TYPE-GUIDED WORST-CASE INPUT GENERATION

Di Wang, Jan Hoffmann
Carnegie Mellon University
RESOURCE ANALYSIS

Programs
RESOURCE ANALYSIS

Programs

Performance
RESOURCE ANALYSIS

Programs

Performance

- Time
- Memory
- Power
- ...
RESOURCE ANALYSIS

- Performance bottlenecks
- Algorithmic complexity vulnerabilities
- Timing side channels

Programs → Performance
- Time
- Memory
- Power
- ...

Worst-Case Analysis
EXAMPLE OF WORST-CASE ANALYSIS

EXAMPLE OF WORST-CASE ANALYSIS

Potential Denial-of-Service attack

EXAMPLE OF WORST-CASE ANALYSIS

Potential Denial-of-Service attack

Concrete exploits (by hash collisions)

EXAMPLE OF WORST-CASE ANALYSIS

Potential Denial-of-Service attack

Concrete exploits (by hash collisions)

Example of Worst-Case Analysis

Potential Denial-of-Service attack

Worst-case inputs are instrumental to understand and fix performance bugs!

Concrete exploits (by hash collisions)

EXISTING APPROACHES
EXISTING APPROACHES

Dynamic

- Fuzz testing
- Symbolic execution
- Dynamic worst-case analysis
- …

- Flexible & universal
- Potentially unsound: The resulting inputs might not expose the worst-case behavior.
EXISTING APPROACHES

Dynamic

- Fuzz testing
- Symbolic execution
- Dynamic worst-case analysis
- ...

- Flexible & universal
- Potentially unsound: The resulting inputs might not expose the worst-case behavior.

Static

- Type systems
- Abstract interpretation
- ...

- Sound upper bounds
- Potentially not tight: No concrete witness — the bound might be too conservative.
Contributions

- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability
CONTRIBUTIONS

- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability
CONTRIBUTIONS

- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)
Contributions

- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)
CONTRIBUTIONS

- A type-guided **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability
Contributions

- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)

Guide

Symbolic Execution
OVERVIEW

- Motivation
- Resource Aware ML (RaML)
- Type-Guided Worst-Case Input Generation
- Evaluation
Amortized Resource Analysis

- The potential method
Amortized Resource Analysis

The potential method
**Amortized Resource Analysis**

- The potential method

![Diagram](image)
**Amortized Resource Analysis**

- The potential method

- \( D_0 \)’s are program states
- Arrows are transitions with actual costs
**Amortized Resource Analysis**

- The potential method

\[ \Phi(D_0), \Phi(D_1), \Phi(D_2), \Phi(D_3), \ldots, \Phi(D_n) \]

- Di’s are program states
- Arrows are transitions with actual costs
Amortized Resource Analysis

The potential method

$D_i$'s are program states

Arrows are transitions with actual costs

$\Phi(D_0)$  $\Phi(D_1)$  $\Phi(D_2)$  $\Phi(D_3)$  $\Phi(D_n)$

The potential function maps program states to nonnegative numbers
AMORTIZED RESOURCE ANALYSIS

The potential method

\[ \Phi(D_0) \quad \Phi(D_1) \quad \Phi(D_2) \quad \Phi(D_3) \quad \Phi(D_n) \]

\[ \Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3) \]

Diagrams:
- \( D_i \)'s are program states
- Arrows are transitions with actual costs
- The potential function maps program states to nonnegative numbers
The potential method

$\Phi(D_0)$ $\Phi(D_1)$ $\Phi(D_2)$ $\Phi(D_3)$ $\Phi(D_n)$

$D_i$’s are program states

Arrows are transitions with actual costs

$\Phi(D_2) \geq \text{Cost}(D_2, D_3) + \Phi(D_3)$

The initial potential is an upper bound!

The potential function maps program states to nonnegative numbers
The potential at a program point is defined by a **static** annotation of data structures.

A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.
**Type-Based Analysis**

```ocaml
let rec lpairs l = match l with
| [] -> []
| x1 :: xs -> match xs with
  | [] -> []
  | x2 :: xs' ->
    if (x1:int) < (x2:int) then (x1, x2) :: lpairs xs'
    else lpairs xs'
```

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.

Each of $[\ ]$, $::$, $(\ )$ consumes 2 memory cells.
Type-Based Analysis

\[ \text{Cost} = 2 \cdot |\ell| + 2 \]

```coq
let rec lpairs l =  
  match l with  
  | [] -> []  
  | x1 :: xs ->  
    match xs with  
    | [] -> []  
    | x2 :: xs' ->  
      if (x1:int) < (x2:int) then  
        (x1, x2) :: lpairs xs'  
      else  
        lpairs xs'  
```

- The **potential** at a program point is defined by a `static` annotation of data structures.
- A list of length \( n \) annotated with a nonnegative number \( q \) has \( q \cdot n \) units of potential.

Each of `[], ::, (,) consumes 2 memory cells.
Type-Based Analysis

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

let rec lpairs l =
match l with
| [] -> []
| x1 :: xs ->
  match xs with
  | [] -> []
  | x2 :: xs' ->
    if (x1:int) < (x2:int) then
      (x1, x2) :: lpairs xs'
    else
      lpairs xs'

Each of [], ::, (,) consumes 2 memory cells.

- The potential at a program point is defined by a static annotation of data structures.
- A list of length \(n\) annotated with a nonnegative number \(q\) has \(q \cdot n\) units of potential.
The potential at a program point is defined by a static annotation of data structures.

A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.

Each of $[\ ], :, (, )$ consumes 2 memory cells.
The potential at a program point is defined by a static annotation of data structures.

A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.
The potential at a program point is defined by a static annotation of data structures.

A list of length \( n \) annotated with a nonnegative number \( q \) has \( q \cdot n \) units of potential.

Each of \([], ::, (,)\) consumes 2 memory cells.
The potential at a program point is defined by a \textit{static} annotation of data structures.

A list of length \( n \) annotated with a nonnegative number \( q \) has \( q \cdot n \) units of potential.

Each of \([], ::, (,)\) consumes 2 memory cells.
**Type-Based Analysis**

The potential at a program point is defined by a **static** annotation of data structures.

A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.

Each of [], ::, (,) consumes 2 memory cells.
Let rec lpairs l = match l with | [] -> [] | x1 :: xs -> match xs with | [] -> [] | x2 :: xs' -> if (x1:int) < (x2:int) then (x1, x2) :: lpairs xs' else lpairs xs'

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

The potential at a program point is defined by a static annotation of data structures.

A list of length \( n \) annotated with a nonnegative number \( q \) has \( q \cdot n \) units of potential.

Each of \([], ::, (\,)\) consumes 2 memory cells.
The potential at a program point is defined by a static annotation of data structures.

A list of length $n$ annotated with a nonnegative number $q$ has $q \cdot n$ units of potential.

Each of $[\ ], ::, (, )$ consumes 2 memory cells.
OVERVIEW

- Motivation
- Resource Aware ML (RaML)
  - Type-Guided Worst-Case Input Generation
  - Evaluation
SYMBOLIC EXECUTION

- **Idea**: search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]
**Symbolic Execution**

- **Idea:** search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]
Symbolic Execution

**Idea:** search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]

symbolic environment

expression
Symbolic Execution

- **Idea**: search all execution paths, record path constraints, and compute resource usage

\[ \gamma \triangleright e \Rightarrow \langle \psi, S \rangle \]

- symbolic environment
- expression
- path constraints
Symbolic Execution

- **Idea**: search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]

- symbolic environment
- expression
- path constraints
- symbolic evaluation result
Symbolic Execution

- **Idea:** search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]

- Symbolic environment
- Expression
- Path constraints
- Symbolic evaluation result

- Symbolic execution rules for conditional expressions
Symbolic Execution

- **Idea:** search all execution paths, record path constraints, and compute resource usage

\[ \gamma \vdash e \Rightarrow \langle \psi, S \rangle \]

- Symbolic execution rules for conditional expressions

  **Then** \[ \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \]
  \[ \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \]

  **Else** \[ \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \]
  \[ \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle \]
let rec lpairs l =
match l with
    | [] -> []
    | x1 :: xs ->
        match xs with
            | [] -> []
            | x2 :: xs' ->
                if (x1:int) < (x2:int) then
                    (x1, x2) :: lpairs xs'
                else
                    lpairs xs'
**Symbolic Execution**

An example of worst-case execution paths for input lists of length 4

\[ \ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4] \models \]

\[ \text{lpairs } \ell \Rightarrow (\text{int}^1 < \text{int}^2) \land (\text{int}^3 < \text{int}^4), \]
\[ [(\text{int}^1, \text{int}^2), (\text{int}^3, \text{int}^4)] \]
Symbolic Execution

An example of worst-case execution paths for input lists of length 4

\[
\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4] \vdash \\
\text{lpairs } \ell \Rightarrow ((\text{int}^1 < \text{int}^2) \land (\text{int}^3 < \text{int}^4), \\
[(\text{int}^1, \text{int}^2), (\text{int}^3, \text{int}^4)])
\]

Invoke an SMT solver to find a model, e.g., \([0, 1, 0, 1]\)
**Type-Guided Symbolic Execution**

- **Nondeterminism** leads to state explosion

\[
\begin{align*}
\text{Then} & \quad \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \\
\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \\
\text{Else} & \quad \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \\
\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle
\end{align*}
\]
**Type-Guided Symbolic Execution**

- **Nondeterminism** leads to state explosion
  
  
  **Then** \( \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \)
  
  \( \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \)
  
  **Else** \( \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \)
  
  \( \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle \)

  Use the information about **potentials** obtained from **resource aware type checking** to **prune** the search space of symbolic execution.
**Type-Guided Symbolic Execution**

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

```ocaml
let rec lpairs l =
  match l with
  | []    -> []
  | x1 :: xs ->
    match xs with
    | []    -> []
    | x2 :: xs' ->
      if (x1:int) < (x2:int) then
        (x1, x2) :: lpairs xs'
      else
        lpairs xs'
```

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]
TYPE-GUIDED SYMBOLIC EXECUTION

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

``` Ocaml 
let rec lpairs l = match l with |
| [] -> [] |
| x1 :: xs -> match xs with |
| [] -> [] |
| x2 :: xs' -> if (x1:int) < (x2:int) then (x1, x2) :: lpairs xs'
else lpairs xs'
```

\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]
**Type-Guided Symbolic Execution**

\[
L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})
\]

```ocaml
let rec lpairs l =
  match l with
  | [] -> []
  | x1 :: xs ->
    match xs with
    | [] -> []
    | x2 :: xs' ->
      if (x1:int) < (x2:int) then
        (x1, x2) :: lpairs xs'
      else
        lpairs xs'
```

\[\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]\]

\[\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]\]

\[\ell' \mapsto [\text{int}^3, \text{int}^4]\]
TYPE-GUIDED SYMBOLIC EXECUTION

\[
L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})
\]

let rec lpairs l =
match l with
| [] -> []
| x1 :: xs ->
match xs with
| [] -> []
| x2 :: xs' ->
if (x1:int) < (x2:int) then
  (x1, x2) :: lpairs xs'
else
  lpairs xs'

\[ \Phi_2 = 2 \cdot |xs'| + 6 = 10 \]

\[ x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4] \]
TYPE-GUIDED SYMBOLIC EXECUTION

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1 :: xs ->
    match xs with
    | [] -> []
    | x2 :: xs' ->
      if (x1:int) < (x2:int) then
        (x1, x2) :: lpairs xs'
      else
        lpairs xs'

Φ₂ = 2 \cdot |xs'| + 6 = 10
```

\( x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto \text{int}^3, \text{int}^4 \)

Cost = 4
**Type-Guided Symbolic Execution**

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

\[
\text{let rec } \text{lpairs} \ l = \begin{cases} 
\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4] \\
\end{cases} 
\]

\[
\begin{align*}
\Phi_2 &= 2 \cdot |xs'| + 6 = 10 \\
x_1 &\mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4] \\
\end{align*}
\]

\[
\Phi_3 = 2 \cdot |xs'| + 2 = 6
\]

Cost = 4
**Type-Guided Symbolic Execution**

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

\[
\begin{align*}
\text{let rec } & \text{lpairs } l = \\
\text{match } & l \text{ with } \\
| \text{[]} & \rightarrow \text{[]} \\
| \text{x1 :: xs } & \rightarrow \\
\text{match } & xs \text{ with } \\
| \text{[]} & \rightarrow \text{[]} \\
| \text{x2 :: xs'} & \rightarrow \\
\text{if } & (x1 : \text{int}) < (x2 : \text{int}) \text{ then } \\
& (x1, x2) :: \text{lpairs xs'} \\
\text{else } & \text{lpairs xs'}
\end{align*}
\]

\[ \ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4] \]

\[ \Phi_2 = 2 \cdot |xs'| + 6 = 10 \]

\[ x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4] \]

\[ \Phi_3 = 2 \cdot |xs'| + 2 = 6 \]

\[ \text{Cost} = 4 \]
**Type-Guided Symbolic Execution**

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

```
let rec lpairs l = ℓ ↦ [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]
match l with
| [] -> []
| x1 :: xs ->
  match xs with
  | [] -> []
  | x2 :: xs' ->
    if (x1:int) < (x2:int) then
      (x1, x2) :: lpairs xs'
    else
      lpairs xs'
```

Cost = 4

Waste!

\[ \Phi_2 = 2 \cdot |xs'| + 6 = 10 \]

\[ \Phi_3 = 2 \cdot |xs'| + 2 = 6 \]
If an execution path does not have potential waste, it must expose the worst-case resource usage.

Cost = 4

Waste!
**Type-Guided Symbolic Execution**

If an execution path does not have potential waste, it must expose the worst-case resource usage.

\[ L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int}) \]

\[
\begin{align*}
\text{let rec } & \text{lpairs } l = \ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4] \\
\text{match } l \text{ with } & \\
& | [] \rightarrow [] \\
& | x1 :: xs \rightarrow \\
\text{match } & xs \text{ with } \\
& | [] \rightarrow [] \\
& | x2 :: xs' \rightarrow \\
\text{if } & (x1: \text{int}) < (x2: \text{int}) \text{ then } \\
& (x1, x2) :: \text{lpairs } xs' \\
\text{else } & \text{lpairs } xs' \\
\end{align*}
\]

Cost = 4

Waste!
Soundness & Completeness
Soundness & Completeness

Soundness: If the algorithm generates an input, then the input will cause the program to consume exactly the same amount of resource as the inferred upper bound (by RaML).
**Soundness & Completeness**

- **Soundness**: If the algorithm generates an input, then the input will cause the program to consume exactly the same amount of resource as the inferred upper bound (by RaML).

- **Relative completeness**: If there is an input of some given shape that causes the program to consume exactly the same amount of resource as the inferred upper bound (by RaML), then the algorithm is able to find a corresponding execution path.
SPEED UP INPUT GENERATION

\[ \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \]
\[ \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \]

\[ \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \]
\[ \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle \]
How about **eliminating** some generation rules?

<table>
<thead>
<tr>
<th>Then</th>
<th>γ ⊢ e₁ ⇒ ⟨ψ, S⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ ⊢ if e then e₁ else e₂ ⇒ ⟨γ(e) ∧ ψ, S⟩</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Else</th>
<th>γ ⊢ e₂ ⇒ ⟨ψ, S⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ ⊢ if e then e₁ else e₂ ⇒ ⟨¬γ(e) ∧ ψ, S⟩</td>
<td></td>
</tr>
</tbody>
</table>
How about **eliminating** some generation rules?

\[
\begin{align*}
\text{Then:} & \quad \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \\
& \quad \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \\
\text{Else:} & \quad \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \\
& \quad \gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle
\end{align*}
\]
How about eliminating some generation rules?

\[
\begin{align*}
\text{Then} & \quad \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \\
\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \\
\text{Else} & \quad \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \\
\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle
\end{align*}
\]

Still Sound!
How about eliminating some generation rules?

\[
\begin{align*}
&\text{Then} & \gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle \\
&\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \land \psi, S \rangle \\
\end{align*}
\]

\[
\begin{align*}
&\text{Else} & \gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle \\
&\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg \gamma(e) \land \psi, S \rangle \\
\end{align*}
\]

Still Sound!

Generalization: enforce all the calls with the same shape of inputs execute the same path in the function body.
OVERVIEW

- Motivation
- Resource Aware ML (RaML)
- Type-Guided Worst-Case Input Generation
- Evaluation
We implemented the generation algorithm for a purely functional fragment of Resource Aware ML (RaML), including higher-order functions, user-defined data structures, and polynomial resource bounds.

We used the off-the-shelf SMT solver Z3.
## Benchmarks (Selected)

<table>
<thead>
<tr>
<th>Description</th>
<th>Shape</th>
<th>ALG</th>
<th>ALG+H1</th>
<th>ALG+H2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>200 integers</td>
<td>7.74s</td>
<td>6.97s</td>
<td>94.81s</td>
</tr>
<tr>
<td>Quicksort</td>
<td>200 integers</td>
<td>T/O</td>
<td>53.23s</td>
<td>157.2s</td>
</tr>
<tr>
<td>Lexicographic quicksort</td>
<td>Lists of length 100, 99, ..., 1</td>
<td>439.35s</td>
<td>438.79s</td>
<td>T/O</td>
</tr>
<tr>
<td>Functional queue</td>
<td>200 operations</td>
<td>444.64s</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td>Zigzag on a tree</td>
<td>200 internal nodes</td>
<td>T/O</td>
<td>T/O</td>
<td>4.87s</td>
</tr>
<tr>
<td>Hash table for 8-char strings</td>
<td>64 insertions</td>
<td>7.64s</td>
<td>7.62s</td>
<td>181.74s</td>
</tr>
</tbody>
</table>
Example: Hash Table
EXAMPLE: HASH TABLE

- Customized resource metric: count for hash collisions
EXAMPLE: HASH TABLE

- Customized resource metric: count for hash collisions
- Use a hash function from a vulnerable PHP implementation
Example: Hash Table

- Customized resource metric: count for hash collisions
- Use a hash function from a vulnerable PHP implementation
- The program inserts 64 strings into an empty hash table
EXAMPLE: HASH TABLE

- Customized resource metric: count for **hash collisions**
- Use a hash function from a vulnerable PHP implementation
- The program inserts 64 strings into an empty hash table
- Our algorithm “**realizes**” that it should find 64 strings with the same hash key, in order to trigger the most collisions
SUMMARY

TYPE-GUIDEDシンボリック実行

FOR WORST-CASE INPUT GENERATION
SUMMARY

TYPE-GUIDED SYMBOLIC EXECUTION FOR WORST-CASE INPUT GENERATION

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results
Summary

Type-Guided Symbolic Execution for Worst-Case Input Generation

- Formally developed algorithm
- Soundness & relative completeness

- Integrated with RaML
- Effective on 22 benchmark programs

Theoretical Results

Experimental Results
**Summary**

**Limitations:**
- Purely functional programs
- Only work for tight bounds
- Depend on RaML

---

**Type-Guided Symbolic Execution for Worst-Case Input Generation**

**Theoretical Results**
- Formally developed algorithm
- Soundness & relative completeness

**Experimental Results**
- Integrated with RaML
- Effective on 22 benchmark programs
Limitations:
- Purely functional programs
- Only work for tight bounds
- Depend on RaML

Future work:
- Support side effects
- Interact with resource analysis
- General theory for worst-case analysis

Type-Guided Symbolic Execution for Worst-Case Input Generation

Theoretical Results
- Formally developed algorithm
- Soundness & relative completeness

Experimental Results
- Integrated with RaML
- Effective on 22 benchmark programs