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

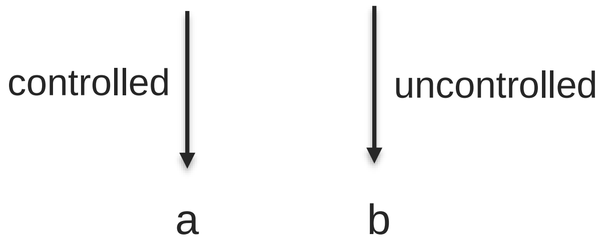
We need to characterize the replicability of bugs

Robust Reachability [Girol et. al., 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
        ...
}
```



Robust Reachability [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

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

controlled uncontrolled

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

Robust Reachability [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

```
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 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

```
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

Example 3

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

```
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

safe

buggy

```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

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

memory alignment constraint

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

safe

buggy

```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];
}
```

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

```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];
}
```

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];
}
```

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

```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];
}
```

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

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

Abduction of Robust Reachability Constraints

Abductive Reasoning

[Josephson and Josephson, 1994]

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

Abduction of Robust Reachability Constraints

Abductive Reasoning

[Josephson and Josephson, 1994]

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

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

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

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)

$\overrightarrow{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

Abduction Procedure

Our Solution (Framework)

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

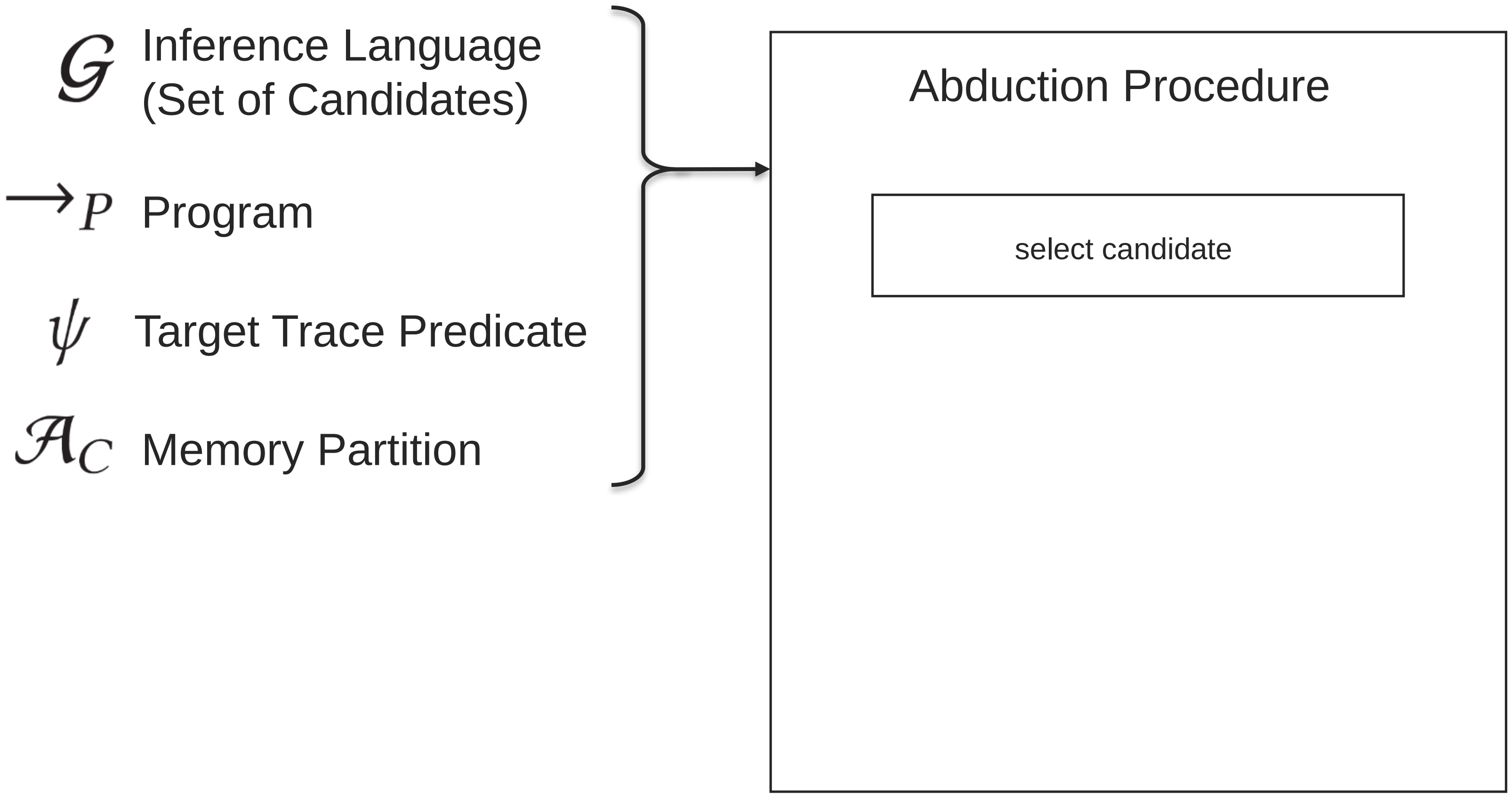
$\overrightarrow{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition

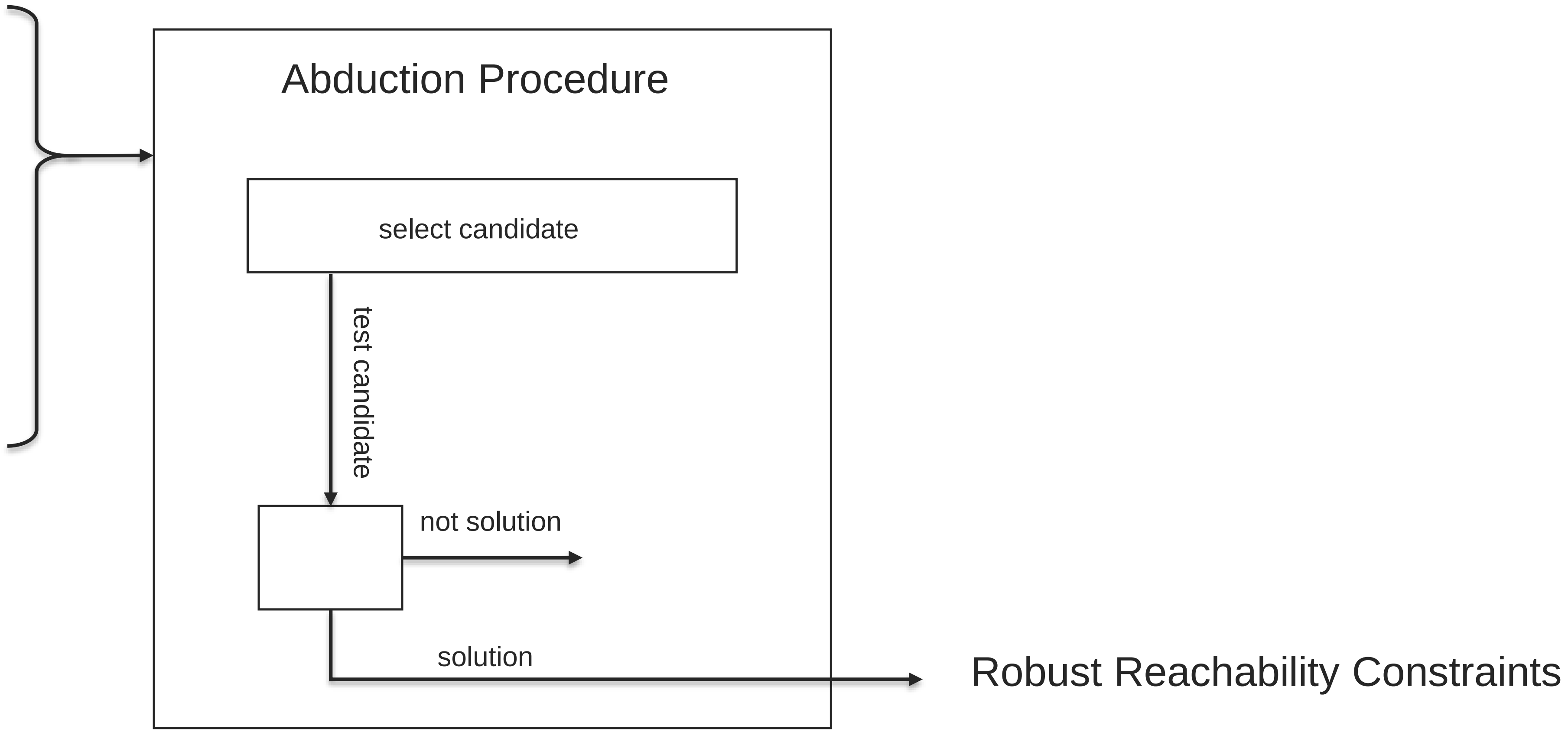
Abduction Procedure

Our Solution (Framework)



Our Solution (Framework)

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



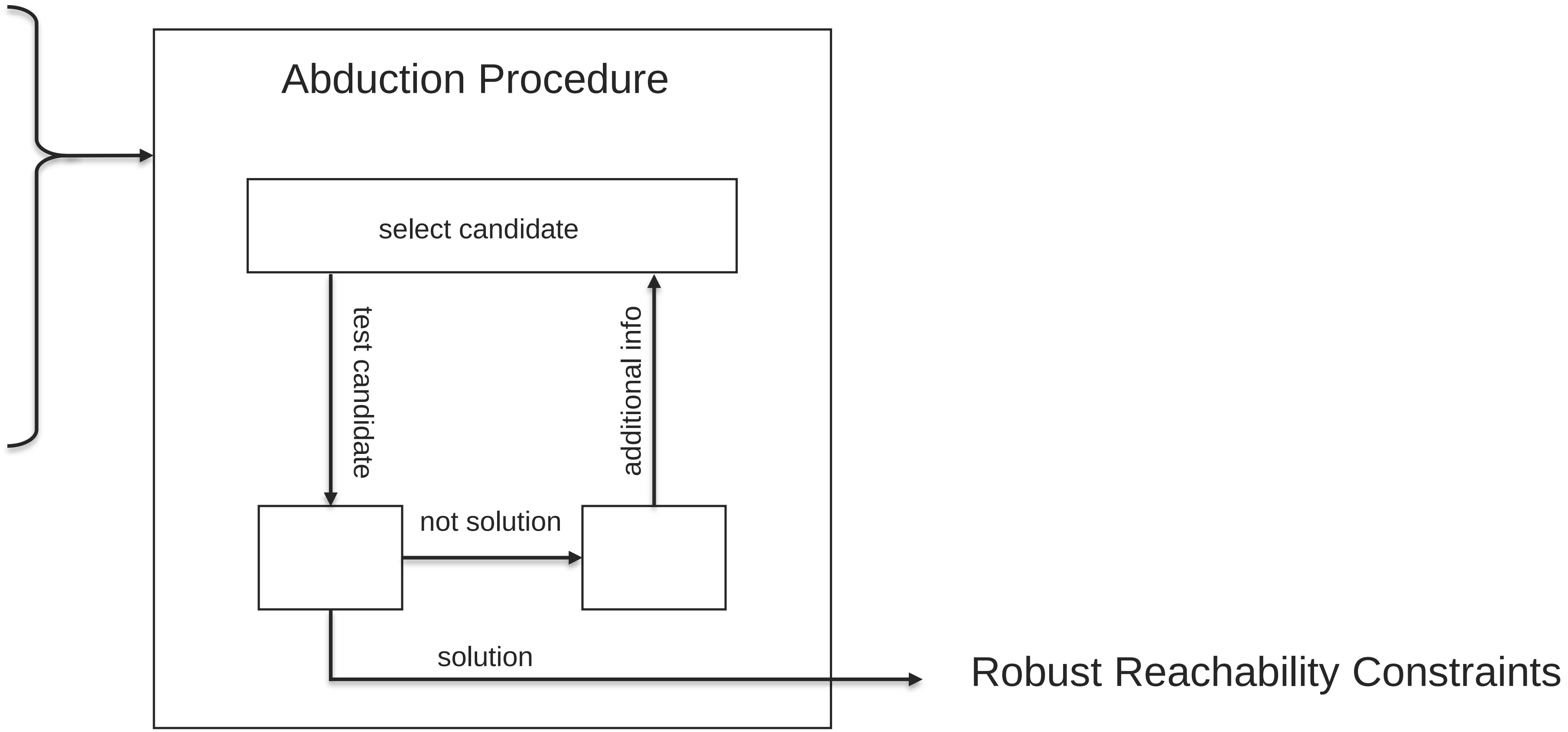
Our Solution (Framework)

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

$\vec{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



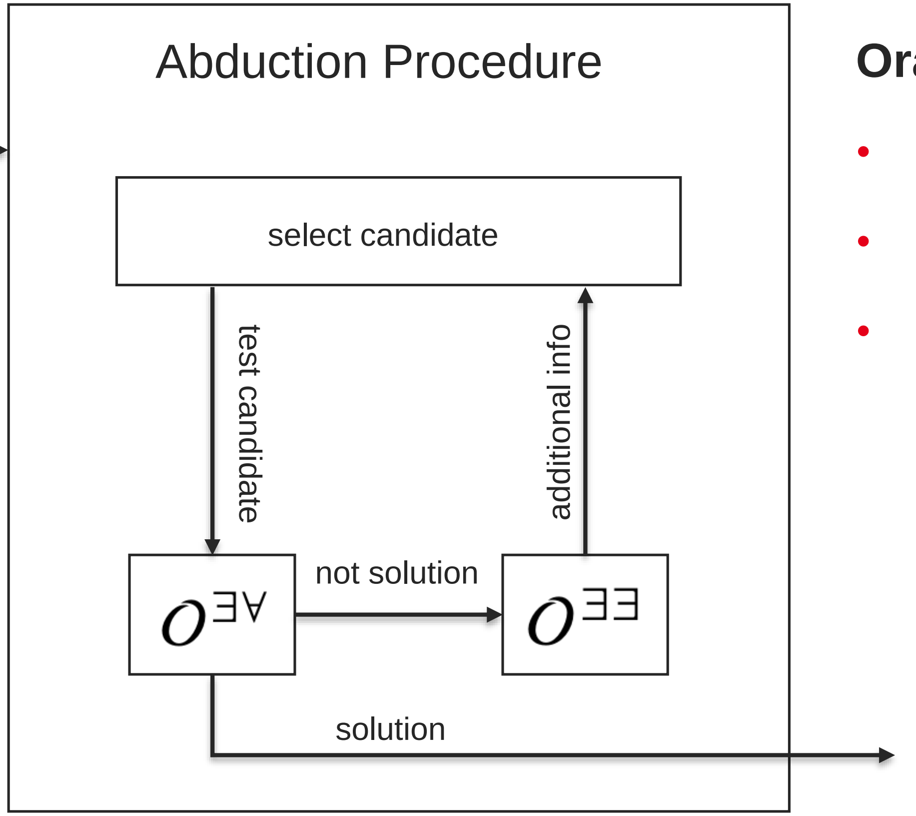
Our Solution (Framework)

$\mathcal{G}$ 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)

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

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

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 $\{\perp\}$;

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

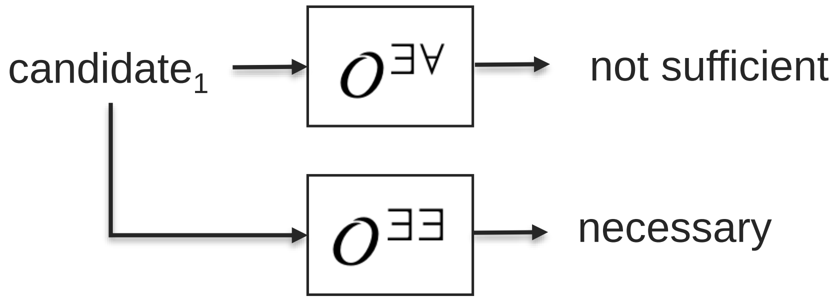
Key Strategies for Efficient Exploration

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

Making it Work: Necessary Constraints

The Idea

- Find and store Necessary Constraints



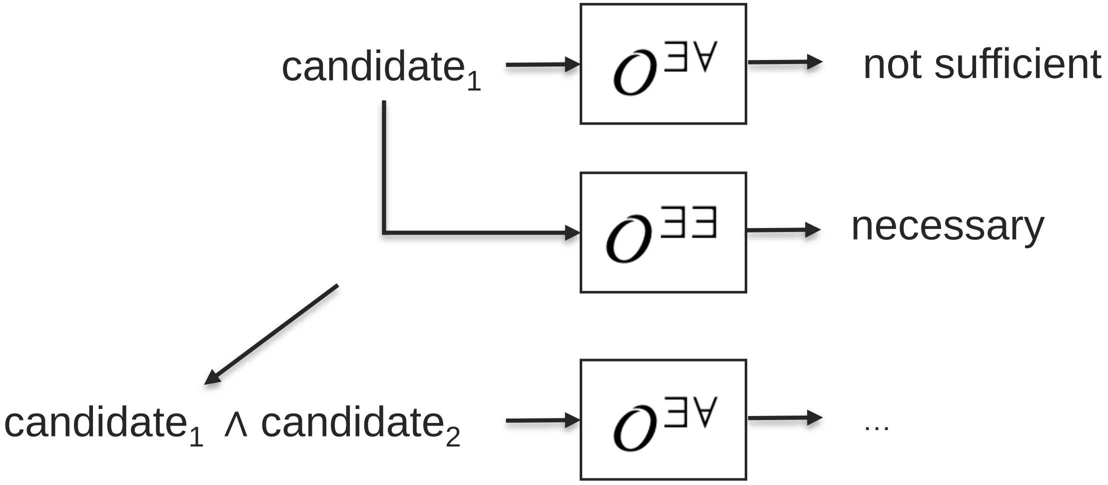
Making it Work: Necessary Constraints

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

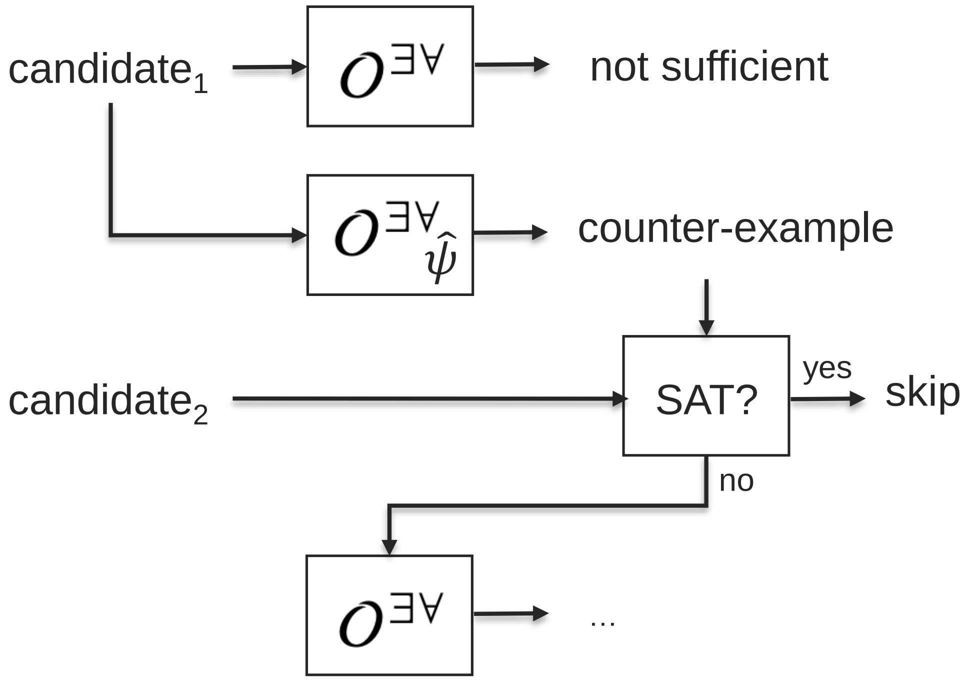
- Reuse information from failed candidate checks

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

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

We can find more reliable bugs than Robust Symbolic Execution

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

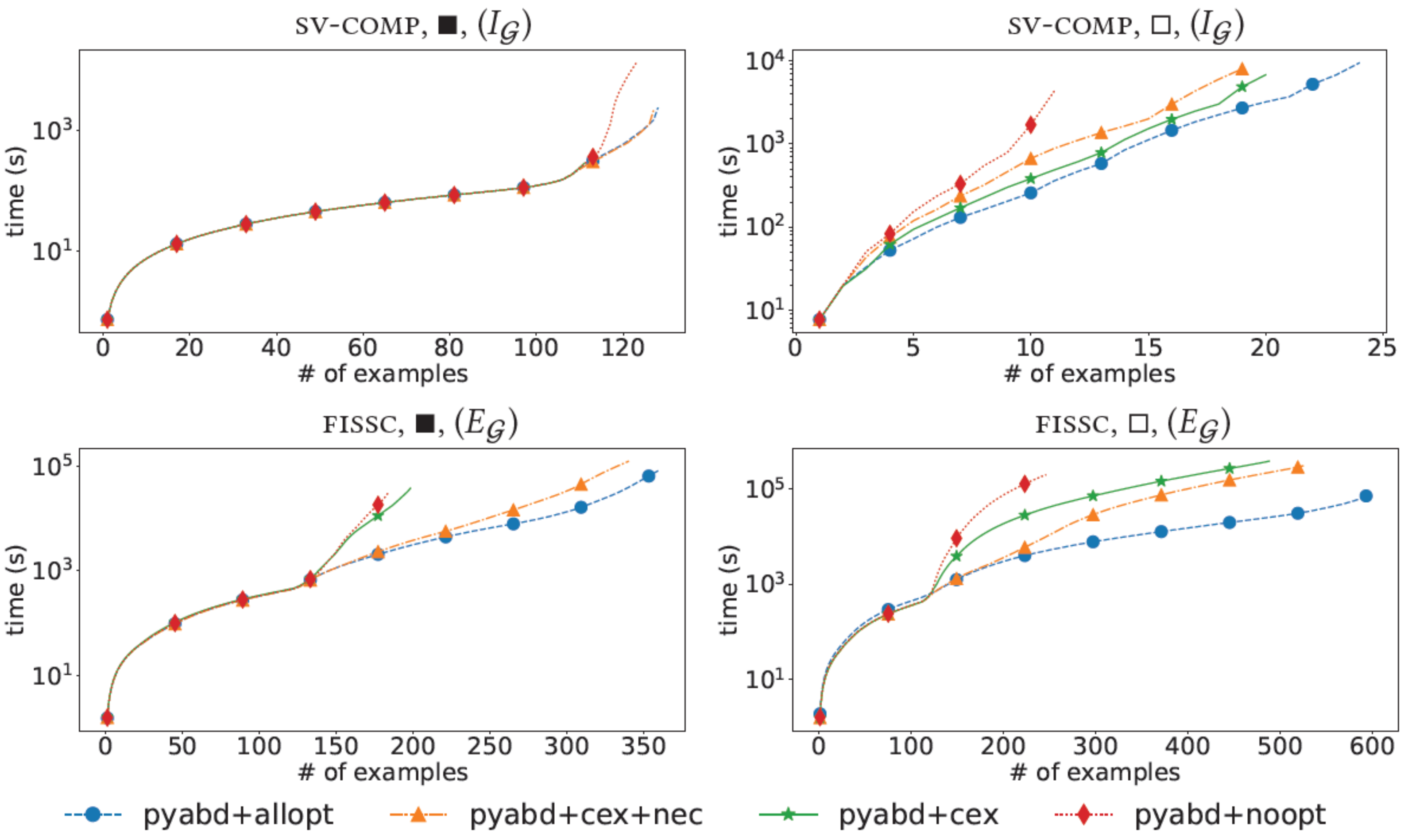


Fig. 5. Cactus plot showing the influence of the strategies of Section 5 on the computation of the first sufficient k-reachability constraint with PyAbd.

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

(hiring)

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

(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?

BINSEC

(hiring)

Questions

Artifacts Available

acm

V1.1

BINSEC

(hiring)

Results: Example of FISSC Constraints

- CardPIN[0] == UserPIN[0] && UserPIN[0] == 3
  Authentication when first digit is 3
- UserPIN[0] == UserPIN[1] && UserPIN[0] == UserPIN[2] && UserPIN[0] == UserPIN[3] && UserPIN[0] != 0
  Authentication when all digits are equal and non zero
- CardPIN[2] != UserPIN[2] && CardPIN[3] == UserPIN[3] && UserPIN[1] == 5
  Authentication when we know the last digit, the 3rd is not correct and the 2^{nd} is 5.
- R0 == UserPIN[3] && UserPIN[3] == UserPIN[2] && UserPIN[3] == UserPIN[1] && UserPIN[3] == UserPIN[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

Making it Work: Ordering Candidates

The Idea

- Some candidates are more likely to be solutions
- Try to guess which ones and try them first

What we already have

- Examples satisfying the goal
- Syntactic information on the candidates

Proposal

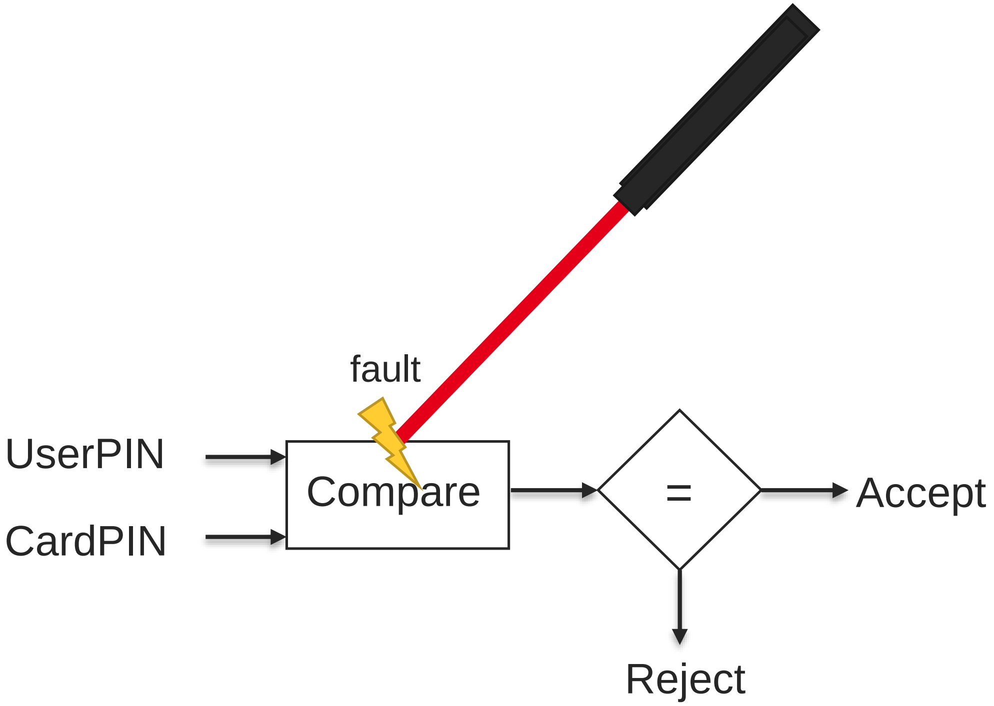
- Use an ordering heuristic on the candidates

$$\phi \mapsto (count(\wedge, \phi), -\textsc{Card}(\{s \in V \mid \phi[s]\}))$$

Benchmark: FISSC

Fault Injection Attacks

- Physical perturbation of the system executing the program
- Changes the program behavior
- Leverages new vulnerabilities
- Goal: Characterize these vulnerabilities



VerifyPINs

- 10 protected implementations
- 4800 faulted binary programs

Benchmark: SVComp

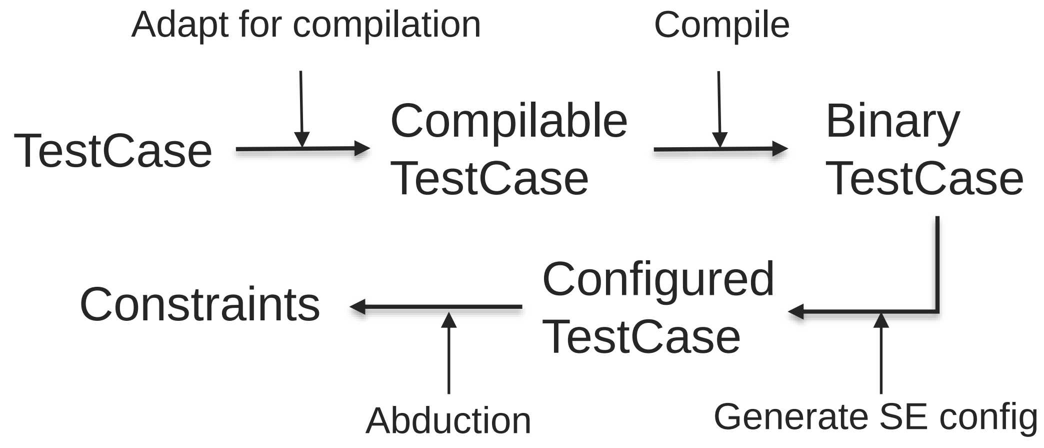
SVComp

- Collection of software verification tasks
- Evaluate the generative capabilities of our method
- Select 147 reachable reachability test cases for which BINSEC answers under 1 minute

Setup

- Use the target location of the benchmark
- Use the input variables as program input
- Abduce constraints on these variables to reach the target location

Process



Example

```c
void init (int* x)
{
        for (int i = 0; i < 4; i++)
                x[i] = __VERIFIER_nondet_int();
}

int verify(int * x)
{
        int stuff[4] = { 1, 2, 3, 4 };
        for (int i = 0; i < 4; i++)
                if (x[i] != stuff[i])
                        return 1;
        return 0;
}
```

```c
int main ()
{
  int x[4];

  init(x);
  assert(verify(x) != 0);

  return 0;
}
```

Inference Languages

Program Variables

$$\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}}$$

Equalities

$*a_8 = *a'_8$ $\quad$ $*a_{32} = *a'_{32}$

$*a_8 = v_8$ $\quad$ $*a_{32} = v_{32}$

Register-Memory Bytes Equalities

$*a_{32} = \texttt{0x000000}:(*a_8)$

$*a_{32} = \texttt{0x000000}:v_8$

Inequalities, Negation, Conjunction

$*a_8 \leq *a'_8$

$\neg \langle nliteral \rangle$

$*a_{32} \leq *a'_{32}$

$*a_8 \leq v_8$

$\langle constraint \rangle \wedge \langle constraint \rangle$

Two Inference Languages

- One with equalities and disequalities $(E_{\mathcal{G}})$
- One with added inequalities $(I_{\mathcal{G}})$

Controlled Variables

- Recovered from the Symbolic Execution Queries
- One setup with controlled variables ■
- One setup without □

Final Algorithm

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

Input: $\mathcal{G}$: inference language, $\rightarrow_P$: program, ψ: prop, $\widehat{\psi}$: prop breaking ψ, $\mathcal{A}_C$: 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 ψ 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 δ_R *and* $\top, s \leftarrow O^{\exists\exists}(\rightarrow_P, \psi, \phi)$ **then** // ensure ψ satisfiable under ϕ
      $V \longleftarrow V \cup \{s\}$; // new trace example
      if $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ **then** // check candidate ϕ
        $R \longleftarrow \Delta_{min}(R \cup \{\phi\})$; // 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 δ_R **then**
      $N \longleftarrow N \cup \{\neg\phi\}$; // new necessary constraint
    if δ_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, ψ: prop, $\widehat{\psi}$: prop breaking ψ, $\mathcal{A}_C$: controlled variables, V: examples of input states of $\rightarrow_P$ satisfying ψ, R: known sufficient constraints, N: known necessary constraints, U: known breaking constraints, prunef: strategy flags
Output: $\phi_{\mathcal{K}}$: core candidate, ϕ: candidate, δ_N: check for necessary flag, δ_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 ϕ *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

Robust Reachability Examples

Example 1

```
void memcpy1(char* dst, const char* src, size_t n) {
    for (size_t i = 0; i < n; i += 1)
        dst[i] = src[i];
}
```

Example 2

```
void memcpy2(char* dst, const char* src, size_t n) {
    for (size_t i = 0; i <= (n >> 3); i += 1)
        (uint64_t*)dst[i] = (uint64_t*)src[i];
}
```

```
void main(const char* argv) {
    char buf[64];
    char var = random();
    memcpy(buf, argv, 64);
    assert(var != 0xee);
    return 0;
}
```

controlled ↓ uncontrolled ↓

$\exists$ argv $\forall$ var
overflow happens ?

No ↙ ↘ Yes

Motivation (1)

```c
void f(const char* input) {
    char buf[64] = {0};
    memcpy(buf, input, 64);
    buf[63] = 0;
    return
}

void g() { puts("hijacked"); }

void memcpy1(void* dst, const void* src, size_t n) {
    /* assume src[n - 1] == 0 */
    size_t i = 0;
    while (*((uint8_t*) src + i) != 0)
        *((uint8_t*) dst + i) = *((uint8_t*) src + i++);
}
```

Automatic Detection and Characterization of Bugs

Example: Buffer Overflow

- Overwriting the return address by copying too much data

Automatic Detection: Symbolic Execution

- Look for paths reaching g()
- Express paths as logical predicates
- Use an SMT-Solver to evaluate the feasibility of the path
- Conclusion: memcpy1 is buggy

Motivation (2)

```
void memcpy2(void* dst, const void* src, size_t n) {
    uint32_t canary = 0xdeadbeef;
    size_t i = 0;
    while (*((uint8_t*) src + i) != 0)
        *((uint8_t*) dst + i) = *((uint8_t*) src + i++);
    if (canary != 0xdeadbeef) exit(-2);
}
```

Possible counter-measure: Canary Implementation

- Unless the attacker knows the canary (or is lucky), prevents branching to g()

Symbolic Execution Evaluation

- memcpy2 is buggy
- No difference between memcpy1 and memcpy2

Robust Symbolic Execution [Girol et. al. 2020]

- Partition the input space: what an attacker controls x what an attacker doen’t control
- Build reaching controlled inputs that do not depend on the value of the uncontrolled memory
- $\exists\ c, controlled, \forall\ u,\ uncontrolled,\ reach\_g(c, u)$
- Conclusion:
  - memcpy1 is buggy
  - memcpy2 is not

Motivation (3)

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

void memcpy3(void* dst, const void* src, size_t n) {
    if ((((intptr_t) dst | (intptr_t) src | n) & 0b11111))
        /* slow path */
        for (size_t i = 0; i < n; i += 1)
            *((uint8_t*) dst + i) = *((uint8_t*) src + i);
    else /* fast path */
        for (size_t i = 0; i <= (n >> 5); i += 1)
            *((uint256_t*) dst + i) = *((uint256_t*) src + i);
}
```

Only one path is buggy (fast path)

- Overflow is only possible if both dst and src are aligned on 32bits
- This is not an unrealistic precondition

Problem with Robust Symbolic Execution

- memcpy3 is not buggy

We need a more precise bug characterization method to distinguish between such cases

- Our proposal: use assumptions on top of robust reachability
- $\exists c, \forall u, assumption(c,u) \Rightarrow \; reach\_g(c,u)$
- Example: $dst \;\%\; 32 = 0 \;\wedge src \;\%\; 32 = 0$
- Helps express that memcpy3 is more vulnerable than memcpy2

Contributions

1. **Abduction Procedure for Robust Assumptions of Program Trace Properties**
   A. Preliminaries
   B. Framework
   C. Baseline Algorithm
2. **Efficient Strategies for Abduction under Robustness**
   A. Necessary Constraints
   B. Counter-Examples
   C. Ordering Heuristic
3. **Experimental Evaluation and Application to Realistic Security Evaluation Scenarii**
   A. Implementation, Benchmark and Setup
   B. Generation Capabilities
   C. Influence of the Strategies
   D. Vulnerability Characterization

1. Abduction Procedure for Robust Assumptions of Program Trace Properties

Abduction of Robust Assumptions

Abductive Reasoning [Josephson and Josephson, 1994]

- Search for missing precondition to entail an unexplained goal
- Compute ϕ_M in $\phi_H \wedge \phi_M \vDash \phi_G$
- Efficient procedures exist for computation of theory-agnostic first-order solutions [Echenim et al. 2018, Reynolds et al. 2020]

Program-level Abduction

- Goal: Predicate on program traces ψ
- Search for a predicate on program inputs that entails ψ for the given program

Trace Predicate Oracles

- We assume the existence of oracles (tools)
  - Checking whether a trace predicate is satisfiable for a program under an assumption

$$\mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \phi)$$

  - Checking whether a trace predicate is robustly satisfiable for a program under an assumption and a given controlled/uncontrolled input partition

$$\mathcal{O}^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$$

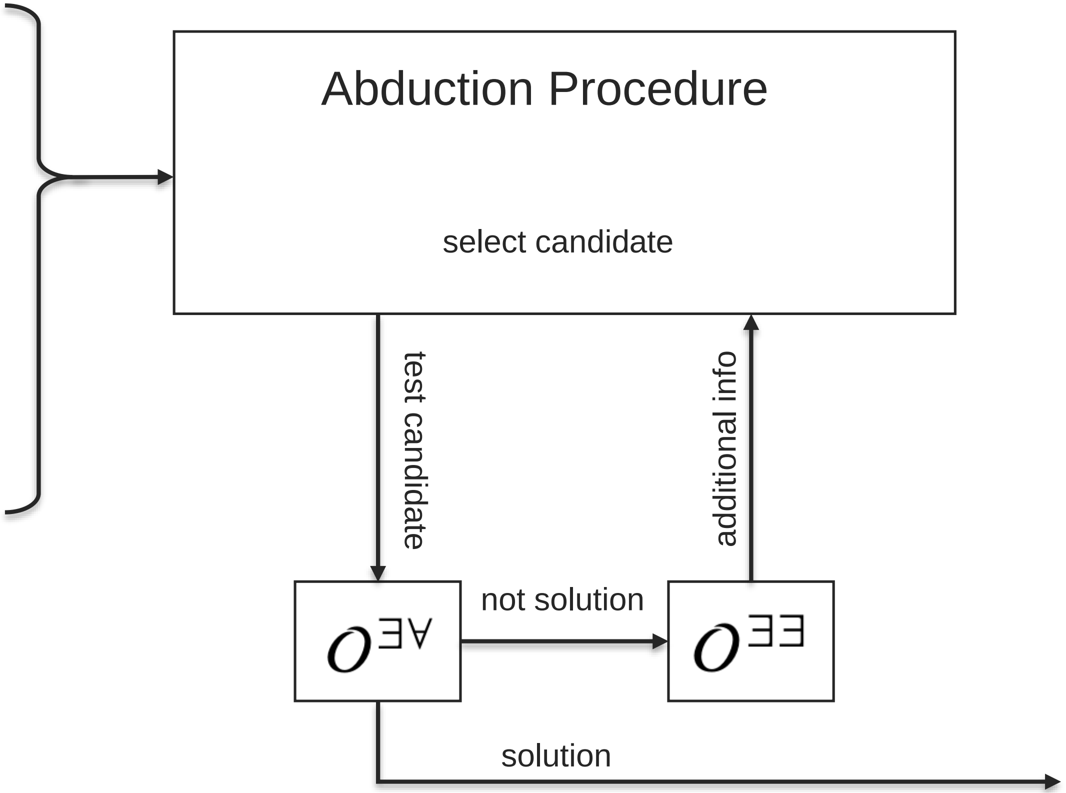
Framework

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

$\vec{P}$ Program

ψ Target Trace Predicate

$\mathcal{A}_C$ Memory Partition



robust reachability constraints

Baseline Algorithm

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

1 **if** $\top, s \leftarrow \mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \top)$ **then**
2 $\quad R \longleftarrow \{y = s\}$ **if** $y = s \in \mathcal{G}$ **else** $\emptyset$;
3 $\quad$ **for** $\phi \in \mathcal{G}$ **do**
4 $\quad\quad$ **if** $\mathcal{O}^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ **then**
5 $\quad\quad\quad R \longleftarrow \Delta_{min}(R \cup \{\phi\})$;
6 $\quad\quad\quad$ **if** $\neg\mathcal{O}^{\exists\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi' \in R} \phi'))$ **then**
7 $\quad\quad\quad\quad$ **return** R;
8 $\quad$ **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
- Under correction and completeness of the oracles
  - Minimality w.r.t. the inference language
  - Weakest constraint generation when returning from line 7

2. Efficient Strategies for Abduction under Robustness

Reusing Necessary Constraints

The Idea

- At each step, we can also check if the candidate is necessary for the satisfaction on the target property
- We can exploit such constraints to build a solution faster by constraining the seach space
  - We can skip candidates that are consequences of these necessary constraints
  - We can add to the candidates we test the conjunction of these necessary constraints

How it is done

- Additional $\mathcal{O}^{\exists\exists}$ query at each step
- For a candidate ϕ_K, try instead its conjunction with the maximal subset of necessary constraints that remain in the inference language, that is

$$\max_{\mathcal{G}} : \phi_{\mathcal{K}}, \mathcal{G}, N \mapsto \max_{\widehat{\lesssim}}\left(\left\{N_{\phi_{\mathcal{K}}} \subset N \,\middle|\, \phi_{\mathcal{K}} \wedge \bigwedge_{\phi \in N_{\phi_{\mathcal{K}}}} \phi \in \mathcal{G}\right\}\right)$$

$$\widehat{\lesssim} : N_1, N_2 \mapsto \textsc{Card}(N_1) < \textsc{Card}(N_2) \vee (\textsc{Card}(N_1) = \textsc{Card}(N_2) \wedge \bigwedge_{\phi \in N_1} \phi \prec \bigwedge_{\phi \in N_2} \phi)$$

Counter-Examples for Robustness

The Idea

- Similarly to what is done in the non-robust case, use counter-examples of the target property to prune the search space of candidates

The Issue

- Robustness (∃∀ quantification) prevents us from obtaining these examples from the verification oracle queries

Proposal

- Consider a second trace property, $\hat{\psi}$, the satisfaction of which ensuring that our target property ψ is not satisfied
- Additional $\mathcal{O}^{\exists\exists}$ query for this second property to obtain counter-examples
- Prune candidates that are satisfiable when they are evaluated to one of these counter-examples on the uncontrolled memory

Ordering Candidates with Heuristics

The Idea

- Some candidates are more likely to be solutions than others
- Reuse information obtained during the computation to make informed guesses on which candidates are better

What we already have

- Examples satisfying the constraint
- Syntactic information on the candidates

Proposal

- Use an ordering heuristic on the candidates

$$\phi \mapsto (count(\wedge, \phi), -\textsc{Card}(\{s \in V \mid \phi[s]\}))$$

- A few examples can be recovered at the begining of the execution
- Resorting may be applied during the execution to take into account newly obtained examples

3. Experimental Evaluation and Application to Realistic Security Evaluation Scenarii

Implementation

Restriction

- Restriction to bounded reachability and robust reachability
- Use of Symbolic Execution and Robust Symbolic Execution as Oracles
  - BINSEC [Djoudi and Bardin 2015]
  - Symbolic execution at binary level
- Use of SMT-Solver for formula handling and evaluation (z3, cvc5)

Prototype

- PyAbd, Python implementation of the abduction procedure
- Takes as input the binary program and the target reachability target as a BINSEC configuration
- Outputs the robust reachability constraints it found

Benchmark: SVComp

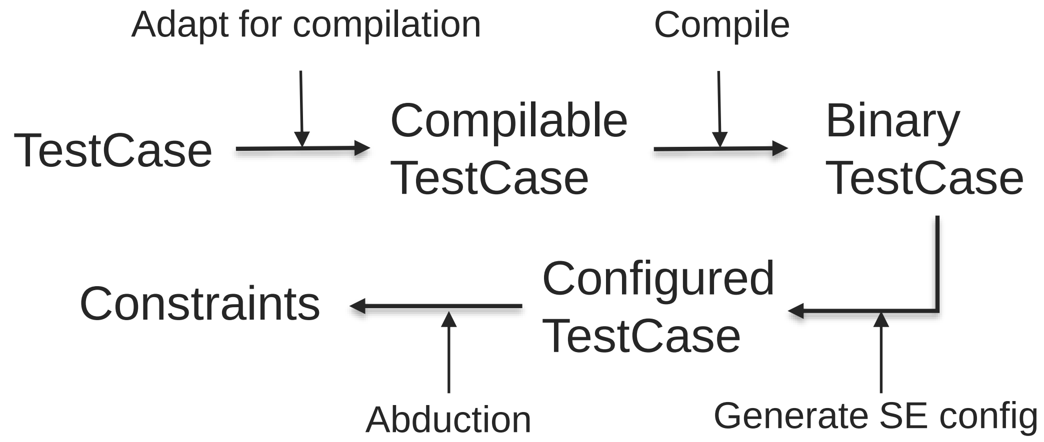
SVComp

- Collection of software verification tasks
- Evaluate the generative capabilities of our method
- Select 147 reachable reachability test cases for which BINSEC answers under 1 minute

Setup

- Use the target location of the benchmark
- Use the input variables as program input
- Abduce constraints on these variables to reach the target location

Process



Example

```c
void init (int* x)
{
        for (int i = 0; i < 4; i++)
                x[i] = __VERIFIER_nondet_int();
}

int verify(int * x)
{
        int stuff[4] = { 1, 2, 3, 4 };
        for (int i = 0; i < 4; i++)
                if (x[i] != stuff[i])
                        return 1;
        return 0;
}
```

```c
int main ()
{
  int x[4];

  init(x);
  assert(verify(x) != 0);

  return 0;
}
```

Benchmark: FISSC

FISSC VerifyPINs

- Collection of verifyPIN C implementations, protected against fault-injection attack
- Reachability: location of incorrect auth

Setup

- Compile source to initial binary
- Simulate 1 instruction skip fault injection by program mutation
- Select 719 reachable mutant programs
- Look for constraints on PIN inputs that lead to an authentication with a wrong PIN

Example

```c
#ifdef LAZART
inline BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2) __attribute__((always_inline))
#else
BOOL NOINLINE_BAC byteArrayCompare(UBYTE* a1, UBYTE* a2)
#endif
{
    int i = 0;
    BOOL status = BOOL_FALSE;
    BOOL diff = BOOL_FALSE;
    for(i = 0; i < PIN_SIZE; i++)
        if(a1[i] != a2[i]) diff = BOOL_TRUE;
    if((i == PIN_SIZE) && (diff == BOOL_FALSE)){
      //__begin__secure__("stepCounter");
      status = BOOL_TRUE;
      //__end__secure__("stepCounter");
    }
    return status;
}

void verifyPIN_A()
{
    g_authenticated = BOOL_FALSE;

    if(g_ptc > 0) {
        if(byteArrayCompare(g_userPin, g_cardPin) == BOOL_TRUE) {
success:
            //__begin__secure__("stepCounter");
            g_ptc = g_ptc_INIT;
            g_authenticated = BOOL_TRUE; // Authentication();
            //__end__secure__("stepCounter");
        }
        else {
            g_ptc--;
        }
    }
}
```

Experimental Setup

What we have

- Program: SVComp or FISSC
- Vulnerability property: Reachability
- Tool: BINSEC
- Tool for robustness: BINSEC-RSE
- Inference Language:
- Implementation: PyAbd

What we do

- Check if PyAbd is able to generate a constraint

Setup

- 1 hour timeout per program
- 60 seconds timeout for BINSEC queries
- 10 seconds timeout for SMT queries

- Run in parallel on an Intel® Xeon® Gold 5220 CPU @2.20GHz machine with 36 cores, 72 threads and 96 GB of RAM

Results: Generating Constraints

Table 1. PyAbd analysis results for each benchmark configuration, with (■) or without (□) controlled variables. We detail the total number of programs analyzed, the number of programs for which the target location is robustly k-reachable, and the number of programs for which PyAbd finds sufficient robust k-reachability constraints (rrc), split between any sufficient rrc, and the weakest rrc constraints only.

	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

Conclusion: We can compute non-trivial sufficient constraints that refine the results of existing tools

Results: Influence of the 'Efficient Strategies'

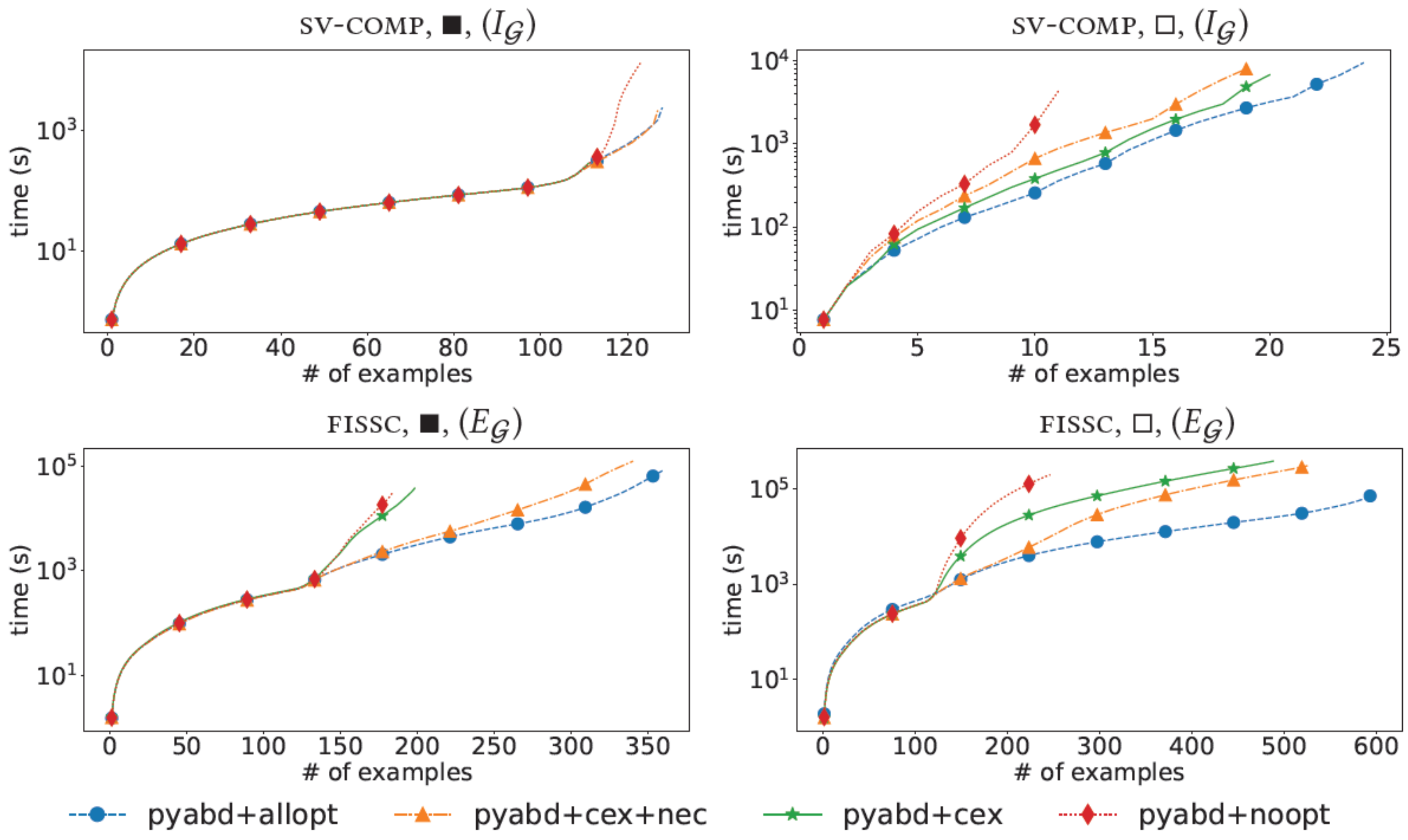


Fig. 5. Cactus plot showing the influence of the strategies of Section 5 on the computation of the first sufficient k-reachability constraint with PyAbd.

- The strategies significantly improve the capabilities of the tool
- Each strategy impacts a different part of the algorithm
- Some strategies lead to increased computation time on simple examples but permit to solve more difficult ones

Results: Vulnerability Characterization

Table 3. Analysis results for the VerifyPIN fault mutants (FISSC). Top group of rows: overall results. Bottom group: number of vulnerable mutants for which at least a given percentage of tuples (userPIN / cardPIN) lead to an incorrect authentication. Analysis methods considered: PyAbd^{O} on PIN values ; PyAbd^{P} on PIN values plus additional constraints on other input variables; Binsec; Binsec/RSE; single simulation with Qemu; and exhaustive simulation Qemu+L. Each analysis method considers a total of 4810 mutant programs.

	PyAbd^{O}	PyAbd^{P}	Binsec/RSE	Binsec	Qemu	Qemu+L
unknown	170	170	273	170	243	284
not vulnerable (0 input)	4414	4042	4419	3921	4398	4220
vulnerable ($\geq$ 1 input)	226	598	118	719	169	306
$\geq$ 0.0001%	226	598	118	–	–	306
$\geq$ 0.01%	209	582	118	–	–	281
$\geq$ 0.1%	173	514	118	–	–	210
$\geq$ 1.0%	167	472	118	–	–	199
$\geq$ 5.0%	166	471	118	–	–	196
$\geq$ 10.0%	118	401	118	–	–	148
$\geq$ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

We characterize more cases than other tools and with a more precise solution

Results: Example of FISSC Constraints

- 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

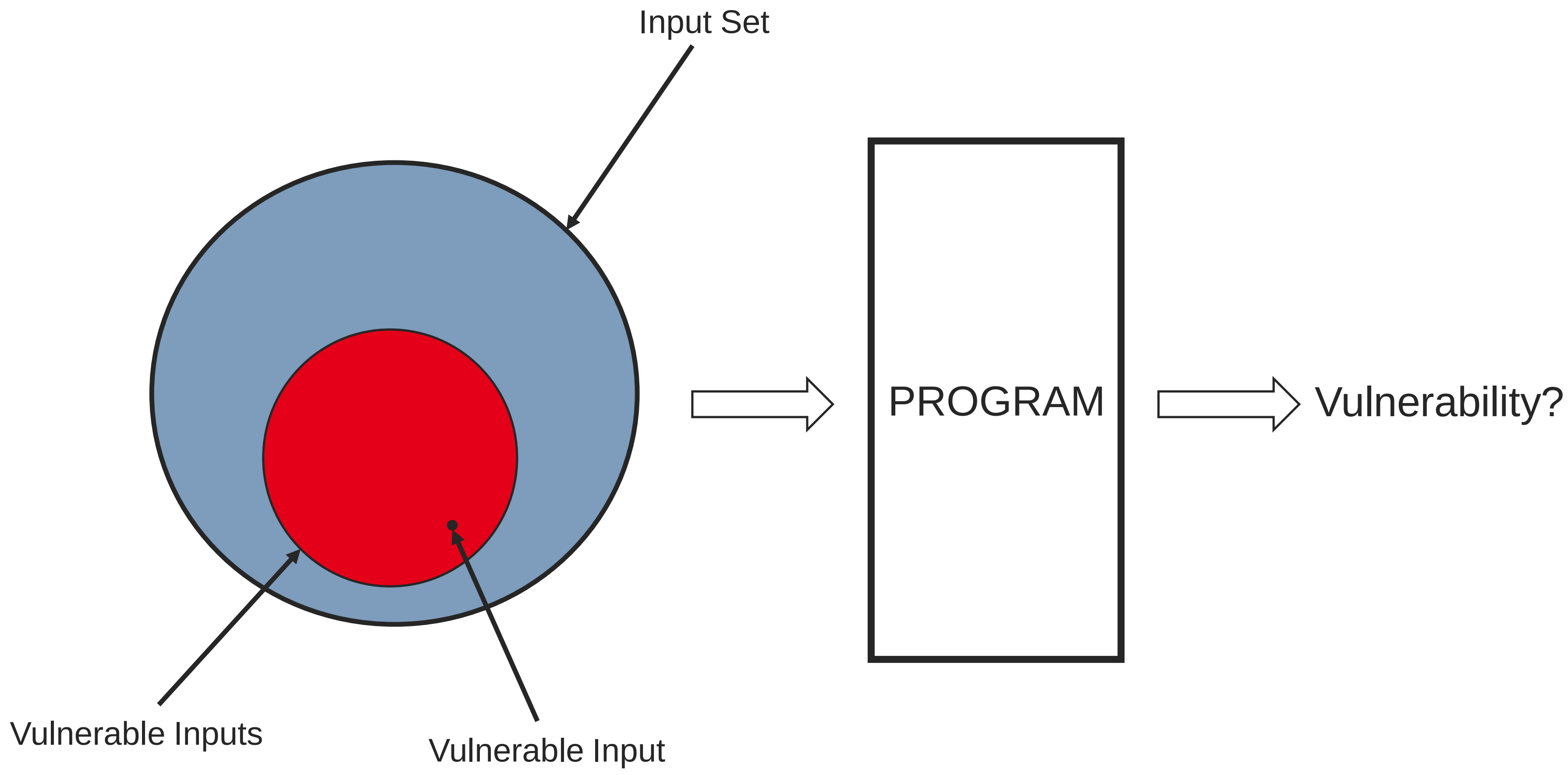
Conclusion

Conclusion

- We propose a novel abduction-based approach to automatically infer robust reachability constraints
- We implement a restriction of this approach to reachability properties on top the BINSEC robust symbolic execution engine
- We demonstrate its ability to refine the notion of robust reachability on standards benchmarks from software verification and security analysis

Questions

Recap: Formal Characterization



- **Usual Formal Evaluation (SE, BMC)**: Returns 1 *Vulnerable Input* if it exists
- **Robust Formal Evaluation (RSE, RBMC)**: Returns wether *Input Set* = *Vulnerable Inputs* or not
- **Formal Characterization**: Returns a logical description of *Vulnerable Inputs*

Possible Solution: Simulation

Simulation

- From a given set of possible inputs
- Execute/Simulate the program on each input
- Check if the input leads to the targeted bug

Advantages

- Very fast

Issues

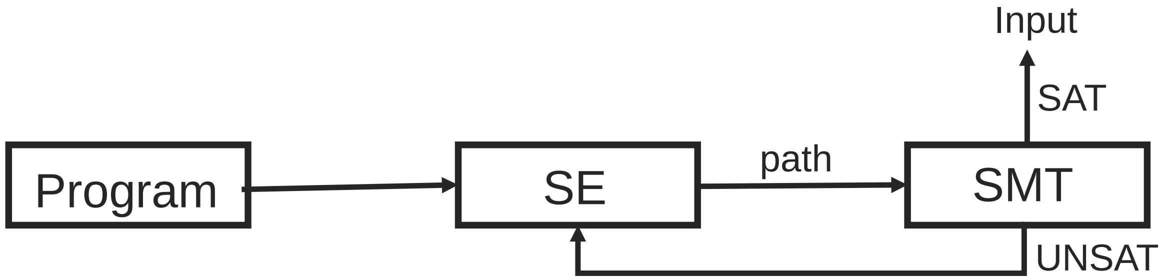
- Depends on the set of inputs
- Easily miss corner cases
- No guarantee if nothing is found

Exhaustive Simulation

- Ensures no case is missed
- Prohibitively time consuming
- Results are hard to exploit

SE Example

- Define a Target Location in a program *l*
- Express program execution as logic constraints
  - One formula for each possible path containing *l*
- Let program inputs be free variables
- Use a logic constraints solver (SMT-Solver) to look for assignments of free variables satisfying the reachability predicate



Algorithm 1: $VerifyPin(user, card)$

Input: $user$: user input, $card$: card pin
Output: $status$: authentication iff true

$status \leftarrow \bot$;
$diff \leftarrow \bot$;
for $i = 0; i < 4; i++$ **do**
  if $user[i] \neq card[i]$ **then**
    $diff \leftarrow \top$;
if $i = 4 \land \neg diff$ **then**
  $status \leftarrow \top$;
return $status$;

Target AND user != card

Algorithm 2: $VerifyPinSMTConstraints$

Input: `(declare-var user)`, `(= card card-value)`
Output: `SAT(user)/UNSAT`

```
(= status_0 false);
(= diff_0 false);
(= i_0 0);
(= user[i_0] card[i_0]);
(= i_1 (+ i_0 1));
(= user[i_1] card[i_1]);
(= i_2 (+ i_1 1));
(= user[i_2] card[i_2]);
(= i_3 (+ i_2 1));
(distinct user[i_1] card[i_1]);
(= diff_1 true);
(= i_4 (+ i_3 1));
(and (= i_4 4) (not diff_1));
(distinct (user card));
```

Inference Languages: Variables

Symbolic Execution Recovery

- Vulnerability examples can be retrieved from the Symbolic Execution Queries
- In these examples, most variables are set to default value, because they are unconstrained: We can discard these variables
- We add to our set of variables all the others and the values they were assigned
- We build constraints following the previous grammar with these variables

Insight

- The set of variable, hence the inference language, depends on the evaluated program
- Future Sybolic Execution queries can exhibit new variables, the language can be updated accordingly

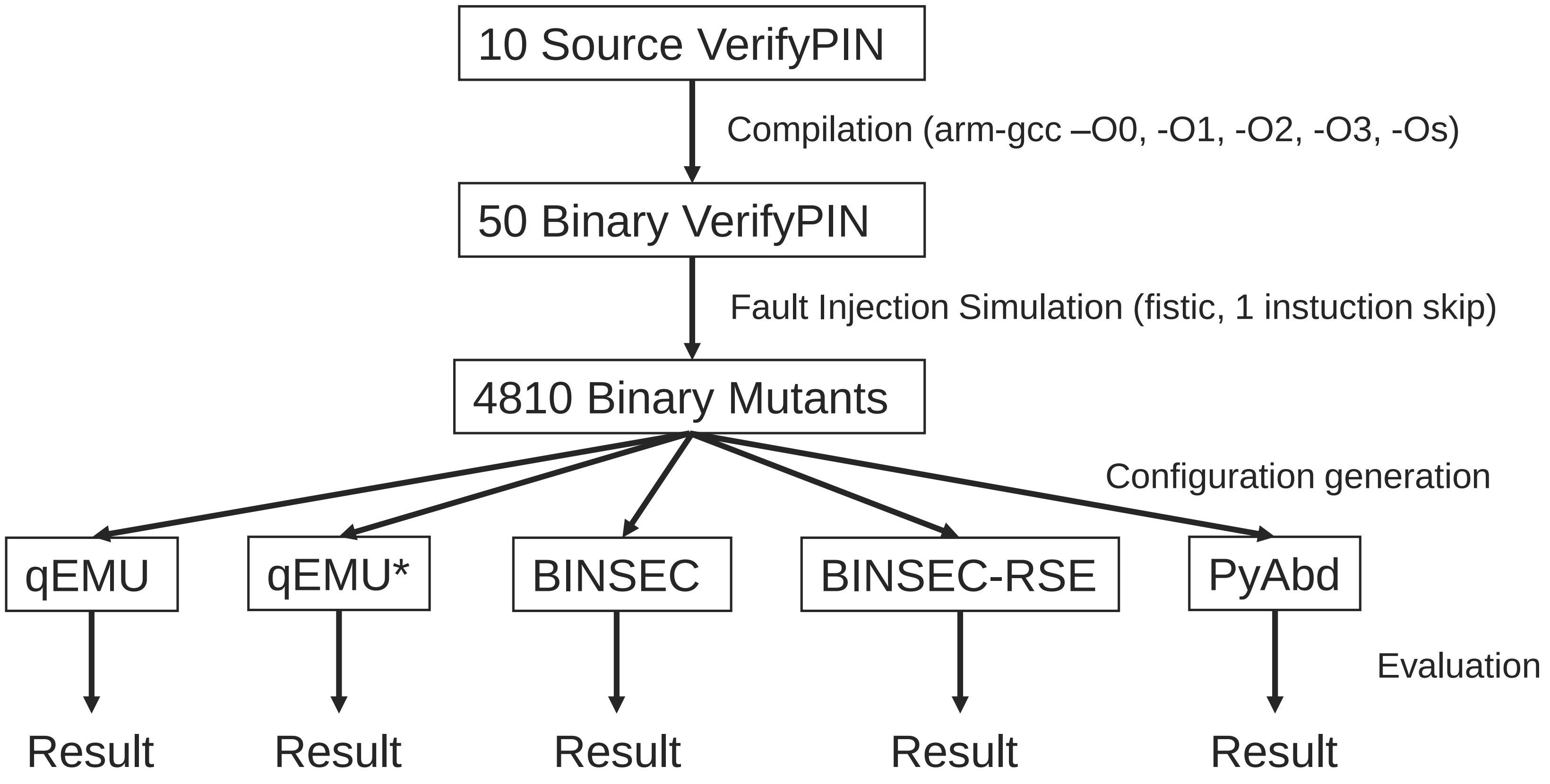
Research Questions

For our benchmarks

- Can we characterize the vulnerabilities?
- Can we generate complete characterizations?
- Do our optimization strategies improve the efficiency of the algorithm?
- How efficiently do we explore the inference language?

Instruction Skip on the FISSC VerifyPINs

Evaluation



Comparing Tools

Simulation

- We find one input that triggers the vulnerability
- Two variants:
  - Predefined test input (Qemu)
  - Loop over all possible userPIN/cardPIN values (Qemu+L)

Symbolic Execution

- The engine finds a vulnerable input
- Two variants:
  - No robustness (Binsec)
  - Robustness (Binsec/RSE)

Abduction

- Generate constraints on PIN variables and other input variables
- Two variants:
  - Constraints on non-PIN variables assumed True
  - Constraints on non-PIN variables assumed False

Comparing Tools: Severity Evaluation

- Count the number of mutant programs for which a vulnerability is found
- Count the number of couples userPIN/cardPIN for which the vulnerability is known to be triggered
- Sort the results with the assumption that more vulnerable userPIN/cardPIN couples means a more serious vulnerability
- Compare the results of the different methods

Instruction Skip Vulnerabilities of the FISSC VerifyPIN: Analysis Time

Table 4. Analysis times (hours:minutes:seconds) for VerifyPIN (FISSC) for the analysis methods considered in Table 3. For PyAbd$^{O/P}$, we report the complete analysis time (PyAbd$^{O/P}$), the time for returning the first constraint (PyAbd$^{O/P}_{first}$), and the time for returning the last constraint (PyAbd$^{O/P}_{last}$, *i.e.* timeouts excluded).

	PyAbd$^{O/P}$	PyAbd$^{O/P}_{first}$	PyAbd$^{O/P}_{last}$	Binsec/RSE	Binsec	Qemu	Qemu+l
average	0:16:57	0:01:53	0:02:45	0:00:13	0:00:04	0:00:01	1:08:43
median	0:01:25	0:00:46	0:00:46	0:00:06	0:00:03	0:00:01	1:11:38