New frontiers in Security Verification: Fuzzing and Penetration Testing

Farimah Farahmandi
Wally Rhines Endowed Professor in Hardware Security
Assistant Professor, Dept. of Electrical and Computer Engineering, UF
Associate Director, Florida Institute for Cybersecurity (FICS)

FICS Research | UF UNIVERSITY of FLORIDA

05/17/2024

Outline

1

SoC Security Verification

Definition of Security Verification, The "Verification Crisis", Challenges, Promising Solutions

2

Part 1: Fuzzy/Fuzz Testing

Background, High-Level Overview, Proposed Approaches, Results, Comparison, Summary

3

Part 2: Penetration (Pen) Testing

Background, Definition, High-Level Overview, Example Framework, Results, Summary

Outline

I

SoC Security Verification

Definition of Security Verification, The "Verification Crisis", Challenges, Promising Solutions

Part 1: Fuzzy/Fuzz Testing

Background, High-Level Overview, Proposed Approaches, Results, Comparison

Part 2: Penetration (Pen) Testing

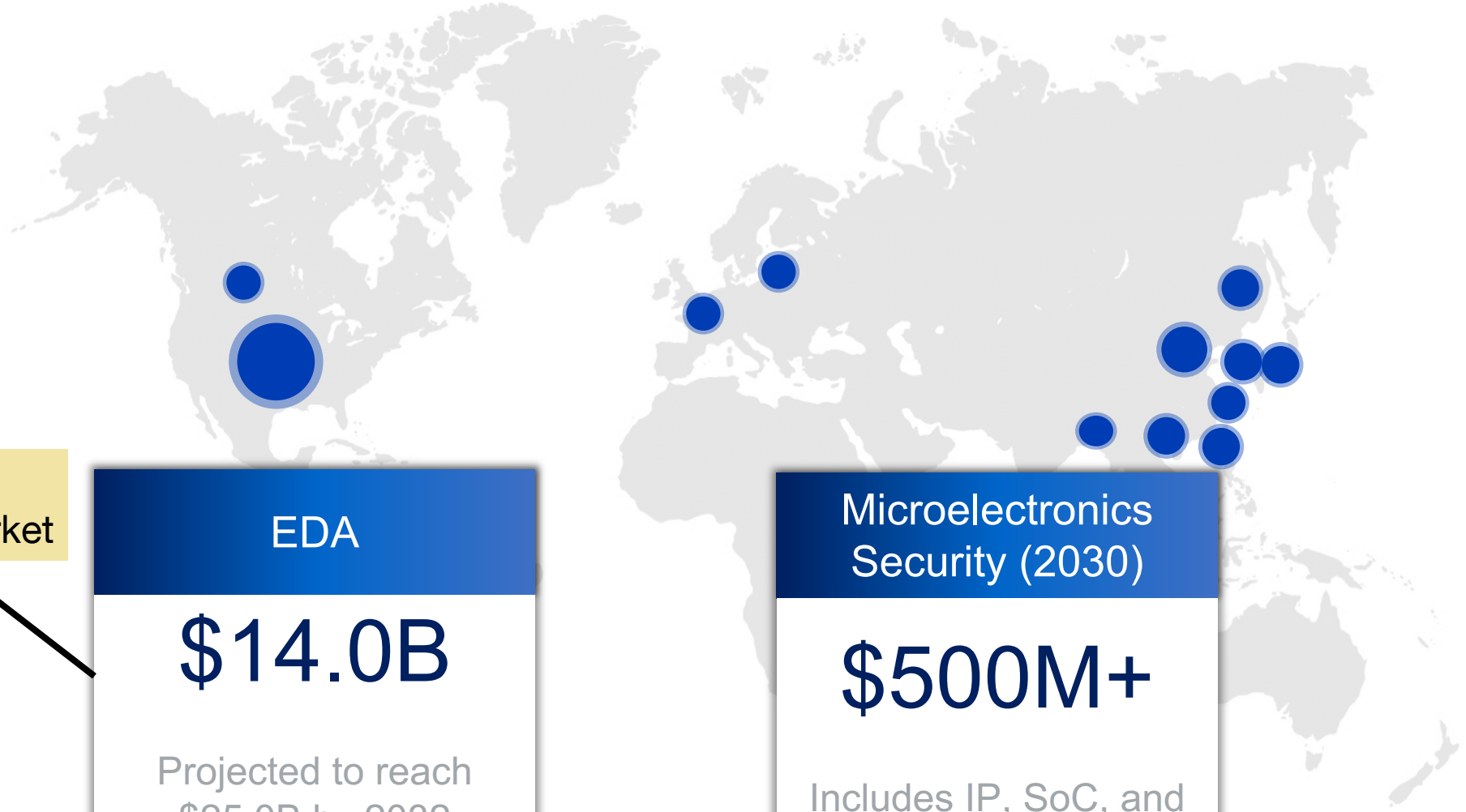
Background, Definition, High-Level Overview, Example Framework, Results, Summary

SoC Market Size

Semiconductor Market Size	SoC Market Size	PCB Market Size
$573.4B	$159.8B	$82.4B
Projected to reach $1,380.8B by 2029 (CAGR 12.2%)	Projected to reach $234B by 2028 (CAGR 8%)	Projected to reach $140.0B by 2032 (CAGR 5.6%)

SoC market value is growing thanks to AI accelerators, increased connectivity, and diversity of applications

EDA Market Size



$1B+
Verification market

EDA

$14.0B

Projected to reach
$25.0B by 2032
(CAGR 9.1%)

Microelectronics Security (2030)

$500M+

Includes IP, SoC, and
PCB security solutions

EDA solutions are growing to address the need to power and performance and move to 2.5D and 3D

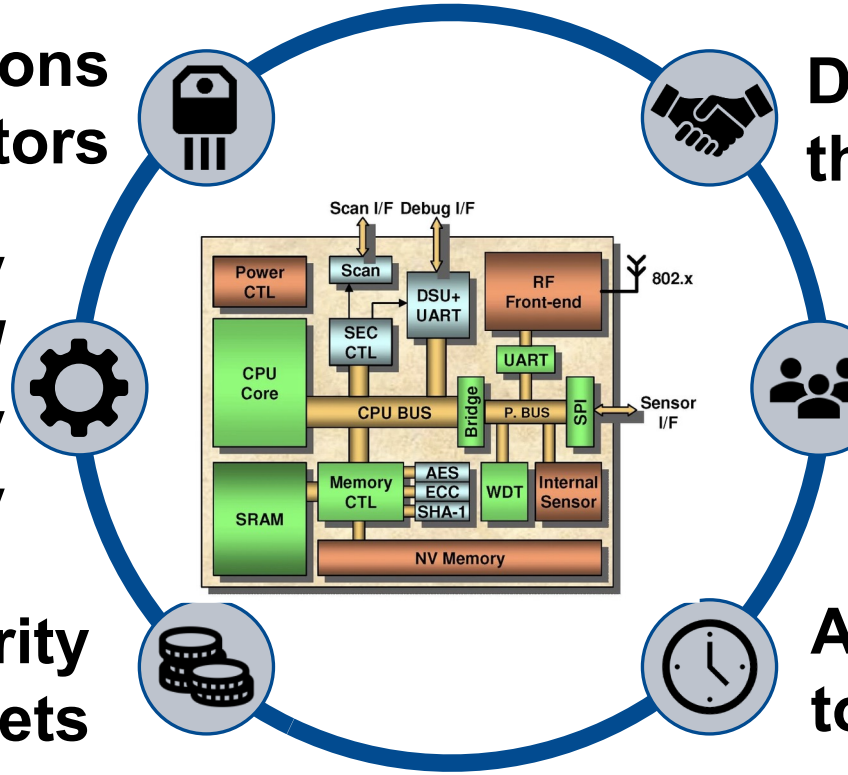
Modern SoCs: Security is a Challenge

On-chip device keys (developer/OEM)
Device configuration
Manufacturer Firmware
Application software
On-device sensitive data
Communication credentials
Random number or entropy
E-fuse,
PUF, and more…

Tens of billions transistors

Many custom/legacy/ security functionality

Many security critical assets



Designed around the globe

Tens of IPs from 3P vendors

Aggressive time-to-market

Ensuring security is a challenge

Image Source: https://novelbits.io/chipset-vs-module-bluetooth-le-solutions-the-ultimate-guide/

Growing # Vulnerabilities – Verification is a MUST

- Fault Injection
- Privilege Escalation
- Trojan Insertion
- Trace Buffer
- EM Side-Channel
- CLKSCREW
- Denial-of-Service
- Vector Rewrite
- Rowhammer
- Power Side-Channel
- Direct Memory Access
- BranchScope
- Bitstream Encryption Cracking
- Plundervolt
- Access Control
- Meltdown and Spectre
- Machine Learning
- Information Leakage
- Trusted Execution Environment Breaking
- Reset and Flush
- Branch Shadowing
- Bitstream Tampering
- Reverse Engineering
- Timing Side-Channel
- Integrity

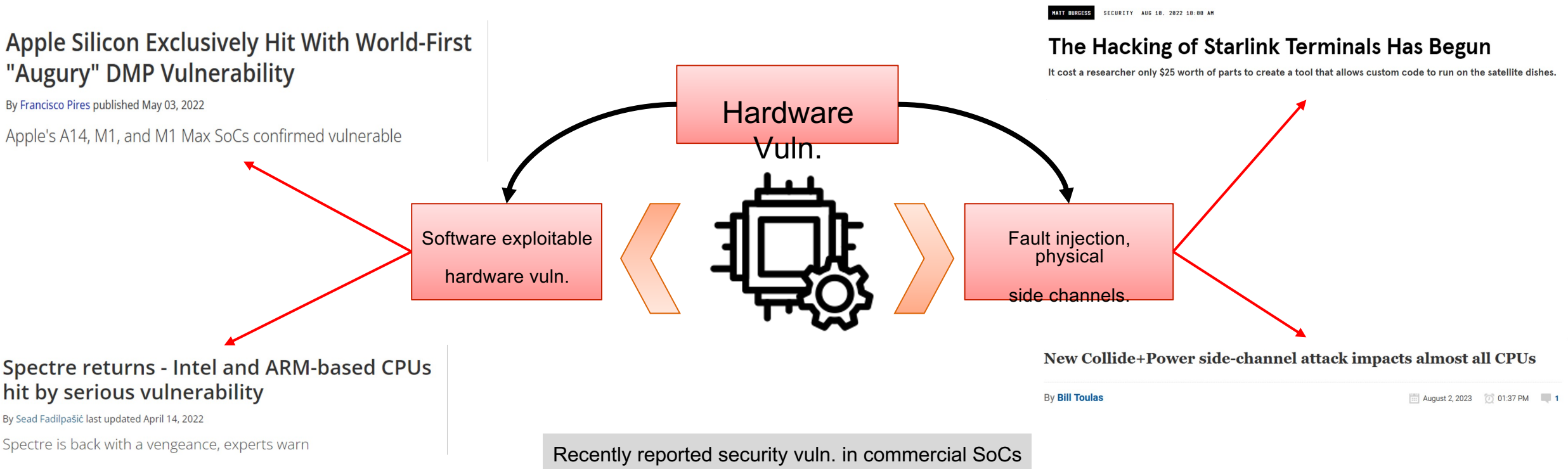
Strong Algorithm & Architecture

Weak Implementation & Execution

Security: An Indispensable Aspect of Verification

- **In addition to functional, performance requirements, security requirements have recently emerged important concerns:**
  - Lack of understanding of security vulnerabilities by designers
  - Recent reports of critical hardware security vulnerabilities in commercial products

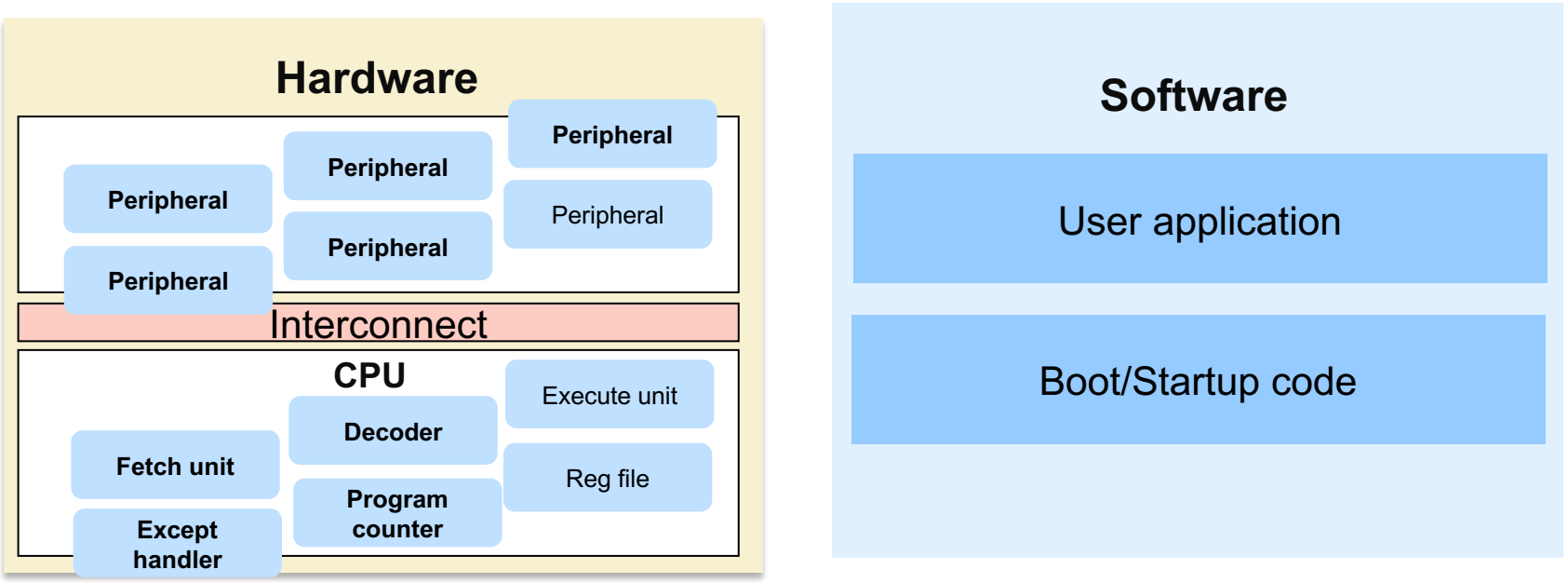
Information leakage, μ-architectural side channels, fault injection, hardware trojans, etc.…



Recently reported security vuln. in commercial SoCs

Need for Dynamic Verification: A Case Study

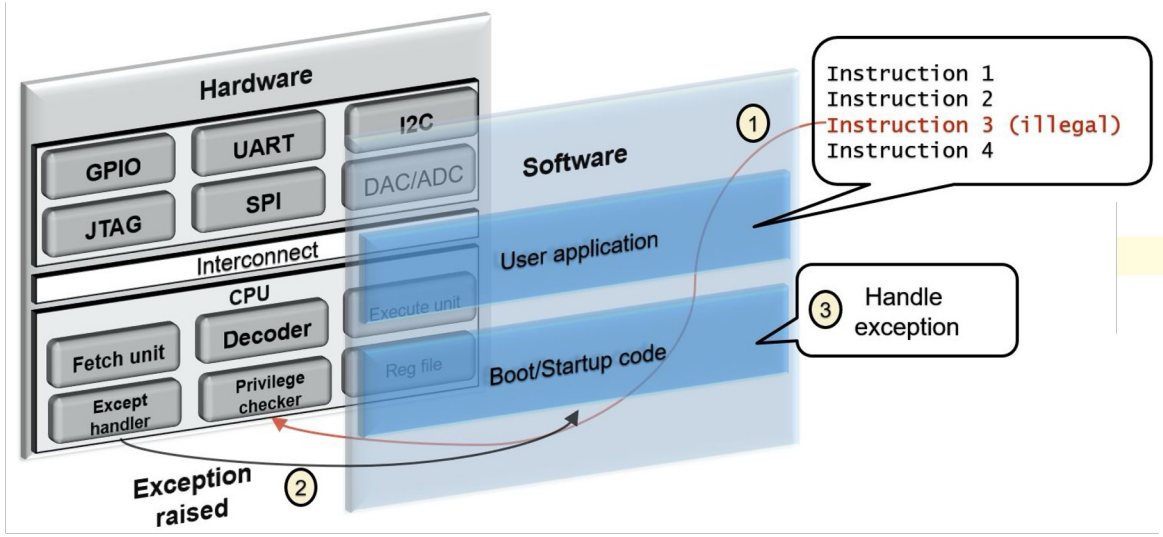
- In a RISC-V SoC
  - There may be up to 3 privilege levels: **M, S** and **U**
  - Privileged access is protected through a complex interplay between hardware and software



Assume: M and U-modes are implemented

Need for Dynamic Verification: A Case Study

- In a RISC-V SoC
  - When an illegal access is detected, hardware raises an ***illegal instruction exception***
  - The hardware program counter will jump to address stored in register ***mtvec***
    → ***mtvec*** needs to be configured properly
  - Software trap handler code will handle the response
- **Hardware-software interplay is indispensable for protection of assets.**




```
la   x1,      __crt0_trap_handler // configure early-boot trap handler
csrw mtvec,   x1
csrw mie,     zero               // disable all interrupt sources
```

Assume: M and U-modes are implemented, baremetal scenario

Alibaba's T-Head C910 RISC-V, April 2024

- German Center for Information Security has **found serious security flaws** in Alibaba's T-Head Semiconductors containing RISC-V processors
- T-Head 4 core, **TH1520 SoC** now dubbed "GhostWrite", allowing hackers access to **read and write physical memory** and **execute arbitrary code**
- Bad Actors can take over the SoC and hijack the host, the vulnerability lies in **faulty instructions in the RTL chip design**
- Given the instructions are "baked" into the design, silicon, it **cannot be fixed with a Software update**

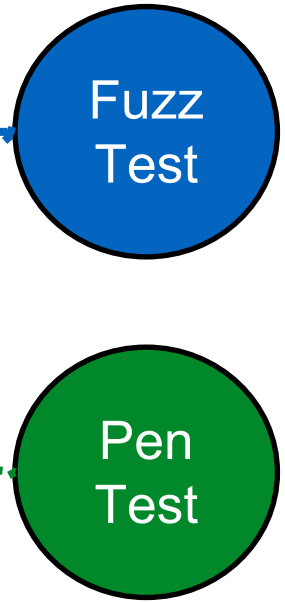
Comparison of Traditional Methods

Method	Model	Limitation	Scalability	Automation
Formal property verification	White-box	False positive, state explosion, low accuracy	Low	Low
Symbolic, Concolic	White-box	Path explosion, low accuracy	Low	Moderate
Genetic Algorithms	White-box	Reliance on structural features	Low	Moderate
ML	White-box	Reliance on structural features	High	Moderate

- **Traditional methods presuppose white-box knowledge**
  - A major pitfall given today's SoC supply chain context
- **Vast majority of traditional HW verification methods are not dynamic**
  - Unable to consider HW-SW interactions

Promising Solutions: Test Generation for Security

- To keep up with increasing design complexity
- To deal with realities of the supply chain
- To detect vulnerabilities and desired coverage quickly
- To guide pattern generation effectively through feedback
- To maximize coverage
- To account for HW-SW interactions



Outline

SoC Security Verification

Definition of Security Verification, The "Verification Crisis", Challenges, Promising Solutions

2

Part 1: Fuzzy/Fuzz Testing

Background, High-Level Overview, Proposed Approaches, Results, Comparison, Summary

Part 2: Penetration (Pen) Testing

Background, Definition, High-Level Overview, Example Framework, Results, Summary

Evolutionary Fuzzing Approach

An automatic testing technique

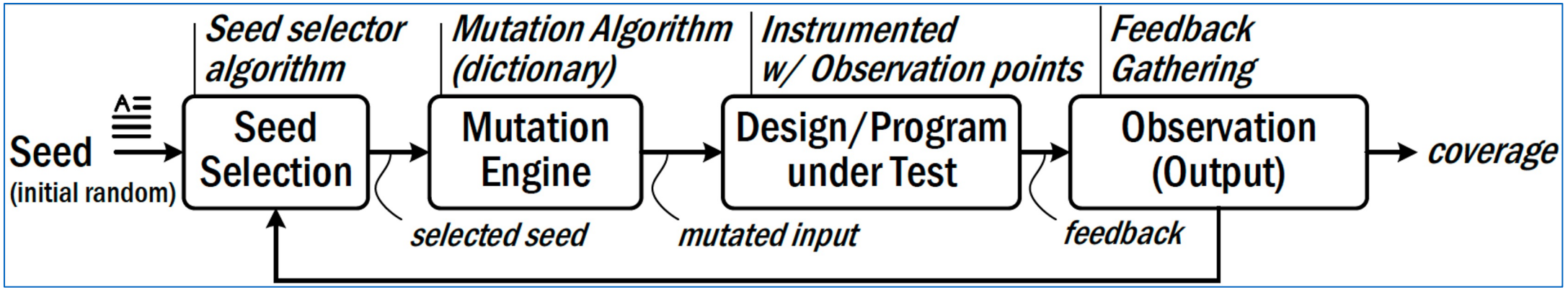
Uses invalid/semi-valid/valid data as application input

Targets numerous boundary/corner cases

Used to generate and feed the target program with plenty of test cases to trigger bugs

Ensures the absence of exploitable vulnerabilities

Widely used in software domain for bug detection



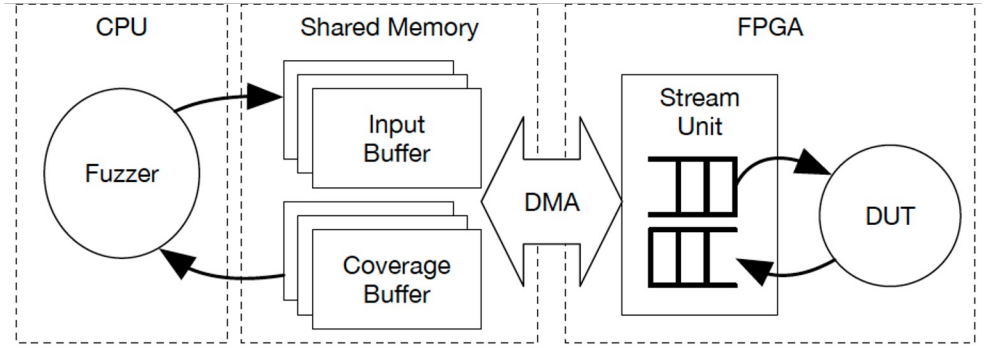
Typical Fuzzing Technique Overview

Current Fuzzing Practices for Hardware Verification

Direct Fuzzing on Hardware

- Apply mutated inputs to HW directly
  - Software simulation
  - FPGA based emulation
- Advantages
  - Faster (emulation) and scalable
  - Effective for HW/SW co-layer verification
- Disadvantages
  - Limited permissions in some low-level

Ref. Laeufer, Kevin, et al. "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs." *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2018.

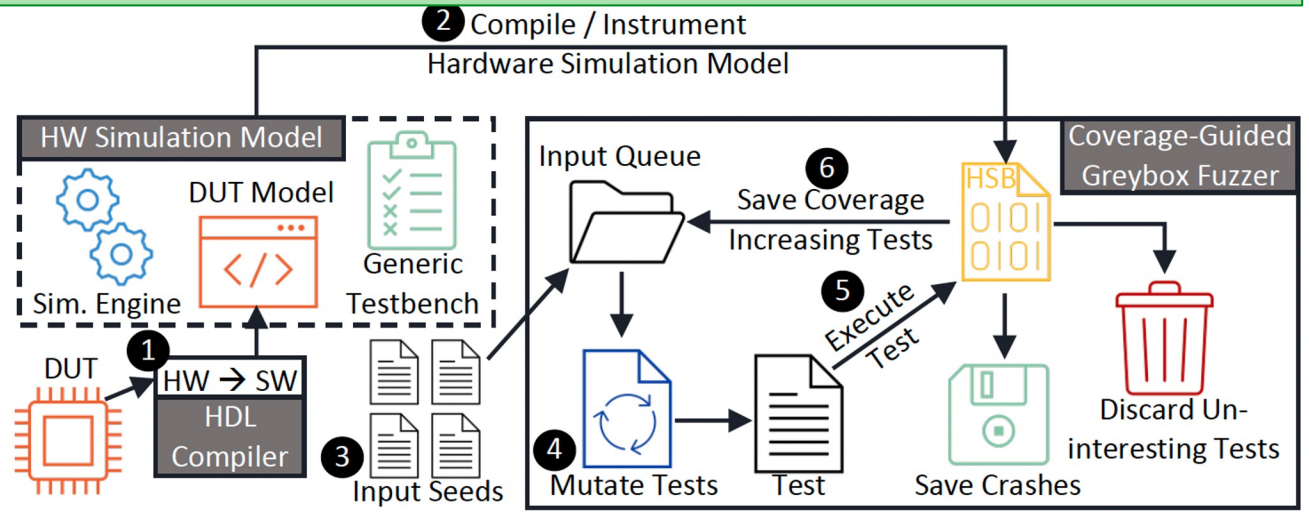


Fuzzer w/ FPGA-accelerated Simulation

Motivation: Utilizing a real-time HW Emulation Platform for Fuzzing

Fuzzing Hardware as Software

- HW RTL □ SW model by Verilator
- Fuzz SW model
- Crashes □ Vul. Detection
- Advantages
  - Less time required for initial preparation
- Disadvantages
  - Emerging vulnerabilities due to translation



Ref. Trippel, Timothy, et al. "Fuzzing hardware like software." *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

Experiment: RTL Code with Vulnerability

```
1  module locked_register_example
2  (
3      input [15:0] Data_in,
4      input Clk,
5      input resetn,
6      input write,
7      input Lock,
8      input scan_mode,
9      input debug_unlocked,
10     output reg [15:0] Data_out
11 );
12
13 reg lock_status;
14
15 always @(posedge Clk or negedge resetn)
16 begin
17     if (~resetn) // Register is reset resetn
18     begin
19         lock_status <= 1'b0;
20     end
21     else if (Lock)
22     begin
23         lock_status <= 1'b1;
24     end
25     else if (~Lock)
26     begin
27         lock_status <= lock_status;
28     end
29 end
```

CWE-1234: HW Internal or Debug Modes Allow Override of Locks

```
31 always @(posedge Clk or negedge resetn)
32 begin
33     if (~resetn) // Register is reset resetn
34     begin
35         Data_out <= 16'h0003;
36     end
37     else if (write & (~lock_status | scan_mode | debug_unlocked) )
38     // Register protected by Lock bit input, overrides supported
39         //for scan_mode & debug_unlocked
40     begin
41         Data_out <= Data_in;
42     end
43     else if (~write)
44     begin
45         Data_out <= Data_out;
46     end
47 end
48 endmodule
```

When lock bit is set, memory overwrite by enabling scan or debug mode

Experiment: Security Property

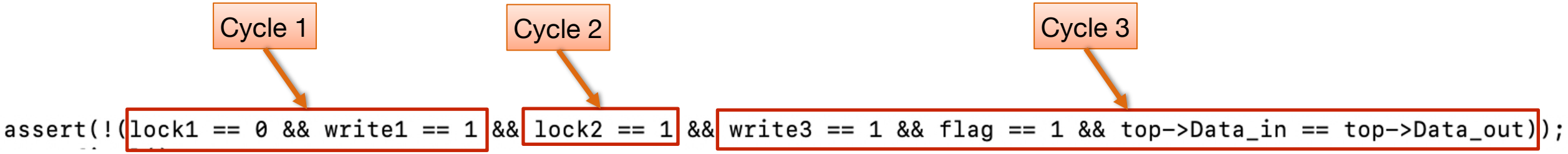
Assume: Data write enable bit is never been set in any previous cycle

Data write happens and lock bit set set in a cycle

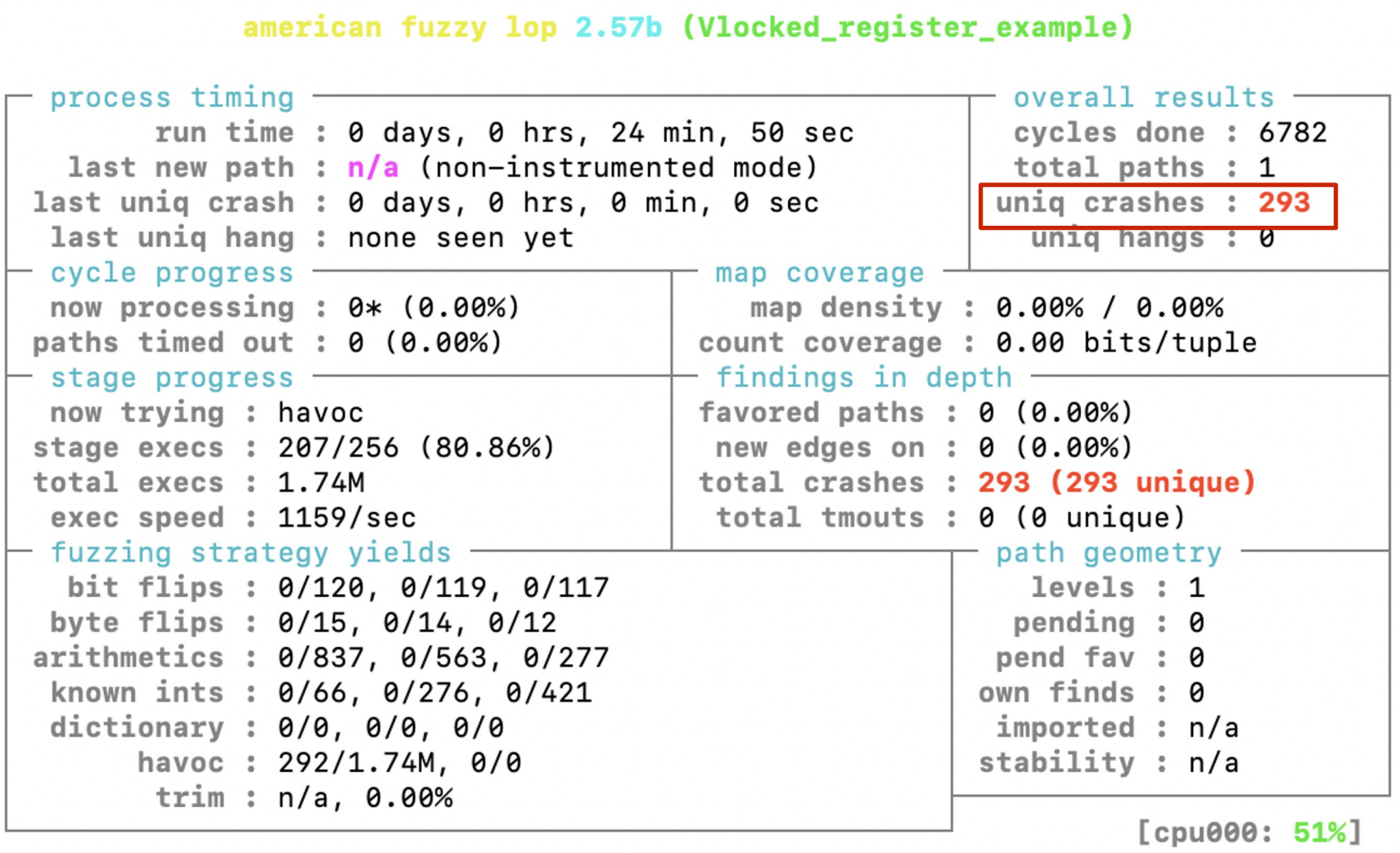
Provided written data and data intended to be written next are not same

Check intended data to be written in figure goes to the output port

Assertion failure happens!!!!



Experiment: Fuzzing Result



293 Unique crashes identified in 24 minutes!

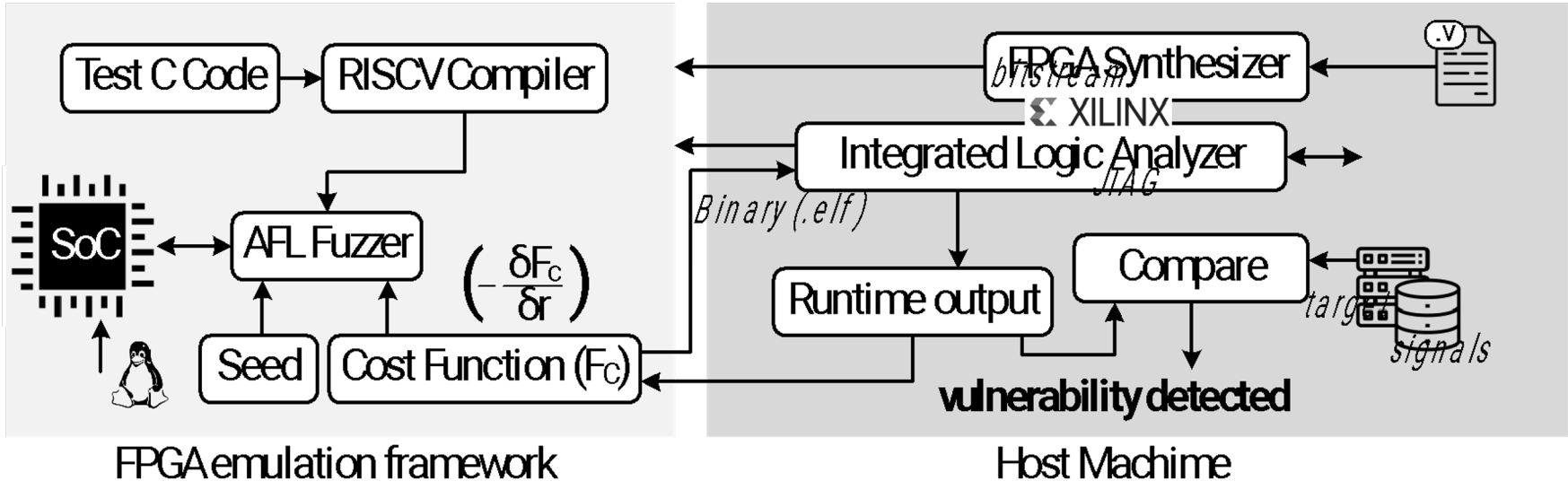
Our Proposed Framework

SoCFuzzer

Proposed Direct Fuzzing Framework: SoCFuzzer

SoCFuzzer SoC Vulnerability Detection using Cost Function enabled Fuzz Testing

- Scalable and automated framework for SoC security verification
- Develop evaluation metrics, cost function, and feedback for runtime update of mutation strategies
- Global minima of cost function □ Vulnerability detection
- Implementation □ Emulation board: Genesys 2 Kintex-7 FPGA Development Board, SoC: 64-bit RISC-V Ariane
  - HW debugging: Xilinx Integrated Logic Analyzer (ILA), Monitoring: JTAG, OS: Linux, Mutation Engine: AFL Fuzzer



Ref. Hossain, Muhammad Monir, et al. "SoCFuzzer: SoC Vulnerability Detection using Cost Function enabled Fuzz Testing." *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023.

SoCFuzzer Evaluation Metrics

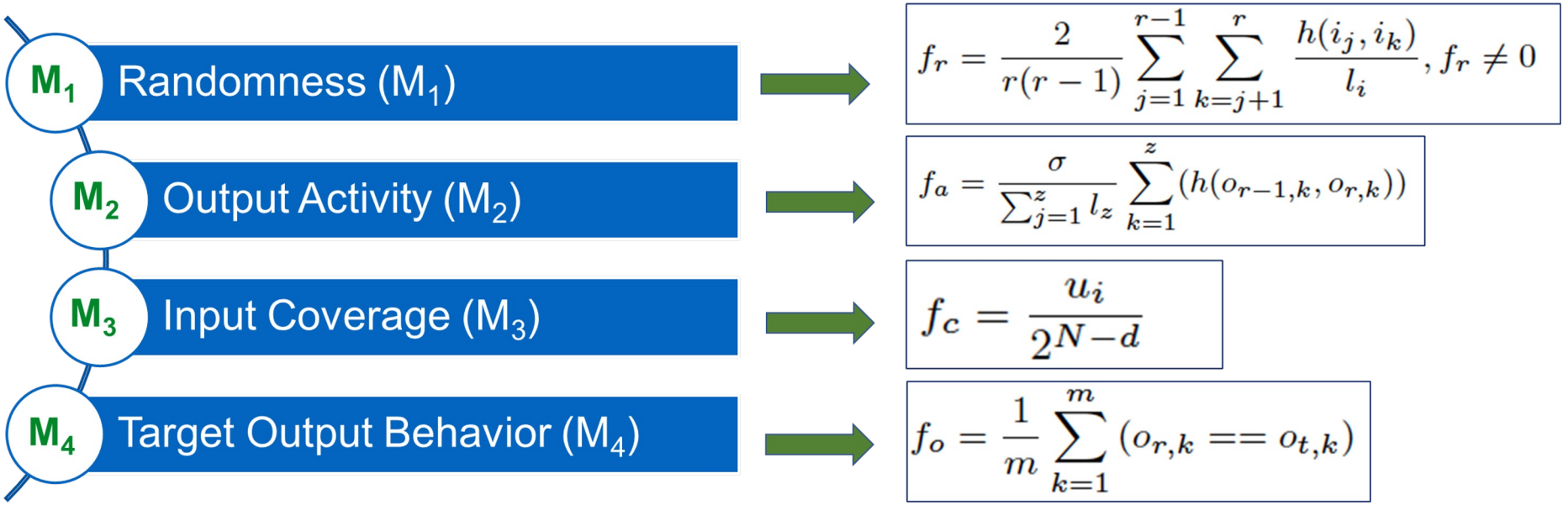
Objective: guides fuzzing to generate smarter-than-random test inputs based on metrics

- Few metrics based on security properties

Evaluate the quality of mutated inputs

Estimate the chances of hitting a potential malicious behavior

Faster convergence by a feedback utilizing metrics □ faster triggering the vulnerability



$$f_r = \frac{2}{r(r-1)} \sum_{j=1}^{r-1} \sum_{k=j+1}^{r} \frac{h(i_j, i_k)}{l_i}, f_r \neq 0$$

$$f_a = \frac{\sigma}{\sum_{j=1}^{z} l_z} \sum_{k=1}^{z} (h(o_{r-1,k}, o_{r,k}))$$

$$f_c = \frac{u_i}{2^{N-d}}$$

$$f_o = \frac{1}{m} \sum_{k=1}^{m} (o_{r,k} == o_{t,k})$$

SoCFuzzer Cost Function and Feedback

Cost Function

$$F_c = 1 - \frac{(1 - f_r) + f_a + f_c + f_o}{n} = \frac{n-1}{n} - \frac{1}{n}(f_a + f_c + f_o - f_r)$$

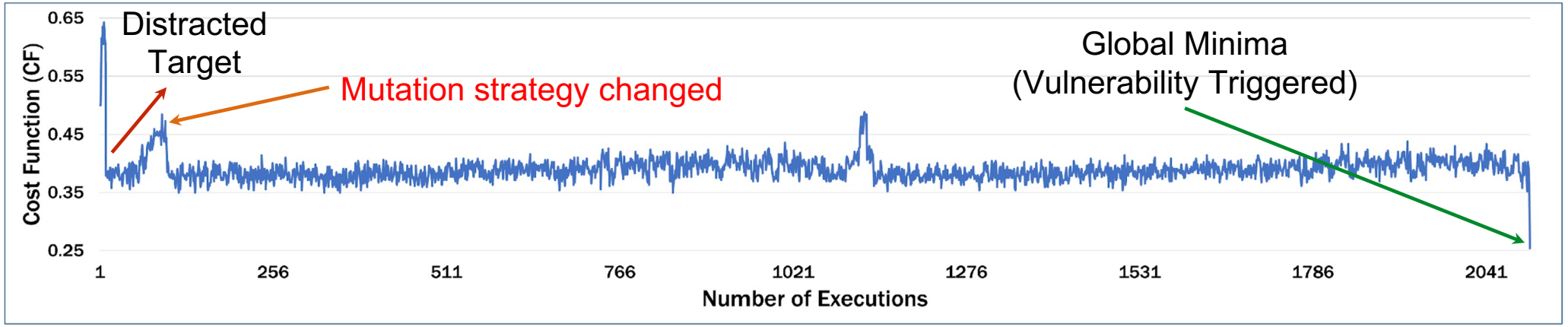
- Developed based on proposed fuzzing evaluation metrics, M1, M2, M3, and M4
- Fuzzing objective: minimizing cost function (global minima □ vulnerability triggered)
- ↑ Metrics □ better fuzzing □ faster vulnerability trigger (global minima)

Feedback

$$CFIR = -\frac{\delta F_c}{\delta r} = -\frac{F_{c2} - F_{c1}}{r_2 - r_1}$$

- Positive CFIR □ better fuzzing □ Continue mutation w/ the same mutation strategy
- Negative CFIR □ Mutation against objective □ Change the mutation strategy
- CFIR estimated after each frequency of feedback generation ($FREQ_{fb}$) iterations

Cost Function & Feedback Realization



Experimental Setup

Vulnerabilities in Ariane SoCs

Index	Vulnerability	Location	Triggering Condition	Output Behavior	Ref.
SV1	Leaking the AES secret key through the common bus in the SoC	AES IP	Specific plaintext	AES Key leaks to PO	CVE-2018-8922
SV2	A Trojan injects a delay in the AES IP in cipher conversion	AES IP	Specific plaintext	Ciphertext not resulted in time	AES-T500
SV3	Incorrect implementation of logic to detect the FENCE.I instruction	CPU (dec)	*imm* ≠ 0 & *rs1* ≠ 0	Illegal instr exception raised	CWE-440
SV4	Execute machine-level instructions from user mode	CPU (dec)	Execute ”mret” instruction	No exception exhibited	CWE-1242
SV5	Access to CSRs from lower privilege level	Register file	*mstatus_reg* r/w from user space	No exception exhibited	CWE-1262

Cost Function Development

Index	σ	Input		Output		Effective Length (*N-d*)
		Data	Length	Signal	Length	
SV1	1	Plaintext	128 bits	Ciphertext	128 bits	128 bits
SV2	1	Plaintext	128 bits	Control Register	32 bits	128 bits
SV3	0	Instruction	32 bits	Exception Raised	N/A	32 bits
SV4	0	Instruction	32 bits	No Exception	N/A	32 bits
SV5	0	CSR instruction	32 bits	No Exception	N/A	12 bits

Example Cost Function for SV1

$$F_{c,SV1} = \frac{n-1}{n} - \frac{1}{n}\Big[\frac{\sigma}{\sum_{j=1}^{z} l_z}\sum_{k=1}^{z}(h(C_{r-1,k}, C_{r,k})) + \frac{u_i}{2^{N-d}}$$

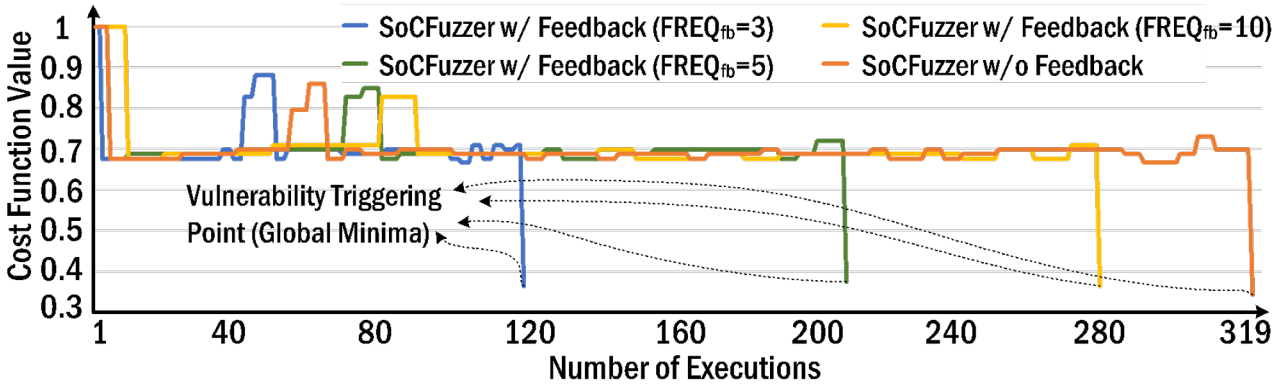
$$+\frac{1}{m}\sum_{k=1}^{m}(C_{r,k} == AES_{key}) - \frac{2}{r(r-1)}\sum_{j=1}^{r-1}\sum_{k=j+1}^{r}\frac{h(P_j, P_k)}{l_P}\Big]$$

Results Analysis Contd…

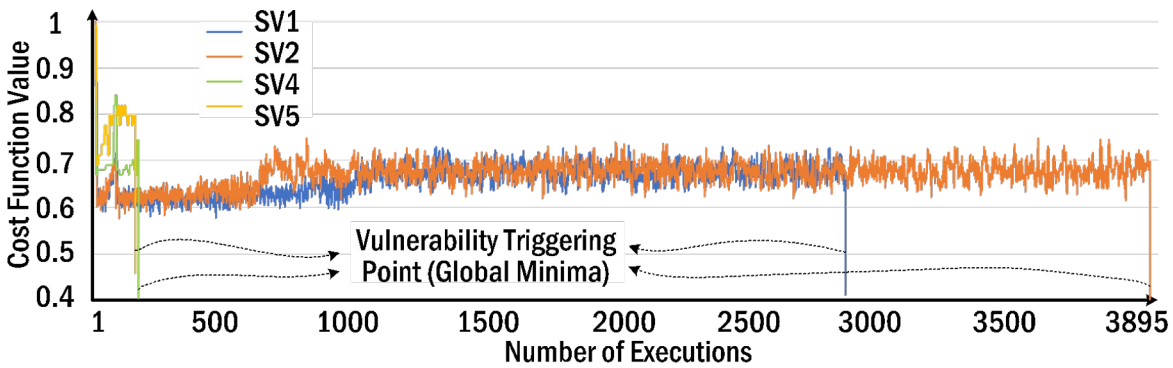
SoCFuzzer with the feedback CFIR outperforms the conventional fuzzing without our proposed feedback

An optimum value required for $FREQ_{fb,}$ a low value insufficient samples to evaluate, high value too much time with an inefficient mutation strategy

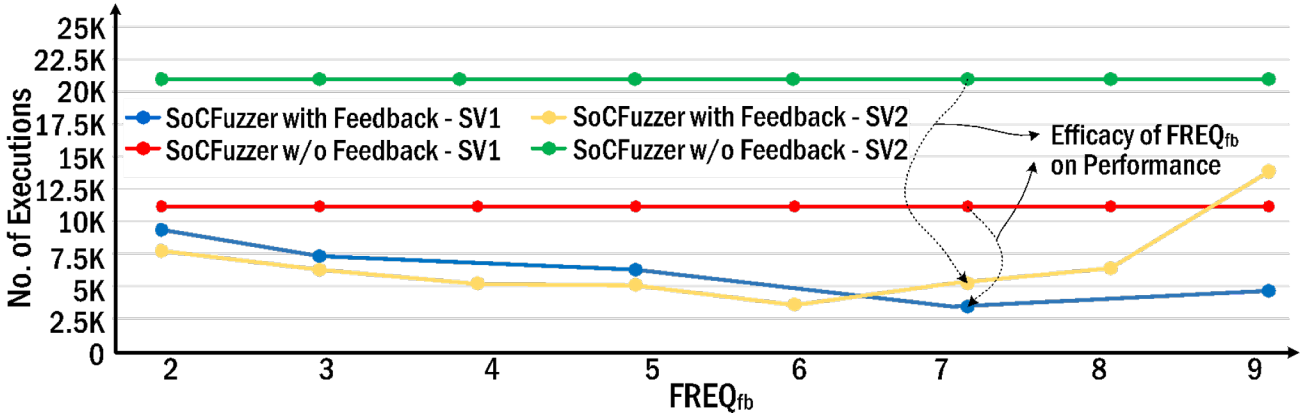
For all quality of seeds, SoCFuzzer with the proposed feedback shows excellency in faster triggering vul.



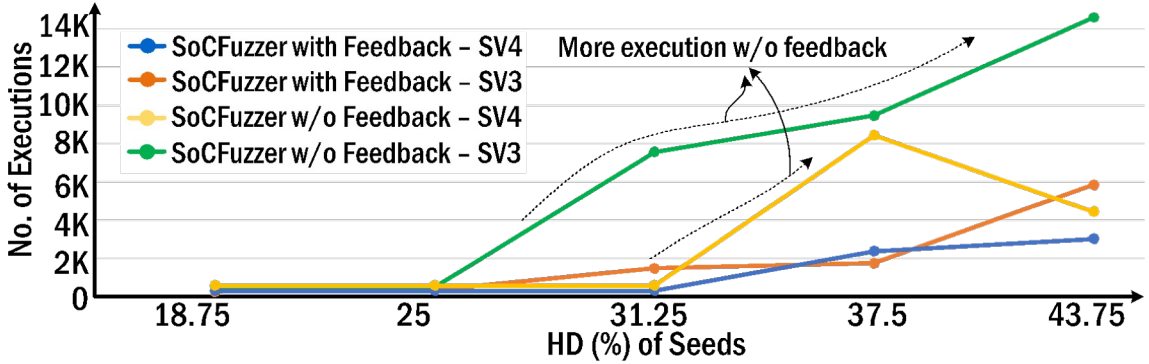
Cost Function and Feedback Analysis in Detecting Vulnerability (SV3)



Cost Function Analysis in Detecting SV1, SV2, SV4, and SV5



Performance of SoCFuzzer with Feedback for Variation in FREQfb



Performance of SoCFuzzer for Various Quality Seeds

Results Analysis

Summary Results: Vulnerability Detection

For optimum $FREQ_{fb}$, SoCFuzzer detects all vul. in less time w/ variety quality of seeds

SoCFuzzer saved at least 50% of verification time

Index	HD(seed, VTI)	$FREQ_{fb}$	No. of Executions		Speed up
			Fuzzing w/o CF_FB	SoCFuzzer	
SV1	62.50%	7	10968	2862	73.91%
SV2	62.50%	6	21319	2999	85.93%
SV3	25%	6	361	180	50.14%
SV4	43.75%	5	415	162	60.96%
SV5	66.67%	6	968	275	71.59%

HD(seed, VTI): HD of Seed and Vulnerability Triggering Input CF_FB: Cost Function enabled Feedback $FREQ_{fb}$: Optimum $FREQ_{fb}$

Compare w/ HW Fuzzing Approach

Index	Fuzzer Engine	Frame-work	Target	Feedback to Mutation Engine	Golden Model
RFUZZ	HW Fuzzer	FPGA Emu	IP Designs	MUX coverage	Yes
DifuzzRTL	HW Fuzzer	FPGA	CPU Design	Control-register Code coverage	Yes
Hyperfuzzing	SW Fuzzer	SW Sim	SoC Design	NoC, Instruction and Bitflip Monitors	Yes
TheHuzz	HW Fuzzer	HDL Sim	CPU Design	Statement, toggle, branch expression, condition, FSM	Yes
SoCFuzzer	Modified HW Fuzzer using CF and Metrics	FPGA	SoC Design	Cost Function based (vulnerability-oriented)	No

Improvement Scopes in SoCFuzzer

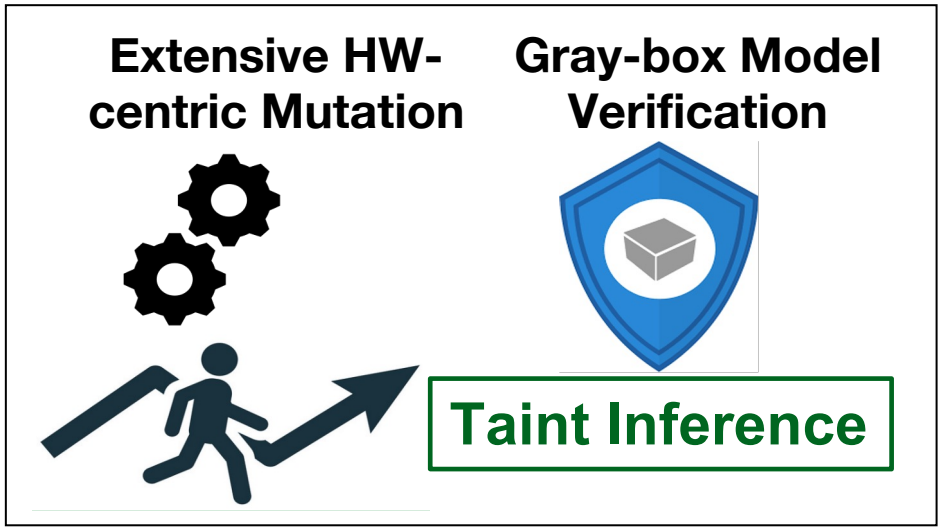
Efficient Seeds for HW-oriented Fuzzing

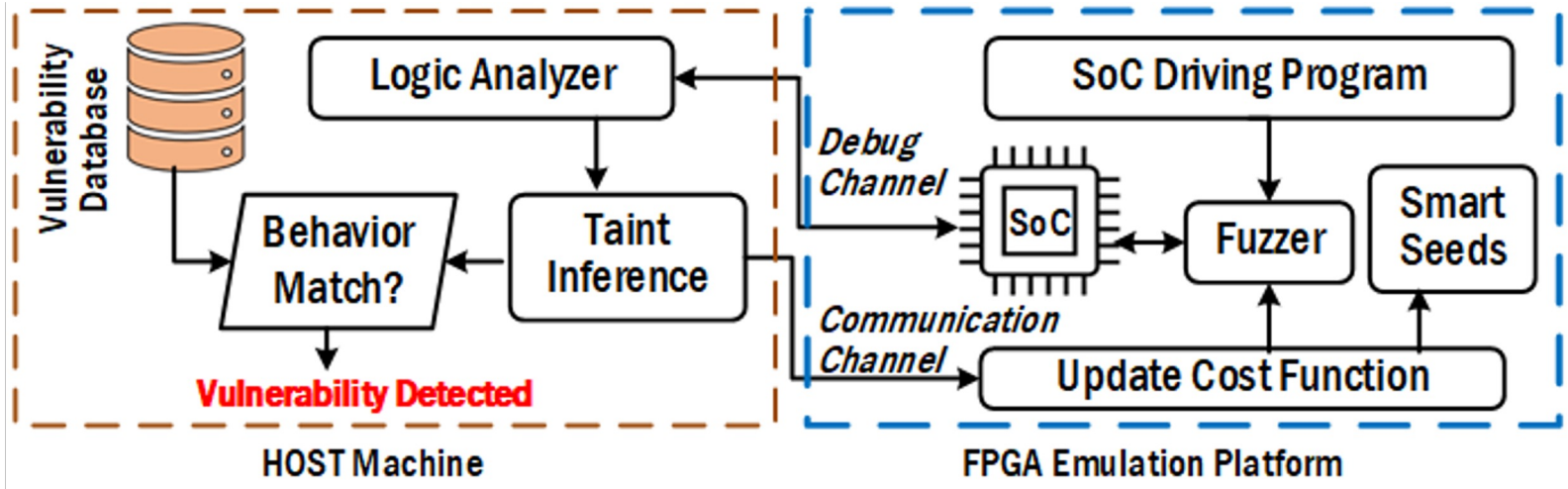
- Optimization and selection of seed(s)
- Potentially less dependency on initial seed

Taint Inference-enabled Fuzzing

Proposed Fuzzing Framework: TaintFuzzer

- Scalable and automated framework for SoC security verification
- Leverage taint propagation for more HW-oriented fuzzing (mutation)
- Leverage taint inference for generating smart seeds from initials
- Utilize taint inference in cost function and feedback development for efficiency in HW fuzzing
- Proposed cost function and feedback for runtime update of mutation strategies





Smart Seeds Evolution, Cost Function and Feedback

Smart Seeds Evolution

- Definition: smart seeds expedite mutations in triggering vulnerability
- Challenging to select an initial seed: Can't be selected in a trivial process or arbitrarily
- Inefficient seeds □ increased verification time
- Smart seeds derived based on proposed metric *HW Seed Impact Factor* (I_s)
- Randomly mutated inputs from initial with seed w/ higher Is □ smart seeds
- Is considers impact on HW behavior and proximity of malicious behavior
- Smart seeds used for longer deterministic mutation (efficient)

HW Seed Impact Factor

$$I_s = \frac{1}{l_i \times z} \sum_{k=1}^{l_i} W_{S[k]} + \{1 - \frac{1}{m} \sum_{k=1}^{m} h(o_{r,k}, o_{t,k})\}$$

Cost Function

$$F_c = 1 - \frac{1}{4} \times [\frac{n_t}{z} + \{1 - \frac{2}{r(r-1)} \sum_{j=1}^{r-1} \sum_{k=j+1}^{r} h(S_j, S_k)\} + \{1 - \frac{1}{m} \sum_{k=1}^{m} h(o_{r,k}, o_{t,k})\} + \frac{n_u}{2^N}]$$

Feedback

$$CDCF = -\sum_{k=2}^{f_f} (F_{c_k} - F_{c_{k-1}})$$

Results Analysis

Summary Results: Vulnerability Detection

- Only few iterations required to mutate smart seeds (~6x to ~12x)
- Saved at least 32% of verification time compared to fuzzing w/o proposed feedback
- Saved 11.68% verification time compared to SoCFuzzer framework

Index	#SS	f_f	Iterations for Smart Seeds	No. of Iterations			Reduction of Verification Time w.r.t.	
				TF_{NF}	*SoCFuzzer*	TF_{CDCF}	TF_{NF}	*SoCFuzzer*
SV1	10	5	79	1610	784	629	56.02%	9.69%
SV2	10	4	64	808	269	197	67.70%	2.97%
SV3	10	6	88	4789	2548	1945	57.55%	20.21%
SV4	5	5	34	454	364	272	32.56%	15.93%
SV5	5	5	31	413	237	181	48.69%	10.54%
SV6	20	7	243	13311	6389	5261	58.65%	13.85%
SV7	30	5	315	6063	3186	2597	51.97%	8.60%
USV1	20	7	225	N/A	N/A	10783	N/A	N/A
USV2	30	5	327	N/A	N/A	7311	N/A	N/A
USV3	10	5	83	N/A	N/A	1147	N/A	N/A

#SS: No. of Smart Seeds f_f: Frequency of Feedback Evaluation.
TF_{NF}: TaintFuzzer w/o proposed Feedback TF_{CDCF}: TaintFuzzer w/ CDCF.

TaintFuzzer also detected few unknown vulnerabilities in the Ariane SoC

Index	Vulnerability	Location	Triggering Condition	Output Behavior
USV1	Allow retrieving last ciphertext of encryption operation for AES module in SoC	AES IP	Specific plaintext	Read last ciphertext
USV2	Release intermediate result of encryption as final ciphertext	AES IP	*Specific plaintext*	Intermediate ciphertext
USV3	No exception while executing an illegal instruction (MULH)	CPU (Dec)	(*rd = rs1* $\vee$ *rd = rs2*)	No exception raised

Improvement Scopes in Prior Frameworks

When to evaluate a new feedback?

- Auto tuning of frequency of feedback evaluation

SoCFuzzer+ Implementation

Reinforcement learning guided emulation based framework developed based on SoCFuzzer

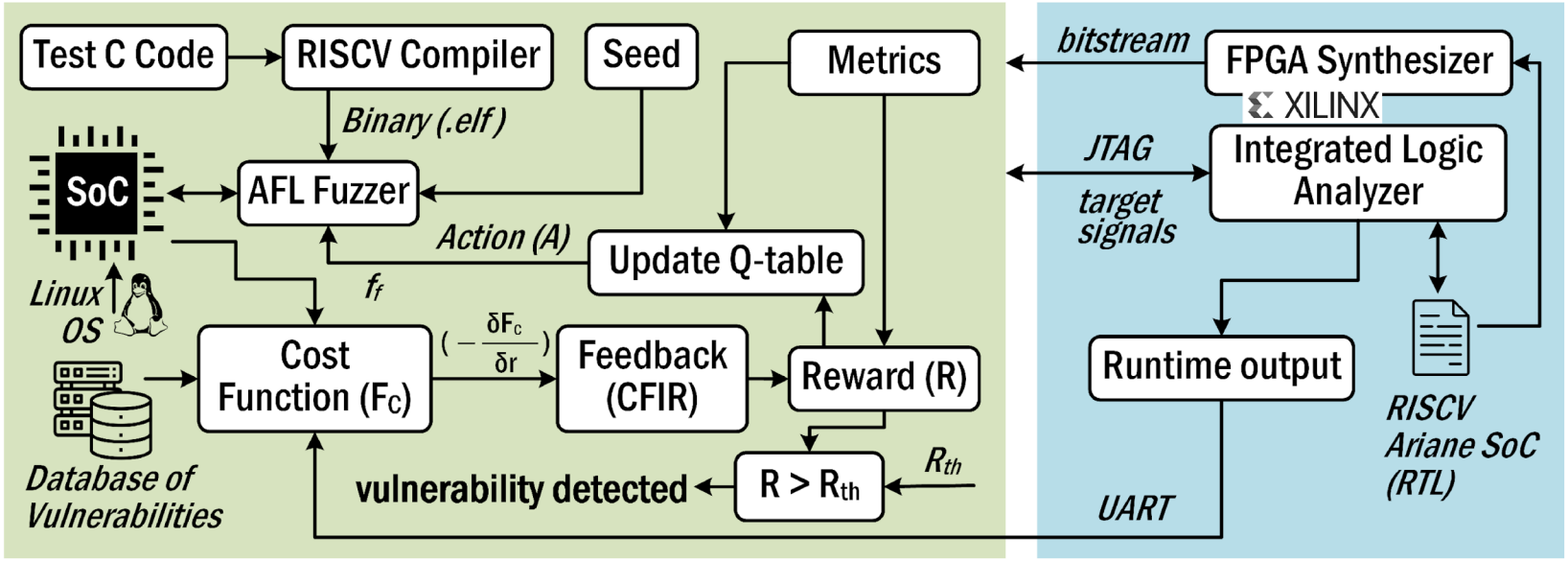
RL utilizes cost function-based feedback to tune mutation strategies and when getting feedback

Increasing cost function RL penalty, decreasing cost function RL reward

Vulnerability trigger: Global minima (based on properties) or local minima of cost function

SoC: RISC-V-based 64-bit Ariane SoC, HW debugging: Xilinx Integrated Logic Analyzer (ILA)

Emulation board: Genesys 2 Kintex-7 FPGA Development Board



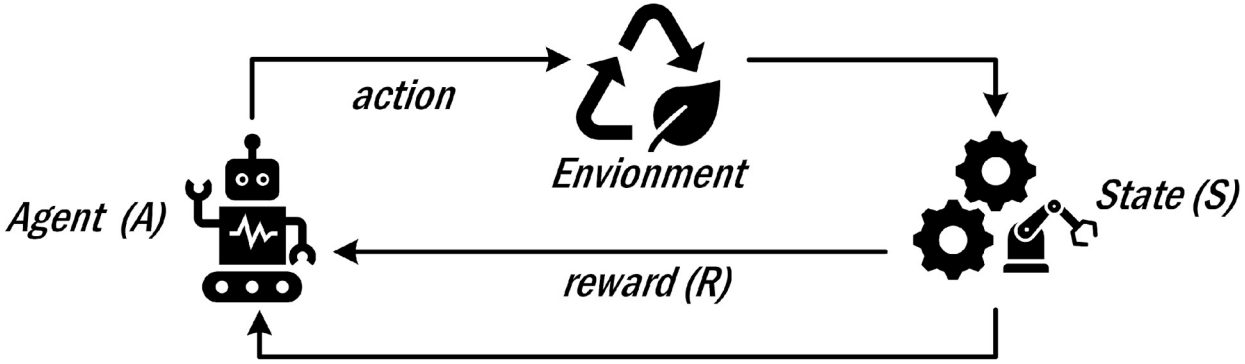
SoCFuzzer+ AI Model

RL Learning

An agent learns to interact w/ environment to maximize cumulative reward

Training agent to make action towards a goal

Action results in feedback as reward or penalty



State Set

$\mathcal{S}_r : \{\Delta m_1, \Delta m_2, \Delta m_2, \Delta m_2\}$

$\mathcal{S} = \{\mathcal{S}_{f_{min}}, ..., \mathcal{S}_i, \mathcal{S}_{i+1}, ..., \mathcal{S}_{f_{max}}\}$

$f_{min} \leq i \leq f_{max}$

Change in metric values define the state

A state represents a frequency of feedback evaluation and a mutation strategy

$\mathcal{S}_i = \{(\mathcal{S}_i, 1), (\mathcal{S}_i, 2), ..., (\mathcal{S}_i, j)\}$

Action Set

$\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, ...\mathcal{A}_m\}$

$\mathcal{A}_m = \{a_m, f_{f_i}\}$

$f_{f_{min}} \leq f_{f_i} \leq f_{f_{max}}$

An action set is chosen from a particular state for a particular number of iterations in fuzzing

Action set consists of a particular mutation strategy and frequency of feedback evaluation

SoCFuzzer+ AI Model

State Transition Function

$$N_{States} = \frac{2}{\Delta t} \times \frac{2}{\Delta t} \times \frac{2}{\Delta t} \times \frac{2}{\Delta t} = \frac{16}{(\Delta t)^4}$$

$$N_t = \binom{N_{States}}{2}$$

Determines how likely an against move from one to another state

Implement stochastic state transition for a new module, i.e., initially randomized Q-table

Utilize mature Q-table over time for more deterministic state transition

Action chosen best on the highest reward in Q-table for the current state

Agent Rewards

Provided based on cost function improvement rate (CFIR)

+ve CFIR □ Reward, Max reward □ CF minima

Negative CFIR □ Penalty (negative reward)

$$\mathcal{R} = \begin{cases} -r_1, & CFIR < 0 \\ r_2, & CFIR = 0 \vee \Delta CFIR \approx 0 \\ r_3, & CFIR > 0, r_3 > r_2 \\ T_r, & M4 = 0 \wedge CFIR = 0 \end{cases}$$

Q-Table

Q-table updated after certain iterations

α □ Learning rate

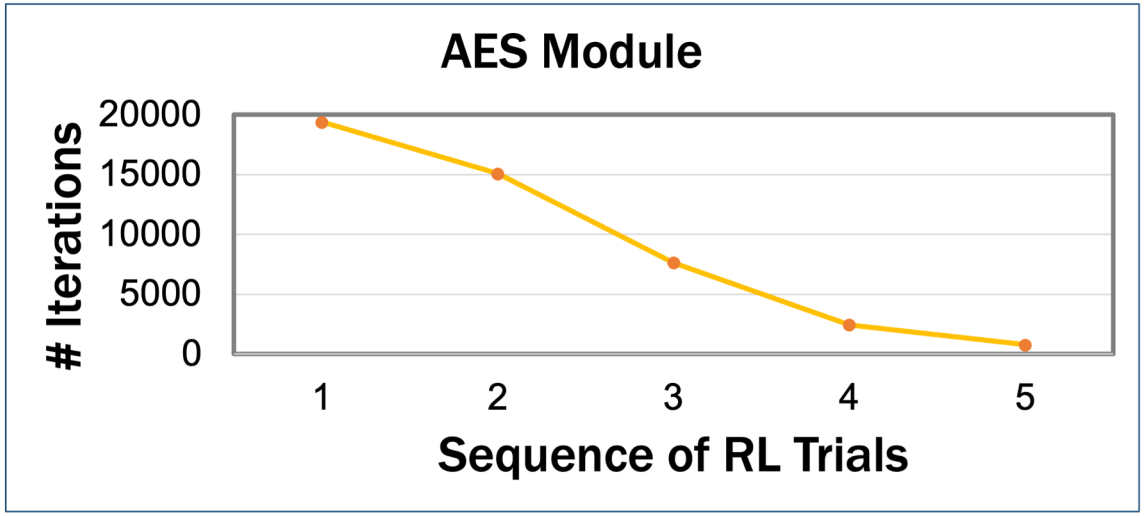
γ □ discount factor

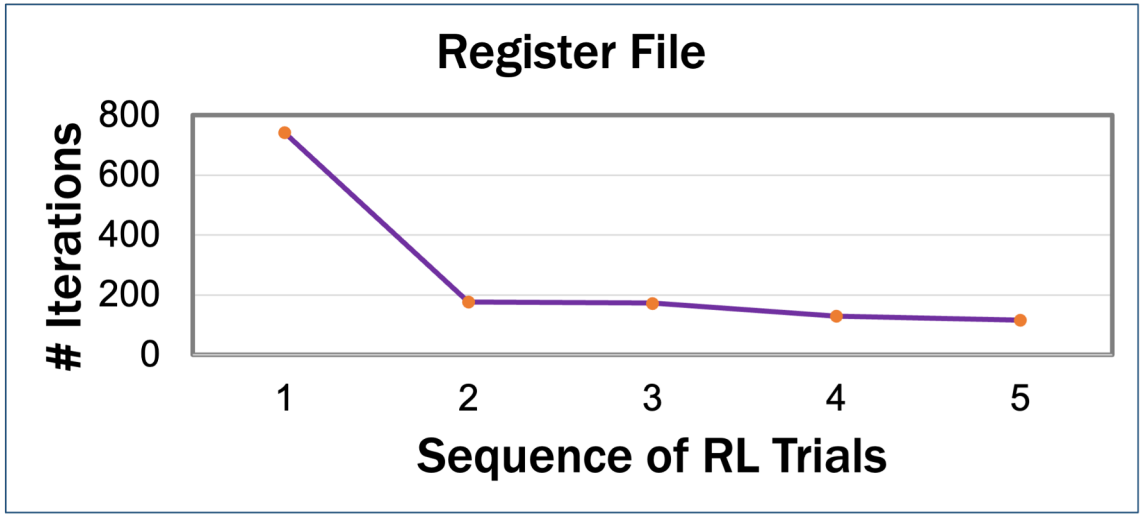
$$\mathcal{Q}^{new}(\mathcal{S}_t, \mathcal{A}_t) = (1-\alpha)\mathcal{Q}(\mathcal{S}_t, \mathcal{A}_t) + \alpha(\mathcal{R}_{t+1} + \gamma \max_{\mathcal{A}_t} \mathcal{Q}(\mathcal{S}_{t+1}, \mathcal{A}_t)$$

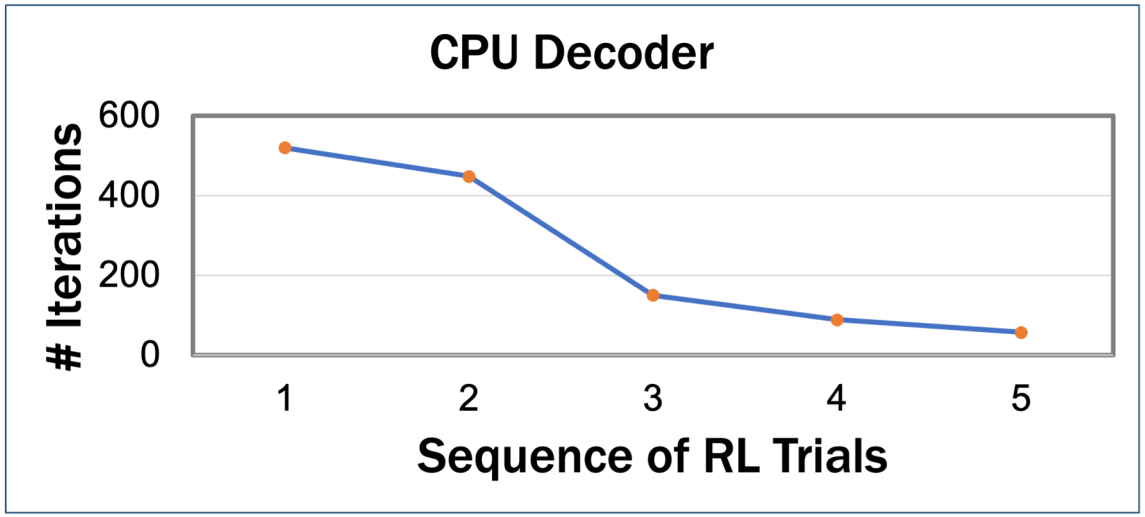
Results Analysis Contd...

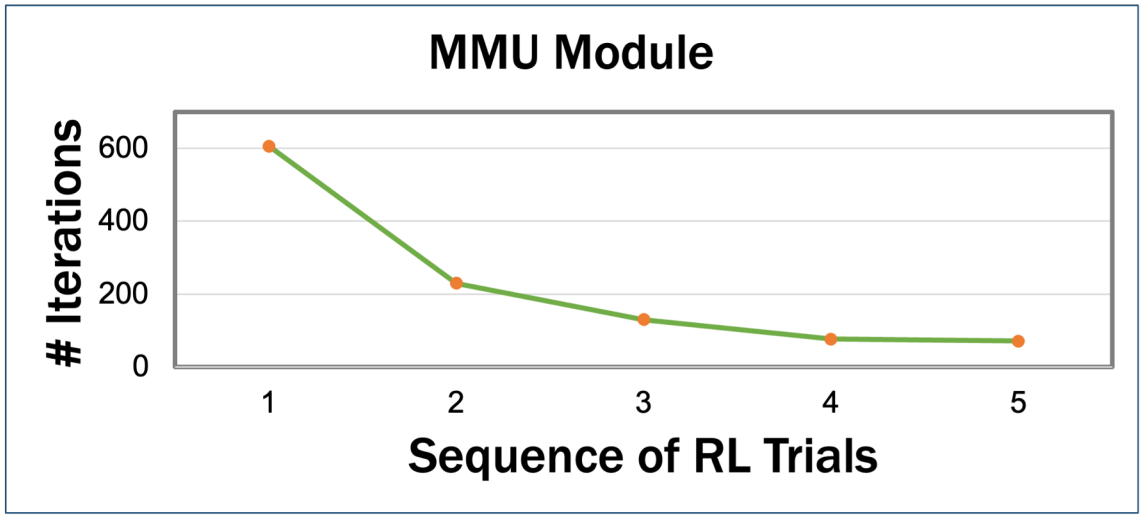
Decreasing iterations significantly for triggering a vulnerability over the trial in RL model

E.g., in worst case, first vulnerability detection in 20K, decreases to 15K in second trial









Performance evolution over the trials while verifying the Ariane SoC modules

Results Analysis

Summary Results: Vulnerability Detection

- Saved at least 47% of verification time comparing w/o proposed AI feedback
- Saved at least 15% of verification time comparing to SoCFuzzer framework
- Saved 23% verification time on average comparing to prior SoCFuzzer framework
- Detected three unknown vulnerabilities

Index	Number of Executions (Average)			Speed Up	
	F_{NPNF}	*SoCFuzzer [32]*	*SoCFuzzer+*	S1	S2
SV1	23489	10832	9161	61.00%	15.43%
SV2	21938	8387	6644	69.71%	20.78%
SV3	483	258	190	60.66%	26.36%
SV4	452	311	237	47.57%	23.79%
SV5	478	293	242	49.44%	17.52%
SV6	397	226	140	64.82%	38.20%
SV7	449	214	172	61.77%	19.78%
SV8	3247	1922	492	84.85%	74.40%
USV1	N/A	N/A	16392	N/A	N/A
USV2	N/A	N/A	37163	N/A	N/A
USV3	N/A	N/A	826	N/A	N/A

F_{NPNF}: Fuzzing w/o proposed cost-function-based feedback and AI model.
S1: *SoCFuzzer+* speedup w.r.t. F_{NPNF} and S2: Speedup w.r.t. *SoCFuzzer*.

Unknown Vulnerability Detection

Index	Vulnerability	Location	Triggering Condition	Output Behaviour	Reference
USV1	A Trojan allows exposure of the previous cipher for AES module in SoC	AES IP	Specific plaintext	Read last ciphertext	CWE-401
USV2	A Trojan leaks intermediate result as final cipher	AES IP	Specific plaintext	Intermediate ciphertext	[4]
USV3	CSR read access to undefined High Performance Counter (HPC)	Register File	hpmcounter read	No exception raised	CWE1281

Outline

SoC Security Verification

Definition of Security Verification, The "Verification Crisis", Challenges, Promising Solutions

Part 1: Fuzzy/Fuzz Testing

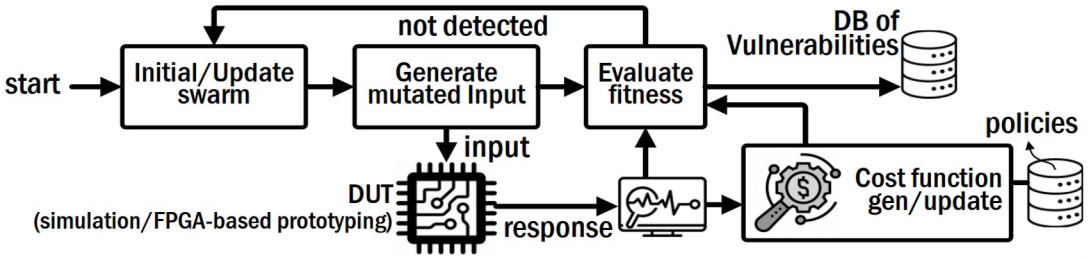
Background, High-Level Overview, Proposed Approaches, Results, Comparison, Summary

3 Part 2: Penetration (Pen) Testing

Background, Definition, High-Level Overview, Example Framework, Results, Summary

Security Verification by Hardware Penetration Test

SHarPen: Security Verification by Hardware Penetration Test

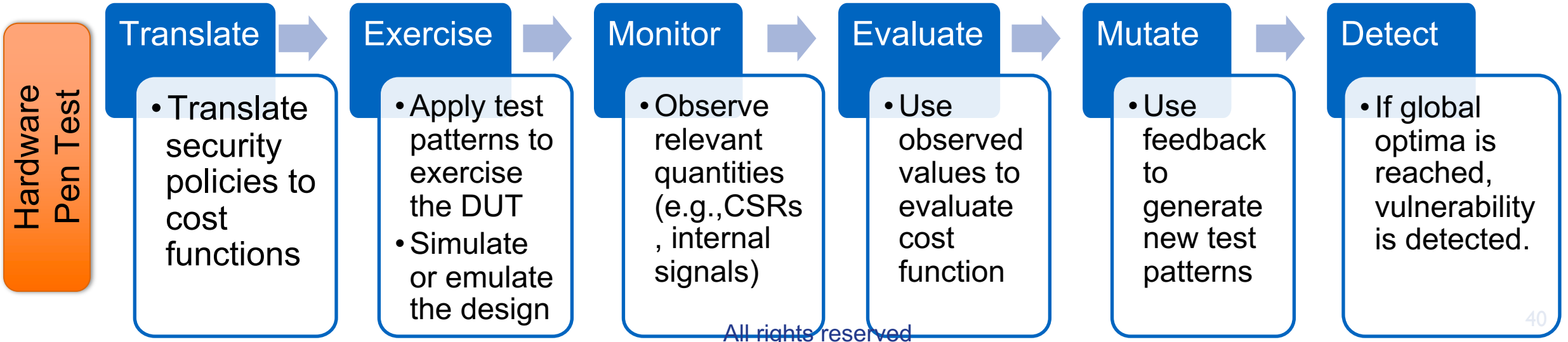


- **High level overview:**
  - Cost function driven grey box hardware penetration testing framework
  - Both simulation and emulation compatible
  - Self-evolutionary test pattern generation through Binary Particle Swarm Optimization
  - Capable of triggering hardware vulnerabilities even with remote (software) access
  - Applicable across a wide variety of threat models
- **Fundamental insight** Security policies can be represented as mathematical cost functions such that
  - finding global optimal will trigger the vulnerabilities (if they exist) that violate them

SHarPen Requirements and Steps

- **Requirements:**
  - **PR1:** The tester possesses a high-level knowledge of potential vulnerabilities and their high-level impact on neighboring signals/modules.
  - **PR2:** The tester can observe a sub-set of signals of the design that might be impacted by the vulnerabilities (grey-box).
  - **PR3:** The tester can control a sub-set of relevant signals of the design to the vulnerabilities (grey-box).

Steps:

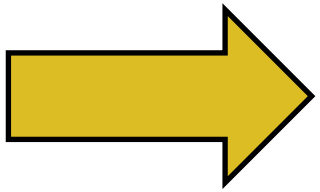


Step 1: Translate Security Policies

Translate: Translate security policies to **grey box** cost functions

- Example 1:
  - Detecting information leaking + DoS trojans in AES
  - No structural/implementation knowledge assumed about the AES (Grey-Box).

Security policies
Asset should not show up at: • Read/write channel (r_{data} / w_{data}) of shared bus • Memory locations (mem_{data} accessible by untrusted IPs • Handshaking signal (vo_{clk}) should be asserted at the enc encryption within few clock cycles.

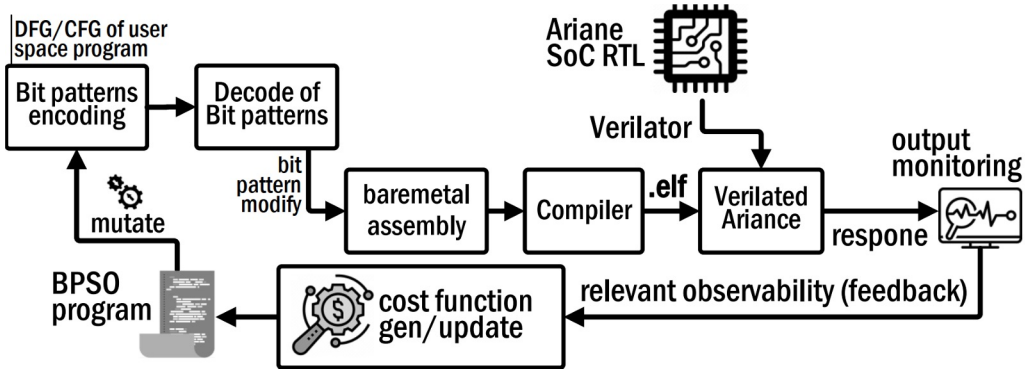


$$F = \alpha_1 HD(r_{data}, SC)$$
$$\alpha_2 HD(w_{data}, SC) +$$
$$\alpha_3 \sum HD(mem_{data}, SC) +$$
$$\alpha_4 HD(v_{out}, 0)$$

HD:= Hamming Distance
SC:= Secret Key; α= C. Parameter

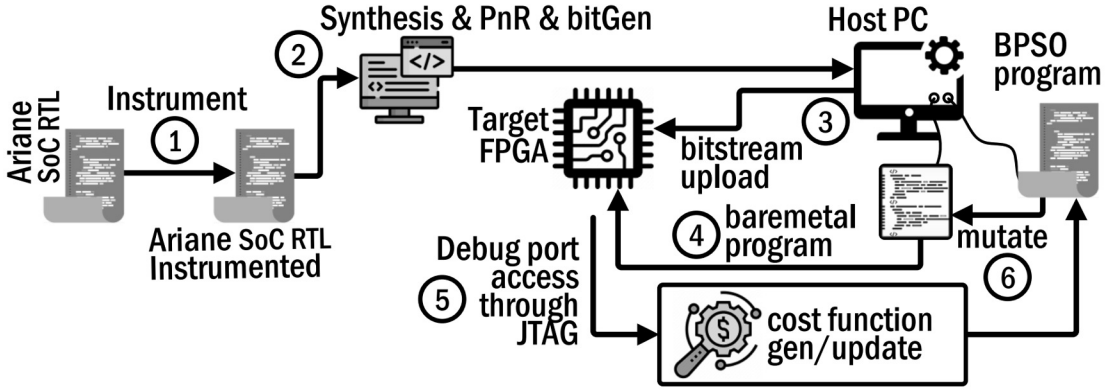
Step 2 and 3: Exercise and Monitor

- We propose two variants of **SHarPen**
  - One is simulation compatible, other emulation compatible



Simulation Framework

- RTL is converted to equivalent C++ model
- The user space program is run on the software model (Exercise)
- The .vcd file is parsed to observe relevant signals (Monitor)



Emulation Framework

- RTL is instrumented, synthesized and then prototyped on an FPGA.
- The user space program is directly run on the hardware (Exercise at-speed)
- Monitor relevant signals through Host PC via debug port (Monitor)/ ILA core

Step 4 and 5: Evaluate and Mutate

- For example 1:
  - If there's a violation of security policy
    → **At least one term becomes 0**

$$F = \alpha_1 HD(r_{data}, SC) + \alpha_2 HD(w_{data}, SC) + \alpha_3 \sum HD(mem_{data}, SC) + \alpha_4 HD(v_{out}, 1)$$

→ set to 0 when any of the terms is 0

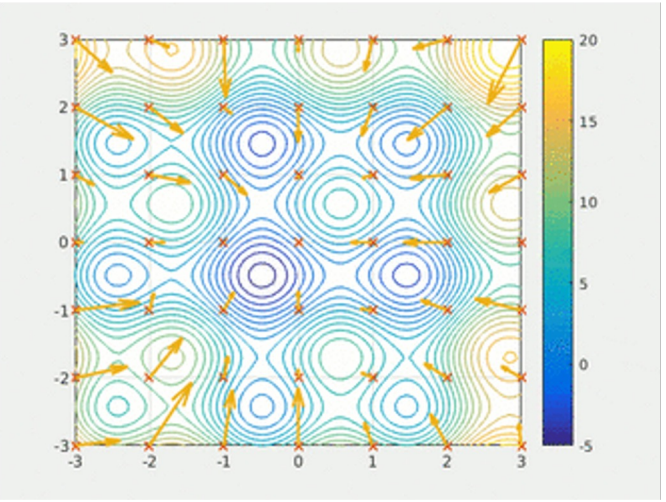


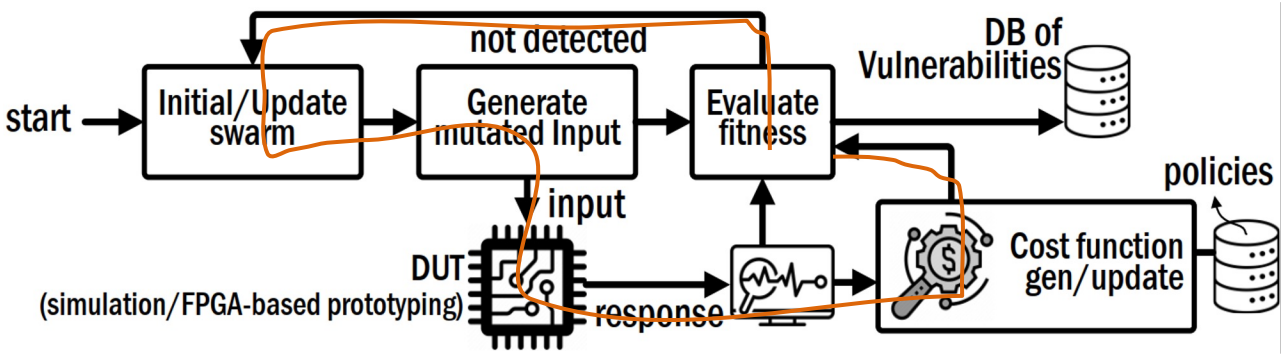
Figure: BPSO finding minima of a function *

*Source: Wikipedia

- **Insight:**
  - This becomes a mathematical function minimization problem
- Binary Particle Swarm Optimization (BPSO) is a viable solution
  - No training required
  - Adaptable to binary input vectors
  - Less prone to getting stuck in local minima.

Step 6: Detect

- Mutate test patterns (Baremetal C code for example 1) until convergence.
- $$F = \alpha_1 HD(r_{data}, SC) + \alpha_2 HD(w_{data}, SC) + \alpha_3 \sum HD(mem_{data}, SC) + \alpha_4 HD(vo_{clk}, 1)$$
- ***If there's a violation of a security policy, corresponding to current generation of swarm → At least one term becomes 0***
  - ***F*** evaluates to 0
  - The vulnerability is triggerable/exploitable with current input
    → Vulnerability exists and is detected



SHarPen

Summary of Results

- **Key takeaways:**
  - Both simulation and emulation frameworks **successful in detecting vulnerabilities**
    → Grey box cost functions
  - Emulation framework provides **nearly 15x improvement** in time required
    → Emulation offers greater scalability
  - BPSO offers **quick and reliable convergence** to global minima
  - User configurable parameter α has minimal impact on framework performance.

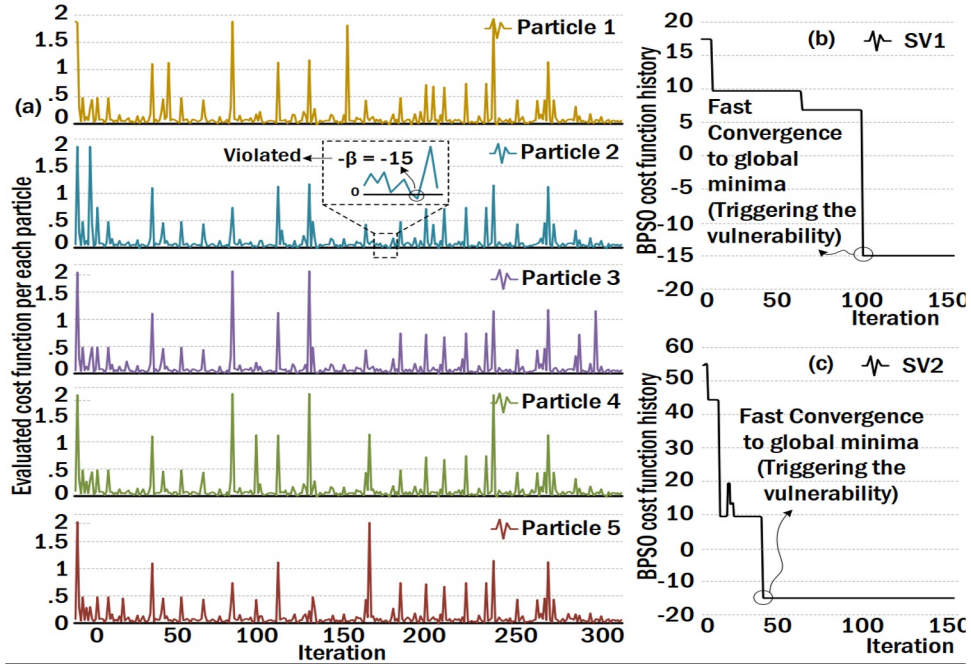


Figure: Finding Global Optima of Cost Function

Required time for Vuln. Detection

Targeted vulnerability	SV1	SV2
Simulation	3050-3150(s)	3050-3150(s)
FPGA emulation	180-190(s)	190-200 (s)

Summary

- To keep up with increasing design complexity

FPGA emulation compatibility enables at-speed execution of test-patterns

- To deal with realities of the supply chain

Requires minimal knowledge of internal signals

- To detect vulnerabilities and desired coverage quickly

Utilizes security relevant feedback for intelligent test generation

- To guide pattern generation effectively
- To maximize coverage

Utilizes security-oriented metrics

- To account for HW-SW interactions

Fuzzer running natively on DUT

Conclusion

Problem

Vast SoC threat surface. Traditional verification methods do not scale well for complex designs

Solution

Combination of Fuzz and Pen Testing offer promising alternatives with better scalability and generalizability.

Potential

Have already been successfully employed in detecting vuln. in open-source benchmarks