ABSTRACT

Title of dissertation: TOOLS AND EXPERIMENTS FOR SOFTWARE SECURITY

Andrew Ruef
Doctor of Philosophy, 2018

Dissertation directed by: Professor Michael Hicks
Department of Computer Science

The computer security problems that we face begin in computer programs that we write.

The exploitation of vulnerabilities that leads to the theft of private information and other nefarious activities often begins with a vulnerability accidentally created in a computer program by that program's author. What are the factors that lead to the creation of these vulnerabilities? Software development and programming is in part a synthetic activity that we can control with technology, i.e. different programming languages and software development tools. Does changing the technology used to program software help programmers write more secure code? Can we create technology that will help programmers make fewer mistakes?

This dissertation examines these questions. We start with the Build It Break It Fix It project, a security focused programming competition. This project provides data on software security problems by allowing contestants to write security focused software in any programming language. We discover that using C leads to memory

safety issues that can compromise security.

Next, we consider making C safer. We develop and examine the Checked C programming language, a strict super-set of C that adds types for spatial safety. We also introduce an automatic re-writing tool that can convert C code into Checked C code. We evaluate the approach overall on benchmarks used by prior work on making C safer.

We then consider static analysis. After an examination of different parameters of numeric static analyzers, we develop a disjunctive abstract domain that uses a novel merge heuristic, a notion of volumetric difference, either approximated via MCMC sampling or precisely computed via conical decomposition. This domain is implemented in a static analyzer for C programs and evaluated.

After static analysis, we consider fuzzing. We consider what it takes to perform a good evaluation of a fuzzing technique with our own experiments and a review of recent fuzzing papers. We develop a checklist for conducting new fuzzing research and a general strategy for identifying root causes of failure found during fuzzing. We evaluate new root cause analysis approaches using coverage information as inputs to statistical clustering algorithms.

TOOLS AND EXPERIMENTS FOR SOFTWARE SECURITY

by

Andrew Ruef

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Professor Jeffrey Foster
Professor Michelle Mazurek
Professor David Levin
Professor Joseph JaJa, Dean's Representative

Acknowledgments

Thank you Mike, my advisor, for your support, encouragement, feedback, criticism, and suggestions. They have summed to form a vector nudging me in the direction of becoming a scientist. Thank you to my collaborators over the years, it was fun, and when it wasn't fun, it was educational. I would like to thank the following people and institutions, the order is not important: the University System of Maryland, Oreo cookies, Anna, Dana, Earth Treks, Joman, the students and alumni of PLUM, Dr. Elizabeth Lovegrove, Hoppy, Car, Arlen, Mike, David, Mike, David, and the other Mike, The Board and Brew, Kelly, the Howard County Public Library system, Rachel, my family, Trail of Bits and affiliates, Holly, all the members of each "bad attitude" channel I am or was a member of, "The Institute for the Advancement of Memory Corruption", the University of Oregon and OPLSS, and Infected Mushroom. If you are not named here, you could assume that I thought about you but decided not to name you for privacy reasons. Especial thanks to my wife.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The computer security problems that we face begin in computer programs that we write. The theft of private information and other nefarious activities often begin with a vulnerability accidentally created in a computer program by the program's author. What are the factors that lead to the creation of these vulnerabilities? Software development and programming is in part a synthetic activity that we can control with technology, i.e. different programming languages and software development tools. Does changing the technology used to program software help programmers write more secure code? Finally, can we create technology that will help programmers make fewer mistakes?

There are many variables that could effect the presence and severity of defects in software, such as the choice of programming language, the use (or non-use) of static analyzers, the use of fuzzing or after the fact audits of the code, and development methodology, to name a few. When Equifax was hacked, resulting in the loss of millions of records, an old version of a library used in web applications, Apache Struts, was to blame [1,2]. A simple static analyzer would have revealed the presence of a component that was known to be vulnerable, if the analysis was run at some point during software development or maintenance. The Heartbleed vulnerability in

openSSL tells a different story, where a low level programming error in a C library allowed attackers to read sensitive information from SSL server applications [3].

Different methods one might use to make software more secure have different trade-offs, this dissertation is a partial enumeration and consideration of these trade-offs, across a spectrum of secure software development: from language design, to software design, and then with the design and use of tools to help secure software such as static analyzers and fuzzers. Specifically, in Chapter 2 we consider a study of secure software development. In Chapter 3, we consider how to make C a safer language. In Chapter 4, we consider improvements to numeric static analyzers. In Chapter 5, we consider the evaluation of fuzz testers. In Chapter 6, we consider root cause analysis, input minimization, and fault clustering.

A study of secure software development: Build It, Break It, Fix It

How would we experimentally study secure software development? Studying this seems useful because different software development projects (e.g. Mozilla Firefox vs. Google Chrome) seem to have different security successes and failures. However, retroactively studying their histories has some problems. In the case of Firefox and Chrome, for example, there are similarities between the projects that make a comparison seem fruitful, but sufficient differences that the two projects are probably incomparable. Both projects implement web browsers with feature parity, both enjoy popular use, and both are written in a low level programming language (C/C++). However, there are many differences as well, they have been developed

with different amounts of financial support for different periods of time, and with differing amounts of focus on security.

In Chapter 2, we describe the Build It Break It Fix It contest. Build It Break It Fix It is a programming contest designed to provide an environment in which we could study the interaction between software development and security. This contest has incentives similar to the "real world" but with controls in place for the previously identified confounds inherent in empirically analyzing software security projects. The contest was split into distinct phases: in the first phase, contestants wrote software to a specification and were scored based on that software's performance. In the next phase, contestants tried to find security bugs in the software that was written in the first phase. Contestants could write their software in any programming language. In the final phase, contestants receive evidence of security bugs in their programs and try to fix them. At the end, contestants have two scores: a break score, for the number of bugs they identified in the software of others, and a build score, which is a factor of the performance of their software in terms of both computational efficiency as well as the number of security bugs discovered during the break phase.

The fix it phase has a necessary logistical purpose: de-duplication of bugs reported during the break it phase. What if two teams discover "the same" bug in a program? What if there is one particularly bad program which has a glaringly obvious flaw that is discovered by every team? We thought that, like in the real world, it is better to discover bugs that no one else has, that breadth is good. To incentivise this behavior, we outline in the rules that "duplicate" bugs are scaled in

value proportional to the number of teams that discover it. However, this leaves us with the problem of determining if two reported bugs are the same or not. This is a question that haunts us through the rest of the dissertation, specifically in Chapter 5 and Chapter 6. In Chapter 2, we approximate the solution to this problem by presenting bug reports to the original authors and asking them to disambiguate the bugs with patches to their own programs. Here, perhaps, the incentives line up: as a contestant identifies duplicates, they lose fewer points, as points are detracted from their Build It score only for unique bugs.

In addition to providing a platform for gathering data, the contest is also a natural learning environment for secure software development. Our experiments use a student population from a massive online open course (MOOC) with a mix of demographics. Learners had many positive experiences to share from learning about software security in this environment.

The contest outcomes provide data we can analyze about the interaction between programming languages, developer experience and practices, tool usage, and security bugs [4]. Some of the evidence supports our intuitive understanding of the world. For example, writing safe C programs is very difficult, which is supported by both a casual analysis of vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database as well as evidence gathered during the Build It Break It contest. However, also during Build It Break It, we observed that C programs had better performance overall, leading us to ask, is there a way to get the best of both worlds with C? Can we achieve C's expressiveness and performance, but coupled with freedom from memory safety bugs?

Making safer languages: Checked C

The programming language community has responded to this challenge by developing a series of "safer" languages, but adoption has been slow because using these languages is costly: programmers would have to learn a safer language and then re-write all of their code in that language. We propose *extending* the C language to allow programmers to avoid security bugs like buffer overflows by having them express and reason about invariants describing the extents of memory objects. We present this extension in Chapter 3 as the Checked C project [5].

The Checked C project aims to solve the problem of spatial safety by having the programmer explicitly express the bounds for memory objects used by a C program, and checking those bounds. The bounds are expressed as new types in the C programming language: `_Array_ptr<T, 1, u>` for bounded arrays and `_Ptr<T>` for checked-nullable pointers. As an example, consider the following code declarations:

```
size_t dst_count;
_Array_ptr<char> dst : count(dst_count);
```

The `_Array_ptr<char>` type is a Checked C type for a bounds-checked array, and the `count` annotation indicates how the bounds should be computed. In this case `dst`'s bounds are stored in the variable `dst_count`, but other specifications, such as pointer ranges, are also possible. Checked type information is used by the compiler to either prove that an access is safe, or else to insert a bounds check when such a proof is too difficult.

To help with porting old C code to use these new types, we develop an au-

tomatic re-writer that will translate old C code into new C code that uses these types. We evaluate both the re-writer and the Checked C language against several different open source projects and benchmarks, examining how many pointer types can be converted automatically, what overhead is introduced by dynamic checking, and the difficulty of manual conversion. We find that $\sim 30\%$ of the pointer values in open source code can be converted automatically to `_Ptr<`T`>`, and that the run-time overhead of our checking is modest at $\sim 8\%$.

Improving numeric static analyzers: managing disjunctive invariants

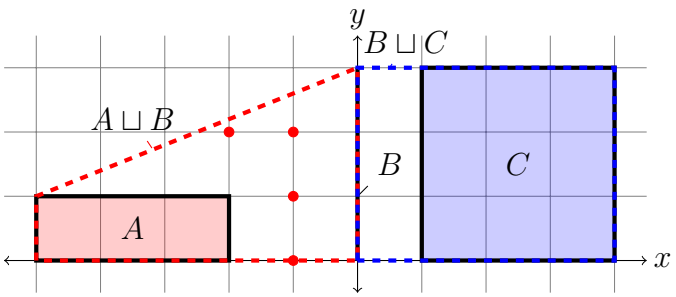
The Checked C re-writer is a static analyzer that performs a mixture of local and global reasoning on pointer properties. This is useful for some safety properties, but not all. Numeric static analyzers can help with finding and proving other properties like side channels [6], properties about NUL-terminated strings [7], and array bounds checks [8].

In Chapter 4, we look at how to extend a numeric static analysis to allow for disjunctive constraints [9]. This allows the inference and use of non-convex constraints in describing numeric invariants, which could be much more precise. Our contribution is concerned with how these constraints should be abstracted to balance precision and performance.

Frequently, numeric static analysis is called upon to produce program invariants that describe the relationship between numerics in the program. These invariants take the form of, for example, $x < 0$ or $x < y$. Individual variables (x,y) are

"abstracted" using a representation that soundly over-approximates the values that the variable might take on. These approximations are referred to in the literature as "domains". One domain that is frequently used is that of convex polyhedra. When a static analyzer is using this domain to approximate the values of program variables, the precision of the invariants is constrained by the precision of the convex polyhedra.

```
assume(0 <= y <= 3);
assume(-5 <= x <= 4)
    ;
if(x <= -2) {
  assume(y <= 1);
  // A
} else if(x == 0) {
  // B
} else if(x >= 1) {
  // C
} else {
  assume(false);
}
```



a b

Figure 1.1: Example of merging disjuncts. (a) Program that produces three disjuncts. (b) Three disjuncts shown as three convex polytopes. Red (resp. blue) dashed lines show the merge results of A and B (resp. B and C). Dots show integer points added in merging.

A loss of precision results in false alarms from the static analyzer. Precision loss can arise when the program contains branches and the same numeric variable

is manipulated in mutually exclusive branches. When the two branches "merge" the analyzer must also merge the abstractions of the variables. For an example, consider Figure 1.1. In this example, we have precise abstractions of the values that may be taken by x and y in each branch. However, when we merge them together, we lose precision and our invariants could lead us to false alarms. Avoiding this is challenging. We would need to change our abstraction to allow for a non-convex representation, which has other costs. Another option is to "lift" our abstraction up from a single polyhedra to a list of polyhedra, where the list signifies a disjunction of constraint systems. This allows us to maintain the best of both worlds: when considering only two abstractions, we enjoy the benefits of a convex representation, but when considering two disjunctions, we have some practical benefits of concavity.

Though now we have a problem: how do we manage which disjunctive elements to keep separate, and which to join together? We introduce the notion of "volume affinity" as a heuristic to manage which disjunctive elements to join. This heuristic is different from previous heuristics in that it considers the semantics of polytopes via their volume, rather than relying on syntactic elements as previous work does. This heuristic guides the choice of which polytopes under management by the disjunctive domain are merged and which are left stand-alone.

Computing the volume of higher dimensional polytopes is complicated. We use two different methods: approximating the real relaxation volume via Markov Chain Monte Carlo sampling, and computing the exact count of contained integer points via a conical decomposition, implemented in an existing open source library, libbarvinok [10].

We implement this heuristic as an abstract domain on top of an existing C static analyzer, *CRAB* [11]. We evaluated the finished analyzer on a benchmark suite of C programs drawn from the verification community, and compare to prior syntactic measures along dimensions of both precision and performance. We also compare using approximate real volume to exact integer point counts as a heuristic. We find that, counter-intuitively, using exact algorithms is both more precise and more efficient than using approximate real volume when managing disjunctive numeric domains.

Evaluating fuzz testing

Static analysis is one method to find bugs and prove their absence, but fuzzing and dynamic analysis are popular in both industry and research. In Chapter 5, we consider the problem of *evaluating* fuzzing.

Fuzzing is used to identify bugs in programs by automatically generating and executing a large quantity of test cases in the program. Fuzzing has a good track record in the software security ecosystem, but exactly why and under what circumstances fuzzers will be successful is not well understood. Sometimes a fuzzing campaign can find 2 bugs, and other times it can find 4 bugs, or even 6 bugs. The fuzzing process is randomized, so there being variance makes sense, however this poses a problem for the interpretation of papers from the fuzzing literature and complicates the matter of evaluating fuzzing in general.

We conducted a survey of existing fuzzing papers to identify how well the

existing research literature accounts for variance in their evaluation of fuzzers, as well as how explicitly their evaluations call out and control for different "knobs" or configurable parameters of the fuzzing process. We found that few papers have "tight" evaluation stories. We conducted our own experiments with fuzzing, but with a large number of independent trials while also varying knobs, and configuration parameters. The experiments produce data demonstrating a high variance in the number of crashes found from one trial to the next. Analyzing our experimental setup, we create a check list for fuzzing evaluators to follow when designing their evaluation [12].

Root cause analysis

In Chapter 6, we consider the problems of test case de-duplication, root cause analysis, and input minimization, as applied to fuzzing. Even when the problems of Chapter 5 are put to rest, we still have the problem of interpreting the results of a given fuzzing campaign. Fuzzers will find inputs that crash the program (or violate some other property) but they will not immediately disambiguate two inputs that exhibit "the same bug." What exactly does it mean for two bugs to be "the same?"

We present a strategy to use historical commit information to dis-ambiguate crashing inputs in one setting. Then, we analyze the root causes of the bugs identified and discuss why it is difficult for current techniques to dis-ambiguate, or cluster, inputs that trigger those bugs. The historical analysis consumes a large amount of resources as it requires the cross product of the crashing inputs found with the num-

ber of "buildable" commits over a long period. This analysis gives us a good sense of the actual number of bugs found during our fuzzing campaign and the actual number of bugs found on a trial by trial basis.

We investigate existing methods to perform crashing input clustering, including stack hashing, minimization, and encoding of coverage information as feature vectors to off the shelf clustering algorithms. We report on the accuracy of each of these measures, compared to ground truth established by our historical analysis.

Contributions

This dissertation contributes:

- Build It Break It Fix It, a contest for measuring software security
- Checked C and an automatic re-writing tool that converts C programs to Checked C
- A novel disjunctive abstract domain for numeric invariant generation via abstract interpretation
- A framework for evaluating fuzz testers
- An investigation of methodologies to measure the effectiveness of fuzzers

Chapter 2: Build It, Break It, Fix It

Cybersecurity contests [13–17] are popular proving grounds for cybersecurity talent. Existing contests largely focus on *breaking* (e.g., exploiting vulnerabilities or misconfigurations) and *mitigation* (e.g., rapid patching or reconfiguration). They do not, however, test contestants' ability to *build* (i.e., design and implement) systems that are secure in the first place. Typical programming contests [18–20] do focus on design and implementation, but generally ignore security. This state of affairs is unfortunate because experts have long advocated that achieving security in a computer system requires treating security as a first-order design goal [21], and is not something that can be added after the fact. As such, we should not assume that good breakers will necessarily be good builders [22], nor that top coders necessarily produce secure systems.

This chapter presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification that includes security goals. The software is scored for being correct, efficient, and feature-ful. The second phase, *Break-it*, asks teams to find defects in other teams' build-it submissions. Reported defects, proved via

test cases vetted by an oracle implementation, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect.

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. BIBIFI's structure and scoring system also aim to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to look for bugs broadly (in many submissions) and deeply (to uncover hard-to-find bugs).

In addition to providing a novel educational experience, BIBIFI presents an opportunity to study the building and breaking process scientifically. In particular, BIBIFI contests may serve as a quasi-controlled experiment that correlates participation data with final outcome. By examining artifacts and participant surveys, we can study how the choice of build-it programming language, team size and experience, code size, testing technique, etc. can influence a team's (non)success in the build-it or break-it phases. To the extent that contest problems are realistic and contest participants represent the professional developer community, the results of

this study may provide useful empirical evidence for practices that help or harm real-world security. Indeed, the contest environment could be used to incubate ideas to improve development security, with the best ideas making their way to practice.

This chapter studies the outcomes of three BIBIFI contests that we held during 2015, involving two different programming problems. The first contest asked participants to build a *secure, append-only log* for adding and querying events generated by a hypothetical art gallery security system. Attackers with direct access to the log, but lacking an "authentication token," should not be able to steal or corrupt the data it contains. The second and third contests were run simultaneously. They asked participants to build a pair of *secure, communicating programs*, one representing an ATM and the other representing a bank. Attackers acting as a man in the middle (MITM) should neither be able to steal information (e.g., bank account names or balances) nor corrupt it (e.g., stealing from or adding money to accounts). Two of the three contests drew participants from a MOOC (Massive Online Open Courseware) course on cybersecurity. These participants (278 total, comprising 109 teams) had an average of 10 years of programming experience and had just completed a four-course sequence including courses on secure software and cryptography. The third contest involved U.S.-based graduate and undergraduate students (23 total, comprising 6 teams) with less experience and training.

BIBIFI's design permitted it to scale reasonably well. For example, one full-time person and two part-time judges ran the first 2015 contest in its entirety. This contest involved 156 participants comprising 68 teams, which submitted more than 20,000 test cases. And yet, organizer effort was limited to judging whether the few

hundred submitted fixes addressed only a single conceptual defect; other work was handled automatically or by the participants themselves.

Rigorous quantitative analysis of the contests' outcomes revealed several interesting, statistically significant effects. Considering build-it scores: Writing code in C/C++ increased build-it scores initially, but also increased chances of a security bug found later. Interestingly, the increased insecurity for C/C++ programs appears to be almost entirely attributable to memory-safety bugs. Teams that had broader programming language knowledge or that wrote less code also produced more secure implementations. Considering break-it scores: Larger teams found more bugs during the break-it phase. Greater programming experience and knowledge of C were also helpful. Break-it teams that also qualified during the build-it phase were significantly more likely to find a security bug than those that did not. Use of advanced tools such as fuzzing or static analysis did not provide a significant advantage among our contest participants.

We manually examined both build-it and break-it artifacts. Successful build-it teams typically employed third-party libraries—e.g., SSL, NaCL, and BouncyCastle—to implement cryptographic operations and/or communications, which freed up worry of proper use of randomness, nonces, etc. Unsuccessful teams typically failed to employ cryptography, implemented it incorrectly, used insufficient randomness, or failed to use authentication. Break-it teams found clever ways to exploit security problems; some MITM implementations were quite sophisticated.

In summary, this chapter makes two main contributions. First, it presents BIBIFI, a new security contest that encourages building, not just breaking. Sec-

ond, it presents a detailed description of three BIBIFI contests along with both a quantitative and qualitative analysis of the results. We made the BIBIFI code and infrastructure available to other institutions and organizations, which have run their own versions of this event; we hope that this opens up a line of research built on empirical experiments with secure programming methodologies.

The rest of this chapter is organized as follows. We present the design of BIBIFI in §2.1 and describe specifics of the contests we ran in §2.2. We present the quantitative analysis of the data we collected from these contests in §2.3, and qualitative analysis in §2.4. We review related work in §2.5 and conclude in §2.6.

Attribution and acknowledgments

This chapter was adapted from a paper appearing at ACM CCS 2016 [4]. The paper was written in collaboration with James Parker, Mike Hicks, Dave Levin, Piotr Mardziel, and Michelle Mazurek. That paper, in turn, was based on a prior paper that was additionally authored with Jandelyn Plane. Andrew and Mike conceived of the contest design and structure, which was refined through further discussion with Dave, James, Jan, and Michelle. The contest infrastructure was implemented by James and Andrew, and execution of the contests was supervised (at different times) by James, Andrew and Mike. Data analysis was conducted by James, Andrew and Michelle.

2.1 Build-it, Break-it, Fix-it

This section describes the goals, design, and implementation of the BIBIFI competition. At the highest level, our aim is to create an environment that closely reflects real-world development goals and constraints, and to encourage build-it teams to write the most secure code they can, and break-it teams to perform the most thorough, creative analysis of others' code they can. We achieve this through a careful design of how the competition is run and how various acts are scored (or penalized). We also aim to minimize the manual work required of the organizers—to allow the contest to scale—by employing automation and proper participant incentives.

2.1.1 Competition phases

We begin by describing the high-level mechanics of what occurs during a BIBIFI competition. BIBIFI may be administered on-line, rather than on-site, so teams may be geographically distributed. The contest comprises three phases, each of which last about two weeks for the contests we describe in this chapter.

BIBIFI begins with the **build-it phase**. Registered build-it teams aim to implement the target software system according to a published specification created by the contest organizers. A suitable target is one that can be completed by good programmers in a short time (just about two weeks, for the contests we ran), is easily benchmarked for performance, and has an interesting attack surface. The software should have specific security goals—e.g., protecting private information or

communications—which could be compromised by poor design and/or implementation. The software should also not be too similar to existing software to ensure that contestants do the coding themselves (while still taking advantage of high-quality libraries and frameworks to the extent possible). The software must build and run on a standard Linux VM made available prior to the start of the contest. Teams must develop using Git [23]; with each push, the contest infrastructure downloads the submission, builds it, tests it (for correctness and performance), and updates the scoreboard. §2.2 describes the two target problems we developed: (1) an append-only log; and (2) a pair of communicating programs that simulate a bank and an ATM.

The next phase is the **break-it phase**. Break-it teams can download, build, and inspect all qualifying build-it submissions, including source code; to qualify, the submission must build properly, pass all correctness tests, and not be purposely obfuscated (accusations of obfuscation are manually judged by the contest organizers). We randomize each break-it team's view of the build-it teams' submissions,[1] but organize them by meta-data, such as programming language. When they think they have found a defect, breakers submit a test case that exposes the defect and an explanation of the issue. To encourage coverage, a break-it team may only submit up a fixed number of test cases per build-it submission. BIBIFI's infrastructure automatically judges whether a submitted test case truly reveals a defect. For example, for a correctness bug, it will run the test against a reference implementation

[1]This avoids spurious unfair effects, such as if break-it teams investigating code in the order in which we give it to them.

("the oracle") and the targeted submission, and only if the test passes on the former but fails on the latter will it be accepted.[2] More points are awarded to bugs that clearly reveal security problems, which may be demonstrated using alternative test formats. The auto-judgment approaches we developed for the two different contest problems are described in §2.2.

The final phase is the **fix-it phase**. Build-it teams are provided with the bug reports and test cases implicating their submission. They may fix flaws these test cases identify; if a single fix corrects more than one failing test case, the test cases are "morally the same," and thus points are only deducted for one of them. The organizers determine, based on information provided by the build-it teams and other assessment, whether a submitted fix is "atomic" in the sense that it corrects only one conceptual flaw; if not, the fix is rejected. This problem is a thorny one in general, we will re-visit this problem and possibly automated solutions to it in Chapter 6.

Once the final phase concludes, prizes are awarded to the best builders and best breakers as determined by the scoring system described next.

2.1.2 Competition scoring

BIBIFI's scoring system aims to encourage the contest's basic goals, which are that the winners of the build-it phase truly produced the highest quality software, and that the winners of the break-it phase performed the most thorough, creative

[2] Teams can also earn points by reporting bugs in the oracle, i.e., where its behavior contradicts the written specification; these reports are judged by the organizers.

analysis of others' code. The scoring rules create incentives for good behavior (and disincentives for bad behavior).

2.1.2.1 Build-it scores

To reflect real-world development concerns, the winning build-it team would ideally develop software that is correct, secure, and efficient. While security is of primary interest to our contest, developers in practice must balance these other aspects of quality against security [24, 25], leading to a set of trade-offs that cannot be ignored if we wish to understand real developer decision-making.

To encourage these, each build-it team's score is the sum of the *ship* score[3] and the *resilience* score. The ship score is composed of points gained for correctness tests and performance tests. Each mandatory correctness test is worth M points, for some constant M, while each optional correctness test is worth $M/2$ points. Each performance test has a numeric measure depending on the specific nature of the programming project—e.g., latency, space consumed, files left unprocessed—where lower measures are better. A test's worth is $M \cdot (worst - v)/(worst - best)$, where v is the measured result, *best* is the measure for the best-performing submission, and *worst* is the worst performing. As such, each performance test's value ranges from 0 to M.

The resilience score is determined after the break-it and fix-it phases, at which point the set of unique defects against a submission is known. For each *unique* bug found against a team's submission, we subtract P points from its resilience score;

[3]The name is meant to evoke a quality measure at the time software is shipped.

as such, it is non-positive, and the best possible resilience score is 0. For correctness bugs, we set P to $M/2$; for crashes that violate memory safety we set P to M, and for exploits and other security property failures we set P to $2M$.

2.1.2.2 Break-it scores

Our primary goal with break-it teams is to encourage them to find as many defects as possible in the submitted software, as this would give greater confidence in our assessment that one build-it team's software is of higher quality than another's. While we are particularly interested in obvious security defects, correctness defects are also important, as they can have non-obvious security implications.

After the break-it phase, a break-it team's score is the summed value of all defects they have found, using the above P valuations. After the fix-it phase, this score is reduced. In particular, each of the N break-it teams' scores that identified the same defect are adjusted to receive P/N points for that defect, splitting the P points among them.

Through a combination of requiring concrete test cases and scoring, BIBIFI encourages break-it teams to follow the spirit of the competition. First, by requiring them to provide test cases as evidence of a defect or vulnerability, we ensure they are providing useful bug reports. By providing $4\times$ more points for security-relevant bugs than for correctness bugs, we nudge break-it teams to look for these sorts of flaws, and to not just focus on correctness issues. (But a different ratio might work better; see §2.1.3.2.) Because break-it teams are limited to a fixed number

of test cases per submission, and because they could lose points during the fix-it phase for submitting test cases that could be considered "morally the same," break-it teams are encouraged to submit tests that demonstrate different bugs. Limiting per-submission test cases also encourages examining many submissions. Finally, because points for defects found by other teams are shared, break-it teams are encouraged to look for hard-to-find bugs, rather than just low-hanging fruit.

2.1.3 Discussion

The contest's design also aims to enable scalability by reducing work on contest organizers. In our experience, BIBIFI's design succeeds at what it sets out to achieve, but is not perfect. We close by discussing some limitations.

2.1.3.1 Minimizing manual effort

Once the contest begins, manual effort by the organizers is, by design, limited. All bug reports submitted during the break-it phase are automatically judged by the oracle; organizers only need to vet any bug reports against the oracle itself. Organizers may also need to judge accusations by breakers of code obfuscation by builders. Finally, organizers must judge whether submitted fixes address a single defect; this is the most time-consuming task. It is necessary because we cannot automatically determine whether multiple bug reports against one team map to the same software defect. Instead, we incentivize build-it teams to demonstrate overlap through fixes; organizers manually confirm that each fix addresses only a single

defect, not several.

Previewing some of the results presented later, we can confirm that the design works reasonably well. For example, as detailed in Table 2.4, 68 teams submitted 24,796 test cases in our Spring 2015 contest. The oracle auto-rejected 15,314 of these, and build-it teams addressed 2,252 of those remaining with 375 fixes, a 6× reduction. Most confirmations that a fix truly addressed a single bug took 1–2 minutes each. Only 30 of these fixes were rejected. No accusations of code obfuscation were made by break-it teams, and few bug reports were submitted against the oracle. All told, the Spring 2015 contest was successfully managed by one full-time person, with two others helping with judging.

2.1.3.2 Limitations

While we believe BIBIFI's structural and scoring incentives are properly designed, we should emphasize several limitations.

First, there is no guarantee that all implementation defects will be found. Break-it teams may lack the time or skill to find problems in all submissions, and not all submissions may receive equal scrutiny. Break-it teams may also act contrary to incentives and focus on easy-to-find and/or duplicated bugs, rather than the harder and/or unique ones. Finally, break-it teams may find defects that the BIBIFI infrastructure cannot automatically validate, meaning those defects will go unreported. However, with a large enough pool of break-it teams, and sufficiently general defect validations automation, we still anticipate good coverage both in

breadth and depth.

Second, builders may fail to fix bugs in a manner that is in their best interests. For example, in not wanting to have a fix rejected as addressing more than one conceptual defect, teams may use several specific fixes when a more general fix would have been allowed. Additionally, teams that are out of contention for prizes may simply not participate in the fix-it phase.[4] We observed this behavior for our contests, as described in §2.3.5. Both actions decrease a team's resilience score (and correspondingly increase breakers' scores). We can mitigate these issues with sufficiently strong incentives, e.g., by offering prizes to all participants commensurate with their final score, rather than offering prizes only to the top scorers.

Finally, there are several design points in the problem definition that may skew results. For example, too few correctness tests may leave too many correctness bugs to be found during break-it (distracting break-it teams' attention from security issues); too many correctness tests may leave too few (meaning teams are differentiated insufficiently by general bug-finding ability). Scoring prioritizes security problems 4 to 1 over correctness problems, but it is hard to say what ratio makes the most sense when trying to maximize real-world outcomes; both higher and lower ratios could be argued. Finally, performance tests may fail to expose important design trade-offs (e.g., space vs. time), affecting the ways that teams approach maximizing their ship scores. For the contests we report in this chapter, we are fairly comfortable with these design points. In particular, our earlier con-

[4]Hiding scores during the contest might help mitigate this, but would harm incentives during break-it to go after submissions with no bugs reported against them.

test [26] prioritized security bugs 2-to-1 and had fewer interesting performance tests, and outcomes were better when we increased the ratio.

2.1.3.3 Discouraging collusion

BIBIFI contestants may form teams however they wish, and may participate remotely. This encourages wider participation, but it also opens the possibility of collusion between teams, as there cannot be a judge overseeing their communication and coordination. There are three broad possibilities for collusion, each of which BIBIFI's scoring discourages.

First, two break-it teams could consider sharing bugs they find with one another. By scaling the points each finder of a particular bug obtains, we remove incentive for them to both submit the same bugs, as they would risk diluting how many points they both obtain.

The second class of collusion is between a build-it team and a break-it team, but neither have incentive to assist one another. The zero-sum nature of the scoring between breakers and builders places them at odds with one another; revealing a bug to a break-it team hurts the builder, and not reporting a bug hurts the breaker.

Finally, two build-it teams could collude, for instance by sharing code with one another. It might be in their interests to do this in the event that the competition offers prizes to two or more build-it teams, since collusion could obtain more than one prize-position. We use judging and automated tools (and feedback from break-it teams) to detect if two teams share the same code (and disqualify them), but

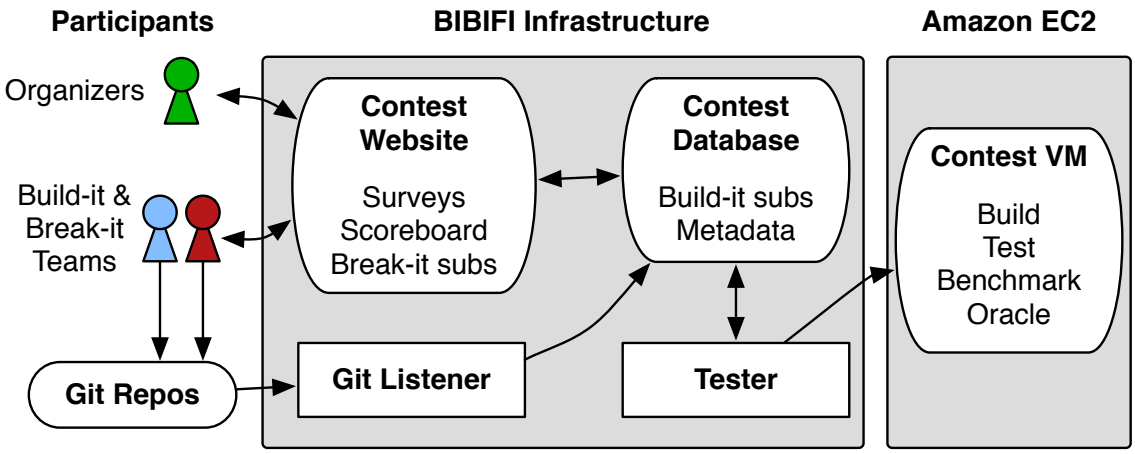


Figure 2.1: Overview of BIBIFI's implementation.

it is not clear how to detect whether two teams provided out-of-band feedback to one another prior to submitting code (e.g., by holding their own informal "break-it" and "fix-it" stages). We view this as a minor threat to validity; at the surface, such assistance appears unfair, but it is not clear that it is contrary to the goals of the contest, that is, to develop secure code.

2.1.4 Implementation

Figure 2.1 provides an overview of the BIBIFI implementation. It consists of a web frontend, providing the interface to both participants and organizers, and a backend for testing builds and breaks. Two key goals of the infrastructure are security—we do not want participants to succeed by hacking BIBIFI itself—and scalability.

Web frontend

Contestants sign up for the contest through our web application frontend, and fill out a survey when doing so, to gather demographic and other data potentially relevant to the contest outcome (e.g., programming experience and security training). During the contest, the web application tests build-it submissions and break-it bug reports, keeps the current scores updated, and provides a workbench for the judges for considering whether or not a submitted fix covers one bug or not.

To secure the web application against unscrupulous participants, we implemented it in ∼11,000 lines of Haskell using the Yesod [27] web framework backed by a PostgreSQL [28] database. Haskell's strong type system defends against use-after-free, buffer overrun, and other memory safety-based attacks. The use of Yesod adds further automatic protection against various attacks like CSRF, XSS, and SQL injection. As one further layer of defense, the web application incorporates the information flow control framework LMonad [29], which is derived from LIO [30], in order to protect against inadvertent information leaks and privilege escalations. LMonad dynamically guarantees that users can only access their own information.

Testing backend

The backend infrastructure is used during the build-it phase to test for correctness and performance, and during the break-it phase to assess potential vulnerabilities. It consists of ∼5,100 lines of Haskell code (and a little Python).

To automate testing, we require contestants to specify a URL to a Git [23]

repository hosted on either Github or Bitbucket, and shared with a designated `bibifi` username, read-only. The backend "listens" to each contestant repository for pushes, upon which it downloads and archives each commit. Testing is then handled by a scheduler that spins up an Amazon EC2 virtual machine which builds and tests each submission. We require that teams' code builds and runs, without any network access, in an Ubuntu Linux VM that we share in advance. Teams can request that we install additional packages not present on the VM. The use of VMs supports both scalability (Amazon EC2 instances are dynamically provisioned) and security (using fresh VM instances prevents a team from affecting the results of future tests, or of tests on other teams' submissions).

All qualifying build-it submissions may be downloaded by break-it teams at the start of the break-it phase. As break-it teams identify bugs, they prepare a (JSON-based) file specifying the buggy submission along with a sequence of commands with expected outputs that demonstrate the bug. Break-it teams commit and push this file (to their Git repository). The backend uses the file to set up a test of the implicated submission to see if it indeed is a bug.

2.2 Contest Problems

This section presents the two programming problems we developed for the contests held during 2015, including problem-specific notions of security defect and how breaks exploiting such defects are automatically judged.

2.2.1 Secure log (Spring 2015)

The secure log problem was motivated as support for an art gallery security system. Contestants write two programs. The first, `logappend`, appends events to the log; these events indicate when employees and visitors enter and exit gallery rooms. The second, `logread`, queries the log about past events. To qualify, submissions must implement two basic queries (involving the current state of the gallery and the movements of particular individuals), but they could implement two more for extra points (involving time spent in the museum, and intersections among different individuals' histories). An empty log is created by `logappend` with a given authentication token, and later calls to `logappend` and `logread` on the same log must use that token or the requests will be denied.

A canonical way of implementing the secure log is to treat the authentication token as a symmetric key for authenticated encryption, e.g., using a combination of AES and HMAC. There are several tempting shortcuts that we anticipated build-it teams would take (and that break-it teams would exploit). For instance, one may be tempted to encrypt and sign individual log records as opposed to the entire log, thereby making `logappend` faster. But this could permit integrity breaks that duplicate or reorder log records. Teams may also be tempted to implement their own encryption rather than use existing libraries, or to simply sidestep encryption altogether. §2.4 reports several cases we observed.

A submission's performance is measured in terms of time to perform a particular sequence of operations, and space consumed by the resulting log. Correctness

(and *crash*) bug reports comprise sequences of `logread` and/or `logappend` operations with expected outputs (vetted by the oracle). Security is defined by *privacy* and *integrity*: any attempt to learn something about the log's contents, or to change them, without the use of the `logread` and `logappend` and the proper token should be disallowed. How violations of these properties are specified and tested is described next.

Privacy breaks

When providing a build-it submission to the break-it teams, we also included a set of log files that were generated using a sequence of invocations of that submission's `logappend` program. We generated different logs for different build-it submissions, using a distinct command sequence and authentication token for each. All logs were distributed to break-it teams without the authentication token; some were distributed without revealing the sequence of commands (the "transcript") that generated them. For these, a break-it team could submit a test case involving a call to `logread` (with the authentication token omitted) that queries the file. The BIBIFI infrastructure would run the query on the specified file with the authentication token, and if the output matched that specified by the breaker, then a privacy violation is confirmed.

Integrity breaks

For about half of the generated log files we also provided the transcript of the `logappend` operations (*sans* auth token) used to generate the file. A team could submit a test case specifying the name of the log file, the contents of a corrupted version of that file, and a `logread` query over it (without the authentication token). For both the specified log file and the corrupted one, the BIBIFI infrastructure would run the query using the correct authentication token. An integrity violation is detected if the query command produces a non-error answer for the corrupted log that differs from the correct answer (which can be confirmed against the transcript using the oracle).

This approach to determining privacy and integrity breaks has the drawback that it does not reveal the *source* of the issue, only that there is (at least) one. As such, we cannot automatically tell two privacy or two integrity breaks apart. We sidestep this issue by counting only up to one integrity break and one privacy break against the score of each build-it submission, even if there are multiple defects that could be exploited to produce privacy/integrity violations.

2.2.2 Securing ATM interactions (Fall 2015)

The ATM problem asks builders to construct two communicating programs: `atm` acts as an ATM client, allowing customers to set up an account, and deposit and withdraw money, while `bank` is a server that processes client requests, tracking bank balances. `atm` and `bank` should only permit a customer with a correct *card*

file to learn or modify the balance of their account, and only in an appropriate way (e.g., they may not withdraw more money than they have). In addition, `atm` and `bank` should only communicate if they can authenticate each other. They can use an *auth file* for this purpose; it will be shared between the two via a trusted channel unavailable to the attacker.[5] Since the `atm` is communicating with `bank` over the network, a "man in the middle" (MITM) could observe and modify exchanged messages, or insert new messages. The MITM could try to compromise security despite not having access to auth or card files.

A canonical way of implementing the `atm` and `bank` programs would be to use public key-based authenticated and encrypted communications. The auth file could be used as the `bank`'s public key to ensure that key negotiation initiated by the `atm` is with the `bank` and not the MITM. When creating an account, the card file should be a suitably large random number, so that the MITM is unable to feasibly predict it. It is also necessary to protect against replay attacks by using nonces or similar mechanisms. As with the secure log, a wise approach would be use a library like OpenSSL to implement these features. Both good and bad implementations we observed in the competition are discussed further in §2.4.

Build-it submissions' performance is measured as the time to complete a series of benchmarks involving various `atm`/`bank` interactions.[6] Correctness (and *crash*)

[5] In a real deployment, this might be done by "burning" the auth file into the ATM's ROM prior to installing it.

[6] This transcript was always serial, so there was no direct motivation to support parallelism for higher throughput.

bug reports comprise sequences of `atm` commands where the targeted submission produces different outputs than the oracle (or crashes). Security defects are specified as follows.

Integrity breaks

Integrity violations are demonstrated using a custom MITM program that acts as a proxy: It listens on a specified IP address and TCP port,[7] and accepts a connection from the `atm` while connecting to the `bank`. The MITM program can thus observe and/or modify communications between `atm` and `bank`, as well as drop messages or initiate its own. We provided a Python-based proxy as a starter MITM: It sets up the connections and forwards communications between the two endpoints.

To demonstrate an integrity violation, the MITM sends requests to a *command server*. It can tell the server to run inputs on the `atm` and it can ask for the card file for any account whose creation it initiated. Eventually the MITM will declare the test complete. At this point, the same set of `atm` commands is run using the oracle's `atm` and `bank` *without the MITM*. This means that any messages that the MITM sends directly to the target submission's `atm` or `bank` will not be replayed/sent to the oracle. If the oracle and target both complete the command list without error, but they differ on the outputs of one or more commands, or on the balances of accounts at the bank whose card files were not revealed to the MITM during the test, then there is evidence of an integrity violation.

As an example (based on a real attack we observed), consider a submission

[7] All submissions were required to communicate via TCP.

that uses deterministic encryption without nonces in messages. The MITM could direct the command server to withdraw money from an account, and then replay the message it observes. When run on the vulnerable submission, this would debit the account twice. But when run on the oracle without the MITM, no message is replayed, leading to differing final account balances. A correct submission would reject the replayed message, which would invalidate the break.

Privacy breaks

Privacy violations are also demonstrated using a MITM. In this case, at the start of a test the command server will generate two random values, "amount" and "account name." If by the end of the test no errors have occurred and the attacker can prove it knows the actual value of either secret (by sending a command that specifies it), the break is considered successful. Before demonstrating knowledge of the secret, the MITM can send commands to the server with a *symbolic* "amount" and "account name"; the server fills in the actual secrets before forwarding these messages. The command server does not automatically create a secret account or an account with a secret balance; it is up to the breaker to do that (referencing the secrets symbolically when doing so).

As an example, suppose the target does not encrypt exchanged messages. Then a privacy attack might be for the MITM to direct the command server to create an account whose balance contains the secret amount. Then the MITM can observe an unencrypted message sent from `atm` to `bank`; this message will contain the actual

amount, filled in by the command server. The MITM can then send its guess to the command server showing that it knows the amount.

As with the log problem, we cannot tell whether an integrity or privacy test is exploiting the same underlying weakness in a submission, so we only accept one violation of each category against each submission.

Timeouts and denial of service

One difficulty with our use of a MITM is that we cannot reliably detect bugs in submissions that would result in infinite loops, missed messages, or corrupted messages. This is because such bugs can be simulated by the MITM by dropping or corrupting messages it receives. Since the builders are free to implement any protocol they like, our auto-testing infrastructure cannot tell if a protocol error or timeout is due to a bug in the target or due to misbehavior of the MITM. As such, we conservatively disallow reporting any such errors. Such flaws in builder implementations might exist but evidence of those bugs might not be realizable in our testing system.

2.3 Quantitative Analysis

This section analyzes data we have gathered from three contests we ran during 2015.[8] We consider participants' performance in each phase of the contest, including which factors contribute to high scores after the build-it round, resistance to

[8]We also ran a contest during Fall 2014 [26] but exclude it from consideration due to differences in how it was administered.

breaking by other teams, and strong performance as breakers.

We find that on average, teams that program in languages other than C and C++, and those whose members know more programming languages (perhaps a proxy for overall programming skill), are less likely to have security bugs identified in their code. However, when memory management bugs are not included, programming language is no longer a significant factor, suggesting that memory safety is the main discriminator between C/C++ and other languages. Success in breaking, and particularly in identifying security bugs in other teams' code, is correlated with having more team members, as well as with participating successfully in the build-it phase (and therefore having given thought to how to secure an implementation). Somewhat surprisingly, use of advanced techniques like fuzzing and static analysis did not appear to affect breaking success. Overall, integrity bugs were far more common than privacy bugs or crashes. The Fall 2015 contest, which used the ATM problem, was associated with more security bugs than the Spring 2015 secure log contest.

2.3.1 Data collection

For each team, we collected a variety of observed and self-reported data. When signing up for the contest, teams reported standard demographics and features such as coding experience and programming language familiarity. After the contest, each team member optionally completed a survey about their performance. In addition, we extracted information about lines of code written, number of commits, etc. from

teams' Git repositories.

Participant data was anonymized and stored in a manner approved by our institution's human-subjects review board. Participants consented to have data related to their activities collected, anonymized, stored, and analyzed. A few participants did not consent to research involvement, so their personal data was not used in the data analysis.

2.3.2 Analysis approach

To examine factors that correlated with success in building and breaking, we apply regression analysis. Each regression model attempts to explain some outcome variable using one or more measured factors. For most outcomes, such as participants' scores, we can use ordinary linear regression, which estimates how many points a given factor contributes to (or takes away from) a team's score. To analyze binary outcomes, such as whether or not a security bug was found, we apply logistic regression. This allows us to estimate how each factor impacts the likelihood of an outcome.

We consider many variables that could potentially impact teams' results. To avoid over-fitting, we initially select as potential factors those variables that we believe are of most interest, within acceptable limits for power and effect size. (Our choices are detailed below.) In addition, we test models with all possible combinations of these initial factors and select the model with the minimum Akaike Information Criterion (AIC) [31]. Only the final models are presented.

This was not a completely controlled experiment (e.g., we do not use random assignment), so our models demonstrate correlation rather than causation. Our observed effects may involve confounding variables, and many factors used as independent variables in our data are correlated with each other. This analysis also assumes that the factors we examine have linear effect on participants' scores (or on likelihood of binary outcomes); while this may not be the case in reality, it is a common simplification for considering the effects of many factors. We also note that some of the data we analyze is self-reported, and thus may not be entirely precise (e.g., some participants may have exaggerated about which programming languages they know); however, minor deviations, distributed across the population, act like noise and have little impact on the regression outcomes.

2.3.3 Contestants

We consider three contests offered at two times:

Spring 2015: We held one contest during May–June 2015 as the capstone to a Cybersecurity MOOC sequence.[9] Before competing in the capstone, participants passed courses on software security, cryptography, usable security, and hardware security. The contest problem was the secure log problem (§2.2.1).

Fall 2015: During Oct.–Nov. 2015, we offered two contests simultaneously, one as a MOOC capstone, and the other open to U.S.-based graduate and undergraduate students. We merged the contests after the build-it phase, due to low participation in the open contest; from here on we refer to these two as a single

[9] `https://www.coursera.org/specializations/cyber-security`

Contest	USA	Brazil	Russia	India	Other
Spring 2015	30	12	12	7	120
Fall 2015	64	20	12	14	110

Table 2.1: Contestants, by self-reported country.

contest. The contest problem was the ATM problem (§2.2.2).

The U.S. had more contestants than any other country. There was representation from developed countries with a reputation both for high technology and hacking acumen. Details of the most popular countries of origin can be found in Table 2.1, and additional information about contestant demographics is presented in Table 2.2.

2.3.4 Ship scores

We first consider factors correlating with a team's *ship* score, which assesses their submission's quality before it is attacked by the other teams (§2.1.1). This data set contains all 101 teams from the Spring 2015 and Fall 2015 contests that qualified after the build-it phase. Both contests have nearly the same number of correctness and performance tests, but different numbers of participants. We set the constant multiplier M to be 50 for both contests, which effectively normalizes the scores.

Contest	**Spring '15†**	**Fall '15†**	**Fall '15**
# Contestants	156	122	23
% Male	91%	89%	100%
% Female	5%	9%	0%
Age	34.8/20/61	33.5/19/69	25.1/17/31
% with CS degrees	35%	38%	23%
Years programming	9.6/0/30	9.9/0/37	6.6/2/13
# Build-it teams	61	34	6
Build-it team size	2.2/1/5	3.1/1/5	3.1/1/6
# Break-it teams	65	39	4
(that also built)	(58)	(32)	(3)
Break-it team size	2.4/1/5	3.0/1/5	3.5/1/6
# PLs known per team	6.8/1/22	10.0/2/20	4.2/1/8

Table 2.2: Demographics of contestants from qualifying teams. † indicates MOOC participants. Some participants declined to specify gender. Slashed values represent mean/min/max.

Model setup

To ensure enough power to find meaningful relationships, we decided to aim for a prospective effect size roughly equivalent to Cohen's *medium* effect heuristic, $f^2 = 0.15$ [32]. An effect this size suggests the model can explain up to 13% of the variance in the outcome variable. With an assumed power of 0.75 and population

$N = 101$, we limited ourselves to nine degrees of freedom, which yields a prospective $f^2 = 0.154$. (Observed effect size for the final model is reported with the regression results below.) Within this limit, we selected the following potential factors:

Contest: Whether the team's submission was for the Spring 2015 contest or the Fall 2015 contest.

 # Team members: A team's size.

Knowledge of C: The fraction of team members who listed C or C++ as a programming language they know. We included this variable as a proxy for comfort with low-level implementation details, a skill often viewed as a prerequisite for successful secure building or breaking.

 # Languages known: How many unique programming languages team members collectively claim to know (see the last row of Table 2.2). For example, on a two-member team where member A claims to know C++, Java, and Perl and member B claims to know Java, Perl, Python, and Ruby, the language count would be 5.

Coding experience: The average years of programming experience reported by a team's members.

Language category: We manually identified each team's submission as having one "primary" language. These languages were then assigned to one of three categories: C/C++, statically-typed (e.g., Java, Go, but not C/C++) and dynamically-typed (e.g., Perl, Python). C/C++ is the baseline category for the regression. Precise category allocations, and total submissions for each language, segregated by contest, are given in Figure 2.2.

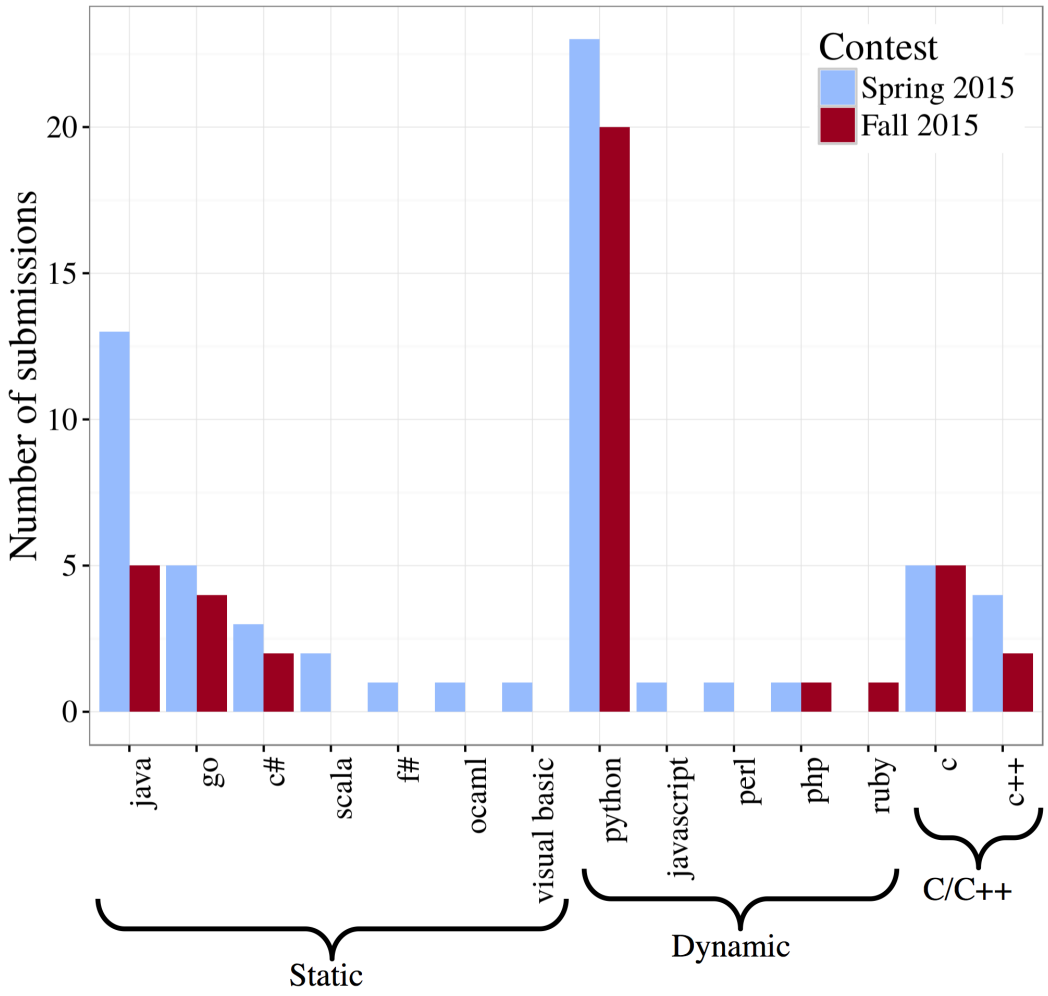


Figure 2.2: The number of build-it submissions in each contest, organized by primary programming language used. The brackets group the languages into categories.

Lines of code: The SLOC[10] count of lines of code for the team's final submission at qualification time.

MOOC: Whether the team was participating in the MOOC capstone project.

[10] `http://www.dwheeler.com/sloccount`

Results

Our regression results (Table 2.3) indicate that ship score is strongly correlated with language choice. Teams that programmed in C or C++ performed on average 121 and 92 points better than those who programmed in dynamically typed or statically typed languages, respectively. Figure 2.3 illustrates that while teams in many language categories performed well in this phase, only teams that did not use C or C++ scored poorly.

The high scores for C/C++ teams could be due to better scores on performance tests and/or due to implementing optional features. We confirmed the main cause is the former. Every C/C++ team for Spring 2015 implemented all optional features, while six teams in the other categories implemented only 6 of 10, and one team implemented none; the Fall 2015 contest offered no optional features. We artificially increased the scores of those seven teams as if they had implemented all optional features and reran the regression model. The resulting model had very similar coefficients.

Our results also suggest that teams that were associated with the MOOC capstone performed 119 points better than non-MOOC teams. MOOC participants typically had more programming experience and CS training.

Finally, we found that each additional line of code in a team's submission was associated with a drop of 0.03 points in ship score. Based on our qualitative observations (see §2.4), we hypothesize this may relate to more reuse of code from libraries, which frequently are not counted in a team's LOC (most libraries were

Factor	Coef.	SE	p-value
Fall 2015	-21.462	28.359	0.451
Lines of code	-0.031	0.014	0.036*
Dynamically typed	-120.577	40.953	0.004*
Statically typed	-91.782	39.388	0.022*
MOOC	119.359	58.375	0.044*

Table 2.3: Final linear regression model of teams' ship scores, indicating how many points each selected factor adds to the total score. Overall effect size $f^2 = 0.163$.

installed directly on the VM, not in the submission itself). We also found that, as further noted below, submissions that used libraries with more sophisticated, lower-level interfaces tended to have more code and more mistakes; their use required more code in the application, lending themselves to missing steps or incorrect use, and thus security and correctness bugs. As shown in Figure 2.3, LOC is also (as expected) associated with the category of language being used. While LOC varied widely within each language type, dynamic submissions were generally shortest, followed by static submissions, and then those written in C/C++ (which has the largest minimum size).

2.3.5 Code quality measures

Now we turn to measures of a build-it submission's quality—in terms of its correctness and security—based on how it held up under scrutiny by break-it teams.

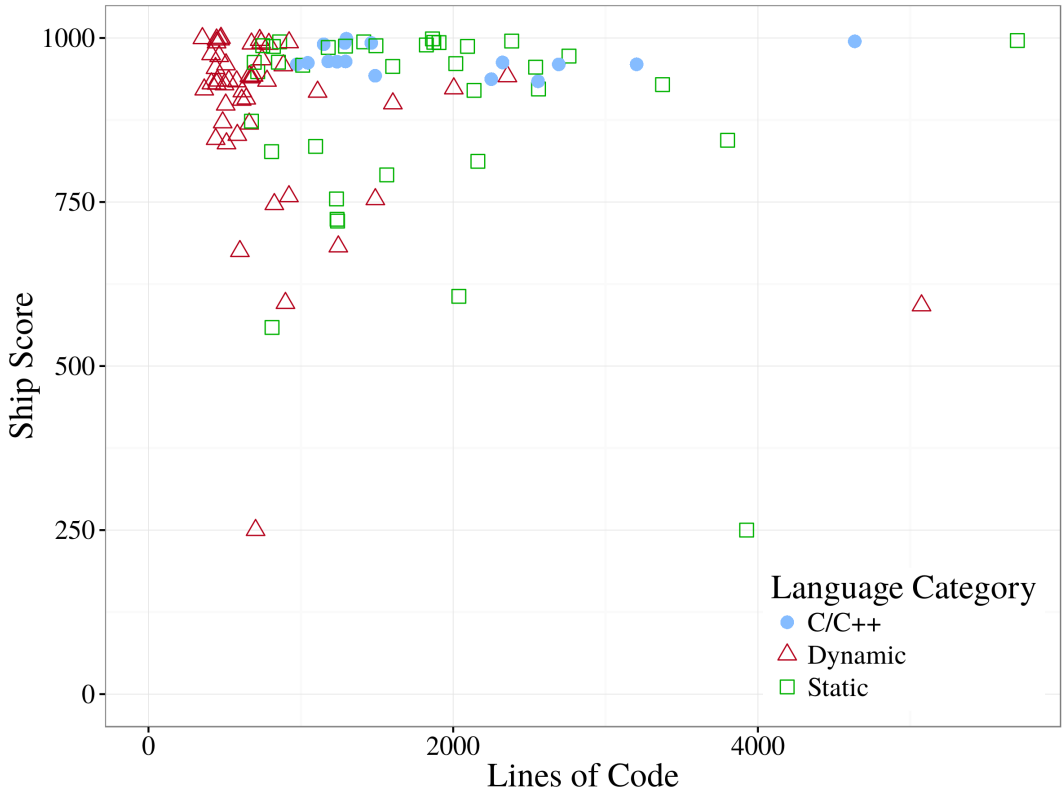


Figure 2.3: Each team's ship score, compared to the lines of code in its implementation and organized by language category. Fewer LOC and using C/C++ correlate with a higher ship score.

Resilience

The total build-it score is the sum of ship score, just discussed, and *resilience*. Resilience is a non-positive score that derives from break-it teams' test cases that prove the presence of defects. Builders may increase this score during the fix-it phase, as fixes prevent double-counting test cases that identify the same defect (see §2.1.1).

Unfortunately, upon studying the data we found that a large percentage of build-it teams opted not to fix any bugs reported against their code, forgoing the

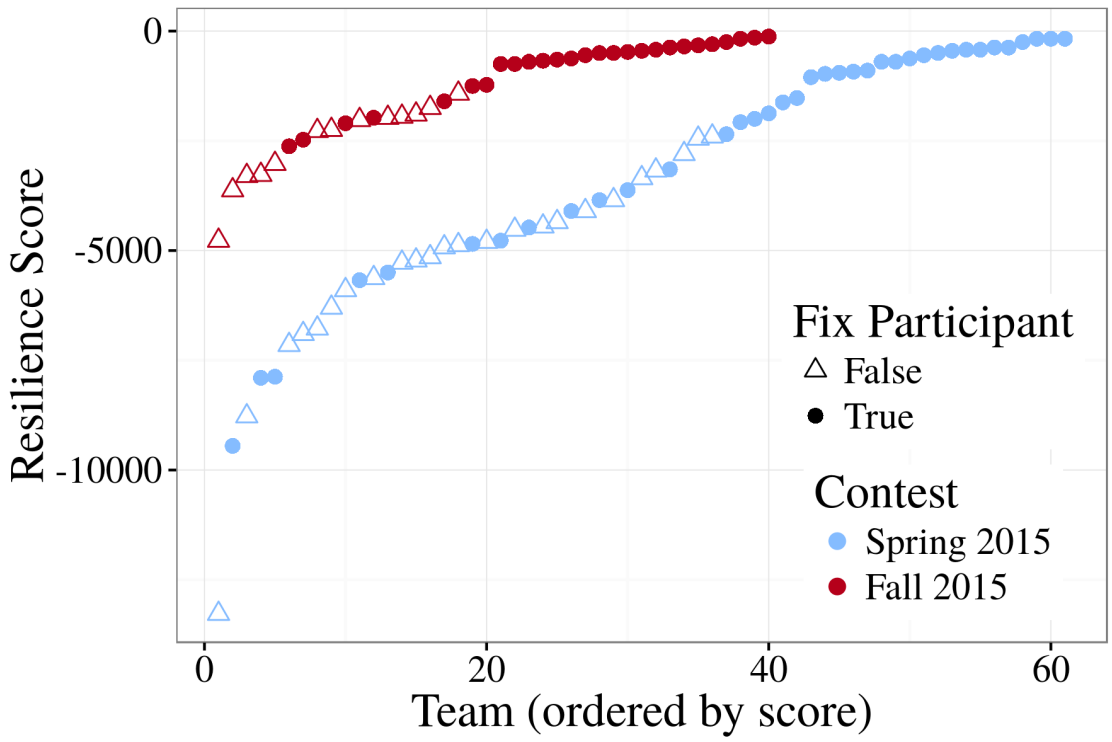


Figure 2.4: Final resilience scores, ordered by team, and plotted for each contest problem. Build-it teams who did not bother to fix bugs generally had lower scores.

scoring advantage of doing so. We can see this in Figure 2.4, which graphs the resilience scores (Y-axis) of all teams, ordered by score, for the two contests. The circles in the plot indicate teams that fixed at least one bug, whereas the triangles indicate teams that fixed no bugs. We can see that, overwhelmingly, the teams with the lower resilience scores did not fix any bugs. We further confirmed that fixing, or not, was a dominant factor by running a regression on resilience score that included fix-it phase participation as a factor (not shown). Overall, teams fixed an average of 34.5% of bugs in Spring 2015 and 45.3% of bugs in Fall 2015. Counting only "active" fixers who fixed at least one bug, these averages were 56.9% and 72.5%

	Spring 2015	Fall 2015
Bug reports submitted	24,796	3,701
Bug reports accepted	9,482	2,482
Fixes submitted	375	166
Bugs addressed by fixes	2,252	966

Table 2.4: Break-it teams in each contest submitted bug reports, which were judged by the automated oracle. Build-it teams then submitted fixes, each of which could potentially address multiple bug reports.

respectively.

Table 2.4 digs a little further into the situation. It shows that of the bug reports deemed acceptable by the oracle (the second row), submitted fixes (row 3) addressed only 23% of those from Spring 2015 and 38% of those from Fall 2015 (row 4 divided by row 2).

This situation is disappointing, as we cannot treat resilience score as a good measure of code quality (when added to ship score). Our hypothesis is that participants were not sufficiently incentivized to fix bugs, for two reasons. First, teams that are sufficiently far from the lead may have chosen to fix no bugs because winning was unlikely. Second, for MOOC students, once a minimum score is achieved they were assured to pass; it may be that fixing (many) bugs was unnecessary for attaining this minimum score.

Presence of security bugs

While resilience score is not sufficiently meaningful, a useful alternative is the likelihood that a build-it submission contains a security-relevant bug; by this we mean any submission against which at least one crash, privacy, or integrity defect is demonstrated. In this model we used logistic regression over the same set of factors as the ship model.

Table 2.5 lists the results of this logistic regression; the coefficients represent the change in log likelihood associated with each factor. Negative coefficients indicate lower likelihood of finding a security bug. For categorical factors, the exponential of the coefficient (Exp(coef)) indicates roughly how strongly that factor being true affects the likelihood relative to the baseline category.[11] For numeric factors, the exponential indicates how the likelihood changes with each unit change in that factor.

Fall 2015 implementations were 296× as likely as Spring 2015 implementations to have a discovered security bug.[12] We hypothesize this is due to the increased security design space in the ATM problem as compared to the gallery problem. Although it is easier to demonstrate a security error in the gallery problem, the ATM problem allows for a much more powerful adversary (the MITM) that can

[11] In cases (such as the Fall 2015 contest) where the rate of security bug discovery is close to 100%, the change in log likelihood starts to approach infinity, somewhat distorting this coefficient upwards.

[12] This coefficient is somewhat exaggerated (see prior footnote), but the difference between contests is large and significant.

Factor	Coef.	Exp(coef)	SE	p-value
Fall 2015	5.692	296.395	1.374	<0.001*
# Languages known	-0.184	0.832	0.086	0.033*
Lines of code	0.001	1.001	0.0003	0.030*
Dynamically typed	-0.751	0.472	0.879	0.393
Statically typed	-2.138	0.118	0.889	0.016*
MOOC	2.872	17.674	1.672	0.086

Table 2.5: Final logistic regression model, measuring log likelihood of a security bug being found in a team's submission.

interact with the implementation; breakers often took advantage of this capability, as discussed in §2.4.

The model also shows that C/C++ implementations were more likely to contain an identified security bug than either static or dynamic implementations. For static languages, this effect is significant and indicates that a C/C++ program was about 8.5× (that is, 1/0.118) as likely to contain an identified bug. This effect is clear in Figure 2.5, which plots the fraction of implementations that contain a security bug, broken down by language type and contest problem. Of the 16 C/C++ submissions (see Figure 2.2), 12 of them had a security bug: 5/9 for Spring 2015 and 7/7 for Fall 2015. All 5 of the buggy implementations from Spring 2015 had a crash defect, and this was the only security-related problem for three of them; none of the Fall 2015 implementations had crash defects.

If we reclassify crash defects as not security relevant and rerun the model we

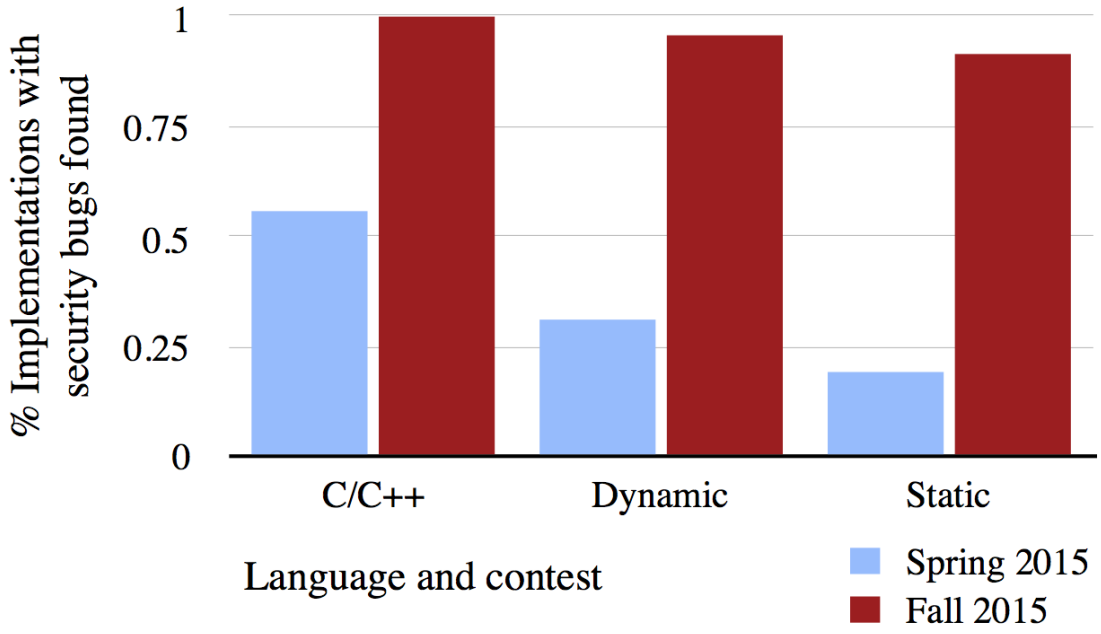


Figure 2.5: The fraction of teams in whose submission a security bug was found, for each contest and language category.

find that the impact due to language category is no longer statistically significant. This may indicate that lack of memory safety is the main disadvantage to using C/C++ from a security perspective, and thus a memory-safe C/C++ could be of significant value. Figure 2.6 shows how many security bugs of each type (memory safety, integrity, privacy) were found in each language category, across both contests. This figure reports bugs before unification during the fix-it phase, and is of course affected by differences among teams' skills and language choices in the two contests, but it provides a high-level perspective.

Our model shows that teams that knew more unique languages (even if they did not use those languages in their submission) performed slightly better, about

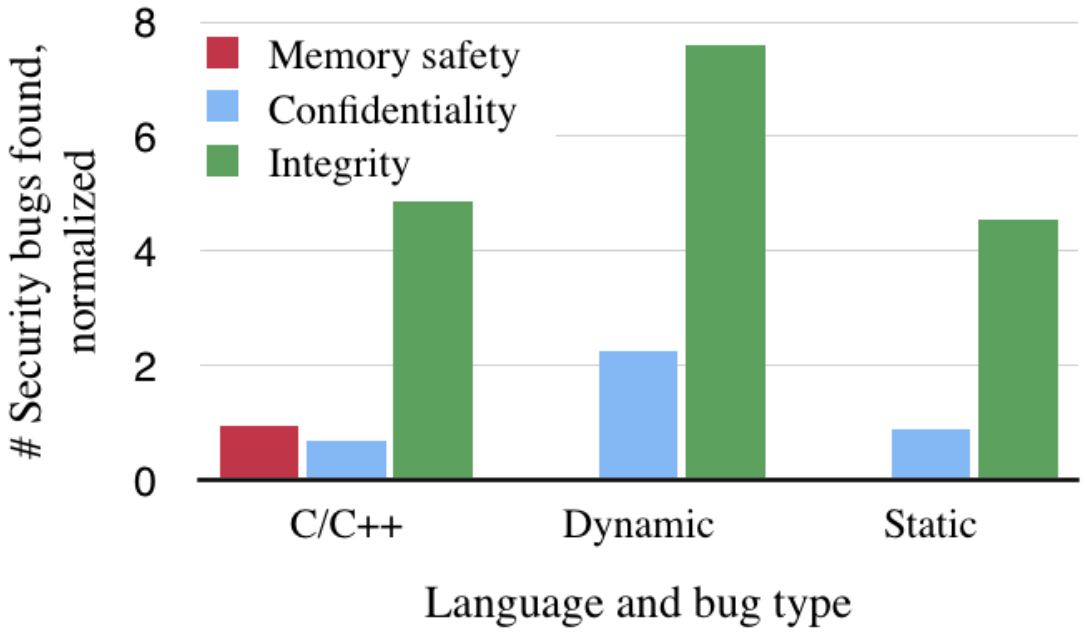


Figure 2.6: How many of each type of security bug were found, across both contests, for each language category. Counts are normalized by the number of qualified Build-it submissions in each language category.

1.2× for each language known. Additional LOC in an implementation were also associated with a very small increase in the presence of an identified security bug.

Finally, the model shows two factors that played a role in the outcome, but not in a statistically significant way: using a dynamically typed language, and participating in the MOOC. We see the effect of the former in Figure 2.5. For the latter, the effect size is quite large; it is possible that the MOOC security training played a role.

2.3.6 Breaking success

Now we turn our attention to break-it team performance, i.e., how effective teams were at finding defects in others' submissions. First, we consider how and why teams performed as indicated by their (normalized) break-it score *prior to the fix-it phase*. We do this to measure a team's raw output, ignoring whether other teams found the same bug (which we cannot assess with confidence due to the lack of fix-it phase participation per §2.3.5). This data set includes 108 teams that participated in the break-it phase in Spring and Fall 2015. We also model which factors contributed to **security bug count**, or how many total security bugs a break-it team found. Doing this disregards a break-it team's effort at finding correctness bugs.

We model both break-it score and security bug count using several of the same potential factors as discussed previously, but applied to the breaking team rather than the building team. In particular, we include which contest they participated in, whether they were **MOOC** participants, the number of break-it **Team members**, average team-member **Coding experience**, average team-member **Knowledge of C**, and unique **Languages known** by the break-it team members. We also add two new potential factors:

Build participant: Whether the breaking team also qualified during the build-it phase.

Advanced techniques: Whether the breaking team reported using software analysis or fuzzing to aid in bug finding. Teams that only used manual inspection and testing are categorized as false. 26 break-it teams (24%) reported using

Factor	Coef.	SE	p-value
Fall 2015	-2406.89	685.73	<0.001*
# Team members	430.01	193.22	0.028*
Knowledge of C	-1591.02	1006.13	0.117
Coding experience	99.24	51.29	0.056
Build participant	1534.13	995.87	0.127

Table 2.6: Final linear regression model of teams' break-it scores, indicating how many points each selected factor adds to the total score. Overall effect size $f^2 = 0.039$.

advanced techniques.

For these two initial models, our potential factors provide eight degrees of freedom; again assuming power of 0.75, this yields a prospective effect size $f^2 = 0.136$, indicating we could again expect to find effects of roughly medium size by Cohen's heuristic [32].

Break score

The model considering break-it score is given in Table 2.6. It shows that teams with more members performed better, with an average of 430 additional points per team member. Auditing code for errors is an easily parallelized task, so teams with more members could divide their effort and achieve better coverage. Recall that having more team members did not help build-it teams (see Tables 2.3 and 2.5); this makes sense as development requires more coordination, especially during the

early stages.

The model also indicates that Spring 2015 teams performed significantly better than Fall 2015 teams. Figure 2.9 illustrates that correctness bugs, despite being worth fewer points than security bugs, dominate overall break-it scores for Spring 2015. In Fall 2015 the scores are more evenly distributed between correctness and security bugs. This outcome is not surprising to us, as it was somewhat by design. The Spring 2015 problem defines a rich command-line interface with many opportunities for subtle errors that break-it teams can target. It also allowed a break-it team to submit up to 10 correctness bugs per build-it submission. To nudge teams toward finding more security-relevant bugs, we reduced the submission limit from 10 to 5, and designed the Fall 2015 interface to be far simpler.

Interestingly, making use of advanced analysis techniques did not factor into the final model; i.e., such techniques did not provide a meaningful advantage. This makes sense when we consider that such techniques tend to find generic errors such as crashes, bounds violations, or null pointer dereferences. Security violations for our problems are more semantic, e.g., involving incorrect design or use of cryptography. Many correctness bugs were non-generic too, e.g., involving incorrect argument processing or mishandling of inconsistent or incorrect inputs.

Being a build participant and having more coding experience is identified as a postive factor in the break-it score, according to the model, but neither is statistically significant (though they are close to the threshold). Interestingly, knowledge of C is identified as a strongly negative factor in break-it score (though again, not

Factor	**Coef.**	**SE**	***p*-value**
Fall 2015	3.847	1.486	0.011*
# Team members	1.218	0.417	0.004*
Build participant	5.430	2.116	0.012*

Table 2.7: Final linear regression modeling the count of security bugs found by each team. Coefficients indicate how many security bugs each factor adds to the count. Overall effect size $f^2 = 0.035$.

statistically significant). Looking closely at the results, we find that *lack* of C knowledge is extremely *uncommon*, but that the handful of teams in this category did unusually well. However, there are to few of them for the result to be significant.

Security bugs found

We next consider breaking success as measured by the count of security bugs a breaking team found. This model (Table 2.7) again shows that team size is important, with an average of one extra security bug found for each additional team member. Being a qualified builder also significantly helps one's score; this makes intuitive sense, as one would expect to gain a great deal of insight into how a system could fail after successfully building a similar system. Figure 2.10 shows the distribution of the number of security bugs found, per contest, for break-it teams that were and were not qualified build-it teams. Note that all but three of the 108 break-it teams made some attempt, as defined by having made a commit, to partic-

ipate during the build-it phase—most of these (93) qualified, but 12 did not. If the reason was that these teams were less capable programmers, that may imply that programming ability generally has some correlation with break-it success.

On average, four more security bugs were found by a Fall 2015 team than a Spring 2015 team. This contrasts with the finding that Spring 2015 teams had higher overall break-it scores, but corresponds to the finding that more Fall 2015 submissions had security bugs found against them. As discussed above, this is because correctness bugs dominated in Spring 2015 but were not as dominant in Fall 2015. Once again, the reasons may have been the smaller budget on per-submission correctness bugs in Fall 2015, and the greater potential attack surface in the ATM problem.

2.4 Qualitative Analysis

As part of the data gathered, we also have the entire program produced during the build-it phase as well as the programs patched during the fix-it phase. We can then perform a qualitative analysis of the programs which is guided by knowing the security outcome of a given program. Did lots of break-it teams find bugs in the program, or did they not? What are traits or characteristics of well-designed programs?

2.4.1 Success Stories

The success stories bear out some old chestnuts of wisdom in the security community: submissions that fared well through the break-it phase made heavy use of existing high-level cryptographic libraries with few "knobs" that allow for incorrect usage [33].

One implementation of the ATM problem, written in Python, made use of the SSL PKI infrastructure. The implementation used generated SSL private keys to establish a root of trust that authenticated the `atm` program to the `bank` program. Both the `atm` and `bank` required that the connection be signed with the certificate generated at run-time. Both the `bank` and the `atm` implemented their communication protocol as plain text then wrapped in HTTPS. This put the contestant on good footing; to find bugs in this system, other contestants would need to break the security of OpenSSL.

Another implementation, also for the ATM problem, written in Java, used the NaCl library. This library intentionally provides a very high level API to "box" and "unbox" secret values, freeing the user from dangerous choices. As above, to break this system, other contestants would need to first break the security of NaCl.

An implementation of the log reader problem, also written in Java, achieved success using a high level API. They used the BouncyCastle library to construct a valid encrypt-then-MAC scheme over the entire log file.

2.4.2 Failure Stories

The failure modes for build-it submissions are distributed along a spectrum ranging from "failed to provide any security at all" to "vulnerable to extremely subtle timing attacks." This is interesting because it is a similar dynamic observed in the software marketplace today.

Many implementations of the log problem lacked encryption or authentication. Exploiting these design flaws was trivial for break-it teams. Sometimes log data was written as plain text, other times log data was serialized using the Java object serialization protocol.

One break-it team discovered a privacy flaw which they could exploit with at most fifty probes. The target submission truncated the "authentication token," so that it was vulnerable to a brute force attack.

The ATM problem allows for interactive attacks (not possible for the log), and the attacks became cleverer as implementations used cryptographic constructions incorrectly. One implementation used cryptography, but implemented RC4 from scratch and did not add any randomness to the key or the cipher stream. An attacker observed that the ciphertext of messages was distinguishable and largely unchanged from transaction to transaction, and was able to flip bits in a message to change the withdrawn amount.

Another implementation used encryption with authentication, but did not use randomness; as such error messages were always distinguishable success messages. An attack was constructed against this implementation where the attack leaked

the bank balance by observing different withdrawal attempts, distinguishing the successful from failed transactions, and performing a binary search to identify the bank balance given a series of withdraw attempts.

Some failures were common across ATM problem implementations. Many implementations kept the key fixed across the lifetime of the `bank` and `atm` programs and did not use a nonce in the messages. This allowed attackers to replay messages freely between the `bank` and the `atm`, violating integrity via unauthorized withdrawals. Several implementations used encryption, but without authentication. These implementations used a library such as OpenSSL, the Java cryptographic framework, or the Python pycrypto library to have access to a symmetric cipher such as AES, but either did not use these libraries at a level where authentication was provided in addition to encryption, or they did not enable authentication.

Some failures were common across log implementations as well: if an implementation used encryption, it might not use authentication. If it used authentication, it would authenticate records stored in the file individually and not globally. The implementations would also relate the ordering of entries in the file to the ordering of events in time, allowing for an integrity attack that changes history by re-ordering entries in the file.

As a corpus for research, this data set is of interest for future mining. What common design patterns were used and how did they impact the outcome? Are there any metrics we can extract from the code itself that can predict break-it scores? We defer this analysis to future work.

2.5 Related work

BIBIFI bears similarity to existing programming and security contests but is unique in its focus on building secure systems. BIBIFI also is related to studies of code and secure development, but differs in its open-ended contest format.

Contests

Cybersecurity contests typically focus on vulnerability discovery and exploitation, and sometimes involve a system administration component for defense. One popular style of contest is dubbed *capture the flag* (CTF) and is exemplified by a contest held at DEFCON [34]. Here, teams run an identical system that has buggy components. The goal is to find and exploit the bugs in other competitors' systems while mitigating the bugs in your own. Compromising a system enables a team to acquire the system's key and thus "capture the flag." In addition to DEFCON CTF, there are other CTFs such as iCTF [35, 36] and PicoCTF [37]. The use of this style of contest in an educational setting has been explored in prior work [38–40]. The Collegiate Cyber Defense Challenge [14, 41, 42] and the Maryland Cyber Challenge & Competition [13] have contestants defend a system, so their responsibilities end at the identification and mitigation of vulnerabilities. These contests focus on bugs in systems as a key factor of play, but neglect software development.

Programming contests challenge students to build clever, efficient software, usually with constraints and while under (extreme) time pressure. The ACM programming contest [19] asks teams to write several programs in C/C++ or Java

during a 5-hour time period. Google Code Jam [43] sets tasks that must be solved in minutes, which are then graded according to development speed (and implicitly, correctness). Topcoder [18] runs several contests; the Algorithm competitions are small projects that take a few hours to a week, whereas Design and Development competitions are for larger projects that must meet a broader specification. Code is judged for correctness (by passing tests), performance, and sometimes subjectively in terms of code quality or practicality of design. All of these resemble the build-it phase of BIBIFI but typically consider smaller tasks; they do not consider the security of the produced code.

Studies of secure software development

There have been a few studies of different methods and techniques for ensuring security. Work by Finifter and Wagner [44] and Prechelt [45] relates to both our build-it and break-it phases: they asked different teams to develop the same web application using different frameworks, and then subjected each implementation to automated (black box) testing and manual review. They found that both forms of review were effective in different ways, and that framework support for mitigating certain vulnerabilities improved overall security. Other studies focused on the effectiveness of vulnerability discovery techniques, e.g., as might be used during our break-it phase. Edmundson et al. [46] considered manual code review; Scandariato et al. [47] compared different vulnerability detection tools; other studies looked at software properties that might co-occur with security problems [48–50]. BIBIFI dif-

fers from all of these in its open-ended, contest format: Participants can employ any technique they like, and with a large enough population and/or measurable impact, the effectiveness of a given technique will be evident in final outcomes.

2.6 Conclusions

This chapter has presented Build-it, Break-it, Fix-it (BIBIFI), a new security contest that brings together features from typical security contests, which focus on vulnerability detection and mitigation but not secure development, and programming contests, which focus on development but not security. During the first phase of the contest, teams construct software they intend to be correct, efficient, and secure. During the second phase, break-it teams report security vulnerabilities and other defects in submitted software. In the final, fix-it, phase, builders fix reported bugs and thereby identify redundant defect reports. Final scores, following an incentives-conscious scoring system, reward the best builders and breakers.

During 2015, we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the best performing build-it submissions used C/C++, but submissions coded in a statically-typed language were less likely to have a security flaw; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

There are many interesting areas of future work that BIBIFI opens up. The

BIBIFI design lends itself well to conducting more focused studies; our competitions allow participants to use any languages and tools they desire, but one could narrow their options for closer evaluation. Although we have reduced manual judging considerably, an interesting technical problem that often arises is determining whether two bugs are morally equivalent; an automated method for determining this could be broadly applicable. Finally, one limitation of our study is that we do not evaluate whether break-it teams find all of the bugs there are to find; one improvement would be to apply a set of fuzzers and static analyzers, or to recruit professional teams to effectively participate in the break-it phase as a sort of baseline against which to compare the break-it teams' performance.

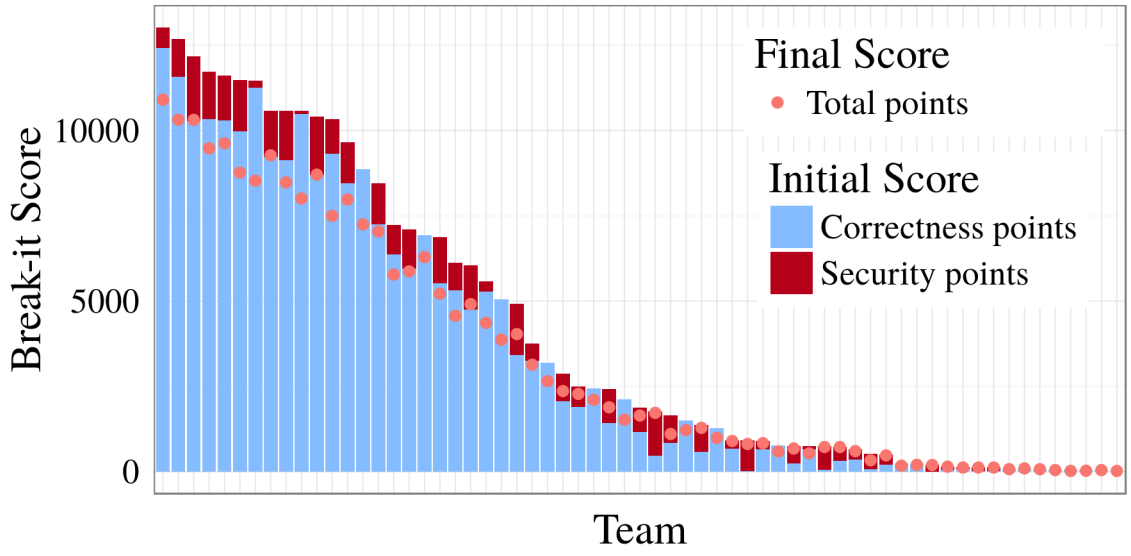


Figure 2.7: Spring 2015

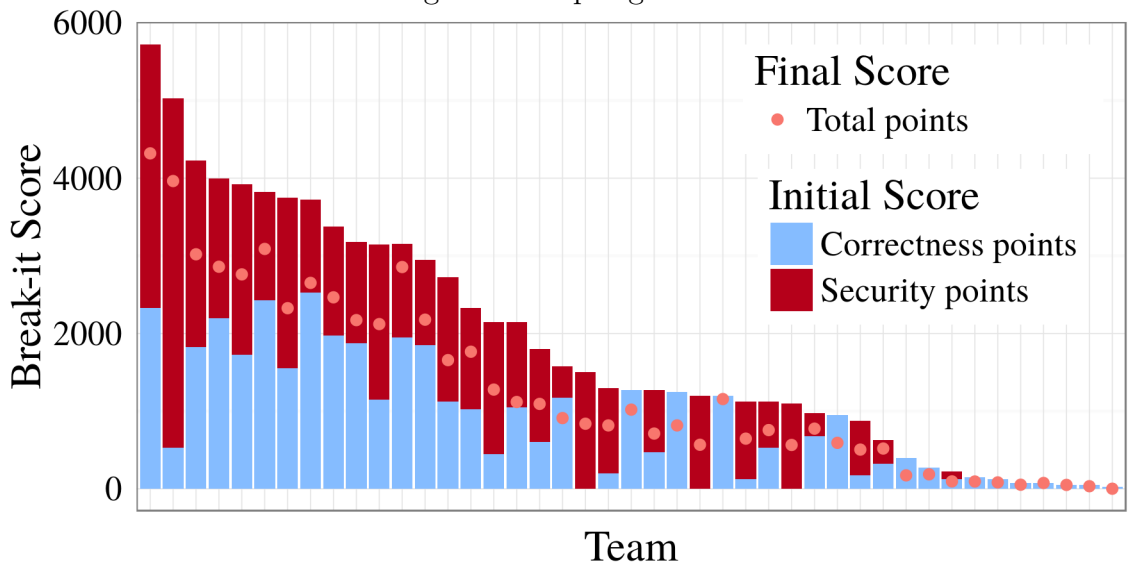


Figure 2.8: Fall 2015

Figure 2.9: Scores of break-it teams prior to the fix-it phase, broken down by points from security and correctness bugs. The final score of the break-it team (after fix-it phase) is noted as a dot. Note the different ranges in the y-axes; in general, the Spring 2015 contest (secure log problem) had higher scores for breaking.

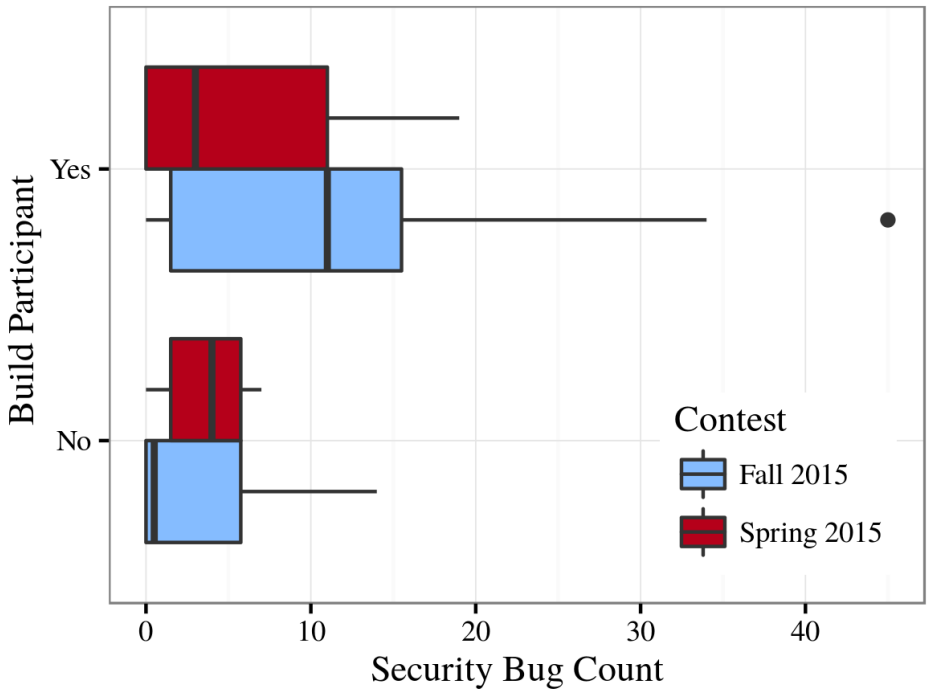


Figure 2.10: Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within $\pm 1.5\times$ the interquartile range. Dots indicate further outliers.

Chapter 3: Checked C

Vulnerabilities that compromise *memory safety* are at the heart of many devastating attacks. Memory safety has two aspects. *Temporal safety* is ensured when memory is never used after it is freed. *Spatial safety* is ensured when any pointer dereference is always within the memory allocated to that pointer. *Buffer overruns*—a spatial safety violation—still constitute a frequent and pernicious source of vulnerability, despite their long history. During the period 2012–2016, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [51], with the highest numbers occurring in 2016. During that time, buffer overruns were the leading single cause of CVEs. Additionally, many problems discovered in Chapter 2 were related to violations of spatial memory safety, when C was used by contestants.

Spatial safety violations commonly arise when programming low-level, performance critical code in C and C++. While a type-safe language disallows such violations [52], using one is impractical when low-level control is needed. Building on research from projects such as Cyclone [53] and Deputy [54], modern languages like Rust [55] and Go [56] provide a promising balance of safety and performance, but to use them requires programmer retraining and extensive rewrites of legacy

code.

As discussed in depth in Section 3.5, several efforts have attempted to make C programs safe. Static analysis tools [57–59] aim to find vulnerabilities pre-deployment, but may miss bugs, have trouble scaling, or emit too many alarms. Security mitigations, such as W⊕X [60] and CFI [61], can mute the impact of vulnerabilities by making them harder to exploit, but provide no guarantee; e.g., data leaks and mimicry attacks may still be possible. Several efforts have aimed to provide spatial safety by adding run-time checks; these include CCured [62], Softbound [63], and ASAN [64]. The added checks can add substantial overhead and can complicate interoperability with legacy code if pointer representations are changed. Lower overhead can be achieved by reducing safety, e.g., by checking only writes, or ignoring overruns within a memory region (e.g., from one stack variable to another, or one struct field to another). In the end, no existing approach is completely satisfying.

This chapter presents a new effort towards achieving a spatially-safe C that we call *Checked C*. Checked C borrows many ideas from prior safe-C efforts but ultimately differs in that its design focuses on interoperability, developer usability, and enabling highly performant code. Checked C and legacy C can coexist, so developers are able to port legacy code incrementally. This approach does allow for defects and vulnerabilities in non-converted regions of the program. However, taking inspiration from work on *gradual typing* [65–67], Checked C gives developers a way to distinguish "checked" from "unchecked" regions. The former can be held blameless as the source of any safety violation, so software assurance attention can be focused on the latter.

Technically speaking, Checked C's design has three key features. First, all pointers in Checked C are represented as in normal C—no changes to pointer layout are imposed. This eases interoperability.

Second, the legal boundaries of pointed-to memory are specified explicitly; the goal here is to enhance human readability and maintainability while supporting efficient compilation and running times. As an example, consider the following code declarations:

```
size_t dst_count;
_Array_ptr<char> dst : count(dst_count);
```

The `_Array_ptr<char>` type is a Checked C type for a bounds-checked array, and the `count` annotation indicates how the bounds should be computed. In this case `dst`'s bounds are stored in the variable `dst_count`, but other specifications, such as pointer ranges, are also possible. Checked C also has a `_Ptr<`T`>` type for pointers to single T values, and a `_Nt_array_ptr<`T`>` type for pointers to NUL (zero) terminated arrays. Checked type information is used by the compiler to either prove that an access is safe, or else to insert a bounds check when such a proof is too difficult. Programmers can also use annotations to help the compiler safely avoid adding unnecessary checks in performance-critical code.

Finally, Checked C supports the concept of designated *checked regions* of code. Within these regions, use of unchecked pointers is essentially disallowed, so the above-mentioned checks are sufficient to ensure that *execution is spatially safe*: no failure will occur within the region assuming its checked pointers are well formed (i.e., they have not been corrupted through prior execution of unchecked code). In short, in the parlance of gradual typing, "checked code cannot be blamed" [67] for a spatial safety violation.

Several prior efforts have eschewed annotations, citing the programmer cost of adding them to legacy code. However, in our experience programmers have a sense of the extents and invariants of memory objects and prefer to document and enforce them, but C gives them no easy mechanism to write them down. To assist in the process of updating legacy code, Checked C employs an automated tool to partially rewrite an application to use Checked C types. We believe this approach strikes the right balance: A best-effort analysis can be applied to the whole program to assist in porting, but once ported, a program's annotations ensure efficient checking and assist readability and maintainability. The rewriter uses a global, path-insensitive unification-based algorithm to infer when variables, structure fields, function parameters, and function return values might be converted to Checked C `_Ptr<`T`>` and `_Array_ptr<`T`>` types. It automatically rewrites the program to add the former types, and points to locations for the latter, at which the programmer can convert them by hand, adding needed bounds expressions. To avoid one unsafe pointer use forcing all transitive uses to be unchecked, the rewriter may insert casts, taking advantage of Checked C's ability to mix checked and unchecked code.

Contributions

This chapter makes four main contributions.

First, in Section 3.1, we present Checked C's design and its rationale, introducing its various features by example.

Second, as described in Section 3.2, we have implemented Checked C as an

extension to Clang and LLVM. Since Checked C is a backwards compatible superset of C, any project that compiles today with Clang and LLVM can compile with Checked C. As reported in Section 3.4, we converted most of the standard Olden and Ptrdist benchmark suites to use Checked C. On average, we modified 17.5% of the benchmark code so that 90.7% of it could be placed in checked regions. The mean run-time slowdown is 8.6%, which generally matches or betters Deputy [54] and CCured [62] on the same benchmarks.

Finally, as described in Section 3.3 we have implemented a tool to automatically convert existing C programs to Checked C programs. This tool performs a whole-program, context- and flow-insensitive analysis to identify types that can be replaced with Checked C types, and automatically rewrites them. In about 35 minutes of work the rewriter was able to replace between 37% and 69% of C pointer types with `_Ptr`<T> types in six benchmark programs, comprising more than 290KLOC.

Checked C is under active and ongoing development, and available on the Internet at `https://github.com/Microsoft/checkedc`.

Attribution and acknowledgments

This chapter was adapted from a paper appearing at IEEE SecDev, authored by Sam Elliott, Andrew Ruef, David Tarditi, and Mike Hicks. Checked C the language was designed by David Tarditi. The compiler was implemented by David Tarditi and Sam Elliott with review and feedback from Andrew Ruef. Some elements of the Checked C language were designed by David, Andrew and Mike after

discussion and thought about C programming idioms and past experiences with other language research projects. The rewriting tool was designed, implemented and evaluated by Andrew.

3.1 Checked C

This section presents an overview of Checked C.

3.1.1 Basics

The Checked C extension extends the C language with two additional *checked pointer types*: `_Ptr<T>`, `_Array_ptr<T>` and `_Nt_array_ptr<T>`.[1] The `_Ptr<T>` type indicates a pointer that is used for dereference only and has no arithmetic performed on it, while `_Array_ptr<T>` and `_Nt_array_ptr<T>` support arithmetic with bounds declarations provided in the type. The latter requires NUL termination, which affords some flexibility on determining bounds. The compiler dynamically confirms that checked pointers are valid when they are de-referenced. In blocks or functions designated as *checked code*, it imposes stronger restrictions to uses of unchecked pointers that could corrupt checked pointers, e.g., via aliases. We would expect a Checked C program to involve a mixed of both checked and unchecked code, and a mix of checked and unchecked pointer types.

[1] We use the C++ style syntax for programmer familiarity, and precede the names with an underscore to avoid parsing conflicts in legacy libraries.

```
void next(int *b, int idx, _Ptr<int>out) {
  int tmp = *(b+idx);
  *out = tmp;
}
```

Figure 3.1: Example use of _Ptr<*T*>

3.1.2 Simple pointers

Using _Ptr<*T*> is straightforward: any pointer to an object that is only referenced indirectly, without any arithmetic or array subscript operations, can be replaced with a _Ptr<*T*>. For example, one frequent idiom in C programs is an out parameter, used to indicate an object found or initialized during parsing. Figure 3.1 shows using a _Ptr<int> for the out parameter. When this function is called, the compiler will confirm that it is given a valid pointer, or null. Within the function, the compiler will insert a null check before writing to out. Null checks are elided when the compiler can prove they are unnecessary.

3.1.3 Arrays

The _Array_ptr<*T*> type identifies a pointer to an array of values. Prior safe-C efforts sometimes involve the use of *fat pointers*, which consist both of the actual pointer and information about the bounds of pointed-to memory. Rather than changing the run-time representation of a pointer in order to support bounds checking, in Checked C the programmer associates a *bounds expression* with each _Array_ptr<*T*>-typed variable and member to indicate where the bounds are stored. The compiler inserts a run-time check that ensures that deferencing an _Array_ptr

```
void append(
  _Array_ptr<char> dst : count(dst_count),
  _Array_ptr<char> src : count(src_count),
  size_t dst_count, size_t src_count)
{
  _Dynamic_check(src_count <= dst_count);
  for (size_t i = 0; i < src_count; i++) {
    if (src[i] == '\0') {
      break;
    }
    dst[i] = src[i];
  }
}
```

Figure 3.2: Example use of `_Array_ptr<`T`>`

`<`T`>` is safe (the compiler may optimize away the run-time check if it can prove it always passes). Bounds expressions consist of non-modifying C expressions and can involve variables, parameters, and `struct` field members. For bounds *on* members, the bounds can refer only to other members declared in the same structure. Bounds declarations on members are type-level program invariants that can be suspended temporarily when updating a specific struct object.

Figure 3.2 shows using `_Array_ptr<`T`>` with declared bounds as parameters to a function. In particular, the types of the `dst` and `src` arrays have bound expressions that refer to the function's other two respective parameters. In the body of the function, both `src` and `dst` are accessed as expected. The compiler inserts run-time checks before accessing the memory locations `src[i]` and `dst[i]`. The compiler optimizes away the check on `src[i]` because it can prove that `i` $<$ `src_count`, the size of `src`. The compiler also optimizes away the check `dst[i]` thanks to the

`_Dynamic_check` placed outside the loop. Like an `assert`, this predicate evaluates the given condition and signals a run-time error if the condition is false; unlike `assert`, this predicate is not removed unless proven redundant. Here, its existence assures the compiler that `i` $<$ `dst_count` (transitively), so no per-iteration checks are needed.

There are two other ways to specify array bounds. The first is a *range*, specified by base and upper bound pointers. For example, the bounds expression on `dst` from Figure 3.2 could have been written `bounds(dst,dst+dst_count)`. The second is an alternative to `count` called `bytecount`, which can be applied to either `void*` or `_Array_ptr<void>` types. A `bytecount(n)` expression applied to a pointer `p` would be equivalent to the range `p` through `(char *)p+n`. An example of this is given at the end of this section.

We can also annotate an array declaration as `_Checked`. Any implicit conversion of the array to a pointer value is treated as a `_Array_ptr<`T`>`. We add a restriction that all inner dimensions of checked arrays also be checked. We see both of these situations in Figure 3.4, shortly. Parameters with checked array types are treated as having `_Array_ptr<`T`>` types. If no bounds are declared, the bounds are implied by the array size, if it is known. T `_Checked[]` is a synonym for `_Array_ptr <`T`>`.

```
size_t my_strlcpy(
 _Nt_array_ptr<char> dst: count(dst_sz),
 _Nt_array_ptr<char> src, size_t dst_sz)
{
  size_t i = 0;
  _Nt_array_ptr<char> s : count(i) = src;
  while (s[i] != '\0' && i < dst_sz) {
    dst[i] = s[i];
    ++i;
  }
  dst[i] = '\0';
  return i;
}
```

Figure 3.3: Example use of `_Nt_array_ptr<T>`

3.1.4 NUL-terminated Arrays

The `_Nt_array_ptr<T>` type identifies a pointer to an array of values (often `char`s) that ends with a NUL (`'\0'`). The bounds expression identifies the known-to-be-valid range of the pointer. This range can be expanded by reading the character *just past* the bounds to see if it is NUL.[2] If not, then the bounds can be expanded by one. Otherwise, the current bounds cannot be expanded, and only a `'\0'` may be written to this location. `_Nt_array_ptr<T>` types without explicit bounds default to bounds of `count(0)`, meaning that index 0 can be read safely. A `_Nt_array_ptr<T>` can be cast to a `_Array_ptr<T>` safely; as an `_Array_ptr<T>` the character just past the bounds can no longer be read or written, thus preserving the zero-termination invariant for any aliases.

An example use of `_Nt_array_ptr<T>` is given in Figure 3.3. It implements the `strlcpy` libC routine, which copies `src` to `dst`, which can contain at most `dst_sz`

[2]This means that bounds of `count(n)` requires allocating `n+1` bytes.

characters. We must alias `src` into the local variable `s` so that its count, `i`, can grow dynamically as the loop executes.

3.1.5 Checked and Unchecked Regions

The safety provided by checked pointers can be thwarted by unsafe operations, such as writes to traditional pointers. For example, consider this variation of the code in Figure 3.1:

```
void more(int *b, int i, _Ptr<int *>out) {
  int oldi = i, c;
  do {
    c = readvalue();
    b[i++] = c;
  } while (c != 0);
  *out = b+i-oldi;
}
```

This function repeatedly reads an input value into `b` until a 0 is read, at which point it returns an updated `b` pointer via the checked `out` parameter. While we might expect that writing to `out` is safe, since it is a checked pointer, it will not be safe if the loop overflows `b` and in the process modifies `out` to point to invalid memory.

In a program with a mix of checked and unchecked pointers we cannot and should not expect complete safety. However, we would like to isolate which code is possibly dangerous, i.e., whether it could be the source of a safety violation. Code review and other efforts can then focus on that code. For this purpose Checked C introduces the notion of *checked code regions*. Such code is designated specifically at the level of a file (using a pragma), a function (by annotating its prototype), or a single block (by labeling that block, similar to an `asm` block). Explicitly labeled *un*checked regions may also appear within checked ones.

```
int *out;
_Checked void foo(void) {
  _Ptr<int> ptrout = 0;
  _Unchecked {
    if (out != (int *)0) {
      ptrout = (_Ptr<int>)out; // cast OK
    } else { return; }
  }
  int b _Checked[5][5];
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
      b[i][j] = -1; // access safe
  } }
  *ptrout = b[0][0];
}
```

Figure 3.4: `_Unchecked` and `_Checked` regions (and array)

Figure 3.4 shows a checked function `foo`, which references unchecked pointer `out` within an explicitly labeled `_Unchecked` block. Without this label, the compiler would forbid this cast since it is a potential source of problems (i.e., if `out` was bogus). Within a checked region both null and bounds checks on checked pointers are employed as usual, but additional restrictions are also imposed. In particular, explicit declarations of and casts, reads, and writes from unchecked pointer types are disallowed. Checked regions may neither use varargs nor K&R-style prototypes. These restrictions are meant to ensure that the entire execution of a checked region is *spatially safe*. This means that assuming checked pointers have been constructed properly (in particular, they have not been corrupted by the execution of unchecked code prior to entering the checked region), no safety violations will occur due to dereferencing a pointer into illegal memory.

Checked C also permits ascribing checked types to unchecked functions. We

```
size_t fwrite(
  const void * ptr : byte_count(size*nmemb),
  size_t size, size_t nmemb,
  FILE * stream : itype(_Ptr<FILE>));
```

Figure 3.5: Standard library checked interface

use this feature in a set of *checked headers* for the C standard library. As an example, the type we give to the `fwrite` function is shown in Figure 3.5. The first argument to the function is the target buffer whose size (in bytes) is given by the second and third arguments. The final argument is a `FILE` pointer whose type depends on whether it is being called from checked or unchecked code. For the former, the type is given by the `itype` annotation, indicating it is expected to be a checked pointer. For the latter, it is the "normal" type of the argument.

3.1.6 Restrictions and Limitations

Checked C's design currently imposes several restrictions.

First, to ensure that checked pointers are valid by construction, we require that checked pointer variables be initialized when they are declared. In addition, heap-allocated memory that contains checked pointers (like a struct or array of checked pointers) or is pointed to by a `_Nt_array_ptr`<T> must use `calloc` to ensure safe initialization. We plan to employ something akin to Java's *definite initialization* analysis to relax this requirement, at least somewhat.

Second, we disallow taking the address of variables/members with bounds, variables used in bounds expressions, and members used in structure member bounds expressions. These pointers could be used to subvert the validity of bounds decla-

rations.

Third, `_Array_ptr<T>` values can be dereferenced following essentially arbitrary arithmetic; e.g., if `x` is an `_Array_ptr<int>` we could dereference it via `*(x+y-n+1)` and the compiler will insert any needed checks to ensure the access is legal. However, *updates* to `_Array_ptr<T>` variables are currently more limited. The bounds for a variable are declared when the variable is declared. It is possible, however, that a variable may need different bounds at different points in the program. For example, we might like to replace the loop in Figure 3.2 with:

```
size_t i = 0;
for ( ; i < src_count; i++) {
  if (*src == '\0') break;
  *dst = *src;
  src++; dst++;
}
```

The problem is that the bounds declared for `src` are tantamount to the range (`src`,`src+src_count`). This means that updating `src` to `src+1` would invalidate them, as the upper bound would be off by one. We would like to declare `src` to have new bounds before entering the loop, such as (`src - i`, `src + src_count - i`). We plan to support flow-sensitive declarations of bounds, so that variables can have different bounds at different program points.

Finally, some elements of our static analysis for confirming safe usage are designed but not fully implemented. We elaborate on these in Section 3.2.

3.2 Implementation

We have implemented Checked C as an extension to the Clang/LLVM 5.0 compiler, comprising about 16.5k LoC added or changed (per git diff). This section describes the various changes we made. Our fork of Clang is available online at `https://github.com/Microsoft/checkedc-clang`.

3.2.1 Overview

We extended the Clang C front-end to support the changes described in Section 3.1; the LLVM IR's analyses and optimizers were unchanged. We extended the C grammar to support checked pointers, bounds expressions, and (un)checked blocks, and made corresponding changes to Clang's data structures and static type checker. The compiler enforces the restrictions described in Section 3.1 by statically confirming that array pointer bounds are correctly ascribed and maintained, and by inserting run-time bounds and non-null checks on pointer accesses, which are optimized away by LLVM if they can be proved redundant.

3.2.2 Checking Bounds

Checked C's bounds expressions provide a static description of the bounds on a pointer. We check statically that the sub-expressions of a bounds expression are *non-modifying expressions*: they do not contain any assignment, increment or decrement operators, or function calls. This ensures that using the expressions at bounds checks does not cause unexpected side-effects.

Checked C performs inference to compute a bounds expression that conservatively describes the bounds on a pointer-typed expression. Inference uses bounds expressions normalized into `bounds(`l`,`u`)` form. The inferred bounds are used to check memory accesses using the value of the pointer expression.

For pointer variables, the inferred bounds are the declared bounds. For pointer arithmetic expressions, the inferred bounds are those of the pointer-typed subexpression. When taking the address of a `struct`'s member (`&p->f`), the bounds are those of the particular field. On the other hand, the address of an array element retains the bounds of the whole array. For example, the bounds of `int x[5]` are

```
bounds(x, x+5*sizeof(int))
```

as are the bounds of `&x[3]`, rather than (say)

```
bounds(x+3*sizeof(int),x+4*sizeof(int))
```

The compiler must statically ensure that bounds declarations are valid after assignments and initialization. This requires two steps. First, a *subsumption check* confirms that assigning to a variable (an lvalue, more generally) meets the bounds required of pointers stored in the variable (lvalue). The required pointer bounds must be within (subsumed) by the inferred bounds of the right-hand expression. (Subsumption also applies to initialization and function parameter passing.) This check allows assignment to narrow, but not to widen, the bounds of the right-hand side value. Determining the required bounds is generally straightforward. In the simplest case, the bounds for pointers stored in a variable (lvalue) are directly declared, e.g., for a local variable or function parameter. For assignments to struct members, uses of struct members within the bounds expression for

the member are replaced with an appropriate struct access, For example, given:

```
struct S {
  int len;
  _Array_ptr<int> buf : count(len);
};
```

the required bounds for an assignment to `a.buf` are `a.len`.

Second, the compiler ensures bounds expressions are still valid after a statement modifies a variable in that expression. For example, in Figure 3.3 the bounds of `s` is `count(i)`, but `i` is modified in a loop that iterates over `s` looking for a NUL terminator. For `_Array_ptr`<*T*> types, the modification is justified by subsumption: The updated bounds can be narrowed but not widened. For `_Nt_array_ptr`<*T*> types, we can widen the bounds by 1 byte if we know that the rightmost byte is `'\0'`, e.g., due to a prior check, as is the case in Figure 3.3.

At the moment subsumption checking is rather primitive. Some subsumption checks for bounds declarations that could be statically proven are not. Currently, the static analysis can only reason about bounds expressions that are syntactically equivalent (modulo constant-folding and ignoring non-value changing operations) and bounds expressions that are constant-sized ranges (syntactically equivalent base expressions +/- constant offsets). The main issue is the need to perform a more sophisticated dataflow analysis (at the Clang AST level) to gather and consider relevant facts about relationships between variables (such as equalities and inequalities).

The compiler complains when it cannot (dis)prove a subsumption check in checked code. In our experimental evaluation, we manually review the warnings. We insert the code in an `_Unchecked` block (for checks that are trivially obvious)

or perform a dynamic subsumption check with `_Dynamic_bounds_cast` (which eliminates the error).

We have designed but not yet implemented the analysis that checks assignments to variables used in bounds declarations. For our experimental evaluation, we verified by hand that such assignments do not happen.

3.2.3 Run-time Checks

The compiler inserts run-time checks into the evaluation of lvalue expressions whose lvalue is derived from a checked pointer and whose lvalue will be used to access memory. For example, `*p` produces an lvalue; the run-time check is part of the evaluation of `*p`. These checks are added to the AST, which allows LLVM's optimizers to remove them if it can prove they will always pass.

Before any `_Ptr<`T`>` accesses the compiler inserts a check that the pointer is non-null. Before any `_Array_ptr<`T`>` or `_Nt_array_ptr<`T`>` access the compiler inserts a non-null check followed by the required bounds check computed from the inferred bounds. The compiler does not perform any bounds checks during pointer arithmetic. Programmers can insert dynamic checks (per Figure 3.2) via `_Dynamic_check` and `_Dynamic_bounds_cast`; these, too, may be optimized away.

In some cases, such as a nested dereference expression like `**p`, we may have to emit more than one set of dynamic checks: the first for the outer pointer dereference and another for the inner pointer dereference. Similarly, these checks can be emitted during the calculation of the upper or lower bounds for a bounds check.

The compiler should also disallow arithmetic on a checked pointer if (a) that pointer is null, or (b) the arithmetic would overflow, since both operations could produce a bogus pointer. We have not implemented these checks yet, but doing so should be straightforward. The lack of these checks should not negatively impact our experimental comparison in Section 3.4. Closely related systems Deputy [54] and CCured [62] lack the overflow check, too, and null checks on pointer arithmetic should be inexpensive because they are easily optimized. E.g., the null check on `for` loop-guard `*p` would make redundant the null check on `p++`. Prior instructions and control-flow often imply that null checks are redundant and we're already doing null checks for all checked memory access and struct base expressions. For example *p; p++ is already doing the null check needed on p, as is p-¿f; p++. p[i] already has a null check built in too (we check p before doing the pointer computation).

3.3 Automatic Porting

Porting legacy code to use Checked C's features can be time consuming. To assist the process, we developed a source-to-source translator called *checked-c-convert* that discovers safely-used pointers and rewrites them to be checked. This section describes the tool; it is evaluated in Section 3.4.2.

3.3.1 Conversion tool design and overview

checked-c-convert aims to be sound while also producing edits that are minimal and unsurprising. A rewritten program should be recognizable by the author and

it should be usable as a starting point for both the development of new features and additional porting. A particular challenge is to preserve syntactic structure of the program. Previous, similar analyses rarely interact well with the preprocessor (e.g., they consider macro expansions rather than the original macro) and sometimes require combining multiple source files into one, prior to analysis. These choices are problematic for us: We prefer to rewrite one file at a time (perhaps taking into account whole-program knowledge) and preserve the definition and use of macros, and other formatting, in the source code.

The *checked-c-convert* tool is implemented as a clang libtooling application. It traverses a program's abstract syntax tree (AST) to generate constraints based on pointer usage, solves those constraints, and rewrites the program by promoting some declared pointer types to be checked, and inserting some casts. The tool operates on post-preprocessed code, but has sufficient location information to be able to rewrite the original source files. Moreover, for macro expansions that have parameters, it considers all expansions of those parameters together, so as to be able to rewrite the original macro's definition. In effect, this produces a context- and flow-insensitive rewriting of macros, just as would occur for functions.

3.3.2 Constraint logic and solving

The basic approach is to infer a *qualifier* q_i for each defined pointer variable i. Inspired by CCured's approach [62], qualifiers can be either PTR, ARR and UNK, ordered as a lattice $PTR < ARR < UNK$. Those variables with inferred

qualifier PTR can be rewritten into `_Ptr<`T`>` types, while those with UNK are left as is. Those with the ARR qualifier are eligible to have `_Array_ptr<`T`>` type. For the moment we only signal this fact in a comment and do not rewrite because we cannot always infer proper bounds expressions.

Qualifiers are introduced at each pointer declaration, i.e., parameter, variable, field, etc. Constraints are introduced as a pointer is used, and take one of the following forms:

$$
\begin{array}{l}
q_i = PTR \mid ARR \mid UNK \mid q_j \\
q_i = ARR \quad \Rightarrow \quad q_j = ARR \\
q_i = UNK \quad \Rightarrow \quad q_j = UNK \\
\neg(q_i = PTR \mid ARR \mid UNK)
\end{array}
$$

An expression that performs arithmetic on a pointer with qualifier q_i, either via `+` or `[]`, introduces a constraint $q_i = ARR$. Assignments between pointers introduce aliasing constraints of the form $q_i = q_j$. Casts introduce implication constraints based on the relationship between the sizes of the two types. If the sizes are not comparable, then both constraint variables in an assignment-based cast are constrained to UNK via an equality constraint. One difference from CCured is the use of negation constraints, which are used to fix a constraint variable to a particular Checked C type (e.g., due to a `_Ptr<`T`>` annotation). These would cause problems for CCured, as they might introduce unresolvable conflicts. But Checked C's allowance of checked and unchecked code can resolve them using explicit casts and bounds-safe interfaces, as discussed below.

Constraints are generated for each file individually, at first. Then these constraints are "linked" when it can be determined that they refer to the same global

definition. Solving the constraints produces a qualifier assignment $q_i = X$ where X is PTR, ARR, or UNK. Each qualifier is initially assigned to PTR. Solving the constraints works iteratively by propagating aliasing and equality constraints to UNK first; then aliasing, equality and implication constraints involving UNK; then aliasing, implication and equality constraints to ARR. This algorithm runs in linear time proportional to the number of pointer variables in the program.

One problem with unification-based analysis is that a single unsafe use might "pollute" the constraint system by introducing an equality constraint to UNK that transitively constrains unified qualifiers to UNK as well. For example, casting a `struct` pointer to a `unsigned char` buffer to write to the network would cause all transitive uses of that pointer to be unchecked. The tool takes advantage of Checked C's ability to mix checked and unchecked pointers to solve this problem. In particular, constraints for each function are solved locally, using separate qualifier variables for each external function's declared parameters.

Our modular algorithm runs as follows:

1. The AST for every compilation unit is traversed and constraints are generated based on the uses of pointer variables. Each pointer variable that appears at a physical location in the program is given a unique constraint variable. A distinction is made for parameter and return variables depending on if the associated function definition is a *declaration* or a *definition*:

    - *Declaration*: There may be multiple declarations. The constraint variables for the parameters and return values in the declarations are all

constrained to be equal to each other.

- *Definition*: There will only be one definition. These constraint variables are not constrained to be equal to the variables in the declarations. This enables modular reasoning.

At call sites, the constraint variables used for a functions parameters and return values come from those in the declaration, not the definition.

2. After the AST is traversed, the constraints are solved using unification. The result is a set of satisfying assignments to pointer constraint variables.

3. Then, the AST is re-traversed. At each physical location associated with a constraint variable, a re-write decision is made based on the value of the constraint variable. These physical locations are variable declaration statements, either as members of a `struct`, function variable declarations, or parameter variable declarations. There is a special case, which is any constraint variable appearing at a parameter position, either at a function declaration/definition, or, a call site.

   - *Declaration or definition*: The re-write decision is based off of the imbalance between the constraint variables on the declaration, and the constraint variables on the definition. There are three cases:

     – *No imbalance*: In this case, the re-write is made based on the value of the constraint variable in the solution to the unification

     – *Declaration is safer than definition*: In this case, there is nothing to

do for the function, since the function does unknown things with the pointer. This case will be dealt with at the call site.

  - *Decalaration is less safe than definition*: In this case, there are call sites that are unsafe, but the function itself is fine. We can re-write the function declaration and definition with a bounds-safe interface.

- *Call site*: The only decision here is whether or not the *declaration is safer than definition* case holds for the function called at the call site. If it does, a cast is inserted.

4. All of the re-write decisions are then applied to the source code of the program.

This modular reasoning can result in conflicting constraints once completed, i.e. the constraint variable for the parameter in the function definition has a different valuation than the constraint variable for the parameter at the functions use. These conflicts are the result of two different cases: either the caller is less safe than the callee, or vice versa. These cases can be determined by considering the relationship between the valuations on the PTR to UNK lattice.

For an example of when the caller is less safe than the callee, consider the case where a function might make safe use of the parameter within the body of the function, but a caller of the function might perform casts or operations on the parameter value that would prohibit the value from becoming a `_Ptr`<T>. In this situation, we would like to make the parameter a `_Ptr`<T>, to allow for checked uses of the function and to check the function itself, but we cannot do so straight away because the function is used in at least one "unsafe" context. Our solution is

to instead change the parameter to a bounds safe interface type. This allows the function to be simultaneously addressed from both checked and unchecked code.

If instead the caller is safer than the callee (e.g., the caller passes a `_Ptr<int>` to a function that requires a `int*`), we insert a cast at the call site. This cast makes evident to the programmer the potential risk of the call, and can be fixed manually if the callee is conservatively misclassified by the tool.

This approach has advantages and disadvantages. It favors making the fewest number of modifications across a project. One alternative would be to change the parameter type to a `_Ptr<`T`>`directly, and then insert casts at each call site. This would tell the programmer where potentially bogus pointer values were, but would also increase the size of the changes made. Our approach does not immediately tell the programmer where the pointer changes need to be made. However, the Checked C compiler will, if the programmer takes a bounds-safe interface and manually converts it into a non-interface `_Ptr<`T`>`type. Every location that would require a cast will fail to type check.

3.3.3 Example

Consider the following function and it's calling context:

```
void f1(int *a) {
  *a = 0;
}

void caller(void) {
  int q = 0;
  f1(&q);
  f1(((int*) 0x8f8000));
}
```

The function `f1` is safe and the parameter could be re-written to be a `_Ptr` <T>if considered in isolation. However, it is used from an unsafe context. This is unfortunate - perhaps the unsafe context ought to be re-visited. Either way, it would be nice to have an incremental conversion that makes `f1` a little safer than it was before.

Instead of tying the constraints for formal and actual parameters together globally, we can keep them separate, run unification, and then compare the difference between the formal and actual parameters to functions. If the actual parameters, i.e. the variables constrained by uses of `f1` in `caller`, are less permissive than those in the function definition, then that implies there is an "unsafe" context the function is called from, but the function itself is "safe". Checked C gives us a mechanism to reason about and express this situation: bounds-safe interfaces. The re-writer would, in the face of the above situation, output the following re-write to `f1`:

```
void f1(int *a : itype(_Ptr<int>)) {
  *a = 0;
}
```

The `itype` syntax indicates that `a` can be supplied by the caller as either an `int*` or a `_Ptr`<T>, but the function body will treat `a` as a `_Ptr`<T>.

3.4 Empirical Evaluation

This section presents an evaluation of the Checked C compiler and porting tool, considering performance and efficacy.

Name	LoC	Description
bh	1,162	Barnes & Hut N-body force computation
bisort	262	Sorts using two disjoint bitonic sequences
em3d	476	Simulates electromagnetic waves in 3D
health	338	Simulates Columbian health-care system
mst	325	Minimum spanning tree using linked lists
perimeter	399	Perimeter of quad-tree encoded images
power	452	The Power System Optimization problem
treadd	180	Sums values in a tree
tsp	415	Estimates Traveling-salesman problem
voronoi	814	Voronoi diagram of a set of points
anagram	346	Generates anagrams from a list of words
bc	5,194	An arbitrary precision calculator
ft	893	Fibonacci heap Minimum spanning tree
ks	549	Schweikert-Kernighan partitioning
yacr2	2,529	VLSI channel router

Table 3.1: Compiler Benchmarks. Top group is the Olden suite, bottom group is the Ptrdist suite. Descriptions are from [68,69]. We were unable to convert *voronoi* from the Olden suite and *bc* from the Ptrdist suite using the current version of Checked C.

3.4.1 Compiler evaluation

We converted two existing C benchmarks as an initial evaluation of the consequences of porting code to Checked C. We quantify both the changes required for the code to become checked, and the overhead imposed on compilation, running time, and executable size.

We chose the Olden [68] and Ptrdist [69] benchmark suites, described in Table 3.1, because they are specifically designed to test pointer-intensive applications, and they are the same benchmarks used to evaluate both Deputy [54] and CCured [62]. We did not convert *bc* from the Ptrdist suite and *voronoi* from the Olden suite for lack of time, but plan to soon.

	Code Changes			Observed Overheads		
Name	*LM %*	*EM %*	*LU %*	*RT ±%*	*CT ±%*	*ES ±%*
bh	10.0	76.7	5.2	+0.2	+23.8	+6.2
bisort	21.8	84.3	7.0	0.0	+7.3	+3.8
em3d	35.3	66.4	16.9	+0.8	+18.0	-0.4
health	24.0	97.8	9.3	+2.1	+18.5	+6.7
mst	30.1	75.0	19.3	0.0	+6.3	-5.0
perimeter	9.8	92.3	5.2	0.0	+4.9	+0.8
power	15.0	69.2	3.9	0.0	+21.6	+8.5
treadd	17.2	92.3	20.4	+8.3	+83.1	+7.0
tsp	9.9	94.5	10.3	0.0	+47.6	+4.6
anagram	26.6	67.5	10.7	+23.5	+16.8	+5.1
ft	18.7	98.5	6.3	+25.9	+16.5	+11.3
ks	14.2	93.4	8.1	+12.8	+32.3	+26.7
yacr2	14.5	51.5	16.2	+49.3	+38.4	+24.5
Mean:	17.5	80.1	9.3	+8.6	+24.3	+7.4

Table 3.2: Benchmark Results. Key: *LM %*: Percentage of Source LoC Modified, including Additions; *EM %*: Percentage of Code Modifications deemed to be Easy (see 3.4.1.1); *LU %*: Percentage of Lines remaining Unchecked; *RT ±%*: Percentage Change in Run Time; *CT ±%*: Percentage Change in Compile Time; *ES ±%*: Percentage Change in Executable Size (`.text` section only). *Mean*: Geometric Mean.

The evaluation results are presented in Table 3.2. These were produced using a 12-Core Intel Xeon X5650 2.66GHz, with 24GB of RAM, running Red Hat Enterprise Linux 6. All compilation and benchmarking was done without parallelism. We ran each benchmark 21 times with and without the Checked C changes using the test sizes from the LLVM versions of these benchmarks. We report the median; we observed little variance.

3.4.1.1 Code Changes

On average, we modified around 17.5% of benchmark lines of code. Most of these changes were in declarations, initializers, and type definitions rather than in the program logic. In the evaluation of Deputy [70], the reported figure of lines changed ranges between 0.5% and 11% for the same benchmarks, showing they have a lower annotation burden than Checked C.

We modified the benchmarks to use checked blocks and the top-level checked pragma. We placed code that could not be checked because it used unchecked pointers in unchecked blocks. On average, about 9.3% of the code remained unchecked after conversion, with a minimum and maximum of 3.9% and 20.4%. The cause was almost entirely variable-argument `printf` functions.

We manually inspected changes and divided them into *easy* changes and *hard* changes. Easy changes include: replacing included headers with their checked versions; converting a $T*$ to a `_Ptr`<T>; adding the `_Checked` keyword to an array declaration; introducing a `_Checked` or `_Unchecked` region; adding an initializer; and replacing a call to `malloc` with a call to `calloc`. Hard changes are all other changes, including changing a $T*$ to a `_Array_ptr`<T> and adding a bounds declaration, adding structs, struct members, and local variables to represent run-time bounds information, and code modernization.

In all of our benchmarks, we found the majority of changes were easy. In six of the benchmarks, the only "hard" changes were adding bounds annotations relating to the parameters of `main`.

In three benchmarks—em3d, mst, and yacr2—we had to add intermediate structs so that we could represent the bounds on `_Array_ptr<T>`s nested inside arrays. In mst we also had to add a member to a struct to represent the bounds on an `_Array_ptr<T>`. In the first case, this is because we cannot represent the bounds on nested `_Array_ptr<T>`s, in the second case this is because we only allow bounds on members to reference other members in the same struct. In em3d and anagram we also added local temporary variables to represent bounds information. In yacr2 there are a lot of bounds declarations that are all exactly the same where global variables are passed as arguments, inflating the number of "hard" changes.

3.4.1.2 Observed Overheads

The average run-time overhead introduced by added dynamic checks was 8.6%. In more than half of the benchmarks the overhead was less than 1%. We believe this to be an acceptably low overhead that better static analysis may reduce even further.

In all but two benchmarks—treadd and ft—the added overhead matches (is within 2%) or betters that of Deputy. For yacr2 and em3d, Checked C does substantially better than Deputy, whose overheads are 98% and 56%, respectively. Checked C's overhead betters or matches that reported by CCured in every case but ft.

On average, the compile-time overhead added by using Checked C is 24.3%. The maximum overhead is 83.1%, and the minimum is 4.9% faster than compiling with C.

We also evaluated code size overhead, by looking at the change in the size of `.text` section of the executable. This excludes data that might be stripped, like debugging information. Across the benchmarks, there is an average 7.4% code size overhead from the introduction of dynamic checks. Ten of the programs have a code size increase of less than 10%.

3.4.2 Porting Tool Evaluation

We also evaluated the efficacy of our porting tool. To do so, we ran it on six programs and libraries and recorded how many pointer types the rewriter converted and how many casts were inserted. We chose these programs as they represent legacy, low level libraries that are used in commodity systems and frequently in security-sensitive contexts.

Table 3.3 contains the results. The value in the `_Ptr`<T> column indicates the number of `_Ptr`<T> added to the program that replace standard C pointers. These are re-written at the location they are declared. After investigation, there are usually two reasons that a pointer cannot be replaced with a `_Ptr`<T>: either some arithmetic is performed on the pointer, or it is passed as a parameter to a library function for which a bounds-safe interface does not exist. We consider a value replaced by `_Ptr`<T>whether or not it was inserted as a bounds-safe interface. The table also indicates the versions of each program as computed with cloc and the number of casts inserted, compared to the percentage of call sites re-written to include casts, as well as the number of functions replaced with bounds-safe interfaces.

This experiment represents the first step a developer would take to adopting Checked C into their project. The values converted into `_Ptr<`T`>` by the re-writer need never be considered again during the rest of the conversion or by subsequent software assurance / bug finding efforts.

Program	**# of ∗**	**%/# _Ptr**	**Arr.**	**Unk.**	**Casts(Calls)**	**Interfaces(Functions)**	**LOC**
zlib 1.2.8	4514	46%/2076	5%/222	49%/2216	8 (300)	464 (1188)	17388
sqlite 3.18.1	34230	38%/13148	3%/910	59%/20172	2096 (29462)	9132 (23305)	106806
parson	1132	35%/392	1%/14	64%/726	3 (378)	340 (454)	2320
lua 5.3.4	15114	23%/3464	1%/103	76%/11547	175 (1443)	784 (2708)	13577
libtiff 4.0.6	34518	26%/8940	1%/424	73%/25154	495 (1986)	1916 (5812)	62439

Table 3.3: Number of pointer types converted. The # of ∗ column represents the number of pointer types in the program. The Arr and Unk columns represent constraints where the rewriter determined that the access into the pointer was via indexing (Arr) or that the constraints can't be captured by the rewriter (Unk) due to casts, assignment to a non-zero literal, or some other operation. The Casts column represents the number of casts inserted, compared to the percentage of call sites that were re-written to include a cast. The Interfaces column represents the functions that were modified to use a bounds-safe interface.

3.5 Related work

There has been extensive research addressing out-of-bounds memory accesses in C [52]. The research falls into 4 categories: languages, implementations, static analysis, and security mitigations.

Safe languages

Cyclone [53] and Deputy [54, 70] are type-safe dialects of C. Cyclone's key novelty is its support for GC-free temporal safety [71, 72]. Checked C differs from Cyclone by being backward compatible (Cyclone disallowed many legacy idioms) and avoiding pointer format changes (e.g., Cyclone used "fat" pointers to support arithmetic). Deputy keeps pointer layout unchanged by allowing a programmer to describe the bounds using other program expressions. Checked C builds on this, but make bounds checking a first-class part of the language. Deputy incorporates the bounds information into the types of pointers by using dependent types. This makes type checking hard to understand. Deputy requires that values of all pointers stay in bounds so that they match their types. To enforce this invariant (and make type checking decidable), it inserts run-time checks before pointer arithmetic, not at memory accesses. Checked C uses separate annotations that describe bounds invariants instead of incorporating bounds into pointer types and inserts run-time checks at memory accesses.

Like Cyclone, programming languages like D [73] and Rust [55] aim to support safe, low-level systems-oriented programming without requiring GC. Go [56] and C#

[74] target a similar domain. Legacy programs would need to be ported wholesale to take advantage of these languages, which could be a costly affair.

Safe C implementations

Rather than use a new language, several projects have looked at new ways to implement legacy C programs so as to make them spatially safe. The *bcc* source-to-source translator [75] and the *rtcc* compiler [76] changed the representations of pointers to include bounds. The *rtcc*-generated code was 3 times larger and about 10 times slower. Fail-Safe C [77] changed the representation of pointers and integers to be pairs. Benchmarks were 2 to 4 times slower. CCured [62] employed a whole-program analysis for transforming programs to be safe. Its transformation involved changes to data layout (e.g., fat and "wild" pointers), which could cause interoperation headaches. Compilation was all-or-nothing: unhandled code idioms in one compilation unit could inhibit compilation of the entire program. Our rewriting algorithm is inspired by CCured's analysis with the important differences that (a) not every pointer need be made safe, and (b) the output is not a step in compilation, but programmer-maintainable source code.

Safety can also be offered by the loader and run-time system. "Red zones", used by Purify [78, 79] are inserted before and after dynamically-allocated object and between statically-allocated objects, where bytes in the red zone are marked as inaccessible (at a cost of 2 bits per protected byte). Red-zone approaches cannot detect out-of-bounds accesses that occur entirely within valid memory for other

objects or stack frames or intra-object buffer overruns (a write to an array in a struct that overwrites another member of the struct). Checked C detects accesses to unrelated objects and intra-object overruns.

Similar tools include Bounds Checker [80], Dr. Memory [81, 82], Intel Inspector [83], Oracle Solaris Studio Code Analyzer [84], Valgrind Memcheck [85, 86], Insure++ [87], and AddressSanitizer (ASAN) [64]. ASAN is incorporated into the LLVM and GCC compilers. It tracks the state of 8-byte chunks in memory. It increases SPEC CPU program execution time by 73% when checking reads and writes and 26% when only checking writes. SPEC CPU2006 average memory usage is 3.37 times larger. Light-weight Bounds Checking [88] uses a two-level table to reduce memory overhead.

Checking that accesses are to the proper objects can be done using richer side data structures that track object bounds and by checking that pointer arithmetic stays in bounds [63, 89–94]. Baggy Bounds Checking [92] provides a fast implementation of object bounds by reserving 1/n of the virtual address space for a table, where n is the smallest allowed object size and requiring object sizes be powers of 2. It increases SPECINT 2000 execution time by 60% and memory usage by 20%. SoftBound [63] tracks bounds information by using a hash table or a shadow copy of memory. It increases execution time for a set of benchmarks by 67% and average memory footprint by 64%. SoftBound can check only writes, in which case execution time increases by 22%. For libraries that cannot be recompiled, wrapper functions must be provided that update metadata. Checked C only requires that checked headers be provided.

There is also work on adding temporal safety with different memory allocation implementations, e.g., via conservative garbage collection [95] or regions [71, 72]. Checked C focuses on spatial safety both due to its importance at stopping code injection style attacks as well as information disclosure attacks, though temporal safety is important and we plan to investigate it in the future.

Static analysis

Static analysis tools take source or binary code and attempt to find possible bugs, such as out-of-bounds array accesses, by analyzing the code. Commercial tools include CodeSonar, Coverity Static Analysis, HP Fortify, IBM Security AppScan, Klocwork, Microsoft Visual Studio Code Analysis for C/C++, and Polyspace Static Analysis [57, 96, 97]. Static analysis tools have difficulty balancing precision and performance. To be precise, they may not scale to large programs. While imprecision can aid scalability, it can result in false positives, i.e., error reports that do not correspond to real bugs. False positives are a significant problem [57]. As a result, tools may make unsound assumptions (e.g., inspecting only a limited number of paths through function [96]) but the result is they may also miss genuine bugs (false negatives). Alternatively, they may focus on supporting coding styles that avoid problematic code constructs, e.g., pointer arithmetic and dynamic memory allocation [58, 59, 98, 99]. Or, they may require sophisticated side conditions on specifications, i.e., as pre- and post-conditions at function boundaries, so that the analysis can be modular, and thus more scalable [100]. Additionally, some work [7]

focuses on analyzing string manipulating programs written in C and inferring modular contracts for those programs.

Checked C occupies a different design point than static analysis tools. It avoids problems with false positives by deferring bounds checks to run-time—in essence, it trades run-time overhead for soundness and coding flexibility. In addition, Checked C avoids complicated specifications on functions. For example, a modular static analysis might have required the code in Figure 3.2 to include that `src_count` $\leq$ `dst_count` as a function pre-condition. While this constraint is not particularly onerous, some specifications can be. In Checked C, such side conditions are unnecessary; instead, soundness ensured by occasional dynamic checks.

Security mitigations

Security mitigations employ run-time-only mechanisms that detect whether memory has been corrupted or prevent an attacker from taking control of a system after such corruption. They include data execution prevention (DEP), software fault isolation (SFI) [101] , address-space layout randomization (ASLR) [102, 103], stack canaries [104], shadow stacks [105, 106], and control-flow integrity (CFI) [61]. DEP, ASLR, and CFI focus on preventing execution of arbitrary code and control-flow modification. Stack protection mechanisms focus on protecting data or return addresses on the stack.

Checked C provides protection against data modification and data disclosure attacks, which the other approaches do not. For example, ASLR does not protect

against data modification or data disclosure attacks. Data may be located on the stack adjacent to a variable that is subject to a buffer overrun; the buffer overrun can be be used reliably to overwrite or read the data. Shadow stacks do not protect stack-allocated buffers or arrays, heap data, and statically-allocated data. Chen et al. [107] show that data modification attacks that do not alter control-flow pose a serious long-term threat. The Heartbleed attack illustrates the damage possible.

3.6 Summary

This chapter presented Checked C, an extension to C to help enforce spatial safety. Checked C's design is focused on interoperability with legacy C, usability, and efficiency. Checked C's novel notion of *checked regions* ensures that "checked code cannot be blamed" for a safety violation. Our implementation of Checked C as a Clang/LLVM extension enjoys good performance. To assist in incrementally strengthening legacy code, we have developed a porting tool for automatically rewriting code to use checked pointers.

Chapter 4: Volume Estimation for Numeric Invariant Generation

4.1 Introduction

One approach taken to identify defects in software is static analysis via abstract interpretation. A key part of this analysis is choosing an abstraction for program states, usually program variables. Sometimes this abstraction can be simple while enabling useful analyses, for example identifying unused variables or constant pointer values. However, sometimes a numeric abstraction is required that will relate numeric variables to either each other or numeric constants. We first encountered this when conceptualizing a static analyzer that could be used to prove the absence of timing based side channels [6], a project for which we wrote a static analyzer for Java bytecode that used numeric abstractions.

While implementing this analyzer, we surveyed the literature on the design and implementation of static analyzers, especially with regards to key features: should the analyzer operate in a top down or bottom up manner? Should the analysis be intra-procedural or inter-procedural? What abstraction should be made of references or the heap? We saw no clear systematic evaluation of the trade-offs involved in constructing such analyzer, so we conducted one, published in the European Symposium on Programming (ESOP) 2018 [8].

This evaluation instantiated our numeric analyzer with the different combination of configurations. Each analyzer configuration tried to prove whether or not array accesses contained within the DaCapo [108] were in bounds, and a configuration was judged to be better than another if it could prove more accesses in bounds than the other.

However, we only implemented and evaluated numeric abstraction using single element abstractions. One restriction with these abstractions, such as intervals [109], octagons [110], and polyhedra [111] is that they are convex. A convex abstraction is unable to represent the absence of one or more concrete states within its volume. This causes problems, for instance, when attempting to prove that a division by zero is not possible because the set of all integers except zero is not representable with a single convex abstraction. To work around this problem, a common approach [112, 113] is to use a powerset abstraction [114]. A *powerset abstraction* represents a non-convex set of states as a finite set of convex abstractions. Since (linear) convex abstractions are representable as a conjunction of hyperplanes, we refer to these powerset abstractions as *disjunction abstractions*. For instance, we can represent that x is equal to any integer except zero using a disjunction of two interval constraints: $x \leq -1 \vee x \geq 1$.

While disjunction abstractions solve the problem of representing holes within a convex numeric abstraction, they also introduce a new problem: performance. Instead of performing an operation on a single convex abstraction, the analysis must perform operations on each convex abstraction. Furthermore, if a disjunction is introduced at each branch in the program [115], the number of disjuncts is expo-

nential in the number of branches. Loops cause further problems because they can effectively introduce an unbounded number of branches, leading to an analysis that does not terminate.

To resolve this problem we turn to merging. A *merge heuristic* is responsible for determining whether two convex numeric abstractions should be combined using a hulling operation or maintained as separate disjuncts. In [112], the authors propose a heuristic based on Hausdorff distance. In [113], the authors propose a heuristic based on the number of common hyperplanes. The problem with both of these approaches is that they do not relate directly to what the abstractions represent: concrete states.

This chapter studies merge heuristics based on the number of concrete states that are affected by a potential merge. For instance, if two abstract states have no concrete states in common, then perhaps they should not be merged. Alternatively, if hulling two abstract states yields the same set of concrete states as taking their union, they should be merged as in [116].

In this chapter we focus on *bounded polytope abstractions*, which are polytope abstractions with finite bounds. We study polytopes because, unlike intervals [117,118], appropriate merge heuristics for polytopes are non-obvious. Furthermore, operations on polytopes have higher complexity than operations on octagons or intervals and thus make differences between merge heuristics more obvious in empirical study. Regardless, we expect that the precision results would extend to other convex numeric abstractions. In order to study volume-based merge heuristics, we require computable volumes. Therefore we restrict polytopes to machine

integer bounds with the assumption that integer overflow is checked. We make the following contributions.

- In Section 4.3 we develop heuristics based on the volume of the intersection of two polytopes relative to the volume of their union. We also develop heuristics based on the volume of the hull of two polytopes relative to the volume of their union.

- In Section 4.4 we describe how to use Markov Chain Monte Carlo algorithms to incrementally approximate relative volumes of polytopes, and we show how to use Barvinok's algorithm [119] to count integer points in polytopes. We also describe a segments-based affinity score that does not require a direct hull computation.

- In Section 4.5 we present a disjunctive abstract domain that utilizes various heuristics to determine which disjuncts to merge.

- In Sections 4.6 and 4.7 we integrate the abstract domain into an analyzer and use that analyzer to produce invariants for a range of programs in the SV-COMP and WCET benchmark suites.

Attribution and acknowledgment

This work was previously published in the 2018 proceedings of the Static Analysis Symposium as "Volume-Based Merge Heuristics for Disjunctive Numeric Domains" with co-authors Kesha Hietala and Arlen Cox. Arlen had the insight to

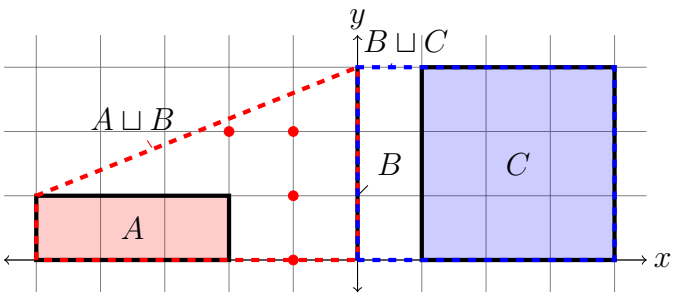
use volume as a heuristic, Andrew investigated and implemented the inclusion of the heuristic into a static analyzer. Arlen, Andrew and Kesha all thought about how intersection and hull volume could be used for affinity, Andrew conceived of and implemented using Barvinok for affinity measurement. Andrew also implemented the static analysis integration and carried out the experiments. Kesha and Arlen implemented the MCMC hull relaxation volume estimator.

4.2 Overview and Example

Consider a typical forward abstract interpretation [120] of the program shown in Figure 4.1a. The analysis should establish as strong an invariant as possible at the point where the branches A, B, and C have been joined together. We want to strike a balance between precision and performance: our invariant should be strong enough to allow us to prove interesting properties of the program, but we should not have to spend an unreasonable amount of computational power.

Figure 4.1b shows the situation that arises. There are three disjoint polytopes. Each describes a range of values that can be assumed by `x` and `y` when the branches are joined together at the end of Figure 4.1a. Now consider a case where we are allowed to describe the state using a disjunction of at most two polytopes. Then we must choose to merge two of A, B, or C. The question is, which two will result in the least loss of precision? The observation we make is that precision loss is related to volume. When the volume increases as a result of a merge, that represents a precision loss. The magnitude of the increase in volume is also related to the

```
assume(0 <= y <= 3);
assume(-5 <= x <= 4)
    ;
if(x <= -2) {
  assume(y <= 1);
  // A
} else if(x == 0) {
  // B
} else if(x >= 1) {
  // C
} else {
  assume(false);
}
```



a b

Figure 4.1: Example of merging disjuncts. (a) Program that produces three disjuncts. (b) Three disjuncts shown as three convex polytopes. Red (resp. blue) dashed lines show the merge results of A and B (resp. B and C). Dots show integer points added in merging.

magnitude of the loss of precision. It is therefore desirable to merge the disjuncts that minimize the change in volume. In short: can we speculatively calculate or estimate the volume increase from a proposed disjunctive merge, and let that guide the management of our disjuncts? We will consider answering this question using two different volume calculation methods.

First, we consider an integer point counting method. We can see in Figure 4.1b that merging A and B will cause four new points (shown in red) to be added to the approximation of the state space, while merging B and C will not result in any

change in the number of integer points. Therefore we choose to merge disjuncts B and C while keeping A distinct.

Second, we consider a real approximation of the integer points methods. We can see that if we merge A and B, the volume increases by 7 (red dashed shape), whereas if we merge B and C the volume increases by 3 (blue dashed shape). Therefore we choose to merge disjuncts B and C while keeping A distinct.

In the remainder of the chapter we precisely describe the comparison techniques used in this section. Both integer point methods and real approximation methods are considered.

4.3 Semantic Comparison of Polytopes

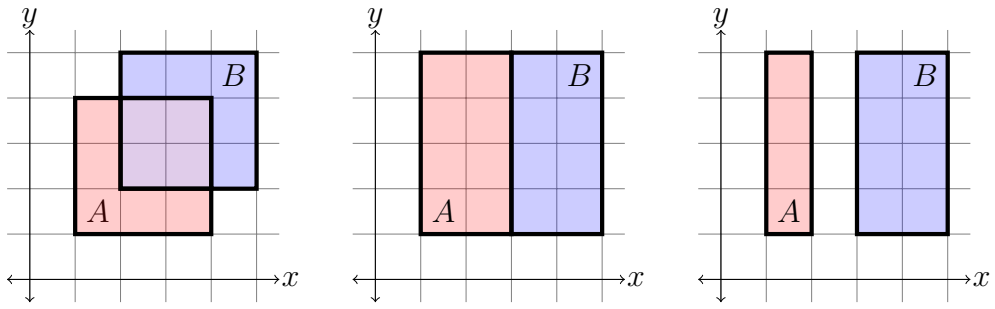
This section develops affinity scores between polytopes. An *affinity score* is a value in the range [0,1] assigned to a pair of polytopes where a 0 suggests that the polytopes may not be related and a 1 suggests that the polytopes are definitely related. Polytopes with an affinity score higher than a (user-specified) threshold will be merged. Table 4.1 summarizes the two affinity scoring mechanisms evaluated in this chapter.

Each affinity score is defined in two ways: over integers and over reals. For integers, the affinity score is given by the cardinality of point sets. For reals, the affinity score is given by the volume of the solids. The computation of both the cardinality of the point sets and the volume of the solids requires that the polytopes are bounded to avoid infinite results. Integer affinity scores are given a $\mathbb{Z}$ subscript

Table 4.1: Affinity scores measure the similarity between two convex polytopes and can be used to determine which polytopes to merge. We define $|A|_{\mathbb{Z}}$ to be the cardinality of $\{ \mathbf{x} \in \mathbb{Z}^d \mid \mathbf{x} \in A \}$ and $|A|_{\mathbb{R}}$ to be the volume of the polytope A.

Affinity Score	Integer	Real
Intersection volume	$i_{\mathbb{Z}}(A,B) = \dfrac{\lvert A \cap B \rvert_{\mathbb{Z}}}{\lvert A \cup B \rvert_{\mathbb{Z}}}$	$i_{\mathbb{R}}(A,B) = \dfrac{\lvert A \cap B \rvert_{\mathbb{R}}}{\lvert A \cup B \rvert_{\mathbb{R}}}$
Added hull volume	$h_{\mathbb{Z}}(A,B) = \dfrac{\lvert A \cup B \rvert_{\mathbb{Z}}}{\lvert \text{hull}(A,B) \rvert_{\mathbb{Z}}}$	$h_{\mathbb{R}}(A,B) = \dfrac{\lvert A \cup B \rvert_{\mathbb{R}}}{\lvert \text{hull}(A,B) \rvert_{\mathbb{R}}}$

and real affinity scores are given a $\mathbb{R}$ subscript.



a b c

Figure 4.2: Example polytopes that motivate different affinity scoring systems

To motivate the different affinity scoring systems, we use the examples shown in Figure 4.2. Figure 4.2a shows two polytopes that are similar because they have a large overlap. Figure 4.2b shows two polytopes that are similar because they abut (and hence merging them will result in no loss of precision). During static analysis, we often encounter examples like the one in Figure 4.2b because when we branch on an if statement, for the 'then' branch we assume one half-space and for the 'else' branch we assume the other half-space (in this case separated by $x = 3$). We also

often encounter examples like Figure 4.2c, which has a gap of size one. This is due to branching on integer variables: if we branch on $x \geq 3$, our else constraint is $x \leq 2$.

Definition 1 (Intersection volume affinity)**.** *Intersection volume affinity is defined as the ratio between the volume of the intersection of the polytopes and the volume of the union of the polytopes. It is defined in the first row of Table 4.1.*

Intuitively, intersection volume is a good scoring mechanism because it merges polytopes that have large overlaps. The hull of two polytopes with a large intersection will not be significantly larger than the polytopes themselves. However, a small or non-existent intersection between two polytopes does not indicate anything about the size of their hull. What is particularly useful about this scoring mechanism is that the hulling operation can be skipped if unneeded. Since the hulling operation is potentially exponential time for arbitrary polytopes, this could lead to performance benefits.

Example 1 (Intersection volume affinity)**.** *For Figure 4.2a, $i_{\mathbb{Z}}$ is $\frac{9}{23} \approx 0.39$ and $i_{\mathbb{R}}$ is $\frac{4}{14} \approx 0.29$. For Figure 4.2b, $i_{\mathbb{Z}}$ is $\frac{1}{5} = 0.2$ and $i_{\mathbb{R}}$ is 0. For Figure 4.2c, both $i_{\mathbb{Z}}$ and $i_{\mathbb{R}}$ are 0.*

Definition 2 (Added hull volume affinity)**.** *Added hull volume affinity is defined as the ratio between the volume of the union of the polytopes and the volume of the hull of the polytopes. It is defined in the second row of Table 4.1.*

Due to the situation that occurs in Figures 4.2b and 4.2c, we also consider hull volume affinity, which corresponds directly to the volume/number of points that are

gained through the hulling process. This scoring mechanism aims to minimize the total number of points represented by an abstraction.

Example 2 (Added hull volume affinity)**.** *For Figure 4.2a, $h_{\mathbb{R}}$ is $\frac{14}{15} \approx 0.93$. For Figure 4.2b, $h_{\mathbb{R}}$ is 1. For Figure 4.2c, $h_{\mathbb{R}}$ is $\frac{12}{16} = 0.75$. For all three figures, $h_{\mathbb{Z}}$ is 1.*

Other affinity scores are documented in the literature. The simplest affinity [121] is the null affinity, which always returns an affinity score of zero. Another affinity score [113] is the ratio between the number of half planes preserved by a hulling operation and the number of half planes in the two polytopes. This is biased to preserve complexity in the representation, but shares with the hull volume affinity the property that it tends to assign high scores to polytopes that do not add too many points in hulling. In [112] there is an affinity score that is based on the Hausdorff distance. This affinity tends to merge polytopes that are not too far apart, but does not consider points gained by the hulling operation.

4.4 Sampling and Counting Points

In this section we describe the techniques we use to implement affinity scores. Affinity scores are computed with one of two general techniques. They are either computed by counting integer points within polytopes or by calculating ratios of volumes of polytopes.

4.4.1 Integer-Point-Based Affinity

To implement $i_{\mathbb{Z}}$ and $h_{\mathbb{Z}}$ we need to be able to compute answers to problems of the form $|A|_{\mathbb{Z}} \,/\, |B|_{\mathbb{Z}}$. We accomplish this by computing individually $|A|_{\mathbb{Z}}$ and $|B|_{\mathbb{Z}}$ and then dividing. The key to doing this is the use of the Barvinok algorithm [119] and its corresponding tool [10]. The Barvinok algorithm has complexity $L^{O(d \log d)}$ for L input constraints and dimension d [122]. The Barvinok library (developed from PolyLib [123]) is an optimized implementation of this algorithm and can efficiently compute the precise cardinality of integer polytopes. The details of this algorithm are beyond the scope of this chapter.

4.4.2 Volume-Ratio-Based Affinity

To implement $i_{\mathbb{R}}$ and $h_{\mathbb{R}}$ we need to be able to compute ratios of volumes of high-dimension polytopes. Directly computing the volume of high-dimension polytopes is a computationally complex problem and we need to do the operation twice for each merge candidate. Therefore, we develop the methodology used here more carefully.

For our purposes, it is not strictly necessary to compute volumes because the end result is not a volume, but rather a ratio of volumes. Exploiting this reduces the amount of computation that we have to do. If we can sample uniformly from the polytope in the denominator, we can count the number of samples that occur in the numerator to iteratively approximate the ratio of the volumes of the polytopes. To sample from a polytope, we borrow from techniques for approximating the volume of

polytopes [124,125], which use Markov Chain Monte Carlo (MCMC) [126] sampling algorithms to produce a Markov chain whose limiting distribution is equal to a given distribution.

Definition 3 (Sampling intersection volume affinity)**.** *Let $R(A)^n$ be an n-cardinality set of random points uniformly distributed in a polytope A. The sampling intersection volume ratio of polytopes A and B given n samples is*

$$i^n_{\mathbb{R}}(A, B) = \frac{|\{\, x \in R(A \cup B)^n \mid x \in A \cap B \,\}|}{n}$$

Definition 4 (Sampling added hull volume affinity)**.** *Given $R(A)^n$ as above, the sampling hull volume ratio of polytopes A and B given n samples is*

$$h^n_{\mathbb{R}}(A, B) = \frac{|\{\, x \in R(\text{hull}(A, B))^n \mid x \in A \cup B \,\}|}{n}$$

These definitions give iterative approximations of the affinity functions that become closer to the actual function as the number of samples increases. In the limit they compute the precise volume ratios given in Table 4.1.

The complexity of MCMC sampling is polynomial in the dimension of the polytope. Generating each sample is polynomial, and typically a polynomial number of samples is sufficient to get decent coverage of the polytope. However, the complexity of the hull operation is potentially exponential in the dimension of the polytope. Therefore the dominating factor in the complexity of the sampling hull volume affinity is the hull operation. The sampling intersection volume affinity is attractive because it does not incur this exponential cost. However, it does require uniform sampling from a union of two convex polytopes, which basic MCMC

sampling does not support. We get around this with the following modification:

$$i^n_{\mathbb{R}}(A,B) = \frac{\left|\left\{\, x \in R(A)^{n/2} \cup R(B)^{n/2} \;\middle|\; x \in A \cap B \,\right\}\right|}{n}$$

This only requires sampling from convex polytopes and is thus polynomial time, but results in increased sample density in the smaller polytope and in the intersecting region.

4.4.3 Segment-Sample Volume-Ratio-Based Affinity

To avoid the complexity of the hull operation used in the sampling hull volume affinity, we also define a segment-sample-based affinity. This affinity is inspired by the definition of convex hull, where every point on every line segment between points in the two polytopes is included in the hull.

Definition 5 (Segment-sample volume-ratio-based affinity)**.** *Let* $S(A,B)^n = R(A)^n \times R(B)^n\big|_n$ *where* $R(A)^n$ *is as given above and* $\cdot\big|_n$ *randomly picks* n *elements of the set. Let* $\ell((x,y),A)$ *be the length of the line segment between* x *and* y *contained within the polytope* A*. Define* $|x-y|$ *to be the distance between* x *and* y*. The segment-sample volume-ratio-based affinity is*

$$s^n_{\mathbb{R}}(A,B) = \frac{\sum_{\bar{s}\in\bar{S}} \ell(\bar{s},A) + \ell(\bar{s},B) - \ell(\bar{s},A\cap B)}{\sum_{(x,y)\in\bar{S}} |x-y|} \text{ where } \bar{S} = S(A,B)^n.$$

This affinity's main interesting property is that it approximates the hull without actually computing the hull. As a result it has a polynomial time bound as opposed to an exponential time bound like other hull-based techniques. Unfortunately, this approximation is poor as the sampling is not uniform. Sampling end

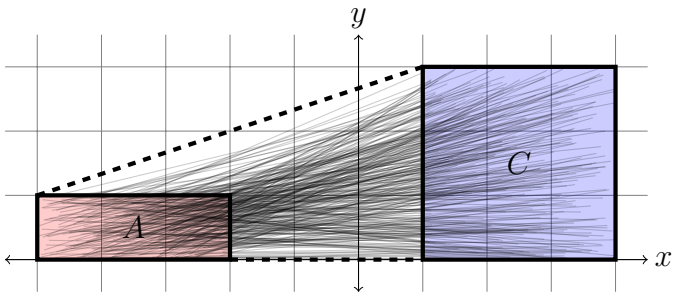


Figure 4.3: The sampled segments approximate the hull of A and C non-uniformly. Note that the upper portion is underrepresented.

points uniformly from two polytopes individually does not yield a uniform sampling of segments between those polytopes. Because the segments on average end in the middle of each polytope, the portion of the polytopes that are farther away from each other may be underrepresented in the calculation. We include this heuristic here because we believe that it is an interesting approach despite its shortcomings.

Example 3 (Segment-sample volume-ratio-based affinity)**.** *If we sample 300 segments, we get a picture like the one shown in Figure 4.3. The segments in this figure do not cover the topmost part of hull of A and C (shown by dashed lines), but instead repeatedly cover the center of the hull. However, these segments can be computed without computing the hull itself, which means that hulling is not necessary to reason about the volume introduced by hulling.*

4.4.4 Inflating Polytopes

With the volume-ratio-based affinities, there is the problem of abutment. When a conditional branch is interpreted, this splits the abstract state into two

separate abstract states that may be re-merged with a disjunction. Identifying when these branches have come back together is important for reducing the number of disjuncts. Unfortunately, there are cases where an integer gap may be introduced, as shown in Figure 4.2c. In this case, if the two abutting polytopes have a low total volume, the volume of the gap may outweigh the volume of the polytopes in the computation of the $h_{\mathbb{R}}$ affinity, and the two polytopes will be given a low score. Regardless of an integer gap, two abutting (but not intersecting) polytopes will be assigned a $i_{\mathbb{R}}$ affinity score of zero because their intersection volume is zero.

To avoid these issues, we use an inflation technique, which takes every face of the polytope and pushes it out by some amount. For example, in Figure 4.2c, inflating by one will cause the two polytopes to have an intersection of width one. Now the $h_{\mathbb{R}}$ affinity score will be one, which is the same as $h_{\mathbb{Z}}$, and $i_{\mathbb{R}}$ will be nonzero. An inflation of 0.5 is sufficient to bridge the integer gap, but larger inflation values may be beneficial. For instance, in the case of intersection volume, a larger inflation can boost the affinity of nearby (but not intersecting) polytopes without boosting the affinity of far apart polytopes. This naturally biases closer polytopes to be merged.

4.5 Disjunctive Abstract Domain

A concrete state is a point in d-dimensional space $\mathbb{Z}^d$. Convex polytope abstract states $q, r \in D^{\#}$ are instances of an abstract domain. The concretization of an abstract state $\gamma(q)$ is a set of concrete states. An abstract domain is a lat-

tice ordered by inclusion $\sqsubseteq$ that defines least upper bound $\sqcup$. An abstract domain defines monotone transfer functions f that map abstract states to other abstract states. Abstract domains also define a widening operator ∇ that predicts possible post-fixpoints and guarantees termination of the analysis.

We employ a typical disjunctive abstract domain. Disjunctive abstract states $Q = (q_1, \ldots, q_k), R = (r_1, \ldots, r_k) \in D^{\#^k}$ are k-element vectors of underlying convex numeric abstract states. The concretization is given as function of the underlying domain's concretization: $\gamma(Q) = \bigcup_{i \in [1,k]} \gamma(q_i)$ for $q_i \in Q$. Figure 4.4c shows the basic domain operations including join, a naive widening algorithm (for simplicity, not [127]), transfer function, and inclusion.

Following [112] and [128], we define disjunctive abstract domain operations using a selection function σ. The selection function shown in Figure 4.4b determines which among a set of abstract states is most similar to another abstract state. To do this it makes three comparisons. The first two check if the parameter q is contained in any of the $r_i \in R$ or if any of the $r_i \in R$ are contained in q. If so, the least index is chosen. This takes care of initialization because $\bot$ is trivially contained in any q. The last comparison checks if some affinity score a indicates that the two abstract states have a similarity higher than some threshold Θ. The threshold Θ is a parameter to the analysis. If all three comparisons fail, the index containing the most similar abstract state will be selected. The threshold check is important to ensure that similar, but not contained, abstract states do not fill up all k positions first and then force dissimilar abstract states to choose the best of several poor

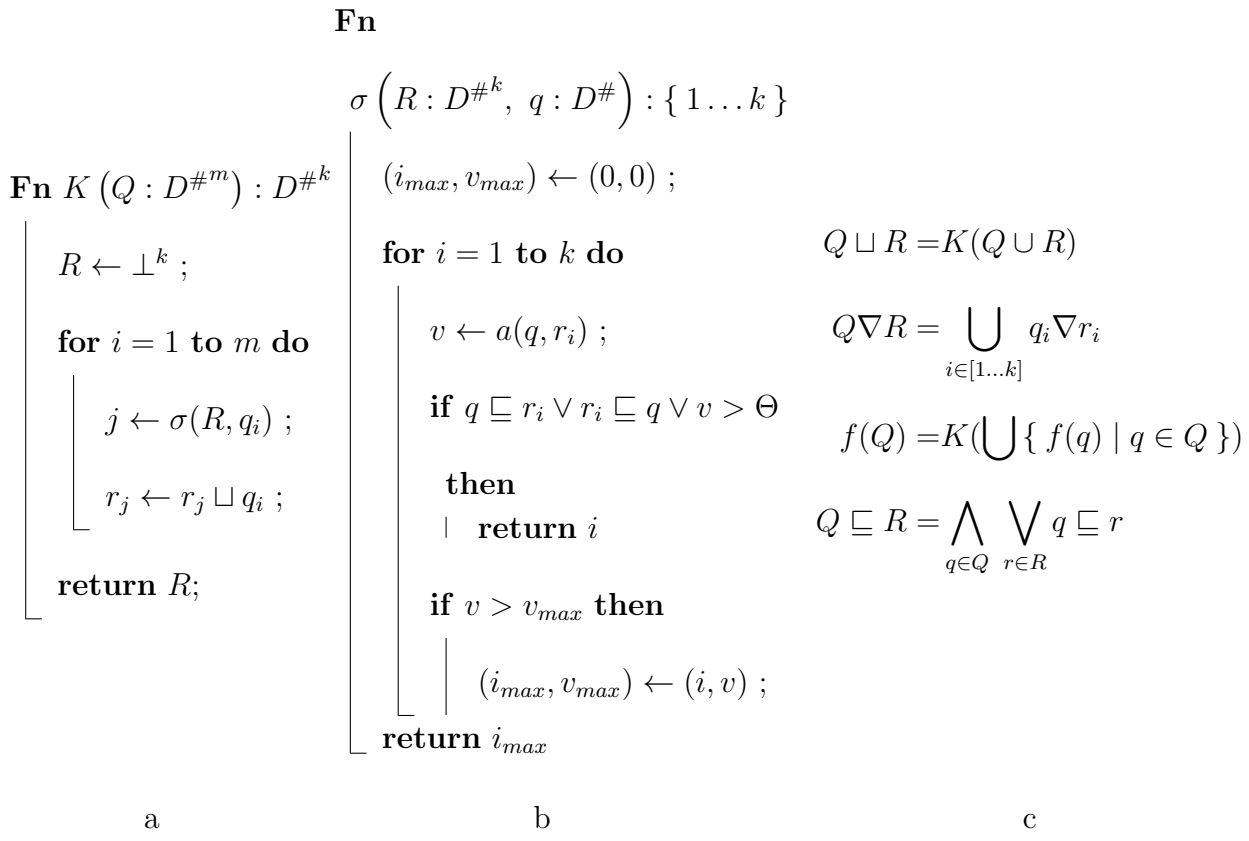


Figure 4.4: (a) Compaction function K and (b) corresponding selection function σ, where a is the affinity function. These are responsible for reducing the number of disjuncts in an abstract state down to k. (c) Domain operations join, widening, transfer function, and inclusion defined using K.

choices.

The σ function is then used by a compaction function K, which is shown in Figure 4.4a, to reduce an overly large set of disjuncts down to a smaller set. This is necessary to ensure termination of abstract interpretation by preventing the number of disjunctions from growing indefinitely. The compaction function works by iteratively inserting elements from Q into a result disjunction R according to the selection function σ. All of the abstract domain operations are defined using this

compaction function. They are implemented in the obvious way for a disjunctive abstract domain, and are compacted if too many disjuncts are produced by an operation. The soundness of this domain follows from definitions in prior work on disjunctive domains [112].

Example 4 (Compacting a disjunction)**.** *Consider the example shown in Figure 4.1b. There are three disjuncts A, B, and C, but we wish to compact that to $k = 2$ disjuncts with a threshold $\Theta = 0.8$ and an affinity score $a = h_{\mathbb{Z}}$. To begin, A is placed into r_1 because $\bot \sqsubseteq A$. Next, B is placed into r_2 because $\bot \sqsubseteq B$ and the affinity score assigned to A and B is $0.75 < 0.8$. Finally, C is merged into r_2 because the affinity score of r_2 and C is $1.0 > 0.8$. This is significantly higher than the affinity score of r_1 and C, which is approximately 0.77.*

4.6 Implementation

We implemented a disjunctive abstract domain in the CRAB C++ abstract interpretation framework [11], which builds upon Clang and LLVM version 3.8.0. C and C++ programs are compiled into LLVM IR and then optimized with a set of optimizations targeted at static analysis, such as pointer to array conversion [129]. The resulting LLVM IR files are then converted into a CRAB-specific intermediate representation for analysis with a selectable domain. This implementation is wholly separate from the previously described Java bytecode analyzer.

The disjunctive abstract domain is parameterized by the maximum number of disjuncts k, the similarity threshold Θ, and the choice of affinity scoring function a.

The underlying numeric abstraction is the NewPolka abstraction from the APRON abstract domain library [130]. NewPolka is convenient because it provides fairly low-level access to the constraint matrix and separates equality constraints from inequality constraints. To circumvent problems with infinite volume polytopes, we impose reasonable machine integer bounds. All variables are restricted to be in the range -2^{63} to $2^{64}-1$ to cover both signed and unsigned machine integers.

The null affinity scoring function 0 is trivially implemented: new polytopes are merged with either an existing polytope that wholly subsumes the new one, or if no such polytope is found the new polytope is added to the end. If there is no more room, the new polytope is merged with the last element in the disjunct. We also implemented an affinity measure that counts the number of common hyperplanes, c, as described in [113]. The $i_{\mathbb{Z}}$ and $h_{\mathbb{Z}}$ affinity functions are implemented as described in Section 4.4 using the Barvinok library [10] to implement integer counting within polytopes. The $i_{\mathbb{R}}$, $h_{\mathbb{R}}$ and $s_{\mathbb{R}}$ affinity scoring functions are implemented as described in Section 4.4 using our own implementation of polytope sampling (described in the next section). The $i_{\mathbb{R}}$, $h_{\mathbb{R}}$ and $s_{\mathbb{R}}$ scoring functions are additionally parameterized by the number of samples n. We scale the number of samples taken linearly with the number of dimensions in the polytope to ensure better coverage.

4.6.1 Random Sampling Within Polytopes

To implement the $R(A)^n$ operation we use a Markov Chain Monte Carlo (MCMC) technique called hit-and-run sampling [131,132], which performs a random

walk to generate points within a polytope. We use hit-and-run sampling because of its relative ease of implementation. Note that hit-and-run sampling only guarantees uniformity in the limit, so our implementation, which uses a limited number of samples, does not provide completely uniform random sampling.

One challenge with hit-and-run sampling (or any technique that randomly explores the interior of a polytope) is how to handle zero-volume polytopes, which occur often in abstract interpretation. Zero-volume polytopes inhibit random walks because the probability of selecting a valid direction in which to step is zero. To get around this, we do a dimension reduction that converts a zero-volume polytope to a non-zero-volume polytope of lower dimension [133, 134]. The lower-dimension polytope can then be sampled and each point mapped back to a point in the original polytope. These mapped points can then be used in one of the affinity scoring algorithms.

We also encounter difficulties with the representation of coefficients in the constraint matrices. For performance reasons it is desirable to use floating-point numbers in the constraint matrix. However, because the dynamic range of coefficients is very large, floating-point precision is insufficient and during sampling, rounding error may cause the invariant of the hit-and-run algorithm (that the current point is always inside the polytope) to be violated. To get around this, we represent coefficients using rational numbers. This gives us the precision we need, but adds significant overhead and makes it more difficult to do certain operations required by the hit-and-run algorithm, such as generating random points on a line segment.

We solve the problem of generating random points by introducing a new parameter m, which fixes the number of points that we can choose during any iteration of hit-and-run. To generate a random point on a line segment, we first split the segment into m sub-segments, and then choose an endpoint of a randomly selected sub-segment. Note that for a fixed number of samples, this limits the granularity of our samples. To get around some of the performance problems caused by using rational values we introduce another parameter, b, which is the batch size. The batch size determines how many points to sample from a segment once a hit-and-run direction has been chosen. This reduces the total number of directions sampled, and thus decreases the uniformity in exchange for increased performance.

4.7 Evaluation

In this section we evaluate the various affinity scores detailed in this chapter. This evaluation attempts to answer the following research questions.

- **RQ1**: Does merging the most similar polytopes increase analysis precision?
- **RQ2**: Does sampling provide better performance characteristics than exact computation?
- **RQ3**: Is exact computation efficient enough for large-scale analysis?
- **RQ4**: Is sampling efficient enough for large-scale analysis?

Table 4.2: Descriptions of the different affinity scores considered.

Affinity score	Description
0	Null affinity
c	Common hyperplanes [113]
$i_{\mathbb{Z}}$	Integer intersection volume
$h_{\mathbb{Z}}$	Integer added hull volume
$i_{\mathbb{R}}$	Sampling intersection volume
$h_{\mathbb{R}}$	Sampling added hull volume
$s_{\mathbb{R}}$	Segment-sample volume ratio

Table 4.3: WCET and SV-COMP benchmark sets used for evaluation. Lines of code counted with `cloc`.

Dataset	LOC
`wcet`	907
`loops`	2866
`ssh`	60463
`ntdrivers`	39173
`busybox-1.22.0`	58997
`loop-invgen`	441
`loop-acceleration`	637
`loop-industry-pattern`	3114
`array-industry-pattern`	551
`array-examples`	2941
Total	170090

4.7.1 Experimental Setup

To answer these research questions, we evaluate our implementation of the affinity scores listed in Table 4.2 on the SV-COMP [135] and WCET [136] benchmark suites. Specifically, we used the subset of programs from SV-COMP described in Table 4.3. We chose these benchmarks because they focus on numeric properties (e.g. `loops`) and represent interesting and significant programs (e.g. `busybox`). In total, we analyzed 170,090 lines of C code.

The benchmarks were executed on a 36-core, 72-thread Intel Xeon E5-2699 system with 512GB of RAM. We evaluated 72 benchmarks at a time and ran each benchmark five times to get an average for performance. Each benchmark was allowed up to 60 minutes of run time before being declared a time out.

We fixed the parameters in the following way based on a handful of small examples before evaluating on the full benchmark suite. The number of disjuncts k was limited to 3. The number of samples per dimension parameter n was set to 10. The number of segments parameter m was set to 1024. The batch size parameter b was set to 4. The threshold parameter Θ was set to 0.4. The inflation parameter was set to 0.5 for $h_{\mathbb{R}}$ and $s_{\mathbb{R}}$, 1.0 for $i_{\mathbb{R}}$, and 0 for $h_{\mathbb{Z}}$ and $i_{\mathbb{Z}}$. Recall that m and b are parameters used by our sampling implementation as described in Section 4.6. The evaluation proceeds with these settings.

To evaluate precision, we compared the invariants inferred for each program point across all of the different analyses. For each pair of analyses M, N, we queried the number of program points where $M \sqsubseteq N$ and $M = N$. This comparison gives us a fine grained measurement of the relative precision of different analyzers. Instead of asking, for example, how many array bounds checks or other assertions were proven (as we did in earlier work), we ask how about precision at every point in the program. An increase in precision would be valuable to any downstream client that sought to prove some numeric property of the program. We used the Yices SMT solver [137] to answer these queries. This query time is not counted as part of analysis time. Some M, N might be incomparable, and those are not represented in the table. One choice we have made in this experimental measurement is to not

Table 4.4: Ratio of program points where M is more precise than N. The upper diagonal is augmented with the percentage of when $M = N$.

		M											
		0	c		$i_{\mathbb{Z}}$		$h_{\mathbb{Z}}$		$i_{\mathbb{R}}$		$h_{\mathbb{R}}$		$s_{\mathbb{R}}$	
		$\sqsubset$	$\sqsubset$	$=$	$\sqsubset$	$=$	$\sqsubset$	$=$	$\sqsubset$	$=$	$\sqsubset$	$=$	$\sqsubset$	$=$
N	0	-	.05	.64	.25	.42	.25	.42	.07	.79	.20	.43	.19	.45
	c	.18	-	-	.31	.36	.31	.36	.19	.58	.23	.42	.24	.43
	$i_{\mathbb{Z}}$	.08	.03	-	-	-	0	1	.07	.45	.05	.33	.04	.42
	$h_{\mathbb{Z}}$	.08	.03	-	0	-	-	-	.07	.45	.05	.33	.04	.42
	$i_{\mathbb{R}}$	.02	.05	-	.23	-	.23	-	-	-	.18	.43	.18	.49
	$h_{\mathbb{R}}$	.19	.14	-	.29	-	.29	-	.22	-	-	-	.12	.64
	$s_{\mathbb{R}}$	.18	.13	-	.26	-	.26	-	.19	-	.08	-	-	-

determine if either M or N are sufficient to prove a property about the program, but to instead compare the relative precision between the two invariants when they can be related. This choice was made due to the impact of improving precision early in an analysis and due to the relatively few properties to prove compared to the number of program points.

4.7.2 Results

The precision results are presented in Table 4.4. The performance results are presented in Table 4.5 and shown graphically in Figure 4.5. We use this information to address the research questions.

RQ1

Does merging the most similar polytopes increase analysis precision? Table 4.4 shows that on average, yes. Both of the precise counting affinities produce more

Table 4.5: Aggregate performance of different analyzer configurations across all programs. Each program was analyzed 5 times. We took the mean of 5 runs and report on that mean when aggregating across all programs. Times reported are in seconds.

Analyzer	Mean	Min	Max	Median
0	5.375	0.242	79.291	0.866
c	7.883	0.254	240.350	0.786
$i_{\mathbb{Z}}$	19.058	0.250	806.325	0.581
$h_{\mathbb{Z}}$	33.202	0.250	1493.689	0.605
$i_{\mathbb{R}}$	29.522	0.241	1204.348	4.013
$h_{\mathbb{R}}$	56.186	0.243	932.781	7.236
$s_{\mathbb{R}}$	86.074	0.254	1928.718	8.367

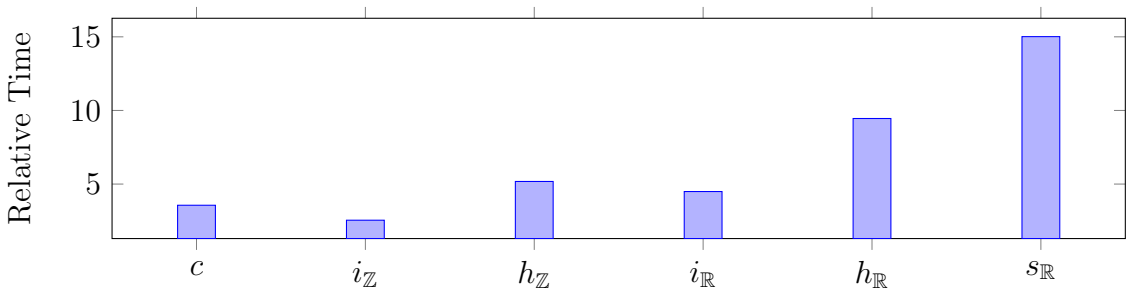


Figure 4.5: Run-time performance of analyzer configurations relative to null affinity.

precise results 25% of the time, whereas the null affinity is more precise only 8% of the time. The sampling-based techniques fare slightly worse against the null affinity, scoring 7%, 20%, and 19% better, whereas the null affinity performs better 2%, 19%, and 18% of the time. However, the sampling-based techniques perform better against the c affinity measure in all cases. This suggests that when the volumetric comparison is precise (i.e. either $i_{\mathbb{Z}}$ or $h_{\mathbb{Z}}$), there is a significant benefit over the basic strategy. This also suggests that either the number of samples or the specific samples that we chose were insufficient to identify the truly related polytopes.

RQ2

Does sampling provide better performance characteristics than exact computation? No. With the parameters that we have chosen, the performance is roughly comparable with the precise counting techniques generally being faster. Table 4.5 shows that the sampling techniques on average take twice as long as the precise counting techniques. However, for the added hull volume affinity scores ($h_{\mathbb{Z}}$ and $h_{\mathbb{R}}$), the maximum run time for the sampling technique is significantly better than the maximum run time for the precise counting technique. This suggests that the asymptotic complexity advantage of sampling pays off when the problem gets particularly difficult for precise counting. Even so, with the implementation we have developed and the parameters that we have chosen, the sampling techniques are generally not worth using.

RQ3

Is exact computation efficient enough for large-scale analysis? Yes. Table 4.5 shows that the exact computation techniques have non-trivial overhead over the null affinity case. However, depending on the situation, $i_{\mathbb{Z}}$ may provide a fair trade-off: a 4x increase in analysis time in exchange for invariants that are stronger 25% of the time. $h_{\mathbb{Z}}$ is less favorable: a 6x increase in analysis time for exactly the same 25% improvement in invariant strength.

RQ4

Is sampling efficient enough for large-scale analysis? Yes, though in its current state it is probably not worth using. Like $i_{\mathbb{Z}}$ and $h_{\mathbb{Z}}$, $i_{\mathbb{R}}$ and $h_{\mathbb{R}}$ are more expensive than the null affinity. In general a 6x overhead of $i_{\mathbb{R}}$ is not necessarily too expensive, although it depends on the situation. The 10x and 17x overheads of $h_{\mathbb{R}}$ and $s_{\mathbb{R}}$ are probably too expensive, especially as they seem to provide no precision benefit over the precise methods.

4.7.3 Limitations and Discussion

There are a number of limitations to our implementation, experimentation, and analysis. The most significant is the choice of parameters for the analysis. Ideally we would have chosen parameters for the sampling-based approaches on a large set of benchmarks. This limitation shows because, in the limit, the sampling should be similar to the exact counting methods. Due to the fact that the results are quite different, this suggests that we are not yet approaching that limit. We should probably increase the number of samples, increase the number of segments, or decrease the batch size to improve this result.

The choice of Θ is somewhat arbitrary. While the exact counting techniques do show a benefit with a Θ of 0.4, it is not clear that this is an optimum value. It is also not clear whether the sampling approaches should have different thresholds than the exact computation. It seems like that should be unnecessary, but we have not explored that space.

The results are somewhat unfairly biased against the sampling technique. The library for exact computation has been under development in some form for around 20 years. As a result it employs careful memory management for all of its computations to ensure that no extra memory is being allocated or freed. Furthermore it enjoys an optimized matrix library that has been custom built for this application and caching of intermediate results so that it can both avoid re-computation and re-allocation. In spot checks we have observed that the sampler is spending nearly 50% of its time doing memory allocation or freeing. If the sampler could manage memory more efficiently it may be possible to get it into the same realm of performance as the exact counting method.

We are currently using a fairly naive coordinate direction hit-and-run sampler. The reason for this was to increase the number of samples we could collect per second. It might be a fair trade to use a more advanced algorithm that is slower if it yielded more uniformly distributed samples. In particular, the coordinate direction hit-and-run sampler can get stuck in corners if a polytope is long and narrow and there is no coordinate direction that covers a large percentage of the space.

Finally, these results are dependent on the widening strategy. A poor choice when performing widening could easily cause one disjunct to go to top or close to top. Future disjuncts would be trivially merged with that particular disjunct resulting in an overall loss of precision. It is unclear how to account for this when analyzing results. While a loss of precision during widening is acceptable, it would be interesting to know how an ideal strategy would compare. Unfortunately, this is not possible.

4.8 Related Work

Disjunctions have been a widely studied topic. In abstract interpretation they were introduced with powerset domains in [114]. Jeannet [138] explored partitioning schemes for disjunctive invariants. More generally, the theory of disjunctive invariants is explored in [115]. This, along with [112], develops a relationship between disjunctive invariants and control flow path refinement. In effect, refining control flow such that multiple paths are presented for a single syntactic path is equivalent to a disjunctive analysis. This leads a significant quantity of work on control flow refinement [139–143], which can be viewed as applications of disjunctive techniques.

In [128] a theorem is given that a best disjunction merge policy can be statically computed. This theorem assumes that widening is not required and thus is not generally applicable to abstract interpretation. Furthermore, it is not obvious how to statically compute a merge policy in the context of a general abstract interpreter. [121] claims to do this but instead implements the null affinity score.

Model checking procedures [144, 145] typically produce disjunctive invariants. The way they do this is different in its operation than what we present. They first analyze programs without any disjunctions and then introduce them by learning where a coarse abstraction has caused a property to not be proven. While this approach is quite effective, it does not work for unguided analysis such as program understanding and it may not scale as well as non-refinement-based analyses such as Astrée [146].

We are most related to work that performs forward disjunctive analyses using

numeric domains and no refinement. In [112], the authors use a similar formulation of the abstraction. The key difference is in the choice of merge heuristic. The choice in [112] is to merge according to a simplified Hausdorff distance, which is shown by [113] to be less desirable than other heuristics. In [113], the authors use a syntactic property of polytopes to decide merging. This technique counts the number of hyperplanes in common between an input polytope and the result of a join. Another possible merge heuristic is the similarity of Boolean variables. In [118], a binary decision diagram is used to determine which numeric domains should be merged and which should not.

Our merging heuristics are based on volume and counting computations for polytopes. Barvinok develops the core theory [119] for counting procedures. Approximate volume computations based on sampling are alternatively used [124,125]. The idea of using the Barvinok algorithm came from [147].

4.9 Conclusion

In this chapter we have shown a number of new affinity scoring algorithms for determining which disjuncts should be merged in a disjunctive abstraction. The new affinity scoring algorithms are all based on points within the polytopes. Those points are either sampled or counted in order to compute proxies for polytope volume. We demonstrated that these techniques work by analyzing a large selection of benchmark programs. In the future we would like to further optimize sampling to make it more performant to determine if the difference in complexity yields tangible differences

in performance. We would also like to explore adaptations of the segment sampling approach to find something that has some degree of uniformity.

Chapter 5: Evaluating Fuzz Testing

5.1 Backround and overview

In Chapter 2 and 3, we considered the effect of language on security, and in Chapter 3, we consider eliminating security bugs by design. In Chapter 3 and 4, we consider static analyzers to identify security problems. These automated techniques could scale further than manual techniques used by contestants in Chapter 2. However, there are other automated techniques besides static analyzers, and we should consider them as well.

A *fuzz tester* (or *fuzzer*) is a tool that iteratively and randomly generates inputs with which it tests a target program. Despite appearing "naive" when compared to more sophisticated tools involving SMT solvers, symbolic execution, and static analysis, fuzzers are surprisingly effective. For example, the popular fuzzer AFL has been used to find hundreds of bugs in popular programs [148]. Comparing AFL head-to-head with the symbolic executor *angr*, AFL found 76% more bugs (68 vs. 16) in the same corpus over a 24-hour period [149]. The success of fuzzers has made them a popular topic of research.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When

a researcher develops a new fuzzer algorithm (call it A), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling *baseline* fuzzer B to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when A and B are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *seed file* (or files) to start fuzzing with, and the *timeout* (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random nature of fuzzing: Each fuzzing run on a target program may produce different results than the last due to the use of randomness. As such, an evaluation should measure *sufficiently many trials* to sample the overall distribution that represents the fuzzer's performance, using a *statistical test* [150] to determine that A's measured improvement over B is real, rather than due to chance.

Failure to perform one of these steps, or failing to follow recommended practice when carrying it out, could lead to misleading or incorrect conclusions. Such conclusions waste time for practitioners, who might profit more from using alternative methods or configurations. They also waste the time of researchers, who make overly strong assumptions based on an arbitrary tuning of evaluation parameters.

We examined 32 recently published papers on fuzz testing (see Table 5.1) located by perusing top-conference proceedings and other quality venues, and studied their experimental evaluations. We found that no fuzz testing evaluation carries out all of the above steps properly (though some get close). This is bad news in theory, and after carrying out more than 50000 CPU hours of experiments, we believe it is bad news in practice, too. Using AFLFast [151] (as A) and AFL (as baseline B), we carried out a variety of tests of their performance. We chose AFLFast as it was a recent advance over the state of the art; its code was publicly available; and we were confident in our ability to rerun the experiments described by the authors in their own evaluation and expand these experiments by varying parameters that the original experimenters did not. This choice was also driven by the importance of AFL in the literature: 14 out of 32 papers we examined used AFL as a baseline in their evaluation. We targeted three binutils programs (*nm*, *objdump*, and *cxxfilt*) and two image processing programs (*gif2png* and *FFmpeg*) used in prior fuzzing evaluations [152–156]. We found that experiments that deviate from the above recipe could easily lead one to draw incorrect conclusions, for these reasons:

Fuzzing performance under the same configuration can vary substantially from run to run. Thus, comparing single runs, as nearly $\frac{2}{3}$ of the examined papers seem to, does not give a full picture. For example, on *nm*, one AFL run found just over 1200 crashing inputs while one AFLFast run found around 800. Yet, comparing the median of 30 runs tells a different story: 400 crashes for AFL and closer to 1250 for AFLFast. Comparing averages is still not enough, though: We found that in some cases, via a statistical test, that an apparent difference in performance was

not statistically significant.

Fuzzing performance can vary over the course of a run. This means that short timeouts (of less than 5 or 6 hours, as used by 11 papers) may paint a misleading picture. For example, when using the empty seed, AFL found no crashes in *gif2png* after 13 hours, while AFLFast had found nearly 40. But after 24 hours AFL had found 39 and AFLFast had found 52. When using a non-empty seed set, on *nm* AFL outperformed AFLFast at 6 hours, with statistical significance, but after 24 hours the trend reversed.

We similarly found *substantial performance variations based on the seeds used*; e.g., with an empty seed AFLFast found more than 1000 crashes in *nm* but with a small non-empty seed it found only 24, which was statistically indistinguishable from the 23 found by AFL. And yet, most papers treated the choice of seed casually, apparently assuming that any seed would work equally well, without providing particulars.

Turning to measures of performance, *14 out of 32 papers we examined used code coverage to assess fuzzing effectiveness.* Covering more code intuitively correlates with finding more bugs [157, 158] and so would seem to be worth doing. But the correlation may be weak [159], so directly measuring the number of bugs found is preferred. *Yet only about* $\frac{1}{4}$ *of papers used this direct measure.* Most papers instead counted the number of *crashing inputs* found, and then applied a heuristic procedure in an attempt to de-duplicate inputs that trigger the same bug (retaining a "unique" input for that bug). The two most popular heuristics were AFL's coverage profile (used by 7 papers) and (fuzzy) stack hashes [160] (used by 7 papers). Unfortunately,

there is reason to believe these *de-duplication heuristics are ineffective*. We explore this further in Chapter 6.

Overall, *fuzzing performance may vary with the target program*, so it is important to evaluate on a diverse, representative benchmark suite. In our experiments, we found that AFLFast performed generally better than AFL on *binutils* programs (basically matching its originally published result, when using an empty seed), but did not provide a statistically significant advantage on the image processing programs. Had these programs been included in its evaluation, readers might have drawn more nuanced conclusions about its advantages. In general, *few papers use a common, diverse benchmark suite*; about 6 used CGC or LAVA-M, and 2 discussed the methodology in collecting real-world programs, while the rest used a few handpicked programs, with little overlap in these choices among papers. The median number of real-world programs used in the evaluation was 7, and the most commonly used programs (*binutils*) were shared by only four papers (and no overlap when versions are considered). As a result, individual evaluations may present misleading conclusions internally, and results are hard to compare across papers.

Our study (outlined in Section 5.3) suggests that meaningful scientific progress on fuzzing requires that claims of algorithmic improvements be supported by more solid evidence. *Every evaluation in the 32 papers we looked at lacks some important aspect in this regard.* In this chapter we propose some clear guidelines to which future papers' evaluations should adhere. In particular, researchers should perform multiple trials and use statistical tests (Section 5.4); they should evaluate different seeds (Section 5.5), and should consider longer ($\geq$ 24 hour vs. 5 hour) timeouts

(Section 5.6). We defer a more in depth discussion of evaluating the performance of a fuzzer using ground truth, and the deficiencies of metrics and heuristics like "stack hashes" and "unique crashes" to Chapter 6(6). Finally, we argue for the establishment and adoption of a good fuzzing benchmark, and sketch what it might look like. The practice of hand selecting a few particular targets, and varying them from paper to paper, is problematic (Section 5.8). A well-designed and agreed-upon benchmark would address this problem. We also identify other problems that our results suggest are worth studying, including the establishment of better de-duplication heuristics (a topic of recent interest [161,162]), and the use of algorithmic ideas from related areas, such as SAT solving.

Attribution and acknowledgments

This chapter is adapted from a paper appearing at ACM CCS 2018 [12]. The paper authors were George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Mike Hicks. Mike and Andrew devised the initial experiments, refined by discussions with George. George conducted the initial experiments. Andrew and Benji analyzed the crashes identified to discover the root cause of the bugs. Andrew designed and implemented the experimental methodology to use different versions of the program to identify root cause patches. Andrew, Mike, and Shiyi analyzed prior experimental designs from the fuzzing literature.

5.2 Background

There are many different dynamic analyses that can be described as "fuzzing." A unifying feature of fuzzers is that they operate on, and produce, concrete inputs. Otherwise, fuzzers might be instantiated with many different design choices and many different parameter settings. In this section, we outline the basics of how fuzzers work, and then touch on the advances of 32 recently published papers which form the core of our study on fuzzing evaluations.

5.2.1 Fuzzing Procedure

Most modern fuzzers follow the procedure outlined in Figure 5.1. The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout.

Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the execution, for example, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White

Core fuzzing algorithm:

```
corpus ← initSeedCorpus()
queue ← initQueue(corpus)
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **initQueue**: Initialize the queue, potentially from a corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Figure 5.1: Fuzzing, in a nutshell

box" fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness.

Usually, the ultimate goal of a fuzzer is to generate an input that causes the program to crash. In some fuzzer configurations, *isDone* checks the queue to see if there have been any crashes, and if there have been, it breaks the loop. Other fuzzer configurations seek to collect as many different crashes as they can, and so will not stop after the first crash. For example, by default, `libfuzzer` [163] will stop when it discovers a crash, while AFL will continue and attempt to discover different crashes. Other types of observations are also desirable, such as longer running times that could indicate the presence of algorithmic complexity vulnerabilities [164]. In any of these cases, the output from the fuzzer is some concrete input(s) and configurations that can be used from outside of the fuzzer to reproduce the observation. This allows software developers to confirm, reproduce, and debug issues.

5.2.2 Recent Advances in Fuzzing

The effectiveness of fuzz testing has made it an active area of research. Performing a literature search we found 32 papers published between 2012 and 2018 that propose and study improvements to various parts of the core fuzzing algorithm; 25 out of 32 papers we examined were published since 2016. To find these papers, we started from 10 high-impact fuzzing papers published in top security venues.

Then we chased citations to and from these papers. As a sanity check, we also did a keyword search of titles and abstracts of the papers published since 2012. Finally, we judged the relevance based on target domain and proposed advance, filtering papers that did not fit.

Table 5.1 lists these papers in chronological order. Here we briefly summarize the topics of these papers, organized by the part of the fuzzing procedure they most prominently aim to improve. Ultimately, our interest is in how these papers *evaluate* their claimed improvements, as discussed more in the next section.

initSeedCorpus. Skyfire [165] and Orthrus [166] propose to improve the initial seed selection by running an up-front analysis on the program to bootstrap information both for creating the corpus and assisting the mutators. QuickFuzz [167, 168] allows seed generation through the use of grammars that specify the structure of valid, or interesting, inputs. DIFUZE performs an up-front static analysis to identify the structure of inputs to device drivers prior to fuzzing [169].

mutate. SYMFUZZ [154] uses a symbolic executor to determine the number of bits of a seed to *mutate*. Several other works change *mutate* to be aware of taint-level observations about the program behavior, specifically mutating inputs that are used by the program [153, 170–172]. Where other fuzzers use pre-defined data mutation strategies like bit flipping or rand replacement, MutaGen uses fragments of the program under test that parse or manipulate the input as mutators through dynamic slicing [173]. SDF uses properties of the seeds themselves to guide mutation [174]. Sometimes, a grammar is used to guide mutation [175, 176]. Chizpurfle's [177] mutator exploits knowledge of Java-level language constructs to assist in-process

fuzzing of Android system services.

eval. Driller [149] and MAYHEM [170] observe that some conditional guards in the program are difficult to satisfy via brute force guessing, and so (occasionally) invoke a symbolic executor during the *eval* phase to get past them. S2F also makes use of a symbolic executor during *eval* [156]. Other work focuses on increasing the speed of *eval* by making changes to the operating system [178] or using different low level primitives to observe the effect of executions [176, 179, 180]. T-Fuzz [181] will transform the program to remove checks on the input that prevent new code from being reached. MEDS [182] performs finer grained run time analysis to detect errors during fuzzing.

isInteresting. While most papers focus on the crashes, some work changes *observation* to consider different classes of program behavior as interesting, e.g., longer running time [164], or differential behavior [183]. Steelix [171] and Angora [172] instrument the program so that finer grained information about progress towards satisfying a condition is exposed through *observation*. Dowser and VUzzer [153, 184] uses a static analysis to assign different rewards to program points based on either a likely-hood estimation that traveling through that point will result in a vulnerability, or for reaching a deeper point in the CFG.

choose. Several works select the next input candidate based on whether it reaches particular areas of the program [151, 153, 185, 186]. Other work explores different algorithms for selecting candidate seeds [152, 155].

Table 5.1 *(following page)*: Summary of past fuzzing evaluation. Blank cell means that the paper's evaluation did not mention this item; - means it was not relevant; ? means the element was mentioned but with insufficient detail to be clear about it. **Benchmarks**: R means real-world programs, C means CGC data-set, L means LAVA-M benchmark, S means programs with manually injected bugs, G means Google fuzzer test suite. **Baseline**: A means AFL, B means BFF [189], L means libfuzzer [163], R means Radamsa [190], Z means Zzuf [191], O means other baseline used by no more than 1 paper. **Trials**: number of trials. **Variance**: C means confidence intervals. **Crash**: S means stack hash used to group related crashes during triage, O means other tools/methods used for triage, C means coverage profile used to distinguish crashes, G means crashes triaged according to ground truth, G* means manual efforts partially obtained ground truth for triaging. **Coverage**: L means line/instruction/basic-block coverage, M means method coverage, E means control-flow edge or branch coverage, O means other coverage information. **Seed**: R means randomly sampled seeds, M means manually constructed seeds, G means automatically generated seed, N means non-empty seed(s) but it was not clear if the seed corpus was valid, V means the paper assumes the existence of valid seed(s) but it was not clear how the seed corpus was obtained, E means empty seeds, / means different seeds were used in different programs, but only one kind of seeds in one program. **Timeout**: times reported in minutes (M), hours (H) and/or days (D).

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM [170]	R(29)				G	?	N	-
FuzzSim [152]	R(101)	B	100	C	S		R/M	10D
Dowser [184]	R(7)	O	?		O		N	8H
COVERSET [155]	R(10)	O			S, G*	?	R	12H
SYMFUZZ [154]	R(8)	A, B, Z			S		M	1H
MutaGen [173]	R(8)	R, Z			S	L	V	24H
SDF [174]	R(1)	Z, O			O		V	5D
Driller [149]	C(126)	A			G	L, E	N	24H
QuickFuzz-1 [167]	R(?)		10		?		G	-
AFLFast [151]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz [187]	R(5)	O			M	O	G, R	2H
[175]	R(2)	A, O				L, E	V	2H
AFLGo [186]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer [153]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz [164]	R(10)	O	100		-		N	
Steelix [171]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire [165]	R(4)	O			?	L, M	R, G	LONG
kAFL [179]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE [169]	R(7)	O			G*		G	5H
Orthrus [166]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle [177]	R(1)	O			G*		G	-
VDF [180]	R(18)				C	E	V	30D
QuickFuzz-2 [168]	R(?)	O	10		G*		G, M	
IMF [176]	R(1)	O			G*	O	G	24H
[188]	S(?)	O	5		G		G	24H
NEZHA [183]	R(6)	A, L, O			O		R	
[178]	G(10)	A, L					V	5M
S2F [156]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz [185]	R(9)	A	20	C		E	V/M	24H
Angora [172]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz [181]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS [182]	S(2), R(12)	O	10		C		N	6H

5.3 Overview and Experimental Setup

Our interest in this chapter is assessing the existing research practice of experimentally evaluating fuzz testing algorithms. As mentioned in the introduction, evaluating a fuzz testing algorithm A requires several steps: (a) choosing a baseline algorithm B against which to compare; (b) choosing a representative set of target programs to test; (c) choosing how to measure A's vs. B's performance, ideally as bugs found; (d) filling in algorithm parameters, such as how seed files are chosen and how long the algorithm should run; and (e) carrying out multiple runs for both A and B and statistically comparing their performance.

Research papers on fuzz testing differ substantially in how they carry out these steps. For each of the 32 papers introduced in Section 5.2.2, Table 5.1 indicates what **benchmark** programs were used for evaluation; the **baseline** fuzzer used for comparison; the number of **trials** carried out per configuration; whether **variance** in performance was considered; how **crash**ing inputs were mapped to bugs (if at all); whether code **coverage** was measured to judge performance; how **seed** files were chosen; and what **timeout** was used per trial (i.e., how long the fuzzer was allowed to run). Explanations for each cell in the table are given in the caption; a blank cell means that the paper's evaluation did not mention this item.

For example, the AFLFast [151] row in Table 5.1 shows that the AFLFast's evaluation used 6 real-world programs as benchmarks (column 2); used AFL as the baseline fuzzer (column 3); ran each experiment 8 times (column 4) without reporting any variance (column 5); measured and reported crashes, but also conducted

manual triage to obtain ground truth (column 6); did not measure code coverage (column 7); used an empty file as the lone input seed (column 8); and set 6 hours and 24 hours as timeouts for different experiments (column 9).

Which of these evaluations are "good" and which are not, in the sense that they obtain evidence that supports the claimed technical advance? In the following sections we assess evaluations both theoretically and empirically, carrying out experiments that demonstrate how poor choices can lead to misleading or incorrect conclusions about an algorithm's fitness. In some cases, we believe it is still an open question as to the "best" choice for an evaluation, but in other cases it is clear that a particular approach should be taken (or, at least, certain naive approaches should *not* be taken). Overall, we feel that *every existing evaluation is lacking in some important way.*

We conclude this section with a description of the setup for our own experiments.

Fuzzers

For our experiments we use AFL (with standard configuration parameters) 2.43b as our baseline B, and AFLFast [151] as our "advanced" algorithm A. We used the AFLFast version from July 2017 (cloned from Github) that was based on AFL version 2.43b. Note that these are more recent versions than those used in Böhme et al's original paper [151]. Some, but not all, ideas from the original AFLFast were incorporated into AFL by version 2.43b. This is not an issue for us

since our goal is not to reproduce AFLFast's results, but rather to use it as a representative "advanced" fuzzer for purposes of considering (in)validity of approaches to empirically evaluating fuzzers. (We note, also, that AFL served as the baseline for 14/32 papers we looked at, so using it in our experiments speaks directly to those evaluations that used it.) We chose it and AFL because they are open source, easy to build, and easily comparable. We also occasionally consider a configuration we call *AFLNaive*, which is AFL with coverage tracking turned off (using option `-n`), effectively turning AFL into a black box fuzzer.

Benchmark programs

We used the following benchmark programs in our experiments: *nm*, *objdump*, *cxxfilt* (all from *binutils-2.26*), *gif2png*, and *FFmpeg*. All of these programs were obtained from recent evaluations of fuzzing techniques. *FFmpeg-n0.5.10* was used in FuzzSim [152]. *binutils-2.26* was the subject of the AFLFast evaluation [151], and only the three programs listed above had discoverable bugs. *gif2png-2.5.8* was tested by VUzzer [153].[1] We do not claim that this is a complete benchmark suite; in fact, we think that a deriving a good benchmark suite is an open problem. We simply use these programs to demonstrate how testing on different targets might lead one to draw different conclusions.

[1] Different versions of FFmpeg and gif2png were assessed by other papers [154–156], and likewise for *binutils* [183, 185, 186].

Performance measure

For our experiments we measured the number of "unique" crashes a fuzzer can induce over some period of time, where uniqueness is determined by AFL's notion of coverage. In particular, two crashing inputs are considered the same if they have the same (edge) coverage profile. Though this measure is not uncommon, it has its problems; Chapter 6(6) discusses why, in detail.

Platform and configuration

Our experiments were conducted on three machines. Machines I and II are equipped with twelve 2.9GHz Intel Xenon CPUs (each with 2 logical cores) and 48GB RAM running Ubuntu 16.04. Machine III has twenty-four 2.4GHz CPUs and 110GB RAM running Red Hat Enterprise Linux Server 7.4. To account for possible variations between these systems, each benchmark program was always tested on the same machine, for all fuzzer combinations. Our testing script took advantage of all the CPUs on the system to run as many trials in parallel as possible. One testing subprocess was spawned per CPU and confined to it through CPU affinity. Every trial was allowed to run for 24 hours, and we generally measured at least 30 trials per configuration. We also considered a variety of seed files, including the empty file, randomly selected files of the right type, and manually-generated (but well-formed) files.

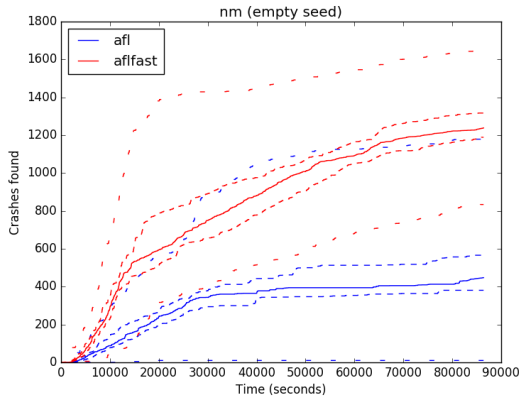


Figure 5.2: nm Crashes found over time (empty seed). Solid line is median; dashed lines are confidence intervals, and max/min. $p < 10^{-13}$

5.4 Statistically Sound Comparisons

All modern fuzzing algorithms fundamentally employ randomness when performing testing, most notably when performing mutations, but sometimes in other ways too. As such, it is not sufficient to simply run fuzzer A and baseline B once each and compare their performance. Rather, both A and B should be run for many trials, and differences in performance between them should be judged.

Perhaps surprisingly, Table 5.1 shows that most (18 out of 32) fuzzing papers we considered make no mention of the number of trials performed. Based on context clues, our interpretation is that they each did one trial. One possible justification is that the randomness "evens out;" i.e., if you run long enough, the random choices will converge and the fuzzer will find the same number of crashing inputs. It is clear from our experiments that this is not true—fuzzing performance can vary

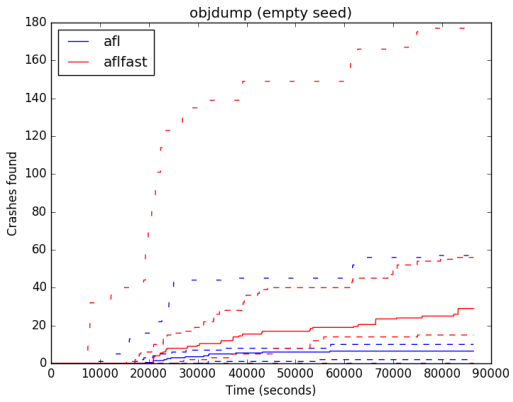


Figure 5.3: objdump Crashes found over time (empty seed). Solid line is median; dashed lines are confidence intervals, and max/min. $p < 0.001$

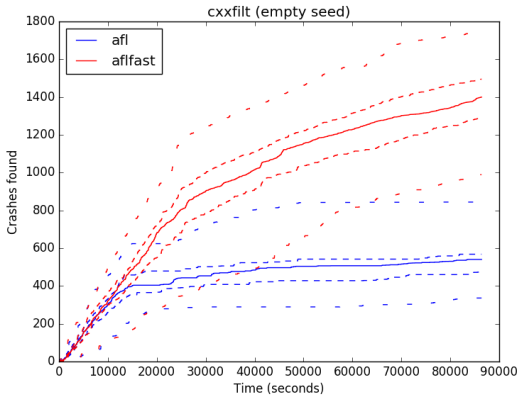


Figure 5.4: cxxfilt Crashes found over time (empty seed). Solid line is median; dashed lines are confidence intervals, and max/min. $p < 10^{-10}$

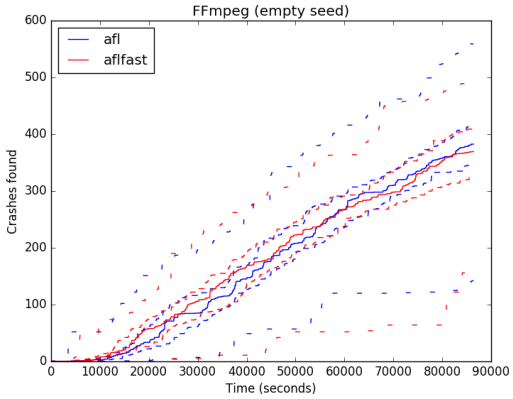


Figure 5.5: FFmpeg Crashes found over time (empty seed). Solid line is median; dashed lines are confidence intervals, and max/min. $p = 0.379$

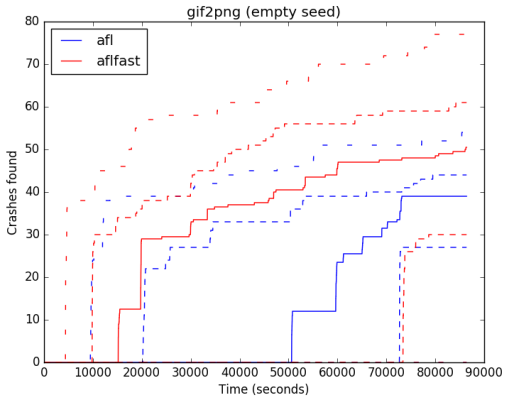


Figure 5.6: gif2png Crashes found over time (empty seed). Solid line is median; dashed lines are confidence intervals, and max/min. $p = 0.0676$

dramatically from run to run.

Consider the results presented in Figures 5.2, 5.3, 5.4, 5.5, and 5.6, which graphs the cumulative number of crashes (the Y axis) we found over time (the X axis) by AFL (blue), and AFLFast (red), each starting with an empty seed. In each plot, the solid line represents the median result from 30 runs while the dashed lines represent the maximum and minimum observed results, and the lower and upper bounds of 95% confidence intervals for a median [192]. (The outermost dashed lines are max/min, the inner ones are the CIs.)

It should be clear from the highly varying performance on these plots that considering only a single run could lead to the wrong conclusion. For example, suppose the single run on *FFmpeg* for AFL turned out to be its maximum, topping out at 550 crashes, while the single run on AFLFast turned out to be its minimum, topping out at only 150 crashes (Figure 5.5). Just comparing these two results, we might believe that AFLFast provides no advantage over AFL. Or we might have

observed AFLFast's maximum and AFL's minimum, and concluded the opposite.

Performing multiple trials and reporting averages is better, but not considering variance is also problematic. In Table 5.1, we can see that 11 out of the 14 papers that did consider multiple trials did not characterize the performance variance (they have a blank box in the **variance** column). Instead, each of them compared the "average" performance (we assume: arithmetic mean) of A and B when drawing conclusions, except for Dowser [184] that reported median, and two [167, 188] that did not mention how the "average" was calculated.

The problem is that with a high enough variance, a difference in averages may not be statistically significant. A solution is to use a *statistical test* [150]. Such a test indicates the likelihood that a difference in performance is real, rather than due to chance. Arcuri and Briand [193] suggest that for randomized testing algorithms (like fuzzers), one should use the Mann Whitney U-test to determine the stochastic ranking of A and B, i.e., whether the outcomes of the trials in A's data sample are more likely to be larger than outcomes in B's. Mann Whitney is *non parametric* in that it makes no assumption about the distribution of a randomized algorithm's performance; by contrast, the standard t-test assumes a normal distribution.

Returning to our experiments, we can see where simply comparing averages may yield wrong conclusions. For example, for *gif2png*, after 24 hours AFLFast finds a median of 51 crashes while for AFL it is 39 (a difference of 12). But performing the Mann Whitney test yields a p value of greater than 0.05, suggesting the difference may not be statistically significant, even on the evidence of thirty 24-hour trials. For *FFmpeg*, AFLFast's median is 369.5 crashes while AFL's is 382.5, also a difference

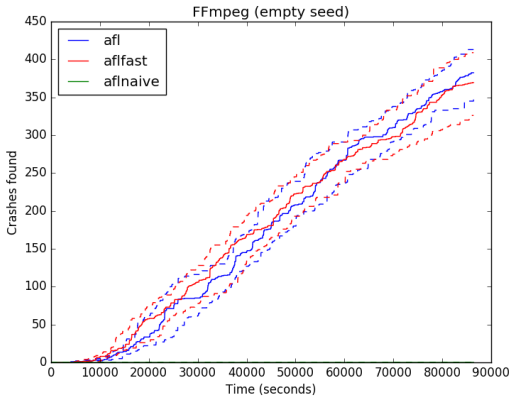


Figure 5.7: empty seed: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 = 0.379$,$p_2 < 10^{-15}$

of about 12 crashes, this time favoring AFL. Likewise, Mann Whitney deems the difference insignificant. On the other hand, for *nm* the advantage of AFLFast over AFL is extremely unlikely to occur by chance.

The three papers in Table 5.1 with "C" in the **variance** column come the closest to the best practice by at least presenting confidence intervals along with averages. But even here they stop short of statistically comparing the performance of their approach against the baseline; they leave it to the reader to visually judge this difference. This is helpful but not as conclusive as a (easy to perform) statistical test.

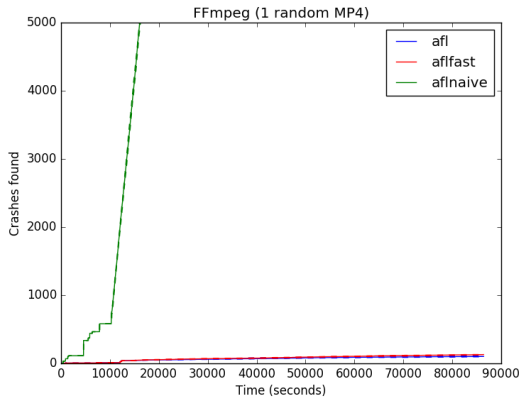


Figure 5.8: 1-made seed: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 = 0.048$,$p_2 < 10^{-11}$

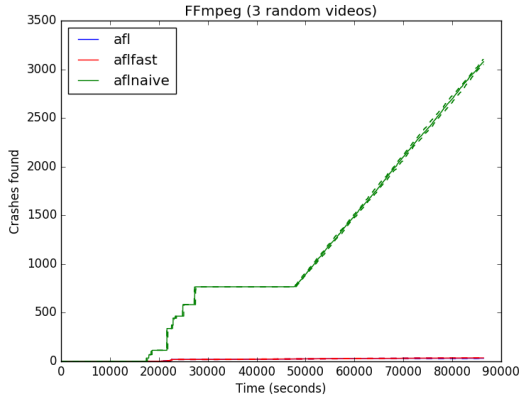


Figure 5.9: 3-made seeds: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 > 0.05$,$p_2 < 10^{-10}$

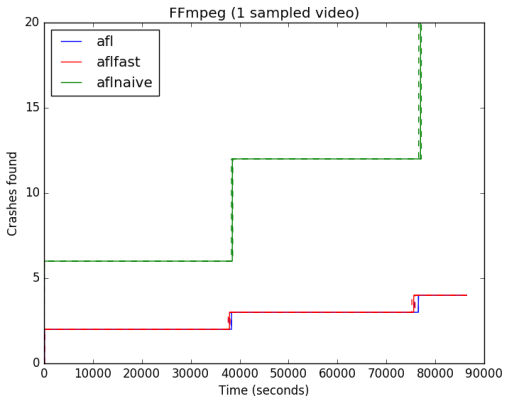


Figure 5.10: 1-sampled seeds: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 > 0.05$,$p_2 < 10^{-5}$

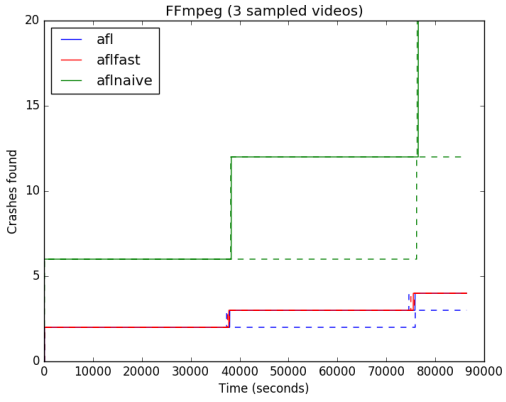


Figure 5.11: 3-sampled seeds: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 > 0.05$,$p_2 < 10^{-5}$

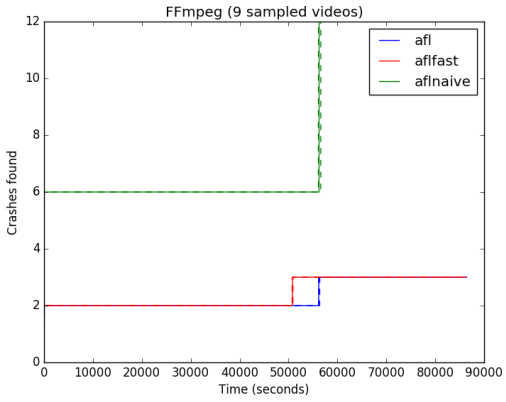


Figure 5.12: 9-sampled seeds: Solid line is median result; dashed lines are confidence intervals. p_1 and p_2 are the p-values for the statistical tests of AFL vs. AFLFast and AFL vs. AFLNaive, respectively. $p_1 > 0.05$,$p_2 < 10^{-6}$

Discussion

While our recommendation to use statistical tests should be uncontroversial, there can be further debate on the best choice of test. In particular, two viable alternatives are the *permutation test* [194] and *bootstrap*-based tests [195]. These tests work by treating the measured data as a kind of stand-in for the overall population, systematically comparing permutations and re-samples of measured data to create rankings with confidence intervals. Whether such methods are more or less appropriate than Mann Whitney is unclear to us, so we follow Arcuri and Briand [193].

Determining that the median performance of fuzzer A is greater than fuzzer B is paramount, but a related question concerns *effect size*. Just because A is likely to be better than B doesn't tell us *how much* better it is. We have been implicitly answering this question by looking at the difference of the measured medians. Sta-

tistical methods could also be used to determine the likelihood that this difference represents the true difference. Arcuri and Briand suggest Vargha and Delaney's $\hat{A}_{12}$ statistics [196] (which employ elements of the Mann Whitney calculation). Bootstrap methods can also be employed here.

5.5 Seed Selection

Recall from Figure 5.1 that prior to iteratively selecting and testing inputs, the fuzzer must choose an initial corpus of *seed* inputs. Most (27 out of 32, per Section 5.2.2) recent papers focus on improving the main fuzzing loop. As shown in column **seed** in Table 5.1, most papers (30/32) used a non-empty seed corpus (entries with G, R, M, V, or N). A popular view is that a seed should be well-formed ("valid") and small—such seeds may drive the program to execute more of its intended logic quickly, rather than cause it to terminate at its parsing/well-formedness tests [153,155,165,197]. And yet, many times the details of the particular seeds used were not given. Entry 'V' appears 9 times, indicating a valid seed corpus was used, but providing no details. Entry 'N' appears 10 times, indicating a non-empty seed, but again with no details as to its content. Two papers [151,186] opted to use an empty seed (entry 'E'). When we asked them about it, they pointed out that using an empty seed is an easy way to baseline a significant variable in the input configuration. Other papers used manually or algorithmically constructed seeds, or randomly sampled ones.

It may be that the details of the initial seed corpus are unimportant; e.g.,

that no matter which seeds are used, algorithmic improvements will be reflected. But it's also possible that there is a strong and/or surprising interaction between seed format and algorithm choice which could add nuance to the results [198]. And indeed, this is what our results suggest.

We tested *FFmpeg* with different seeds including the empty seed, samples of existing video files ("sampled" seeds) and randomly-generated videos ("made" seeds). For the sampled seeds, videos were drawn from the *FFmpeg* samples website.[2] Four samples each were taken from the AVI, MP4, MPEG1, and MPEG2 sub-directories, and then the files were filtered out to only include those less than 1 MiB, AFL's maximum seed size, leaving *9-sampled seeds* total. This set was further pared down to the smallest of the video files to produce *3-sampled* and *1-sampled* seeds. For the made seeds, we generated video and GIF files by creating 48 random video frames with `videogen` (a tool included with *FFmpeg*), 12 seconds of audio with `audiogen` (also included), and stitching all of them together with *FFmpeg* into *3-made* MP4, MPG, and AVI files, each at 4 fps. The *1-made* seed is the generated MP4 file. We also tested *nm*, *objdump*, and *cxxfilt* using the empty seed, and a 1-made seed. For *nm* and *objdump*, the 1-made seed was generated by compiling a hello-world C program. The 1-made seed of *cxxfilt* was generated as a file with 16 random characters, chosen from the set of letters (uppercase and lowercase), digits 0-9, and the underscore, which is the standard alphabet of mangled C++ names.

Results with these different seed choices for *FFmpeg* are shown in Figures 5.7, 5.8, 5.9, 5.10, 5.11, and fig:ffmpeg-9sampled-seed. One clear trend is that for AFL

[2] `http://samples.ffmpeg.org`

	empty	**1-made**
FFmpeg, AFLNaive	0 ($< 10^{-15}$)	5000 ($< 10^{-11}$)
FFmpeg, AFL	382.5	102
FFmpeg, AFLFast	369.5 ($= 0.379$)	129 (< 0.05)
nm, AFL	448	23
nm, AFLFast	1239 ($< 10^{-13}$)	24 ($= 0.830$)
objdump, AFL	6.5	5
objdump, AFLFast	29 ($< 10^{-3}$)	6 ($< 10^{-2}$)
cxxfilt, AFL	540.5	572.5
cxxfilt, AFLFast	1400 ($< 10^{-10}$)	1364 ($< 10^{-10}$)

Table 5.2: Crashes found with different seeds. Median number of crashes at the 24-hour timeout.

and AFLFast, the empty seed yields far more crashing inputs than any set of valid, non-empty ones. On the other hand, for AFLNaive the trend is reversed. Among the experiments with non-empty seeds, performance also varies. For example, Figure 5.8 and Figure 5.10 show very different performance with a single, valid seed (constructed two different ways). The former finds around 100 crashes for AFL and AFLFast after 24 hours, while the latter finds less than 5.

The top part of Table 5.2 zooms in on the data from Figure 5.7 and Figure 5.8 at the 24-hour mark. The first column indicates the target program and fuzzer used; the second column ("**empty**") indicates the median number of crashes found when using an empty seed; and the last column ("**1-made**") indicates the median number of crashes found when using a valid seed. The parenthetical in the last two columns is the p-value for the statistical test of whether the difference of AFLFast or AFLNaive performance from AFL is real, or due to chance. For AFL and AFLFast, an empty

seed produces hundreds of crashing inputs, while for AFLNaive, it produces none. However, if we use 1-made or 3-made seeds, AFLNaive found significantly more crashes than AFL and AFLFast (5000 vs. 102/129).

The remainder of Table 5.2 reproduces the results of the AFLFast evaluation [151] in the **empty** column, but then reconsiders it with a valid seed in the **1-made** column. Similar to the conclusion made by the AFLFast paper, AFLFast is superior to AFL in crash finding ability when using the empty seed (with statistical significance). However, when using 1-made seeds, AFLFast is not quite as good: it no longer outperforms AFL on *nm*, and both AFL and AFLFast generally find fewer crashes.

In sum, it is clear that a fuzzer's performance on the same program can be very different depending on what seed is used. Even valid, but different seeds can induce very different behavior. Assuming that an evaluation is meant to show that fuzzer A is superior to fuzzer B *in general*, our results suggest that it is prudent to consider a variety of seeds when evaluating an algorithm. Papers should be specific about how the seeds were collected, and better still to make available the actual seeds used. We also feel that the empty seed should be considered, despite its use contravening conventional wisdom. In a sense, it is the most general choice, since an empty file can serve as the input of any file-processing program. If a fuzzer does well with the empty seed across a variety of programs, perhaps it will also do well with the empty seed on programs not yet tested. And it takes a significant variable (i.e., which file to use as the seed) out of the vast configuration space.

5.6 Timeouts

Another important question is how long to run a fuzzer on a particular target. The last column of Table 5.1 shows that prior experiments of fuzzers have set very different timeouts. These generally range from 1 hour to days and weeks.[3] Common choices were 24 hours (10 papers) and 5 or 6 hours (7 papers). We observe that recent papers that used LAVA as the benchmark suite chose 5 hours as the timeout, possibly because the same choice was made in the original LAVA paper [199]. Six papers ran fuzzers for more than one day.

Most papers we considered reported the timeout without justification. The implication is that beyond a certain threshold, more running time is not needed as the distinction between algorithms will be clear. However, we found that relative performance between algorithms can change over time, and that terminating an experiment too quickly might yield an incomplete result. As an example, AFLFast's evaluation shows that AFL found no bugs in *objdump* after six hours [151], but running AFL longer seems to tell a different story, as shown in Figure 5.3. After six hours, both AFL and AFLFast start to find crashes at a reasonable clip. Running AFL on *gif2png* shows another interesting result in Figure 5.6. The median number of crashes found by AFL was 0 even after 13 hours, but with only 7 more hours, it found 40 crashes. Because bugs often reside in certain parts of the program, fuzzing detects the bugs only when these parts are eventually explored. Figure 5.13 presents

[3] [178] is an outlier that we do not count here: it uses 5-minute timeout because its evaluation focuses on test generation rate instead of bug finding ability.

the results of AFL and AFLFast running with three sampled seeds on *nm*. After 6 hours none of the AFL runs found any bugs in *nm*, while the median number of crashes found by AFLFast was 4; Mann Whitney says that this difference is significant. But at 24 hours, the trend is reversed: AFL has found 14 crashes and AFLFast only 8. Again, this difference is significant.

What is a reasonable timeout to consider? Shorter timeouts are convenient from a practical perspective, since they require fewer overall hardware resources. Shorter times might be more useful in certain real-world scenarios, e.g., as part of an overnight run during the normal development process. On the other hand, longer runs might illuminate more general performance trends, as our experiments showed. Particular algorithms might also be better with longer running times; e.g., they could start slow but then accelerate their bug-finding ability as more tests are generated. For example, Skyfire took several days before its better performance (over AFL) became clear [165].

We believe that evaluations should include plots, as we have been (e.g., in Figure 5.13), that depict performance over time. These runs should consider at least a 24 hour timeout; performance for shorter times can easily be extracted from such longer runs.

Discussion

In addition to noting performance at particular times (e.g., crash counts at 5, 8 and 24 hours), one could also report *area under curve* (AUC) as a less punctuated

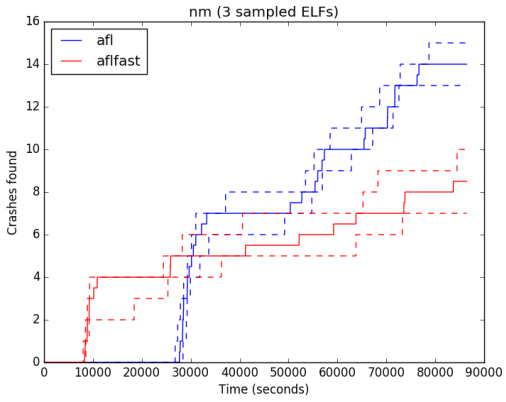


Figure 5.13: *nm* with three sampled seeds. At 6 hours: AFLFast is superior to AFL with $p < 10^{-13}$. At 24 hours: AFL is superior to AFLFast with $p = 0.000105$.

performance measure. For example, a fuzzer that found one crash per second for five seconds would have an AUC of 12.5 crash-seconds whereas a fuzzer that found five crashes too, but all between seconds 4 and 5, would have an AUC of 2.5 crash-seconds. These measures intuitively reflect that finding crashes earlier and over time is preferred to finding a late burst. On the other hand, this measure might prefer a steady crash finder that peaks at 5 crashes to one that finds 10 at the last gasp; aren't more crashes better? As such, AUC measures are not a substitute for time-based performance plots.

5.7 Performance

Many papers employ some strategy to *de-duplicate* (or *triage*) crashes, so as to map them to unique bugs. There are two popular automated heuristics for doing this: using AFL's notion of *coverage profile*, and using *stack hashes*. In Table 5.1, these are marked 'C' (7 papers) and 'S' (7 papers) in the **crash** column. There

are four papers using other tools/methods for triage, marked 'O'. For example, VUzzer additionally used a tool called `!Exploitable` to assess the exploitability of a crash caused by a bug [153]. Crashes that have a low likelihood of being turned into an attack could be discounted by a user, so showing that a fuzzer finds more dangerous bugs is advantageous. The de-duplication strategy used in our experiments corresponds to 'C'.

Several papers do consider some form of ground truth. Six papers use it as their main performance measure, marked 'G' in the table. By virtue of their choice of benchmark programs, they are able to map crashing inputs to their root cause perfectly. Eight other papers, marked 'G*' in the table, make some effort to triage crashes to identify their root cause, but do so imperfectly. Typically, such triage is done as a 'case study' and is often neither well founded nor complete—ground truth is not used as the overall (numeric) performance measure.

Unfortunately, commonly used de-duplication heuristics are poor at clustering crashing inputs according to their root cause. We demonstrate this further in the next chapter.

5.7.1 Code Coverage

Fuzzers are run to find bugs in programs. A fuzzer that runs for a long period of time and finds no bugs would be seen as unsuccessful by its user. It seems logical to evaluate a fuzzer based on the number of bugs that fuzzer finds. However, just because a fuzzer does not find a bug may not tell us the whole story about the

fuzzer's efficacy. Perhaps its algorithm is sound but there are few or no bugs to find, and the fuzzer has merely gotten unlucky.

One solution is to instead (or also) measure the improvement in code coverage made by fuzzer A over baseline B. Greybox fuzzers already aim to optimize coverage as part of the `isInteresting` function, so surely showing an improved code coverage would indicate an improvement in fuzzing. This makes sense. To find a crash at a particular point in the program, that point in the program would need to execute. Prior studies of test suite effectiveness also suggest that higher coverage correlates with bug finding effectiveness [157, 158]. Nearly half of the papers we considered measured code coverage; FairFuzz *only* evaluated performance using code (branch) coverage [185].

However, there is no *fundamental* reason that maximizing code coverage is directly connected to finding bugs. While the general efficacy of coverage-guided fuzzers over black box ones implies that there's a strong correlation, particular algorithms may eschew higher coverage to focus on other signs that a bug may be present. For example, AFLGo [186] does not aim to increase coverage globally, but rather aims to focus on particular, possibly error-prone points in the program. Even if we assume that coverage and bug finding are correlated, that correlation may be weak [159]. As such, a substantial improvement in coverage may yield merely a negligible improvement in bug finding effectiveness.

In short, we believe that code coverage makes sense as a secondary measure, but that ground truth, according to bugs discovered, should always be primary.

5.8 Target Programs

We would like to establish that one fuzzing algorithm is *generally* better than another, i.e., in its ability to find bugs in any target program drawn from a (large) population. Claims of generality are usually made by testing the fuzzer on a benchmark suite that purports to represent the population. The idea is that good performance on the suite should translate to good performance on the population. How should we choose such a benchmark suite?

Recent published works have considered a wide variety of benchmark programs. Broadly, these fall into two categories, as shown in the second column in Table 5.1: real programs and artificial programs (or bugs). Examples of the former include the Google fuzzer test suite ("G") [200] and ad hoc selections of real programs ("R"). The latter comprises CGC ("C") [201], LAVA-M ("L") [199], and hand-selected programs with synthetically injected bugs ("S"). Some papers' benchmarks drew from both categories (e.g., VUzzer [153] and Steelix [171]). As we discuss below, no existing benchmark choice is entirely satisfying, thus leaving open the important question of developing a good fuzzing benchmark.

5.8.1 Real programs

According to Table 5.1, nearly all papers used some real-world programs in their evaluations. Two of these papers [166, 178] used the Google Fuzzer Test suite [200], a set of real-world programs and libraries coupled with harnesses to focus fuzzing on a set of known bugs. The others evaluated on a hand selected set

of real-world programs.

We see two problems with the way that real programs have been used as fuzzing targets. First, most papers consider only a small number of target programs without clear justification of their representativeness. The median number of programs, per Table 5.1, is seven. Sometimes a small count is justified; e.g., IMF was designed specifically to fuzz OS kernels, so its evaluation on a single "program," the MacOS kernel, is still interesting. On the other hand, most fuzzers aim to apply to a larger population (e.g., all file processing programs), so 7 would seem to be a small number. A positive outlier was FuzzSim, which used a large set of programs (more than 100) and explained the methodology for collecting them.

As evidence of the threat posed by a small number of insufficiently general targets, consider the experimental results reported in Figures 5.2, 5.3, 5.4, 5.5, and 5.6, which match the results of Böhme et al [151]. The first row of the figure shows results for *nm*, *objdump* and *cxxfilt*, which were the three programs in which Böhme et al found crashes.[4] Focusing our attention on these programs suggests that AFLFast is uniformly superior to AFL in crash finding ability. However, if we look at the second row of the figure, the story is not as clear. For both *FFmpeg* and *gif2png*, two programs used in other fuzzing evaluations, the Mann Whitney

[4]Figure 6 of their paper presents a similar series of plots. The differences in their plots and ours are the following: they plot the results on log scale for the Y axis; they consider six-hour trials rather than 24-hour trials; and they do not plot median and confidence intervals computed over 30+ runs, but rather plot the mean of 8 runs. They also use different versions of AFL and AFLFast.

U test shows no statistical difference between AFL and AFLFast. Including these programs in our assessment weakens any claim that AFLFast is an improvement over AFL.

The second problem we see with the use of real programs to date is that few papers use the same targets, at the same versions. As such, it is hard to make even informal comparisons across different papers. One overlapping set of targets were *binutils* programs, used in several evaluations [151, 172, 185, 186]. Multiple papers also considered *FFmpeg* and *gif2png* [152–156]. However, none used the same versions. For example, the versions of *binutils* were different in these papers: AFLFast [151] and AFLGo [186] used 2.26; FairFuzz [185] used 2.28; Angora [172] used 2.29.

The use of Google Fuzzer Suite would seem to address both issues: it comprises 25 programs with known bugs, and is defined independently of any given fuzzer. On the other hand, it was designed as a kind of regression suite, not necessarily representative of fuzzing "in the wild;" the provided harnesses and seeds mostly intend that fuzzers should find the targeted bugs within a few seconds to a few minutes.

5.8.2 Suites of artificial programs (or bugs)

Real programs are fickle in that the likelihood that bugs are present depends on many factors. For example, programs under active development may well have more bugs than those that are relatively stable (just responding to bug reports).

In a sense, we do not care about any particular set of programs, but rather a representative set of programming (anti)patterns in which bugs are likely to crop up. Such patterns could be injected artificially. There are two popular suites that do this: CGC, and LAVA-M.

The CGC suite comprises 296 buggy programs produced as part of DARPA's Cyber Grand Challenge [201]. This suite was specifically designed to evaluate bug finding tools like fuzz testers—the suite's programs perform realistic functions and are seeded with exploitable bugs. LAVA (which stands for Large-scale Automated Vulnerability Addition) is a tool for injecting bugs into known programs [199]. The tool is designed to add crashing, input-determinate bugs along feasible paths. The LAVA authors used the tool to create the LAVA-M suite, which comprises four bug-injected *coreutils* programs: *base64*, *md5sum*, *uniq*, and *who*. Unlike the CGC programs, which have very few injected bugs, the LAVA-M programs have many: on the order of a few dozen each for the first three, and more than 2000 for *who*. For both suites, if a fuzzer triggers a bug, there is a telltale sign indicating which one it is, which is very useful for understanding how many bugs are found from the total possible.

CGC and LAVA-M have gained popularity as the benchmark choices for evaluating fuzzers since their introduction. Within the past two years, CGC and LAVA-M have been used for evaluating 4 and 5 fuzzers, respectively. VUzzer [153], Steelix [171], and T-Fuzz [181] used both benchmarks in their evaluation. However, sometimes the CGC benchmark was subset: Driller [149], VUzzer [153], and Steelix [171] were evaluated on 126, 63, and 17 out of the 296 programs, respectively.

While CGC programs are hand-designed to simulate reality, this simulation may be imperfect: Performing well on the CGC programs may fail to generalize to actual programs. For example, the average size of the CGC cqe-challenge programs was (only) 1774 lines of code, and many programs use telnet-style, text-based protocols. Likewise, LAVA-M injected bugs may not sufficiently resemble those found "in the wild." The incentives and circumstances behind real-world software development may fail to translate to synthetic benchmarks which were specifically designed to be insecure. The LAVA authors write that, "A significant chunk of future work for LAVA involves making the generated corpora look more like the bugs that are found in real programs." Indeed, in recent experiments [202], they also have shown that relatively simple techniques can effectively find all of the LAVA-M bugs, which follow a simple pattern. We are aware of no study that independently assesses the extent to which these suites can be considered "real" or "general."

5.8.3 Toward a Fuzzing Benchmark Suite

Our assessment leads us to believe that there is a real need for a solid, independently defined benchmark suite, e.g., a DaCapo [203] or SPEC[5] for fuzz testing. This is a big enough task that we do not presume to take it on in this chapter. It should be a community effort. That said, we do have some ideas about what the result of that effort might look like.

First, we believe the suite should have a selection of programs with clear indicators of when particular bugs are found, either because bugs are synthetically

[5] `https://www.spec.org/benchmarks.html`

introduced (as in LAVA-M and CGC) or because they were previously discovered in older versions (as in our ground truth assessment in Section 6.2). Clear knowledge of ground truth avoids overcounting inputs that correspond to the same bug, and allows for assessing a tool's false positives and false negatives. We lean toward using real programs with known bugs simply because their ecological validity is more assured.

Second, the suite should be large enough (both in number of programs, and those programs' sizes) to represent the overall target population. How many programs is the right number? This is an open question. CGC comprises ~ 300 small programs; Google Fuzzer Suite has 25; most papers used around 7. Our feeling is that 7 is too small, but it might depend on which 7 are chosen. Perhaps 25 is closer to the right number.

Finally, the testing methodology should build in some defense against overfitting. If a static benchmark suite comes into common use, tools may start to employ heuristics and strategies that are not of fundamental advantage, but apply disproportionately to the benchmark programs. One way to deal with this problem is to have a fixed standard suite and an "evolvable" part that changes relatively frequently. One way to support the latter is to set up a fuzzing competition, similar to long-running series of SAT solving competitions.[6] One effort in this direction is Rode0day, a recurring bug finding competition.[7] Since the target programs would not be known to fuzzing researchers in advance, they should be incentivized to de-

[6] `http://www.satcompetition.org/`

[7] `https://rode0day.mit.edu/`

velop general, reusable techniques. Each competition's suite could be rolled into the static benchmark, at least in part, to make the suite even more robust. One challenge is to regularly develop new targets that are ecologically valid. For example, Rode0day uses automated bug insertion techniques to which a tool could overfit (the issue discussed above for LAVA).

5.9 Conclusions and Future Work

Fuzz testing is a promising technology that has been used to uncover many important bugs and security vulnerabilities. This promise has prompted a growing number of researchers to develop new fuzz testing algorithms. The evidence that such algorithms work is primarily experimental, so it is important that it comes from a well-founded experimental methodology. In particular, a researcher should run their algorithm A on a general set of target programs, using a meaningful set of configuration parameters, including the set of input seeds and duration (timeout), and compare against the performance of a baseline algorithm B run under the same conditions, where performance is defined as the number of (distinct) bugs found. A and B must be run enough times that the inherent randomness of fuzzing is accounted for and performance can be judged via a statistical test.

In this chapter, we surveyed 32 recent papers and analyzed their experimental methodologies. We found that no paper completely follows the methodology we have outlined above. Moreover, results of experiments we carried out using AFLFast [151] (as A) and AFL [148] (as B) illustrate why not following this methodology can lead

to misleading or weakened conclusions. We found that

- Most papers failed to perform multiple runs, and those that did failed to account for varying performance by using a statistical test. This is a problem because our experiments showed that run-to-run performance can vary substantially.

- Many papers measured fuzzer performance not by counting distinct bugs, but instead by counting "unique crashes" using heuristics such as AFL's coverage measure and stack hashes. This is a problem because experiments we carried out showed that the heuristics can dramatically over-count the number of bugs, and indeed may suppress bugs by wrongly grouping crashing inputs. This means that apparent improvements may be modest or illusory. Many papers made some consideration of root causes, but often as a "case study" rather than a performance assessment.

- Many papers used short timeouts, without justification. Our experiments showed that longer timeouts may be needed to paint a complete picture of an algorithm's performance.

- Many papers did not carefully consider the impact of seed choices on algorithmic improvements. Our experiments showed that performance can vary substantially depending on what seeds are used. In particular, two different non-empty inputs need not produce similar performance, and the empty seed can work better than one might expect.

- Papers varied widely on their choice of target programs. A growing number are using synthetic suites CGC and/or LAVA-M, which have the advantage that they are defined independently of a given algorithm, and bugs found by fuzzing them can be reliably counted (no crash de-duplication strategy is needed). Other papers often picked small, disjoint sets of programs, making it difficult to compare results across papers. Our experiments showed AFLFast performs well on the targets it was originally assessed against, but performed no better than AFL on two targets used by other papers.

Ultimately, our experiments roughly matched the positive results of the original AFLFast paper [151], but by expanding the scope of the evaluation to different seeds, longer timeouts, and different target programs, evidence of AFLFast's superiority, at least for the versions we tested, was weakened. We believe the same could be said of many prior papers—all suffer from problems in their evaluation to some degree. As such, a key conclusion of this chapter is that the fuzzing community needs to start carrying out more rigorous experiments in order to draw more reliable conclusions.

Specifically, we recommend that fuzz testing evaluations should have the following elements:

- multiple trials with statistical tests to distinguish distributions;
- a range of benchmark target programs with known bugs (e.g., LAVA-M, CGC, or old programs with bug fixes);
- measurement of performance in terms of known bugs, rather than heuristics based on AFL coverage profiles or stack hashes; block or edge coverage can be

used as a secondary measure;

- a consideration of various (well documented) seed choices including empty seed;
- timeouts of at least 24 hours, or else justification for less, with performance plotted over time.

We see (at least) three important lines of future work. First, we believe there is a pressing need for well-designed, well-assessed benchmark suite, as described at the end of the last section. Second, and related, it would be worthwhile to carry out a larger study of the value of crash de-duplication methods on the results of realistic fuzzing runs, and potentially develop new methods that work better, for assisting with triage and assessing fuzzing when ground truth is not known. Recent work shows promise [161, 162]. Finally, it would be interesting to explore enhancements to the fuzzing algorithm inspired by the observation that no single fuzzing run found all true bugs in *cxxfilt*; ideas from other search algorithms, like SAT solving "reboots" [204], might be brought to bear.

Chapter 6: De-duplication, clustering, and root cause analysis

As our experiments in Chapter 5 show, fuzzing has real power: it finds real bugs, and those bugs are demonstrated via concrete test cases that show the program failing. These concrete test cases can be used directly to triage and debug the failing program and discover the root cause of the failure. However, we also show that fuzzing produces a great number of these test cases, in Chapter 5 we amassed over 50,000 concrete inputs that cause `nm` to crash, and another 50,000+ for `cxxfilt`. This is problematic because these inputs can be redundant, large, and unstructured.

The large number of crashing inputs presents problems for both the practitioner and researcher. A large number of inputs poses a triaging problem for the practitioner: Which inputs to consider first when chasing down the root cause of a bug [205]? The nature of memory corruption-based crashes means that the source of the failure might be far away from where the crash occurred, creating debugging challenges, particularly when inputs are large. If multiple inputs trigger the same root cause, it would be ideal to focus on just one: the smallest and simplest. Researchers have a similar triaging problem: since researchers are unlikely to understand how the target programs work, they have even less insight into whether each element within a set of concrete test cases represents a failure with the same

root cause, or different root causes, or how many different root causes there might be. We want to fuzzers that find many bugs, not many inputs that are evidence of the same bug.[1]

In this chapter, we consider how to measure the success of a fuzzing campaign. Our approach is to consider the problem as either a root cause analysis problem, or a feature extraction and clustering problem. To begin, we describe a process by which we recovered ground truth for the crashes in *cxxfilt* discovered during Chapter 5. With this ground truth data, we can measure how imprecise existing measures of fuzzing success are. After performing this analysis, we look in more detail at the root causes of each bug. Then, we consider methods for root cause analysis as well as feature extraction and clustering: AFL coverage profiles, stack hashing, and clustering with coverage maps, evaluating each against ground truth. We close with discussion of potential for future work.

6.1 Ground Truth: Bugs Found

The ultimate measure of a fuzzer is the number of unique bugs that it finds. If fuzzer A generally finds more bugs than baseline B then we can view it as more effective. A key question is: What is a (unique) bug? This is a subjective question with no easy answer.

We imagine that a developer will ultimately use a crashing input to debug and

[1]Sometimes this is beneficial, for example if a patch is incomplete and buggy behavior can be reached via a different path with different pre-conditions, but in general, a single, minimal test case will do.

fix the target program so that the crash no longer occurs. That fix will probably not be specific to the input, but will generalize. For example, a bugfix might consist of a length check to stop a buffer overrun from occurring—this will work for all inputs that are too long. As a result, if target p crashes when given input I, but no longer crashes when the bugfix is applied, then we can associate I with the bug addressed by the fix [206]. Moreover, if inputs I_1 and I_2 both induce a crash on p, but both no longer do so once the bugfix is applied, we know that both *identify the same bug* (assuming the fix is suitably "minimal" [207]).

When running on target programs with known bugs, we have direct access to ground truth. Such programs might be older versions with bugs that have since been fixed, or they might be synthetic programs or programs with synthetically introduced bugs. Considering the former category, we are aware of no prior work on fuzzing that uses old programs and their corresponding fixes to completely triage crashes according to ground truth. In the latter category, nine papers we looked at in Chapter 5 use synthetic suites in order to determine ground truth. The most popular suites are CGC (*Cyber Grand Challenge*) [201] and LAVA-M [199]; we discuss these more in the next section. For both, bugs have been injected into the original programs in a way that triggering a particular bug produces a telltale sign (like a particular error message) before the program crashes. As such, it is immediately apparent which bug is triggered by the fuzzer's generated input. If that bug was triggered before, the input can be discarded. Two other papers used hand-selected programs with manually injected vulnerabilities.

6.1.1 Methodology

We examined the crashing inputs our fuzzing runs generated for *cxxfilt* using AFL and AFLFast. Years of development activity have occurred on this code since the version we fuzzed was released, so (most of) the bugs that our fuzzing found have been patched. We used `git` to identify commits that change source files used to compile *cxxfilt*. Then, we built every version of *cxxfilt* for each of those commits. This produced 812 different versions of *cxxfilt*. Then, we ran every crashing input (57,142 of them) on each different version of *cxxfilt*, recording whether or not that version crashed. If not, we consider the input to have been a manifestation of a bug fixed by that program version.

Each of the 812 different versions needed to be evaluated on each of the 57,142 input files, or, 46,399,304 test cases. This problem was outside the scope of small computers, so a distributed system was built using Amazon EC2 and Apache Mesos [208]. This distributed system used 412 CPUs across multiple different virtual machines, coordinated via Mesos. Each core would be assigned a "batch" of test cases to run at once. Batching was performed because each test requires the setup and teardown of a Docker virtual machine. In this case, *cxxfilt* doesn't make any transient modifications to the system, so it poses little risk to start a Docker instance, and then run each test case in a batch serially within the instance, recording the results.

It is natural to think that running the full cross product of 46,399,304 test cases is unnecessary and that the system could instead do a bisection based search

over the commit times to identify the transition point when an input goes from crashing to not crashing. However, this does not work in general, because in the progression from one commit to another, it is possible for an input to cycle from crashing to not crashing and then back to crashing. This non-monotonic behavior has some explanations. For one, due to branching in Git, there is no true "linear" representation of commits: one could test commits on a feature branch and master and the "final" transition occurs when the feature branch is merged. Additionally, since the commits in question include "in progress" work that might represent feature regression, incomplete, or incorrect work, it is possible that one commit causes *every* (or almost every) input to stop crashing.

Mesos works by responding to "resource offers" from available cluster resources. As a resource becomes available, it asks the controller for a single item of work to perform. When that work is assigned and then completed, the results are reported back to the controller. In our setting, those results report for each pair of commit hash and input path assigned as part of the batch, if that run crashed or did not crash, and if it did crash, it reports the results of the ASAN log for the crash. That log is then parsed into a call stack, which is efficiently encoded as a series of relationships between rows in a table for stack frames in a SQLite database, where the result of the run is also recorded.

The controller pre-computes the total "task set" when the process starts and records the entire run into the SQLite database, setting the "status" column for each initially to a sentinel value indicating that test has not run yet. In response to resource availability announcements, the controller queries the database for any

tasks which have not yet run, limited by the batch size. It then assigns that task set to run on a free CPU.

This setup has a few advantages. If a particular worker node falls offline [2] the controller is notified and can re-schedule the tasks originally assigned to that node. If the controller itself is taken offline, when it comes back online it has access to the list of work done and work left to do in the database.

To help ensure that our triaging results are trustworthy, we took two additional steps. First, we ensured that non-crashing behavior was not incidental. Memory errors and other bug categories uncovered by fuzzing may not always cause a crash when triggered. For example, an out-of-bounds array read will only crash the program if unmapped memory is accessed. Thus it is possible that a commit could change some aspect of the program that eliminates a crash without actually fixing the bug. To address this issue, we compiled each *cxxfilt* version with Address Sanitizer and Undefined Behavior Sanitizer (ASAN and UBSAN) [209], which adds dynamic checks for various errors including memory errors. We considered the presence of an ASAN/UBSAN error report as a "crash."

Second, we ensured that each bug-fixing commit corresponds to a single bugfix, rather than several. To do so, we manually inspected every commit that converted a crashing input to a non-crashing one, judging whether we believed multiple unique bugs were being fixed (based on principles we developed previously [207]). If so, we manually split the commit into smaller ones, one per fix. In our experiments, we only had to do this once, to a commit that imported a batch of changes from the

[2] AWS EC2 nodes are less available than you might think.

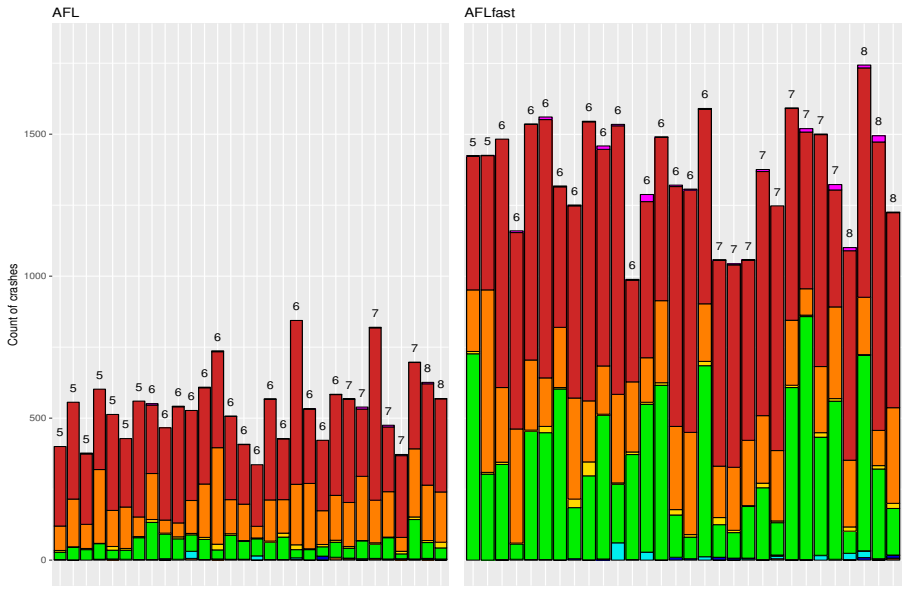


Figure 6.1: Crashes with unique bugs found per run for *cxxfilt*. Each bar represents an independent run of either AFL or AFLfast. The height of the bar is the count of crashing inputs discovered during that run. Each bar is divided by color, clustering inputs with other inputs that share the same root cause. Number of unique bugs is indicated above each bar.

libiberty fork of *cxxfilt* into the main trunk.[3] We looked at the individual *libiberty* commits that made up this batch to help us determine how to split it up. Ultimately we broke it into five distinct bug-fixing commits.

6.1.2 Discussion of bugs

In previous sections one might wonder, what is it about some bugs that either leads them to be triggered by many different inputs, or, leads them to be poorly clustered by either stack hashing or coverage profiles? In this section, we take a closer, qualitative look at each patch discovered in the previous section. Each bug is identified by the date and time the "fixing patch" was committed.

[3] `https://github.com/gcc-mirror/gcc/tree/master/libiberty`

Patch 2016-01-18 10:58:47

This patch made two distinct changes, changing the parameter type of a function from `int` to `long`, and adding a check against `NULL`.

`int` to `long`:

Figure A.1 shows the primary change for this patch, which changes the parameter type. At first, this change seems perplexing: why would such a small change cause such a large number of crashing inputs to be fixed? The answer is that in the `d_identifier` function, the `len` parameter is used to advance the cursor (`struct d_info` can be thought of as a cursor in this context) by `len`, and the caller's value for this parameter is truncated from a `long` to an `int`. This means the `len` parameter value can become a large, negative value, pushing the cursor in any direction, including off either the beginning or end of the input buffer.

Further investigation reveals why both clustering inputs using both program coverage and stack hashes performs poorly. The value in `len` is computed via (essentially) `atoi` on an integer string contained in the input. This is done if there is a specific byte sequence in the input, but that byte sequence can occur at (almost) any position in the input. *cxxfilt* can perform any amount of processing before reading that byte sequence, or that byte sequence could be the first in the input. This results in a large number of different paths that all have a common suffix. However, the AFL representation of a path does not include any notion of time or which program points were reached before or after another, so from AFL's view, the inputs that all

"end the same" are different because they took different paths to reach that ending.

There is a similar problem for stack hashing. There is a separation in time from when the cursor value is corrupted to when it is used. When the corrupt cursor is used, it can be used in multiple different contexts that appear unique when $N = 3$ or $N = 5$.

`NULL` check:

Figure A.2 shows the other change associated with this input, a test of a variable against a `NULL` pointer before using that variable in an argument to `strcmp`. Here, the root cause is a failure to check return values. Functions will correctly identify edge or error cases and return `NULL`. However, the callers previously did not check for this value before use. After this patch, they do.

These two logically different changes are included in the same patch. When we separated them, we found that the `NULL` check did not have any effect on the crashing inputs discovered.

Patch 2016-11-18 05:06:18

Figure A.3 shows a patch where a variable (which is the value resulting from calling `d_expression_1`) is tested against `NULL`, and if the test is true, processing exits early.

Meanwhile, Figure A.4 shows a patch where the `mangled` pointer is interrogated to determine if it contains a `'0'` value. Previously, the logic in the patched function

would advance beyond `NUL` terminated pointers, and then potentially beyond the extent of a buffer. Inputs that are fixed by this patch later exhibit crashes on trying to access the `mangled` variable.

Patch 2017-03-13 13:49:32

This patch (Figure A.5) begins to address an issue with unbounded recursion in the printing of mangled names. The patch has to de-`const` many different pointer parameters to functions that are transitively reachable from `d_print_comp`. That is because the patch to `d_print_comp` uses an existing field in the `struct demangle_component`, `d_printing`, as a "latch" to guard against re-entrance. If the latch is 0, then it is incremented and processing proceeds. If the latch is non-zero, processing ends with an error.

Patch 2016-10-17 05:26:56

This patch also addresses unbounded recursion, but in a different part of the parsing code. Here, the fix (Figures A.6,A.7,A.8) is to remember previously performed work and skip work already done.

Patch 2017-06-25 05:39:05

This patch re-visits the patch to `d_print_comp`. It also addresses unbounded recursion, in the same function. The previous change, using `d_printing` as a "latch", is left in place. A new field, `d_recursion` is added to the `demangle_component` struc-

ture, and that field is incremented on every entry to the function. If the `d_recursion` field is greater than a `MAX_RECURSION_COUNT` constant, or, `d_printing` is set, processing stops. This "layering" of conditions seems to be to account for multiple kinds of recurrence when processing a `struct demangle_component`.

These three patches ("2017-03-13 13:49:32", "2016-10-17 05:26:56", and "2017-06-25 05:39:05") all localize poorly because the way that they manifest is with a stack overflow. The faulting location would be the location in the program where the stack is exhausted. Exactly where this occurs is dependent on the structure of the input and how that structure relates to the amount of stack consumed. So, the failure will be spread across the different locations in the (recursive) call tree where the most stack allocations occur, i.e. different function entry points.

Patch 2016-08-02 07:56:28

Figure A.10 shows a patch that fixes both use-after-free and `NULL` pointer dereference bugs. The issue is that `squangle_mop_up` deallocates members of `struct work_stuff` but leaves fields that indicate the size of allocated objects at their original sizes. The size values are used throughout input processing to determine the validity of a `struct work_stuff` and the amount of data to process. If their sizes are zero, they will be ignored.

This causes similar problems for coverage clustering and stack hashing as "Patch 2016-01-18 10:58:47" because there are multiple different failing contexts "downstream" of a buggy deallocation, as well as multiple different paths that all

lead to the crash, however they are all resolved by the same root cause fix.

Patch 2016-08-02 08:06:28

Figure A.11 shows the simplest patch. This patch adds error checking to the variable `n`, which is assigned to by a call to `consume_count`. That function attempts to convert an ASCII encoded number at the cursor position into an integer by scanning left to right. If either the encoding fails because the number is too large or no number is found, `consume_count` returns -1. However, it does not un-do the advancement on the cursor position, so without error checking, subsequent uses of the cursor will crash.

This explains why this root cause is more effectively triaged via stack hashing: the failure and the check that prevent the failure occur in the same function and there is only one context for reaching this function with either $N =$ or $N = 5$.

Patch 2016-08-02 08:16:28

Figure A.12 shows a series of checks that have been strengthened by this patch. These strengthened checks follow a pattern of ensuring that both the length value recovered by processing is not an error, and also, do not exceed the length of the overall input object.

It could be the case that these checks are semantically "separable" in that each check is one different "bug", however they were all committed together as one patch. We did not attempt to separate this patch for this reason. Perhaps a more

```
int main(int argc, char* argv[]) {
  if (argc >= 2) {
    char b = argv[1][0];
    if (b == 'a') crash();
    else          crash();
  }
  return 0;
}
```

Figure 6.2: How coverage-based deduplication can overcount

advanced technique could do so.

6.2 Approximating ground truth: AFL coverage profiles

When ground truth is not available, researchers commonly employ heuristic methods to de-duplicate crashing inputs. The approach taken by AFL, and used by 7 papers in Table 5.1 (marked 'C'), is to consider inputs that have the same code *coverage profile* as equivalent. AFL will consider a crash "unique" if the edge coverage for that crash either contains an edge not seen in any previous crash, or, is missing an edge that is otherwise in all previously observed crashes.[4]

Classifying duplicate inputs based on coverage profile makes sense: it seems plausible that two different bugs would have different coverage representations. On

[4]AFL also provides a utility, `afl-cmin`, which can be run offline to "prune" a corpus of inputs into a minimal corpus. Specifically, the *afl-cmin* algorithm keeps inputs that contain edges not contained by any other inputs trace. This is different than the AFL on-line algorithm, which also retains inputs missing edges that other inputs' traces have. Only one prior paper that we know of, Angora [172], ran `afl-cmin` on the final set of inputs produced by AFL; the rest relied only on the on-line algorithm, as we do.

the other hand, it is easy to imagine a single bug that can be triggered by runs with different coverage profiles. For example, suppose the function `crash` in the program in Figure 6.2 will segfault unconditionally. Though there is but a single bug in the program, two classes of input will be treated as distinct: those starting with an `'a'` and those that do not.

Assessing against ground truth

In practice, how often does coverage profiling become mistaken or misled about the root causes of bugs?

Our final methodology produced 9 distinct bug-fixing commits, leaving a small number of inputs that still crash the current version of *cxxfilt*. Figure 6.1 organizes these results. Each bar in the graph represents a 24-hour fuzzing trial carried out by either AFL or AFLFast.[5] For each of these, the magnitude of the bar on the y axis is the total number of "unique" (according to coverage profile) crash-inducing inputs, while the bar is segmented by which of these inputs is grouped with a bug fix discovered by our ground truth analysis. Above each bar is the total number of bugs discovered by that run (which is the number of compartments in each bar).

[5]We show each trial's data individually, rather than collecting it all together, because AFL's coverage-based metric was applied to each trial run, not all runs together.

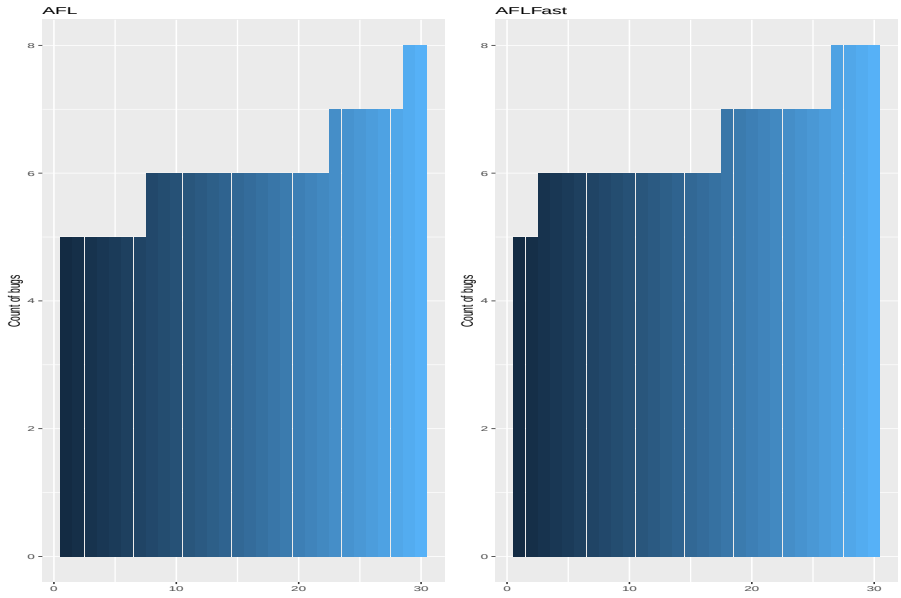


Figure 6.3: The count of unique bugs found by run.

Discussion

The runs are ordered by the number of unique bugs found in the run. We can see that there is at best a weak correlation between the number of bugs found during a run and the number of crashing inputs found in a run. Such a correlation would imply a stronger upward trend of crash counts when moving left to right. We can also see that AFLFast generally found many more "unique" crashing inputs than AFL but the number of bugs found per run is only slightly higher. A Mann Whitney U-test finds that the difference in crashes is statistically significant, with a p-value of 10^{-10}, but the difference in *bugs* is not (but is close)—the p-value is 0.066. This is represented graphically in Figure 6.3.

6.3 Approximating ground truth: Stack hashes

Another common, heuristic de-duplication technique is stack hashing [160]. Seven papers we considered use this technique (marked 'S' in Table 5.1). The idea is the following. Suppose that our buffer overrun bug is in a function deep inside a larger program, e.g., in a library routine. Assuming that the overrun induces a segfault immediately, the crash will always occur at the same place in the program. More generally, the crash might depend on some additional program context; e.g., the overrun buffer might only be undersized when it is created in a particular calling function. In that case, we might look at the *call stack*, and not just the program counter, to map a crash to a particular bug. To ignore spurious variation, we focus on return addresses normalized to their source code location. Since the part of the stack closest to the top is perhaps the most relevant, we might only associate the most recent N stack frames with the triggering of a particular bug. (N is often chosen to be between 3 and 5.) These frames could be hashed for quick comparison to prior bugs—a *stack hash.*

Stack hashing will work as long as relevant context is unique, and still on-stack at the time of crash. But it is easy to see situations where this does not hold—stack hashing can end up both undercounting or overcounting true bugs. Consider the code in Figure 6.4, which has a bug in the `format` function that corrupts a string `s`, which ultimately causes the `output` function to crash (when `s` is passed to it, innocently, by the `prepare` function). The `format` function is called separately by functions `f` and `g`. Prior work has concluded that the stack backtrace alone does

```
void f() { ... format(s1); ... }
void g() { ... format(s2); ... }
void format(char *s) {
  //bug: corrupt s
  prepare(s);
}
void prepare(char *s) {
  output(s);
}
void output(char *s) {
  //failure manifests
}
```

Figure 6.4: How stack hashing can over- and undercount bugs

not contain sufficient information for root cause analysis [210].

Suppose we fuzz this program and generate inputs that induce two crashes, one starting with the call from `f` and the other starting with the call from `g`. Setting N to the top 3 frames, the stack hash will correctly recognize that these two inputs correspond to the same bug, since only `format`, `prepare` and `output` will be on the stack. Setting N to 5, however, would treat the inputs as distinct crashes, since now one stack contains `f` and the other contains `g`. On the other hand, suppose this program had another buggy function that also corrupts `s` prior to passing it to `prepare`. Setting N to 2 would improperly conflate crashes due to that bug and ones due the buggy `format`, since only the last two functions on the stack would be considered.

Assessing against ground truth

We measured the effectiveness of stack hashing by comparing its determinations against the labels for bugs that we identified in the prior experiment. Our implementation of stack hashing uses Address Sanitizer to produce a stack trace for each crashing input to *cxxfilt*, and compute stack hashes of varying depths, from $N = 1$ to $N = 5$.

Our analysis discovered that stack hashing is far more effective at de-duplicating inputs than coverage profiles, but would still over-count the number of bugs discovered. Table 6.1 shows the results of the comparison of stack hashing to the labels we identified. As an example, consider label B, which represents 31,103 inputs (column 5). Of those inputs, for $N = 3$, 362 distinct stack hashes were produced (# column). If the stack hash metric was the only knowledge we had about the distribution of

---

Table 6.1 *(following page)*: Stack hashing results for *cxxfilt*. The first column specifies the label we assign based testing progressive versions of *cxxfilt*. Then, there are three column groups for each set of stack hashes at depth $N = 1$, $N = 2$ and so on. The # column specifies the number of distinct stack hashes among the inputs assigned to the ground truth label. The TP column counts how many of the stack hashes from the second column appear only with those inputs grouped by the label in the first column, while the FP column counts how many stack hashes appear in other labels.

Bug	Inputs	$N=1$			$N=2$			$N=3$			$N=4$			$N=5$		
		#	TP	FP	#	TP	FP	#	TP	FP	#	TP	FP	#	TP	FP
A (2015-11-17 06:37:14)	228	7	1	6	8	2	6	9	2	7	10	3	7	10	3	7
B (2016-01-18 10:58:47)	31,103	49	36	13	173	155	18	362	343	19	644	623	21	1053	1028	25
C (2017-03-13 13:49:32)	106	11	9	2	12	10	2	24	21	3	24	21	3	39	36	3
D (2016-08-02 08:16:28)	12,672	38	15	23	100	65	35	159	119	40	244	192	52	327	270	57
E (2016-08-02 07:56:28)	12,118	2	0	2	7	0	7	15	4	11	31	12	19	51	29	22
F (2016-11-18 05:06:18)	232	8	0	8	13	1	12	15	1	14	19	2	17	24	7	17
G (2017-06-25 05:39:05)	2	2	0	2	2	0	2	2	0	2	2	0	2	2	0	2
H (2016-08-02 08:06:28)	568	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0
I (2016-10-17 05:26:56)	10	4	2	2	4	3	1	4	4	0	4	4	0	5	5	0
unfixed	98	19	7	12	25	11	14	28	12	16	28	12	16	28	12	16
unknown	4	4	0	4	4	0	4	4	0	4	4	0	4	4	0	4

bugs in *cxxfilt*, we would claim to have discovered two orders of magnitude more bugs than we actually did. On the other hand, stack hashing seems to do very well for label H: one hash matched all 568 inputs for all values of N. In sum, across all runs, for $N = 3$, 595 hashes corresponded to 9 bugs, an inflation of 66×, as compared to 57,044 coverage profile-unique inputs for 9 bugs, an inflation of 6339×.[6] 4 crashing inputs were each associated with their own "fixing" commit, but when we inspected the respective code changes we could not see why the changes should fix a crash. As such, we have listed these inputs in Table 6.1 as "unknown." ASAN/UBSAN does not detect all possible undefined behaviors, so it may be that a code or data layout change between compilations or some other form of non-determinism is suppressing the crashing behavior.

While stack hashing does not overcount bugs nearly as much as AFL coverage profiles, it has the serious problem that hashes are not unique. For example, in the $N = 3$ case, only 343 of those for label B matched *only* inputs associated with B (TP column). The remaining 19 *also* matched some other crashing input (FP column). As such, these other inputs would be wrongly discarded if stack hashing had been used for de-duplication. Indeed, for label G, there is no unique hash (there is a 0 in the TP column)—it only falsely matches. Overall, for $N = 3$, about 16% of hashes were non-unique.[7] As such, stack hashing-based deduplication would have

[6]The table tabulates crashing inputs across all trials put together: if instead you consider the stack hashes taken on a per-run basis (as in Figure 6.1), the results will be somewhat different, but the overall trends should remain the same.

[7]This value was computed by summing the total distinct number of hashes that show up in more than one row (a lower bound of the total in column 4) and dividing by the total of distinct

discarded these bugs.

Discussion

Table 6.1 shows another interesting trend also evident, but less precisely, in Figure 6.1. Some bugs are triggered by a very small number of inputs, while others by a very large number. Bugs G and I each correspond to only 2 or 10 inputs, while bugs B, D, and E correspond to more than 10K inputs. Prior fuzzing studies have found similar bug distributions [206]. While Table 6.1 combines all inputs from all trials, considering each trial individually (as per Figure 6.1) we find that no single run found all 9 bugs; all runs found bugs B, D, E, but no run found more than 5 additional bugs.

From our analysis of *cxxfilt*, we draw two tentative conclusions. First, the trends reinforce the problem with bug heuristics: in the presence of "rare" inputs, the difference between finding 100 crashing inputs and 101 (an apparently insignificant difference) could represent finding 1 or 2 unique bugs (a significant one). Second, fuzzers might benefit from an algorithmic trick employed by SAT solvers: randomly "reboot" the search process [204] by discarding some of the current state and starting again with the initial seed, thus simulating the effect of running separate trials. The challenge would be to figure out what fuzzer state to retain across reboots so as to retain important knowledge but avoid getting stuck in a local minimum.

hashes overall (a lower bound of the total in column 2).

Related work

Recent work by [161] also experimentally assesses the efficacy of stack hashing and coverage profiles against ground truth. Like us, they defined ground truth as single conceptual bugs corrected by a particular code patch. They compared how well coverage profiles and stack hashes approximate this ground truth. Like us, they found that both tended to overcount the number of true bugs. As they consider different patches and target programs, their study is complementary to ours. However, their set of crashing inputs was generated via mutations to an initial known crashing input, rather than via a normal fuzzing process. As such, their numbers do not characterize the impact of poor deduplication strategies in typical fuzzing use-cases, as ours do.

[162] also studied how stack hashes, for $N = 1$ and $N = \infty$, can over- and under-count bugs identified through symbolic execution. Their interest was a comparison against their own de-duplication technique, and so their study did not comprehensively consider ground truth.

6.4 Approximating ground truth: Clustering

Another approach to input clustering and root cause analysis is to use coverage information generated by AFL as feature vectors used by more standard clustering algorithms. Both stack hashing and faulting instruction methods can be viewed as clustering on either a faulting-context-sensitive or single variable feature, respectively. This leads us to a hypothesis: would a more precise representation of the

programs behavior while processing a crashing input result in a feature vector that could be used effectively to discriminate between different bugs? Specifically, could we represent the programs behavior as a per-statement count of the number of times that statement had been executed?

AFL uses reinforcement learning to guide fuzzing and the generation of new inputs. Specifically, AFL mutates an existing input, then runs the fuzzed program with that mutation as input and measures how frequently each statement in the fuzzed program is executed. This measurement is scaled using a bucketing strategy: counts are only reported at the granularity of 1, 2, 3, $[4, 7]$, $[8, 15]$, $[16, 31]$, $[32, 127]$, and $[128, \text{inf}]$. This is done for performance reasons, but the choice of buckets is driven by an intuition about what is interesting program behavior: a statement changing from being visited 0 to 1 times is interesting, but probably a change from 17 to 21 is not interesting.

This analysis and instrumentation is done on-line while AFL is fuzzing, however AFL makes available a utility to view a single measurement for a single input. This utility, `afl-showmap`, outputs a map from program points to the bucketed counts of how often that program point was visited during the run with the single input. We can represent this map as a feature vector. Even though this feature vector is collapsing some features due to bucketing, that collapsing seems to work when doing fuzzing, so perhaps it will still work when doing clustering.

We created a software infrastructure to perform feature extraction, clustering and evaluation as a single step. This pipeline is a composition of several tools, implemented in C, C++, Python (using `scikit-learn` [211]), and Bash. The ex-

perimental process is as follows:

1. First, we process meta data about the crashing inputs and ground truth data into a JSON file that lists the full path to each crashing input file along with the cluster (as a numeric value) that crashing input file is associated with. This step produces both the ground truth for the final comparison as well as identifies all of the crashing input files to use for feature vector extraction.

2. Next, we iterate over each crashing input file in the ground truth, producing a feature extraction command for each file using `afl-showmap`. We execute these commands using the GNU parallel [212] tool. Feature extraction commands output a JSON file representing a map from program points (given as hexadecimal encoded addresses) [8] to a coverage count, as an integer between 1 and 2^8.

3. Next, we reconcile every individual coverage map with each other so that they have the same key space. The coverage maps gathered during the previous steps map edge locations to counts, however if an edge location is not visited when producing that specific coverage map, it does not appear in the key set for that map. To reconcile this, we compute the union of the key space for every input map and then "pad out" every map to include the total key space, but with default values of 0 when that coverage location does not appear in a

[8] We made sure to disable Address Space Layout Randomization (ASLR) during this phase as it would have changed the location of program points between the execution of each feature extraction step.

specific map.

4. Next, we vectorize the maps. Each key becomes a column in a matrix with n columns for n observed features and m rows for m crashing input files we are testing.

5. This matrix is then given to a clustering algorithm. As output, this step produces a clustering assignment, which is recorded as a JSON map from crashing input files to assigned cluster. This step is iterated for each clustering algorithm and configuration we want to perform as part of the experiment.

6. Finally, each clustering assignment is evaluated against ground truth. This is done by converting both the ground truth and the clustering assignment into one dimensional vectors, and then using a comparison metric from `scikit-learn`. This step outputs tuples saved into a CSV file noting the metric score assigned to a given clustering produced by a cluster configuration.

6.4.1 Clustering Methods

We used several different clustering methods: k-means [213], spectral clustering [214], mean shift [215], agglomerative, and DBSCAN [216]. We chose k-means as a baseline for clustering. One weakness of k-means, spectral, and agglomerative clustering is that they must be parameterized by the number of clusters. In a realistic setting, this is not known a priori. We did know the number of clusters for our data, though, because we have ground truth. One hope would be that the non-parametric algorithms, mean shift and DBSCAN, perform with reasonable precision

and also discover a reasonable number of clusters.

- Mean shift clustering iteratively identifies regions of higher density in a set of samples. These regions of higher density become clusters. Mean shift has few parameters (specifically, the number of clusters is not an input parameter) and is supposed to make no assumptions about the "shape" of underlying data and use an arbitrary number of features.

- Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering also finds regions of higher density. Unlike Mean shift, DBSCAN performs best when there are clusters of similar density. It also does not require the number of clusters as a parameter.

- Agglomerative clustering is a form of hierarchical clustering that will recursively merge clusters to minimize the distance between those clusters. This does require the number of clusters as a parameter.

- Spectral clustering performs dimensionality reduction using a "similarity matrix" (the distance between points). We compute the similarity matrix by computing the cosine similarity between all input vectors (the normalized dot product). Spectral clustering does require the number of clusters as a parameter, but should perform well if the underlying clusters are non-convex. However, there are apparently limits to this, as we will see when we measure the performance of spectral clustering on our crash data.

In addition, we performed clustering using input minimization via *afl-tmin*.

This program performs "fuzzing in reverse", removing portions of the input that preserve either the crashing behavior of the input, or the coverage behavior of the input. In our setting, all inputs result in a crash, so the algorithm would perform mutations that delete or minimize elements of the input and test if that mutation still crashes. This clustering approach has risks. Specifically, it could be that this process "walks" in one step from a crash caused by one bug to a crash caused by a separate bug.

Clustering by input minimization makes an assumption that a particular root cause bug has a "canonical" input that triggers it. In the small, this makes some intuitive sense: in a piece of code that indexes into a buffer, there is a minimal value for the index that will demonstrate unsafe behavior. Though this pattern can exist for lots of other safety properties, it is difficult to argue that it must exist for every safety property. However, perhaps minimization could cut out a lot of "trivially equal" crashes in that if two inputs do reduce down to the same value where the faulting location is the same, then those do represent "the same bug" (with the caveat of the risk mentioned in the previous paragraph).

There are several different minimization tools. We also evaluated a tool released by Google Project Zero, *halfempty* [217]. This tool performs "pessimistic speculative execution" along with input reduction via bisection. At each bisection, it tests whether or not the resulting transformation still crashes. If a bisection fails to crash, subsequent bisections of that input are discarded. The exploration makes use of parallelism. We evaluated this tool head-to-head against *afl-tmin*. In our experiments, we found that using *halfempty* reduced our inputs to 23,144 unique

inputs. This is worse than using *afl-tmin*, which reduced the inputs to 12,931 unique inputs. Additionally, *halfempty* uses about 4 times on average the clock time that *afl-tmin* does, and much more CPU time, since *halfempty* uses as many cores as are available on the entire system while *afl-tmin* only uses one core at a time.

Assessing against ground truth

As output from the clustering, we produce a mapping from each crashing input file to a cluster. Our ground truth data (as previously described in Table 6.1) has the same mapping. To compare a proposed clustering with the ground truth, we convert both into a 1-dimensional vector where the value at the ith position in the vector gives the cluster for the ith value. These two vectors are then compared using the Adjusted Rand Index (ARI) [218] and the Fowlkes Mallows Index (FMI) [219].

Both ARI and FMI compute a value between 0 and 1 assessing the similarity of two clusterings. This similarity is not sensitive to permutations of labels. We compare each clustering produced to ground truth. Both ARI and FMI take as parameters four counts relating the clusterings they compare. For each pair of points and input clusters X_T (which could be thought of as ground truth) and X_P (which could be thought of as the , let:

- TP be the count of pairs where they are in the same cluster in both X_T and X_P
- FP be the count of pairs that are in the same cluster in X_T but not in the same cluster in X_P

- FN be the count of pairs that are not in the same cluster in X_T but are in the same cluster as X_P
- TN be the count of pairs that are not in the same cluster in both X_T and X_P

We can then define FMI as:

$$\frac{TP}{\sqrt{(TP+FP)(TP+FN)}}$$

The Rand index (un-adjusted) [220] is defined as:

$$\frac{TP+TN}{TP+FP+FN+TN}$$

A weakness of the Rand index is that if you compare two absolutely unrelated clusters, the RI will still approach 1. To fix this, the Adjusted Rand Index (ARI) [218] was proposed. ARI "controls for chance" by correcting the RI according to a random assignment of elements between the union of cluster labels in the two compared clusters. To do this, it computes a "contingency table" with height and width equal to the number of clusters in the first and second parameters to ARI, respectively. Then, the value n_{ij} is the number of times an element occurs in the i-th cluster of the first clustering and the j-th cluster of the second cluster. Finally, there are also "row sums" and "column sums" denoted a_i and b_j, respectively. These are the sums of the co-occurrences across the rows and the columns of the contingency table

The Adjusted Rand index is then defined as:

Clustering Method	Cluster parameter	PCA	Clusters found	ARI	FMI
afl-tmin	-	-	12931	0.04	0.18
Stack hashing ($N = 1$)	-	-	99	0.7	0.82
Stack hashing ($N = 2$)	-	-	289	0.41	0.61
Stack hashing ($N = 3$)	-	-	556	0.24	0.45
Stack hashing ($N = 4$)	-	-	932	0.2	0.42
Stack hashing ($N = 5$)	-	-	1461	0.11	0.3
K-Means	9	-	-	0.19	0.42
	9	0.95	-	0.19	0.42
Spectral	9	-	-	0.03	0.36
	9	0.95	-	0.03	0.27
Mean shift	-	0.95	488	0.18	0.61
Agglomerative	9	-	-	0.15	0.39
DBSCAN	-	-	2	0	0.62

Table 6.2: Results for cxxfilt clustering. When a clustering algorithm was given a number of clusters as an explicit parameter, that is listed under "Cluster parameter", while if a clustering algorithm discovered the number of clusters for itself, that is reported under "Clusters found".

$$\frac{\sum_{ij}\binom{n_{ij}}{2} - [\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}{\frac{1}{2}[\sum_i(\binom{a_i}{2}) + \sum_j(\binom{b_j}{2})] - [\sum_i(\binom{a_i}{2}) \sum_j(\binom{b_j}{2})]/\binom{n}{2}}$$

We also perform a dimension reduction, in some cases, using Principal Component Analysis (PCA) [221]. PCA identifies columns with dependence and replaces them with a single column. We used PCA for dimensionality reduction because the feature vectors represent program coverage, and due to dominance in the control flow graph, some edges would be absolutely correlated with other edges.

6.4.2 Results and discussion

The overall results are given in Table 6.2. In addition to the clustering methods that take features from AFL as input, we also report on the number of clusters found, ARI, and FMI for stack hashing at different levels of context sensitivity as well as using *afl-tmin* to reduce inputs and cluster together inputs that reduce to the same input value as determined using SHA1.

It's difficult to interpret an ARI or FMI score against ground truth directly. This is why we include scores (and clusters found) for stack hashing at different levels of sensitivity. In the previous chapter, we saw that stack hashing with $K = 3$ did not provide good clustering when we considered false and true positives against ground truth. By computing the ARI and FMI for that approach, we can have a functional comparison through the lens of ARI and FMI of the clustering based approaches.

Unfortunately, the results are not favorable for any of the non-parametric clustering approaches. Mean shift identifies slightly fewer clusters than $N = 3$ stack hashing, but with a lower accuracy as measured by ARI (but a higher accuracy as measured by FMI, most likely because FMI does not consider true negatives). These clusters do not exhibit the same "shape" as the clusters in the ground truth, i.e. they are all of roughly equal size. This most likely explains the difference between ARI and FMI as they weight different types of correct or incorrect classification differently. Spectral and agglomerative clustering also performs quite poorly. DBSCAN essentially fails entirely, it identifies 2 clusters with only a few elements and

discards the rest of the input vectors as "noise." Likewise, performing input reduction via *afl-tmin* performs poorly, identifying two orders of magnitude more clusters than even $N = 3$ stack hashing.

It is possible that there is just not enough signal in the coverage count tuples generated by AFL to perform accurate clustering. Other work on using traces to cluster inputs to find bugs [222] uses *predicates* from the program that describe relationships between values at different program points in addition to coverage of program points by test cases. Perhaps this information should be incorporate as well for more accurate clustering to occur.

6.5 Using symbolic path conditions for root cause analysis

A stack backtrace contains only information about the faulting context, and a "fuzzy stack hash" contains even less information, with the context sensitivity arbitrarily chopped off. In exchange for dropping this information, it is efficient to both produce and store these abstractions of program state at the site of the crash. We asked: what is the *most* information that we might gather? Our answer there is a symbolic path condition produced by a symbolic or concolic execution of the program with the crashing input as a concrete input. This symbolic path condition would describe the choices made by the program in response to the input and the whole "path" of the program from start to crash. Some prior work uses this information to "bucket" failing test cases into test cases that likely have the same root cause [223].

We used the KLEE [224] virtual machine to extract the path condition created by running *cxxfilt* with a concrete input that crashes the program. We needed to make some small changes to KLEE to support this. Specifically, KLEE supports a concept of "replay mode" – given an initial concrete input ("seed"), KLEE will symbolically execute but discard any symbolic states where the initial seed is not concretely realizable. We changed this mode to make it stronger: when a symbolic input value is concretized, we temporarily constrain all inputs to be exactly equal to the concrete seed under replay, and we relax those constraints after concretization.

This allows us to gather path constraints for many inputs, though KLEE is slow and sometimes times out even when in "replay mode". We use these path conditions to help explain, in some cases, what "must be true" about a particular input when doing root cause analysis. In cases where this "replay mode" gave us a path condition, we were able to use Z3 [225] to solve for a model for the path condition, producing a new input. This input was approximately half the size of the starting input. However, delta debugging techniques like *afl-tmin* can achieve a much higher reduction, as much as 90%, through brute force. Due to the computational cost of using KLEE to gather path conditions, this information was not directly exploited when performing our root cause analysis. We explore the power of *afl-tmin* to perform clustering via reduction in Section 6.4.1.

6.6 Conclusion

In Chapter 5, we focused on measuring fuzzer success using *"unique" crashes found*, which is to say, inputs that induce a "unique" crash. As crashes are symptoms of potentially serious bugs, measuring a fuzzer according to the number of crashing inputs it produces seems natural. However, as we explore in Chapter 6, bugs and crashing inputs are not the same thing: many different inputs could trigger the same bug. For example, a buffer overrun will likely cause a crash no matter what the input data consists of so long as that data's size exceeds the buffer's length. Simply counting crashing inputs as a measure of performance could be misleading: fuzzer A could find more crashes than fuzzer B but find the same or fewer actual bugs.

Much prior work (some outlined in Chapter 5) employs some strategy to *de-duplicate* (or *triage*) crashes, so as to map them to unique bugs. There are two popular automated heuristics for doing this: using AFL's notion of *coverage profile*, and using *stack hashes*. Unfortunately, as we show experimentally in this section, these de-duplication heuristics are actually poor at clustering crashing inputs according to their root cause.

Despite the steps we took to ensure our triage matches ground truth, we may still have inflated or reduced actual bug counts. As an example of the former, we note that ASAN/UBSAN is not guaranteed to catch all memory safety violations, so we may have attributed an incidental change to a bugfix. We found a few cases (4 of $\tilde{5}$7,000) where we couldn't explain why a commit fixed a crash, and so did

not associate the commit with a bug. On the other hand, we might have counted multiple (on the order of 3 or 4) fixes as a single fix. In any case, the magnitude of the difference between our counts and "unique crashes" means that the top-level result—that "unique crashes" massively overcount the number of true bugs—would hold even if the counts changed a little.

Had ground truth been available in all of our experiments in Chapter 5, it might have changed the character of the results in Sections 5.4–5.6. For example, the performance variations within a configuration due to randomness (e.g., Figures 5.2, 5.3, 5.4, 5.5, and 5.6) may not be as stark when counting bugs rather than "unique" crashing inputs. In this case, our advice of carrying out multiple trials is even more important, as small performance differences between fuzzers A and B may require many trials to discern. It may be that performance differences due to varying a seed (Figures 5.7, 5.8, 5.9, 5.10, 5.11, and 5.12) may also not be as stark—this would be true if one seed found hundreds of crashes and another found far fewer, but in the end all crashes corresponded to the same bug. There may also be less performance variation over time when bugs, rather than crashes, are counted (Figure 5.13). On the other hand, it is also possible that we would find *more* variation over time, and/or with different seeds, rather than less. In either case, our results in Sections 5.4–5.6 raise sufficient concern that our advice to test with longer timeouts and a variety of seeds (including the empty seed) should be followed unless and until experimental results with ground truth data shed more light on the situation.

6.6.1 Future work

Existing methods for measuring the success of a fuzzing campaign are flawed, but it is difficult to come up with better automated techniques. With developer involvement, we can associate crashes with patches that fix the crash, however that both requires either history or developer experience, and also developers might not fully fix a problem, as we saw with the different attempts to fix recursion errors in *cxxfilt*. "What is a bug" remains an open and thorny question.

When considering history, one problem is the relationship between individual changes to a program that happen in a single patch. Perhaps one option would be to develop a source-aware tool that tries to automatically separate or break down larger patches into smaller ones and evaluates their effect on "fixing" crashing inputs individually.

Our investigation of using coverage as features for clustering did not pan out, however this is supported by prior work which used both coverage and predicates as features for clustering. Perhaps this is the reason they did not use coverage alone to cluster. In future work, we could use further predicates such as relationships between parameters and relationships between returned scalar values and 0 as features for clustering.

It is also possible that with more work on scaling concolic execution, full path conditions could be extracted and used for clustering. Exactly what features to use is open. There is also additional work that could be done to use symbolic execution for input minimization. Additionally, the symbolic executor could be used to keep

the minimization done by *afl-tmin* "on track" and hedge the risk of "walking" from one bug to another during minimization.

Chapter 7: Conclusion

Software security remains difficult. Each of the approaches and techniques considered in this body of work have their own share of weaknesses and areas for future improvement. It is both fortunate and unfortunate that these problems will remain with us, as it will give us many things to do at the expense of problems with our technological systems.

Build It, Break It, Fix It

Chapter 2 presents Build It Break It Fix it, a contest for secure software development. We ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the best performing build-it submissions used C/C++, but submissions coded in a statically-typed language were less likely to have a security flaw; Build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful Build-it teams were significantly better at finding security bugs.

Checked C

Chapter 3 sketches out Checked C, an attempt to make C safer. Checked C's design is focused on interoperability with legacy C, usability, and efficiency. Checked C's novel notion of *checked regions* ensures that "checked code cannot be blamed" for a safety violation. Our implementation of Checked C as a Clang/LLVM extension enjoys good performance. To assist in incrementally strengthening legacy code, we have developed a porting tool for automatically rewriting code to use checked pointers. We evaluate that program against open source code.

Managing disjuncts

In Chapter 4, we establish a number of new affinity scoring algorithms for determining which disjuncts should be merged in a disjunctive abstraction. The new affinity scoring algorithms are all based on points within the polytopes, either exactly or approximately. Those points are either sampled or counted in order to compute proxies for polytope volume. We demonstrated that these techniques work by implementing a new abstract domain in an existing static analyzer and analyzing a large selection of benchmark programs.

Evaluating fuzz testing

In Chapter 5, we consider the evaluation of fuzz testing. After a study of prior papers, we find methodological flaws in their evaluation that cast doubt on their reported effectiveness. We conduct our own experiments to show the effect that

these flaws could have, and create a list of guidelines for evaluating fuzz testing.

Root cause analysis

Chapter 6 shows the perils in determining ground truth for the number of bugs discovered in a fuzzing campaign. One successful strategy is employed to get to the bottom of our experiments from Chapter 5, but it relies on having historical data and fuzzing old versions of a program. Chapter 6 explores methods to use clustering to arrive at similar ground truth data and fails. Chapter 6 also shows the problems with using AFL coverage maps and stack hashing to approximate ground truth.

Appendix A: Patches applied to *cxxfilt*

```
static struct demangle_component *
-d_identifier (struct d_info *di, int len)
+d_identifier (struct d_info *di, long len)
```

Figure A.1: `int`/`long` patch

```
@@ -3174,6 +3204,8 @@ d_expression_1 (struct d_info *di)
       struct demangle_component *type = NULL;
       if (peek == 't')
        type = cplus_demangle_type (di);
+      if (!d_peek_next_char (di))
+       return NULL;
       d_advance (di, 2);
       return d_make_comp (di,
          DEMANGLE_COMPONENT_INITIALIZER_LIST,
                           type, d_exprlist (di, 'E'));
@@ -3248,6 +3280,8 @@ d_expression_1 (struct d_info *di)
             struct demangle_component *left;
             struct demangle_component *right;

+            if (code == NULL)
+              return NULL;
             if (op_is_new_cast (op))
               left = cplus_demangle_type (di);
             else
@@ -3275,7 +3309,9 @@ d_expression_1 (struct d_info *di)
             struct demangle_component *second;
             struct demangle_component *third;

-            if (!strcmp (code, "qu"))
+            if (code == NULL)
+              return NULL;
+            else if (!strcmp (code, "qu"))

@@ -4439,10 +4481,16 @@ d_print_comp_inner (struct
   d_print_info *dpi, int options,
             local_name = d_right (typed_name);
             if (local_name->type ==
                DEMANGLE_COMPONENT_DEFAULT_ARG)
               local_name = local_name->u.s_unary_num.sub;
+            if (local_name == NULL)
+              {
+                d_print_error (dpi);
+                return;
+              }
```

Figure A.2: NULL check patch

```
@@ -3345,6 +3415,8 @@ d_expression_1 (struct d_info *di)
                 first = d_expression_1 (di);
                 second = d_expression_1 (di);
                 third = d_expression_1 (di);
+                if (third == NULL)
+                  return NULL;
               }
             else if (code[0] == 'f')
               {
@@ -3352,6 +3424,8 @@ d_expression_1 (struct d_info *di)
                 first = d_operator_name (di);
                 second = d_expression_1 (di);
                 third = d_expression_1 (di);
+                if (third == NULL)
+                  return NULL;
               }
```

Figure A.3: Patch adding `NULL` checks

```
@@ -1654,12 +1697,13 @@ demangle_signature (struct work_stuff
    *work,
                                                    0);
               if (!(work->constructor & 1))
                 expect_return_type = 1;
-              (*mangled)++;
+              if (!**mangled)
+                success = 0;
+              else
+                (*mangled)++;
               break;
             }



@@ -2135,6 +2179,8 @@ demangle_template (struct work_stuff *
   work, const char **mangled,
         {
           int idx;
           (*mangled)++;
+          if (**mangled == '
0')
+            return (0);
           (*mangled)++;
```

Figure A.4: Patch checking for exit conditions

```
 static void
 d_print_comp (struct d_print_info *dpi, int options,
-              const struct demangle_component *dc)
+              struct demangle_component *dc)
 {
   struct d_component_stack self;
+  if (dc == NULL || dc->d_printing > 1)
+    {
+      d_print_error (dpi);
+      return;
+    }
+  else
+    dc->d_printing++;

   self.dc = dc;
   self.parent = dpi->component_stack;
   d_print_comp_inner (dpi, options, dc);

   dpi->component_stack = self.parent;
+  dc->d_printing--;
 }
```

Figure A.5: Patch attempting to control unbounded recursion

```
@@ -144,6 +144,9 @@ struct work_stuff
   string* previous_argument; /* The last function argument
      demangled.  */
   int nrepeats;          /* The number of times to repeat the
       previous argument.  */
+  int *proctypevec;
+  int proctypevec_size;
+  int nproctypes;
 };
```

Figure A.6: Addition of new fields

```
@@ -4526,10 +4591,13 @@ demangle_args (struct work_stuff *
   work, const char **mangled,
                 {
                   string_append (declp, ", ");
                 }
+              push_processed_type (work, t);
               if (!do_arg (work, &tem, &arg))
                 {
+                  pop_processed_type (work);
                   return (0);
                 }
+              pop_processed_type (work);
               if (PRINT_ARG_TYPES)
                 {
                   string_appends (declp, &arg);
```

Figure A.7: Using new functions

```
@@ -3627,8 +3647,15 @@ do_type (struct work_stuff *work,
   const char **mangled, string *result)
               success = 0;
             }
           else
-            {
-              remembered_type = work -> typevec[n];
+            for (i = 0; i < work->nproctypes; i++)
+              if (work -> proctypevec [i] == n)
+                success = 0;
+
+          if (success)
+            {
+              is_proctypevec = 1;
+              push_processed_type (work, n);
+              remembered_type = work->typevec[n];
               mangled = &remembered_type;
             }
           break;
```

Figure A.8: Checks in `do_type`

```
@@ -5691,13 +5694,14 @@ d_print_comp (struct d_print_info *
   dpi, int options,
              struct demangle_component *dc)
 {
   struct d_component_stack self;
-  if (dc == NULL || dc->d_printing > 1)
+  if (dc == NULL || dc->d_printing > 1 || dpi->recursion >
   MAX_RECURSION_COUNT)
     {
       d_print_error (dpi);
       return;
     }
-  else
-    dc->d_printing++;
+
+  dc->d_printing++;
+  dpi->recursion++;
```

Figure A.9: Additional recursion patch

```
@@ -1244,11 +1244,13 @@ squangle_mop_up (struct work_stuff *
   work)
     {
       free ((char *) work -> btypevec);
       work->btypevec = NULL;
+       work->bsize = 0;
     }
   if (work -> ktypevec != NULL)
     {
       free ((char *) work -> ktypevec);
       work->ktypevec = NULL;
+       work->ksize = 0;
     }
 }
```

Figure A.10: Set field to 0 to avoid use after free

```
          success = 1;
          break;
        }
+       else if (n == -1)
+         {
+           success = 0;
+           break;
+         }
```

Figure A.11: Check of return value for failure

```
@@ -2051,7 +2051,8 @@ demangle_template_value_parm (struct
   work_stuff *work, const char **mangled,
       else
        {
          int symbol_len  = consume_count (mangled);
-          if (symbol_len == -1)
+          if (symbol_len == -1
+              || symbol_len > (long) strlen (*mangled))
            return -1;
          if (symbol_len == 0)
            string_appendn (s, "0", 1);
@@ -3611,7 +3612,7 @@ do_type (struct work_stuff *work, const
    char **mangled, string *result)
       /* A back reference to a previously seen type */
       case 'T':
         (*mangled)++;
-          if (!get_count (mangled, &n) || n >= work -> ntypes)
+          if (!get_count (mangled, &n) || n < 0 || n >= work -> ntypes)
            {
              success = 0;
            }
@@ -3789,7 +3790,7 @@ do_type (struct work_stuff *work, const
    char **mangled, string *result)
     /* A back reference to a previously seen squangled type
        */
     case 'B':
       (*mangled)++;
-       if (!get_count (mangled, &n) || n >= work -> numb)
+       if (!get_count (mangled, &n) || n < 0 || n >= work -> numb)
         success = 0;
       else
         string_append (result, work->btypevec[n]);
@@ -4130,7 +4131,8 @@ do_hpacc_template_literal (struct
   work_stuff *work, const char **mangled,

   literal_len = consume_count (mangled);

-   if (literal_len <= 0)
+   if (literal_len <= 0
+       || literal_len > (long) strlen (*mangled))
     return 0;
```

Figure A.12: Additional boundary tests

Bibliography

[1] Equifax 2017 security incident important information. `https://www.equifaxsecurity2017.com/frequently-asked-questions/`. Accessed: 2018-10-10.

[2] CVE-2017-5638. Available from MITRE, CVE-ID CVE-2017-5638., 2017.

[3] CVE-2014-0150. Available from MITRE, CVE-ID CVE-2014-1060., 2014.

[4] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2016.

[5] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked c: Making c safe by extension. In *Proceedings of the IEEE Conference on Secure Development (SecDev)*, September 2018.

[6] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2017.

[7] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in c programs. In *International Static Analysis Symposium*, pages 243–262. Springer, 2018.

[8] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. Evaluating design tradeoffs in numeric static analysis for java. In *Proceedings of the European Symposium on Programming (ESOP)*, April 2018.

[9] Andrew Ruef, Kesha Hietala, and Arlen Cox. Volume-based merge heuristics for disjunctive numeric domains. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, pages 383–401, 2018.

[10] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1), 2007.

[11] Jorge A. Navas. CRAB: A language-agnostic library for static analysis, 2018.

[12] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[13] Maryland cyber challenge & competition. `http://www.fbcinc.com/e/cybermdconference/competitorinfo.aspx`.

[14] National Collegiate Cyber Defense Competition. `http://www.nationalccdc.org`.

[15] DEF CON Communications, Inc. Capture the flag archive. `https://www.defcon.org/html/links/dc-ctf.html`.

[16] Polytechnic Institute of New York University. Csaw - cybersecurity competition 2012. `http://www.poly.edu/csaw2012/csaw-CTF`.

[17] dragostech.com inc. Cansecwest applied security conference. `http://cansecwest.com`.

[18] Top coder competitions. `http://apps.topcoder.com/wiki/display/tc/Algorithm+Overview`.

[19] The ACM-ICPC International Collegiate Programming Contest. `http://icpc.baylor.edu`.

[20] ICFP programming contest. `http://icfpcontest.org`.

[21] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[22] Queena Kim. Want to learn cybersecurity? Head to Def Con. `http://www.marketplace.org/2014/08/25/tech/want-learn-cybersecurity-head-def-con`, 2014.

[23] Git – distributed version control management system. `http://git-scm.com`.

[24] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40, 2009.

[25] Úlfar Erlingsson. personal communication stating that CFI was not deployed at Microsoft due to its overhead exceeding 10%, 2012.

[26] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel. Build it break it: Measuring and comparing development security. In *Workshop on Cyber Security Experimentation and Test (CSET)*, 2015.

[27] Yesod web framework for Haskell. `http://www.yesodweb.com`.

[28] PostgreSQL. `http://www.postgresql.org`.

[29] James Parker. LMonad: Information flow control for Haskell web applications. Master's thesis, Dept of Computer Science, the University of Maryland, 2014.

[30] Deian Stefan, Alejandro Russo, John Mitchell, and David Mazieres. Flexible dynamic information flow control in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2011.

[31] Kenneth P Burnham, David R Anderson, and Kathryn P Huyvaert. AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. *Behavioral Ecology and Sociobiology*, 65(1):23–35, 2011.

[32] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[33] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, 2012.

[34] DEF CON Communications Inc. Def con hacking conference. `http://www.defcon.org`.

[35] Nicholas Childers, Bryce Boe, Lorenzo Cavallaro, Ludovico Cavedon, Marco Cova, Manuel Egele, and Giovanni Vigna. Organizing large scale hacking competitions. In *DIMVA*, 2010.

[36] Adam Doupé, Manuel Egele, Benjamin Caillat, Gianluca Stringhini, Gorkem Yakin, Ali Zand, Ludovico Cavedon, and Giovanni Vigna. Hit 'em where it hurts: A live security exercise on cyber situational awareness. 2011.

[37] Peter Chapman, Jonathan Burket, and David Brumley. PicoCTF: A game-based computer security competition for high school students. In *USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE)*, 2014.

[38] Gregory Conti, Thomas Babbitt, and John Nelson. Hacking competitions and their untapped potential for security education. *Security & Privacy*, 9(3):56–59, 2011.

[39] Chris Eagle. Computer security competitions: Expanding educational outcomes. *IEEE Security & Privacy*, 11(4):69–71, 2013.

[40] Lance J. Hoffman, Tim Rosenberg, and Ronald Dodge. Exploring a national cybersecurity exercise for universities. *IEEE Security & Privacy*, 3(5):27–33, 2005.

[41] Art Conklin. Cyber defense competitions and information security education: An active learning solution for a capstone course. 2006.

[42] Art Conklin. The use of a collegiate cyber defense competition in information security education. In *Information Security Curriculum Development Conference (InfoSecCD)*, 2005.

[43] Google code jam. `http://code.google.com/codejam`.

[44] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *USENIX Conference on Web Application Development (WebApps)*, 2011.

[45] L. Prechelt. Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, 37(1):95–108, 2011.

[46] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2013.

[47] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *IEEE International Symposium on Reliability Engineering (ISSRE)*, 2013.

[48] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *IEEE International Symposium on Software Reliability Engineering*, 2014.

[49] Joonseok Yang, Duksan Ryu, and Jongmoon Baik. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *International Conference on Big Data and Smart Computing (BigComp)*, 2016.

[50] Keith Harrison and Gregory White. An empirical study on the effectiveness of common security measures. In *Hawaii International Conference on System Sciences (HICSS)*, 2010.

[51] NIST vulnerability database. `https://nvd.nist.gov`. Accessed May 17, 2017.

[52] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[53] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, 2002. USENIX.

[54] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*, Seattle, Washington, 2006. USENIX Association.

[55] Rust-lang.org. Rust documentation. `https://www.rust-lang.org/documentation.html`, 2016. Accessed May 13, 2016.

[56] golang.org. The go programming language. `https://golang.org/`, 2016. Accessed May 13, 2016.

[57] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[58] Mathworks. Polyspace code prover: prove the absence of run-time errors in software. `http://www.mathworks.com/products/polyspace-code-prover/index.html`, 2016. Accessed May 12, 2016.

[59] AbsOmt. Astrée: Fast and sound runtime error analysis. `http://www.absint.com/astree/index.htm`, 2016. Accessed May 12, 2016.

[60] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS*, 2004. W⊕X protection is discussed in Section 1.1.

[61] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[62] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[63] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[64] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[65] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. 2006.

[66] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *POPL*, 2007.

[67] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[68] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, March 1995.

[69] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, June 1994.

[70] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of European Symposium on Programming (ESOP '07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535, Heidelberg, 2007. Springer-Verlag.

[71] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM, 2002.

[72] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming (SCP)*, 62(2):122–144, 2006. Special issue on memory management. Expands ISMM conference paper of the same name.

[73] dlang.org. D. `http://dlang.org/`, 2016. Accessed May 13, 2016.

[74] Microsoft Corporation. C# programming guide. `https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx`, 2016. Accessed May 13, 2016.

[75] Samuel C. Kendall. Bcc: runtime checking for C programs. In *USENIX Toronto 1983 Summer Conference*, Berkeley, CA, USA, 1983. USENIX Association.

[76] Joseph L. Steffen. Adding run-time checking to the Portable C Compiler. *Softw. Pract. Exper.*, 22(4):305–316, April 1992.

[77] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[78] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, 1992. USENIX Association.

[79] Inc. Unicom Systems. Purifyplus. `http://unicomsi.com/products/purifyplus/`, 2016. Accessed May 6, 2016.

[80] MicroFocus. Devpartner. `http://www.borland.com/en-GB/Products/Software-Testing/Automated-Testing/Devpartner-Studio`, 2016. Accessed May 6, 2016.

[81] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Dr. Memory. Dr. Memory: Memory debugger for Windows, Linux, and Mac. `http://www.drmemory.org/`, 2016. Accessed May 6, 2016.

[83] Intel Corporation. Intel inspector. `https://software.intel.com/en-us/intel-inspector-xe`, 2016. Accessed May 6, 2016.

[84] Oracle Corporation. Oracle solaris studio. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`, 2016. Accessed May 6, 2016.

[85] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[86] Valgrind. Valgrind. `http://valgrind.org/`, 2016. Accessed May 6, 2016.

[87] Parasoft. Memory error detection. `https://www.parasoft.com/capability/memory-error-detection/`, 2016. Accessed May 6, 2016.

[88] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[89] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In Miriam Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*, Linkoping Electronic Conference Proceedings. Linkoping University Electronic Press, May 1997. `"http://www.ep.liu.se/ea/cis/1997/009/"`.

[90] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice & Experience*, 27(1):87–110, January 1997.

[91] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, Reston, VA, USA, 2004. Internet Society. `http://www.internetsociety.org/doc/practical-dynamic-buffer-overflow-detector`.

[92] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[93] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. ACM.

[94] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM.

[95] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[96] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[97] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, July 2008.

[98] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38(5):196–207, May 2003.

[99] David Delmas and Jean Souyris. Astrée: From research to industry. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 437–451, Berlin, Heidelberg, 2007. Springer-Verlag.

[100] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[101] Robert Wahbe, Steven Lucco, Thoma E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[102] PaX Team. `http://pax.grsecurity.net/docs/aslr.txt`, 2001.

[103] Wikipedia. Address space layout randomization. `https://en.wikipedia.org/wiki/Address_space_layout_randomization`, 2016. Accessed April 25, 2016.

[104] Crispin Cowan, Calton Pu, Dave Maiere, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[105] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[106] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[107] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[108] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[109] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *ISOP*, 1976.

[110] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1), 2006.

[111] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[112] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, 2006.

[113] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, 2006.

[114] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[115] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *(TOPLAS)*, 29(5):26, 2007.

[116] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *Comput. Geom.*, 43(5):453–473, 2010.

[117] Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS*, 2010.

[118] Pascal Sotin, Bertrand Jeannet, Franck Védrine, and Eric Goubault. Policy iteration within logico-numerical abstract domains. In *ATVA*, 2011.

[119] Alexander I Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4), 1994.

[120] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[121] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *SAS*, 2012.

[122] Igor Pak and Greta Panova. On the complexity of computing kronecker coefficients. *computational complexity*, 26(1), 2017.

[123] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report, Université Louis Pasteur de Strasbourg, 1999.

[124] Martin Dyer, Alan Frieze, and Ravi Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. *J. ACM*, 38(1):1–17, January 1991.

[125] Ravi Kannan, László Lovász, and Miklós Simonovits. Random walks and an o*(n5) volume algorithm for convex bodies. *Random Structures & Algorithms*, 11(1):1–50, 1997.

[126] Dirk P. Kroese, Thomas Taimre, and Zdravko I. Botev. *Markov Chain Monte Carlo*, chapter 6, pages 225–280. John Wiley & Sons, Inc., 2011.

[127] Roberto Bagnara, Patricia M Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *SAS*, 2003.

[128] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, 2010.

[129] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, UIUC, 2005.

[130] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.

[131] R. L. Smith. Monte carlo procedures for generating random feasible solutions to mathematical programs. In *ORSA/TIMS Conference*, May 1980.

[132] A. Boneh and A. Golan. Constraints' redundancy and feasible region boundedness by random feasible point generator (rfpg). In *Third European congress on operations research, EURO III*, 1979.

[133] Martin Bromberger and Christoph Weidenbach. Computing a complete basis for equalities implied by a system of LRA constraints. In *SMT*, volume 1617, pages 15–30, 2016.

[134] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV, pages 81–94, 2006.

[135] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016.

[136] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[137] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[138] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1), 2003.

[139] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, 1998.

[140] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Refining the control structure of loops using static analysis. In *EMSOFT*, 2009.

[141] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, 2011.

[142] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, 2007.

[143] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.

[144] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *POPL*, 2002.

[145] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, 2001.

[146] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*. 2002.

[147] Ian Sweet, José Manuel Calderón Trilla, Chad Scherrer, Michael Hicks, and Stephen Magill. What's the over/under? probabilistic bounds on information leakage. In *POST*, 2018.

[148] American fuzzing lop (afl). `http://lcamtuf.coredump.cx/afl/`, 2018.

[149] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[150] R. Lyman Ott and Micheal T. Longnecker. *Introduction to Statistical Methods and Data Analysis (with CD-ROM)*. 2006.

[151] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[152] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[153] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[154] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[155] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[156] Bin Zhang, Jiaxi Ye, Chao Feng, and Chaojing Tang. S2F: discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing. In *International Conference on Computational Intelligence and Security*, 2017.

[157] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015.

[158] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *International Conference on Software Engineering (ICSE)*, 2014.

[159] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering (ICSE)*, 2014.

[160] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, 2009.

[161] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[162] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *International Conference on Fundeamental Approaches to Software Engineering (FASE)*, 2017.

[163] libfuzzer. `https://llvm.org/docs/LibFuzzer.html`, 2018.

[164] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[165] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[166] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[167] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. Quickfuzz: an automatic random fuzzer for common file formats. In *International Symposium on Haskell*, 2016.

[168] Gustavo Grieco, Martn Ceresa, Agustn Mista, and Pablo Buiras. Quickfuzz testing for fun and profit. *J. Syst. Softw.*, 2017.

[169] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: interface aware fuzzing for kernel drivers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[170] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[171] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Foundations of Software Engineering (FSE)*, 2017.

[172] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[173] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Foundations of Software Engineering (FSE)*, 2015.

[174] Ying-Dar Lin, Feng-Ze Liao, Shih-Kun Huang, and Yuan-Cheng Lai. Browser fuzzing by scheduled mutation and generation of document object models. In *International Carnahan Conference on Security Technology*, 2015.

[175] Hyunguk Yoo and Taeshik Shon. Grammar-based adaptive fuzzing: Evaluation on SCADA modbus protocol. In *IEEE International Conference on Smart Grid Communications*, 2016.

[176] HyungSeok Han and Sang Kil Cha. IMF: inferred model-based fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[177] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2017.

[178] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[179] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[180] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: targeted evolutionary fuzz testing of virtual devices. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[181] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[182] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[183] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: efficient domain-independent differential testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[184] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[185] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. 2018.

[186] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[187] Weiguang Wang, Hao Sun, and Qingkai Zeng. Seededfuzz: Selecting and generating seeds for directed fuzzing. In *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2016.

[188] L. Zhang and V. L. L. Thing. A hybrid symbolic execution assisted fuzzing method. In *IEEE Region 10 Conference (TENCON)*, 2017.

[189] Cert basic fuzzing framework (bff). `https://github.com/CERTCC/certfuzz`, 2018.

[190] Radamsa. https://github.com/aoh/radamsa, 2018.

[191] Zzuf. http://caca.zoy.org/wiki/zzuf, 2018.

[192] Confidence intervals for a median. http://www.ucl.ac.uk/ich/short-courses-events/about-stats-courses/stats-rm/Chapter_8_Content/confidence_interval_single_median, 2018.

[193] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering (ICSE)*, 2011.

[194] Gordon B. Drummond and Sarah L. Vowler. Different tests for a difference: how do we research? *British Journal of Pharmacology*, 165(5), 2012.

[195] Guillaume Calmettes, Gordon B. Drummond, and Sarah L. Vowler. Making due with what we have: use your bootstraps. *Journal of Physiology*, 590(15), 2012.

[196] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2), 2000.

[197] lcamtuf. AFL quick start guide. http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt, April 2018.

[198] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[199] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[200] Fuzzer test suite. https://github.com/google/fuzzer-test-suite, 2018.

[201] Darpa cyber grand challenge (cgc) binaries. https://github.com/CyberGrandChallenge/, 2018.

[202] Brendan Dolan-Gavitt. Of bugs and baselines. http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html, 2018.

[203] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas

VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[204] Vadim Ryvchin and Ofer Strichman. Local restarts. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2008.

[205] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, volume 48, pages 197–208. ACM, 2013.

[206] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[207] Michael Hicks. What is a bug? `http://www.pl-enthusiast.net/2015/09/08/what-is-a-bug/`, 2015.

[208] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[209] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[210] Ben Liblit and Alex Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley, October 2002.

[211] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[212] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[213] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, September 2006.

[214] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[215] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 24(5):603–619, 2002.

[216] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[217] Tavis Ormandy. halfempty. `https://github.com/googleprojectzero/halfempty`, 2018.

[218] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[219] Edward B Fowlkes and Colin L Mallows. A method for comparing two hierarchical clusterings. *Journal of the American statistical association*, 78(383):553–569, 1983.

[220] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[221] I. T. Jolliffe. *Principal Component Analysis and Factor Analysis*, pages 115–128. Springer New York, New York, NY, 1986.

[222] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 15–26, 2005.

[223] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, pages 43–59, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[224] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[225] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.