Inference of Robust Reachability Constraints

Yanis Sellami[1,2], 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

Automatic Bug Detection

Programs have bugs

Bugs can be exploited → Vulnerabilities

```
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

```
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 → Vulnerabilities

```
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

a = 0, b = 0

False Positive in Practice

Example

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

False Positive in Practice

Example

```
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

```
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

```
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
[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 → a

uncontrolled → b

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

$\exists\ a \quad \forall\ b \quad \text{error}$

Focus: Reliable Bugs

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

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

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

$\exists$ a $\forall$ b error

Not Robustly Reachable

Focus: Reliable Bugs

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

Extension of Reachability and Symbolic Execution

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) & 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

safe

buggy

```
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

```
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)          <- safe
            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];   <- buggy
}
```

The Remaining Problem

memory alignment constraint

Example 3

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

- Reachability: Vulnerable

safe

buggy

```
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
- Robust Reachability: Not reliably triggerable
  - Taking the fast path depends on uncontrolled initial values

memory alignment constraint

```
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];
}
```

safe

buggy

$\exists * src, \forall src, dst,$ overflow?

Not Robustly Reachable

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

```
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];
}
```

Robust Reachability Constraints

Definition

- Predicate on program input sufficient to have Robust Reachability

```
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];
}
```

$$\exists *src,\ \forall src, dst,\ \boxed{src \,\%\, 32 = 0\ \wedge dst \,\%\, 32 = 0} \Rightarrow \text{overflow}$$

(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

```
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];
}
```

$$\exists *src,\ \forall src, dst,\ \boxed{src \,\%\, 32 = 0\ \wedge dst \,\%\, 32 = 0} \Rightarrow \text{overflow}$$

(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

How to Automatically Generate Such Constraints?

```
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];
}
```

$$\exists * src,\ \forall src, dst,\ \boxed{src \% 32 = 0\ \wedge dst \% 32 = 0} \Rightarrow \text{overflow}$$

(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 $\boldsymbol{\phi}$ such that $\exists\, \boldsymbol{C\ controlled\ value}, \forall\, \boldsymbol{U\ uncontrolled\ value}, \boldsymbol{\phi(C,U) \Longrightarrow reach(C,U)}$

Abduction of Robust Reachability Constraints

Abductive Reasoning

[Josephson and Josephson, 1994]

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

Abduction of Robust Reachability Constraints

Abductive Reasoning

[Josephson and Josephson, 1994**]**

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

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 ϕ_M in $\phi_H \wedge \phi_M \models \phi_G$

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 ϕ_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)

$\vec{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

Our Solution (Framework)

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

$\vec{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

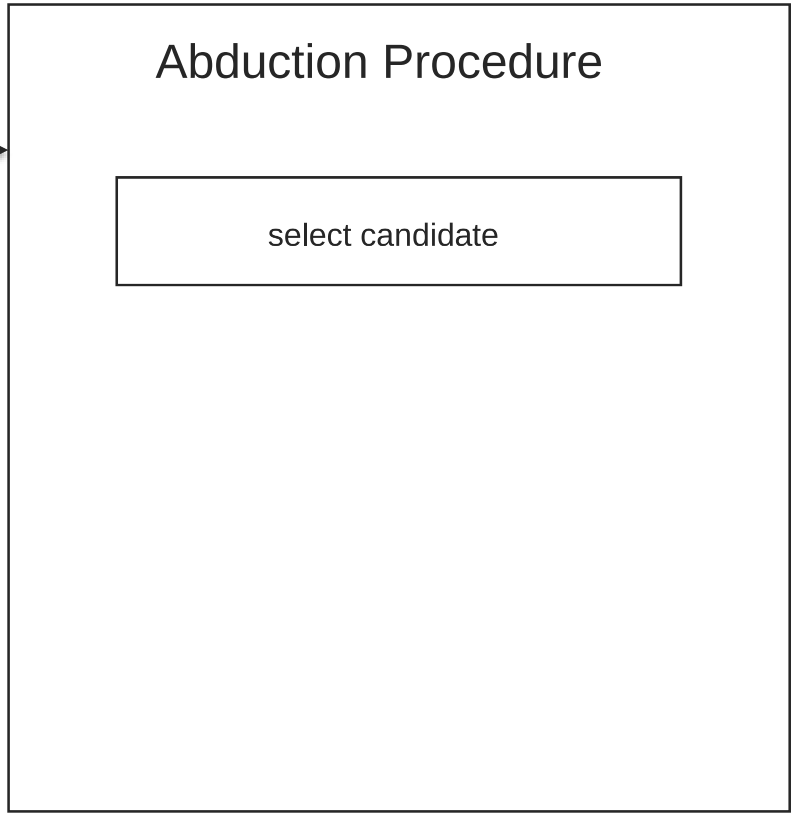
Our Solution (Framework)

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

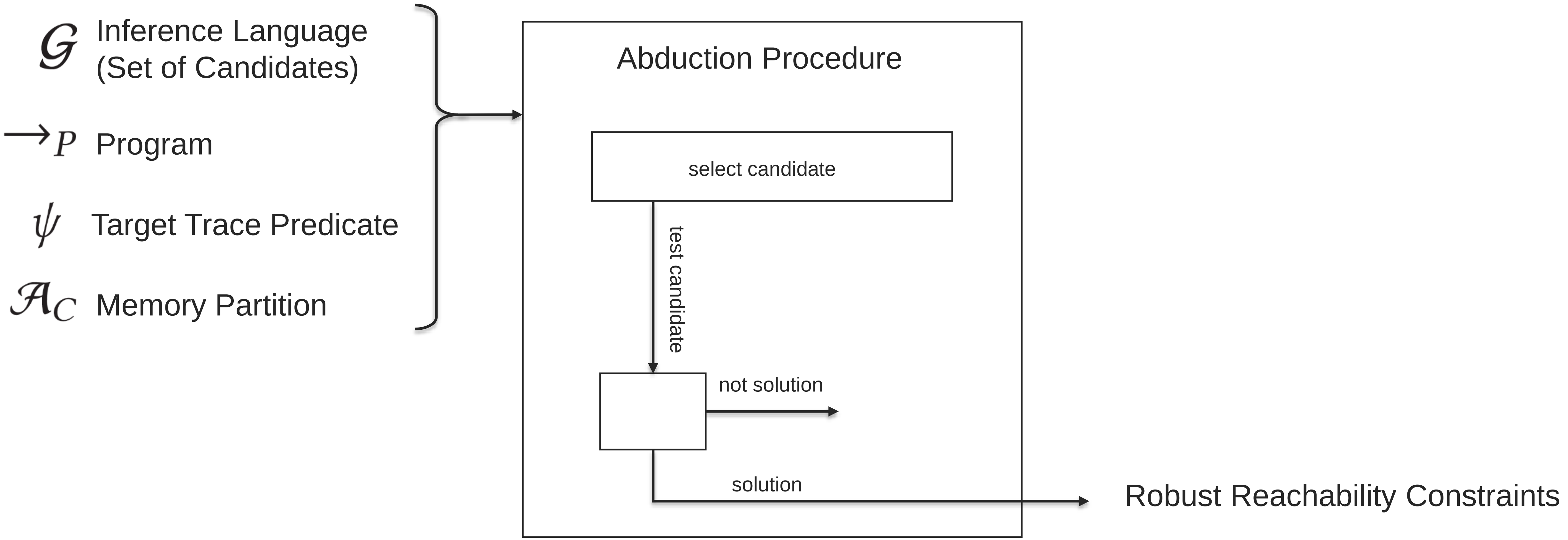
$\vec{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



Our Solution (Framework)



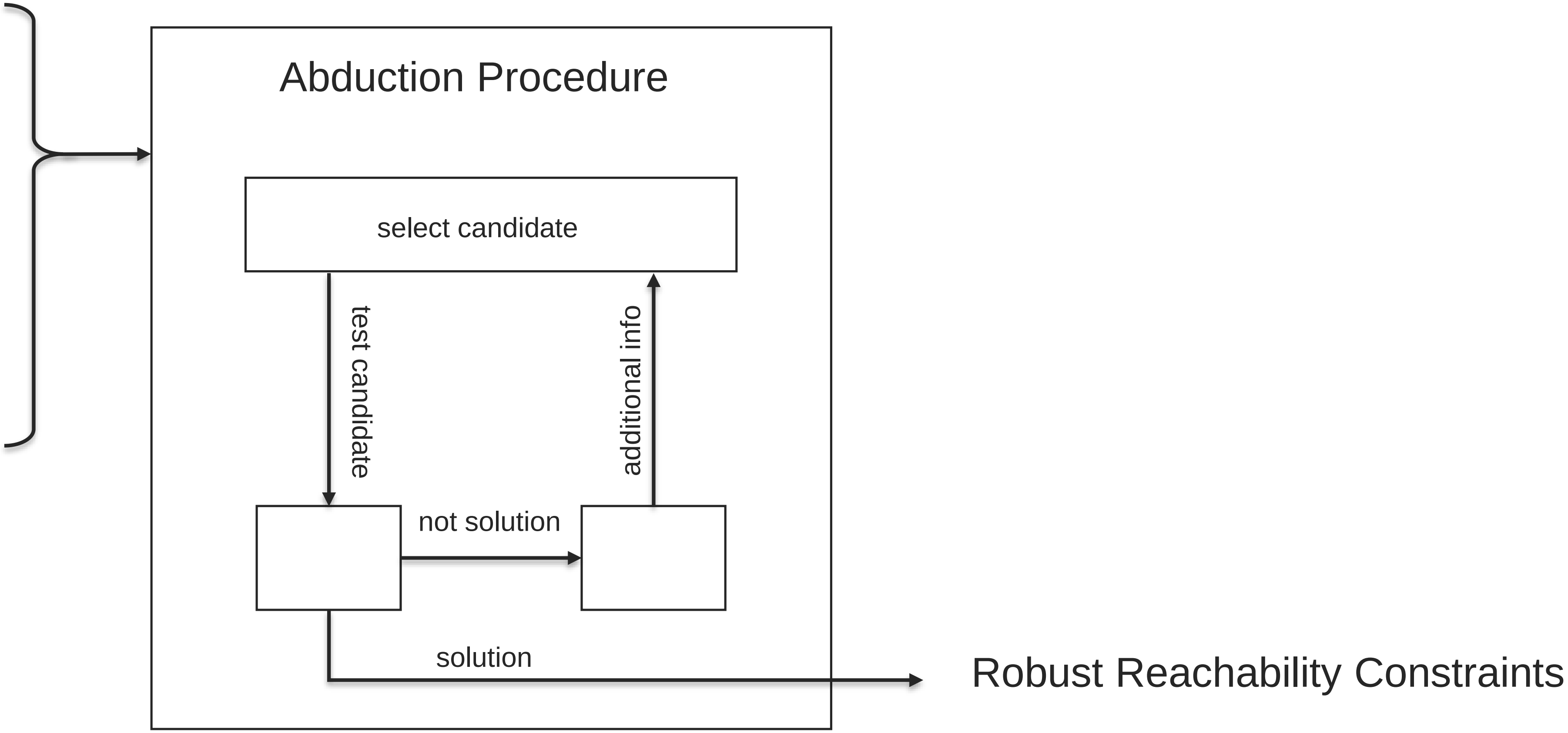
Our Solution (Framework)

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

$\overrightarrow{P}$ Program

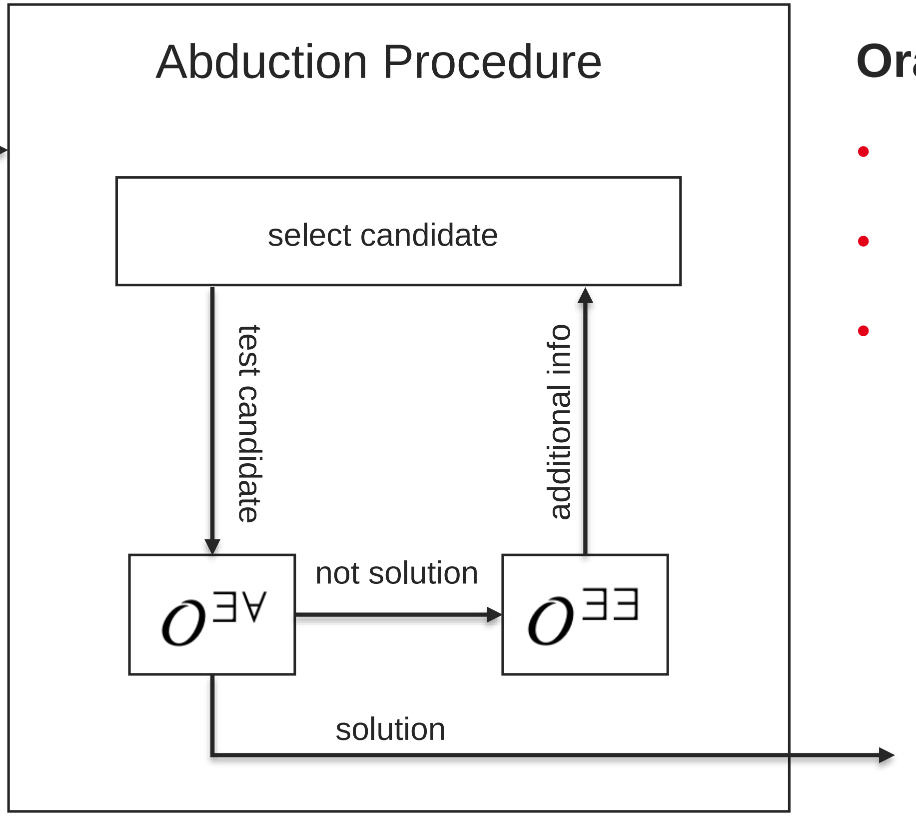
ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



Our Solution (Framework)

- $\mathcal{G}$ Inference Language (Set of Candidates)
- $\overrightarrow{P}$ Program
- ψ Target Trace Predicate
- $\mathcal{A}_C$ Memory Partition



Robust Reachability Constraints

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, …)

Our Solution (Baseline Algorithm)

$\textsc{BaselineRCInfer}(\mathcal{G}, \rightarrow_P, \psi, \mathcal{A}_C)$

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)

$\textsc{BaselineRCInfer}(\mathcal{G}, \rightarrow_P, \psi, \mathcal{A}_C)$

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

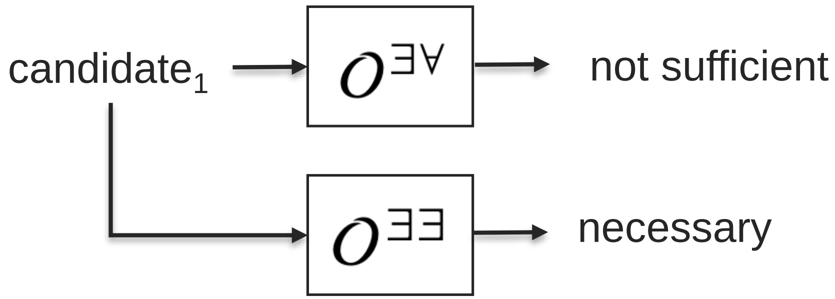
Key Strategies for Efficient Exploration

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

Making it Work: Necessary Constraints

The Idea

- Find and store Necessary Constraints



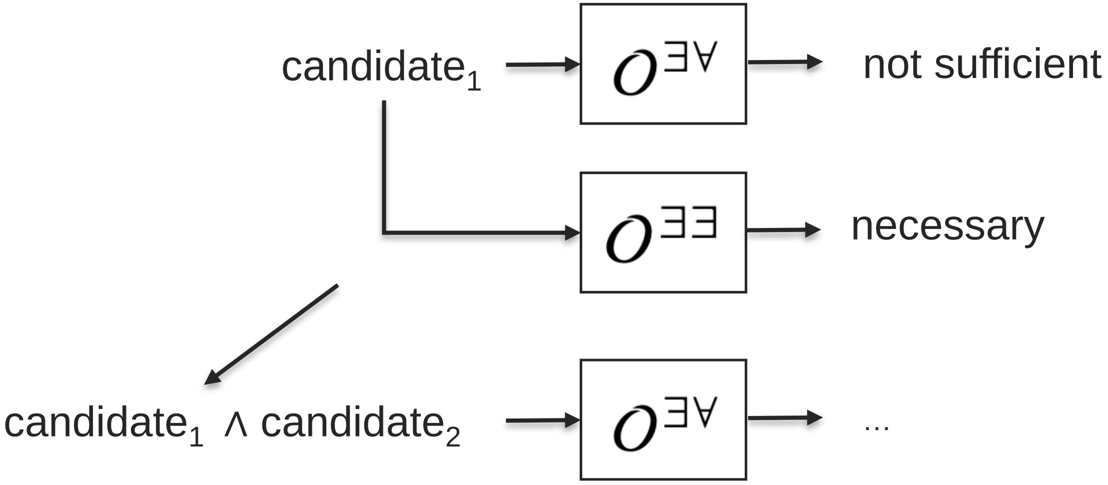
Making it Work: Necessary Constraints

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

$$\text{candidate}_1 \longrightarrow \boxed{\mathcal{O}^{\exists\forall}} \longrightarrow \text{not sufficient}$$

The Issue

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

Making it Work: Counter-Examples

The Idea

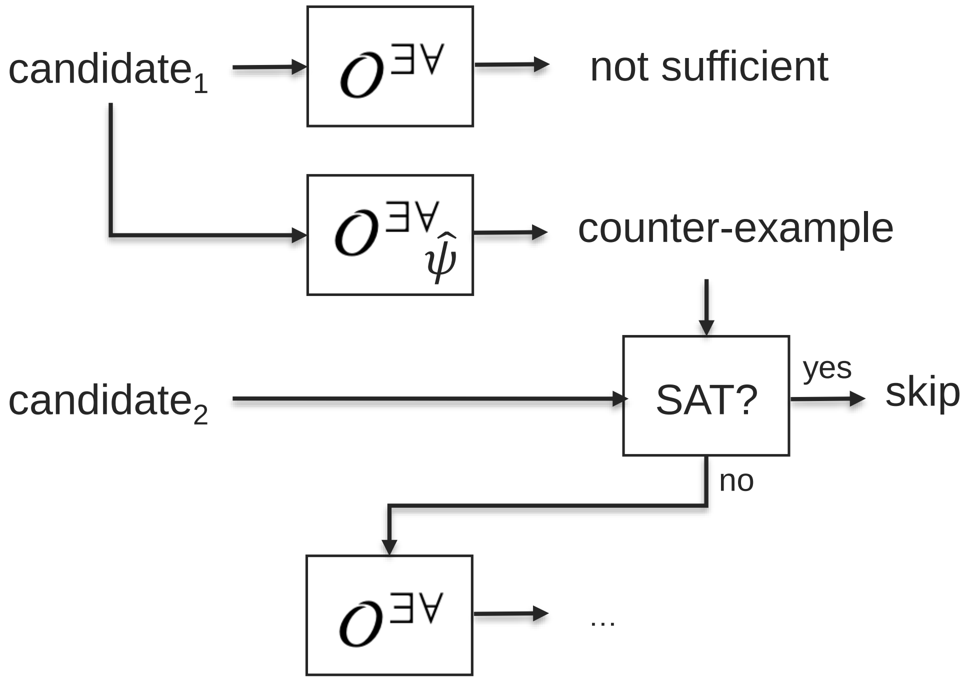
- Reuse information from failed candidate checks

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



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 (RQ1, RQ2)

	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 $(I_\mathcal{G})$ $\rightarrow$ More expressivity but more candidates

We can find more reliable bugs than Robust Symbolic Execution

Results: Influence of the ‘Efficient Strategies’ (RQ4)

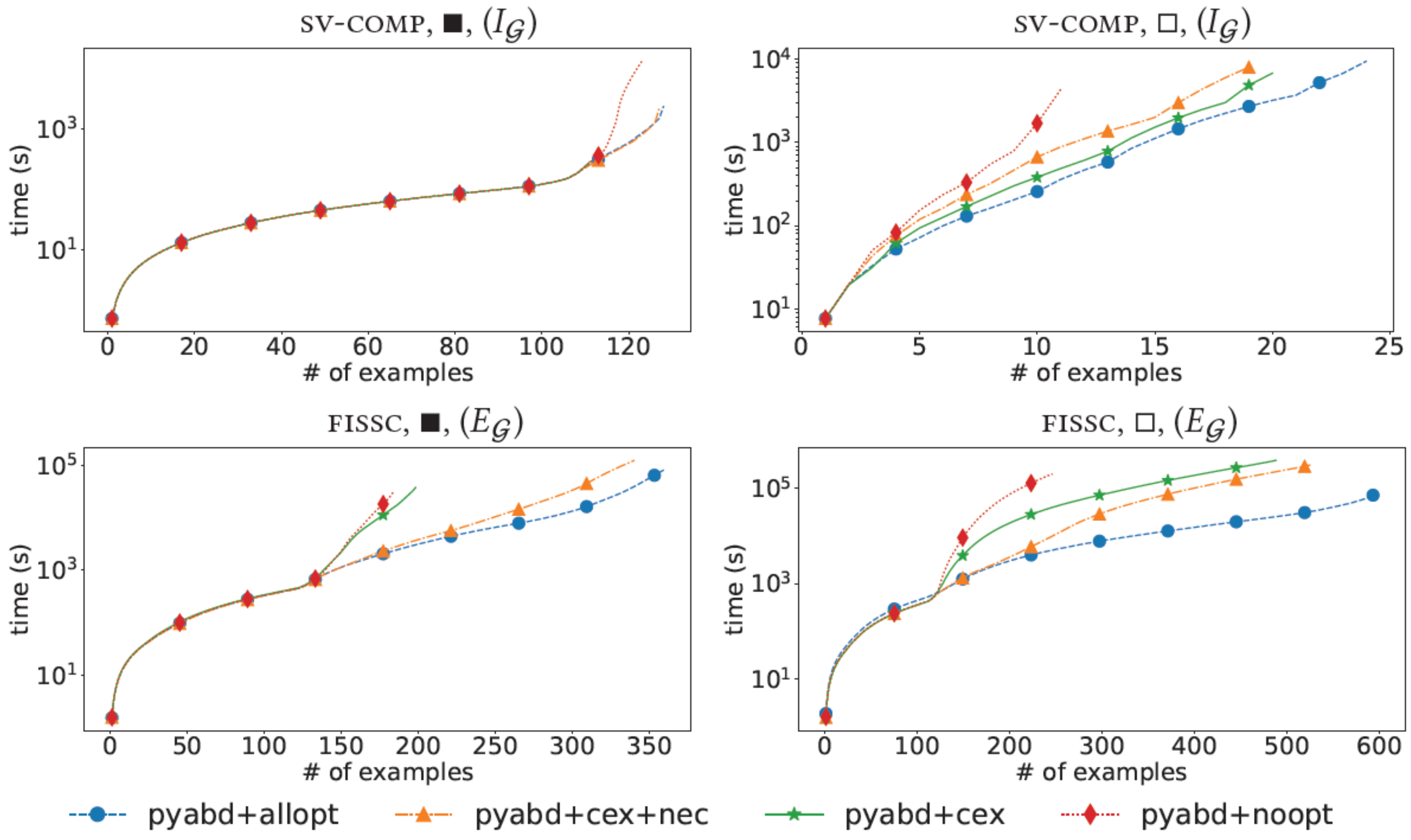


Fig. 5. Cactus plot showing the influence of the strategies of Section 5 on the computation of the first sufficient k-reachability constraint with PyAbd.

Significantly improves the capabilities of the method

Each strategy matters

Results: Vulnerability Characterization on a Fault-Injection Benchmark

	PyAbd	Binsec/RSE	Binsec
unknown	170	273	170
not vulnerable (0 input)	4414	4419	3921
vulnerable ($\geq$ 1 input)	226	118	719
$\geq$ 0.0001%	226	118	–
$\geq$ 0.01%	209	118	–
$\geq$ 0.1%	173	118	–
$\geq$ 1.0%	167	118	–
$\geq$ 5.0%	166	118	–
$\geq$ 10.0%	118	118	–
$\geq$ 50.0%	118	118	–
100.0%	118	118	–

Our Solution:

- Finds and characterize vulnerabilities in-between Reachability and Robust Reachability

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

BINSEC

(hiring)

Conclusion

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

BINSEC

(hiring)

Conclusion

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?

(hiring)

Questions

BINSEC

(hiring)