Fault Injection Vulnerability Characterization by Inference of Robust Reachability Constraints

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

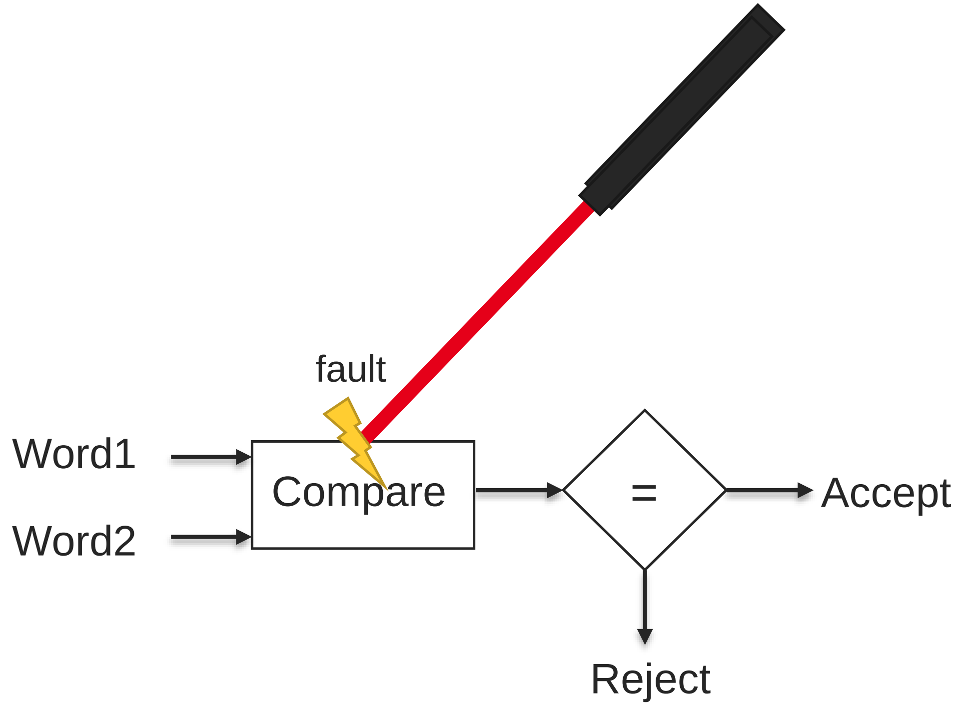
[1] Univ. Grenoble Alpes, CEA List, France

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

Fault Injection Attacks

Fault Injection Attacks

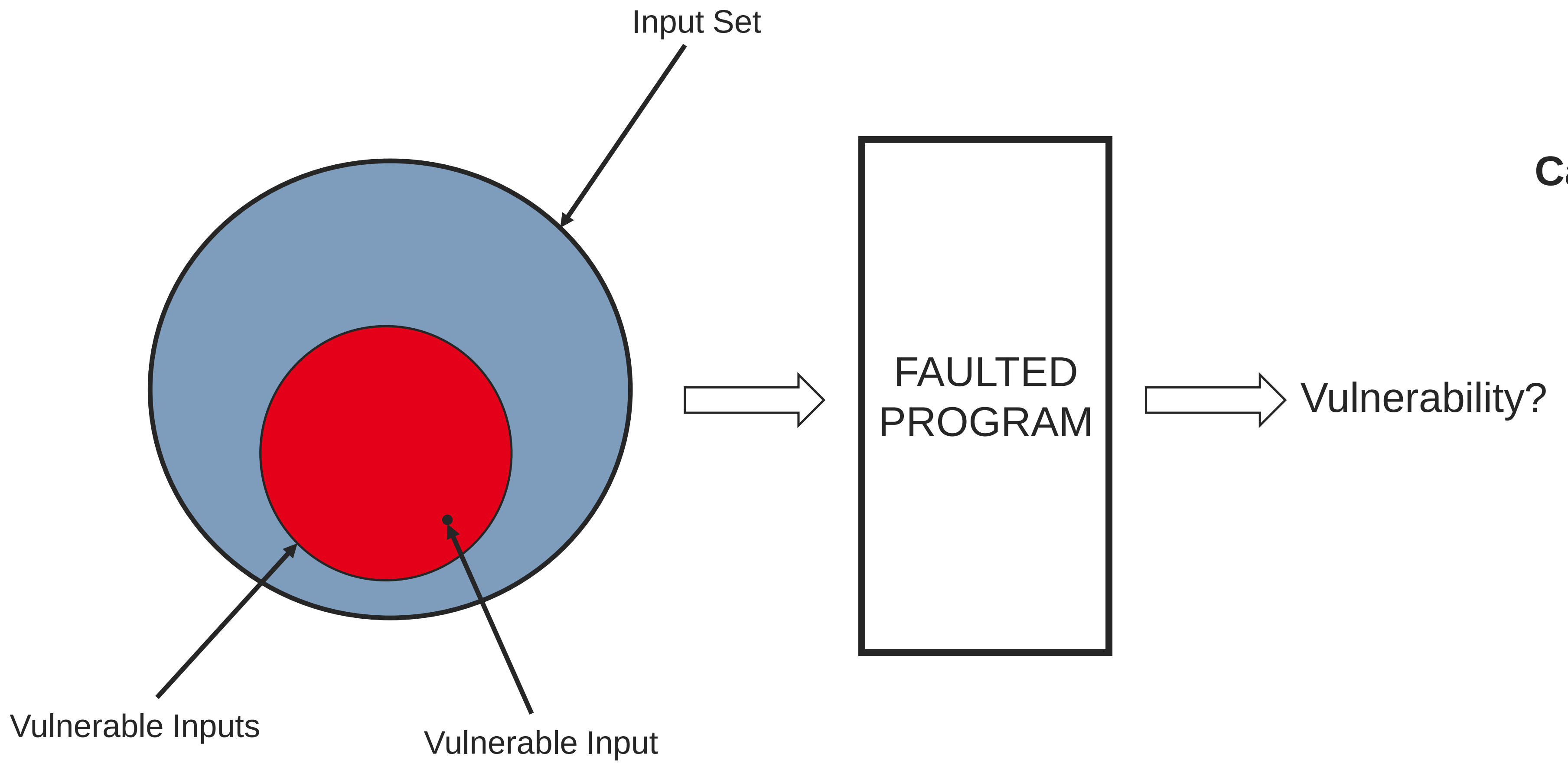
- Apparently safe program
- Physical perturbation of the system
- Changes the program behavior →Vulnerability
- **Goal**: Detect these vulnerabilities



Examples

- Power glitches, clock glitches
- Laser perturbation
- EM pulse

Vulnerability Detection



Can we find a vulnerable input?

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

Extended Simulation / Fuzzing

- Improves coverage
- Important time consumption
- Results may be hard to exploit

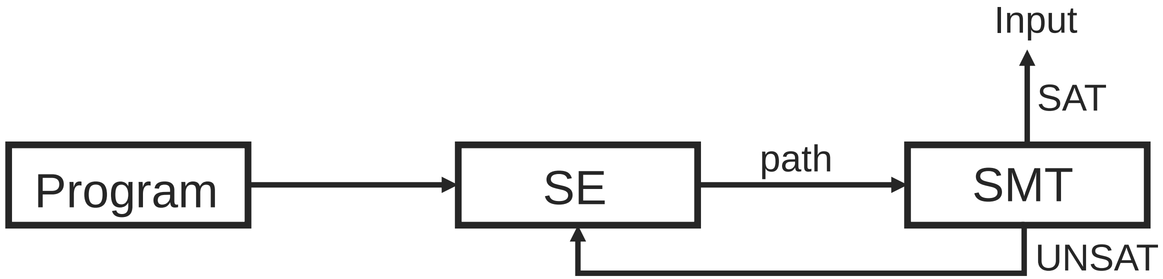
The Issue

Fault Injection may lead to vulnerabilities that depend on the input state

- Cannot be reliably triggered with program execution
- No information when no vulnerability is found
- A reported vulnerability may have been caused by (bad) luck

Possible Solution: Symbolic Execution

- 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 ← ⊥;
diff ← ⊥;
for i = 0; i < 4; i++ do
    if user[i] ≠ card[i] then
        diff ← ⊤;
if i = 4 ∧ ¬diff then
    status ← ⊤;          ← Target AND user != card
return status;
```

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));
```

Symbolic Execution

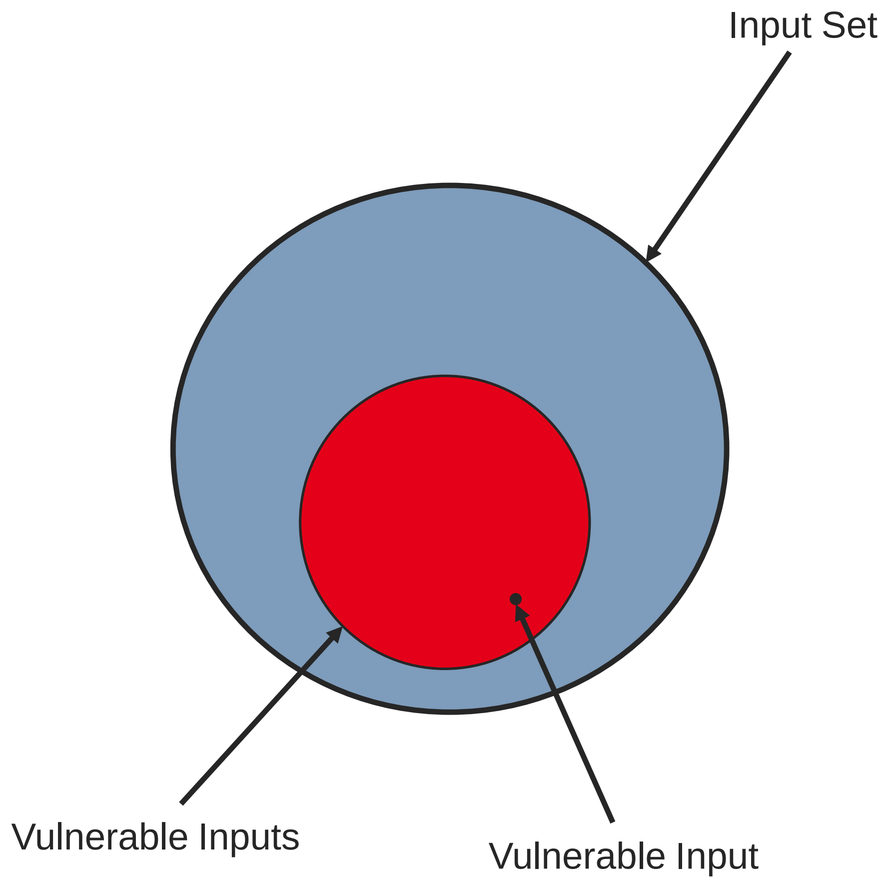
Advantages

- The complete input state is evaluated
- No false positives
- Complete for bounded verification

Issues

- Reported vulnerabilities may be infeasible in practice
- Usually reports a lot of vulnerabilities

Main Problem



We report a vulnerability on **one** vulnerable input only

This says nothing on **other possible vulnerable inputs** or on the ability to produce this input

We need an automated method to **characterize the set of vulnerable inputs**

Robust Reachability
[Girol, Farinier, Bardin: CAV 2021]

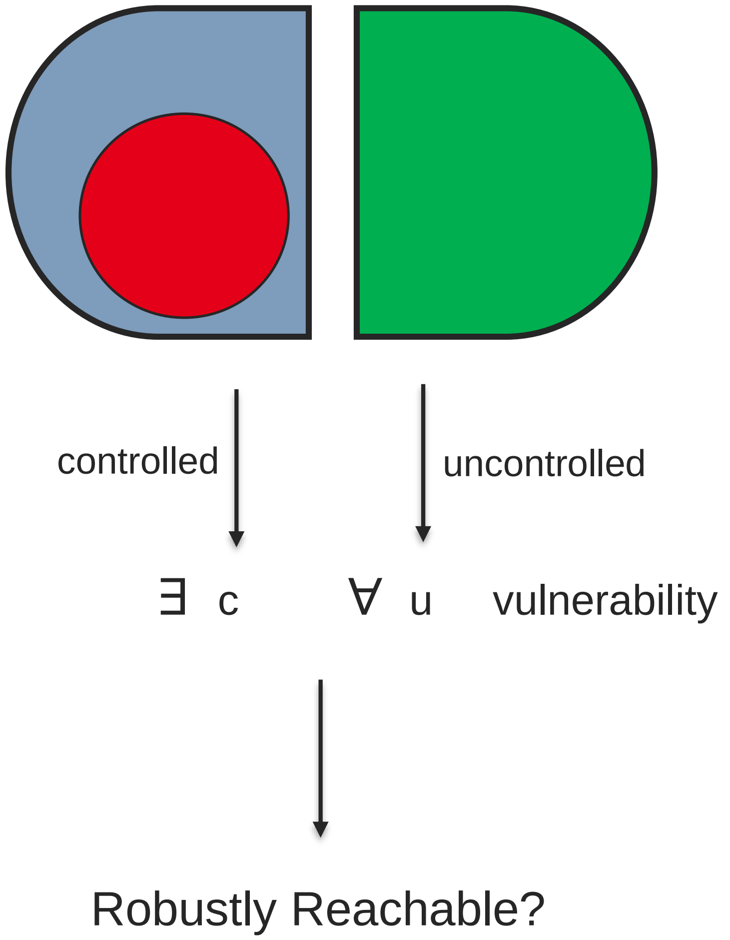
Idea

- 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

Extension of Reachability and Symbolic Execution



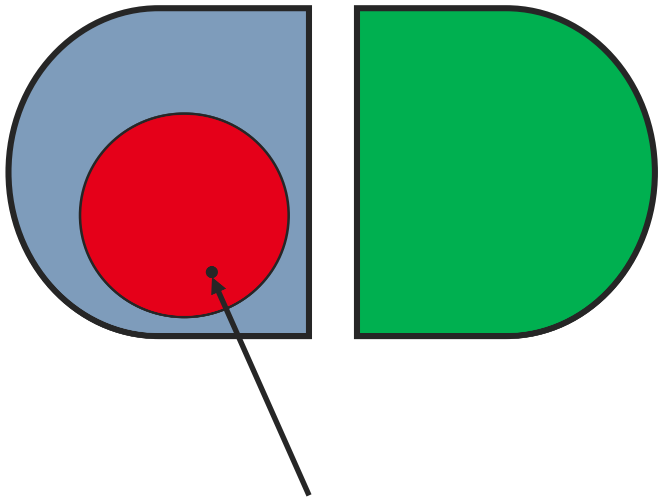
Remaining Problem

Robust Reachability is Too Strong

- May miss vulnerabilities that happen always except in a few corner cases

The problem is unchanged for controlled variables

- We only generate one controlled input for which
  - The vulnerability is replicable
  - We cannot conclude for other inputs



Proposal: Robust Reachability Constraints

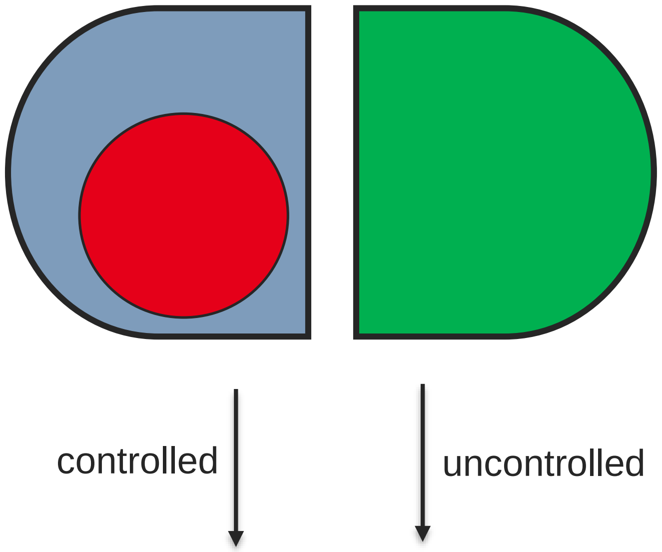
Definition

- Predicate **P** on program input sufficient to have Robust Reachability

Advantages

- Part of the Robust Reachability framework
- Allows precise characterization

How to Automatically Generate Such Constraints?



$\exists\ c \quad \forall\ u \quad P(c, u) \implies$ vulnerability

Robustly Reachable?

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

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

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 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

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 $\phi_H \wedge \phi_M \vDash \phi_G$

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

Theory-Agnostic First-order Abduction

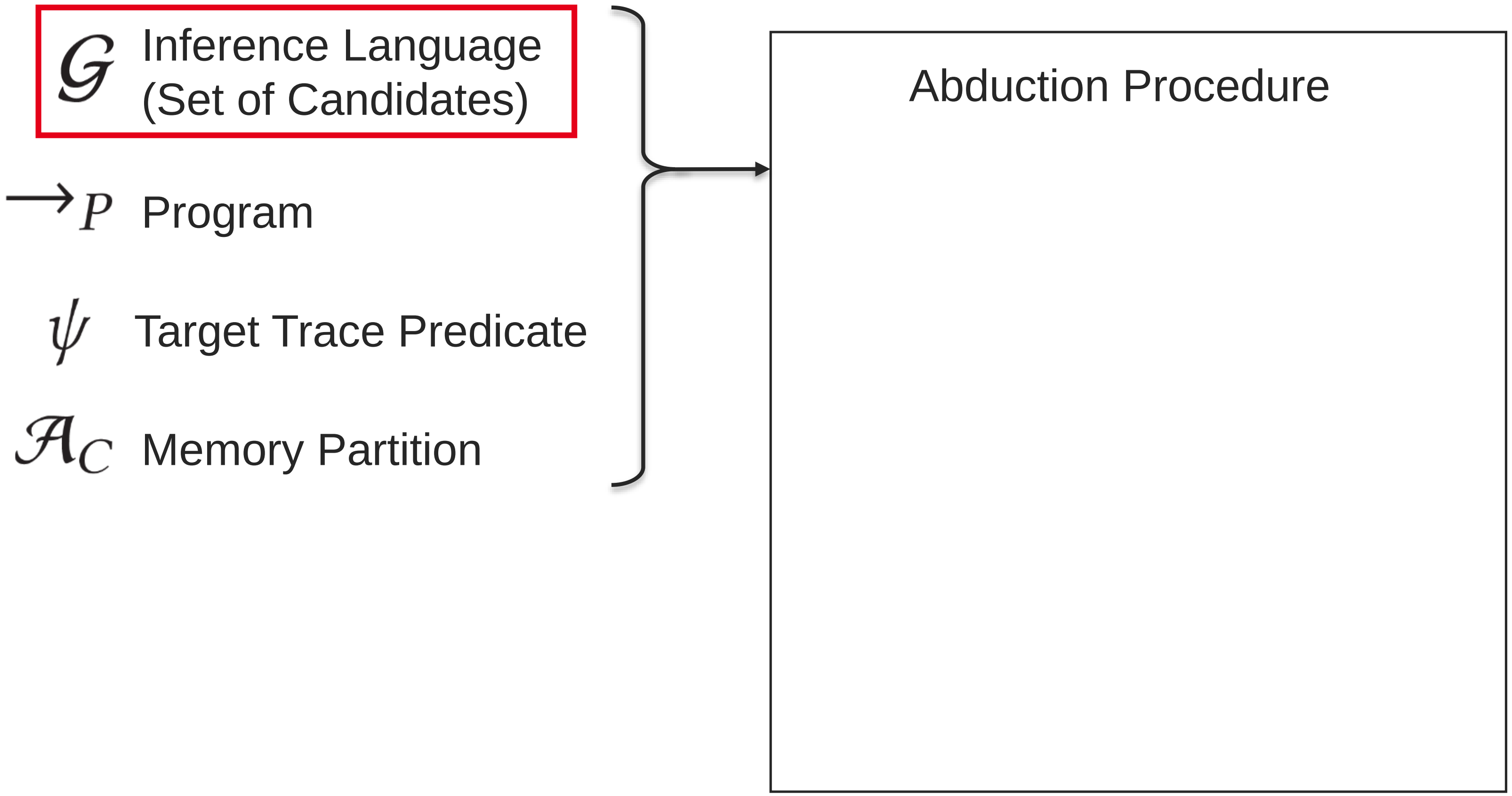
[Echenim et al. 2018, Reynolds et al. 2020]

- Efficient procedures
- Genericity

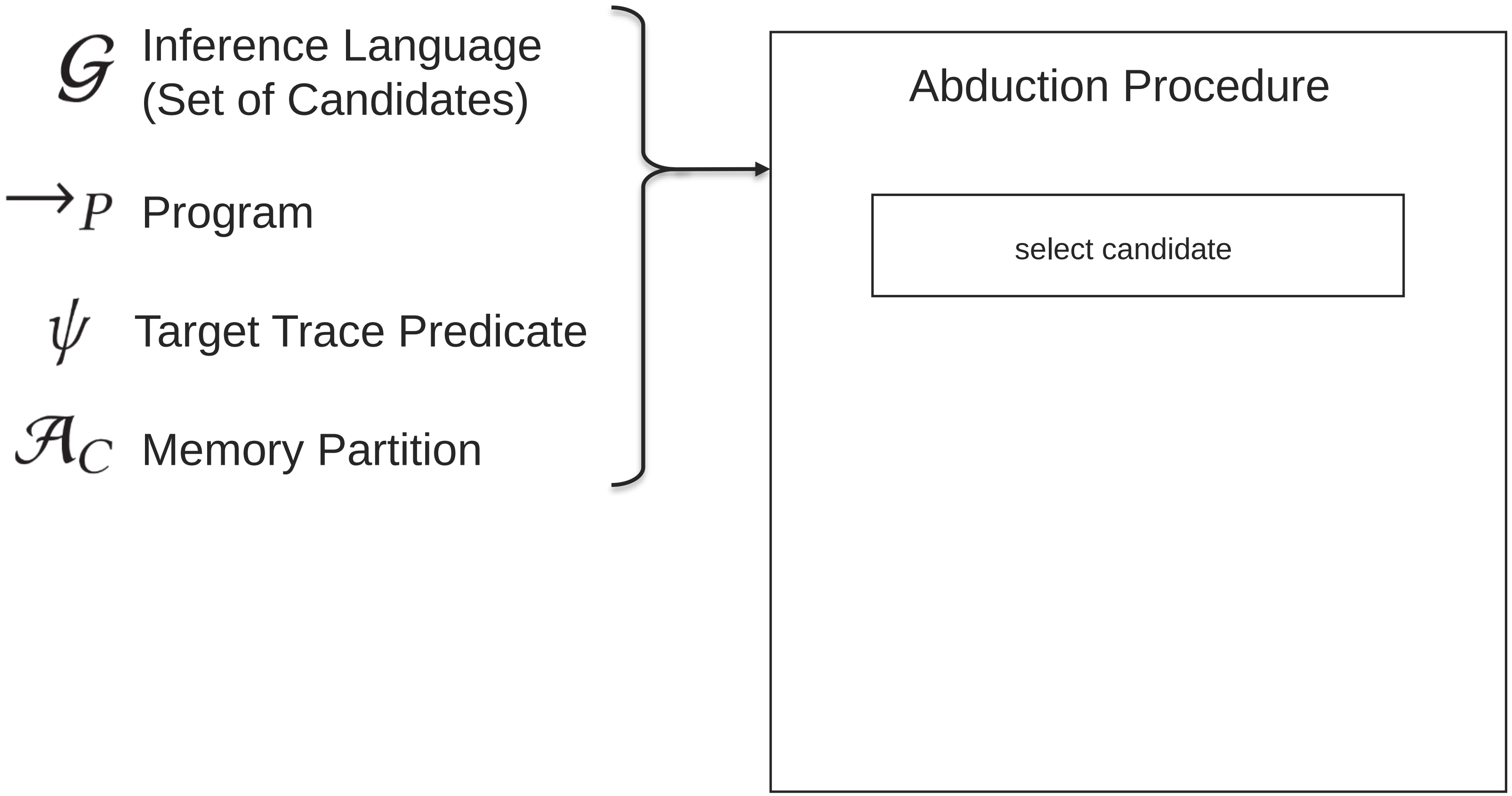
Our Proposal: Adapt Theory-Agnostic Abduction Algorithm to Compute Program-level Robust Reachability Constraints

- Program-level
- Generic

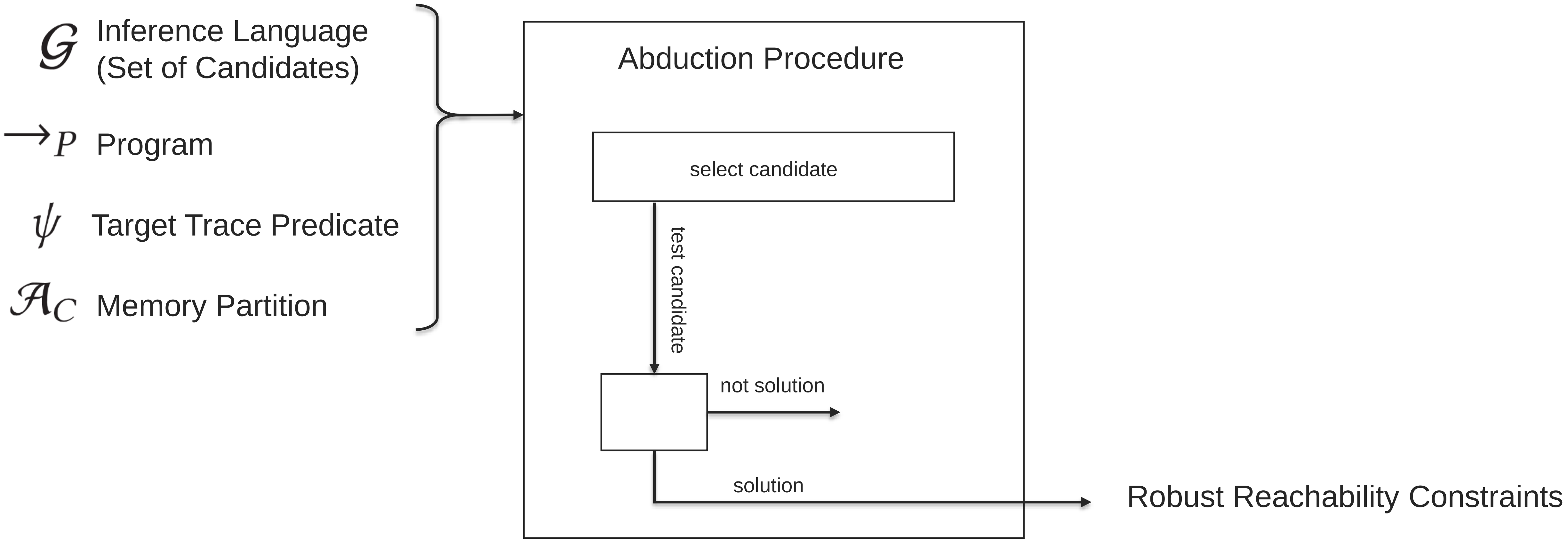
Our Solution (Framework)



Our Solution (Framework)

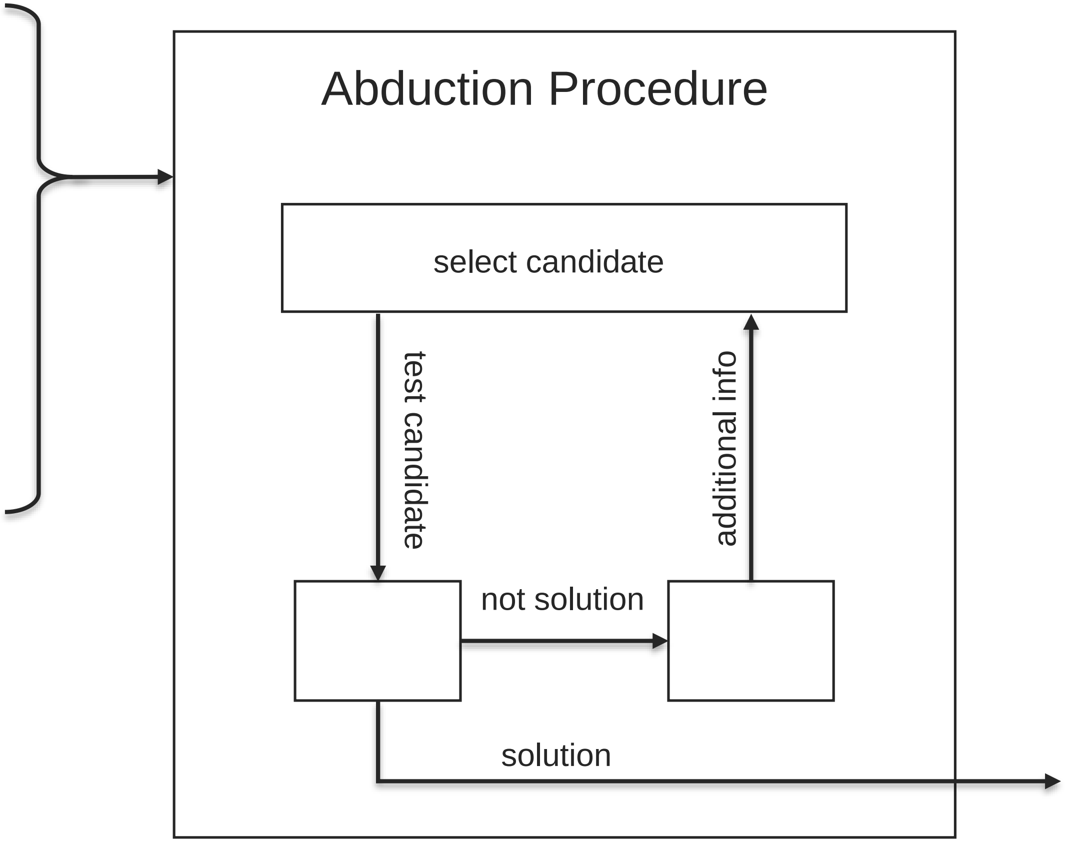


Our Solution (Framework)



Our Solution (Framework)

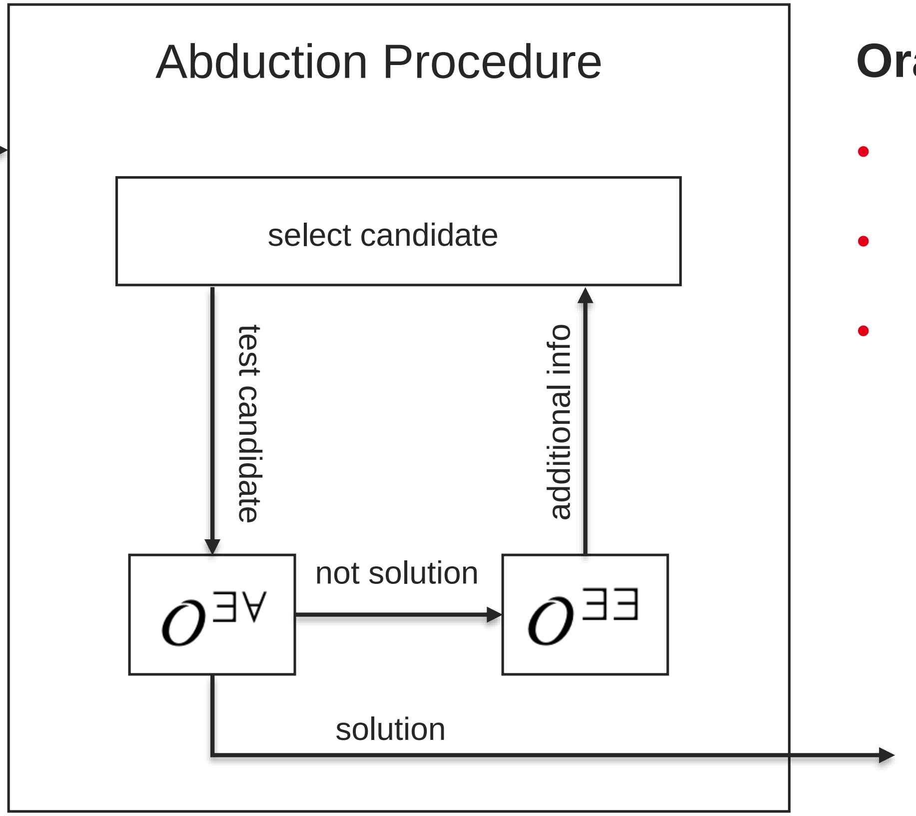
- $\mathcal{G}$ Inference Language (Set of Candidates)
- $\overrightarrow{P}$ Program
- ψ Target Trace Predicate
- $\mathcal{A}_C$ Memory Partition



Robust Reachability Constraints

Our Solution (Framework)

- $\mathcal{G}$ Inference Language (Set of Candidates)
- $\rightarrow P$ 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, …)

Theoretical Results

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 ⊤, s ← O^∃∃(→_P, ψ, ⊤) then                                  // ensure ψ satisfiable
    V ⟵ {s};                                  // init satisfying memory states examples
    R, N, U ⟵ {y = s} if y = s ∈ G else ∅, {⊤}, {⊥};                  // init result sets
    while φ_K, φ, δ_N, δ_R ⟵ NextRC(G, →_P, ψ, ψ̂, A_C, V, R, N, U, prunef) do   // explore G
        if δ_R and ⊤, s ← O^∃∃(→_P, ψ, φ) then           // ensure ψ satisfiable under φ
            V ⟵ V ∪ {s};                                          // new trace example
            if O^∃∀(→_P, A_C, ψ, φ) then                            // check candidate φ
                R ⟵ Δ_min(R ∪ {φ});                          // update and minimize R
                if ¬O^∃∃(→_P, ψ, ¬(⋁_{φ∈R} φ)) then                     // check weakest
                    return (R, {⋁_{φ'∈R} φ'}, U);
            else
                U ⟵ U ∪ {φ};                                // new breaking constraint
        else if δ_R then
            N ⟵ N ∪ {¬φ};                                  // new necessary constraint
        if δ_N and ¬O^∃∃(→_P, ψ, ¬φ_K) then
            N ⟵ N ∪ {φ_K};                                 // new necessary constraint
    return (R, N, U);
return ({⊥}, {⊥}, {⊥});
```

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

```
V̄ ⟵ ∅;                                                      // init. counter-examples
for φ_K ∈ browse(G, V) if prunef.browse else G do              // get candidate from G
    φ ⟵ φ_K ∧ ⋀_{φ'∈max_G(φ_K,G,N)} φ' if prunef.nec else φ_K;  // add nec. constraints
    if φ is unsatisfiable then
        continue;                                               // skip: inconsistent
    if prunef.cex and ∃m, X ∈ V̄, φ ∧ y|_X = m is satisfiable then
        continue;                                    // skip: sat. by counter-example
    if ∃φ_s ∈ R, φ ⊨ φ_s then
        continue;                      // skip: stronger than known suff. constraint
    if prunef.nec and ∃φ_u ∈ U, φ_u ⊨ φ then
        continue;                      // skip: weaker than known break. constraint
    if prunef.nec and (⋀_{φ_n∈N} φ_n) ⊨ φ then
        continue;                        // skip: weaker than known nec. constraint
    if prunef.cex and ⊤, cex ⟵ O^∃∀(→_P, X, ψ̂, φ) for X ⊆ A \ A_C then
        V̄ ⟵ V̄ ∪ {cex}, X;                                       // new counter-example
        yield φ_K, φ, prunef.nec, ⊥;                             // forward for nec. check
    else
        yield φ_K, φ, prunef.nec, ⊤;                   // forward for nec. and suff. checks
```

Theorem

- Termination
- Correction
- Completeness (wrt Oracle)
- Minimality (wrt Inference Language)
- Weakest constraint generation if possible

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: Characterizing Fault Injection Attacks Vulnerabilities

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

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

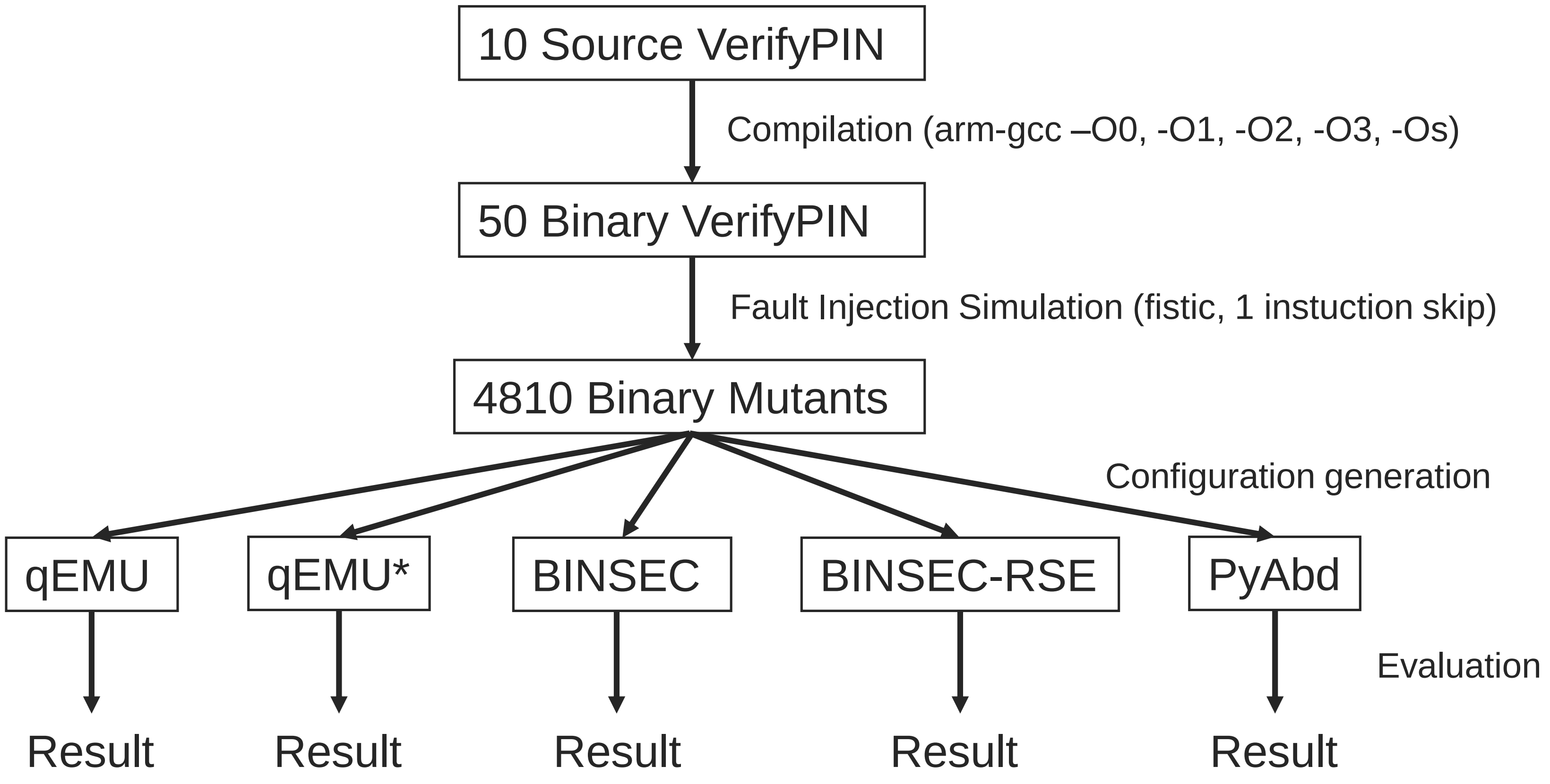
```
#ifdef LAZART
inline BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2) __attribute__((always_inline))
#else
BOOL NOINLINE_BAC byteArrayCompare(UBYTE* a1, UBYTE* a2)
#endif
{
    int i = 0;
    BOOL status = BOOL_FALSE;
    BOOL diff = BOOL_FALSE;
    for(i = 0; i < PIN_SIZE; i++)
        if(a1[i] != a2[i]) 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--;
        }
    }
}
```

Instruction Skip on the FISSC VerifyPINs

Evaluation



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

$*a_8 \leq *a'_8$

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

$*a_8 \leq v_8$

$\neg \langle nliteral \rangle$

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

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 □

Results: Generating Constraints

	FISSC ($E_{\mathcal{G}}$)		FISSC ($I_{\mathcal{G}}$)	
	■	□	■	□
# programs	719	719	719	719
# of robust cases	129	118	129	118
# of sufficient rrc	359	598	351	589
# of weakest rrc	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 vulnerabilities than Robust Symbolic Execution

Results: Characterization

	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 (≥ 1 input)	226	598	118	719	169	306
≥ 0.0001%	226	598	118	–	–	306
≥ 0.01%	209	582	118	–	–	281
≥ 0.1%	173	514	118	–	–	210
≥ 1.0%	167	472	118	–	–	199
≥ 5.0%	166	471	118	–	–	196
≥ 10.0%	118	401	118	–	–	148
≥ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

Non-PIN input state
is **not** satisfied

Non-PIN input state
is satisfied

Results: Characterization

	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 (≥ 1 input)	226	598	118	719	169	306
≥ 0.0001%	226	598	118	–	–	306
≥ 0.01%	209	582	118	–	–	281
≥ 0.1%	173	514	118	–	–	210
≥ 1.0%	167	472	118	–	–	199
≥ 5.0%	166	471	118	–	–	196
≥ 10.0%	118	401	118	–	–	148
≥ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

Many reported vulnerabilities

Non-PIN input state is **not** satisfied

Non-PIN input state is satisfied

Results: Characterization

	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

Many reported vulnerabilities

Non-PIN input state is **not** satisfied

Non-PIN input state is satisfied

No conclusion on more than one input

Results: Characterization

	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 (≥ 1 input)	226	598	118	719	169	306
≥ 0.0001%	226	598	118	–	–	306
≥ 0.01%	209	582	118	–	–	281
≥ 0.1%	173	514	118	–	–	210
≥ 1.0%	167	472	118	–	–	199
≥ 5.0%	166	471	118	–	–	196
≥ 10.0%	118	401	118	–	–	148
≥ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

Many reported vulnerabilities

Non-PIN input state is **not** satisfied

Non-PIN input state is satisfied

No details for less than all inputs

No conclusion on more than one input

Results: Characterization

	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 (≥ 1 input)	226	598	118	719	169	306
≥ 0.0001%	226	598	118	–	–	306
≥ 0.01%	209	582	118	–	–	281
≥ 0.1%	173	514	118	–	–	210
≥ 1.0%	167	472	118	–	–	199
≥ 5.0%	166	471	118	–	–	196
≥ 10.0%	118	401	118	–	–	148
≥ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

Best characterization

Many reported vulnerabilities

Non-PIN input state is **not** satisfied

Non-PIN input state is satisfied

No details for less than all inputs

No conclusion on more than one input

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

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

Additional Results

Can be applied to any program, not necessarily under fault injection

- Generic Framework
- Evaluation on SVComp

Detailed strategies for efficient language exploration

- Analyses of the influence of the strategies

Generalization to trace properties

- Not limited to symbolic execution

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

BINSEC

(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

Questions?

Questions