Fuzzing for SMT Solvers

(A Work in Progress)

Kyle Dewey, Mehmet Emre, Ben Hardekopf

University of California, Santa Barbara

Teaser

- We identify several key assumptions which are made in the domain of SMT fuzzers
- We seek to empirically show these assumptions are false, and we already have data showing that common wisdom is incorrect
- We have already found over a dozen bugs across several popular SMT solvers, including Z3, CVC4, MathSAT5, and Boolector (still plenty to do)
    - Including incorrect results
    - Most promptly fixed by developers (1 week)

Outline

- Motivation and background
- Developing SMT fuzzers
- Evaluation and results so far
- Conclusion

Outline

- Motivation and background
- Developing SMT fuzzers
- Evaluation and results so far
- Conclusion

Motivation

- SMT solvers are frequently employed in automated testing, synthesis, and verification
- Often assumed to be correct, and their correctness is vital

Z3

CVC

MathSAT 5

Motivation

- Problem: SMT solvers can be, and often are, buggy
- Bugs are potentially devastating for downstream applications
  - Automated testing: input tests the wrong component
  - Synthesis: generated program does not have specified behavior
  - Verification: proof does not actually hold

Goal

- Find bugs in SMT solvers, before they cause downstream problems
- We employ black-box *language fuzzing* techniques for this purpose
  - The inputs for SMT solvers are formulas written in SMT-LIB, a standardized language

Background

- We did not invent language fuzzing

Background

- We did not invent language fuzzing

William M. McKeeman

Differential Testing for Software

Differential testing, a form of random testing, is a component of a mature testing technology for large software systems. It complements regression testing based on commercial test suites and tests locally developed during product development and deployment. Differential testing requires that two or more comparable systems be available to the tester. These systems are presented with an exhaustive series of mechanically generated test cases. If (we might say when) the results differ or one of

The Testing Problem

Successful commercial computer systems contain tens of millions of lines of handwritten software, all of which is subject to change as competitive pressures motivate the addition of new features in each release. As a practical matter, quality is not a question of correctness, but rather of how many bugs are fixed and how few are introduced in the ongoing development process. If the bug count is increasing, the software is deteriorating.

Quality

Background

- We did not invent language fuzzing

William M. McKeeman

Differential Testing for Software

Differential testing, a form of random testing, is a component of a mature testing technology for large software systems. It complements regression testing based on commercial test suites and tests locally developed during product development and deployment. Differential testing requires that two or more comparable systems be available to the tester. These systems are presented with an exhaustive series of mechanically generated test cases. If (we might say when) the results differ or one of

The Test

Successfi
of millic
which is
motivate
As a prac
rectness,
how few
process. If the bug count is increasing, the software is deteriorating.

Quality

Introducing jsfunfuzz

I wrote a fuzzer called jsfunfuzz for testing the JavaScript engine in Firefox. Window, Shaver, and I announced it at Black Hat earlier today, as part of Mozilla's presentation, "Building and Breaking the Browser".

It tests the JavaScript language engine itself, not the DOM. (That means that it works with language features such as functions, objects, operators, and garbage collection rather than DOM objects accessed through "window" or "document".)

It has found about 280 bugs in Firefox's JavaScript engine, over two-thirds of which have already been fixed (go Brendan!). About two dozen were memory safety bugs that we believe were likely to be exploitable to run arbitrary code.

In the presentation, I speculated as why it has been able to find so many bugs:

Background

- We did not invent language fuzzing

William M. McKeeman

Differential Testing for Software

Introducing jsfunfuzz

I wrote a fuzzer called jsfunfuzz for testing the JavaScript engine in Firefox. Window, Shaver, and I announced it at Black Hat earlier today, as part of
...e Browser".

...t the DOM. (That means that
...ions, objects, operators, and
...ccessed through "window" or

...cript engine, over two-thirds
...an!). About two dozen were
...ely to be exploitable to run

...s been able to find so many

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

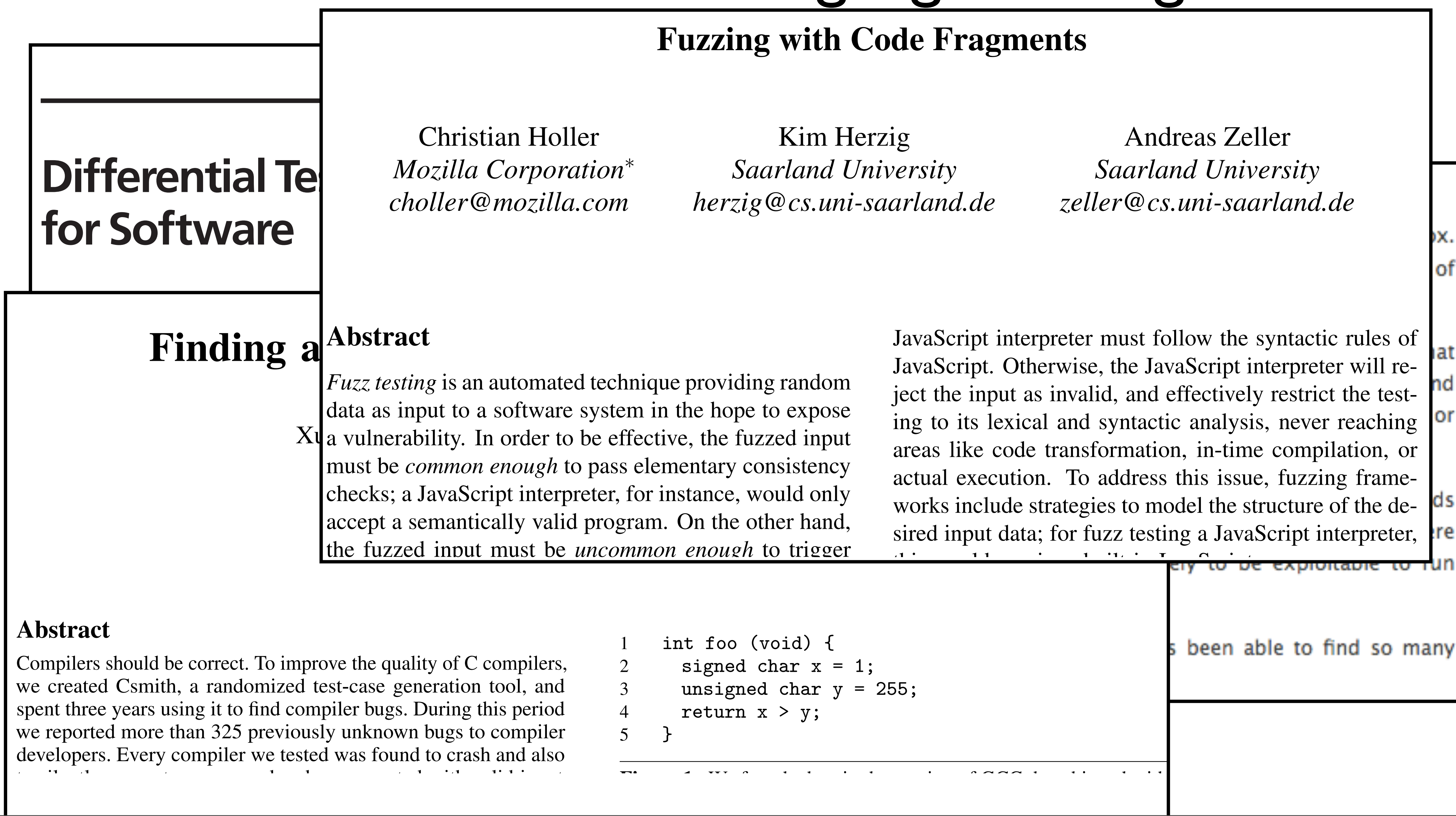
Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also

```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Background

- We did not invent language fuzzing



Differential Te
for Software

Finding a

Xu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also

```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

s been able to find so many

Fuzzing with Code Fragments

Christian Holler
*Mozilla Corporation**
choller@mozilla.com

Kim Herzig
Saarland University
herzig@cs.uni-saarland.de

Andreas Zeller
Saarland University
zeller@cs.uni-saarland.de

Abstract

Fuzz testing is an automated technique providing random data as input to a software system in the hope to expose a vulnerability. In order to be effective, the fuzzed input must be *common enough* to pass elementary consistency checks; a JavaScript interpreter, for instance, would only accept a semantically valid program. On the other hand, the fuzzed input must be *uncommon enough* to trigger

JavaScript interpreter must follow the syntactic rules of JavaScript. Otherwise, the JavaScript interpreter will reject the input as invalid, and effectively restrict the testing to its lexical and syntactic analysis, never reaching areas like code transformation, in-time compilation, or actual execution. To address this issue, fuzzing frameworks include strategies to model the structure of the desired input data; for fuzz testing a JavaScript interpreter,

Background

- We did not invent language fuzzing

Differential Te

for Software

Fuzzing with Code Fragments

Christian Holler
*Mozilla Corporation**
choller@mozilla.com

Kim Herzig
Saarland University
herzig@cs.uni-saarland.de

Andreas Zeller
Saarland University
zeller@cs.uni-saarland.de

Announcing cross_fuzz, a potential 0-day in circulation, and more

I am happy to announce the availability of `cross_fuzz` - a surprisingly effective but notoriously annoying cross-document DOM binding fuzzer that helped identify about one hundred bugs in all browsers on the market - many of said bugs exploitable - and is still finding more.

The fuzzer owes much of its efficiency to dynamically generating extremely long-winding sequences of DOM operations across multiple documents, inspecting returned objects, recursing into them, and creating circular node references that stress-test garbage collection mechanisms.

ely to be exploitable to run

s been able to find so many

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also


```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Background

- ...nor are we the first to apply language fuzzing to SMT solvers

Background

- ...nor are we the first to apply language fuzzing to SMT solvers

Fuzzing and Delta-Debugging SMT Solvers

Robert Brummayer and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Abstract. SMT solvers are widely used as core engines in many applications. Therefore, robustness and correctness are essential criteria. Current testing techniques used by developers of SMT solvers do not satisfy the high demand for correct and robust solvers, as our testing experiments show. To improve this situation, we propose to complement traditional testing techniques with grammar-based blackbox fuzz testing, combined with delta-debugging. We demonstrate the effectiveness of our approach and report on critical bugs and incorrect results which we found in current state-of-the-art SMT solvers for bit-vectors and arrays.

Background

- ...nor are we the first to apply language fuzzing to SMT solvers

Automated Testing and Debugging of SAT and QBF Solvers

Robert Brummayer, Florian Lonsing and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Abstract. Robustness and correctness are essential criteria for SAT and QBF solvers. We develop automated testing and debugging techniques designed and optimized for SAT and QBF solver development. Our fuzz testing techniques are able to find critical solver defects that lead to crashes, invalid satisfying assignments and incorrect satisfiability results. Moreover, we show that sequential and concurrent delta debugging techniques are highly effective in minimizing failure-inducing inputs.

Abs... e-
fore, ... ed
by d... st
solve... to
comp... t-
ing, combined with delta debugging. We demonstrate the effectiveness of our ap-
proach and report on critical bugs and incorrect results which we found in current state-of-the-art SMT solvers for bit-vectors and arrays.

Existing Weaknesses

- Focus has been on **syntax**, not semantics
  - E.g., formulas that syntactically contain 0, as opposed to formulas that evaluate to 0 somewhere
- Tests crafted to "look" like typical inputs, or be time-consuming to solve
  - Not focused on what is difficult to implement

Common Wisdom

- Large inputs mean more bugs
- Random search performs better instead of bounded depth-first
- Not just for SMT solvers, but for language fuzzing overall
- Very little empirical evidence backing these claims (blog posts and a technical report)

Hypotheses (1, 2)

- Semantics-guided approaches can find bugs that purely syntax-oriented approaches practically cannot
  - Suggested to be true by our own prior work
- Constraining the search space to focus on different subsets can effectively find additional bugs
  - Purely syntactic constraining shown effective in *Swarm Testing*

Hypotheses (3, 4)

- Large inputs are not necessarily better for finding bugs
  - Suggested to be true by the need for input reducers, and by our own prior work
- Random search is not necessarily better than bounded depth-first search
  - Suggested to be true by our own prior work

Key Observation

- These four hypotheses are **orthogonal**

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search		
Bounded DFS		

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search		Traditional
Bounded DFS		

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search		Traditional
Bounded DFS	Our prior work	

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features		
Feature Subsets		

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	
Feature Subsets		

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	
Feature Subsets	Traditional w/ *Swarm Testing*	

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Key Observation

- These four hypotheses are **orthogonal**

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Common Wisdom

The highlighted points work well...

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Common Wisdom

...but these highlighted points do not...

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Common Wisdom

...and these highlighted points are atypical.

Course-Grained, Language-Agnostic Properties

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

Fine-Grained, Language-Dependent Properties

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Overall Design Philosophy

- Gather data for each point in this 4D space, specific to SMT solvers
- Determine which setups find the most bugs, and which ones find the same bugs
  - Ultimately, figure out which setups work well and which do not for SMT solvers

Rest of Talk

How each of these positions in the diagram can be filled in, forming different fuzzers

	Small Inputs	Big Inputs
Random Search		
Bounded DFS		

	Syntax Based	Semantic Based
All Features		
Feature Subsets		

Outline

- Motivation and background
- Developing SMT fuzzers
- Evaluation and results so far
- Conclusion

Traditional Syntax-Based Fuzzers

Necessary to have, but only for comparison. Not the focus of this work.

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Semantic-Based Fuzzers

Number of different strategies, depending on the particular semantics

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

ASE'15 Paper

- In prior work, we looked at fuzzing the typechecker in the Rust langauge
  - Focus on *static semantics*: types
- Guiding principles from that work can be applied to fuzzing SMT solvers

Application to SMT Solvers

- SMT-LIB is statically typed, and typing rules are described (though not formalized)
- These typing rules can be used to generate well-typed programs
  - Suitable for ensuring that solver typecheckers accept inputs properly
  - Suitable for differential testing
- Requires implementing a typechecker for SMT-LIB using constraint logic programming (CLP)

A Twist for Dynamic Semantics

- For a static semantics: implement typing rules in CLP
- For a dynamic semantics: implement a definitional oracle
  - Inputs generated explore the semantics, by construction
  - Edge cases fall out naturally (e.g., division by zero as a special case)

Consistency-Based Testing

- Another guiding principle from ASE'15: devise methods to test internal consistency
- Based on generating *pairs* of inputs, which should behave in the same way
  - E.g., both `SAT` or `UNSAT`
- Generally do not know true correct result

Consistency for SMT-LIB

- We devise two novel approaches for finding consistency bugs in SMT-LIB
  - One: equivalence through translation
  - Two: logical implications of mathematical functions

Translation Equivalence

- SMT-LIB features a variety of *theories*, which describe different kinds of domains and operations that can be reasoned about
  - E.g., integers, bitvectors, floating point
- Some queries can be translated between different theories, and should behave the same after translation

Translation Equivalence Example

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

Theory of Integers

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

Theory of Integers

```
0 <= X <= 1
```

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

Theory of Integers

```
0 <= X <= 1
0 <= Y <= 1
```

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

Theory of Integers

```
0 <= X <= 1
0 <= Y <= 1
T = X + Y
Z = (if T == 2 then 0 else T)
```

```
X is a one bit bitvector
Y is a one bit bitvector
Z is X + Y
assert Z == 1
```

Theory of Bitvectors

Theory of Integers

```
0 <= X <= 1
0 <= Y <= 1
T = X + Y
Z = (if T == 2 then 0 else T)
assert Z == 1
```

Exploiting Mathematical Purity for Consistency Checking

Implication of Mathematical Functions

- SMT-LIB is a mathematically pure language
- Solvers generally implement the *theory of uninterpreted functions with equality (EUF)*, which essentially reasons over all possible function definitions

```
f(1, 2) == f(1, 2), ∀f
f(1, 2) != f(2, 1), ∀f
f(1, 2) != g(1, 2), ∀f
```

Exploiting Purity to Find Consistency Bugs

- If something holds in EUF, it **must** hold for any other theory

Exploiting Purity to Find Consistency Bugs

- If something holds in EUF, it **must** hold for any other theory

```
f(1, 2) == f(1, 2), ∀f
```

Exploiting Purity to Find Consistency Bugs

- If something holds in EUF, it **must** hold for any other theory

```
f(1, 2) == f(1, 2), ∀f
```

```
integer_add(1, 2) == integer_add(1, 2)
```

Exploiting Purity to Find Consistency Bugs

- If something holds in EUF, it **must** hold for any other theory

```
f(1, 2) == f(1, 2), ∀f
```

```
integer_add(1, 2) == integer_add(1, 2)
```

```
bitvec_add(1, 2) == bitvec_add(1, 2)
```

Exploiting Purity to Find Consistency Bugs

- Similarly, if something does not hold in another theory, it **must not** hold in EUF

Exploiting Purity to Find Consistency Bugs

- Similarly, if something does not hold in another theory, it **must not** hold in EUF

$\boxed{\texttt{integer\_add(1, 2) != integer\_add(2, 3)}}$

Exploiting Purity to Find Consistency Bugs

- Similarly, if something does not hold in another theory, it **must not** hold in EUF

```
integer_add(1, 2) != integer_add(2, 3)
```

$$\boxed{\texttt{f(1, 2) != f(2, 3), }\forall \texttt{f}}$$

Semantic Feature Subsets

Many possible different instantiations

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Semantic Feature Subsets

- In our case, we focus specifically on the *theory of floating point*
  - Bleeding edge (only two production-quality solvers to test against)
  - Features a semi-formal semantics
  - Quite complex

Semantic Feature Subsets

- We plan to focus on computations that deal with `NaN`, +/- 0, +/- ∞, subnormal numbers
    - All intuitively difficult
    - Some have been challenging to implement ourselves
- Not yet complete

Varying Input Sizes

Fairly trivial, and generally easily composable.

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Search Strategy Variation

Adjusting the search strategy is more difficult, and requires novel techniques

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Search Strategy

- Historically, the search strategy is fundamentally defined by the underlying generation technique, and cannot be varied without devising a whole new technique
  - E.g. cannot run the same fuzzer in a random mode and an exhaustive mode
- This is true even for CLP

Novel Abstraction

- We define a novel abstraction in CLP for varying the search strategy dynamically
  - Fuzzer code is written in a strategy-agnostic way
  - Accomplished via the use of a CLP metainterpreter

Abstraction Idea

- Ultimately, the abstraction is parameterized by a nondeterministic relation `choose`:

```
choose([C], C)
```

Given a list of choices...

...choose one nondeterministically

Possible Instantiations

- Randomly select a single element: random search without backtracking
- Nondeterministically select all in a fixed order: depth-first search
- Many more possible, including more complex ones seen in advanced fuzzers

Caveats

- Not all search strategies fit into this abstraction
  - E.g., breadth-first search
  - Fundamentally, `choice` is applied when selecting the next child to process in a built-in depth-first search
- Still encompasses all search strategies in practice which we are aware of

In Summary

We know how to cover each cell. Onto implementation and evaluation!

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Outline

- Motivation and background
- Developing SMT fuzzers
- **Evaluation and results so far**
- Conclusion

Fuzzers Implemented so Far

Traditional Syntactic Fuzzer

Preexisting, thanks to others

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Equivalency-Based Fuzzer

Uses translation between the theory of bitvectors and the theory of integers

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Fuzzer Based on Well-Typedness

Generates well-typed SMT-LIB formulas

	Small Inputs	Big Inputs
Random Search	???	Traditional
Bounded DFS	Our prior work	???

	Syntax Based	Semantic Based
All Features	Traditional	Our prior work
Feature Subsets	Traditional w/ *Swarm Testing*	???

Remaining Fuzzers

- Still many fuzzers left to implement
    - All are planned out
- Key point: most spaces are empty, but we have enough to compare against more traditional fuzzing strategies

Results so Far

- Traditional fuzzer: **nothing on Z3 in past year**; unknown for other solvers
  - Direct from the Z3 team
- Equivalency-based fuzzer: nothing so far (approximately two weeks)
- Fuzzer based on well-typed programs: **15 bugs**

Bugs Found

- Include crashes and incorrect results
  - Bitvector division by zero is tricky
  - Floating point is problematic on numbers consisting of just a few bits
- Surprisingly, quite a few in Z3
- One required communication with standards committee
- Most fixed within one week of reporting

Outline

- Motivation and background
- Developing SMT fuzzers
- Evaluation and results so far
- Conclusion

Key Points

- Common wisdom: large inputs are necessary to find bugs
  - **False**: all bugs found involve small formulas. Some become exponentially less likely with larger formulas
- Common wisdom: random search is necessary to find bugs
  - **False**: at least for small formulas, depth-first search works fine

In Conclusion

- While this is still incomplete, we have already accumulated some evidence against the common wisdom
- We are transitively improving the reliability of popular SMT solvers
- Still lots more to do