Finding Code That Explodes Under Symbolic Evaluation

James Bornholt
Emina Torlak
University of Washington
unsat.org
Automated reasoning tools help us solve hard programming problems.
Automated reasoning tools help us solve hard programming problems

Does my program still work after the file system crashes? [ASPLOS’16]
Automated reasoning tools help us solve hard programming problems

Does my program still work after the file system crashes? [ASPLOS’16]

Verification

Synthesis

How do I compile code for this weird new architecture? [PLDI’14]
Automated reasoning tools help us solve hard programming problems.

Verification:
- Does my program still work after the file system crashes? [ASPLOS’16]

Synthesis:
- How do I compile code for this weird new architecture? [PLDI’14]

“Programs”:
- How do I teach kids the rules of algebra effectively? [VMCAI’18]
Symbolic evaluators

Does my program still work after the file system crashes? [ASPLOS’16]

How do I compile code for this weird new architecture? [PLDI’14]
Symbolic evaluators

Does my program still work after the file system crashes? [ASPLOS’16]

How do I compile code for this weird new architecture? [PLDI’14]

Interpreter for file system operations

Interpreter for new architecture instructions
Symbolic evaluators

Does my program still work after the file system crashes? [ASPLOS’16]

Interpreter for file system operations

How do I compile code for this weird new architecture? [PLDI’14]

Interpreter for new architecture instructions

Symbolic evaluator
Sketch, Rosette, ...
Symbolic evaluators

Does my program still work after the file system crashes? [ASPLOS’16]

How do I compile code for this weird new architecture? [PLDI’14]

Interpreter for file system operations

Interpreter for new architecture instructions

Symbolic evaluator
Sketch, Rosette, ...

Verification
Synthesis
Angelic Execution
for free!
Symbolic evaluators: no free lunch

Does my program still work after the file system crashes? [ASPLOS’16]

Symbolic evaluator
- Sketch, Rosette, ...

- Verification
- Synthesis
- Angelic Execution

for free!
Symbolic evaluators: no free lunch

Does my program still work after the file system crashes? [ASPLOS’16]

Interpreter for file system operations

Symbolic evaluator
Sketch, Rosette, ...

Verification
Synthesis
Angelic Execution

for free!
Symbolic evaluators: no free lunch

Does my program still work after the file system crashes? [ASPLOS’16]

How do you make these tools scale?

Searching all paths through the interpreter

Symbolic evaluator
Sketch, Rosette, ...

Interpret for file system operations

Verification
Synthesis
Angelic Execution
for free!
Symbolic profiling identifies performance issues in symbolic evaluation
Symbolic profiling identifies performance issues in symbolic evaluation

Symbolic profiling
Data structures and analyses
Symbolic profiling identifies performance issues in symbolic evaluation

Symbolic profiling
Data structures and analyses

Symbolic evaluation anti-patterns
Common issues and source-level repairs
Symbolic profiling identifies performance issues in symbolic evaluation

Symbolic profiling
Data structures and analyses

Symbolic evaluation anti-patterns
Common issues and source-level repairs

Empirical results
300× speedup on real-world tools
Symbolic profiling identifies performance issues in symbolic evaluation

Symbolic evaluation
All-paths execution of programs

Symbolic profiling
Data structures and analyses

Symbolic evaluation anti-patterns
Common issues and source-level repairs

Empirical results
300× speedup on real-world tools
Symbolic evaluation
All-paths execution of programs
Symbolic evaluation executes all paths through a program

```scheme
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```
Symbolic evaluation executes all paths through a program

```scheme
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are unknown (trying to find values that violate spec)
Symbolic evaluation executes all paths through a program

```scheme
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))

(filter even? '(x₀ x₁))
```

Inputs are unknown (trying to find values that violate spec)
Symbolic evaluation executes \textit{all} paths through a program

```racket
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are \textit{unknown} (trying to find values that violate spec)
Symbolic evaluation executes *all* paths through a program

```rosette
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are *unknown* (trying to find values that violate spec)
Symbolic evaluation executes all paths through a program

```lang rosette
(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are unknown (trying to find values that violate spec)
Symbolic evaluation executes all paths through a program

```
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are unknown (trying to find values that violate spec)
Symbolic evaluation executes all paths through a program

```
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are unknown (trying to find values that violate spec)

```
(filter even? '(x₀ x₁))
```

```
¬(even? x₀)  (even? x₀)
```

```
(())  (x₀)
```

```
¬(even? x₁)  (even? x₁)  ¬(even? x₁)  (even? x₁)
```

```
(())  (x₁)  (x₀)  (x₀ x₁)
```

```
k=0  k=1  k=0  k=1  k=0  k=1  k=2
```

```
(())  (())  (x₁)  (())  (x₀)  (())  (x₀)  (x₀ x₁)
```

take runs $2^2$ times
Symbolic evaluation executes all paths through a program

```rosette
#lang rosette

(define (first-k-even lst k)
  (define xs (filter even? lst))
  (take xs k))
```

Inputs are unknown (trying to find values that violate spec)

because filter ran on a list of size 2

take runs $2^2$ times
Blaming filter even though it’s not the slowest
Symbolic profiling
Data structures and metrics
Two data structures to summarize symbolic evaluation

Symbolic evaluation graph
Reflects the evaluator’s strategy for all-paths execution of the program

Symbolic heap
Shape of all symbolic values created by the program

Any symbolic evaluation technique can be summarized by these two data structures
The symbolic evaluation graph summarizes branching and merging.

Symbolic evaluation graph:
- Nodes are program states
- Edges are transitions between states

Example:
\[(\text{filter even? } '(x_0, x_1))\]

\[\neg(\text{even? } x_0) \quad (\text{even? } x_0)\]

\[\neg(\text{even? } x_1) \quad (\text{even? } x_1)\]

\[k=0 \quad k=1\]

\[\neg(\text{even? } x_1) \quad (\text{even? } x_1)\]

\[k=0 \quad k=1\]

\[\neg(\text{even? } x_0) \quad (\text{even? } x_0)\]

\[k=0 \quad k=1 \quad k=2\]

\[\neg(\text{even? } x_0) \quad (\text{even? } x_0) \quad (x_0, x_1)\]
The *symbolic evaluation graph* summarizes branching and merging.

\[\text{(filter even? '(x\_0 x\_1))}\]

\[\neg(\text{even? } x\_0)\quad \rightarrow\quad (\text{even? } x\_0)\]

\[\neg(\text{even? } x\_1)\quad \rightarrow\quad (\text{even? } x\_1)\]

\[\neg(\text{even? } x\_0)\quad \rightarrow\quad (\text{even? } x\_0)\]

\[\neg(\text{even? } x\_1)\quad \rightarrow\quad (\text{even? } x\_1)\]
The *symbolic evaluation graph* summarizes branching and merging.

Symbolic execution:

\[
\text{(filter even? '(x₀ x₁))}
\]

\[
\neg(\text{even? } x₀) \quad \text{(even? } x₀) \\
\text{() } \quad 'x₀' \\
\neg(\text{even? } x₁) \quad (\text{even? } x₁) \\
'()' \quad '(x₁) \quad '(x₀) \quad '(x₀ x₁)
\]
The *symbolic evaluation graph* summarizes branching and merging.
The **symbolic evaluation graph** summarizes branching and merging.

**Symbolic execution**

- \(\text{filter even? } '(x_0 \ x_1)\)
- \(\neg \text{even? } x_0\) \(\rightarrow\) \('()'\)
- \(\text{even? } x_0\) \(\rightarrow\) \(''(x_0)''\)
- \(\neg \text{even? } x_1\) \(\rightarrow\) \(''(x_1)''\)
- \(\text{even? } x_1\) \(\rightarrow\) \(''(x_0)''\)
- \(\neg \text{even? } x_1\) \(\rightarrow\) \(''(x_0 \ x_1)''\)

**Bounded model checking**

- \(\text{filter even? } '(x_0 \ x_1)\)
- \(\neg \text{even? } x_0\) \(\rightarrow\) \('()'\)
- \(\text{even? } x_0\) \(\rightarrow\) \(''(x_0)''\)
- \(\neg \text{even? } x_1\) \(\rightarrow\) \(''(x_1)''\)
- \(\text{even? } x_1\) \(\rightarrow\) \(''(x_0)''\)
- \(\neg \text{even? } x_1\) \(\rightarrow\) \(''(x_0 \ x_1)''\)

\[
y_{\Theta} = \text{ite} (\text{even? } x_{\Theta}) '()' ''(x_{\Theta})''
\]
The symbolic evaluation graph summarizes branching and merging.

Symbolic execution:

\[
\text{(filter even? '(x_0 x_1))} \quad \neg \text{(even? x_0)} \quad \text{(even? x_0)} \\
\quad \text{()} \quad \text{(x_0)} \\
\neg \text{(even? x_1)} \quad \text{(even? x_1)} \quad \neg \text{(even? x_1)} \quad \text{(even? x_1)} \\
\quad \text{()} \quad \text{(x_1)} \quad \text{(x_0)} \quad \text{(x_0 x_1)}
\]

Bounded model checking:

\[
\text{(filter even? '(x_0 x_1))} \quad \neg \text{(even? x_0)} \quad \text{(even? x_0)} \\
\quad \text{()} \quad \text{(x_0)} \\
\neg \text{(even? x_1)} \quad \text{(even? x_1)} \quad \neg \text{(even? x_1)} \quad \text{(even? x_1)} \\
\quad \text{()} \quad \text{(x_1)} \quad \text{(x_0)} \quad \text{(x_0 x_1)}
\]

\[
ys_0 = \text{(ite (even? x_0) '() '(x_0))} \\
ys_1 = \text{(append ys_0 '(x_1))}
\]
The **symbolic evaluation graph** summarizes branching and merging.

**Symbolic execution**

- \((\text{filter even? } '(x_0 \ x_1))\)
  - \(\neg(\text{even? } x_0)\)
  - \((\text{even? } x_0)\)
  - \(x_0\)
  - \(x_1\)
  - \((x_0)\)
  - \((x_1)\)

**Bounded model checking**

- \((\text{filter even? } '(x_0 \ x_1))\)
  - \(\neg(\text{even? } x_0)\)
  - \((\text{even? } x_0)\)
  - \(x_0\)
  - \(x_1\)
  - \((x_0)\)
  - \((x_1)\)

\[
\begin{align*}
\text{ys}_0 &= (\text{ite (even? } x_0) '(x_0) '()) \\
\text{ys}_1 &= (\text{append } \text{ys}_0 '(x_1)) \\
\text{ys}_2 &= (\text{ite (even? } x_1) \text{ys}_1 \text{ys}_0)
\end{align*}
\]
The **symbolic evaluation graph** summarizes branching and merging.

**Symbolic execution**

- `(filter even? '(x_0 x_1))`
- `¬(even? x_0)`
- `(even? x_0)`
- `( )`
- `(x_1)`
- `(x_0)`
- `(x_0)`
- `(x_0 x_1)`

**Bounded model checking**

- `(filter even? '(x_0 x_1))`
- `¬(even? x_0)`
- `(even? x_0)`
- `( )`
- `(x_0)`
- `ys_0`
- `ys_1`
- `ys_2`
- `ys_0 = (ite (even? x_0) ( ) (x_0))`
- `ys_1 = (append ys_0 (x_1))`
- `ys_2 = (ite (even? x_1) ys_1 ys_0)`

More states, but more concrete

Fewer states but less concrete
The *symbolic heap* shows how symbolic values are used.

- **Symbolic execution**
  - Nodes are symbolic terms
  - Edges are sub-terms

- **Symbolic heap**
  - Nodes are symbolic terms
  - Edges are sub-terms
The \textit{symbolic heap} shows how symbolic values are used.

\begin{itemize}
  \item \textbf{Symbolic execution:}
    \begin{align*}
      \text{\symbolcell{\text{even? } x_0}} \land \text{\symbolcell{\text{even? } x_1}} & \rightarrow \text{\symbolcell{\text{even? } x_0}} \\
      \text{\symbolcell{\text{even? } x_1}} & \rightarrow \text{\symbolcell{\text{even? } x_1}}
    \end{align*}
  \\
  \text{Symbolic execution steps:}
  \begin{align*}
    y_{s0} &= \text{ite} \ (\text{even? } x_0) \ (\text{\symbolcell{\text{}()}}) \ (\text{\symbolcell{\text{}(x_0)})} \\
    y_{s1} &= \text{append} \ y_{s0} \ (\text{\symbolcell{\text{}(x_1)})} \\
    y_{s2} &= \text{ite} \ (\text{even? } x_1) \ y_{s1} \ y_{s0}
  \end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{Bounded model checking:}
    \begin{align*}
      y_{s0} &= \text{ite} \ (\text{even? } x_0) \ (\text{\symbolcell{\text{}()}}) \ (\text{\symbolcell{\text{}(x_0)})} \\
      y_{s1} &= \text{append} \ y_{s0} \ (\text{\symbolcell{\text{}(x_1)})} \\
      y_{s2} &= \text{ite} \ (\text{even? } x_1) \ y_{s1} \ y_{s0}
    \end{align*}
\end{itemize}
The **symbolic heap** shows how symbolic values are used

- **Symbolic execution**
  - Conditions and values (lists etc.) in the heap

- **Bounded model checking**
  - Only conditions in the heap

- **Example symbol manipulation**:
  - \( y_0 = \text{ite} (\text{even?} x_0) \)
  - \( y_1 = \text{append} \)
  - \( y_2 = \text{ite} (\text{even?} x_1) \)

- **Example symbolic expressions**:
  - \( y_0 = (\text{ite} (\text{even?} x_0) '() ' (x_0)) \)
  - \( y_1 = (\text{append} y_0 ' (x_1)) \)
  - \( y_2 = (\text{ite} (\text{even?} x_1) y_1 y_0) \)
## Analyzing symbolic data structures

### Call Stack

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>@filter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>first-k-even</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@take</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Function Details

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>4.3</td>
<td>1249</td>
<td>137408</td>
<td>131164</td>
<td>4288</td>
<td>93664</td>
</tr>
<tr>
<td>take</td>
<td>2.8</td>
<td>4692</td>
<td>50312</td>
<td>49986</td>
<td>2209</td>
<td>49986</td>
</tr>
<tr>
<td>andmap</td>
<td>0.3</td>
<td>94</td>
<td>14180</td>
<td>14180</td>
<td>0</td>
<td>4097</td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td>0.1</td>
<td>511</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Analyzing symbolic data structures

For each procedure, measure metrics that summarize the evolution of the symbolic evaluation graph and symbolic heap.

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter 1 call</td>
<td>4.3</td>
<td>1249</td>
<td>137408</td>
<td>131164</td>
<td>4288</td>
<td>93664</td>
</tr>
<tr>
<td>take 1 call</td>
<td>2.8</td>
<td>4692</td>
<td>50312</td>
<td>49986</td>
<td>2209</td>
<td>49986</td>
</tr>
<tr>
<td>andmap 1 call</td>
<td>0.3</td>
<td>94</td>
<td>14180</td>
<td>14180</td>
<td>0</td>
<td>4097</td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td>0.1</td>
<td>511</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Analyzing symbolic data structures

For each procedure, measure metrics that summarize the evolution of the symbolic evaluation graph and symbolic heap.

Summarize metrics as a score to rank procedures in the program.
Symbolic evaluation anti-patterns
Common issues and repairs
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch
  Algorithms or optimizations poorly suited to symbolic evaluation
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch

Algorithms or optimizations poorly suited to symbolic evaluation

\[
\text{(define (list-set lst idx val)}
\begin{align*}
\text{(match lst} & \text{[[ (cons x xs)} \\
& \text{ (if (= idx 0)} \\
& \quad \text{(cons val xs)} \\
& \quad \text{(cons x (list-set xs (- idx 1) val))]} \\
& \text{[_ lst]]})
\end{align*}
\]

Terminates early once \text{idx} is found
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch

Algorithms or optimizations poorly suited to symbolic evaluation

\[
\text{(define (list-set \textit{lst} \textit{idx} \textit{val})}
\]
\[
\text{(match \textit{lst}}
\]
\[
\text{[((cons \textit{x} \textit{x} \textit{xs})}
\]
\[
\text{\text{(if (= \textit{idx} 0)}}
\]
\[
\text{\text{(cons \textit{val} \textit{x} \textit{xs})}
\]
\[
\text{\text{(cons \textit{x} (list-set \textit{x} \textit{xs} (- \textit{idx} 1) \textit{val}))]
\]
\[
\text{[_ \textit{lst}]}
\]
\]

Terminates early once \textit{idx} is found
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch

Algorithms or optimizations poorly suited to symbolic evaluation

\[
(\textit{define} \ (\text{list-set} \ \text{lst} \ \textit{idx} \ \text{val})
\begin{align*}
&\quad \text{(match \ lst} \\
&\quad \quad [(\text{cons} \ x \ \text{xs}) \\
&\quad \quad \quad \text{(cons} \ \textit{if} \ (= \ \textit{idx} \ 0) \ \text{val} \ x) \\
&\quad \quad \quad \quad \text{(list-set} \ \text{xs} \ (- \ \textit{idx} \ 1) \ \text{val}))]
\end{align*}
\]

[_ lst]})}
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch

Algorithms or optimizations poorly suited to symbolic evaluation

```scheme
(define (list-set lst idx val)
  (match lst
    [(cons x xs)
     (cons (if (= idx 0) val x)
           (list-set xs (- idx 1) val))
    [_ lst]))
```

Always recurse to the end of `lst`
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch

Algorithms or optimizations poorly suited to symbolic evaluation

(\textbf{define} (list-set lst \textbf{idx} val)
  (match lst
    [(cons x xs)
      (cons (if (= idx 0) val x)
        (list-set xs (- idx 1) val))]
    [_ lst]))

Always recurse to the end of \textit{lst}
Common anti-patterns and repairs in symbolic evaluation

Algorithmic mismatch
  Algorithms or optimizations poorly suited to symbolic evaluation

Irregular representation
  Data structures of different shapes create different paths

Missed concretization
  Lost opportunities to exploit concrete values
Empirical results
Case studies and evaluation
Three symbolic profilers

We developed two implementations:

• The **Rosette** solver-aided language (Racket)
• The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

• The **Crucible** symbolic simulation library (C, Java, ...) by Galois
Three symbolic profilers

We developed two implementations:

• The **Rosette** solver-aided language (Racket)
• The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

• The **Crucible** symbolic simulation library (C, Java, ...) by Galois
## Actionable: real-world bugs

Case studies on published Rosette-based tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type system soundness checker [POPL’18]</td>
<td>1.35×</td>
</tr>
<tr>
<td>Refinement type checker for Ruby [VMCAI’18]</td>
<td>6×</td>
</tr>
<tr>
<td>File-system crash consistency verifier [ASPLOS’16]</td>
<td>24×</td>
</tr>
<tr>
<td>Cryptographic protocol verifier [FM’18]</td>
<td>29×</td>
</tr>
<tr>
<td>SQL query verifier [CIDR’17]</td>
<td>75×</td>
</tr>
<tr>
<td>Safety-critical radiotherapy system verifier [CAV’16]</td>
<td>290×</td>
</tr>
</tbody>
</table>

Multiple patches accepted by developers
## Actionable: real-world bugs

Case studies on published Rosette-based tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type system soundness checker [POPL’18]</td>
<td>1.35×</td>
</tr>
<tr>
<td>Refinement type checker for Ruby [VMCAI’18]</td>
<td>6×</td>
</tr>
<tr>
<td>File-system crash consistency verifier [ASPLOS’16]</td>
<td>24×</td>
</tr>
<tr>
<td>Cryptographic protocol verifier [FM’18]</td>
<td>29×</td>
</tr>
<tr>
<td>SQL query verifier [CIDR’17]</td>
<td>75×</td>
</tr>
<tr>
<td>Safety-critical radiotherapy system verifier [CAV’16]</td>
<td>290×</td>
</tr>
</tbody>
</table>

Multiple patches accepted by developers

Used in production at the UW Medical Center
Small user study: 8 Rosette users, asked to find known performance bug in 4 programs

Users solved every task more quickly when they had access to symbolic profiling

6 failures without symbolic profiling, none with

Qualitative feedback:
“gave insight into what Rosette is doing”
“even more useful on my own code”
Symbolic profiling identifies performance issues in symbolic evaluation

Does my program work on all inputs?
Verification

Is there a program that does what I want?
Synthesis

raco symprofile file.rkt

https://unsat.org