Fault attack vulnerability assessment of binary code

Journée thématique sur les attaques par injection de fautes [JAIF'19], Minatec, Grenoble

Mai 23, 2019

Jean-Baptiste Bréjon
Emmanuelle Encrenaz
Karine Heydemann
Quentin Meunier
Son-Tuan Vu

Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, F-75005 Paris, France

Plan

- Context
- Our approach to **vulnerability Assessment**
- Results exploitation: **Security Metrics**
- Implementation in a tool: **RobustB**
- Use-Cases
- Conclusion

Context

- **Embedded systems** is now a prime target to attackers as they increasingly manipulate **sensitive data**.
- **Fault attack** is real threat to their security: bypass security mechanisms, performs privilege escalation, ... [Yuce et al. 2018]

How can we protect from them? → **Software protections**

- Can be implemented at all code levels: Source, IR, ASM

⚠ Compiler optimisations and back-end can **alter/remove** them

→ Their design follows a trial-and-error process:

  - Code review → error prone
  - Fault injection campaign → require costly equipment and specific skills

→ Need a more efficient/automatic way to assess the security of low-level code

Vulnerability Assessment

Different approaches to **low-level vulnerability assessment** have been explored

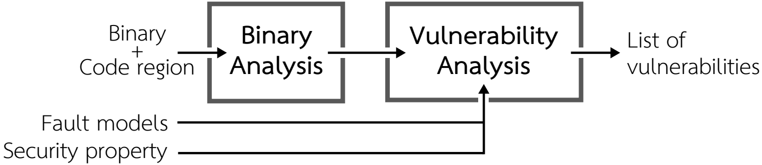
- Symbolic execution + model-checking [Pattabiraman et al. 2013]
- Mutants + model-checking [Given-Wilson et al. 2018]
- Simulation [Dureuil 2016]

Vulnerability assessment approaches face a **precision vs speed** trade-off

Our objective: precision and exhaustiveness

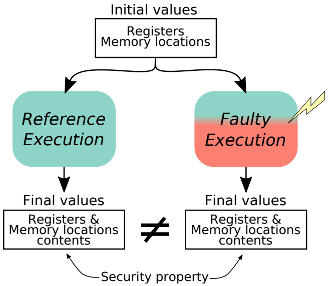
- From the **binary**
- Combines **static** analysis, **dynamic** analysis and **formal methods**

Overview

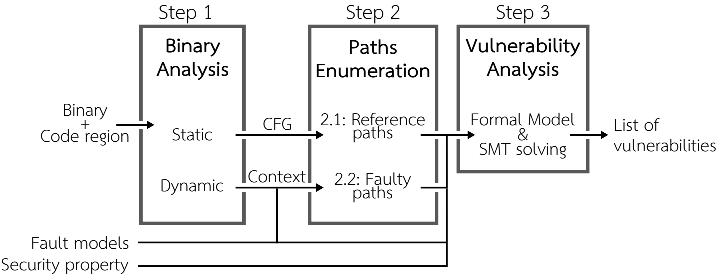


Search for the vulnerabilities (i.e. invalidation of the **security property**) of a **code region** in a binary to a **fault model** (e.g. instruction skip)

- Equivalence-checking: **comparing** a non-faulty execution with a faulty one
- The comparison is carried out under the **same configuration of inputs**
- The **security property** defines the elements (i.e. register) to be compared at the end of both executions

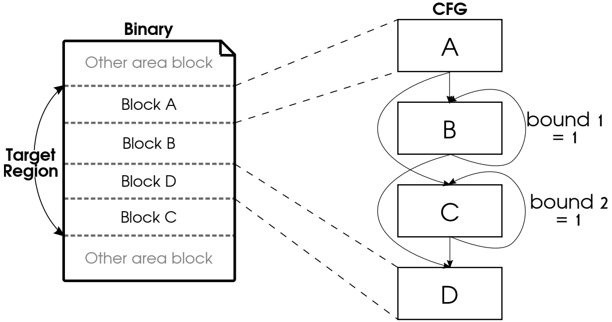


Overview



- 1 - **Extract** a representation of the **code region** and **Context**
- 2.1 - **Determine** the possible **execution paths** within the code region
- 2.2 - Single **fault injection** on the possible execution paths
- 3 - **Search for vulnerabilities** by formal verification of a non-equivalence property (SMT)
  ⇒ **Vulnerability list** including their **locations**

Information Extraction From the Binary



Static analysis

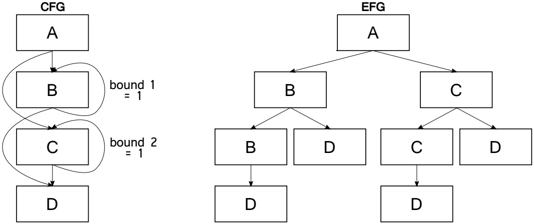
- CFG construction + Blocks order

Dynamic/symbolic analysis

- Extracts execution contexts of the code region
- Extracts loop bounds within the code region

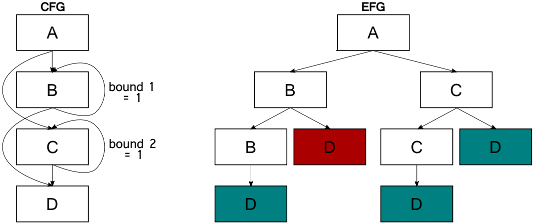
Determining the Possible Execution Paths

- Static bounded unfolding of the CFG

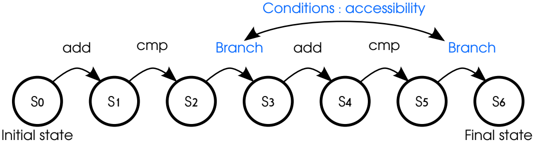


Determining the Possible Execution Paths

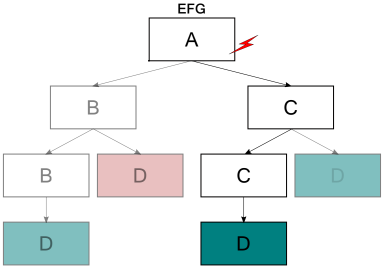
- Static bounded unfolding of the CFG

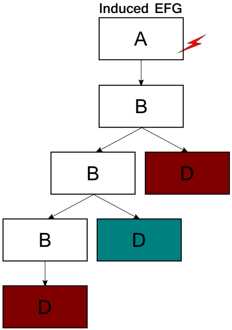


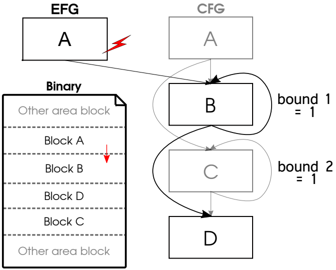
- Resulting paths accessibility test (SMT)
  - → Each instruction is modeled regarding its effect on a machine state model



Determining Faulty Execution Paths



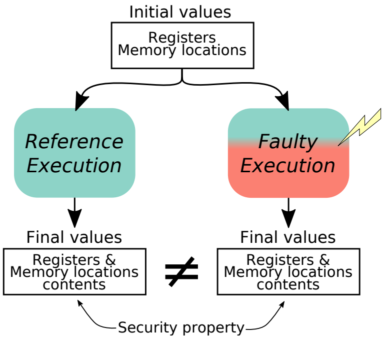




- **A fault may alter the execution flow**
  → Possible execution paths are **recomputed after a fault injection**
- CFG unfolding after the fault
  - Takes into account the code layout
  - Relaxed loop bounds
- Resulting paths are checked for accessibility

Robustness Analysis

- P_Orig → Original execution path
- $P\_Faulted$ → Faulty execution path



- Same context (C)
- When the **final values** of some memorizing elements **differ**, a **vulnerability** is detected

Formula:

$Access(P\_Orig, C) \wedge Access(P\_Faulted, C) \wedge Vuln$

→ **SAT**: The fault in $P\_Faulted$ leads to a vulnerability

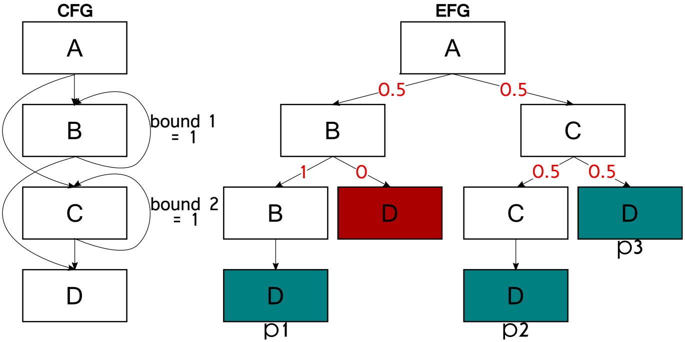
- Repeating this process for all faults on all injection points produces a vulnerability list

Results Synthesis

- Vulnerability list is cumbersome to analyse
  - How dangerous is each vulnerability?
  - How to compare the vulnerabilities of two different implementations?
- Need for a synthetic view
- Introduction of three security metrics
  - **Instruction sensitivity level**
  - **Average number of vulnerabilities in paths**
  - **Vulnerabilities density**

Paths Probabilities

A vulnerability appearing on a path should be **weighted differently** than one appearing on another path depending on the **likelihood of their path**.



- By default: paths have equal probability
- Ideally: user can define the branches probability

Path	Blocks	P(path)
p1	A - B - B - D	0.5
p2	A - C - C - D	0.25
p3	A - C - D	0.25

Instruction Sensitivity (IS)

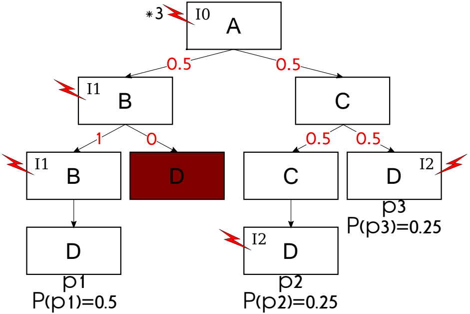
IS(i): score reflecting instruction i sensitivity

$$IS(i) = \sum_{p \in Paths} P(p \ is \ taken) \times NV_i(p)$$

$NV_i(p)$: **Instruction i #Vulnerabilities on path** p

Inst	Score
`I0`	$\mathbf{1} = P(p1) + P(p2) + P(p3)$
`I1`	$\mathbf{1} = 2 * P(p1)$
`I2`	$\mathbf{0.5} = P(p2) + P(p3)$

Each **vulnerable instruction** occurence is **weighted** relatively to the **likelihood of the path** it appears on



Rank the instructions according to their **sensitivity** → helps the designer to focus on the most sensitive instructions

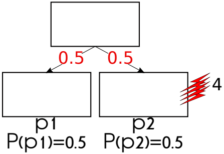
Attack Surface (AS)

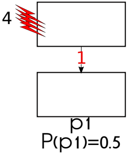
AS: average number of vulnerabilities on an execution path

$$\boxed{AS = \sum_{p \in Paths} P(p\ is\ taken) \times NV(p)}$$

***NV(p)*: #Vulnerabilities appearing on path p**

4 vulnerabilities, on each example, **weighted** by paths probabilities





$\mathbf{AS} = 4 * 0.5 = \mathbf{2}$

2 vulnerabilities found on average

$\mathbf{AS} = 4 * 1 = \mathbf{4}$

4 vulnerabilities found on average

The higher the **attack surface**, the more the attacker will be able to inject a fault leading to a **vulnerability**

Normalized Attack Surface (NAS)

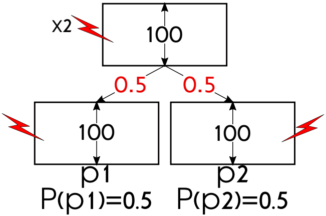
NAS: Average density of vulnerabilities

$$\boxed{NAS = \frac{AS}{\sum_{p \in Paths} P(p\ \ is\ \ taken) \times NI(p)} = \frac{AS}{ANI}}$$

***NI(p)*: Path p #Instructions**

***ANI*: Average number of instructions per path**

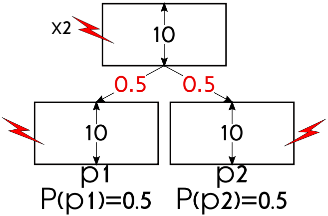
Same vulnerabilities but different **amount of instructions**: affects vulnerability density



$AS = 2 * 0.5 + 2 * 0.5 = 2$

$\textbf{NAS} = 2/(100+100) = \mathbf{0.01}$

Odds for a randomly timed fault injection to lead to a vulnerability: **1%**



$AS = 2 * 0.5 + 2 * 0.5 = 2$

$\textbf{NAS} = 2/(10+10) = \mathbf{0.1}$

Odds for a randomly timed fault injection to lead to a vulnerability: **10%**

RobustB

- The approach has been implemented in a tool called **RobustB**
- Supports **ARM thumb2** instruction set
- Formal models are in **SMT-LIB** standard (Z3, boolector, ...)
- The security property can now be given to RobustB directly from the source code for **more semantic and automatism** (Thesis of Son-Tuan Vu)
- Implements 4 fault models
  - Instruction skip
  - Register corruption
  - Instruction replacement
  - Instruction bit set
- Uses **angr** [Shoshitaishvili et al. 2016] (binary analysis) and **Capstone** (disassembly functionality)

Use-case: VerifyPin

Description

- Belongs to the **FISCC** (Fault Injection and Simulation Secure Code Collection) benchmarks, dedicated to fault injection analysis
- Compares a user PIN with a predefined PIN
    - Authentication "OK" if PINs are identical, "KO" otherwise
- Several versions of the function, each one combining different source-level protections

Analysis

- When **user PIN and predefined PIN differs** the security property is **Authentication = "KO"**
- 4 versions: 1 unprotected, 3 protected
- 2 optimisation levels: O0, O2
- Fault model: instruction skip

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0						
Loop counter*2	VerifyPin_4						
Double call	VerifyPin_5						
Result var*2 Step counter(CFI)	VerifyPin_7						

- Four implementations of VerifyPin

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0					
		O2					
Loop counter*2	VerifyPin_4	O0					
		O2					
Double call	VerifyPin_5	O0					
		O2					
Result var*2 Step counter(CFI)	VerifyPin_7	O0					
		O2					

- Two optimisation levels

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4				
		O2	4				
Loop counter*2	VerifyPin_4	O0	15				
		O2	1				
Double call	VerifyPin_5	O0	15				
		O2	1				
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15				
		O2	1				

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96			
		O2	4	54			
Loop counter*2	VerifyPin_4	O0	15	127			
		O2	1	28			
Double call	VerifyPin_5	O0	15	15			
		O2	1	8			
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67			
		O2	1	24			

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37		
		O2	4	54	10.38		
Loop counter*2	VerifyPin_4	O0	15	127	7.75		
		O2	1	26	26		
Double call	VerifyPin_5	O0	15	15	1		
		O2	1	8	8		
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75		
		O2	1	24	24		

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin$_0$	O0	4	96	18.37	0.25	
		O2	4	54	10.38	0.41	
Loop counter*2	VerifyPin$_4$	O0	15	127	7.75	0.05	
		O2	1	26	26	0.71	
Double call	VerifyPin$_5$	O0	15	15	1	0.01	
		O2	1	8	8	0.17	
Result var*2 Step counter(CFI)	VerifyPin$_7$	O0	15	67	4.75	0.03	
		O2	1	24	24	0.48	

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37	0.25	73.9
		O2	4	54	10.38	0.41	25.3
Loop counter*2	VerifyPin_4	O0	15	127	7.75	0.05	149.1
		O2	1	26	26	0.71	49
Double call	VerifyPin_5	O0	15	15	1	0.01	124.2
		O2	1	8	8	0.17	48
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75	0.03	180.1
		O2	1	24	24	0.48	50

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37	0.25	73.9
		O2	4	54	10.38	0.41	25.3
Loop counter*2	VerifyPin_4	O0	15	127	7.75	0.05	149.1
		O2	1	26	26	0.71	49
Double call	VerifyPin_5	O0	15	15	1	0.01	124.2
		O2	1	8	8	0.17	48
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75	0.03	180.1
		O2	1	24	24	0.48	50

- VerifyPin_5 is the **least sensitive** implementation (for all metrics) → Double call bests targets the instruction skip fault model

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37	0.25	73.9
		O2	4	54	10.38	0.41	25.3
Loop counter*2	VerifyPin_4	O0	15	127	7.75	0.05	149.1
		O2	1	26	26	0.71	49
Double call	VerifyPin_5	O0	15	15	1	0.01	124.2
		O2	1	8	8	0.17	48
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75	0.03	180.1
		O2	1	24	24	0.48	50

- VerifyPin_5 O0 is the **least sensitive** version according to **AS** and **NAS**, the number of raw vulnerabilities disagree

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37	0.25	73.9
		O2	4	54	10.38	0.41	25.3
Loop counter*2	VerifyPin_4	O0	15	127	7.75	0.05	149.1
		O2	1	26	26	0.71	49
Double call	VerifyPin_5	O0	15	15	1	0.01	124.2
		O2	1	8	8	0.17	48
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75	0.03	180.1
		O2	1	24	24	0.48	50

- **NAS** metric shows the odds of a successful randomly timed attack. Higher for O2 versions → smaller code + less paths

Results

- **Vulns**: Raw number of vulnerabilities
- **ANI**: Average number of instructions per path
- **RP**: Number of paths in the original code

Protection	Version	Opt level	#RP	#Vulns	**AS**	**NAS**	**ANI**
None	VerifyPin_0	O0	4	96	18.37	0.25	73.9
		O2	4	54	10.38	0.41	25.3
Loop counter*2	VerifyPin_4	O0	15	127	7.75	0.05	149.1
		O2	1	26	26	0.71	49
Double call	VerifyPin_5	O0	15	15	1	0.01	124.2
		O2	1	8	8	0.17	48
Result var*2 Step counter(CFI)	VerifyPin_7	O0	15	67	4.75	0.03	180.1
		O2	1	24	24	0.48	50

- **VerifyPin_0**: **AS is higher** for O0 version → less instructions = lower attack surface. In protected versions: O2 optimisation level affected the protections.

Other use-cases

Source level hardened code analysis

- Impact of optimisation levels [Dureuil et al. 2016]
  - → Highlighted metrics usefulness to compare different, functionally identical, code versions
- GCC vs Clang
  - → Highlighted redundant protections w.r.t. instruction skip and register corruption fault models

Compile-time hardened code analysis

- Compile-time hardened loop construct [Proy et al. 2017]
  - → Validation of the robustness of the loop under the targeted fault model
  - → One vulnerability found: due to code placement (fault outside the loop construct)
- Compile-time hardened code by instruction duplication [Barry et al. 2016]
  - → Validation of the robustness of the binary against instruction skip

Conclusion

- A tool for analysing **binary code** regions against single **fault attacks**
- Comparison of compilers, **optimisation effects** and **protections effectiveness** on a use-case
- 3 **security metrics** synthetizing the results

Pros

- **Automatic**
- Formal verification (SMT) → **exhaustiveness**
- **Contextual** analysis

Cons

- **Small code regions** → speed of the analysis depends on the number of possible paths and the number of memory accesses
- Exhaustive multiple faults → combinatorial explosion, but the approach does not forbid it

Thanks !

Bibliography I

Thierno Barry, Damien Couroussé, and Bruno Robisson. "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks". In: *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*. ACM. 2016, pp. 1–6.

Louis Dureuil et al. "FISSC: a Fault Injection and Simulation Secure Collection". In: *Proceedings of International Conference on Computer Safety, Reliability and Security*. Vol. 9922. LNCS. Trondheim, Norway: Springer Berlin / Heidelberg, 2016, pp. 3–11. doi: `10.1007/978-3-319-45477-1_1`.

Louis Dureuil. "Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute". PhD thesis. Université Grenoble Alpes, 2016.

Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. "Bridging Software-Based and Hardware-Based Fault Injection Vulnerability Detection". In: (2018). url: `https://hal.inria.fr/hal-01961008/`.

Karthik Pattabiraman et al. "SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework.". In: *IEEE Trans. Computers* 62.11 (2013), pp. 2292–2307.

Julien Proy et al. "Compiler-Assisted Loop Hardening Against Fault Attacks". In: *ACM Transactions on Architecture and Code Optimization* 14.4 (2017), p. 36.

Bibliography II

Yan Shoshitaishvili et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 138–157.

Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault attacks on secure embedded software: threats, design, and evaluation". In: *Journal of Hardware and Systems Security* 2.2 (2018), pp. 111–130.