. Here we are trying to identify from the repository how functions should be invoked in the source code with respect to each other. We believe that discrepancies between how we expect functions to be called and how they are invoked in the current version of the software could be used to highlight code that may be incorrect. These discrepancies may indicate confusion on the part of the programmer. Warnings produced for these snippets of code may be more likely to be true bugs.

A ranking based on the past history is similar to the idea of ranking based on user feedback. However, when using past history the feedback is automatically generated (and could be augmented by interactive user feedback). The initial ranking the user is given will have the benefit of past code changes.

3 References

[1] Kremeneck, T., Ashcraft, K., Yang, J., Engler, D., Correlation Exploitation in Error Ranking, In *Proceedings of Twelfth ACM SIGSOFT Symposium on Foundations of Software Engineering* (SIGSOFT'04) Newport Beach, CA, USA, Nov. 2004.

[2] Kremeneck, T., Engler, D., Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, In *Proceedings of 10th Annual International Static Analysis* Symposium, (SAS '03) San Diego, CA, USA, June 2003.

[3] Williams, C. C., Hollingsworth, J. K., Bug Driven Bug Finders, In *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[4] Williams, C. C., Hollingsworth, J. K., Recovering System Specific Rules from Software Repositories, In *Proceedings of International Workshop on Mining Software Repositories* (MSR '05), St. Louis, MO, USA, May 2005.

Locating defects is uncertain

Andreas Zeller
Department of Computer Science
Saarland University, Saarbrücken, Germany
zeller@acm.org

ABSTRACT

While numerous techniques for detecting the *presence* of defects exist, it is hard to assign the defect to a particular location in the code. In this position paper, I argue that this is necessarily so, and that locating a defect is inseparable from designing a fix—in other words, writing a correct program. This leads to an inherent imprecision, which can be dealt with by ranking locations according to their defect probability.

1. ERRORS AND CAUSES

To explain how failures come to be, one needs two central terms: *errors* and *causes*. An *error* is a deviation from what is correct, right, or true. If we see an error in the program outcome (the *failure*), we can trace back this failure to earlier errors in the program state (*faults* or *infections*), until we finally reach the defect—an error in the program code. This defect causes the initial infection, which propagates until the infection becomes visible as a failure.

In this infection chain, a *cause* is an event without which a subsequent event (the *effect*) would not have occurred. Thus, if the program code had been correct, it would not have caused the infection, which again would not have led to the failure. This causality is normally proven by re-testing the program after the defect has been fixed: If the failure no longer occurs, we have proven that the original defect indeed has caused the failure.

While causality is easy to explain (and easy to verify), the term *error* becomes less certain the further one goes back the chain of events. The key issue is: to decide that something is erroneous, one needs a specification of what is correct, right or true. Telling whether a program *outcome* is a failure is the base of testing—and quite straight-forward. Telling whether a program *state* is infected already requires appropriate conditions or representation invariants. Telling whether some program *code* is incorrect, finally, becomes *more and more difficult as granularity increases.*

2. DEFECTS AND GRANULARITY

Why does the location of a defect become less certain with increasing granularity? If a computation P fails, we know it must have some defect. Let us assume that P can be separated into two sub-computations $P = P_1 \circ P_2$, each at a different location. Then, we can check the result of P_1, and determine whether it is correct (which means that P_2 has a defect) or not (then P_1 has a defect).

Now assume we can decompose P into n executed procedures, or $P = P_1 \circ \cdots \circ P_n$. By checking the outcome of each P_i, we can assign the defect location to a single precise P_j. Obviously, this means specifying the postconditions of every single P_i, including general obligations such as representation invariants.

Let us now assume we can decompose P into m executed lines, or $P = P_1 \circ \cdots \circ P_m$. To locate the defect, we now need a specification of the correct state at each executed line—for instance, the *correct program.* Thus, to precisely locate the defect, we need a specification that is precise enough to tell us where to correct it.

In practice, such a specification is constructed *on demand:* When programmers search for a defect, they reason about whether this location is the correct way to write the program—and if it does not match, they fix it. Therefore, programmers do not "locate" defects; they *design fixes* along with the implicit specification of what *should* be going on at this location—and the location that is changed is defined as the defect in hindsight.

3. DEALING WITH IMPRECISION

As long as a full specification is missing (which we must reasonably assume), it is impossible to locate defects precisely, just as it is impossible to foresee how a problem will be fixed—and whether it will be fixed at all. Therefore, any defect location techniques will always have to live with imprecision. This is not a big deal; we can have our tools make *educated guesses* about where the defect might be located. And we will evaluate our tools by their power to suggest fixes that are as close to some "official" defect as possible.

However, imprecision also must be considered when *evaluating* techniques. For instance, if a technique detects that a function call does not match the function's requirements, the technique cannot decide whether it is better to fix the caller or the callee. Let us now assume that we conduct an evaluation where we have injected a defect in the callee. If the technique now flags the caller as defective, the result is evaluated as being at the wrong location, or even as a false. Nonetheless, the mismatch is helpful for the programmer.

There are entire classes of problems which cannot be located at all. For instance, assume I inject a defect which eliminates the initialization of a variable. Although there are techniques which will detect this situation, they will be unable to tell where the initialization should have taken place. Again, the evaluation will show a mismatch between predicted and expected defect location; nonetheless, the diagnosis will be helpful for the programmer.

How are we going to take this imprecision into account? I suggest to have our tools not only suggest locations, but to actually *rank* the locations by their probability to be related to the defect. The model would be an ideal programmer, starting with the most probable location, and going down the list until the "official" defect is found; obviously, the sooner the "official" defect is found, the better the tool. In a mismatch of caller and callee, both locations would end on top of the list; a missing initialization would result with the function or module containing the declaration being placed at the top. Such *code rankings* would allow us to compare individual defect-locating tools, and eventually establish standards for evaluating them.

Is a Bug Avoidable and How Could It Be Found?
Position Statement

Dan Grossman

March 2005

Using automated tools based on programming-language technologies to find software defects (bugs) is a growing and promising field. As a research area less mature than related ones such as compiler construction, there are not yet widely accepted benchmarks and evaluation techniques. Though neither surprising nor necessarily bad, the lack of evaluation standards begs the question of what criteria should guide the inevitable growth of standards by which research on automated defect detection is judged. I contend that obvious metrics such as number of bugs and seriousness of bugs found are insufficient: We should also consider the technologies that led to the bug existing and whether other existing technologies could have found the bug or rendered it irrelevant.

It is understood that a bug's seriousness is relevant though difficult to quantify. In particular, bugs that have little affect on program behavior, bugs that are more expensive to fix than leave, and bugs for which experts dispute whether the bugs are real should rightfully be devalued. Leaving such issues aside, let us assume an easy-to-use automated tool finds *serious* bugs, the sort that developers acknowledge are errors, want to fix, can fix, do fix, and believe the result is of value. I believe the research community today would deem such work a contribution, but in the future the "bar will be raised" in at least two ways.

First, we will learn to devalue bugs that are easily avoidable with known technologies. For example, suppose a static analysis finds double-free errors in C code, i.e., situations where `free` is called more than once on the same object. Further suppose all applications for which bugs were found were stand-alone desktop applications that ran as fast or faster when recompiled to use conservative garbage collection. Were these bugs worth finding and fixing? Should we require the analysis to prove useful on applications for which manual memory management is considered necessary? As another example, suppose a modified compiler uses sophisticated run-time data structures to track uninitialized memory and abort a program if such memory is accessed. Further suppose the program would run without error or performance loss if the compiler had simply initialized all memory to 0.

Second, we will learn that tools must provide an incremental benefit over the state-of-the-art. Many tools exist; how is a new one better? It should find different bugs, find them in a better way (perhaps faster or with less user intervention), or perhaps use a more elegant approach to finding the same bugs. Would compiling the code with `-Wall` lead to exactly one accurate warning for every bug found? Does the tool work only for ten-line programs that are clearly incorrect when manually examined by an expert? Is the tool strictly less powerful and more complicated than an existing tool?

In general, it may simply be that it is currently easy to succeed with automated tools for defect detection because we assume the lowest possible baseline: We find bugs in whatever code is available, which was typically compiled without compiler warnings enabled and without other tools applied to the code. It seems clear this will change, just as the compiler optimization community must now show that a new optimization is neither made-irrelevant-by nor trivially-encoded-with existing and widely used optimizations.