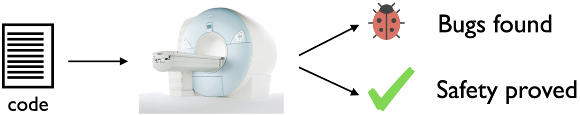
AAA616: Program Analysis

Lecture 1 — Introduction

Hakjoo Oh
2024 Fall

Program Analysis

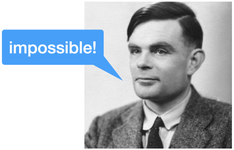
- Technology for catching bugs or proving correctness of software



- Widely used in software industry

A Hard Limit

- The Halting problem is not computable



Alan Turing (1936)

230 A. M. Turing [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

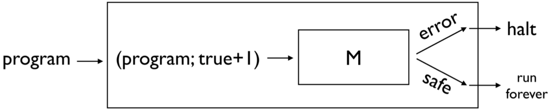
The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173-198.

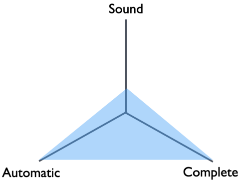
- If exact analysis is possible, we can solve the Halting problem

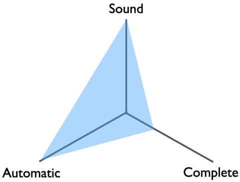


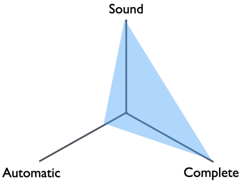
- Rice's theorem (1951): any non-trivial semantic property of a program is undecidable

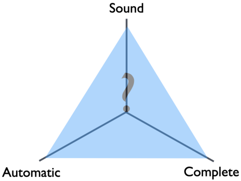
Tradeoff

- Three desirable properties
  - Soundness: all program behaviors are captured
  - Completeness: only program behaviors are captured
  - Automation: without human intervention
- Achieving all of them is generally infeasible







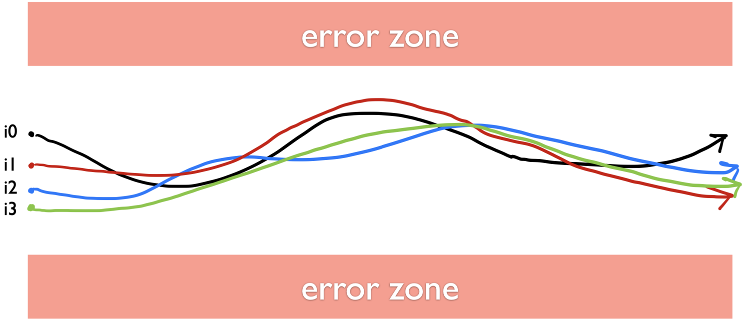


Basic Principle

- Observe the program behavior by “executing” the program
  - Report errors found during the execution
  - When no error is found, report “verified”
- Three types of program execution:
  - Concrete execution
  - Symbolic execution
  - Abstract execution
  - and their combinations, e.g., concolic execution

Program Analysis based on Concrete Execution

- Execute the program with concrete inputs, analyzing individual program states separately



Example: Random Testing / Fuzzing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

1. Error-triggering test?

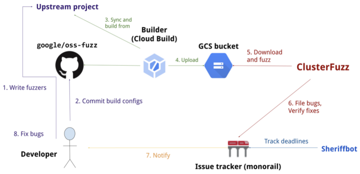
2. Probability of the error?
(assume 0 ≤ x,y ≤ 10,000)

Types of Fuzzing

- Blackbox fuzzing
- Greybox fuzzing
- Whitebox fuzzing

Use Cases

- AFL (`https://github.com/google/AFL`)
- OSS-Fuzz (`https://github.com/google/oss-fuzz`)



Google OSS-Fuzz

Microsoft

DOI:10.1145/3363824

Reviewing software testing techniques for finding security vulnerabilities.

BY PATRICE GODEFROID

Fuzzing: Hack, Art, and Science

FUZZING, OR FUZZ TESTING, is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed, inputs.[35] Since the early 2000s, fuzzing has become a mainstream practice in assessing software security. Thousands of security vulnerabilities have been found while fuzzing all kinds of software applications for processing documents, images, sounds, videos, network packets, Web pages, among others. These applications must deal with untrusted inputs encoded in complex data formats. For example, the Microsoft Windows operating system supports over 360 file formats and includes millions of lines of code just to handle all of these.

Most of the code to process such files and packets evolved over the last 20+ years. It is large, complex, and written in C/C++ for performance reasons. If an attacker could trigger a buffer-overflow bug in one of these applications, s/he could corrupt the memory of the application and possibly hijack its execution to run malicious code (elevation-of-privilege attack), or steal internal information (information-disclosure attack), or simply crash the application (denial-of-service attack).[9] Such attacks might be launched by tricking the victim into opening a single malicious document, image, or Web page. If you are reading this article on an electronic device, you are using a PDF and JPEG parser in order to see Figure 1.

Buffer-overflows are examples of security vulnerabilities: they are programming errors, or bugs, and typically triggered only in specific hard-to-find corner cases. In contrast, an exploit is a piece of code which triggers a security vulnerability and then takes advantage of it for malicious purposes. When exploitable, a security vulnerability is like an unintended backdoor in a software application that lets an attacker enter the victim's device.

There are approximately three main ways to detect security vulnerabilities in software.

Static program analyzers are tools that automatically inspect code and

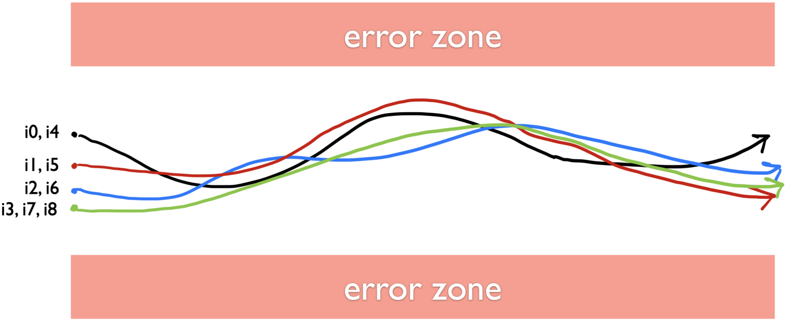
key insights

- **Fuzzing means automatic test generation and execution with the goal of finding security vulnerabilities.**
- **Over the last two decades, fuzzing has become a mainstay in software security. Thousands of security vulnerabilities in all kinds of software have been found using fuzzing.**
- **If you develop software that may process untrusted inputs and have never used fuzzing, you probably should.**

70 COMMUNICATIONS OF THE ACM | FEBRUARY 2020 | VOL. 63 | NO. 2

Program Analysis based on Symbolic Execution

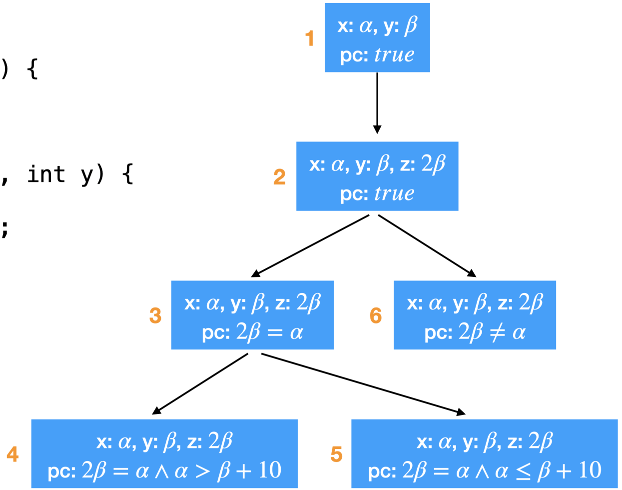
- Execute the program with symbolic inputs, analyzing each program path only once



Example: Symbolic Execution

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
1
  z := double (y);
2
  if (z==x) {
   3
    if (x>y+10) {
     4 Error;
    } else { 5 …}
  }
6 }
```



Example: Concolic Testing

```
int foo (int v) {
  return hash(v);
}

void testme(int x, int y) {

  z := foo (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7	x=α, y=β
	true

Use Cases

- Symbolic execution is good at detecting tricky bugs

<table>
<tr><th>Benchmarks</th><th>Versions</th><th>Error Types</th><th>Bug-Triggering Inputs</th></tr>
<tr><td rowspan="4">vim</td><td>8.1*</td><td>Non-termination</td><td><code>K1!1000100100111110(</code></td></tr>
<tr><td rowspan="3">5.7</td><td>Abnormal-termination</td><td><code>H:w>>`"`\</code> [press ‘Enter’]</td></tr>
<tr><td>Segmentation fault</td><td><code>=ipI\-9~qOqw</code></td></tr>
<tr><td>Non-termination</td><td><code>v(ipaprq&T$T</code></td></tr>
<tr><td rowspan="3">gawk</td><td>4.2.1*</td><td>Memory-exhaustion</td><td><code>'+E_Q$h+w$8==++$6E8#'</code></td></tr>
<tr><td rowspan="2">3.0.3</td><td>Abnormal-termination</td><td><code>'f[][][][][y]^/#['</code></td></tr>
<tr><td>Non-termination</td><td><code>'$g?E2^=-E-2"?^+$=":/?/#["'</code></td></tr>
<tr><td rowspan="4">grep</td><td rowspan="2">3.1*</td><td>Abnormal-termination</td><td><code>'\(\)\1*?*?\|\W*\1W*'</code></td></tr>
<tr><td>Segmentation fault</td><td><code>'\(\)\1^*@*\?\1*\+*\?'</code></td></tr>
<tr><td rowspan="2">2.2</td><td>Segmentation fault</td><td><code>"_^^*9\|^\(\)\'\1*$"</code></td></tr>
<tr><td>Non-termination</td><td><code>'\({**+**\)*\++*\1*\+'</code></td></tr>
<tr><td>sed</td><td>1.17</td><td>Segmentation fault</td><td><code>'{:};:C;b'</code></td></tr>
</table>

(Concolic Testing with Adaptively Changing Search Heuristics. FSE 2019)

Use Cases

DOI:10.1145/2093548.2093564

Article development led by acmqueue
queue.acm.org

SAGE has had a remarkable impact at Microsoft.

BY PATRICE GODEFROID, MICHAEL Y. LEVIN, AND DAVID MOLNAR

SAGE: Whitebox Fuzzing for Security Testing

MOST *COMMUNICATIONS* READERS might think of "program verification research" as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like over 93% of PC users—that is, more than one billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

Every second Tuesday of every month, also known as "Patch Tuesday," Microsoft releases a list of security bulletins and associated security patches to be deployed on hundreds of millions of machines worldwide. Each security bulletin costs Microsoft and its users millions of dollars. If a monthly security update costs you $0.001 (one tenth of one cent) in just electricity or loss of productivity, then this number multiplied by one billion people is $1 million. Of course, if malware were spreading on your machine, possibly leaking some of your private data, then that might cost you much more than $0.001. This is why we strongly encourage you to apply those pesky security updates.

Many security vulnerabilities are a result of programming errors in code for parsing files and packets that are transmitted over the Internet. For example, Microsoft Windows includes parsers for hundreds of file formats.

If you are reading this article on a computer, then the picture shown in Figure 1 is displayed on your screen after a jpg parser (typically part of your operating system) has read the image data, decoded it, created new data structures with the decoded data, and passed those to the graphics card in your computer. If the code implementing that jpg parser contains a bug such as a buffer overflow that can be triggered by a corrupted jpg image, then the execution of this jpg parser on your computer could potentially be hijacked to execute some other code, possibly malicious and hidden in the jpg data itself.

This is just one example of a possible security vulnerability and attack scenario. The security bugs discussed throughout the rest of this article are mostly buffer overflows.

Hunting for "Million-Dollar" Bugs

Today, hackers find security vulnerabilities in software products using two primary methods. The first is code inspection of binaries (with a good disassembler, binary code is like source code).

The second is *blackbox fuzzing*, a form of blackbox random testing, which randomly mutates well-formed program inputs and then tests the program with those modified inputs,[3] hoping to trigger a bug such as a buf-

Symbolic Execution for Software Testing in Practice – Preliminary Assessment

Cristian Cadar, Imperial College London, c.cadar@imperial.ac.uk
Patrice Godefroid, Microsoft Research, pg@microsoft.com
Sarfraz Khurshid, U. Texas at Austin, khurshid@ece.utexas.edu
Corina S. Păsăreanu*, CMU/NASA Ames, corina.s.pasareanu@nasa.gov
Koushik Sen, U.C. Berkeley, ksen@eecs.berkeley.edu
Nikolai Tillmann, Microsoft Research, nikolait@microsoft.com
Willem Visser, Stellenbosch University, visserw@sun.ac.za

ABSTRACT

We present results for the "Impact Project Focus Area" on the topic of symbolic execution as used in software testing. Symbolic execution is a program analysis technique introduced in the 70s that has received renewed interest in recent years, due to algorithmic advances and increased availability of computational power and constraint solving technology. We review classical symbolic execution and some modern extensions such as generalized symbolic execution and dynamic test generation. We also give a preliminary assessment of the use in academia, research labs, and industry.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Symbolic execution

General Terms

Reliability

Keywords

Generalized symbolic execution, dynamic test generation

1. INTRODUCTION

The ACM-SIGSOFT Impact Project is documenting the impact that software engineering research has had on software development practice. In this paper, we present preliminary results for documenting the impact of research in symbolic execution for automated software testing. Symbolic execution is a program analysis technique that was introduced in the 70s [8, 15, 31, 35, 46], and that has found renewed interest in recent years [9, 12, 13, 28, 29, 32, 33, 40, 42, 43, 50–52, 56, 57].

Symbolic execution is now the underlying technique of several popular testing tools, many of them open-source: NASA's Symbolic (Java) PathFinder[1], UIUC's CUTE and jCUTE[2], Stanford's KLEE[3], UC Berkeley's CREST[4] and BitBlaze[5], etc. Symbolic execution tools are now used in industrial practice at Microsoft (Pex[6], SAGE [29], YOGI[7] and PREfix [10]), IBM (Apollo [2]), NASA and Fujitsu (Symbolic PathFinder), and also form a key part of the commercial testing tool suites from Parasoft and other companies [60].

Although we acknowledge that the impact of symbolic execution in software practice is still limited, we believe that the explosion of work in this area over the past years makes for an interesting story about the increasing impact of symbolic execution since it was first introduced in the 1970s. Note that this paper is not meant to provide a comprehensive survey of symbolic execution techniques; such surveys can be found elsewhere [19, 44, 49]. Instead, we focus here on a few modern symbolic execution techniques that have shown promise to impact software testing in practice.

Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for a large fraction of software development and maintenance. Symbolic execution is one of the many techniques that can be used to automate software testing by automatically generating test cases that achieve high coverage of program executions.

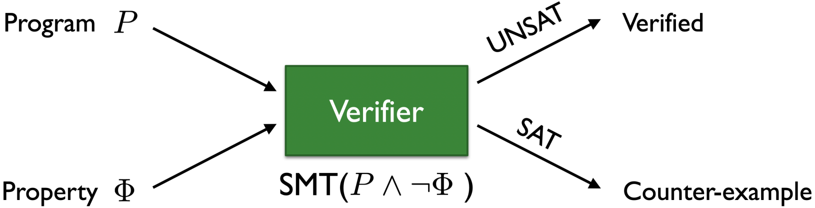
Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs and maintains a *path condition* that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Test generation is performed by solving the collected constraints using a constraint solver. Symbolic execution can also be used for bug finding, where it checks for run-time errors or assertion violations and it generates test inputs that trigger those errors.

The original approaches to symbolic execution [8, 15, 31, 35,

*We thank Matt Dwyer for his advice

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[2] http://osl.cs.uiuc.edu/~ksen/cute/
[3] http://klee.llvm.org/
[4] http://code.google.com/p/crest/
[5] http://bitblaze.cs.berkeley.edu/
[6] http://research.microsoft.com/en-us/projects/pex/
[7] http://research.microsoft.com/en-us/projects/yogi/

Example: Symbolic Verification



- Represent program behavior and property as a formula in logic
- Determine the satisfiability of the formula

Example 1

```
int f(bool a) {
  x = 0; y = 0;
  if (a) {
    x = 1;
  }
  if (a) {
    y = 1;
  }
  assert (x == y)
}
```

① ⇒

Verification Condition:

$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$
$((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge$
$\neg(x == y)$

② SMT solver: unsatisfiable!

Example 2

```
int f(a, b) {
  x = 0; y = 0;
  if (a) {
    x = 1;
  }
  if (b) {
    y = 1;
  }
  assert (x == y)
}
```

① Verification Condition:

$$((a \land x) \lor (\neg a \land \neg x)) \land$$
$$((b \land y) \lor (\neg b \land \neg y)) \land$$
$$\neg(x == y)$$

② SMT solver:

satisfiable when $a=1$ and $b=0$

Challenge: Loop Invariant

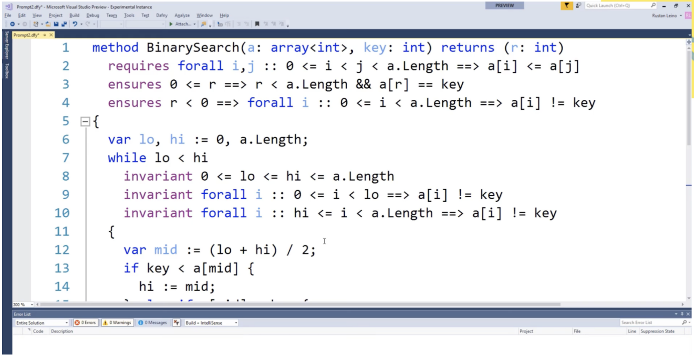
- Property that holds at the beginning of every loop iteration

```
i = 0;
j = 0;
while @(i==j)
(i < 10) {
  i++;
  j++;
}
assert (i-j==0)
```

- Infinitely many invariants exist for a loop. Need to find one strong enough to prove the given property.

Use Cases

- The Dafny programming language used in Amazon



Use Cases

Code-Level Model Checking in the Software Development Workflow

Nathan Chong, Amazon
Byron Cook, Amazon, UCL
Konstantinos Kallas, University of Pennsylvania
Kareem Khazem, Amazon
Felipe R. Monteiro, Amazon
Daniel Schwartz-Narbonne, Amazon
Serdar Tasiran, Amazon
Michael Tautschnig, Amazon, Queen Mary University of London
Mark R. Tuttle, Amazon

ABSTRACT

This experience report describes a style of applying symbolic model checking developed over the course of four years at Amazon Web Services (AWS). Lessons learned are drawn from proving properties of numerous C-based systems, e.g., custom hypervisors, encryption code, boot loaders, and an IoT operating system. Using our methodology, we find that we can prove the correctness of industrial low-level C-based systems with reasonable effort and predictability. Furthermore, AWS developers are increasingly writing their own formal specifications. All proofs discussed in this paper are publicly available on GitHub.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Model checking*; Correctness; • **Theory of computation** → Program reasoning.

KEYWORDS

Continuous Integration, Model Checking, Memory Safety.

ACM Reference Format:
Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381347

1 INTRODUCTION

This is a report on making code-level proof via model checking a routine part of the software development workflow in a large industrial organization. Formal verification of source code can have a significant positive impact on the quality of industrial code. In particular, formal specification of code provides precise, machine-checked documentation for developers and consumers of a code base. They improve code quality by ensuring that the program's *implementation* reflects the developer's *intent*. Unlike testing, which can only validate code against a set of concrete inputs, formal proof can assure that the code is both secure and correct for all possible inputs.

Unfortunately, rapid proof development is difficult in cases where proofs are written by a separate specialized team and *not* the software developers themselves. The developer writing a piece of code has an internal mental model of their code that explains why, and under what conditions, it is correct. However, this model typically remains known only to the developer. At best, it may be partially captured through informal code comments and design documents. As a result, the proof team must spend significant effort to reconstruct the formal specification of the code they are verifying. This slows the process of developing proofs.

Over the course of four years developing code-level proofs in Amazon Web Services (AWS), we have developed a proof methodology that allows us to produce proofs with reasonable and predictable effort. For example, using these techniques, one full-time verification engineer and two interns were able to specify and verify 171 entry points over 9 key modules in the AWS C Common[1] library over a period of 24 weeks (see Sec. 3.2 for a more detailed description of this library). All specifications, proofs, and related artifacts (such as continuous integration reports), described in this paper have been integrated into the main AWS C Common repository on GitHub, and are publicly available at https://github.com/awslabs/aws-c-common/.

1.1 Methodology

Our methodology has four key elements, all of which focus on communicating with the development team using artifacts that fit their existing development practices. We find that of the many different ways we have approached verification engagements, this combination of techniques has most deeply involved software developers in the proof creation and maintenance process. In particular, developers have begun to write formal functional specifications for code as they develop it. Initially, this involved the development team asking the verification team to assist them in writing specifications for new

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05.
https://doi.org/10.1145/3377813.3381347

[1] https://github.com/awslabs/aws-c-common

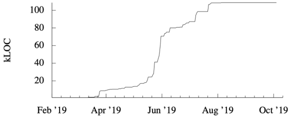


Figure 1: Cumulative number of LOC proven.

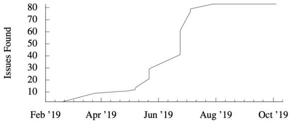


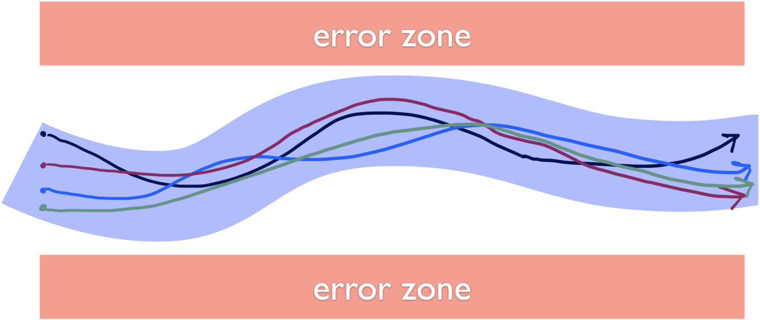
Figure 2: Cumulative number of issues found.

Table 1: Severity and root cause of issues found.

Root cause	# issues	Severity: High	Medium	Low
Integer overflow	10 (12%)	2	8	0
Null-pointer deref.	57 (69%)	0	14	43
Functional	11 (13%)	0	4	7
Memory safety	5 (6%)	0	5	0
Total	83	2 (3%)	31 (37%)	50 (60%)

Program Analysis based on Abstract Execution (Static Analysis)

- Execute the program with abstract inputs, analyzing all program behaviors simultaneously



Principles of Abstract Interpretation

$$30 \times 12 + 11 \times 9 = ?$$

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - "integer", "odd integer", "positive integer", "$400 \leq n \leq 500$", etc
- Static analysis process:
  1. Choose abstract value (domain), e.g., $\hat{V} = \{\top, e, o, \bot\}$
  2. Define the program execution in terms of abstract values:

$\hat{\times}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

$\hat{+}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

  3. "Execute" the program:

$$e \mathbin{\hat{\times}} e \mathbin{\hat{+}} o \mathbin{\hat{\times}} o = o$$

Principles of Abstract Interpretation

- By contrast to testing, static analysis can prove the absence of bugs:

```
void f (int x)  {
    y = x * 12 + 9 * 11;
    assert (y % 2 == 1);
}
```

- Instead, static analysis may produce false alarms:

```
void f (int x)  {
    y = x + x;
    assert (y % 2 == 0);
}
```

Use Cases

DOI:10.1145/3338112

Key lessons for designing static analyses tools deployed to find bugs in hundreds of millions of lines of code.

BY DINO DISTEFANO, MANUEL FÄHNDRICH, FRANCESCO LOGOZZO, AND PETER W. O'HEARN

Scaling Static Analyses at Facebook

Infer

STATIC ANALYSIS TOOLS are programs that examine, and attempt to draw conclusions about, the source of other programs without running them. At Facebook, we have been investing in advanced static analysis tools that employ reasoning techniques similar to those from program verification. The tools we describe in this article (Infer and Zoncolan) target issues related to crashes and to the security of our services, they perform sometimes complex reasoning spanning many procedures or files, and they are integrated into engineering workflows in a way that attempts to bring value while minimizing friction.

These tools run on code modifications, participating as bots during the code review process. Infer targets our mobile apps as well as our backend C++ code, codebases with 10s of millions of lines; it has seen over 100 thousand reported issues fixed by developers before code reaches production. Zoncolan targets the 100-million lines of Hack code, and is additionally integrated in the workflow used by security engineers. It has led to thousands of fixes of security and privacy bugs, outperforming any other detection method used at Facebook for such vulnerabilities. We will describe the human and technical challenges encountered and lessons we have learned in developing and deploying these analyses.

There has been a tremendous amount of work on static analysis, both in industry and academia, and we will not attempt to survey that material here. Rather, we present our rationale for, and results from, using techniques similar to ones that might be encountered at the edge of the research literature, not only simple techniques that are much easier to make scale. Our goal is to complement other reports on industrial static analysis and formal methods,[1,6,13,17] and we hope that such perspectives can provide input both to future research and to further industrial use of static analysis.

Next, we discuss the three dimensions that drive our work: bugs that matter, people, and actioned/missed bugs. The remainder of the article describes our experience developing and deploying the analyses, their impact, and the techniques that underpin our tools.

Context for Static Analysis at Facebook

Bugs that Matter. We use static analysis to prevent bugs that would affect our products, and we rely on our engineers' judgment as well as data from production to tell us the bugs that matter the most.

» key insights

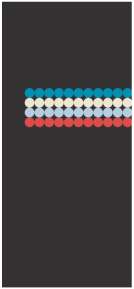
- **Advanced static analysis techniques performing deep reasoning about source code can scale to large industrial codebases, for example, with 100-million LOC.**
- **Static analyses should strike a balance between missed bugs (false negatives) and un-actioned reports (false positives).**
- **A "diff time" deployment, where issues are given to developers promptly as part of code review, is important to catching bugs early and getting high fix rates.**

DOI:10.1145/3188720

For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.

BY CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CIERA JASPAN

Lessons from Building Static Analysis Tools at Google



SOFTWARE BUGS COST developers and software companies a great deal of time and money. For example, in 2014, a bug in a widely used SSL implementation ("goto fail") caused it to accept invalid SSL certificates,[36] and a bug related to date formatting caused a large-scale Twitter outage.[23] Such bugs are often statically detectable and are, in fact, obvious upon reading the code or documentation yet still make it into production software.

Previous work has reported on experience applying bug-detection tools to production software.[6,3,7,29] Although there are many such success stories for developers using static analysis tools, there are also reasons engineers do not always use static analysis tools or ignore their warnings,[6,7,26,30] including:

Not integrated. The tool is not integrated into the developer's workflow or takes too long to run;

Not actionable. The warnings are not actionable;

Not trustworthy. Users do not trust the results due to, say, false positives;

Not manifest in practice. The reported bug is theoretically possible, but the problem does not actually manifest in practice;

» key insights

- **Static analysis authors should focus on the developer and listen to their feedback.**
- **Careful developer workflow integration is key for static analysis tool adoption.**
- **Static analysis tools can scale by crowdsourcing analysis development.**

Summary: Program Analysis

- Basically classified based on how programs are interpreted:
  - Concrete/symbolic/abstract execution
- Each approach has its own strengths and weaknesses: e.g.,

	Automatic	Sound	Complete	When
Testing/Fuzzing				
Symbolic Execution				
Static Analysis				
Program Verification				
?				