Inference of Robust Reachability Constraints

Yanis Sellami\textsuperscript{1,2}, Guillaume Girol\textsuperscript{2}, Frédéric Recoules\textsuperscript{2}, Damien Couroussé\textsuperscript{1}, Sébastien Bardin\textsuperscript{2}

\textsuperscript{1} Univ. Grenoble Alpes, CEA List, France
\textsuperscript{2} Université Paris-Saclay, CEA List, France
Automatic Bug Detection

Programs have bugs

Bugs can be exploited $\rightarrow$ Vulnerabilities

```c
void f() {
    uint a, b = read();
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

We need automated methods to detect bugs
Automatic Bug Detection

Programs have bugs

Bugs can be exploited → Vulnerabilities

```c
void f() {
    uint a, b = read();
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

We need automated methods to detect bugs

Example: Symbolic Execution

- Explore the program paths
- Finds program input that exhibits the bug
- Sound: no false positives
Automatic Bug Detection

Programs have bugs

Bugs can be exploited $\rightarrow$ Vulnerabilities

Example: Symbolic Execution

- Explore the program paths
- Finds program input that exhibits the bug
- Sound: no false positives

```c
void f() {
    uint a, b = read();
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

We need automated methods to detect bugs

$a = 0, b = 0$
False Positive in Practice

Example

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0)
        /* bug */
    else
        ...
}
```
False Positive in Practice

Example

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

Symbolic Execution?
- Very easy: $a = 0$, $b = 0$
False Positive in Practice

Example
```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

Symbolic Execution?
- Very easy: $a = 0, b = 0$

The Issue
- Depends on uncontrolled initial value ($b$)
- The formal result is not reliably reproducible
False Positive in Practice

Example

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0)
        /* bug */
    else
        ...
}
```

Symbolic Execution?
- Very easy: $a = 0, b = 0$

The Issue
- Depends on uncontrolled initial value ($b$)
- The formal result is not reliably reproducible

Practical Causes of Unreliable Assignments
- Interaction with the environment
- Stack canaries
- Uninitialized memory/register dependency
- Choice of undefined behaviors

We need to characterize the replicability of bugs
Robust Reachability
[Farinier et. al., CAV 2018; Girol et. al., CAV 2021]

Idea

• Partition of the input space
  • What is controlled
  • What is uncontrolled

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0) /* bug */
      else
        ...
}
```

controlled  
uncontrolled

a  
b
Robust Reachability
[Farinier et. al., CAV 2018; Girol et. al., CAV 2021]

Idea

- Partition of the input space
  - What is controlled
  - What is uncontrolled

Focus: Reliable Bugs

- Controlled input that triggers the bug independently of the value of the uncontrolled inputs

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0)
        /* bug */
    else
        ...
}

controlled

∃ a

∀ b

error

uncontrolled
Robust Reachability  
[Farinier et. al., CAV 2018; Girol et. al., CAV 2021]

Idea

• Partition of the input space
  • What is controlled
  • What is uncontrolled

Focus: Reliable Bugs

• Controlled input that triggers the bug independently of the value of the uncontrolled inputs

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0) /* bug */
        ...
    else
        ...
}
```

∃ a  ∀ b  error

Not Robustly Reachable
**Robust Reachability**  
*[Farinier et. al., CAV 2018; Girol et. al., CAV 2021]*

**Idea**
- Partition of the input space
  - What is controlled
  - What is uncontrolled

**Focus: Reliable Bugs**
- Controlled input that triggers the bug independently of the value of the uncontrolled inputs

**Extension of Reachability and Symbolic Execution**

```c
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if (a + b == 0) /* bug */
        ... else
    ...}
```

\[\exists a \forall b \text{ error} \]

Not Robustly Reachable
The Remaining Problem

Example 3

- Memcopy with slow and fast path
- Fast path is buggy but slow path is not

```c
typedef struct { unsigned char bytes[32]; } uint256_t;
void memcopy(void* dst, const void* src, size_t n) {
    if (((dst | src | n) & 0b111111))
        /* slow path */
        for (size_t i = 0; i < n; i += 1)
            dst[i] = src[i];
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            (uint256_t*)dst[i] = (uint256_t*)src[i];
}
```
The Remaining Problem

Example 3

- Memcopy with slow and fast path
- Fast path is buggy but slow path is not

```c
typedef struct { unsigned char bytes[32]; } uint256_t;
void memcpy(void* dst, const void* src, size_t n) {
    if (((dst | src | n) & 0b111111)) /* slow path */
        for (size_t i = 0; i < n; i += 1)
            dst[i] = src[i];
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            (uint256_t*)dst[i] = (uint256_t*)src[i];
}
```
The Remaining Problem

Example 3

• Memcopy with slow and fast path
• Fast path is buggy but slow path is not
• Reachability: Vulnerable

```c
typedef struct { unsigned char bytes[32]; } uint256_t;
void memcpy(void* dst, const void* src, size_t n) {
    if (((dst | src | n) & 0b11111))
        /* slow path */
            for (size_t i = 0; i < n; i += 1)
                dst[i] = src[i];
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            (uint256_t*)dst[i] = (uint256_t*)src[i];
}
```
The Remaining Problem

Example 3

- Memcopy with slow and fast path
- Fast path is buggy but slow path is not
- Reachability: Vulnerable

```c
typedef struct { unsigned char bytes[32]; } uint256_t;
void memcpy(void* dst, const void* src, size_t n) {
    if (((dst | src | n) & 0b111111))
        /* slow path */
        for (size_t i = 0; i < n; i += 1)
            dst[i] = src[i];
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            (uint256_t*)dst[i] = (uint256_t*)src[i];
}
```
The Remaining Problem

Example 3

- Memcopy with slow and fast path
- Fast path is buggy but slow path is not
- Reachability: Vulnerable
- Robust Reachability: Not reliably triggerable
  - Taking the fast path depends on uncontrolled initial values

The bug is serious but not robustly reachable – The concept is too strong
Robust Reachability Constraints

Definition

• Predicate on program input sufficient to have Robust Reachability

```c
typedef struct { unsigned char bytes[32]; } uint256_t;

void memcpy(void* dst, const void* src, size_t n) {
    if (((dst | src | n) & 0b111111))
        /* slow path */
        for (size_t i = 0; i < n; i += 1)
            dst[i] = src[i];
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            (uint256_t*)dst[i] = (uint256_t*)src[i];
}
```
Robust Reachability Constraints

Definition

• Predicate on program input sufficient to have Robust Reachability

\[ \exists * \text{src}, \forall \text{src, dst}, \text{src} \% 32 = 0 \land \text{dst} \% 32 = 0 \Rightarrow \text{overflow} \]

(src and dst aligned on 32bits)
Robust Reachability Constraints

Definition

• Predicate on program input sufficient to have Robust Reachability

Advantages

• Part of the Robust Reachability framework
• Allows precise characterization

\[
\exists \ src, \ \forall \ src, \ dst, \ src \mod 32 = 0 \land dst \mod 32 = 0 \Rightarrow \text{overflow}
\]

(src and dst aligned on 32bits)
Robust Reachability Constraints

Definition

• Predicate on program input sufficient to have Robust Reachability

Advantages

• Part of the Robust Reachability framework
• Allows precise characterization

How to Automatically Generate Such Constraints?

\[
\exists * \ src, \ \forall src, \ dst, \ src \ % \ 32 = 0 \ \land \ dst \ % \ 32 = 0 \ \Rightarrow \ \text{overflow}
\]

(src and dst aligned on 32bits)
Contributions

- New program-level abduction algorithm for Robust Reachability Constraints Inference
  - Extends and generalizes Robustness, made more practical
  - Adapts and generalizes theory-agnostic logical abduction algorithm
  - Efficient optimization strategies for solving practical problems

- Implementation of a restriction to Reachability and Robust Reachability
  - First evaluation of software verification and security benchmarks
  - Detailed vulnerability characterization analysis in a fault injection security scenario

Target: Computation of $\phi$ such that $\exists C$ controlled value, $\forall U$ uncontrolled value, $\phi(C, U) \Rightarrow reach(C, U)$
Abduction of Robust Reachability Constraints

Abductive Reasoning

[Josephson and Josephson, 1994]

- Find missing precondition of unexplained goal
- Compute $\phi_M$ in $\phi_H \land \phi_M \models \phi_G$
Abduction of Robust Reachability Constraints

Abductive Reasoning
[Josephson and Josephson, 1994]
- Find missing precondition of unexplained goal
- Compute $\phi_M$ in $\phi_H \land \phi_M \models \phi_G$

Theory-Specific Abduction
[Bienvenu 2007, Tourret et. al. 2017]
- Handle a single theory

Specification Synthesis
- White-box program analysis
Abduction of Robust Reachability Constraints

Abductive Reasoning
[Josephson and Josephson, 1994]

- Find missing precondition of unexplained goal
- Compute $\phi_M$ in $\phi_H \land \phi_M \models \phi_G$

Theory-Specific Abduction
[Bienvenu 2007, Tourret et. al. 2017]

- Handle a single theory

Specification Synthesis

- White-box program analysis

Theory-Agnostic First-order Abduction
[Echenim et al. 2018, Reynolds et al. 2020]

- Efficient procedures
- Genericity
Abduction of Robust Reachability Constraints

Abductive Reasoning
[Josephson and Josephson, 1994]
• Find missing precondition of unexplained goal
• Compute $\phi_M$ in $\phi_H \land \phi_M \models \phi_G$

Theory-Specific Abduction
[Bienvenu 2007, Tourret et. al. 2017]
• Handle a single theory

Specification Synthesis
• White-box program analysis

Theory-Agnostic First-order Abduction
[Echenim et al. 2018, Reynolds et al. 2020]
• Efficient procedures
• Genericity

Our Proposal: Adapt Theory-Agnostic Abduction Algorithm to Compute Program-level Robust Reachability Constraints
• Program-level
• Generic
Our Solution (Framework)

\[ \mathcal{G} \text{ Inference Language (Set of Candidates)} \rightarrow \mathcal{P} \text{ Program} \rightarrow \psi \text{ Target Trace Predicate} \rightarrow \mathcal{A}_C \text{ Memory Partition} \rightarrow \text{Abduction Procedure} \]
Inference of Robust Reachability Constraints

Our Solution (Framework)

\[ G \] Inference Language (Set of Candidates)

\[ \rightarrow P \] Program

\[ \psi \] Target Trace Predicate

\[ A_C \] Memory Partition

Abduction Procedure
Our Solution (Framework)

\[ G \] Inference Language (Set of Candidates)

\[ \rightarrow P \] Program

\[ \psi \] Target Trace Predicate

\[ A_C \] Memory Partition

Abduction Procedure

select candidate
Our Solution (Framework)

\[
G \rightarrow P \quad \text{Inference Language (Set of Candidates)}
\]

\[
\psi \quad \text{Target Trace Predicate}
\]

\[
A_{C} \quad \text{Memory Partition}
\]

Abduction Procedure

- select candidate
- test candidate
- solution
- not solution

Robust Reachability Constraints
Our Solution (Framework)

Inference Language
(Set of Candidates)

Program

Target Trace Predicate

Memory Partition

Abduction Procedure

select candidate

not solution

solution

Robust Reachability Constraints
Our Solution (Framework)

Inference Language (Set of Candidates)

→ P Program

ψ Target Trace Predicate

 lehetőség

A_c Memory Partition

Abduction Procedure

select candidate

Oracles on Trace Properties

• Robust property queries
• Non-robust property queries
• Can accommodate various tools (SE, BMC, Incorrectness, ...)

Robust Reachability Constraints
Our Solution (Baseline Algorithm)

\[
\text{BaselineRCInfer}(G, \rightarrow_P, \psi, \mathcal{A}_C)
\]

1. if $T, s \leftarrow O^{3\exists}(\rightarrow_P, \psi, T)$ then
2. $R \leftarrow \{y = s\}$ if $y = s \in G$ else $\emptyset$;
3. for $\phi \in G$ do
4.  \quad if $O^{3\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ then
5.  \quad \quad $R \leftarrow \Delta_{\text{min}}(R \cup \{\phi\})$;
6.  \quad \quad if $\neg O^{3\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi' \in R} \phi'))$ then
7.  \quad \quad \quad return $R$;
8. return $R$;
9. return $\{\bot\}$;

Theorem:

- **Termination** when the oracles terminate
- **Correction** at any step when the oracles are correct
- **Completeness** w.r.t. the inference language when the oracles are complete
Our Solution (Baseline Algorithm)

**Theorem:**

- **Termination** when the oracles terminate
- **Correction** at any step when the oracles are correct
- **Completeness** w.r.t. the inference language when the oracles are complete
- Under correction and completeness of the oracles
  - **Minimality** w.r.t. the inference language
  - **Weakest** constraint generation when expressible

```plaintext
baselineRcInfer(G, →_p, ψ, A_C)
1 if T, s ← O^∃∃(→_p, ψ, T) then
2 R ← \{y = s\} if y = s ∈ G else \∅;
3 for φ ∈ G do
4 if O^∃∀(→_p, A_C, ψ, φ) then
5 R ← Δ_min(R ∪ \{φ\});
6 if O^∃∃(→_p, ψ, ¬(∀φ' ∈ R φ')) then
7 return R;
8 return R;
9 return \{⊥\};
```
Making it Work

The Issue

- Exhaustive exploration of the inference language is inefficient

Key Strategies for Efficient Exploration

- Necessary constraints
- Counter-examples for Robust Reachability
- Ordering candidates
Making it Work: Necessary Constraints

The Idea

- Find and store Necessary Constraints
Making it Work: Necessary Constraints

The Idea

• Find and store Necessary Constraints
Making it Work: Necessary Constraints

The Idea
- Find and store Necessary Constraints

Usage
- Build a candidate solution faster
- Additional information on the bug
- Emulate unsat core usage in the context of oracles
Making it Work: Counter-Examples

The Idea

• Reuse information from failed candidate checks

The Issue

• Non Robustness (∀∃ quantification) does not give us counter-examples
Making it Work: Counter-Examples

The Idea
- Reuse information from failed candidate checks

The Issue
- Non Robustness (∀∃ quantification) does not give us counter-examples

Proposal
- Use a second trace property that ensures the bug does not arise
- Prune using these counter-examples
Experimental Evaluation

Implementation

- (Robust) Reachability on binaries
- Tool: **BINSEC** [Djoudi and Bardin 2015]
- Tool: **BINSEC/RSE** [Girol at. al. 2020]

Prototype

- **PyAbd**, Python implementation of the procedure
- Candidates: Conjunctions of equalities and disequalities on memory bytes

Research Questions

1) Can we compute non-trivial constraints?
2) Can we compute weakest constraints?
3) What are the algorithmic performances?
4) Are the optimization effective?

Benchmarks

- Software verification (SVComp extract + compile)
- Security evaluation (FISSC, fault injection)
Results: Generating Constraints

<table>
<thead>
<tr>
<th></th>
<th>SV-COMP ($E_G$)</th>
<th>SV-COMP ($I_G$)</th>
<th>FISSC ($E_G$)</th>
<th>FISSC ($I_G$)</th>
</tr>
</thead>
<tbody>
<tr>
<td># programs</td>
<td>147</td>
<td>64</td>
<td>719</td>
<td>719</td>
</tr>
<tr>
<td># of robust cases</td>
<td>111</td>
<td>3</td>
<td>129</td>
<td>129</td>
</tr>
<tr>
<td># of sufficient rrc</td>
<td>122</td>
<td>5</td>
<td>359</td>
<td>351</td>
</tr>
<tr>
<td># of weakest rrc</td>
<td>111</td>
<td>3</td>
<td>262</td>
<td>261</td>
</tr>
</tbody>
</table>

Inference languages

• (dis-)Equality between memory bytes ($E_G$)
• + Inequality between memory bytes ($I_G$) → More expressivity but more candidates
## Results: Generating Constraints

<table>
<thead>
<tr>
<th></th>
<th>SV-COMP ($E_G$)</th>
<th>SV-COMP ($I_G$)</th>
<th>FISSC ($E_G$)</th>
<th>FISSC ($I_G$)</th>
</tr>
</thead>
<tbody>
<tr>
<td># programs</td>
<td>147</td>
<td>64</td>
<td>719</td>
<td>719</td>
</tr>
<tr>
<td># of robust cases</td>
<td>111</td>
<td>3</td>
<td>129</td>
<td>129</td>
</tr>
<tr>
<td># of sufficient rrc</td>
<td>122</td>
<td>5</td>
<td>359</td>
<td>351</td>
</tr>
<tr>
<td># of weakest rrc</td>
<td>111</td>
<td>3</td>
<td>262</td>
<td>261</td>
</tr>
</tbody>
</table>

### Inference languages

- (dis-)Equality between memory bytes ($E_G$)
- + Inequality between memory bytes ($I_G$) → More expressivity but more candidates
Results: Generating Constraints

<table>
<thead>
<tr>
<th></th>
<th>SV-COMP (E_G)</th>
<th>SV-COMP (I_G)</th>
<th>FISSC (E_G)</th>
<th>FISSC (I_G)</th>
</tr>
</thead>
<tbody>
<tr>
<td># programs</td>
<td>147</td>
<td>64</td>
<td>147</td>
<td>64</td>
</tr>
<tr>
<td># of robust cases</td>
<td>111</td>
<td>3</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td># of sufficient rrc</td>
<td>122</td>
<td>5</td>
<td>127</td>
<td>24</td>
</tr>
<tr>
<td># of weakest rrc</td>
<td>111</td>
<td>3</td>
<td>120</td>
<td>4</td>
</tr>
</tbody>
</table>

Inference languages

- (dis-)Equality between memory bytes \(E_G\)
- + Inequality between memory bytes \(I_G\) \rightarrow More expressivity but more candidates

We can find more reliable bugs than Robust Symbolic Execution
Results: Generating Constraints

<table>
<thead>
<tr>
<th></th>
<th>SV-COMP ((E_G))</th>
<th>SV-COMP ((I_G))</th>
<th>FISSC ((E_G))</th>
<th>FISSC ((I_G))</th>
</tr>
</thead>
<tbody>
<tr>
<td># programs</td>
<td>147</td>
<td>64</td>
<td>147</td>
<td>64</td>
</tr>
<tr>
<td># of robust cases</td>
<td>111</td>
<td>3</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td># of sufficient rrc</td>
<td>122</td>
<td>5</td>
<td>127</td>
<td>24</td>
</tr>
<tr>
<td># of weakest rrc</td>
<td>111</td>
<td>3</td>
<td>120</td>
<td>4</td>
</tr>
</tbody>
</table>

Inference languages

- (dis-)Equality between memory bytes \((E_G)\)
- + Inequality between memory bytes \((I_G)\) \to More expressivity but more candidates

We can find more reliable bugs than Robust Symbolic Execution
Benchmark: FISSC

Fault Injection Attacks

- Physical perturbation of the system executing the program
- Changes the program behavior
- Introduces new bugs
- How does each method characterize these bugs?

VerifyPINs

- 10 protected implementations
- 4800 faulted binary programs
Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th></th>
<th>PyAbd²</th>
<th>Binsec/RSE</th>
<th>Binsec</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>4042</td>
<td>4419</td>
<td>3921</td>
<td>4398</td>
<td>4220</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>719</td>
<td>169</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
</tbody>
</table>
Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th></th>
<th>PYABDP</th>
<th>BINSEC/RSE</th>
<th>BINSEC</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>4042</td>
<td>4419</td>
<td>3921</td>
<td>4398</td>
<td>4220</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>719</td>
<td>169</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>135</td>
</tr>
</tbody>
</table>

Many reported vulnerabilities
### Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th></th>
<th>PyABD$^p$</th>
<th>Binsec/RSE</th>
<th>Binsec</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>4042</td>
<td>4419</td>
<td>3921</td>
<td>4398</td>
<td>4220</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>719</td>
<td>169</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
</tbody>
</table>
Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th></th>
<th>PYABDP</th>
<th>BINSEC/RSE</th>
<th>BINSEC</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>4042</td>
<td>4419</td>
<td>3921</td>
<td>4398</td>
<td>4220</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>719</td>
<td>169</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>-</td>
<td>-</td>
<td>135</td>
</tr>
</tbody>
</table>

Many reported vulnerabilities

No conclusion on more than one input

No details for less than all inputs
Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th>Proportion</th>
<th>PYABDP</th>
<th>BINSEC/RSE</th>
<th>BINSEC</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>118</td>
<td>719</td>
<td>169</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
</tbody>
</table>

Many reported vulnerabilities

No conclusion on more than one input

No details for less than all inputs
Number of faulted programs with at least a given proportion of input states triggering the bug

<table>
<thead>
<tr>
<th></th>
<th>PYABDE</th>
<th>BINSEC/RSE</th>
<th>BINSEC</th>
<th>QEMU</th>
<th>QEMU+L</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>170</td>
<td>273</td>
<td>170</td>
<td>243</td>
<td>284</td>
</tr>
<tr>
<td>not vulnerable (0 input)</td>
<td>4042</td>
<td>4419</td>
<td>3921</td>
<td>4398</td>
<td>4220</td>
</tr>
<tr>
<td>vulnerable (≥ 1 input)</td>
<td>598</td>
<td>118</td>
<td>719</td>
<td>169</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.0001%</td>
<td>598</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>306</td>
</tr>
<tr>
<td>≥ 0.01%</td>
<td>582</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>281</td>
</tr>
<tr>
<td>≥ 0.1%</td>
<td>514</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>210</td>
</tr>
<tr>
<td>≥ 1.0%</td>
<td>472</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>199</td>
</tr>
<tr>
<td>≥ 5.0%</td>
<td>471</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>196</td>
</tr>
<tr>
<td>≥ 10.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>148</td>
</tr>
<tr>
<td>≥ 50.0%</td>
<td>401</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
<tr>
<td>100.0%</td>
<td>399</td>
<td>118</td>
<td>–</td>
<td>–</td>
<td>135</td>
</tr>
</tbody>
</table>

Best characterization

Many reported vulnerabilities

No conclusion on more than one input

No details for less than all inputs
Results: Example of Constraints

- **true**
  Authentication is always possible

- **Card[0] == User[0] && User[0] == 3**
  Authentication when first digit is 3

  Authentication when all digits are equal and non zero

  Authentication when we know the last digit, the 3rd is not correct and the 2nd is 5.

  Authentication with four time the initial value of R0

- **R2 = 0xaa && R1 != 0x55 && R1 != 0x00**
  Authentication if R2=0xaa initially and R1 distinct from both 0x55 and 0x00 initially
Conclusion

• We propose a precondition inference technique to improve the capabilities of Robust Reachability

• We adapt theory-agnostic abduction algorithm to $\exists \forall$ formulas and apply it at program-level through oracles

• We demonstrates its capabilities on simple yet realistic vulnerability characterization scenarii
Conclusion

• We propose a precondition inference technique to improve the capabilities of Robust Reachability

• We adapt theory-agnostic abduction algorithm to $\exists \forall$ formulas and apply it at program-level through oracles

• We demonstrates its capabilities on simple yet realistic vulnerability characterization scenarii

Preconditions explain the vulnerability
Can be reused for understanding, counting, comparing
Conclusion

We propose a precondition inference technique to improve the capabilities of Robust Reachability.

We adapt theory-agnostic abduction algorithm to $\exists \forall$ formulas and apply it at program-level through oracles.

We demonstrates its capabilities on simple yet realistic vulnerability characterization scenarii.

Preconditions explain the vulnerability. Can be reused for understanding, counting, comparing.

Questions?

Also, we have open positions.
Questions