Adversarial Reachability for Program-level Security Analysis

Soline Ducousso[1], Sébastien Bardin[1], Marie-Laure Potet[2]

[1] Univ. Paris-Saclay, CEA, List, Saclay, France
[2] Univ. Grenoble Alpes, VERIMAG, Grenoble, France
soline.ducousso@cea.fr, sebastien.bardin@cea.fr, marie-laure.potet@univ-grenoble-alpes.fr

Context - Formal Program Analysis and Safety

- Formal methods
- Worst case ➔ all possible behaviors are studied
- Verification specifications
- Bug finding or absence of bugs
- Industrial success

What About Security ?

Reuse standard safety analyzers:

- ❏ Useful (e.g., buffer overflows) and worst case
- ❏ Weak attacker model ➜ can only craft smart inputs ➜ **still plays the rule**

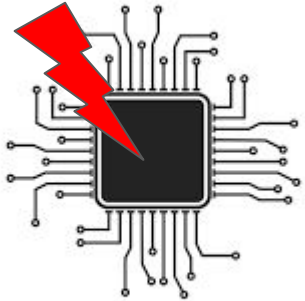
What About Security ?

Reuse standard safety analyzers:

- ❏ Useful (e.g., buffer overflows) and worst case
- ❏ Weak attacker model ➔ can only craft smart inputs ➔ **still plays the rule**

Real-world attackers are more powerful:

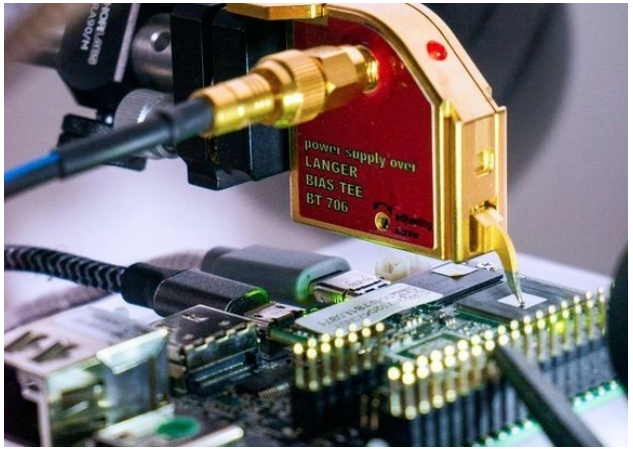
- ❏ Side channels
- ❏ **Fault injection**



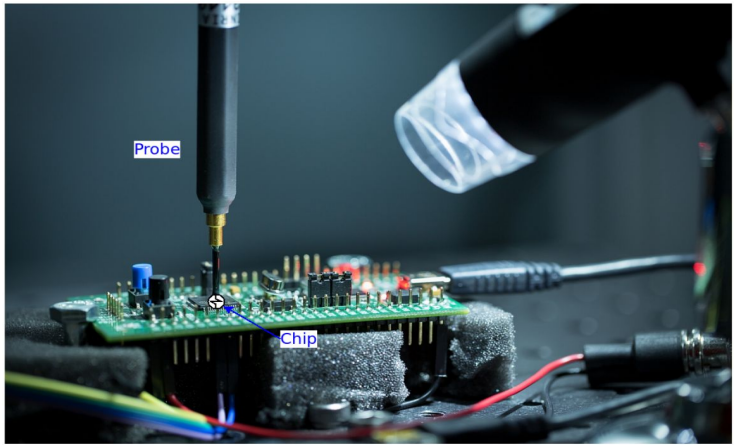
Target crypto. primitives, but also bootloaders, firmware update modules, enclaves, …

Historical Example - Hardware Fault Injection Attacks

- Security critical components (e.g. smartcards)
- Attack technique to trigger other system behaviors ➔ fault injection

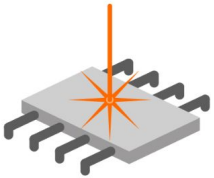


Credit: https://eshard.com

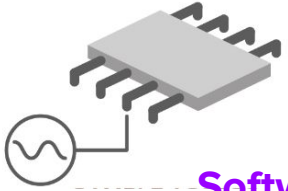


Bukasa et al. *When fault injection collides with hardware complexity*. FPS 2018

Fault Injection Attacks Everywhere

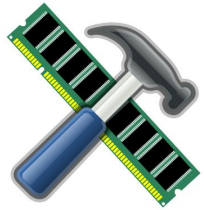


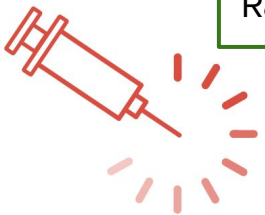
Hardware attacks



Software-implemented hardware attacks

Electromagnetic pulses	Power glitch	Faultline	**Rowhammer**
Clock glitch	Laser beam	Spectre	Binary rewriting	DVFS
Race condition	Load Value Injection	Halt and modify execution	



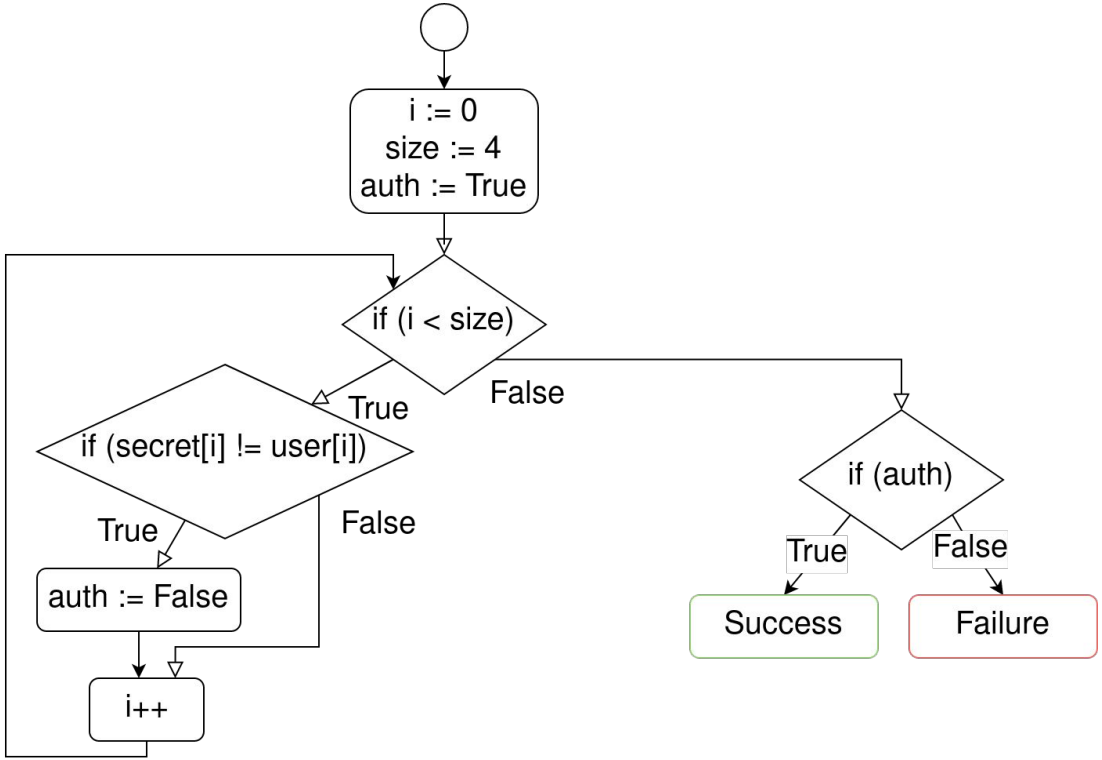


Micro-architectural attacks

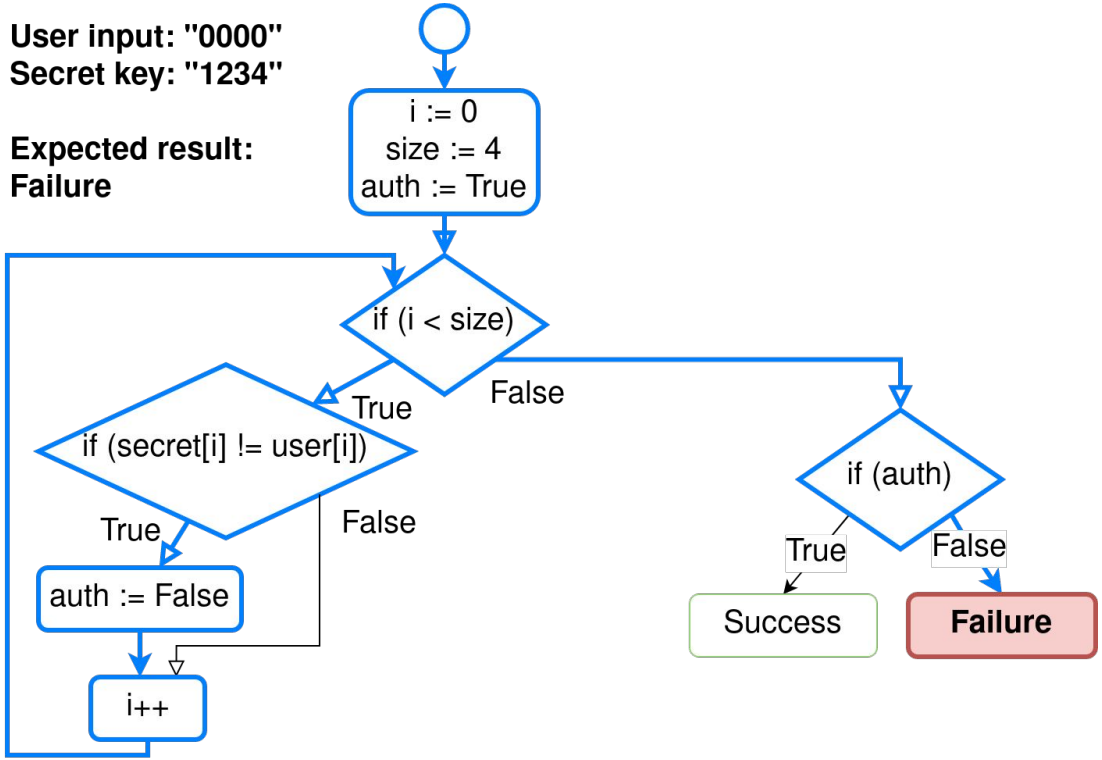
Link with data-only attacks

Man-At-The-End attacks

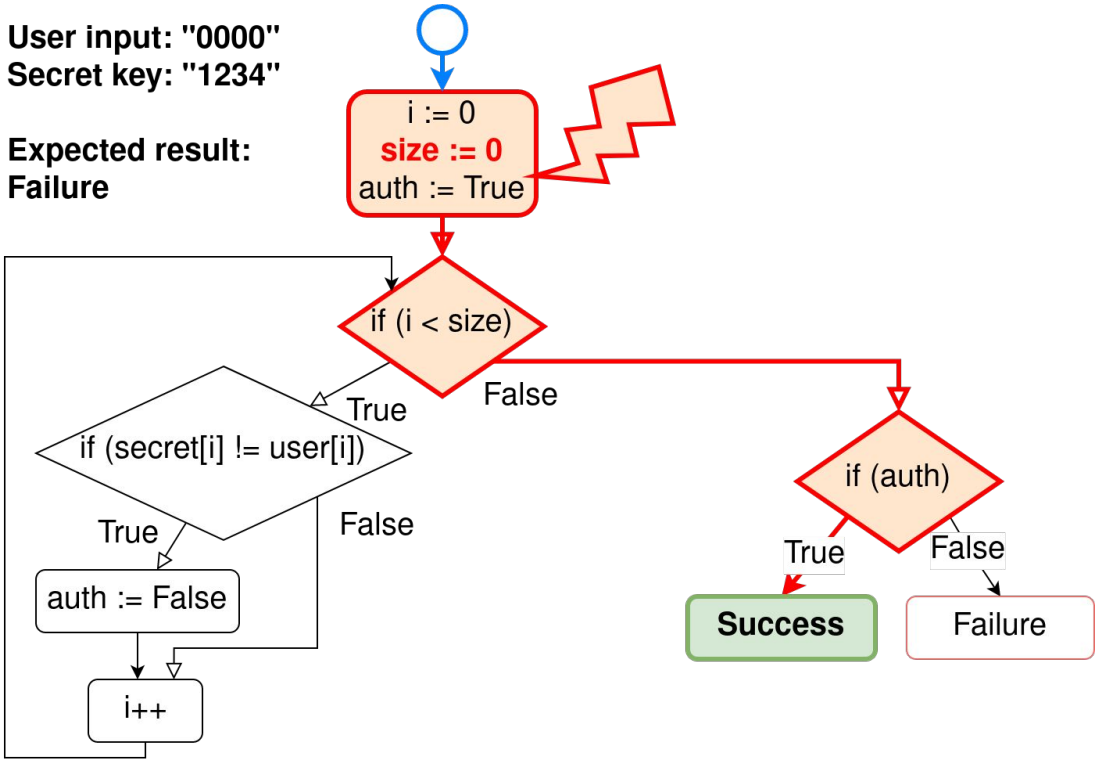
Motivating Example - VerifyPIN



Motivating Example - Without Faults



Motivating Example - Data Fault



Faults at Program-Level

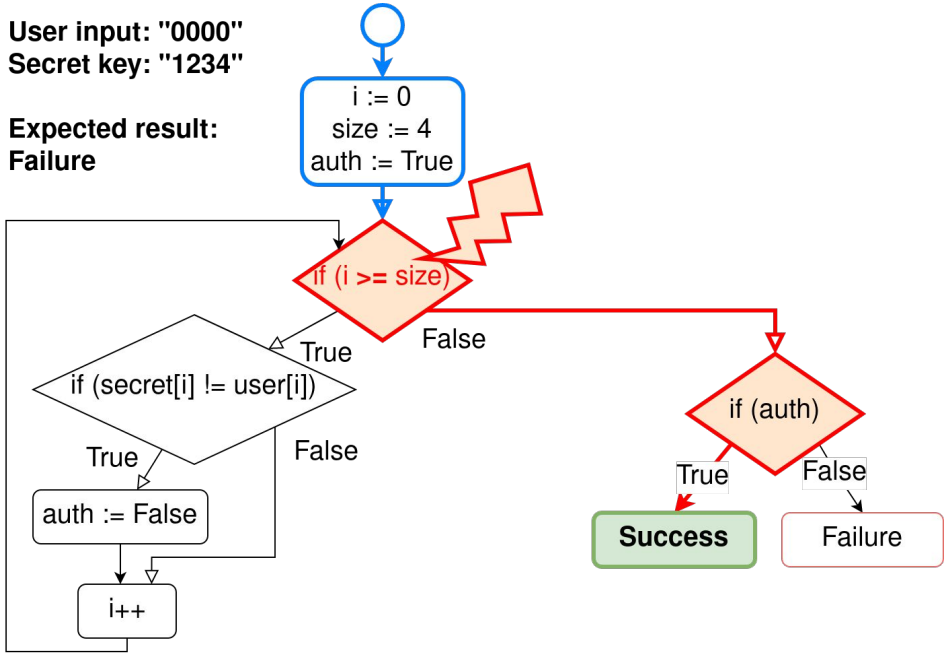
Fault Models:

- ❏ Data faults
- ❏ Control-flow
- ❏ Instruction modifications

Protections:

- ❏ Control-flow integrity
- ❏ Redundancy

Hard to reason about (multi-faults)



Our Goal

Our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto a program security properties.*

Challenges:

C1: Formal framework

Impact of advanced attacker

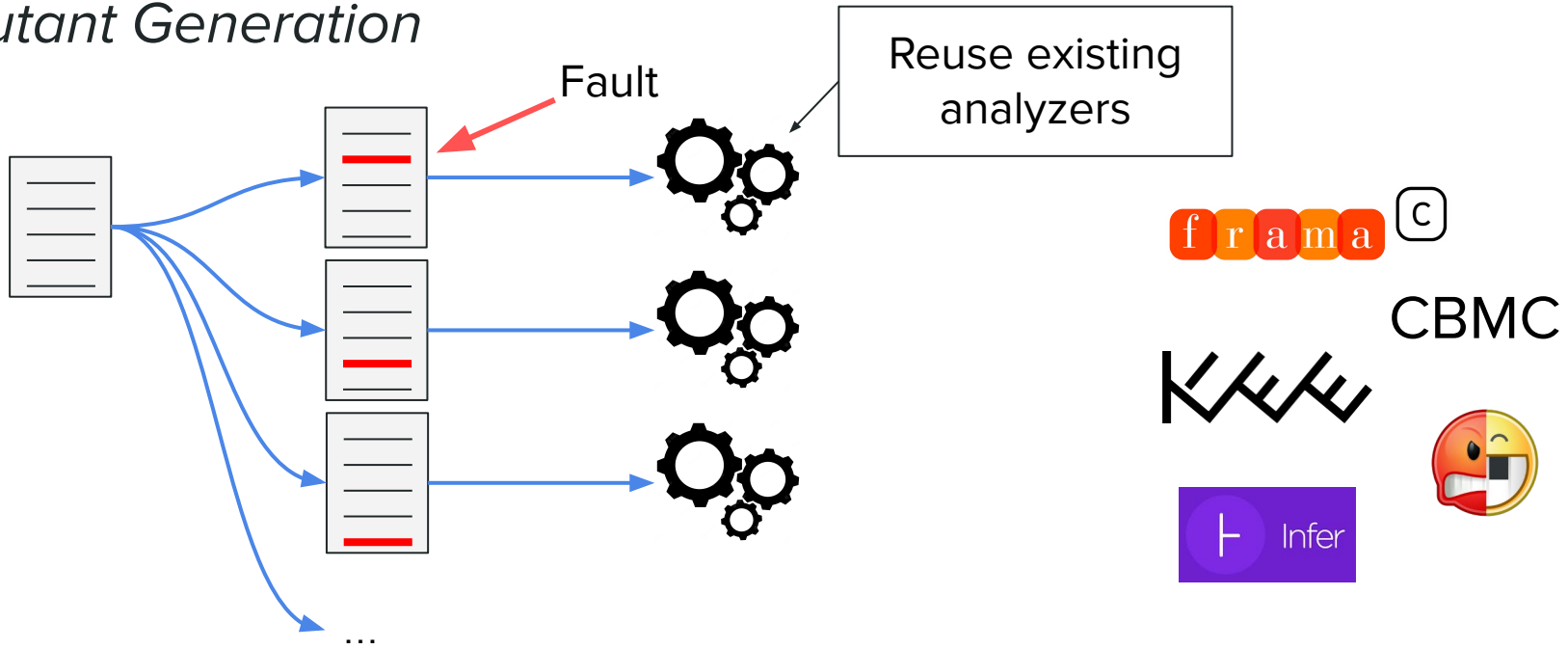
C2: Efficient and generic algorithm

Multi-fault without path explosion

*attacker able to perform multi-fault injections

Christofi 2013, Rauzy 2014, Given-Wilson 2017, Carré 2018

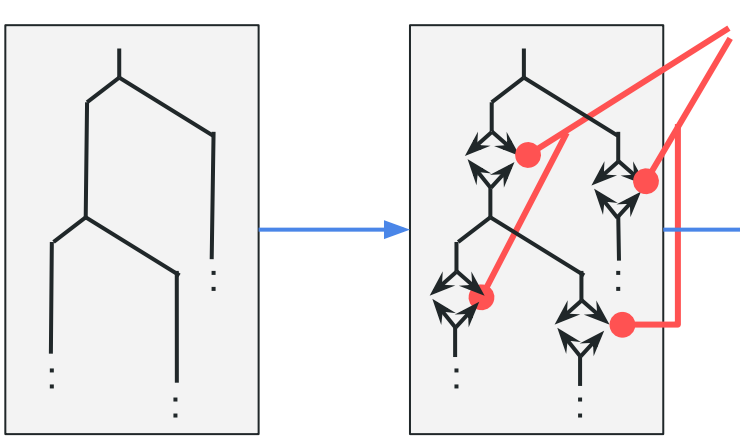
Mutant Generation



Scale for multi-fault?

Potet, Mounier, Puys and Dureuil: *Lazart*, 2014

Forking Technique



Potential fault injections

Reuse existing analyzers

frama C

CBMC

Infer

Scale for multi-fault?

Contributions

- Formalize of the **Adversarial Reachability** problem
- **Adversarial Symbolic Execution** to answer adversarial reachability
  - a novel **forkless fault encodings** preventing path explosion
  - 2 **optimizations** reducing query complexity
- **Implementation** and **evaluation** of our technique
- Security analysis of the **WooKey bootloader**

Introduction

Adversarial Reachability Formalization

Forkless Adversarial Symbolic Execution

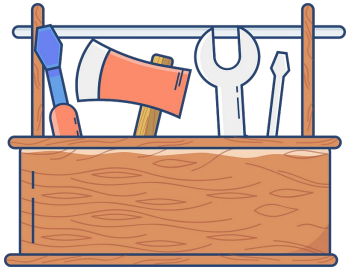
Experimental Evaluation

Conclusion

Attacker model

Advanced attacker:

- ❏ various different attack vectors = various effects
- ❏ multiple actions



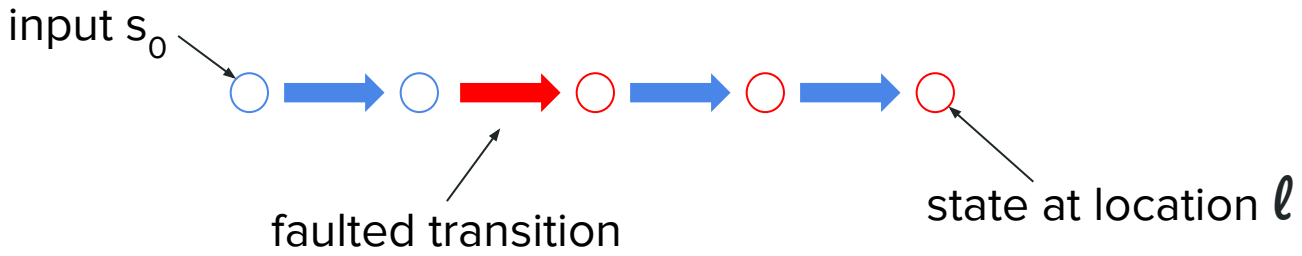
Attacker model:

1) A set of attacker actions (equivalent to fault models)
2) A maximum number of actions
3) A goal expressed as a reachability query

Adversarial reachability

Faults on data | **Faults on control-flow**

Adversarial reachability: A location ℓ is adversarially reachable in a program P for an attacker model A if $S_0 \mapsto^* \ell$,
where $\mapsto^*$ is a succession of **normal transitions** interleaved with **faulty transitions**



Definition of ***correctness*** and ***completeness*** of an analysis w.r.t an attacker model

Introduction

Adversarial Reachability Formalization

Forkless Adversarial Symbolic Execution

Experimental Evaluation

Conclusion

Forkless Adversarial Symbolic Execution (FASE)

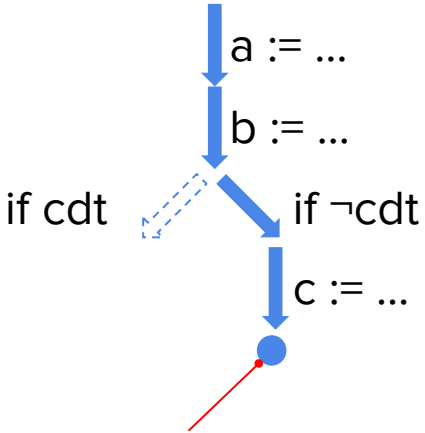
Design guideline	Technical solution
Correct and k-complete for adversarial reachability	Based on Symbolic Execution
Prevent path explosion	Forkless fault encoding
Reduce complexity of created formulas	Avoid introducing extra faults with 2 optimizations

Faults on data | **Faults on control-flow**

Reminder - Symbolic Execution

- **Symbolic inputs**
- **Follow each path**, compute its path predicate
- **Assess reachability** with an SMT solver
- Get **model** for symbolic inputs

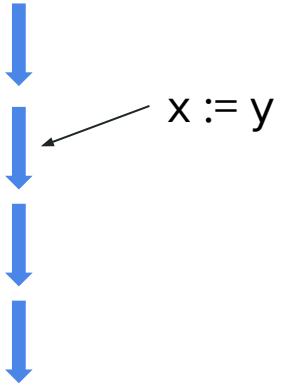
- Properties: **correct and k-complete**



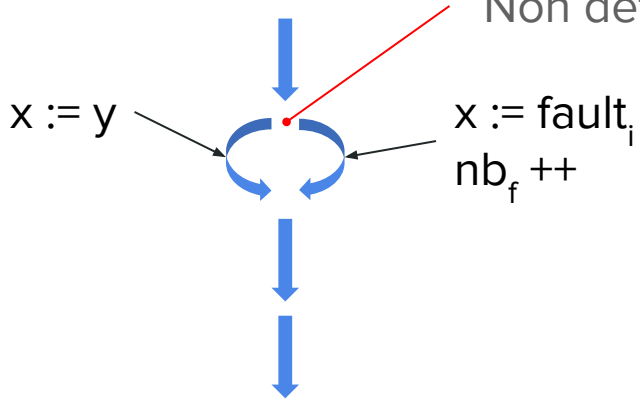
SMT query: $(a = ...) \wedge (b = ...) \wedge (\neg cdt) \wedge (c = ...)$ SAT ?

Problem n°1: path explosion	*Problem n°2: constraint solving*

Symbolic Execution



Forking technique

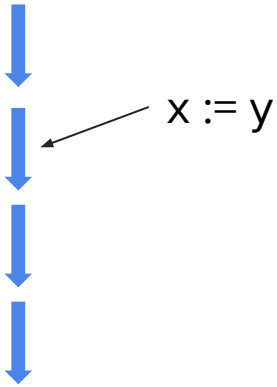


- Covers all adversarial behaviors
- Number of path exponential with # fault injection points

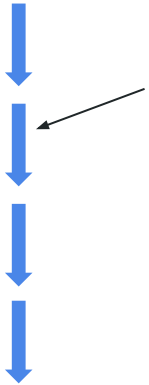
Problem n°1: path explosion ✗	*Problem n°2: constraint solving*

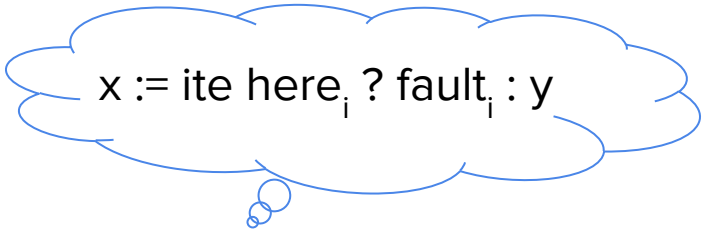
Forkless encodings and FASE

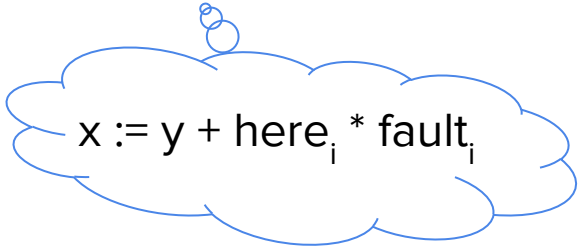
Symbolic Execution



FASE





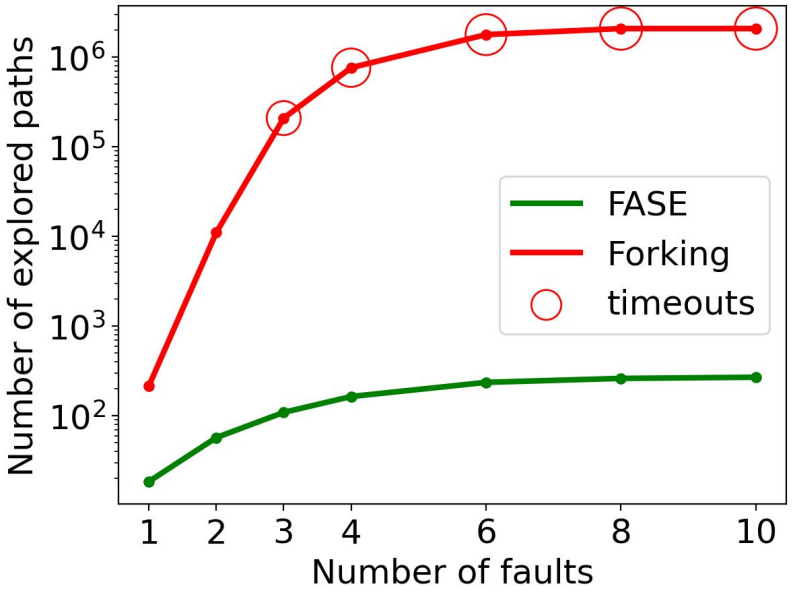


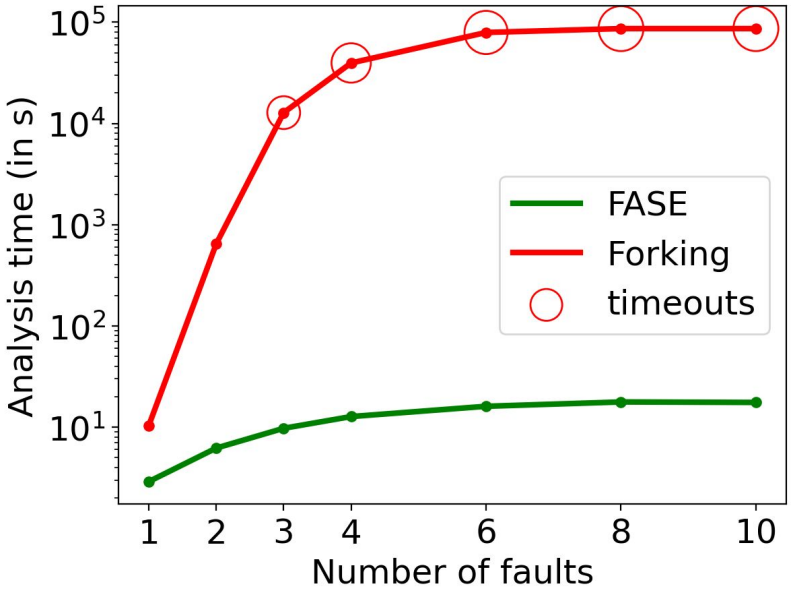
- ❑ Covers all adversarial behaviors
- ❑ **No extra path explosion**
- ❑ **More complex formulas**

$here_i \in [0,1], \Sigma\ here_i \leq max_f$

Problem n°1: path explosion ✅	*Problem n°2: constraint solving* ❌

Experimental Evaluation - Path explosion



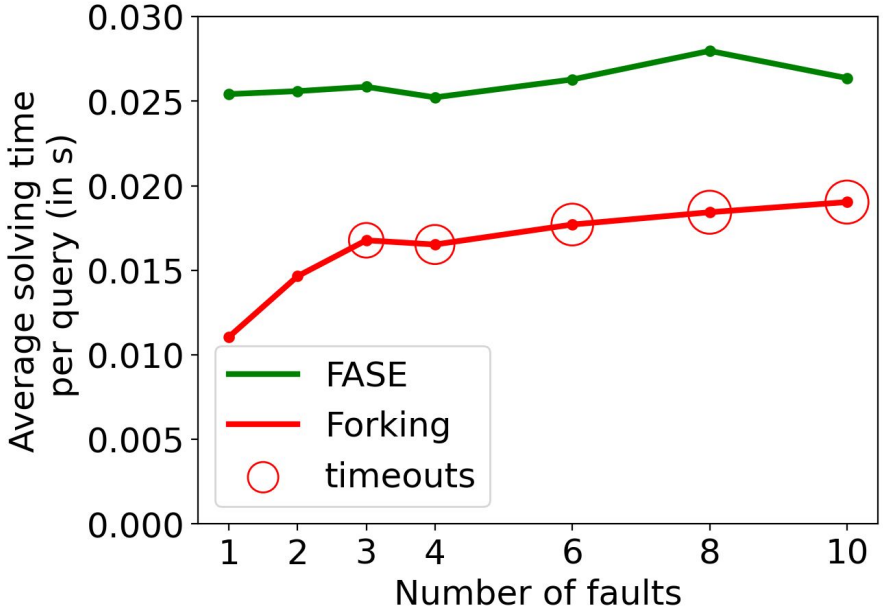


➔ Forking explodes in explored paths while FASE doesn't

➔ Translates to improved analysis time overall

Forkless encodings and FASE

Forking	0 ite /query on average (1 fault)
FASE	5,745 ite /query on average (1 fault)



Problem n°1: path explosion ✅ | *Problem n°2: constraint solving* ❌

Optimizations Overview

Main Goal:

- ❏ Reduce #injection points to simplify formulas
- ❏ Some injection points are not accessible to the attacker model

Early Detection of Fault Saturation (EDS)

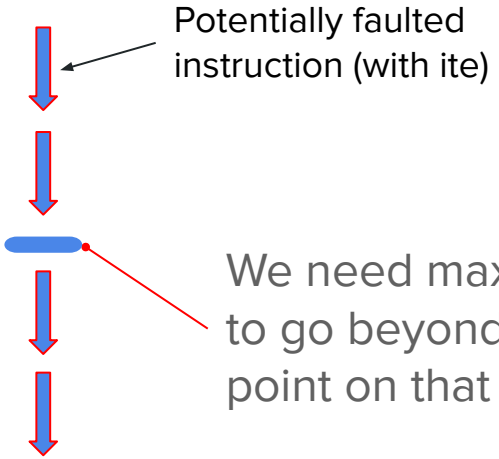
Stop injection as soon as possible

Injection On Demand (IOD)

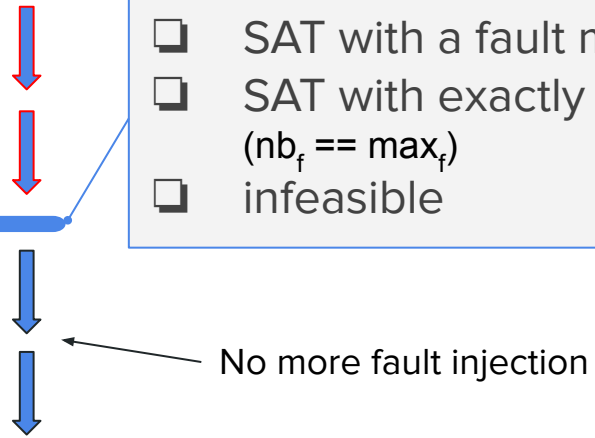
Add faults only when necessary

Early Detection of fault Saturation (EDS)

FASE



FASE-EDS



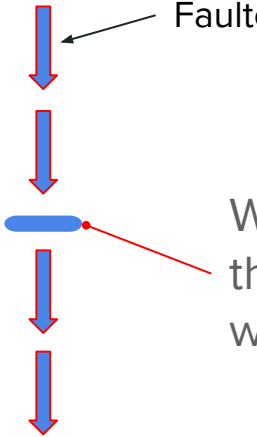
- ❑ SAT with a fault margin ($nb_f < max_f$)
- ❑ SAT with exactly the fault budget ($nb_f == max_f$)
- ❑ infeasible

- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ **Reduce number of fault injections along a path**

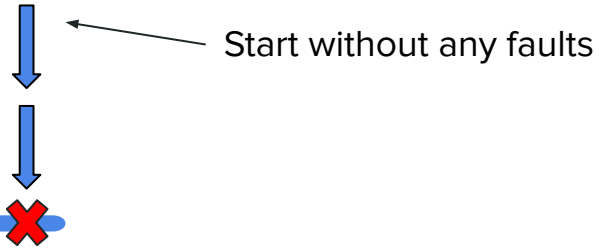
Problem n°1: path explosion ✅	*Problem n°2: constraint solving* ✅

Injection On Demand (IOD)

FASE



FASE-IOD

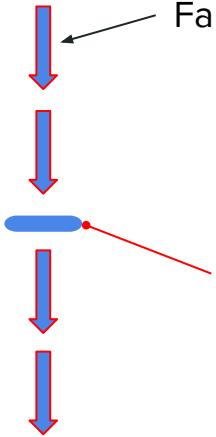


- ❏ Covers all adversarial behaviors, as complete as FASE
- ❏ **Reduce number of fault injections** in SMT queries

Problem n°1: path explosion ✅	*Problem n°2: constraint solving* ✅

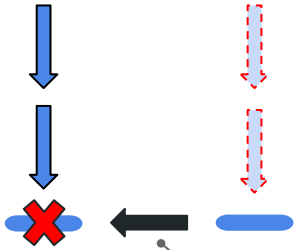
Injection On Demand (IOD)

FASE



We can't go beyond that point on that path without more faults.

FASE-IOD



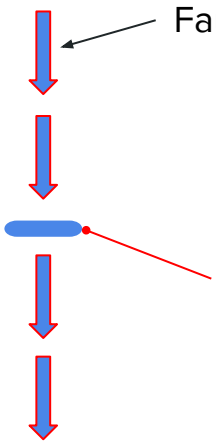
Path predicate switched for the faulted one

- ❏ Covers all adversarial behaviors, as complete as FASE
- ❏ **Reduce number of fault injections** in SMT queries

Problem n°1: path explosion ✅	*Problem n°2: constraint solving* ✅

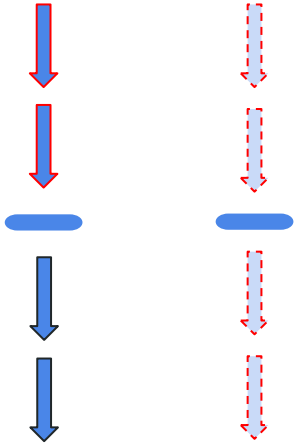
Injection On Demand (IOD)

FASE



We can't go beyond that point on that path without more faults.

FASE-IOD

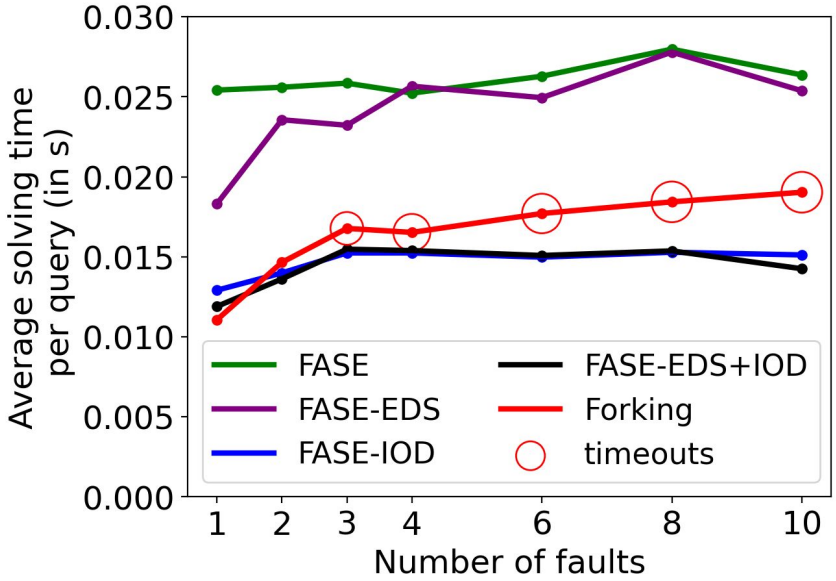


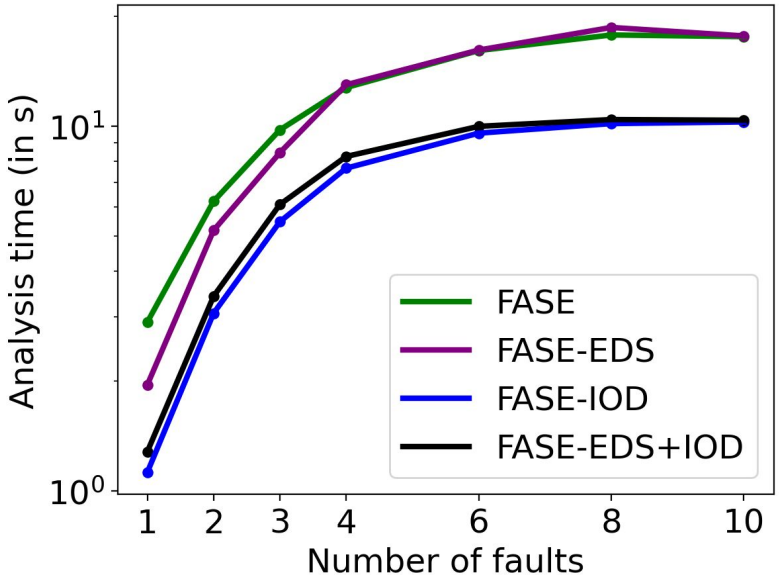
Bonus: under-approximation of nb_f

- ❏ Covers all adversarial behaviors, as complete as FASE
- ❏ **Reduce number of fault injections** in SMT queries

Problem n°1: path explosion ✅	*Problem n°2: constraint solving* ✅

Experimental Evaluation - Optimizations' Impact





- ➔ EDS has a moderate impact
- ➔ IOD halves solving time per query (5745 →3050 avg ite /query) + most efficient
- ➔ IOD+EDS is slightly more expensive

Other Forkless Fault Models

Table 1: Forkless encodings for various fault models

Fault model	original instruction	Forkless encoding
Arbitrary data	$x := expr$	$x := ite\ fault\_here\ ?\ fault\_value\ :\ expr$
Variable reset	$x := expr$	$x := ite\ fault\_here\ ?\ 0x00000000\ :\ expr$
Variable set	$x := expr$	$x := ite\ fault\_here\ ?\ 0xffffffff\ :\ expr$
Bit-flip	$x := expr$	$x := ite\ fault\_here\ ?$ $(expr\ xor\ 1 << fault\_value)\ :\ expr$
Test inversion	$if\ cdt\ then\ goto\ 1$ $else\ goto\ 2$	$if\ (ite\ fault\_here\ ?\ !cdt\ :\ cdt)$ $then\ goto\ 1\ else\ goto\ 2$

[LVI] 

[Rowhammer] 

[CFI Hijack]

Introduction
Adversarial Reachability Formalization
Forkless Adversarial Symbolic Execution
Experimental Evaluation
Conclusion

Evaluation

Implementation inside BINSEC for x86-32 architecture with SMT solver Bitwuzla

Benchmarks (RQ1 to 3) from [1, 2]

- **RQ1:** is our tool correct and k-complete? In particular, can we find attacks on vulnerable programs and prove secure resistant programs?
- **RQ2:** can we scale in number of faults?
- **RQ3:** what is the impact of our optimizations?
- **Different security scenarios** using different fault models
- **Larger case study** of the WooKey bootloader [ANSSI security challenge]

[1] Dureuil et al. *FISSC: A fault injection and simulation secure collection*. 2016.
[2] Le et al. *Resilience evaluation via symbolic fault injection on intermediate code*. 2018

Security scenarios using different fault models

Application	Version	Attacker model	Expected	Result
CRT-RSA [1]	basic	1 reset	vulnerable	vulnerable (BellCore attack)
	Shamir		vulnerable	time-out
	Aumuller		resistant	time-out
Secret keeping machine [2]	linked-list	1 bitflip in memory	vulnerable	vulnerable
	array	1 bitflip in memory	resistant	resistant
	array	1 bitflip anywhere	???	vulnerable
SecSwift protection [3] (ST Micro.)	[4] version applied to VerifyPIN_0	1 arbitrary data (same effect than test inversion)	???	vulnerable (early loop exit, valid in CFG)

[1] Puys et al. *High-level simulation for multiple fault injection evaluation*. 2014
[2] Dullien *Weird machines, exploitability, and provable unexploitability*. 2017
[3] de Ferrière *Software countermeasures in the llvm risc-v compiler.* 2021
[4] Lacombe et al. *Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities*. 2021

Case study

WooKey bootloader [security challenge]: secure data storage by ANSSI, 3.2k loc

Attacker model: 1 arbitrary data — or test inversion with equivalent effect

1. **Find known attacks** (from source-level analysis)
   a. Boot on the old firmware instead for the newest one [1]
   b. A buffer overflow triggered by fault injection [1]
   c. An incorrectly implemented countermeasure protecting against one test inversion [2]
2. **Evaluate recent countermeasures** [1]
   a. Evaluate original code ➔ We **found an attack not mentioned before**
   b. Evaluate existing protection scheme [1]
   c. Propose and evaluate our own protection scheme

[1] Lacombe et al. C*ombining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities*. 2021
[2] Martin et al. *Verifying redundant-check based countermeasures: a case study*. 2022

Introduction
Adversarial Reachability Formalization
Forkless Adversarial Symbolic Execution
Experimental Evaluation
Conclusion

Discussions

Fault models limitations:

- ❑ Instruction corruption
- ❑ Spectre attack ➔ speculative rollback vs. max fault

Other analysis techniques:

- ❑ BMC, CEGAR, … (based on path unrolling) could benefit
- ❑ Possibility of code instrumentation for reuse of analyzers *[Forkless* ✅ *- Optims* ❌*]*
- ❑ Over-approximation analysis

Other properties:

- ❑ We consider location reachability for the sake of simplicity
- ❑ Direct extension to local assertions (buffer-overflow) and finite traces (use-after-free)
- ❑ ? Possible extension to liveness, hyperproperties, …

C1: Formal Framework → Adversarial Reachability

C2: Efficient Algorithm → Forkless Adversarial Symbolic Execution (+ 2 optimizations)

Implementation

Evaluation

Use Case: WooKey Bootloader

Security Scenarios

BINSEC