Creating a Labeled Dataset for Fitting Deep Learning Model on The PicoRV RISC-V Core

Presented by Kamelia Zaman Moon

Introduction

Side-channel attacks exploit physical leakage to break cryptography. RISC-V processors are becoming common in lightweight embedded systems. This project focuses on preparing data for evaluating side-channel vulnerabilities in cryptographic gadgets.

Goal: Create a labeled dataset to train deep learning models for leakage detection.

Problem Statement

Existing datasets for SCA are limited in diversity and do not focus on masking variants in RISC-V. [3] [4]

Need for:

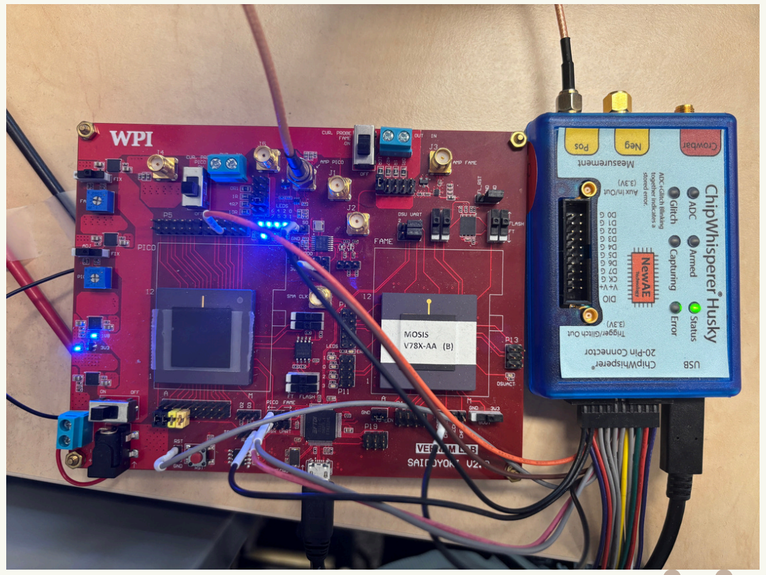
- Diverse gadget-level implementations.
- Simulation and power-based trace generation.
- Accurate labeling of operations and registers.

Objectives

- Implement and simulate multiple masking gadgets in RISC-V.
- Capture execution and power traces.
- Label traces with appropriate leakage models (HW, HD).
- Analyze correlations to verify leakage.
- Produce a dataset suitable for training deep learning models.

Experimental Setup

- **Platform:** RISC-V PicoRV32 core.
- **Hardware:** CW-Husky, Saidoyoki [2]
- **Tools:** Qiling, PoMMES
- Input vectors, randomized masking, power trace collection
- **Leakage Assessment:** Correlation Power Analysis



Gadget 1 - ISW-Masked AND

- Based on the Ishai–Sahai–Wagner (ISW) masking scheme for secure multi-share computation.
- Protects against side-channel leakage by computing on masked shares and using a random mask r.
- Operates on input shares: (a0, a1) and (b0, b1) with output (c0, c1).

```
.section .text
.global isw_and
isw_and:
    # Inputs: a0 = A0, a1 = A1, a2 = B0, a3 = B1
    # a4 = r
    mv t0, a0 # A0
    mv t1, a1 # A1
    mv t2, a2 # B0
    mv t3, a3 # B1
    mv t4, a4 # r
    # Compute partial products
    and t5, t0, t2 # t5 = A0 & B0
    and t6, t1, t3 # t6 = A1 & B1
    # Cross terms
    and t0, t0, t3 # t0 = A0 & B1 (reuse t0)
    and t1, t1, t2 # t1 = A1 & B0 (reuse t1)
    xor t0, t0, t1 # t0 = A0&B1 ^ A1&B0
    xor t0, t0, t4 # t0 = t0 ^ r
    xor t5, t5, t4 # C0 = A0&B0 ^ r
    xor t6, t6, t0 # C1 = A1&B1 ^ cross ^ r
    mv a0, t5 # Return C0
    mv a1, t6 # Return C1
    ret
```

Gadget 1 - ISW-Masked AND

Variants

v1 – Memory-Assisted ISW AND:

- Uses memory instructions (sw, lw) to store/load temporary results.

```
isw_and:
    addi sp, sp, -16
    sw a0, 0(sp) # A0
    sw a1, 4(sp) # A1
    sw a2, 8(sp) # B0
    sw a3, 12(sp) # B1
    lw t0, 0(sp) # A0
    lw t1, 4(sp) # A1
    lw t2, 8(sp) # B0
    lw t3, 12(sp) # B1
    ....
```

v2 – Unmasked AND:

- Performs a direct AND on unmasked inputs.

```
isw_and:
    mv t0, a0
    .....
    xor t0, t0, t4 # ^ r
    # Intentionally leave C0 unmasked
    # xor t5, t5, t4 # OMITTED
    xor t6, t6, t0 # C1
    mv a0, t5 # C0 = A0 & B0 (leaky!)
    mv a1, t6
    ret
```

Gadget 2 - Toffoli Gate

- The Toffoli gate is a 3-bit reversible logic gate, crucial in both classical reversible computing and quantum circuits. [1]
- Implemented in RISC-V assembly using ISW-masked AND operations to maintain side-channel resistance.
- Operates on three masked inputs: (a0, a1), (b0, b1), and (c0, c1) and computes masked outputs via multiple bitwise operations.

```
.section .text
.global toffoli_gate
# Input: a0, b0, c0, a1, b1, c1 stored in registers x10 - x15
# Temp register: x16 (R6)
toffoli_gate:
    and x16, x10, x11  # a0b0 -> R6 (x18)
    xor x12, x12, x16  # c0 + a0b0 -> R2 (x12)
    slli x10, x10, 2  # shift R0 (x10)
    slli x14, x14, 2  #  shift R4 (x14)
    and x16, x10, x14  # a0b1 -> R6 (x18)
    srli x10, x10, 2  # shift back R0 (x10)
    srli x14, x14, 2  # shift back R4 (x14)
    slli x12, x12, 2  # shift R2 (x12)
    xor x12, x12, x16  # c0 + a0b0 + a0b1 -> R2 (x12)
    srli x12, x12, 2  # shift back R2 (x12)
    and x16, x13, x14  # a1b1 -> R6 (x16)
    xor x15, x15, x16  #  a1b1 + c1 -> R5 (x15)
    slli x13, x13, 2  # shift R3 (x13)
    slli x11, x11, 2  # shift R1 (x11)
    and x16, x13, x11  # a1b0 -> R6 (x16)
    srli x13, x13, 2  # shift back R3 (x13)
    srli x11, x11, 2  # shift back R1 (x11)
    slli x15, x15, 2  # shift R5 (x15)
    xor x15, x15, x16  # c1 + a1b1 + a1b0 -> R5 (x15)
    srli x15, x15, 2  # shift back R5 (x15)
    xor x16, x16, x16  # Clear R6 (x16)
    mv x10, x15
    ret
```

Gadget 2 - Toffoli Gate

Variants

v1 – Instruction Reordering:

- Non-dependent instructions reordered.

```
toffoli_gate:
    and x16, x10, x11
    xor x12, x12, x16
    slli x14, x14, 2
    slli x10, x10, 2  # reordered
    and x16, x10, x14
.....
```

v2 – Different Temporary Register:

- Uses x17 instead of x16.

```
toffoli_gate:
    and x17, x10, x11
    xor x12, x12, x17
    slli x10, x10, 2
    slli x14, x14, 2
    and x17, x10, x14
.....
```

Gadget 2 - Toffoli Gate

Variants

v3 – Alternative Shifting Sequence:

- Alters shift operation order.

```
toffoli_gate:
    and x16, x10, x11
    xor x12, x12, x16
    slli x10, x10, 2
    slli x14, x14, 2
    and x16, x10, x14
    srli x14, x14, 2
    srli x10, x10, 2 # altered
    slli x12, x12, 2
.....
```

v4 – Redundant Operation:

- Adds instructions like addi x0, x0, 0.

```
toffoli_gate:
    and x16, x10, x11
    addi x0, x0, 0 # No-op
    xor x12, x12, x16
    mv x17, x17 # Redundant move
    slli x10, x10, 2
    mv x18, x18 # Redundant move
    slli x14, x14, 2
    addi x0, x0, 0 # No-op
.....
```

Gadget 2 - Toffoli Gate

Variants

v5 – Fake Dependencies:

- Adds control flow distortion and dummy XORs.

```
toffoli_gate:
    and x16, x10, x11
    j .L1
.L1:
    xor x12, x12, x16
    slli x10, x10, 2
    slli x14, x14, 2
    and x16, x10, x14
    j .L2
.L2:
    srli x10, x10, 2
.....
```

v6 – Deliberate Data Dependency:

- Adds unnecessary save/restore operations.

```
toffoli_gate:
    and x16, x10, x11
    xor x12, x12, x16
    slli x10, x10, 2
    mv x18, x16   # data dependency
    slli x14, x14, 2
    and x16, x10, x14
    mv x16, x18
    srli x10, x10, 2
.....
```

Gadget 2 - Toffoli Gate

Variants

v7 – Redundant Temporary Register Clearing:

- Clears temporary register explicitly.

```
toffoli_gate:
    and x16, x10, x11
    xor x12, x12, x16
    xor x16, x16, x16  # Clearing register
    slli x10, x10, 2
    slli x14, x14, 2
    and x16, x10, x14
    xor x16, x16, x16  # Clearing register
.....
```

Gadget 3 - Locality Refresh

- The Locality Refresh gadget is a lightweight masking primitive.
- This re-randomizes input shares without altering the underlying secret.
- Commonly used as a preprocessing step or uniformity booster in larger masked circuits.

```
.section .text
.global locality_refresh
locality_refresh:
    # Inputs:
    # a0 = A0 (first share)
    # a1 = A1 (second share)
    # a2 = r (random mask)
    mv t0, a0 # t0 = A0
    mv t1, a1 # t1 = A1
    mv t2, a2 # t2 = r
    xor t0, t0, t2 # t0 = A0 ^ r
    xor t1, t1, t2 # t1 = A1 ^ r
    mv a0, t0 # a0 = refreshed A0
    mv a1, t1 # a1 = refreshed A1
    ret
```

Gadget 3 - Locality Refresh

Variants

v1 – Reduced Operations:

- Directly applies XOR to inputs: xor a0, a0, a2.

```
locality_refresh:
    # a0 = A0, a1 = A1, a2 = r

    xor a0, a0, a2 # A0' = A0 ^ r
    xor a1, a1, a2 # A1' = A1 ^ r

    ret
```

v2 – Swapped Register Usage:

- Changes assignment order and register allocation.

```
locality_refresh:
    # a0 = A0, a1 = A1, a2 = r

    mv t3, a0
    mv t4, a1
    xor t3, t3, a2
    xor t4, t4, a2
    mv a0, t3
    mv a1, t4

    ret
```

Gadget 3 - Locality Refresh

Variants

v3 – Mixed Register Use:

- Refreshes one share (a1) in-place and the other (a0) via a temporary register.

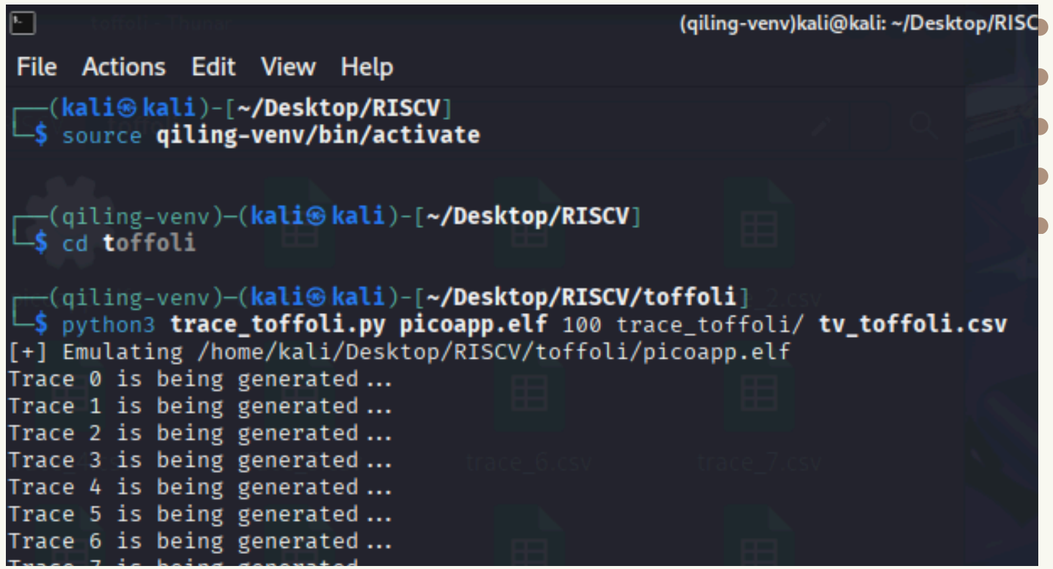
```
locality_refresh:
    # a0 = A0, a1 = A1, a2 = r

    xor t0, a0, a2 # t0 = A0 ^ r
    xor a1, a1, a2 # a1 = A1 ^ r
    mv a0, t0 # a0 = refreshed A0

    ret
```

Trace Generation

- Used Qiling framework to emulate PicoRV32 behavior.
- Captured register states and intermediate variables.
- Enabled controlled, repeatable trace generation.
- Each variant executed 100 times with unique inputs.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	PC	Machine	Ins	Type	Operands	t0	t1	t2	a0	a1	a2	a3	a4
2	0xc94	93020500	mv	ARITHMET	t0, a0	0xa24181	0xa21820	0x2ee9fa4	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
3	0xc98	13830500	mv	ARITHMET	t1, a1	0x379811	0xa21820	0x2ee9fa4	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
4	0xc9c	93030600	mv	ARITHMET	t2, a2	0x379811	0xe661a9(	0x2ee9fa4	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
5	0xca0	138e0600	mv	ARITHMET	t3, a3	0x379811	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
6	0xca4	930e0700	mv	ARITHMET	t4, a4	0x379811	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
7	0xca8	33ff7200	and	LOGICAL	t5, t0, t2	0x379811	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
8	0xcac	b37fc301	and	LOGICAL	t6, t1, t3	0x379811	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
9	0xcb0	b3f2c201	and	LOGICAL	t0, t0, t3	0x379811	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
10	0xcb4	33737300	and	LOGICAL	t1, t1, t2	0x151800(	0xe661a9(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37
11	0xcb8	b3c26200	xor	LOGICAL	t0, t0, t1	0x151800(	0x646008(	0x6c6e5c2	0x379811	0xe661a9(	0x6c6e5c2	0x957de07	0x37

ISW AND

Trace Generation

	A	B	C	D	E	F	G	H	I	J	K	L
1	PC	Machine	Ins	Type	Operands	a0	a1	a2	a3	a4	a5	a6
2	0xc94	3378b500	and	LOGICAL	a6, a0, a1	0x473c087	0xa047ef4	0x1375d27	0xe0278fb	0x9ce2f69	0x32c2561	0x0
3	0xc98	33460601	xor	LOGICAL	a2, a2, a6	0x473c087	0xa047ef4	0x1375d27	0xe0278fb	0x9ce2f69	0x32c2561	0x4084c
4	0xc9c	13152500	slli	SHIFT	a0, a0, 2	0x473c087	0xa047ef4	0x1371da3	0xe0278fb	0x9ce2f69	0x32c2561	0x4084c
5	0xca0	13172700	slli	SHIFT	a4, a4, 2	0x1cf021f	0xa047ef4	0x1371da3	0xe0278fb	0x9ce2f69	0x32c2561	0x4084c
6	0xca4	3378e500	and	LOGICAL	a6, a0, a4	0x1cf021f	0xa047ef4	0x1371da3	0xe0278fb	0x738bda4	0x32c2561	0x4084c
7	0xca8	13552500	srli	SHIFT	a0, a0, 2	0x1cf021f	0xa047ef4	0x1371da3	0xe0278fb	0x738bda4	0x32c2561	0x108000
8	0xcac	13572700	srli	SHIFT	a4, a4, 2	0x73c087f	0xa047ef4	0x1371da3	0xe0278fb	0x738bda4	0x32c2561	0x108000
9	0xcb0	13162600	slli	SHIFT	a2, a2, 2	0x73c087f	0xa047ef4	0x1371da3	0xe0278fb	0x1ce2f69	0x32c2561	0x108000
10	0xcb4	33460601	xor	LOGICAL	a2, a2, a6	0x73c087f	0xa047ef4	0x4dc768c	0xe0278fb	0x1ce2f69	0x32c2561	0x108000
11	0xcb8	13562600	srli	SHIFT	a2, a2, 2	0x73c087f	0xa047ef4	0x5d4768	0xe0278fb	0x1ce2f69	0x32c2561	0x108000
12	0xcbc	33f8e600	and	LOGICAL	a6, a3, a4	0x73c087f	0xa047ef4	0x1751da2	0xe0278fb	0x1ce2f69	0x32c2561	0x108000
13	0xcc0	b3c70701	xor	LOGICAL	a5, a5, a6	0x73c087f	0xa047ef4	0x1751da2	0xe0278fb	0x1ce2f69	0x32c2561	0x228691

Toffoli Gate

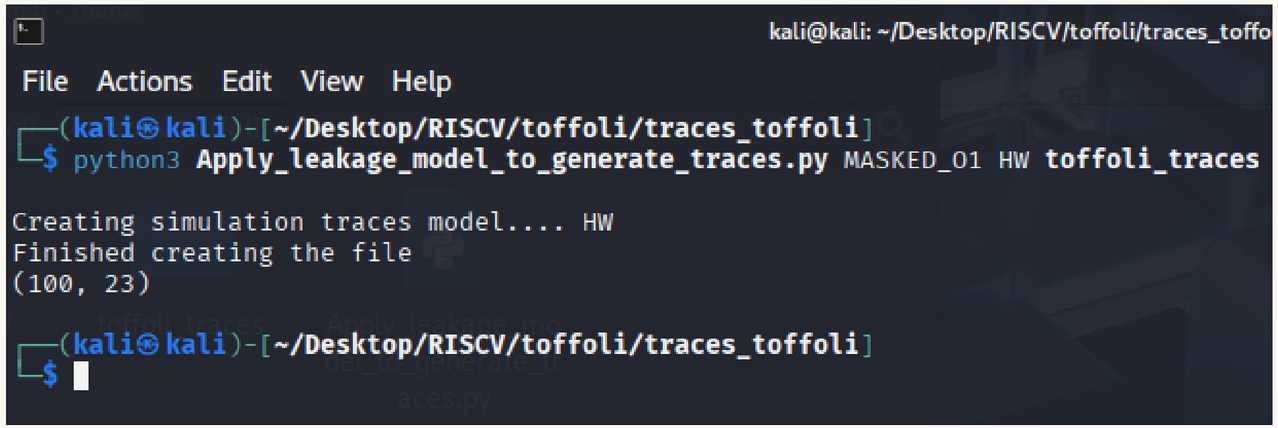
	A	B	C	D	E	F	G	H	I	J	K
1	PC	Machine	Ins	Type	Operands	t0	t1	t2	a0	a1	a2
2	0xc94	93020500	mv	ARITHMET	t0, a0	0x3ad2001	0xd467577	0x27b4a8c	0x814436b	0x9dc70	0x2dcbdb2
3	0xc98	13830500	mv	ARITHMET	t1, a1	0x814436b	0xd467577	0x27b4a8c	0x814436b	0x9dc70	0x2dcbdb2
4	0xc9c	93030600	mv	ARITHMET	t2, a2	0x814436b	0x9dc70	0x27b4a8c	0x814436b	0x9dc70	0x2dcbdb2
5	0xca0	b3c27200	xor	LOGICAL	t0, t0, t2	0x814436b	0x9dc70	0x2dcbdb2	0x814436b	0x9dc70	0x2dcbdb2
6	0xca4	33437300	xor	LOGICAL	t1, t1, t2	0xac8fed9	0x9dc70	0x2dcbdb2	0x814436b	0x9dc70	0x2dcbdb2
7	0xca8	13850200	mv	ARITHMET	a0, t0	0xac8fed9	0x2dc2075	0x2dcbdb2	0x814436b	0x9dc70	0x2dcbdb2
8	0xcac	93050300	mv	ARITHMET	a1, t1	0xac8fed9	0x2dc2075	0x2dcbdb2	0xac8fed9	0x9dc70	0x2dcbdb2

Locality Refresh

Applying Leakage Model

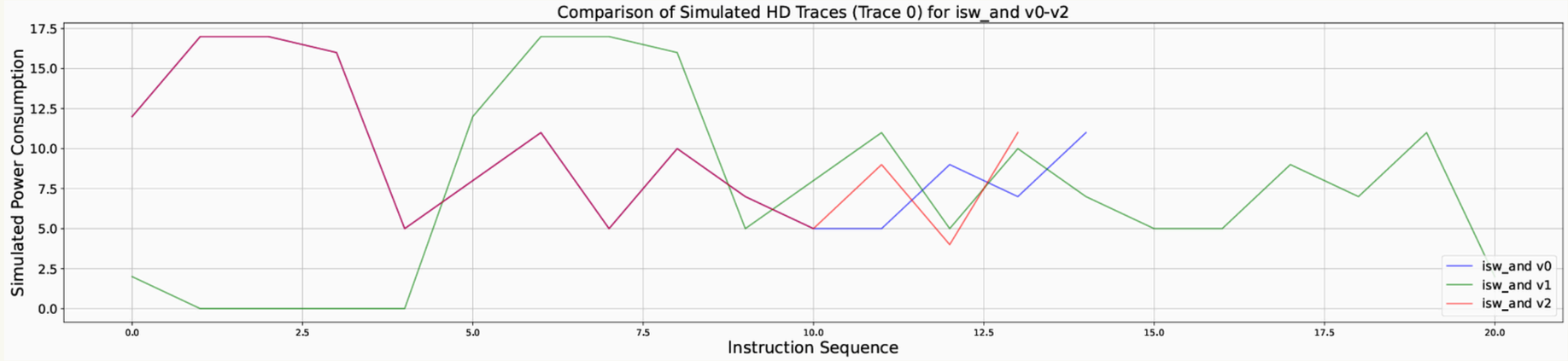
- Hamming Weight (HW): Measures the number of ‘1’s in a binary value. Applied to key intermediate values (e.g., a0 & b0, a1 & b1, a0 + a1, etc.).

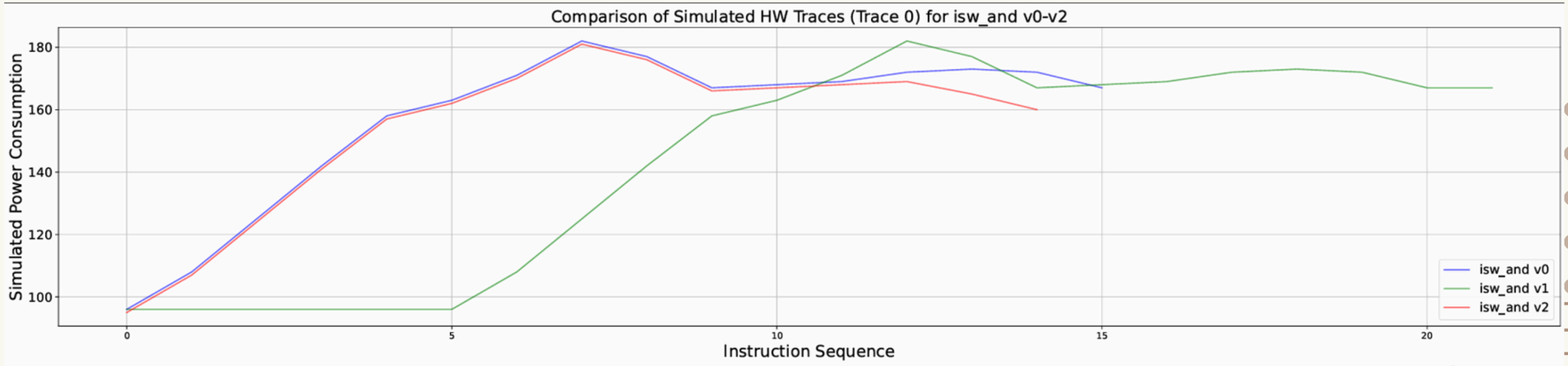
- Hamming Distance (HD): Measures the bitwise difference between two sequential values. Applied to transitions of intermediates (e.g., c_0_prev → c_0, reg(t) → reg(t+1)).



Applying Leakage Model

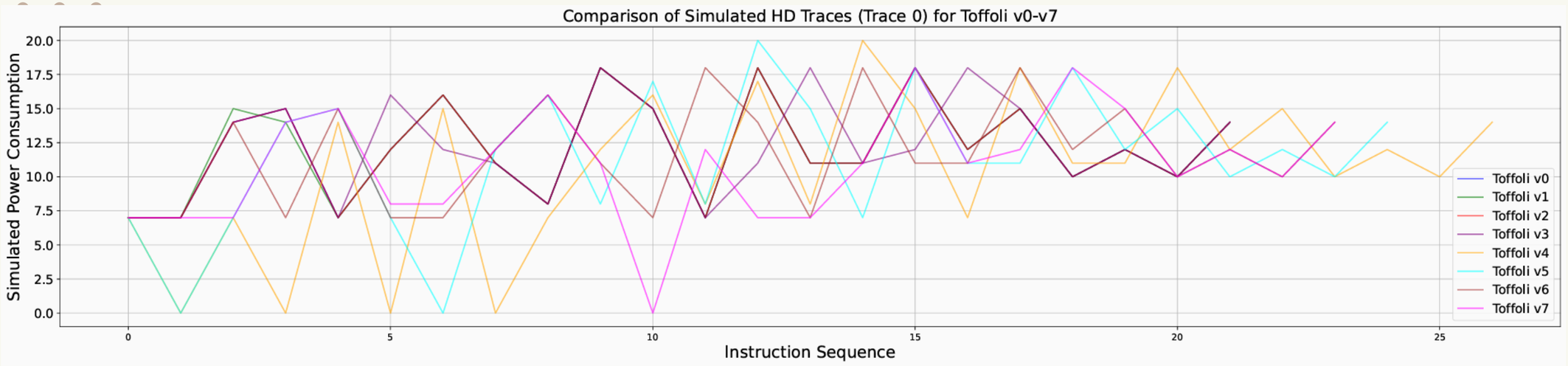
- HW/HD values used to label each trace.
- Label shape: (100,) → one leakage label per trace.
- Matched trace index-wise with .npy trace arrays.

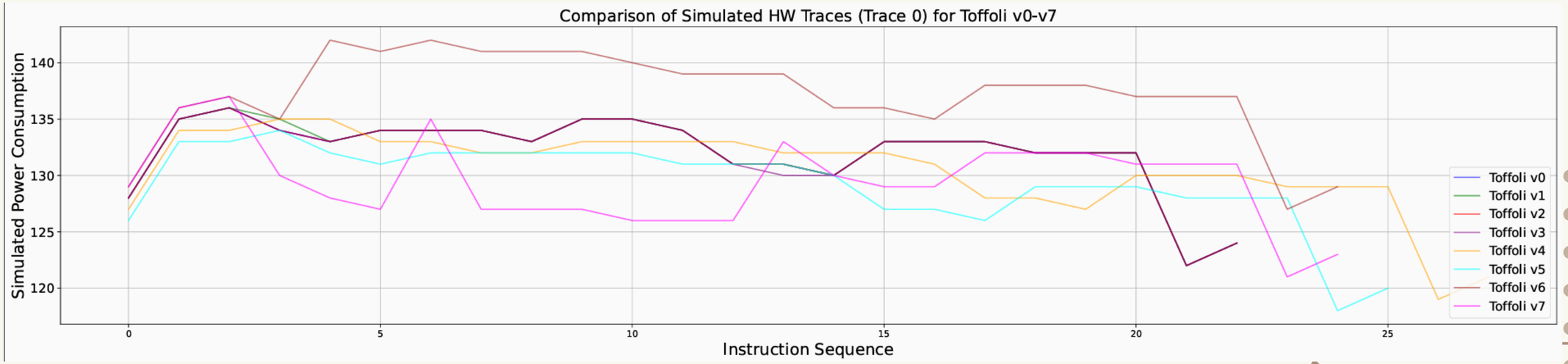




ISW AND

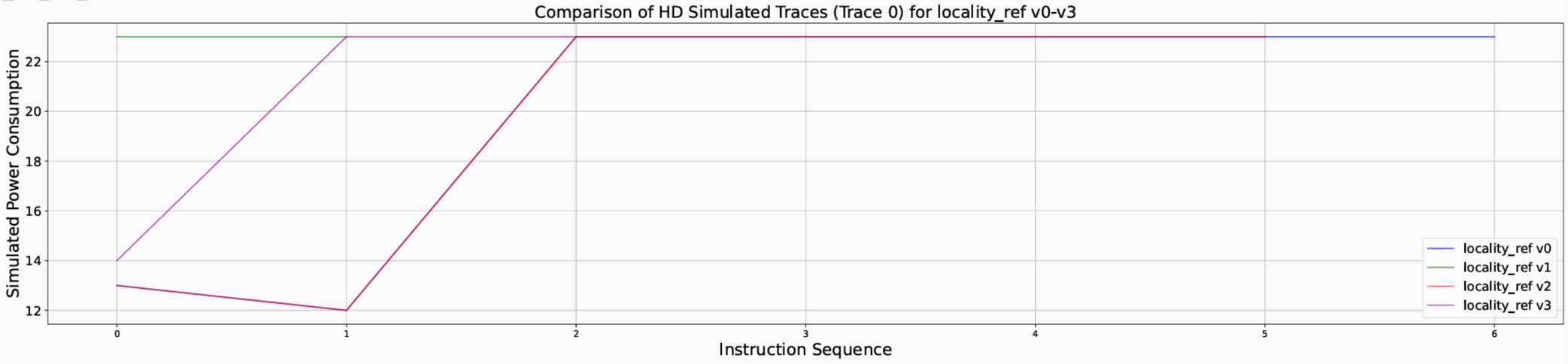
Applying Leakage Model

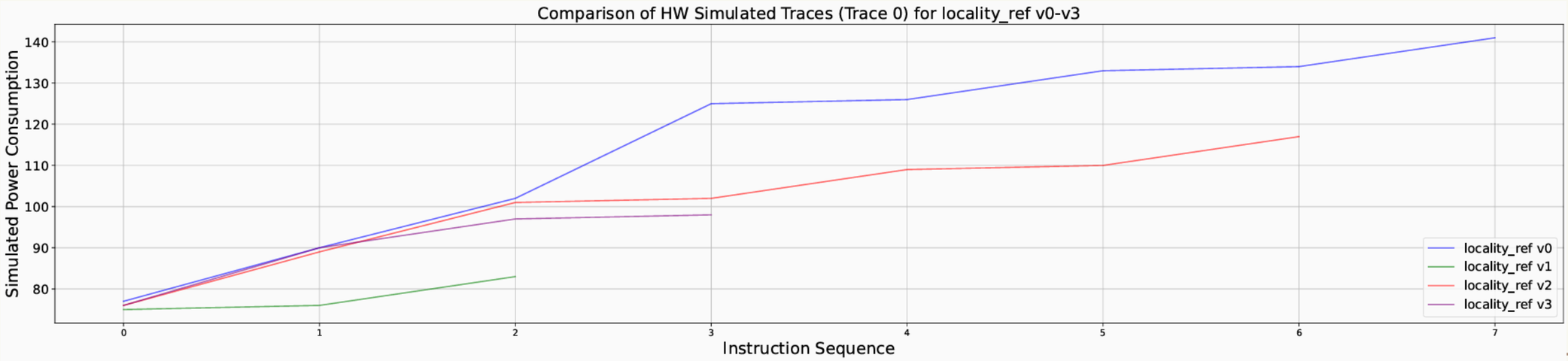




Toffoli Gate

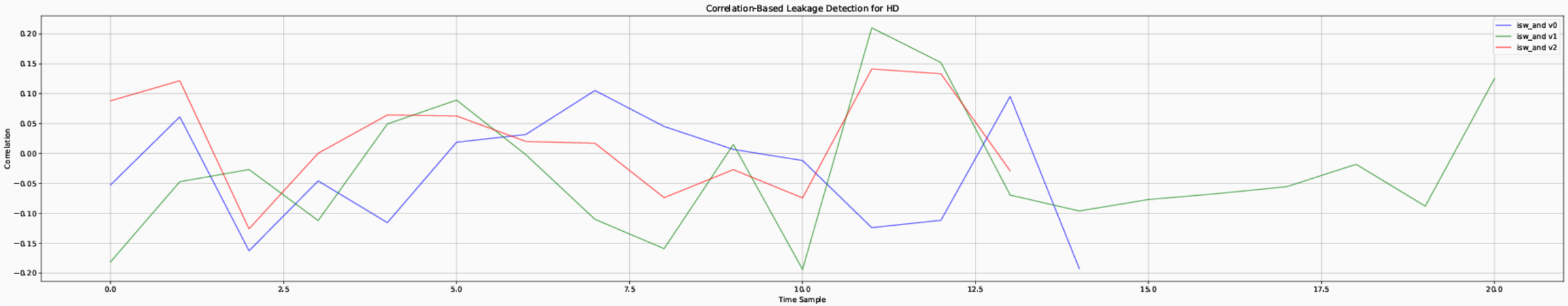
Applying Leakage Model

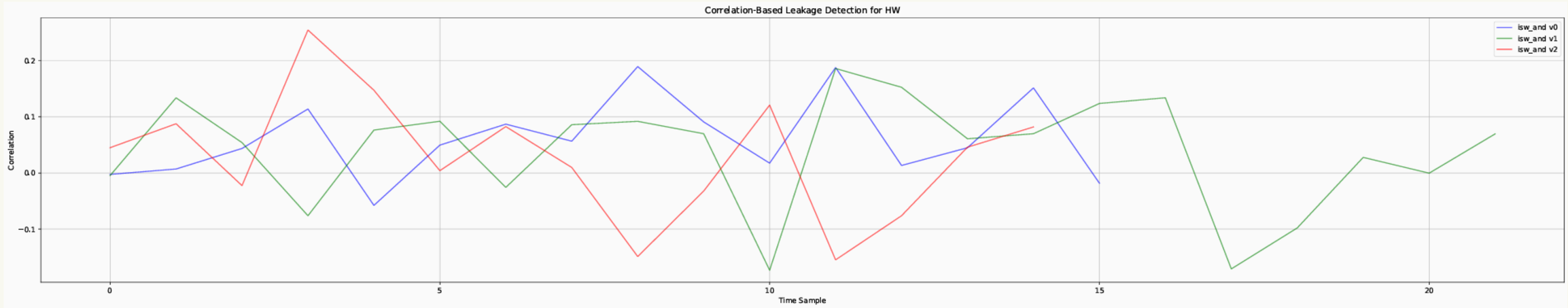




Locality Refresh

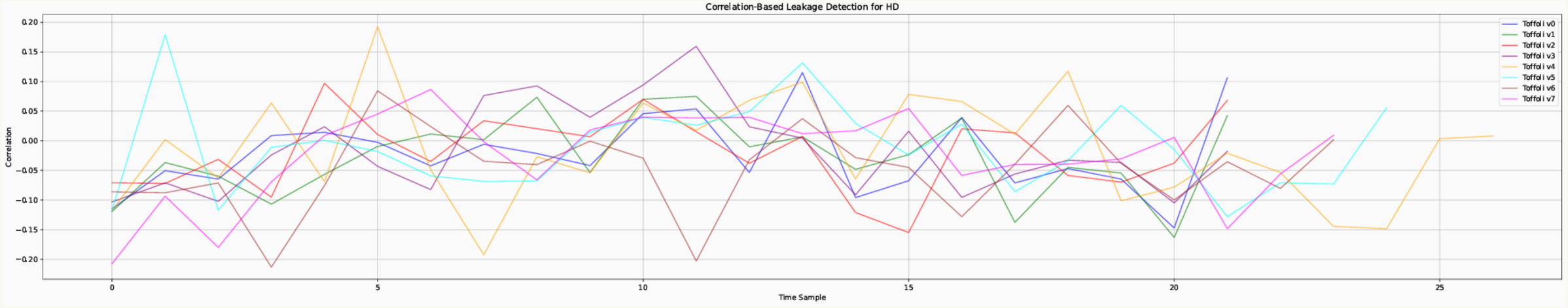
Correlation Analysis

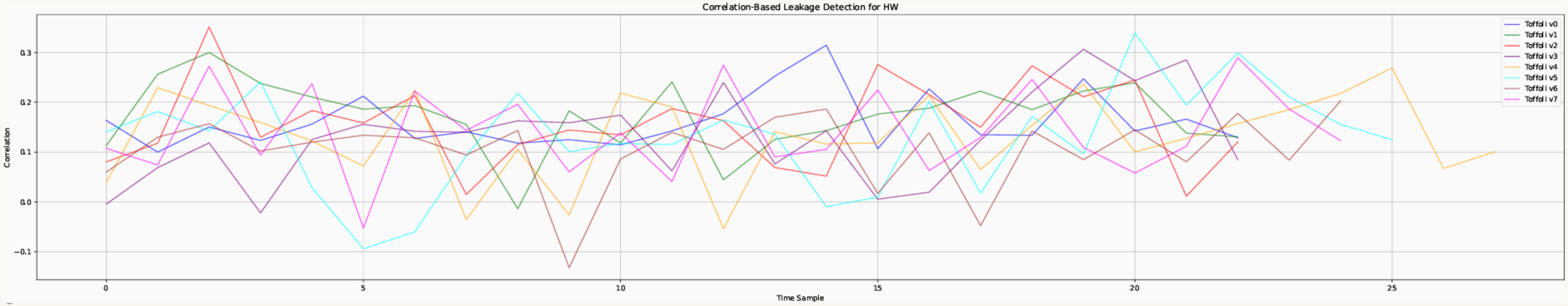




ISW AND

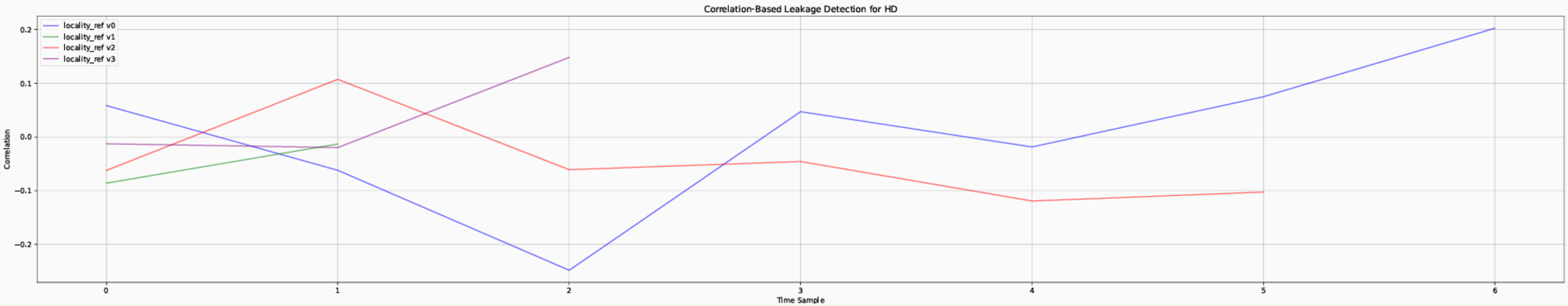
Correlation Analysis

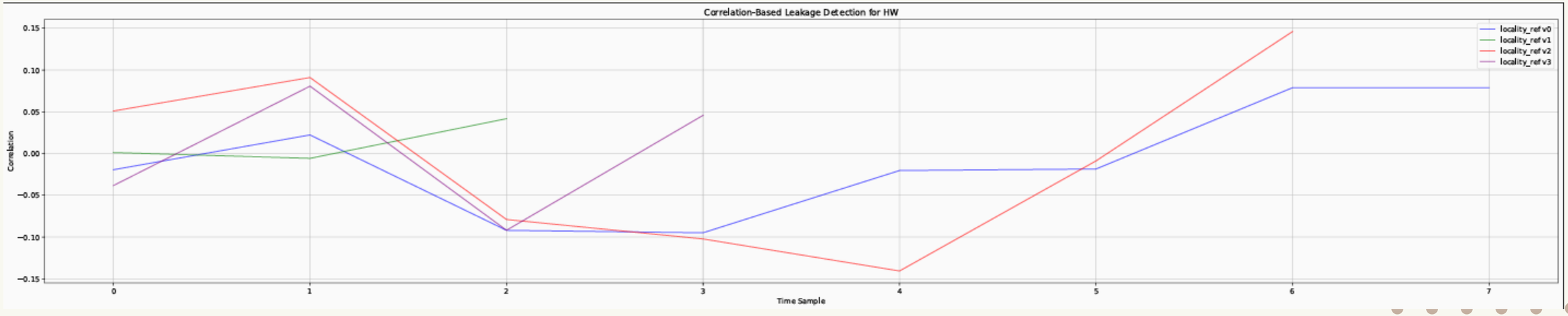




Toffoli Gate

Correlation Analysis





Locality Refresh

Conclusions

- Implemented and analyzed multiple masked gadget variants on PicoRV32 RISC-V core
- Generated synthetic side-channel traces through simulation
- Applied HW/HD leakage models to compute intermediates for each trace
- Conducted correlation analysis to assess leakage and masking effectiveness

Next Steps

- Expand trace collection using real hardware (e.g., CW-Husky, Saidoyoki board) to complement simulation
- Design and evaluate additional masking gadgets (e.g., higher-order masking, S-boxes)
- Train neural networks on the labeled trace dataset to evaluate: Gadget distinguishability, Masking scheme robustness, and Key recovery feasibility

REFERENCE

[1] John Gaspoz and Siemen Dhooghe. “Threshold implementations in software: Micro-architectural leakages in algorithms”. In: IACR Transactions on Cryp tographic Hardware and Embedded Systems (2023), pp. 155–179.

[2] Pantea Kiaei et al. “Saidoyoki: Evaluating side-channel leakage in pre-and post-silicon setting”. In: Cryptology ePrint Archive (2021).

[3] MiaoHeetal.“RTL-PSC:Automatedpowerside-channel leakage assessment at register-transfer level”. In: 2019 IEEE 37th VLSI Test Symposium (VTS). IEEE. 2019, pp. 1–6.

[4] AmishaSrivastavaetal.“SCAR:PowerSide-ChannelAnalysisatRTLLevel”. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2024)

THANK YOU