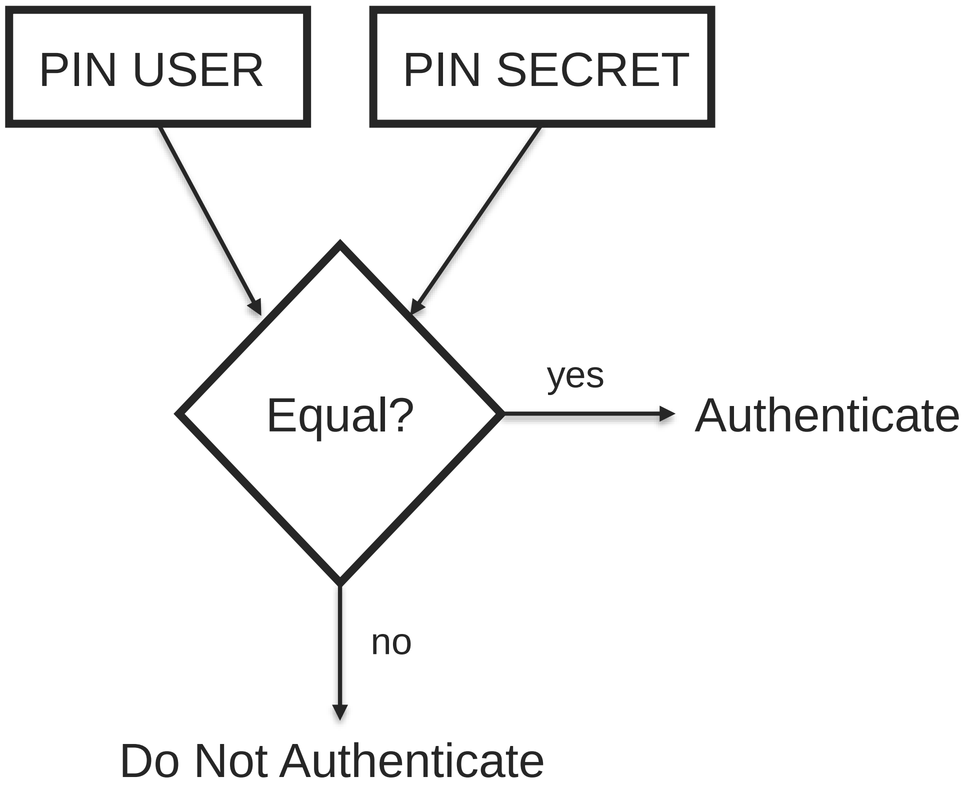
# Inference of Robust Reachability Constraints

**Yanis Sellami**[2,1], Guillaume Girol[2], Frédéric Recoules[2], Damien Couroussé[1], Sébastien Bardin[2]

[1] Univ. Grenoble Alpes, CEA List, France

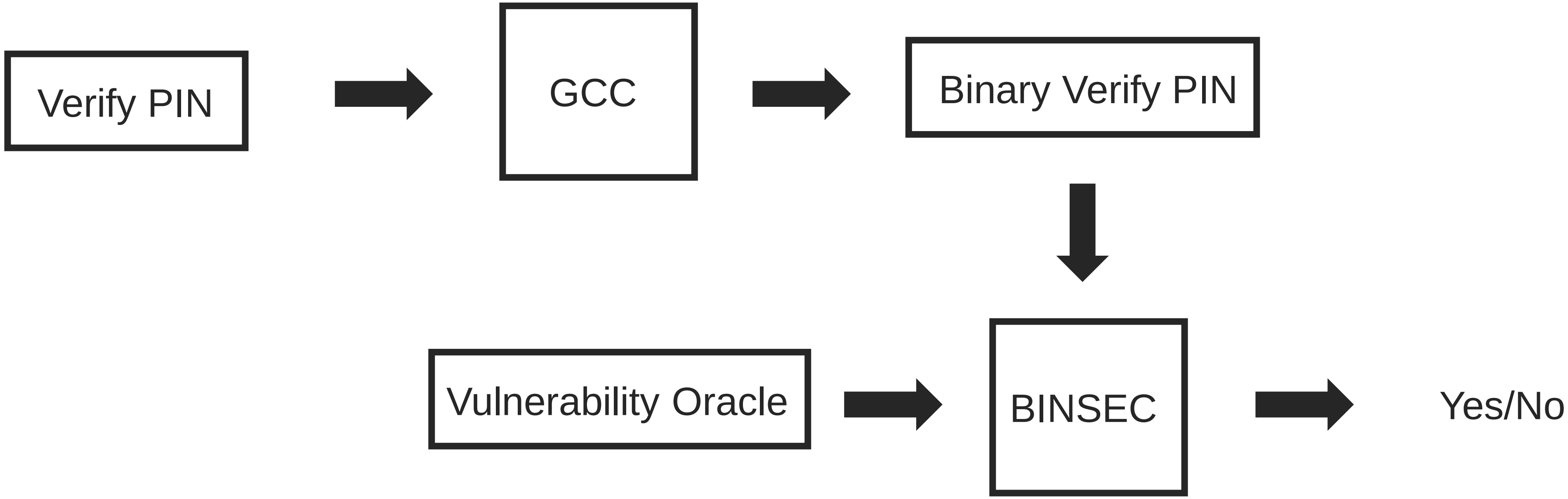
[2] Université Paris-Saclay, CEA List, France

# Example: Verify PIN



- **Compares PINS**
- **Correct when authentication can only happen with correct PIN**
- **Formal Guarantees**

# Example: Verify PIN Formal Verification



Verify PIN

GCC

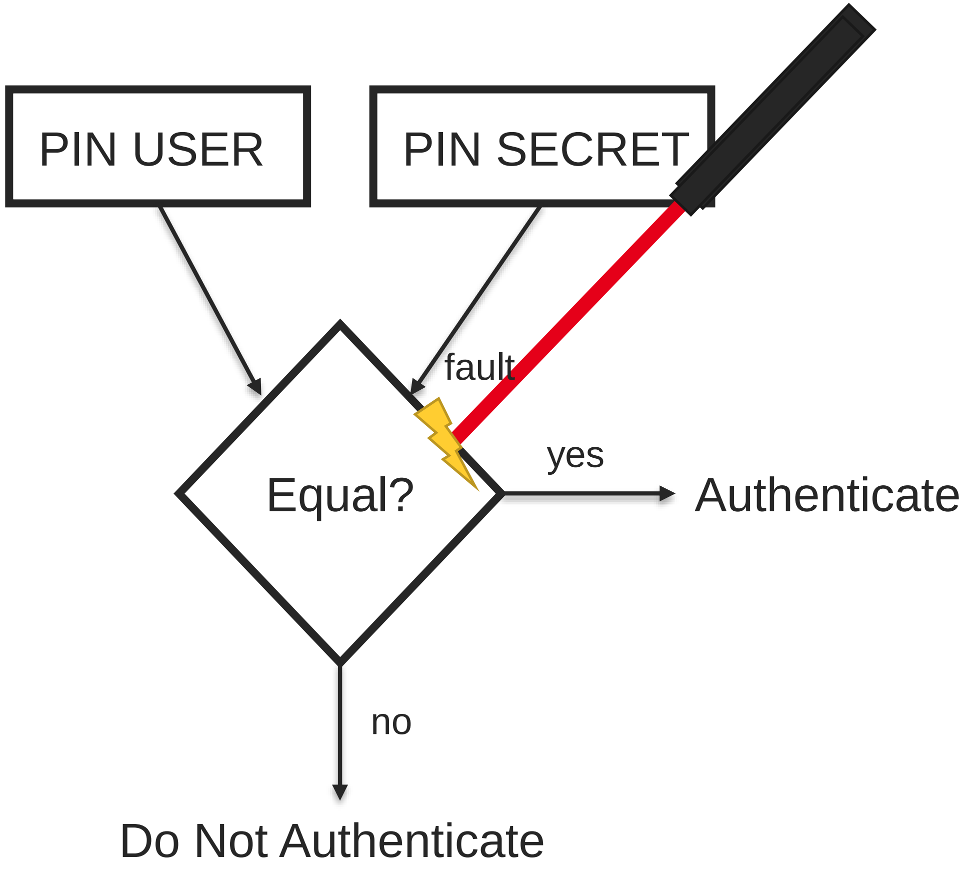
Binary Verify PIN

Vulnerability Oracle

BINSEC

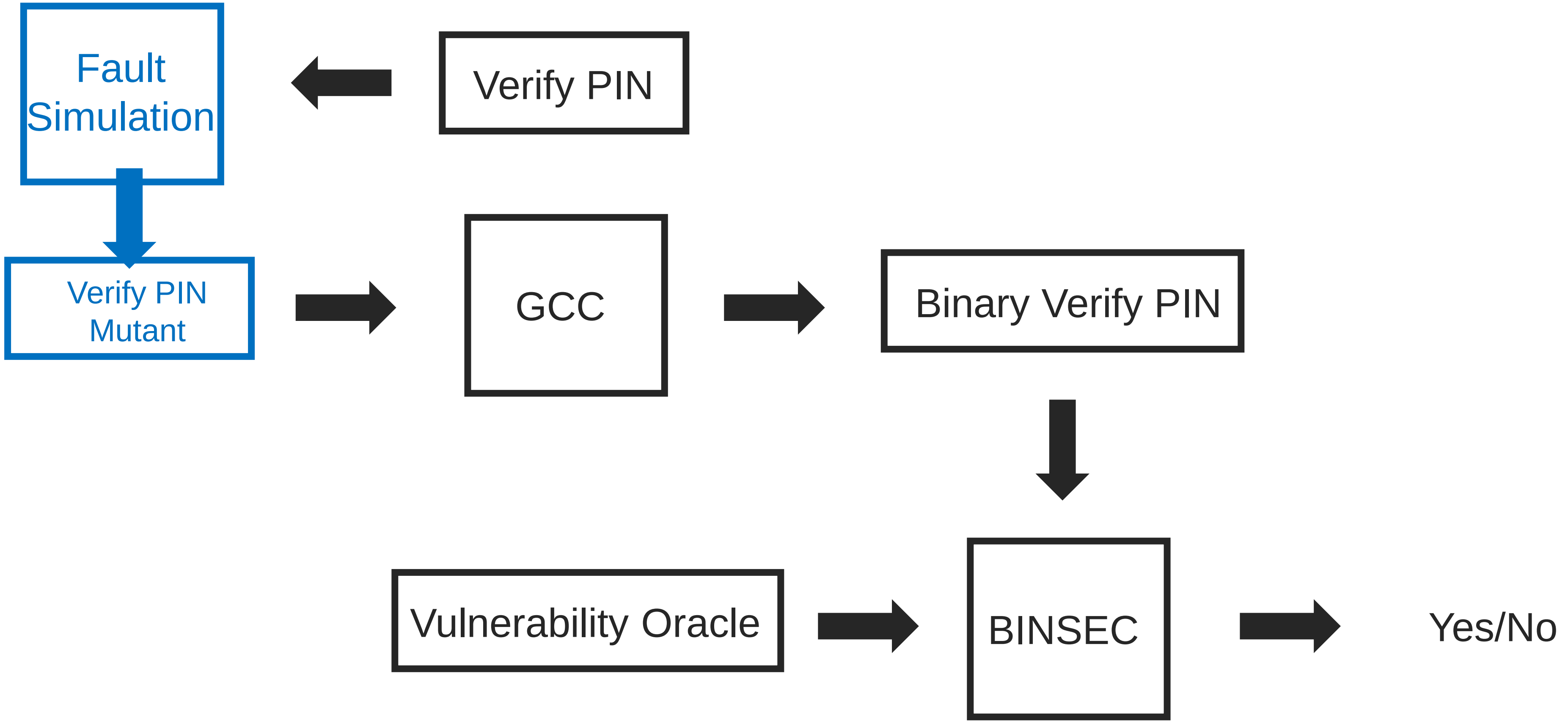
Yes/No

# Example: Verify PIN with Faults



- **Faulted Execution**
- **Alters the behavior of the program**
- **Can we still formally evaluate the feasibility of an unauthorized authentication?**

# Example: Faulted Verify PIN Formal Verification


Fault Simulation

Verify PIN

Verify PIN Mutant

GCC

Binary Verify PIN

Vulnerability Oracle

BINSEC

Yes/No

# Example: Example of Symbolic Execution Result

BINSEC

Yes this VerifyPIN is vulnerable

# Example: Example of Symbolic Execution Result

BINSEC

Yes this VerifyPIN is vulnerable

Because

# Example: Example of Symbolic Execution Result

BINSEC

Yes this VerifyPIN is vulnerable

Because

If R2 contains 0xaa

And

R1 is not 0x55

And

R3 is not 0x00

Then you can authenticate with the wrong PIN

# Example: Example of Symbolic Execution Result

BINSEC

Yes this VerifyPIN is vulnerable

Because

If R2 contains 0xaa

And

R1 is not 0x55

And

R3 is not 0x00

Then you can authenticate with the wrong PIN

**Great!**

**What do I do with this?**

# Formal Characterization of Fault Injection Attacks Vulnerabilities

- Formal evaluation of the faulted program gives no insight on the severity of the problem

- How to design a formal analysis that provides a more expressive result?

- How to characterize the vulnerabilities we discover?

# 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

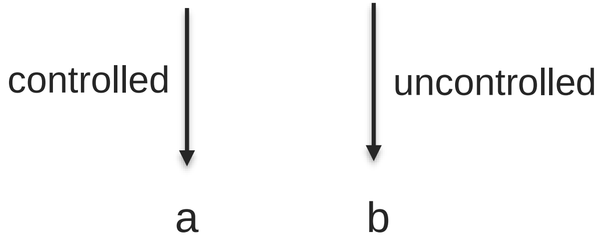
# Robust Reachability

## [Girol, Farinier, Bardin: CAV 2021]

**Idea**

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

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



controlled

uncontrolled

a

b

# Robust Reachability

## [Girol, Farinier, Bardin: CAV 2021]

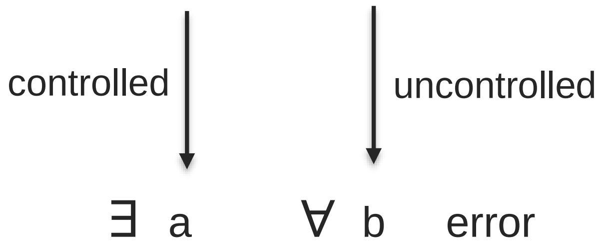
**Idea**

- 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

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



$\exists\ a \quad \forall\ b \quad$ error

# Robust Reachability

## [Girol, Farinier, Bardin: CAV 2021]

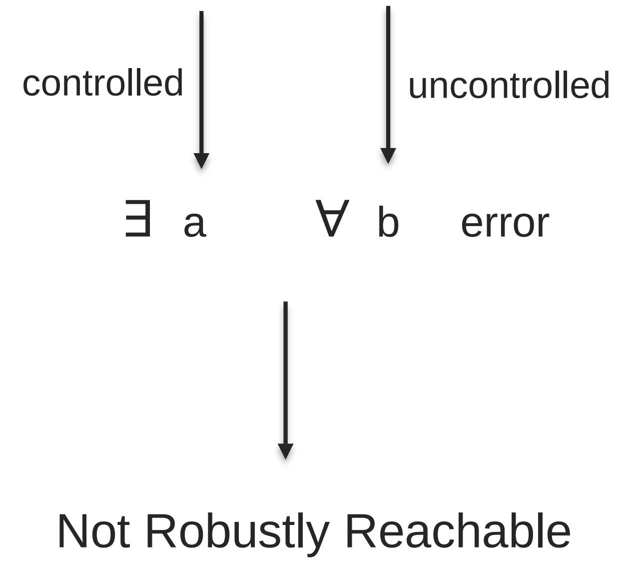
### Idea

- 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

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



controlled

uncontrolled

$\exists$ a $\forall$ b error

Not Robustly Reachable

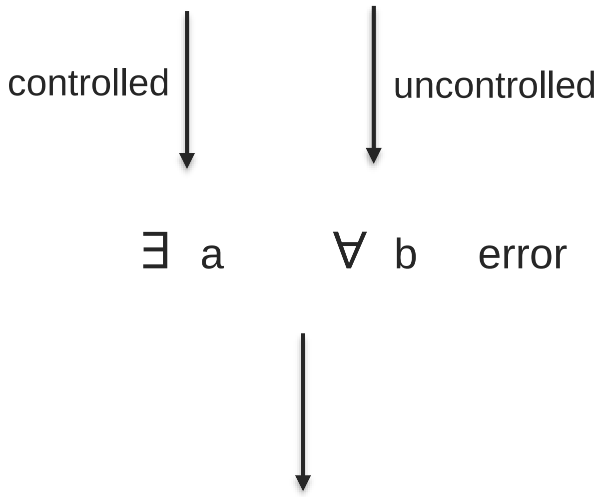
# The Remaining Problem

**Reachability Says: Vulnerable**

**Robust Reachability Says: Not Vulnerable**

**Looks like it can happen**

```
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if ((a + b) % 2 == 0)
        /* bug */
    else
        ...
}
```



controlled

uncontrolled

∃ a ∀ b error

Not Robustly Reachable

# The Remaining Problem

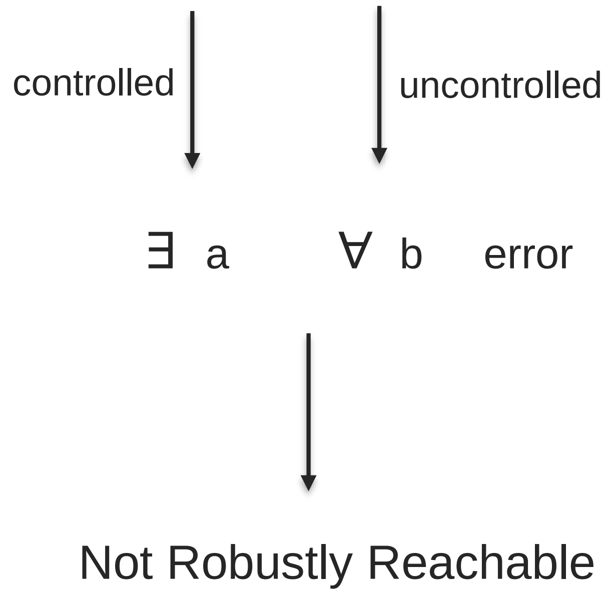
**Reachability Says: Vulnerable**

**Robust Reachability Says: Not Vulnerable**

**Looks like it can happen**

**Robust Reachability is Too Strong**

```
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if ((a + b) % 2 == 0)
        /* bug */
    else
        ...
}
```



controlled

uncontrolled

$\exists$ a $\forall$ b error

Not Robustly Reachable

# Robust Reachability Constraint

**Definition**

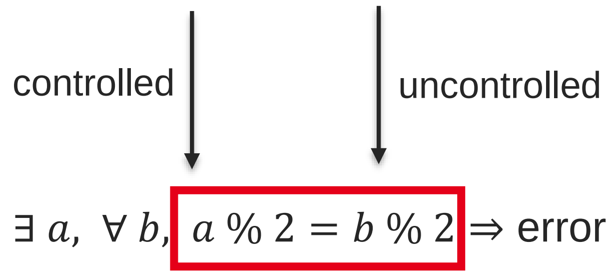
- Predicate on program input sufficient to have Robust Reachability

**Advantages**

- Part of the Robust Reachability framework
- Allows precise characterization

**How to Automatically Generate Such Constraints?**

```
void g() {
    uint a = read();
    uint b; /* uninitialized */
    if ((a + b) % 2 == 0)
        /* bug */
    else
        ...
}
```



$$\exists\, a,\ \forall\, b,\ \boxed{a \,\%\, 2 = b \,\%\, 2} \Rightarrow \text{error}$$

# Abduction of Robust Reachability Constraints

**Abductive Reasoning**

[Josephson and Josephson, 1994]

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

# Abduction of Robust Reachability Constraints

### Abductive Reasoning

[Josephson and Josephson, 1994]

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

### Theory-Specific Abduction

[Bienvenu 2007, Tourret et. al. 2017]

- Handle a single theory

### Specification Synthesis

[Albarghouthi et. al. 2016, Calcagno et. al. 2009,
Zhou et. al. 2021]

- 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 $\phi_H \land \phi_M \vDash \phi_G$

### Theory-Specific Abduction

[Bienvenu 2007, Tourret et. al. 2017]

- Handle a single theory

### Specification Synthesis

[Albarghouthi et. al. 2016, Calcagno et. al. 2009,
Zhou et. al. 2021]

- 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 $\phi_H \land \phi_M \vDash \phi_G$

### Theory-Specific Abduction

[Bienvenu 2007, Tourret et. al. 2017]

- Handle a single theory

### Specification Synthesis

[Albarghouthi et. al. 2016, Calcagno et. al. 2009, Zhou et. al. 2021]

- White-box program analysis

### Theory-Agnostic First-order Abduction

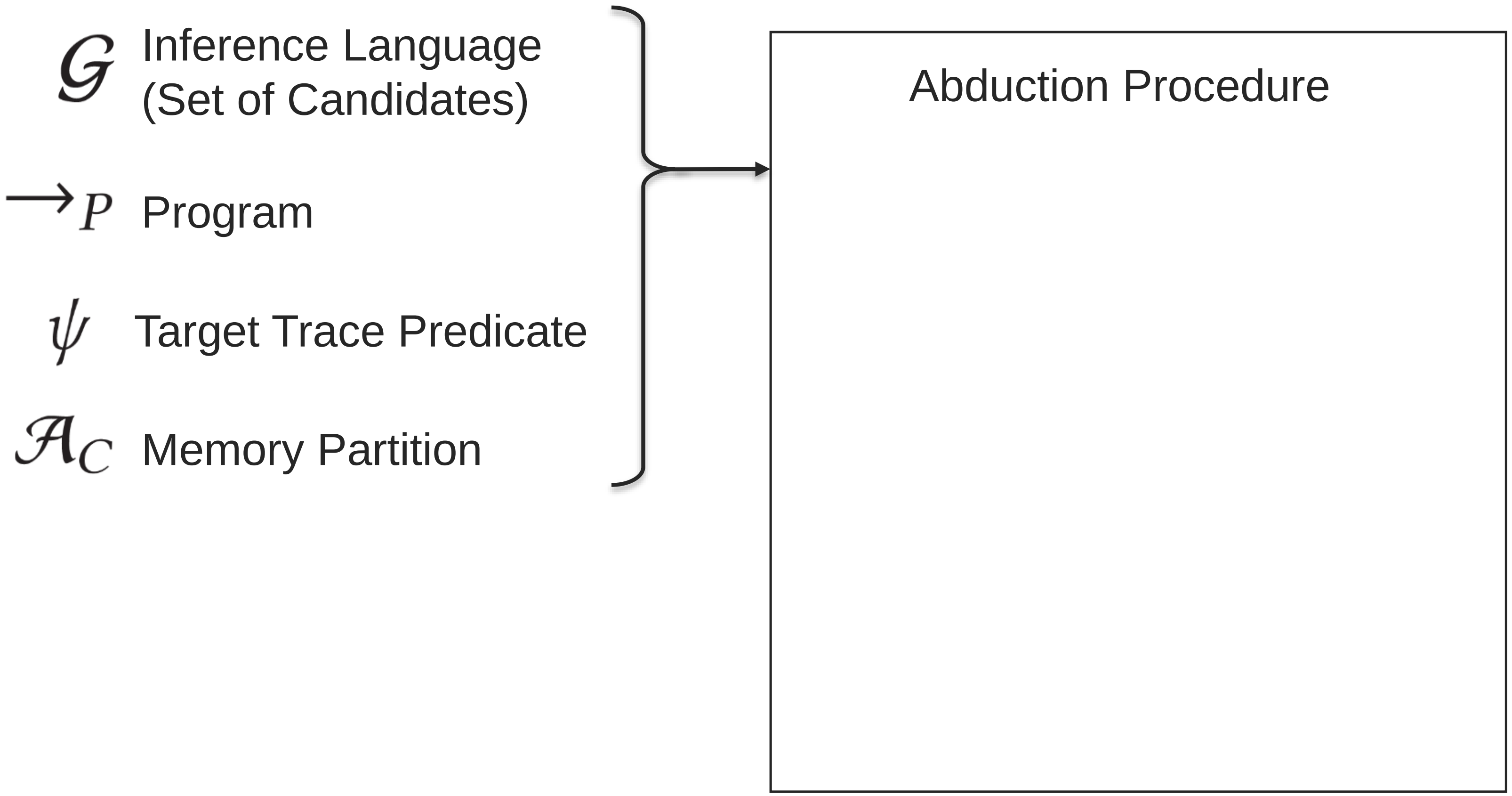
[Echenim et al. 2018, Reynolds et al. 2020]

- Efficient procedures
- Genericity

### Our Proposal: Adapt Theory-Agnostic Abduction Algorithm to Compute Program-level Robust Reachability Constraints

- Program-level
- Generic

# Our Solution (Framework)



$\mathcal{G}$ Inference Language (Set of Candidates)

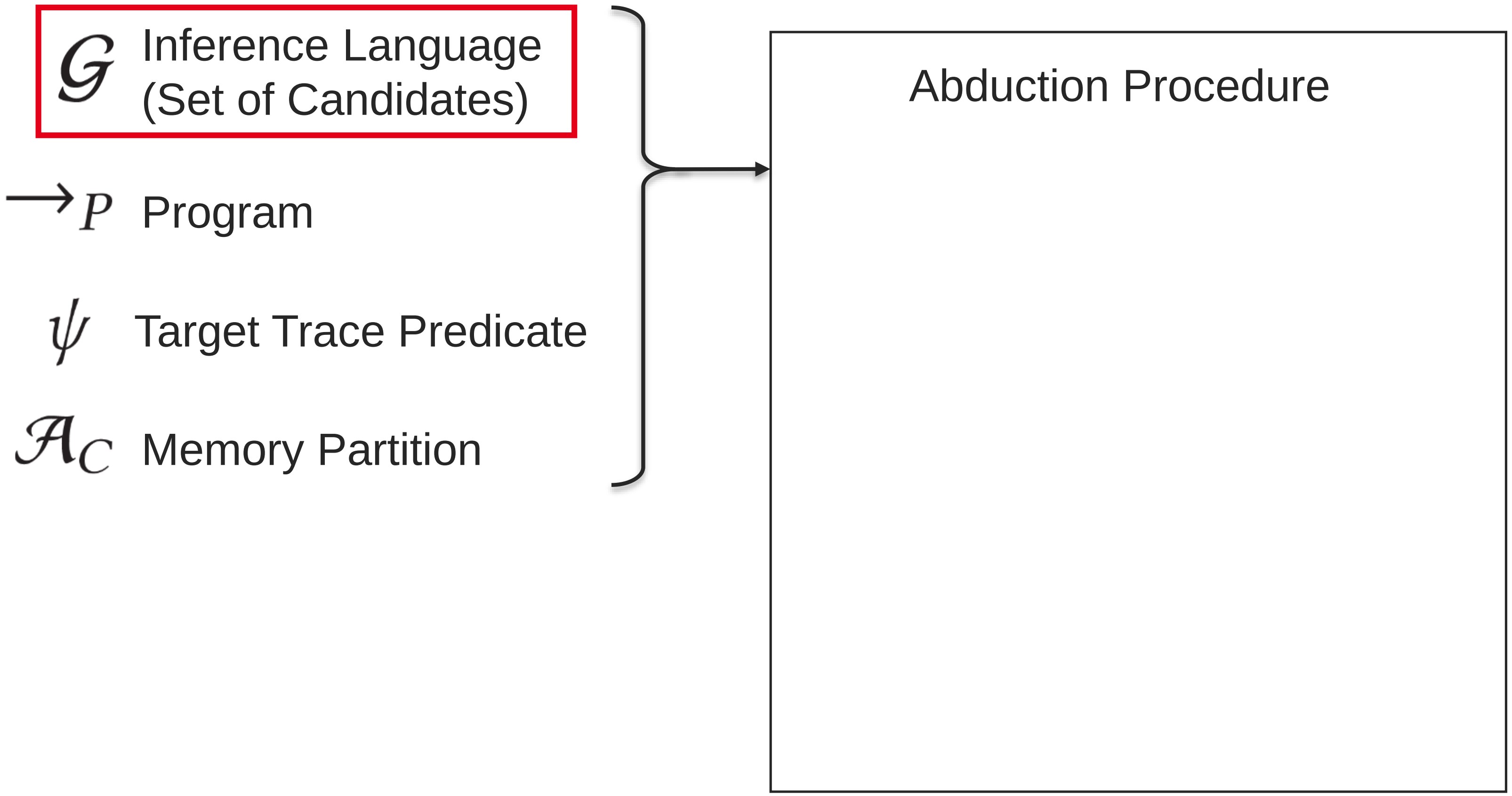
$\longrightarrow P$ Program

$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

# Our Solution (Framework)



$\mathcal{G}$ Inference Language (Set of Candidates)

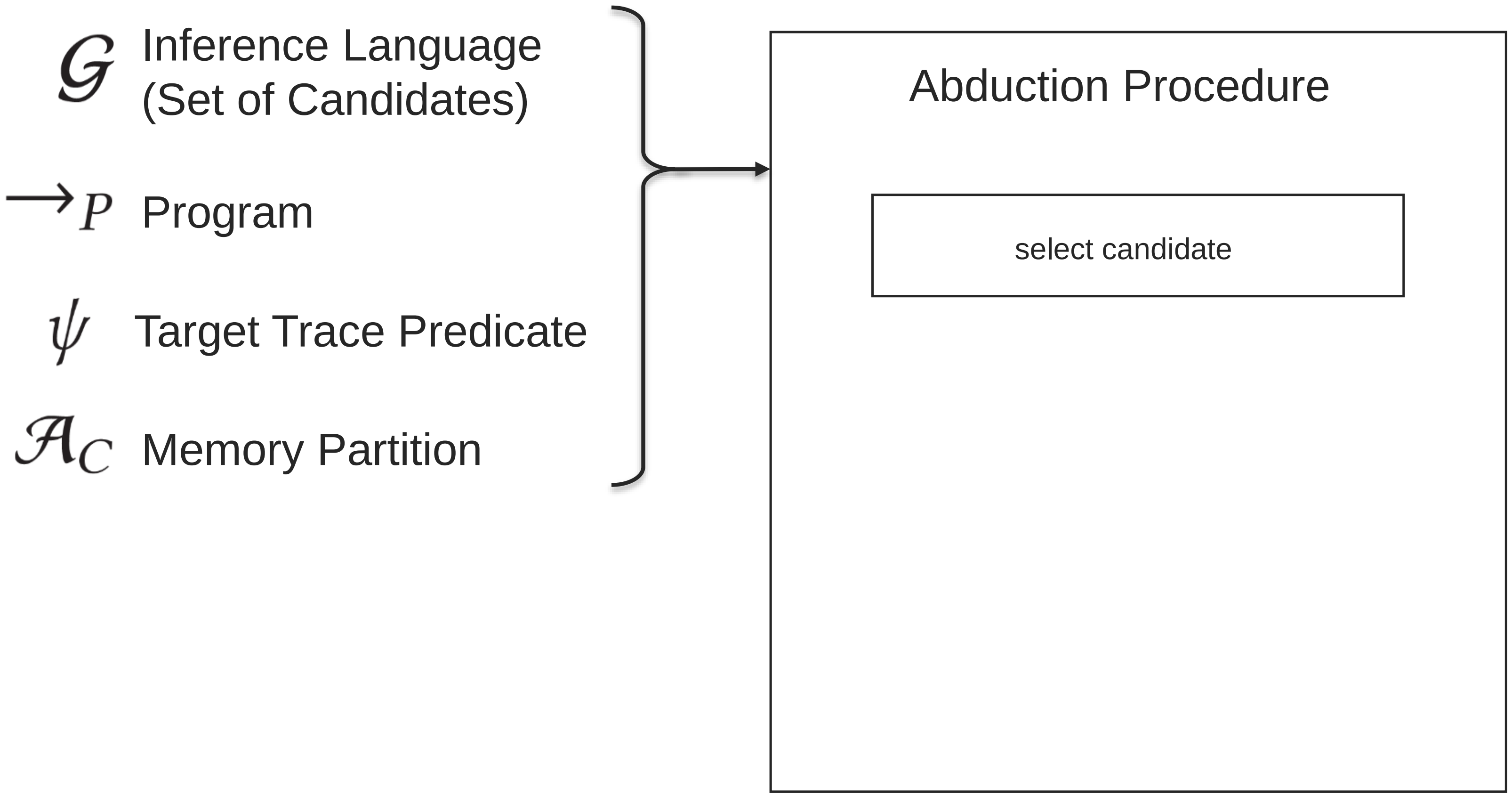
$\overrightarrow{P}$ Program

$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

# Our Solution (Framework)



$\mathcal{G}$ Inference Language (Set of Candidates)

$\overrightarrow{P}$ Program

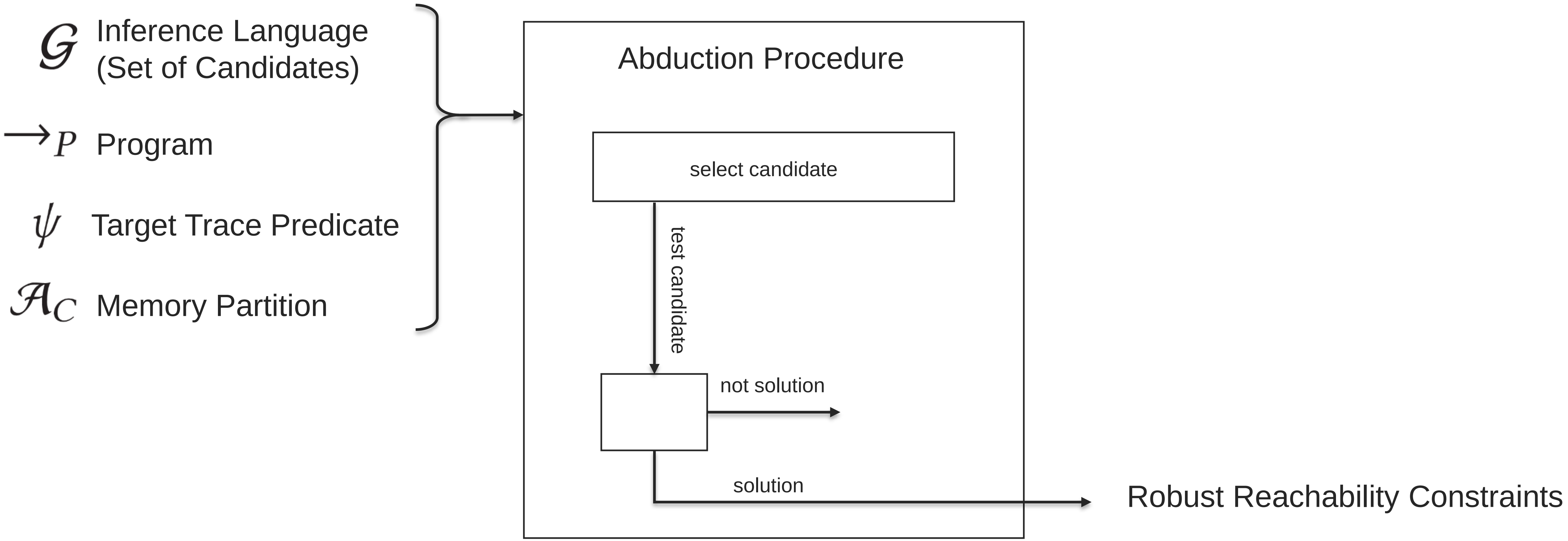
$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

select candidate

# Our Solution (Framework)



$\mathcal{G}$ Inference Language (Set of Candidates)

$\longrightarrow_P$ Program

$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

select candidate

test candidate

not solution

solution

Robust Reachability Constraints

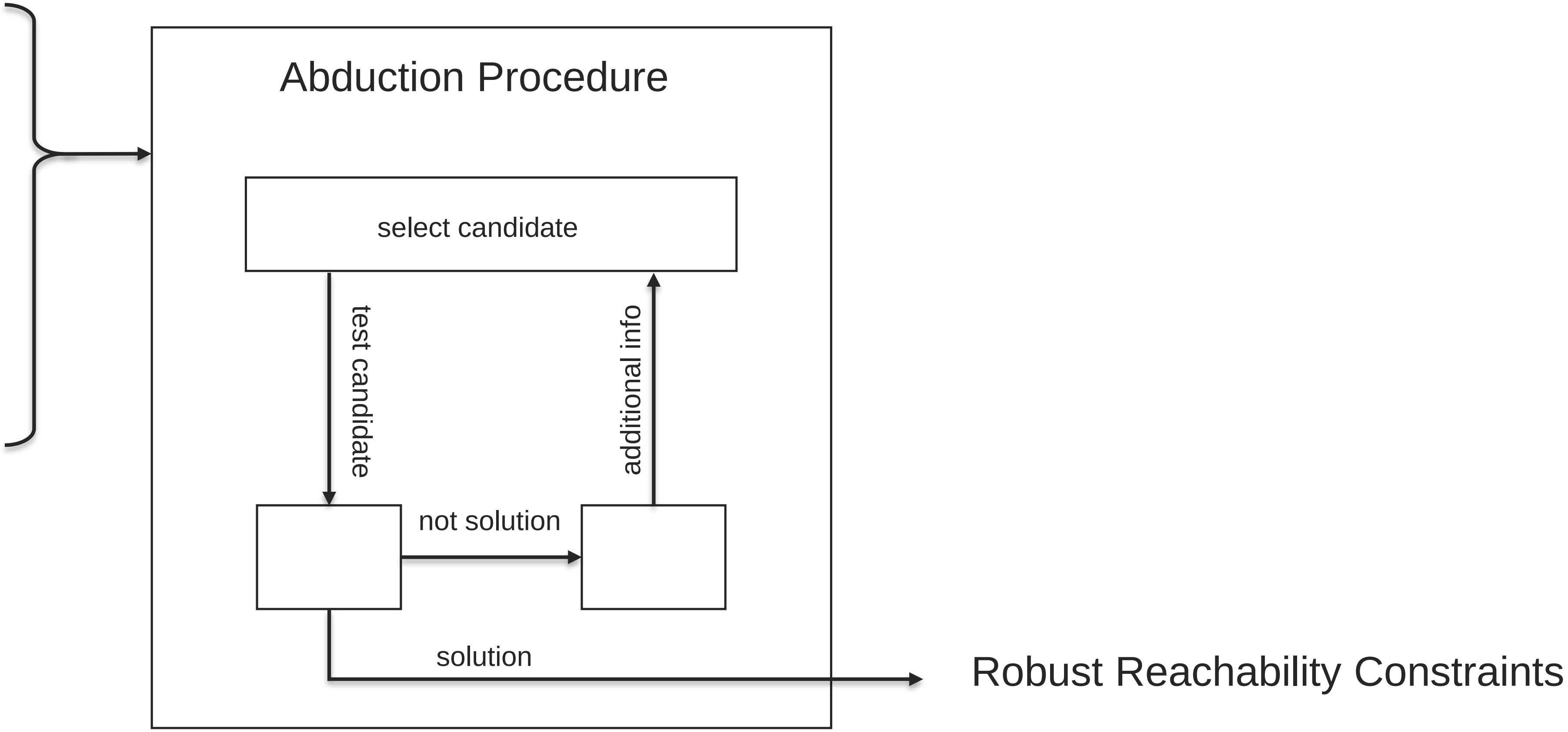
# Our Solution (Framework)

$\mathcal{G}$ Inference Language (Set of Candidates)

$\longrightarrow_P$ Program

$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



Abduction Procedure

select candidate

test candidate

additional info

not solution

solution

Robust Reachability Constraints

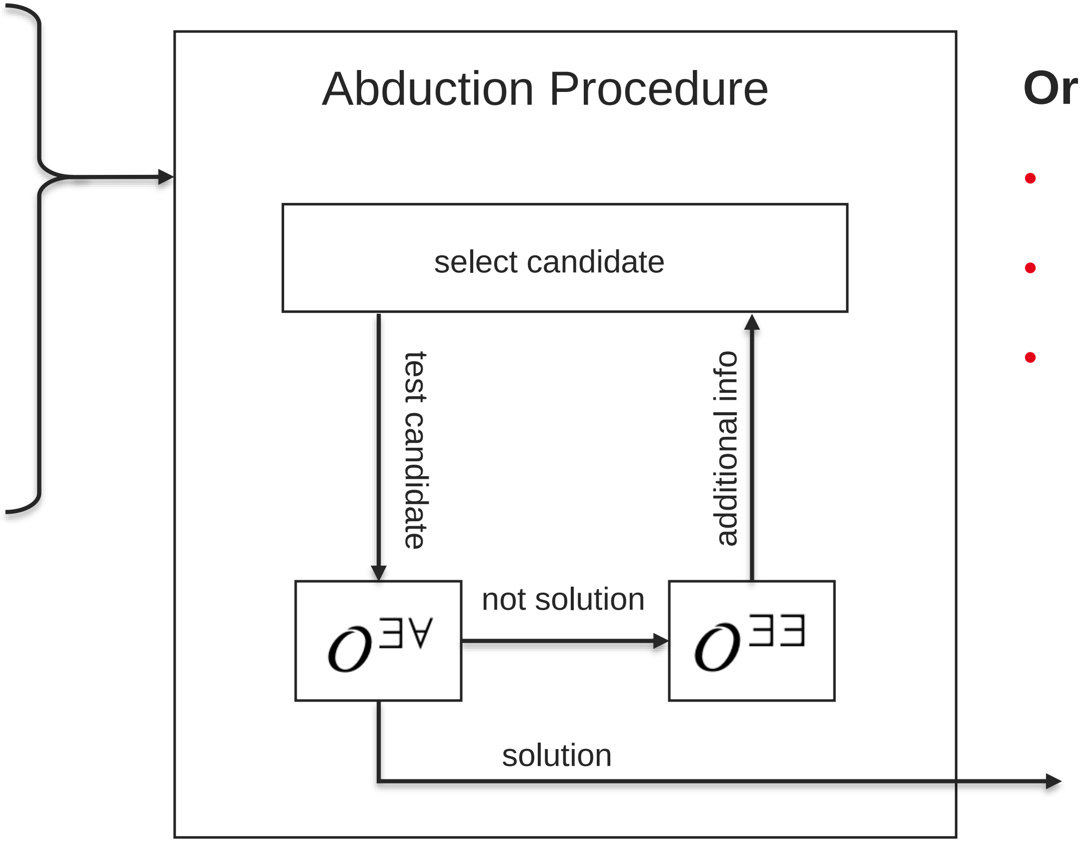
# Our Solution (Framework)

$\mathcal{G}$ Inference Language (Set of Candidates)

$\longrightarrow P$ Program

$\psi$ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



**Oracles on Trace Properties**

- Robust property queries $\mathcal{O}^{\exists\forall}$
- Non-robust property queries $\mathcal{O}^{\exists\exists}$
- Can accomodate various tools (SE, BMC, Incorrectness, …)

Robust Reachability Constraints

# Our Solution (Baseline Algorithm)

```
BaselineRCInfer($\mathcal{G}, \rightarrow_P, \psi, \mathcal{A}_C$)
if $\top, s \leftarrow \mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \top)$ then
    $R \longleftarrow \{y = s\}$ if $y = s \in \mathcal{G}$ else $\emptyset$;
    for $\phi \in \mathcal{G}$ do
        if $\mathcal{O}^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ then
            $R \longleftarrow \Delta_{min}(R \cup \{\phi\})$;
            if $\neg\mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi' \in R} \phi'))$ then
                return $R$;
    return $R$;
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)

```
BaselineRCInfer($\mathcal{G}, \rightarrow_P, \psi, \mathcal{A}_C$)
if $\top, s \leftarrow \mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \top)$ then
    $R \longleftarrow \{y = s\}$ if $y = s \in \mathcal{G}$ else $\emptyset$;
    for $\phi \in \mathcal{G}$ do
        if $\mathcal{O}^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ then
            $R \longleftarrow \Delta_{min}(R \cup \{\phi\})$;
            if $\neg\mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi' \in R} \phi'))$ then
                return $R$;
    return $R$;
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

- Under correction and completeness of the oracles
  - **Minimality** w.r.t. the inference language
  - **Weakest** constraint generation when expressible

# Making it Work

### The Issue

- Exhaustive exploration of the inference language is inefficient

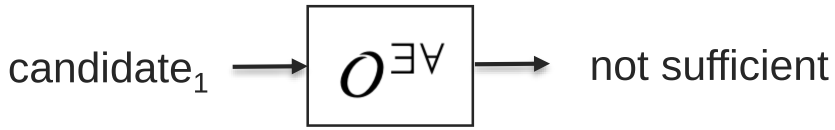
### Key Strategies for Efficient Exploration

- Necessary constraints
- Counter-examples for Robust Reachability
- Ordering candidates

# Making it Work: Necessary Constraints

**The Idea**

- Find and store Necessary Constraints

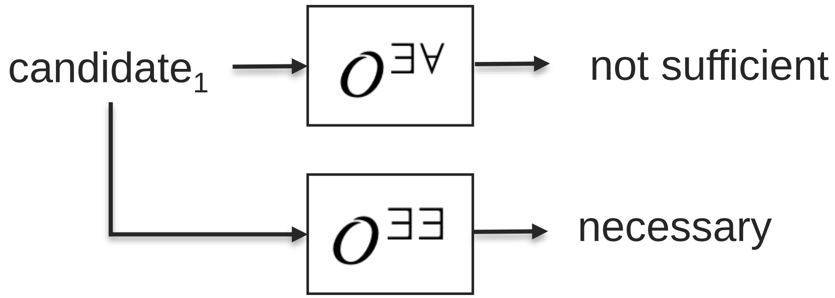


candidate$_1$ → $\mathcal{O}^{\exists\forall}$ → not sufficient

# Making it Work: Necessary Constraints

**The Idea**

- Find and store Necessary Constraints



candidate$_1$

$\mathcal{O}^{\exists\forall}$ → not sufficient

$\mathcal{O}^{\exists\exists}$ → necessary

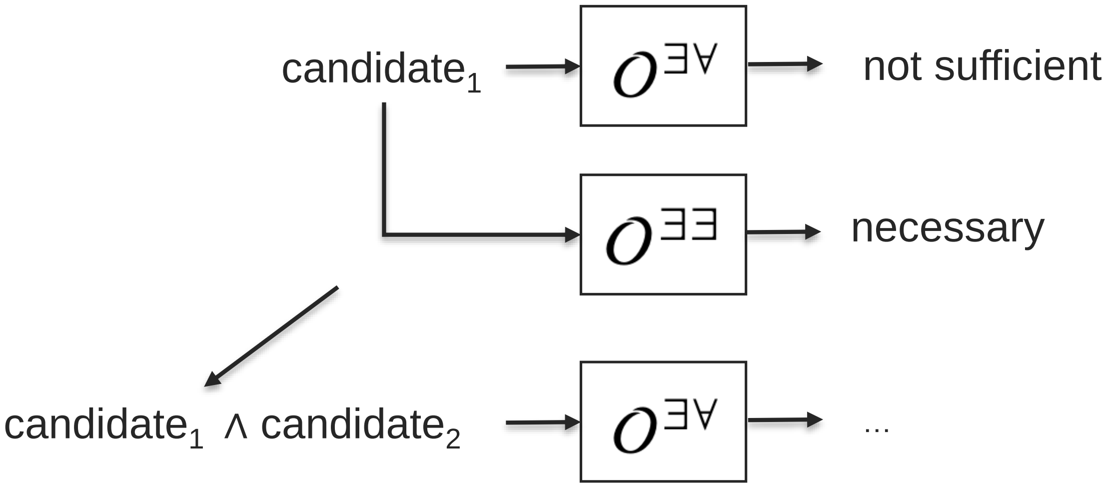
# Making it Work: Necessary Constraints

**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



candidate$_1$ → $\mathcal{O}^{\exists\forall}$ → not sufficient

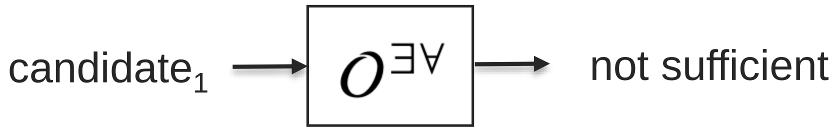
→ $\mathcal{O}^{\exists\exists}$ → necessary

candidate$_1$ ∧ candidate$_2$ → $\mathcal{O}^{\exists\forall}$ → …

# Making it Work: Counter-Examples

**The Idea**

- Reuse information from failed candidate checks



candidate$_1$ → $\mathcal{O}^{\exists\forall}$ → not sufficient

**The Issue**

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

# Making it Work: Counter-Examples

## The Idea

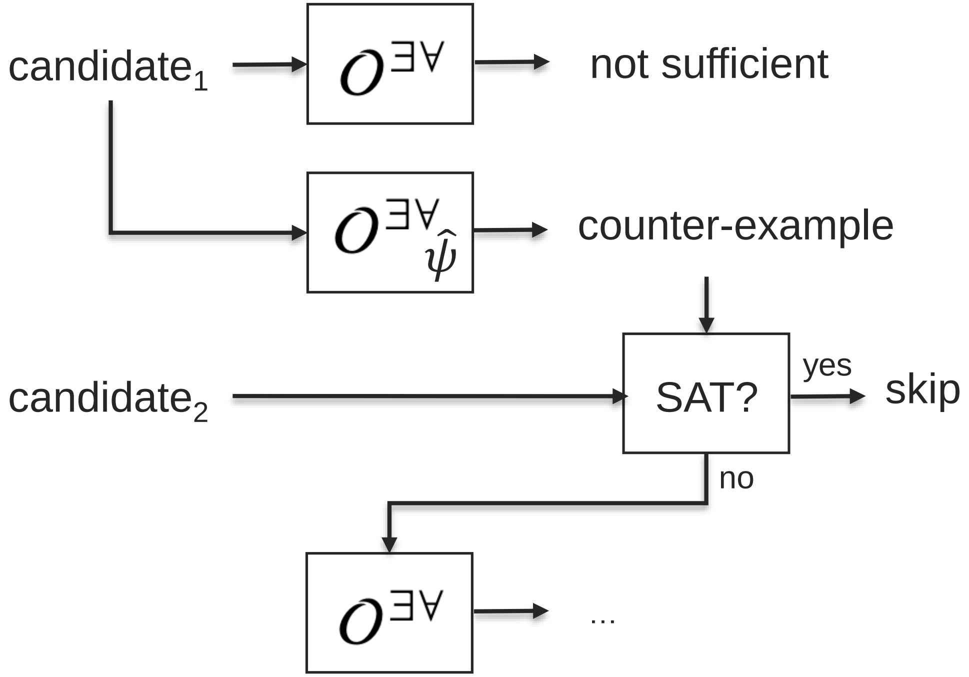
- Reuse information from failed candidate checks

## 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



candidate$_1$

$\mathcal{O}^{\exists\forall}$

not sufficient

$\mathcal{O}^{\exists\forall}_{\hat{\psi}}$

counter-example

candidate$_2$

SAT?

yes

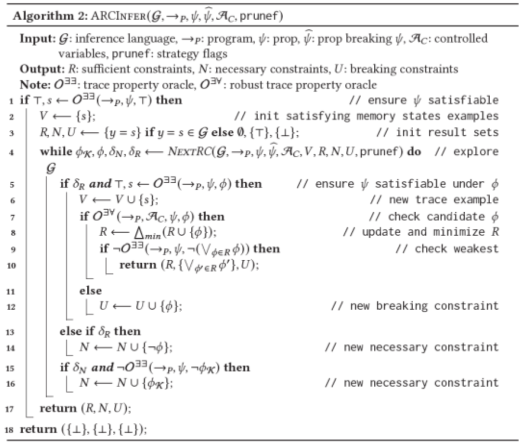
skip

no

$\mathcal{O}^{\exists\forall}$

...

# Final Algorithm



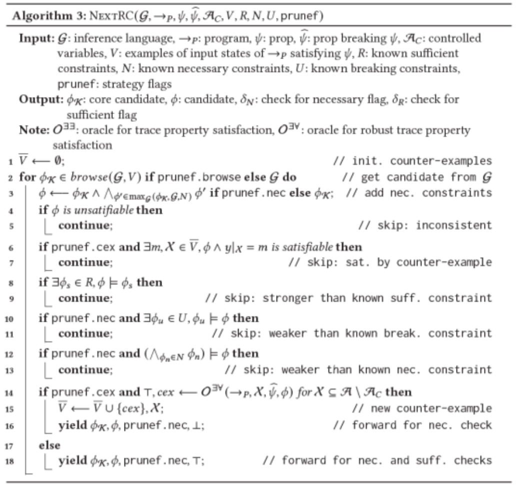
**Algorithm 2:** ARCINFER($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C$, prunef)

**Input:** $\mathcal{G}$: inference language, $\rightarrow_P$: program, $\psi$: prop, $\widehat{\psi}$: prop breaking $\psi$, $\mathcal{A}_C$: controlled variables, prunef: strategy flags

**Output:** $R$: sufficient constraints, $N$: necessary constraints, $U$: breaking constraints

**Note:** $O^{\exists\exists}$: trace property oracle, $O^{\exists\forall}$: robust trace property oracle

```
if $\top, s \leftarrow O^{\exists\exists}(\rightarrow_P, \psi, \top)$ then                          // ensure $\psi$ satisfiable
    $V \longleftarrow \{s\}$;                                   // init satisfying memory states examples
    $R, N, U \longleftarrow \{y = s\}$ if $y = s \in \mathcal{G}$ else $\emptyset, \{\top\}, \{\bot\}$;          // init result sets
    while $\phi_{\mathcal{K}}, \phi, \delta_N, \delta_R \longleftarrow$ NextRC($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C, V, R, N, U$, prunef) do   // explore $\mathcal{G}$
        if $\delta_R$ and $\top, s \leftarrow O^{\exists\exists}(\rightarrow_P, \psi, \phi)$ then     // ensure $\psi$ satisfiable under $\phi$
            $V \longleftarrow V \cup \{s\}$;                                         // new trace example
            if $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ then                        // check candidate $\phi$
                $R \longleftarrow \Delta_{min}(R \cup \{\phi\})$;                          // update and minimize $R$
                if $\neg O^{\exists\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi \in R} \phi))$ then                 // check weakest
                    return $(R, \{\bigvee_{\phi' \in R} \phi'\}, U)$;
            else
                $U \longleftarrow U \cup \{\phi\}$;                             // new breaking constraint
        else if $\delta_R$ then
            $N \longleftarrow N \cup \{\neg\phi\}$;                            // new necessary constraint
        if $\delta_N$ and $\neg O^{\exists\exists}(\rightarrow_P, \psi, \neg\phi_{\mathcal{K}})$ then
            $N \longleftarrow N \cup \{\phi_{\mathcal{K}}\}$;                           // new necessary constraint
    return $(R, N, U)$;
return $(\{\bot\}, \{\bot\}, \{\bot\})$;
```



**Algorithm 3:** NextRC($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C, V, R, N, U$, prunef)

**Input:** $\mathcal{G}$: inference language, $\rightarrow_P$: program, $\psi$: prop, $\widehat{\psi}$: prop breaking $\psi$, $\mathcal{A}_C$: controlled variables, $V$: examples of input states of $\rightarrow_P$ satisfying $\psi$, $R$: known sufficient constraints, $N$: known necessary constraints, $U$: known breaking constraints, prunef: strategy flags

**Output:** $\phi_{\mathcal{K}}$: core candidate, $\phi$: candidate, $\delta_N$: check for necessary flag, $\delta_R$: check for sufficient flag

**Note:** $O^{\exists\exists}$: oracle for trace property satisfaction, $O^{\exists\forall}$: oracle for robust trace property satisfaction

```
$\overline{V} \longleftarrow \emptyset$;                                              // init. counter-examples
for $\phi_{\mathcal{K}} \in browse(\mathcal{G}, V)$ if prunef.browse else $\mathcal{G}$ do           // get candidate from $\mathcal{G}$
    $\phi \longleftarrow \phi_{\mathcal{K}} \wedge \bigwedge_{\phi' \in \max_{\mathcal{G}}(\phi_{\mathcal{K}}, \mathcal{G}, N)} \phi'$ if prunef.nec else $\phi_{\mathcal{K}}$;   // add nec. constraints
    if $\phi$ is unsatifiable then
        continue;                                                // skip: inconsistent
    if prunef.cex and $\exists m, \mathcal{X} \in \overline{V}, \phi \wedge y|_{\mathcal{X}} = m$ is satisfiable then
        continue;                                    // skip: sat. by counter-example
    if $\exists \phi_s \in R, \phi \models \phi_s$ then
        continue;                      // skip: stronger than known suff. constraint
    if prunef.nec and $\exists \phi_u \in U, \phi_u \models \phi$ then
        continue;                       // skip: weaker than known break. constraint
    if prunef.nec and $(\bigwedge_{\phi_n \in N} \phi_n) \models \phi$ then
        continue;                         // skip: weaker than known nec. constraint
    if prunef.cex and $\top, cex \longleftarrow O^{\exists\forall}(\rightarrow_P, \mathcal{X}, \widehat{\psi}, \phi)$ for $\mathcal{X} \subseteq \mathcal{A} \setminus \mathcal{A}_C$ then
        $\overline{V} \longleftarrow \overline{V} \cup \{cex\}, \mathcal{X}$;                               // new counter-example
        yield $\phi_{\mathcal{K}}, \phi$, prunef.nec, $\bot$;                        // forward for nec. check
    else
        yield $\phi_{\mathcal{K}}, \phi$, prunef.nec, $\top$;             // forward for nec. and suff. checks
```

## Theorem

- **Termination**, **Correction**, **Completeness** are preserved
- **Correction for necessary constraints** at any step
- **Minimality** is preserved modulo equivalence between formulas
- **Weakest constraints generation** on given return is preserved

## Remarks

- Generic procedure definition with oracle queries abstraction
- The previously described strategies can be activated/deactivated
- Can be applied to a larger range of program properties (reachability, safety, hypersafety)
- If SMT-Solvers are used as oracles, can be used an ∃∀ abduction solver

# Experimental Evaluation

### Implementation

BINSEC

- (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

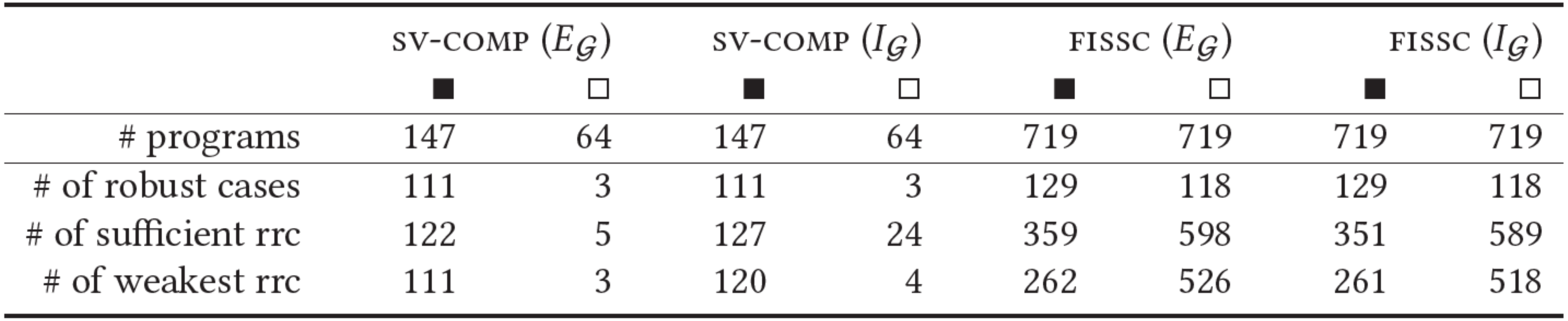
### Research Questions

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



| | SV-COMP ($E_{\mathcal{G}}$) | | SV-COMP ($I_{\mathcal{G}}$) | | FISSC ($E_{\mathcal{G}}$) | | FISSC ($I_{\mathcal{G}}$) | |
|---|---|---|---|---|---|---|---|---|
| | ■ | □ | ■ | □ | ■ | □ | ■ | □ |
| # programs | 147 | 64 | 147 | 64 | 719 | 719 | 719 | 719 |
| # of robust cases | 111 | 3 | 111 | 3 | 129 | 118 | 129 | 118 |
| # of sufficient rrc | 122 | 5 | 127 | 24 | 359 | 598 | 351 | 589 |
| # of weakest rrc | 111 | 3 | 120 | 4 | 262 | 526 | 261 | 518 |

**Inference languages**

- (dis-)Equality between memory bytes $(E_{\mathcal{G}})$
- + Inequality between memory bytes $(I_{\mathcal{G}})$ → More expressivity but more candidates

# Results: Generating Constraints

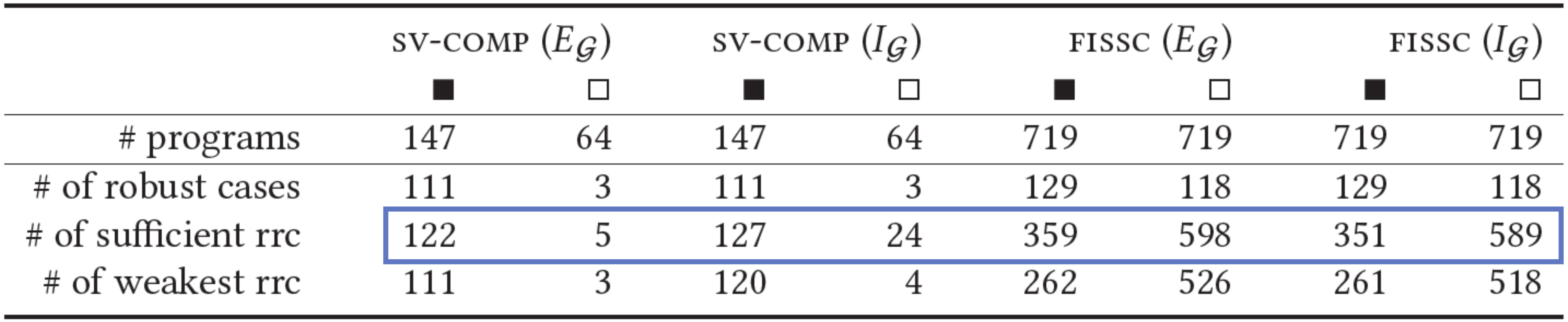
| | SV-COMP ($E_{\mathcal{G}}$) | | SV-COMP ($I_{\mathcal{G}}$) | | FISSC ($E_{\mathcal{G}}$) | | FISSC ($I_{\mathcal{G}}$) | |
|---|---|---|---|---|---|---|---|---|
| | ■ | □ | ■ | □ | ■ | □ | ■ | □ |
| # programs | 147 | 64 | 147 | 64 | 719 | 719 | 719 | 719 |
| # of robust cases | 111 | 3 | 111 | 3 | 129 | 118 | 129 | 118 |
| # of sufficient rrc | 122 | 5 | 127 | 24 | 359 | 598 | 351 | 589 |
| # of weakest rrc | 111 | 3 | 120 | 4 | 262 | 526 | 261 | 518 |

previous
characterization

**Inference languages**

- (dis-)Equality between memory bytes $(E_{\mathcal{G}})$
- + Inequality between memory bytes $(I_{\mathcal{G}})$ → More expressivity but more candidates

# Results: Generating Constraints



| | SV-COMP $(E_{\mathcal{G}})$ | | SV-COMP $(I_{\mathcal{G}})$ | | FISSC $(E_{\mathcal{G}})$ | | FISSC $(I_{\mathcal{G}})$ | |
|---|---|---|---|---|---|---|---|---|
| | ■ | □ | ■ | □ | ■ | □ | ■ | □ |
| # programs | 147 | 64 | 147 | 64 | 719 | 719 | 719 | 719 |
| # of robust cases | 111 | 3 | 111 | 3 | 129 | 118 | 129 | 118 |
| # of sufficient rrc | 122 | 5 | 127 | 24 | 359 | 598 | 351 | 589 |
| # of weakest rrc | 111 | 3 | 120 | 4 | 262 | 526 | 261 | 518 |

previous characterization

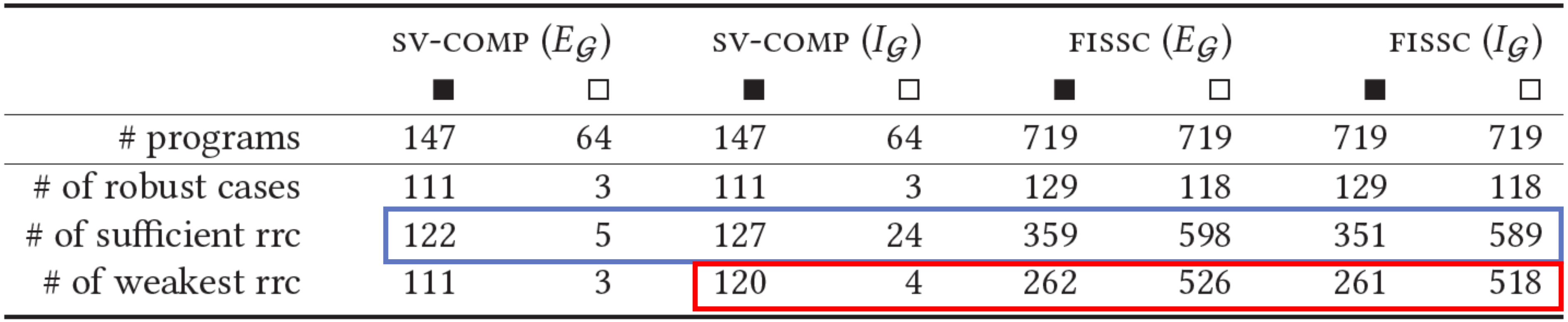
our characterization

**Inference languages**

- (dis-)Equality between memory bytes $(E_{\mathcal{G}})$
- \+ Inequality between memory bytes $(I_{\mathcal{G}})$ → More expressivity but more candidates

**We can find more reliable bugs than Robust Symbolic Execution**

# Results: Generating Constraints



| | SV-COMP $(E_{\mathcal{G}})$ | | SV-COMP $(I_{\mathcal{G}})$ | | FISSC $(E_{\mathcal{G}})$ | | FISSC $(I_{\mathcal{G}})$ | |
|---|---|---|---|---|---|---|---|---|
| | ■ | □ | ■ | □ | ■ | □ | ■ | □ |
| # programs | 147 | 64 | 147 | 64 | 719 | 719 | 719 | 719 |
| # of robust cases | 111 | 3 | 111 | 3 | 129 | 118 | 129 | 118 |
| # of sufficient rrc | 122 | 5 | 127 | 24 | 359 | 598 | 351 | 589 |
| # of weakest rrc | 111 | 3 | 120 | 4 | 262 | 526 | 261 | 518 |

previous characterization

our characterization

**Inference languages**

- (dis-)Equality between memory bytes $(E_{\mathcal{G}})$
- \+ Inequality between memory bytes $(I_{\mathcal{G}})$ → More expressivity but more candidates

**We can find more reliable bugs than Robust Symbolic Execution**

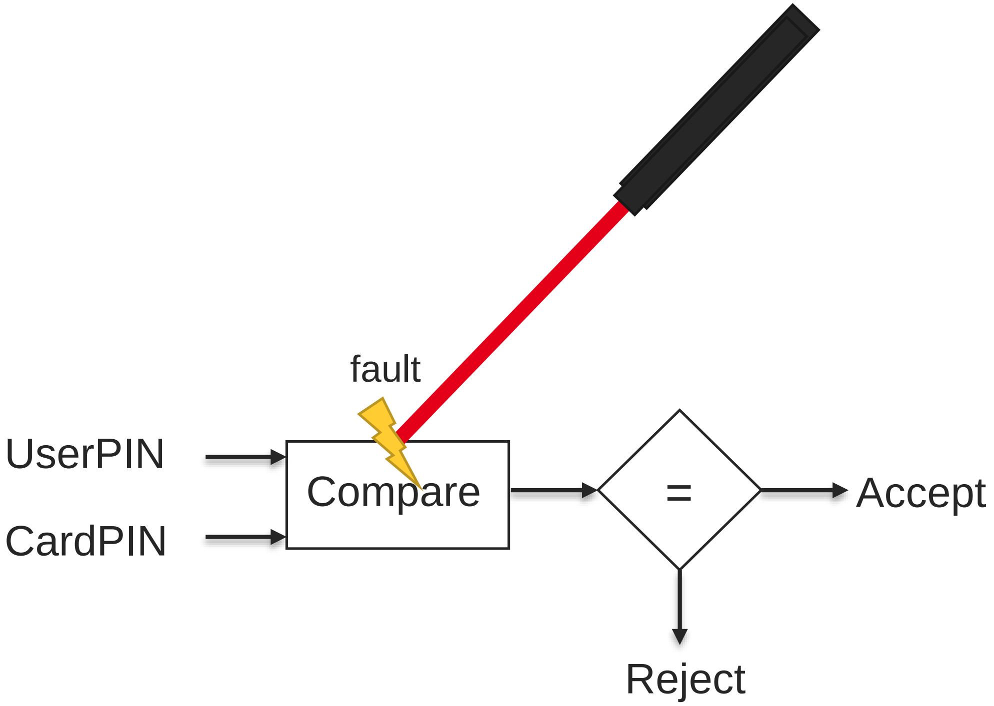
# Benchmark: FISSC

**Fault Injection Attacks**

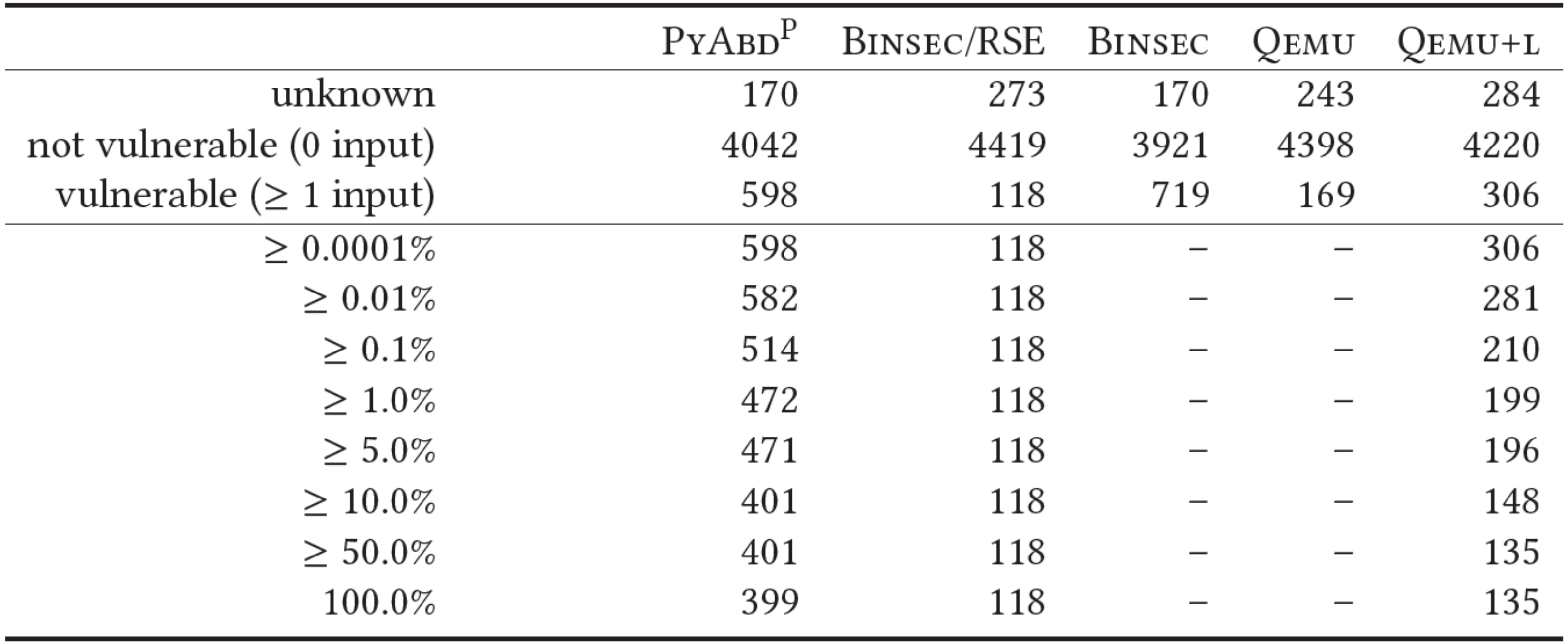
- Physical perturbation of the system executing the program
- Changes the program behavior
- Introduces new bugs
- How does each method characterize these bugs?

**VerifyPINs**

- 10 protected implementations
- 4800 faulted binary programs

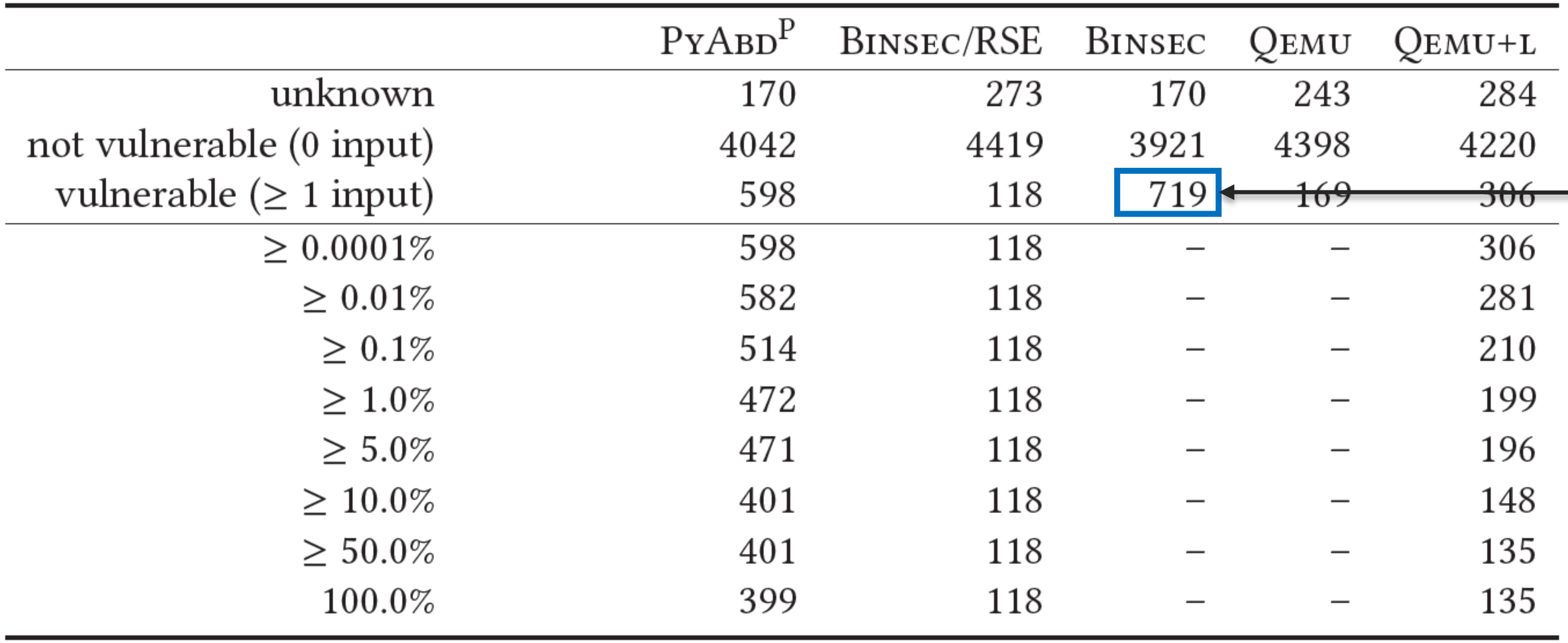


# Number of faulted programs with at least a given proportion of input states triggering the bug



| | PyAbd$^{P}$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---:|---:|---:|---:|---:|---:|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable ($\geq$ 1 input) | 598 | 118 | 719 | 169 | 306 |
| $\geq$ 0.0001% | 598 | 118 | – | – | 306 |
| $\geq$ 0.01% | 582 | 118 | – | – | 281 |
| $\geq$ 0.1% | 514 | 118 | – | – | 210 |
| $\geq$ 1.0% | 472 | 118 | – | – | 199 |
| $\geq$ 5.0% | 471 | 118 | – | – | 196 |
| $\geq$ 10.0% | 401 | 118 | – | – | 148 |
| $\geq$ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

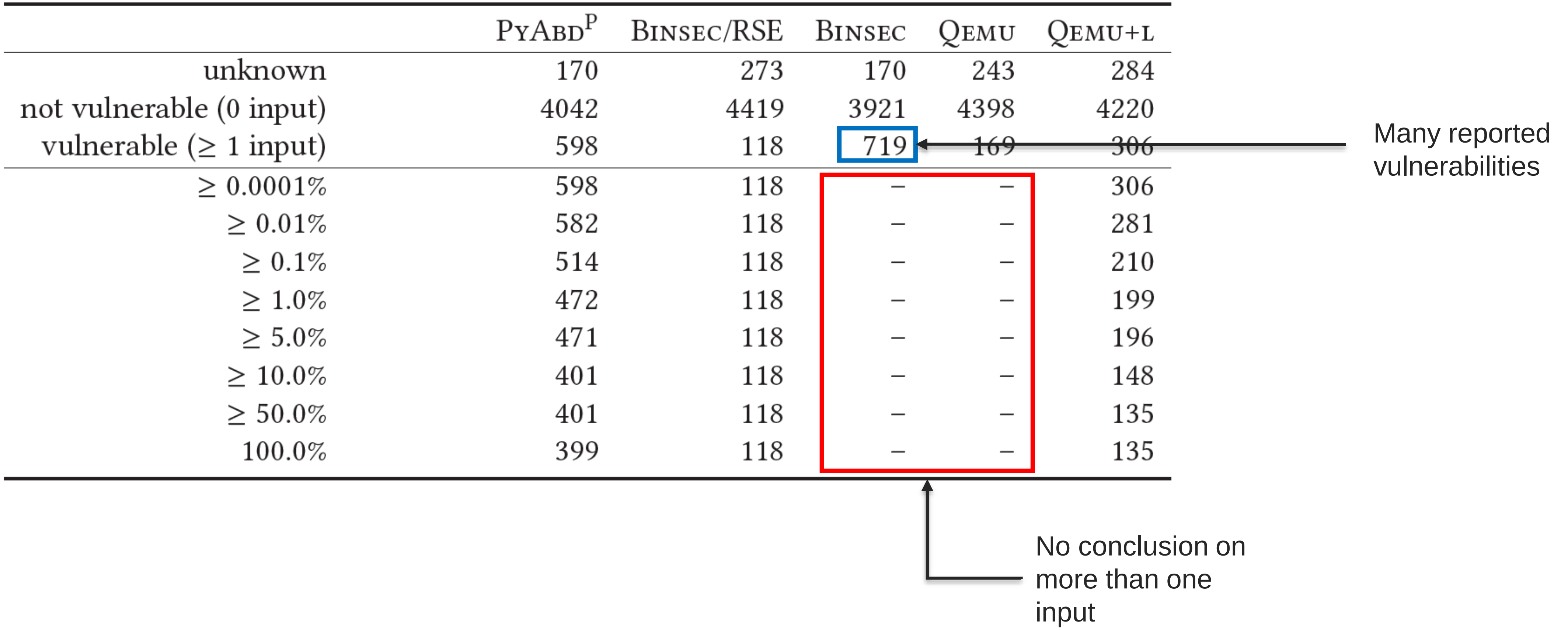
# Number of faulted programs with at least a given proportion of input states triggering the bug



| | PyAbd$^P$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---|---|---|---|---|---|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable (≥ 1 input) | 598 | 118 | 719 | 169 | 306 |
| ≥ 0.0001% | 598 | 118 | – | – | 306 |
| ≥ 0.01% | 582 | 118 | – | – | 281 |
| ≥ 0.1% | 514 | 118 | – | – | 210 |
| ≥ 1.0% | 472 | 118 | – | – | 199 |
| ≥ 5.0% | 471 | 118 | – | – | 196 |
| ≥ 10.0% | 401 | 118 | – | – | 148 |
| ≥ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

Many reported vulnerabilities

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

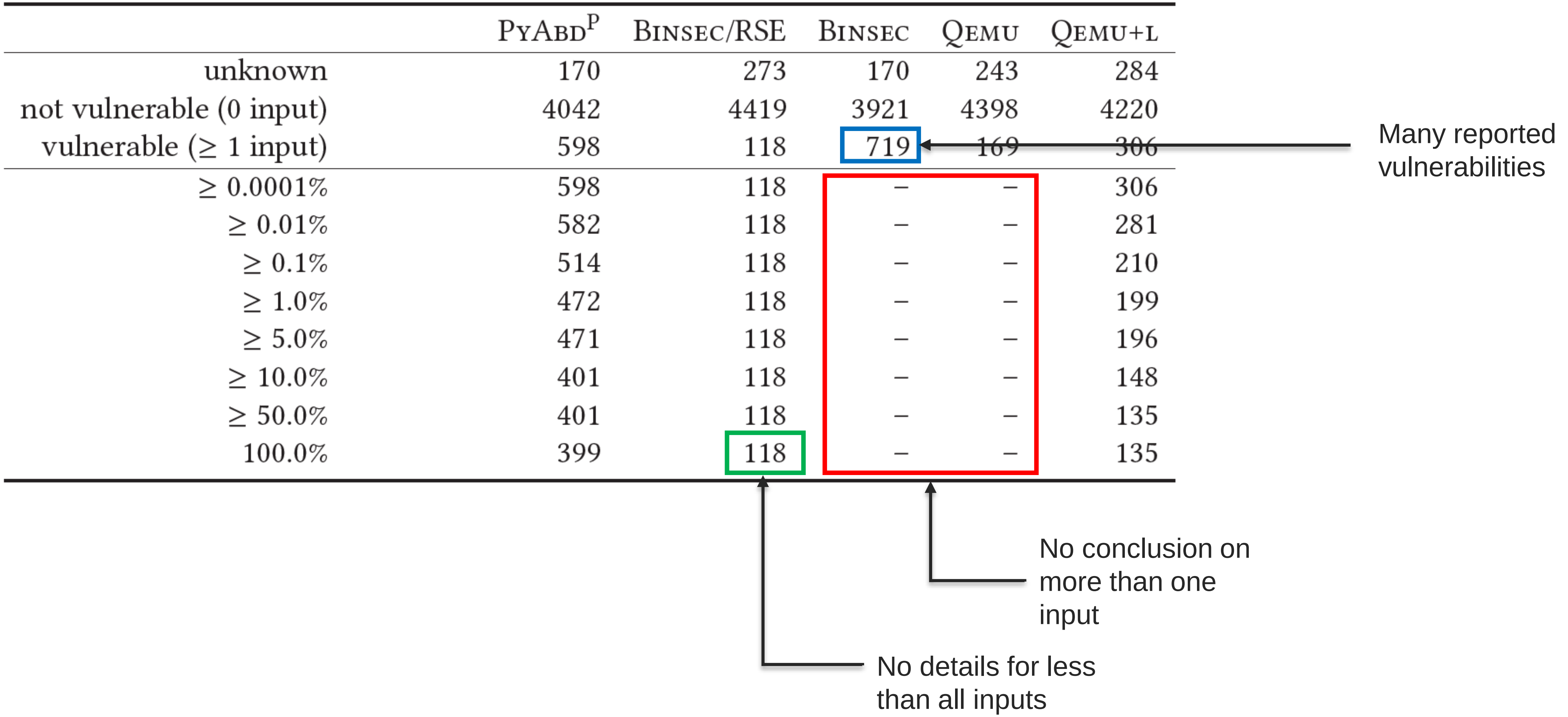


| | PyAbd$^{P}$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---|---|---|---|---|---|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable (≥ 1 input) | 598 | 118 | 719 | 169 | 306 |
| ≥ 0.0001% | 598 | 118 | – | – | 306 |
| ≥ 0.01% | 582 | 118 | – | – | 281 |
| ≥ 0.1% | 514 | 118 | – | – | 210 |
| ≥ 1.0% | 472 | 118 | – | – | 199 |
| ≥ 5.0% | 471 | 118 | – | – | 196 |
| ≥ 10.0% | 401 | 118 | – | – | 148 |
| ≥ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

Many reported vulnerabilities

No conclusion on more than one input

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



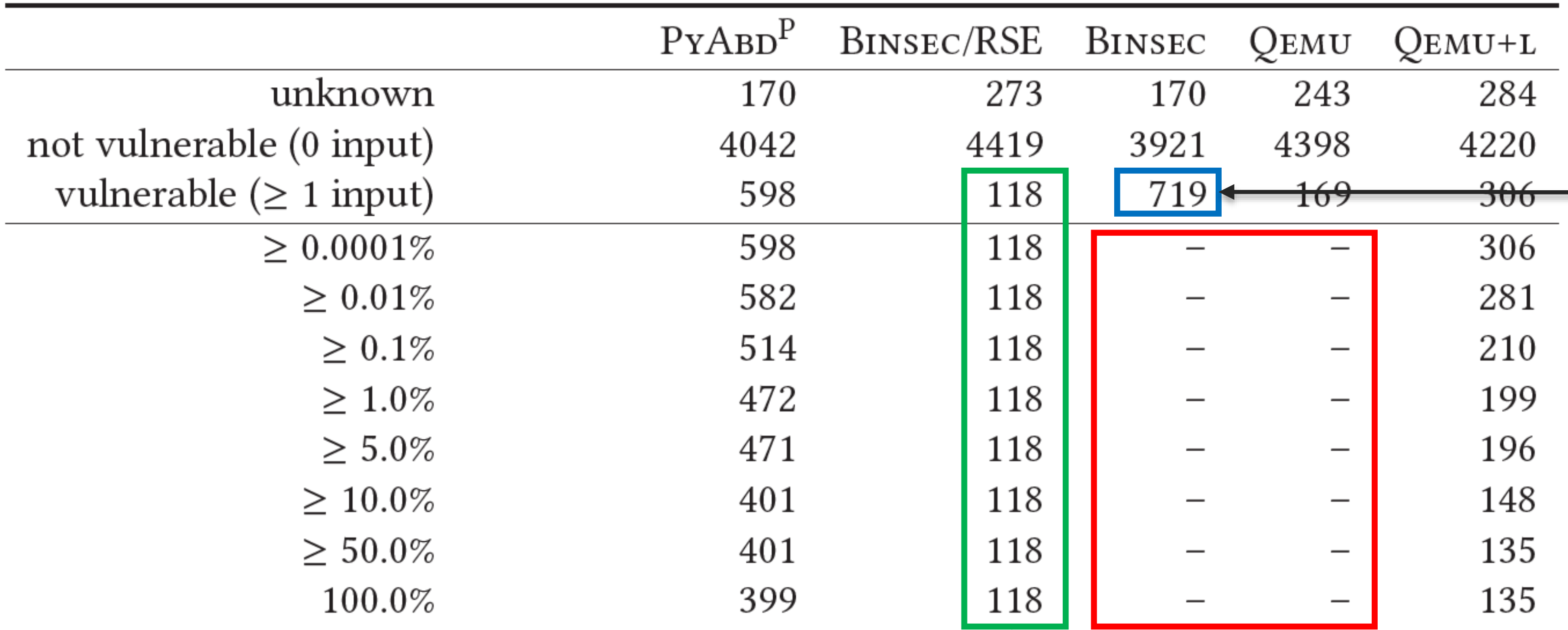
| | PyAbd$^{P}$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---|---|---|---|---|---|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable (≥ 1 input) | 598 | 118 | 719 | 169 | 306 |
| ≥ 0.0001% | 598 | 118 | – | – | 306 |
| ≥ 0.01% | 582 | 118 | – | – | 281 |
| ≥ 0.1% | 514 | 118 | – | – | 210 |
| ≥ 1.0% | 472 | 118 | – | – | 199 |
| ≥ 5.0% | 471 | 118 | – | – | 196 |
| ≥ 10.0% | 401 | 118 | – | – | 148 |
| ≥ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

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



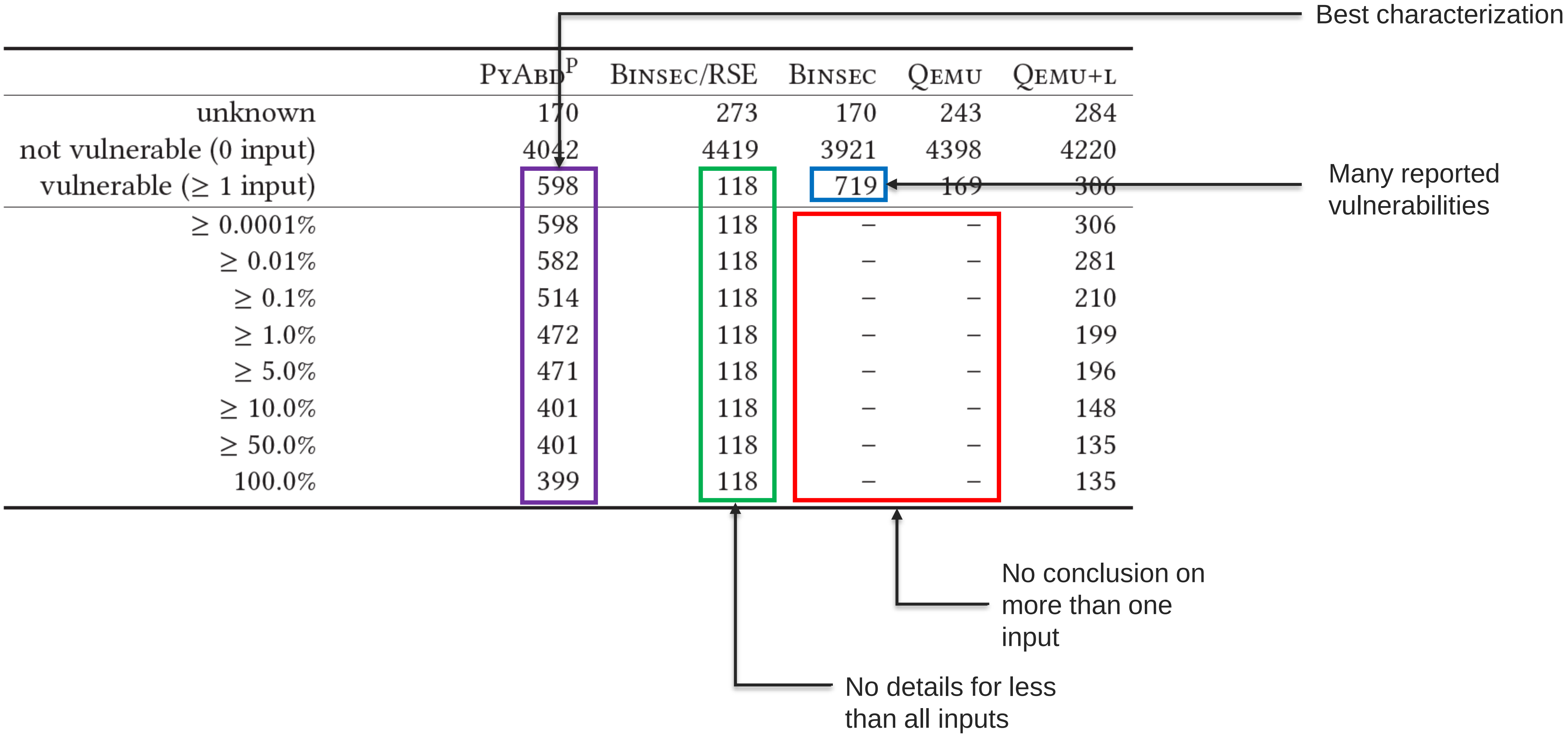
| | PyAbd$^P$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---|---|---|---|---|---|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable ($\geq$ 1 input) | 598 | 118 | 719 | 169 | 306 |
| $\geq$ 0.0001% | 598 | 118 | – | – | 306 |
| $\geq$ 0.01% | 582 | 118 | – | – | 281 |
| $\geq$ 0.1% | 514 | 118 | – | – | 210 |
| $\geq$ 1.0% | 472 | 118 | – | – | 199 |
| $\geq$ 5.0% | 471 | 118 | – | – | 196 |
| $\geq$ 10.0% | 401 | 118 | – | – | 148 |
| $\geq$ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

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



| | PyAbd$^P$ | Binsec/RSE | Binsec | Qemu | Qemu+L |
|---|---|---|---|---|---|
| unknown | 170 | 273 | 170 | 243 | 284 |
| not vulnerable (0 input) | 4042 | 4419 | 3921 | 4398 | 4220 |
| vulnerable (≥ 1 input) | 598 | 118 | 719 | 169 | 306 |
| ≥ 0.0001% | 598 | 118 | – | – | 306 |
| ≥ 0.01% | 582 | 118 | – | – | 281 |
| ≥ 0.1% | 514 | 118 | – | – | 210 |
| ≥ 1.0% | 472 | 118 | – | – | 199 |
| ≥ 5.0% | 471 | 118 | – | – | 196 |
| ≥ 10.0% | 401 | 118 | – | – | 148 |
| ≥ 50.0% | 401 | 118 | – | – | 135 |
| 100.0% | 399 | 118 | – | – | 135 |

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

- User[0] == User[1] && User[0] == User[2] && User[0] == User[3] && User[0] != 0

  Authentication when all digits are equal and non zero

- Card[2] != User[2] && Card[3] == User[3] && User[1] == 5

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

- R0 == User[3] && User[3] == User[2] && User[3] == User[1] && User[3] == User[0]

  Authentication with four time the initial value of R0

- R2 = 0xaa && R1 != 0x55 && R1 != 0

  Authentication if R2=0xaa initially and R1 distinct from both 0x55 and 0x00 initially

# Conclusion

## Conclusion

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

- We adapt theory-agnostic abduction algorithm to ∃∀ formulas and apply it at program-level through oracles

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

# Conclusion

## Conclusion

- We propose a precondition inference technique to improve the capabilities of Robust Reachability
- We adapt theory-agnostic abduction algorithm to ∃∀ 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

Artifacts Available
acm
V1.1

BINSEC

# Conclusion

### Conclusion

- We propose a precondition inference technique to improve the capabilities of Robust Reachability
- We adapt theory-agnostic abduction algorithm to ∃∀ 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?

Artifacts Available acm V1.1

BINSEC

# Questions

Artifacts Available
acm
V1.1

BINSEC