FormalFuzzer: Formal Verification Assisted Fuzz Testing for SoC Vulnerability Detection

ASP-DAC 2024

Nusrat Farzana Dipu, Muhammad Monir Hossain
Kimia Zamiri Azar, Farimah Farahmandi, and **Mark Tehranipoor**

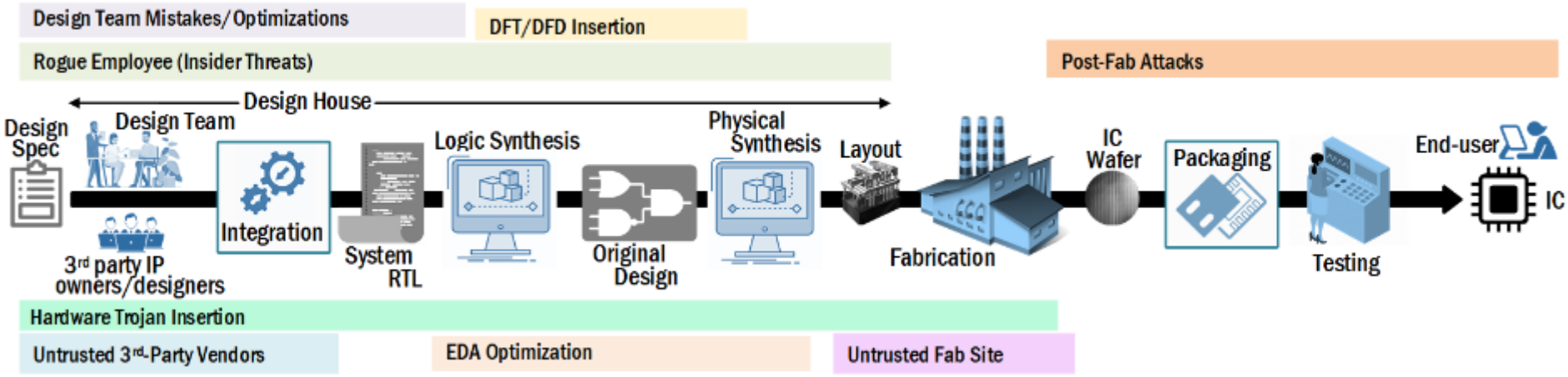
University of Florida, USA
Florida Institute for Cybersecurity Research
Department of Electrical and Computer Engineering

Outline

- Background
- Proposed Framework
- Framework Implementation
- Experimental Setup
- Result Analysis
- Conclusion and Future Work

Background: Security & Trust Issues in SoC Design Cycle

- Diversified sources of security vulnerabilities in SoC design life-cycle
- Attacker's Goal: To extract/get access to the security assets
- Attackers utilize underlying security vul. → security asset leakage
- Security assets leakage → Critically compromised security of SoC applications



Potential sources of security vulnerabilities in modern SoC design flow

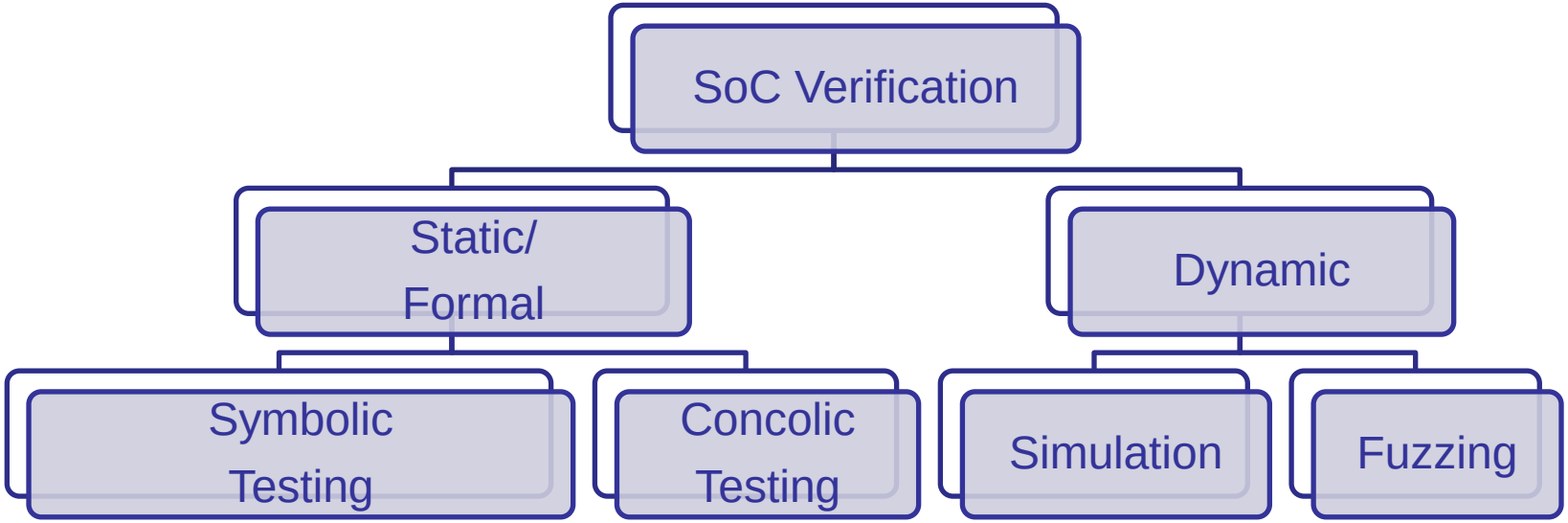
Existing SoC Verification Approaches

Static/Formal Verification

- Check design meets requirements by inspecting the code before it run.
- Verify properties consistency with design constraints and specs.

Dynamic Verification

- Performed during the execution of the program
- Run-time checks of malicious program behavior/asset leakage

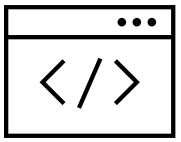


Challenges in Existing Verification Approaches

Cycle-accurate Assertions

Scalability

Manual Process



White-box Model

State-space Explosion

Long Simulation Trace

Incomplete/Low-Coverage

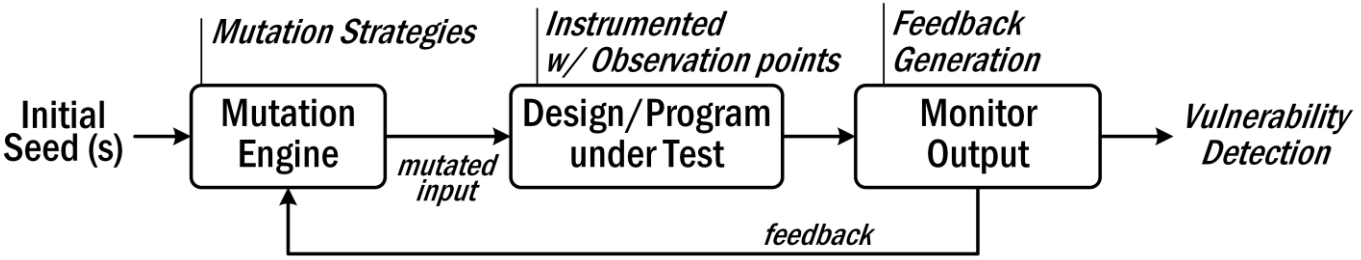
Time Consuming

False Alarms

Fuzzing: Overcome limitations- false alarm, scalability, white-box design requirement, automation!

Fuzzing for SoC Verification

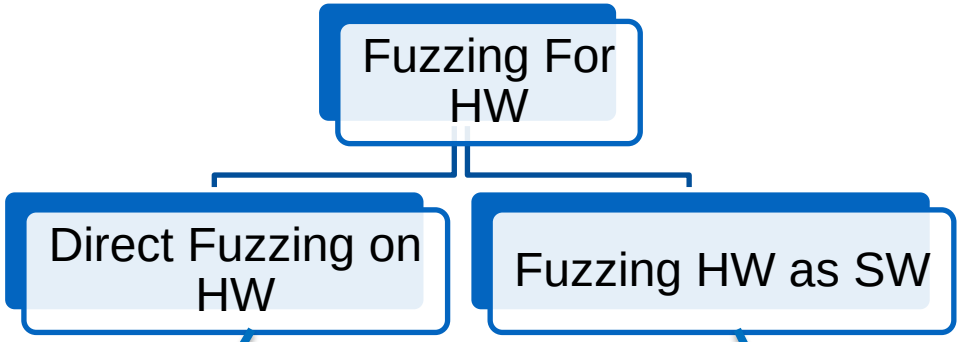
Fuzzing Technique



Motivation

- Fuzzing applicability in both black-box and gray-box model-based verification
- Scalability for any hardware design
- Coverage-guided, focused and cornerstone-directed mutation via feedback

Current Practices



Fuzz HW directly via SW sim/FPGA emulation

1. HW RTL → SW model
2. Fuzz SW model

Challenges in Current Fuzzing Practices

- Reliance on traditional SW-based code coverage (lacking map to HW design)
- More randomness in mutations and increased input space (irrelevant to DUT)
- Lacking security driven mutation and dynamic feedback
- Mostly applicable to white-box (full access to the design) model

Proposed Framework to mitigate existing challenges in SoC Verification!

Proposed Fuzzing Framework: *FormalFuzzer*

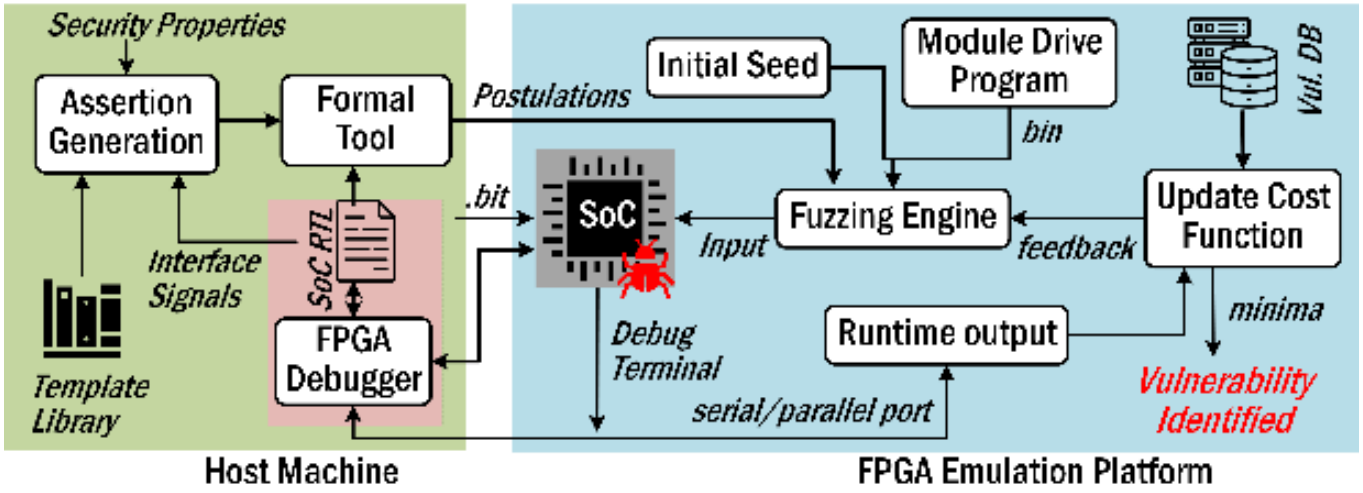
FormalFuzzer? | SoC Security Verification using Formal assisted Fuzzing

- Automated framework assisted by formal verification for SoC security verification
- Leverage property-driven formal verification for reducing input space significantly for Fuzzer
- Leverage counter-example/cover traces generated by formal tool for deriving postulation to identify cross layer bugs
- Utilizing cost function and feedback for runtime update of mutation strategies

Extensive HW-centric Mutation

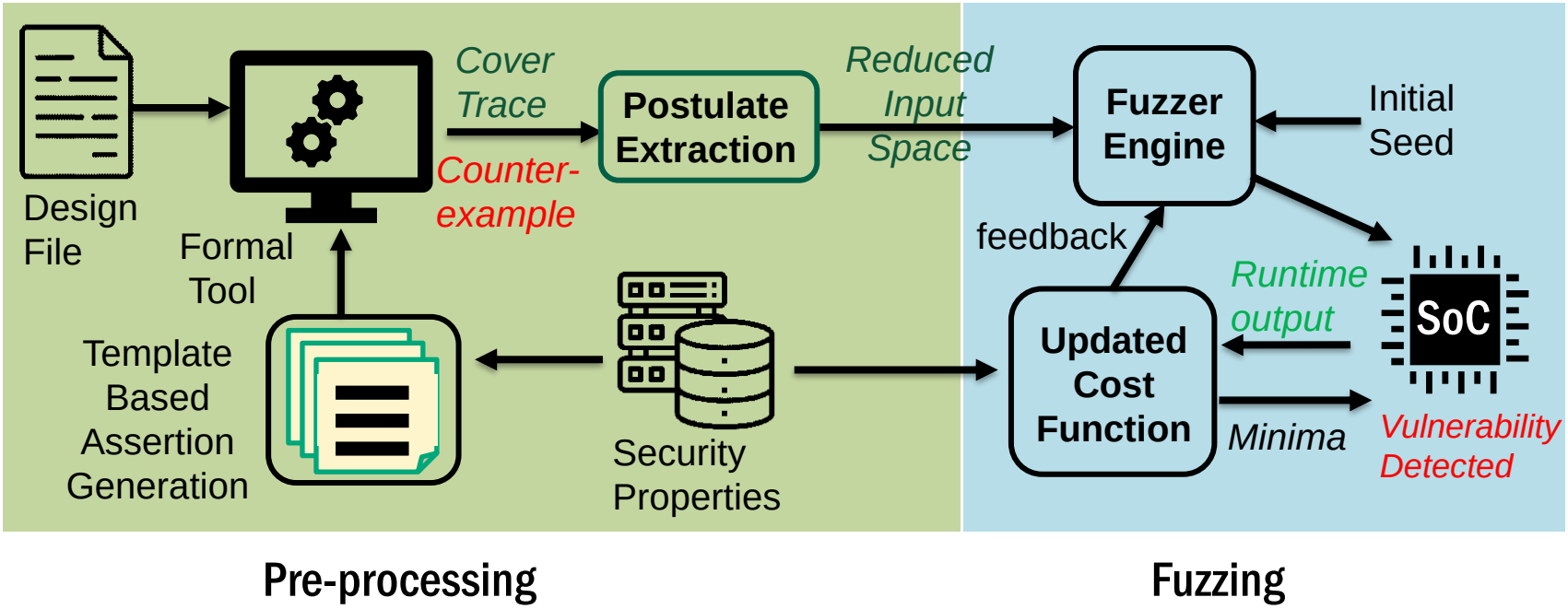
Gray-box Model Verification

Stealthiness of cross layer bugs



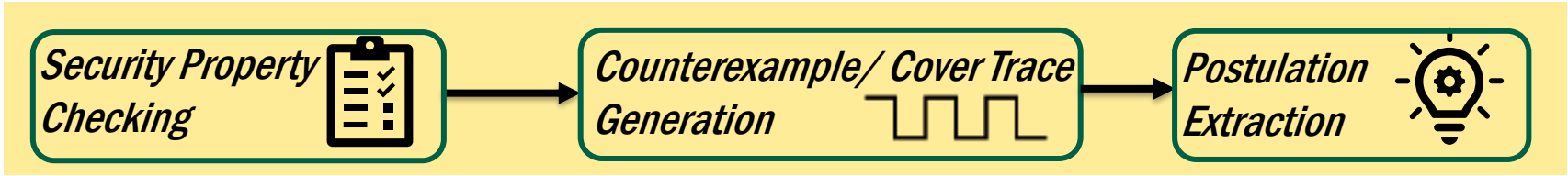
Formal Verification Assisted Pre-processing

- Property-driven formal verification is used for pre-processing to generate postulates
- **Objective:** To reduce input space for the fuzzer engine
  - To generate HW-centric test patterns for SoC vulnerability detection



Postulate Extraction from Formal Verification

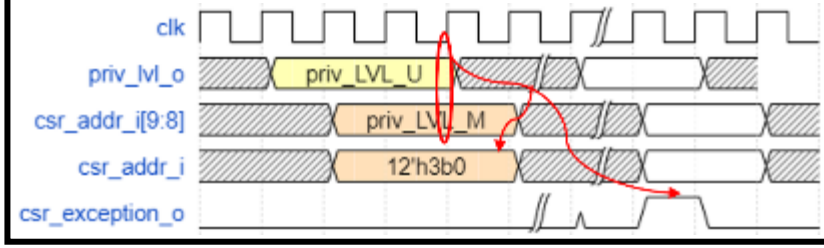
- **Postulation:** A set of constraints for effective test generation
- Guides the fuzzer engine in tuning the mutation strategy



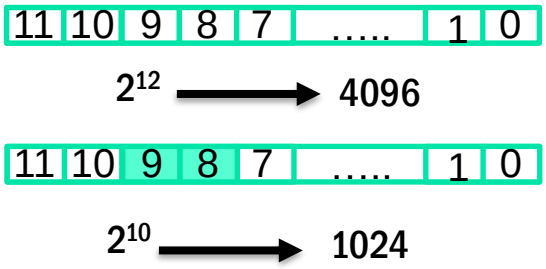
Example

Security Property: CSRs should raise an exception if the access privilege mode of CSR does not match with current privilege mode of the core

Counter-example Trace:



CSR _address Register



Postulation: Fix 9th and 8th bit

csr _address[9:8] = 2'b11=Machine mode

Reduced input space (RIS) = $\frac{(4096-1024)}{4096}$ = 75%

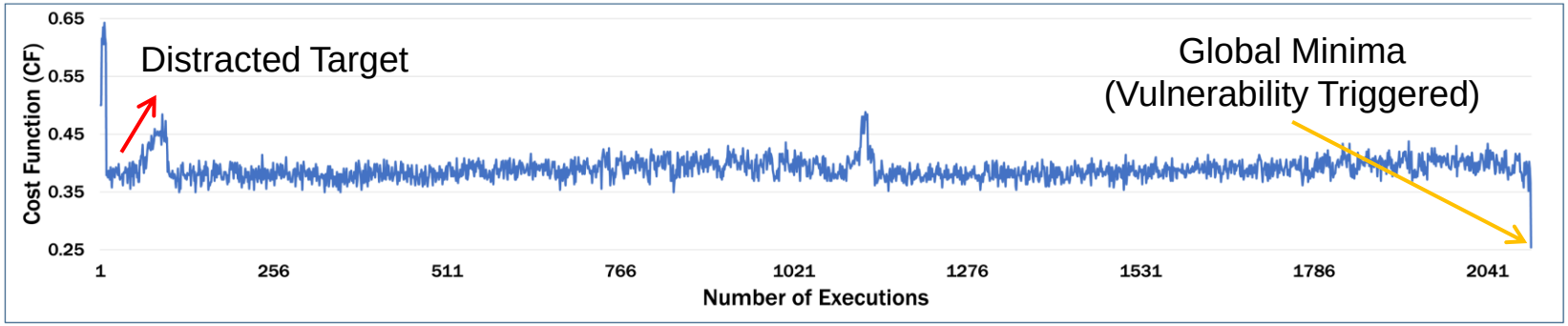
FormalFuzzer Cost Function

- Objective: guides fuzzing to generate smarter-than-random test inputs
- Evaluate run-time fuzzing performance and security properties
- Evaluate the quality of mutated inputs
- Estimate the chances of hitting a potential malicious behavior
- Fuzzing objective: minimizing cost function (global minima → vulnerability triggered)
- ↓ Cost Function → better fuzzing → faster vulnerability trigger (global minima)

Mathematical Formula

$$F_c = 1 - \frac{1}{n}[\{1 - \frac{2}{r(r-1)} \sum_{j=1}^{r-1} \sum_{k=j+1}^{r} \frac{h(i_j, i_k)}{l_i}\} + \frac{u_i}{2^{N-d}} + \frac{\sigma}{\sum_{j=1}^{z} l_z} \sum_{k=1}^{z} (h(o_{r-1,k}, o_{r,k})) + \frac{1}{m} \sum_{k=1}^{m} (o_{r,k} == o_{t,k})]$$

Trends



FormalFuzzer Feedback

- *Cost function improvement rate (CFIR)* evaluates runtime performance of fuzzing
- Positive CFIR → better fuzzing by mutating smart inputs → global minima
  - Continue mutation w/ the same mutation strategy
- Negative CFIR → Mutation against objective → increasing cost function
  - Change the mutation strategy
- Objective: keep CFIR always high positive
- CFIR estimated after each frequency of feedback evaluation (f_f) iterations

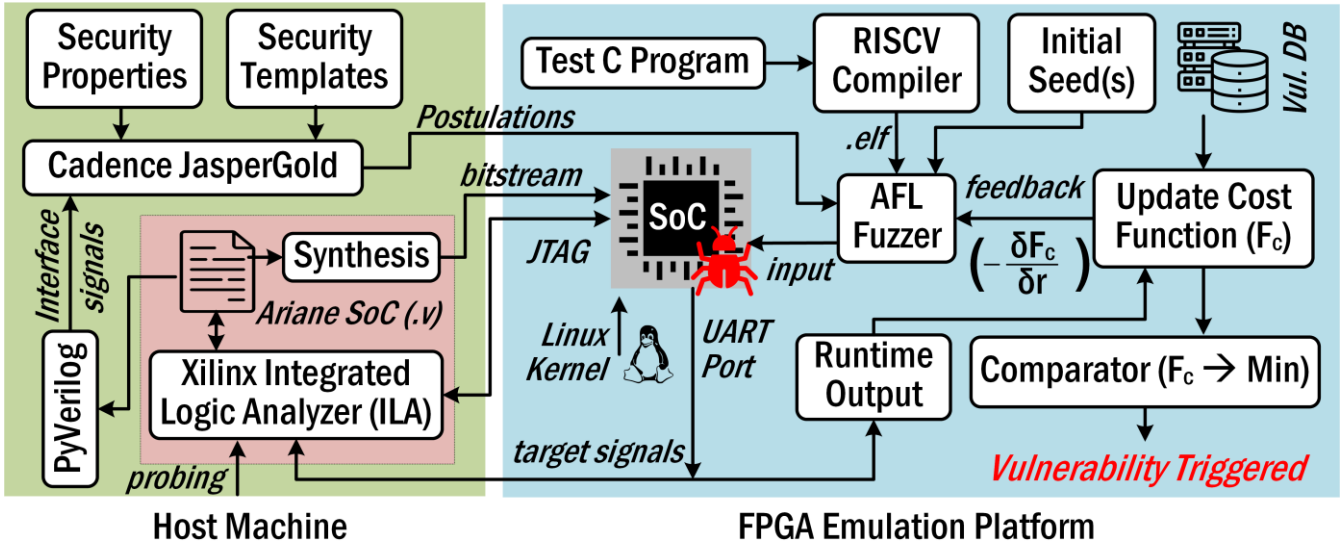
Mathematical Formula

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

Trends

FormalFuzzer Implementation

- SoC: RISC-V-based 64-bit Ariane SoC
- HW debugging: Xilinx Integrated Logic Analyzer (ILA)
- Pre-processing: PyVerilog, Cadence JasperGold (Jasper)
- Emulation: Genesys 2 Kintex-7 FPGA Development Board
- Linux kernel mounted on SoC via SD card on FPGA board
- American Fuzzy Lob (AFL) instrumented for-
  - Utilizing postulations to mutate from targeted input space
  - *Computing F_c , CFIR and* utilize feedback to change the mutation technique



Experimental Setup

Vulnerabilities in Ariane SoCs

Index	Vulnerability	Location	Triggering Condition	Reference
SV1	Access to *mstatus* CSR from lower privilege level	CPU (dec)	Unauthorized read to CSR	CWE-1262
SV2	Allow executing *mret* machine-level ins. from user-mode	CPU (dec)	*Unauthorized instruction execution*	CWE-1242
SV3	Incorrect logic to decode FENCE.I ins.	CPU (dec)	imm ≠ 0 & rs1 ≠ 0	CWE-440
SV4	Leaking AES key through SoC common bus (ciphertext)	Encryption Module	*Specific Plaintext*	CVE-2018-8922
SV5	A Trojan delays cipher conversion in the AES module	Encryption Module	*Specific Plaintext*	AES-T500
SV6	Unauthorized memory access via MMU	CPU(LSU)	Illegal memory access	CWE-269

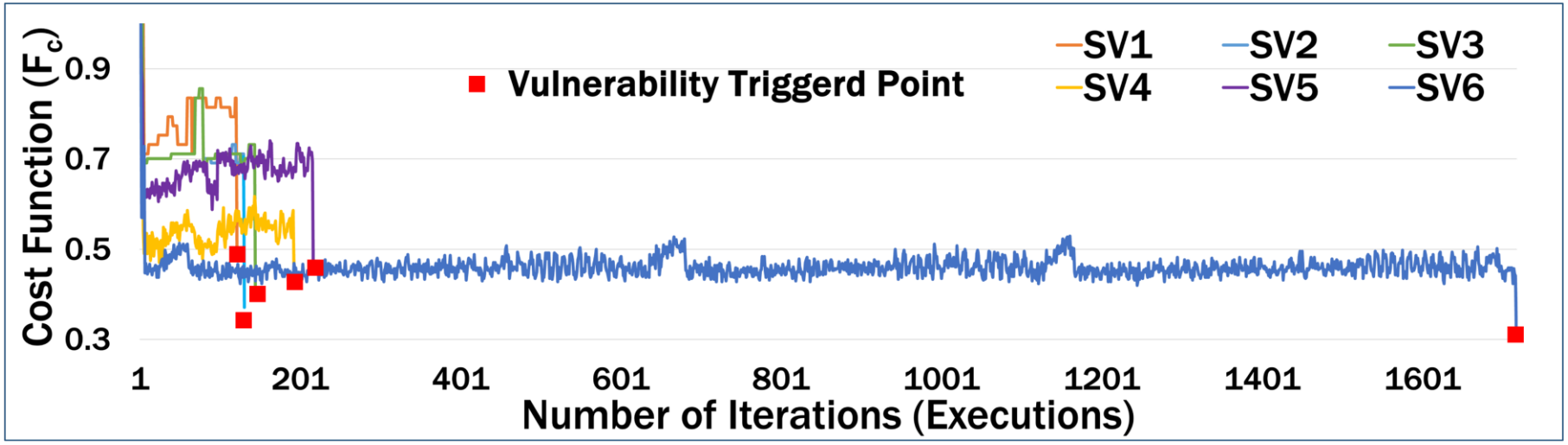
Result Analysis Contd…

Reduced Input Space (RIS) for Fuzzing in Ariane SoCs

Index/ Module	Security Property	Assertion	Postulation	RIS
SM1/ CSR file	**SP11**: CSRs should raise an exception if the access privilege mode of CSR does not match with current privilege mode of the core; **SP12**: CSRs should raise an exception if a wrong access request is made to CSRs	**SA11**: (csr addr i[9 : 8]! = priv lvl o|→ ##[1 : $](csr exception o.valid == 1′b1); SA12: (csr addr i[11 : 10]! = csr op i) |→ ##[1 : $](csr exception o.valid == 1′b1	P11: User's privilege escalation to Machine-mode CSRs; P12: Write request to readonly CSRs	65-73%
SM2/ CPU decoder	**SP21**: If privilege level changes, the corresponding instruction should also change to avoid an instruction specified for one privilege level, to be executed in another privilege level	SA21: ~ $stable(priv lvl i) |→ ##[1 : $] ~ $stable(instr o)	*P21: Machine-level instruction from user mode;* *P22: Mutate specific bits (31:7) of instruction field*	21%
SM3/ AES	**SP31**: Encryption key should not be leaked through cipher text output port; **SP32**: After receiving plain text and encryption key as inputs, the encryption result should be available at the output after a specified number of clock cycles	SA31: encryption key! =encryption result; SA32:~ $stable(encryption key)|| ~ $stable(plain text) |→ ##[1 : a](encryption result! = 0) && ~ $stable(encryption result)	P31: Check (mutate) first byte of plain text	93%
SM4/ MMU	**SP41**: MMU should raise an exception when physical memory access is operated in U/S mode	SA41: (priv lvl i! = P RIV LV L M)&& (icache areq i.fetch req == 1′b1) |→ (icache areq o.fetch exception == 1′b1)	*P41: Access physical memory in user mode*	50%

Results Analysis Contd…

- Identification of known vulnerabilities by analyzing cost function and feedback



Cost function of detected known vulnerabilities by FormalFuzzer

Results Analysis Contd…

- FormalFuzzer also detected few unknown vulnerabilities in the Ariane SoC

Index	Vulnerability	Location	Triggering Condition	Reference
UV1	Retrieve last ciphertext in AES module (Trojan)	Encryption Module	Specific plaintext	CWE-401
UV2	CSR read access to undefined HPC	CSR file	No exception	CWE1281
UV3	UV3 No exception raised for illegal format of MULH	CPU (dec)	rd ∈ {rs1, rs2}	CWE1262

Results Analysis

Summary Results: Vulnerability Detection

- Save 55% of verification time compared to system w/o proposed pre-processing and feedback
- Save 19% compared to SoCFuzzer lacking proposed pre-processing

Index	*Freq. of Feedback Evaluation (f_f)*	No. of Executions (Avg)			Speed Up	
		F_{NPNF}	SoCFuzzer[1]	FormalFuzzer	S1	S2
SV1	6	868	541	201	76.84%	62.85%
SV2	5	421	165	133	68.41%	19.39%
SV3	5	361	208	161	55.90%	22.60%
SV4	5	15254	6687	187	98.77%	97.2%
SV5	5	12896	5210	204	98.42%	96.08%
SV6	6	5331	2592	1937	63.67%	25.27%
UV1	5	N/A	N/A	7389	N/A	N/A
UV2	6	N/A	N/A	758	N/A	N/A
UV3	5	N/A	N/A	737	N/A	N/A

F_{NPNF} : Fuzzing w/o proposed pre-processing and cost-function-based feedback.
S1: FormalFuzzer speedup w.r.t. F_{NPNF} and S2: Speedup w.r.t. SoCFuzzer.

Reference

[1] Hossain, Muhammad Monir, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. "SoCFuzzer: SoC Vulnerability Detection using Cost Function enabled Fuzz Testing." In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1-6. IEEE, 2023.

Comparison w/ HW Fuzzing Approaches

- FormalFuzzer's strengths →
  - Independence from golden model and white-box model
  - Formal assisted pre-processing reducing input space and target oriented fuzzing

Framework	Target Design	Approach	Feedback/Coverage	Automation	Gray-box	Golden Model
HyPFuzz[1]	CPU	HDL Sim	Branch	No	No	Yes
TheHuzz[2]	CPU	HDL Sim	FSM, Statement	No	No	Yes
HyperFuzzing[3]	SoC	SW Sim	NoC, Bitflip	No	Yes	Yes
DifuzztRTL[4]	CPU	FPGA Emu	Control-register	No	No	Yes
RFUZZ[5]	IP	FPGA Emu	MUX	No	No	Yes
Fuzzing HW as SW[6]	SoC	SW Sim	Branch/Code	Yes	No	Yes
FormalFuzzer	SoC	FPGA Emu	Cost Function	Yes	Yes	No

References

[1] Chen, Chen, et al. "HyPFuzz: Formal-Assisted Processor Fuzzing." arXiv preprint arXiv:2304.02485 (2023).

[2] R. Kande et al., "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," USENIX, 2022.

[3] S. K. Muduli et al., "Hyperfuzzing for soc security validation," in International Conference on CAD (ICCAD), 2020, pp. 1–9.

[4] S. Canakci et al., "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in DAC, 2021, pp. 529–534.

[5] K. Laeufer et al., "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in International Conference on CAD (ICCAD), 2018, pp. 1–8.

[6] T. Trippel et al., "Fuzzing hardware like software," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3237–3254.

Conclusion and Future Work

Conclusion

- Developed FormalFuzzer, a fuzzing framework for SoC security verification
- Integrated formal verification strategies for reducing input space and efficient mutation
- Utilized cost function concept targeting vul. and develop feedback for smart mutation
- Experiments proved FormalFuzzer's capability in detecting vul. within reasonable time

Future Works

- Integration of Artificial Intelligence (AI) in generating smart seeds and efficient mutation
- Improvement of cost function to target more unknown vulnerabilities

Questions?

Contact: Dr. Nusrat Farzana Dipu at ndipu@ufl.edu
Muhammad Monir Hossain at hossainm@ufl.edu
Dr. Mark Tehranipoor at tehranipoor@ece.ufl.edu