

Reliability Solutions + Rowhammer Mitigations

Peter Deutsch

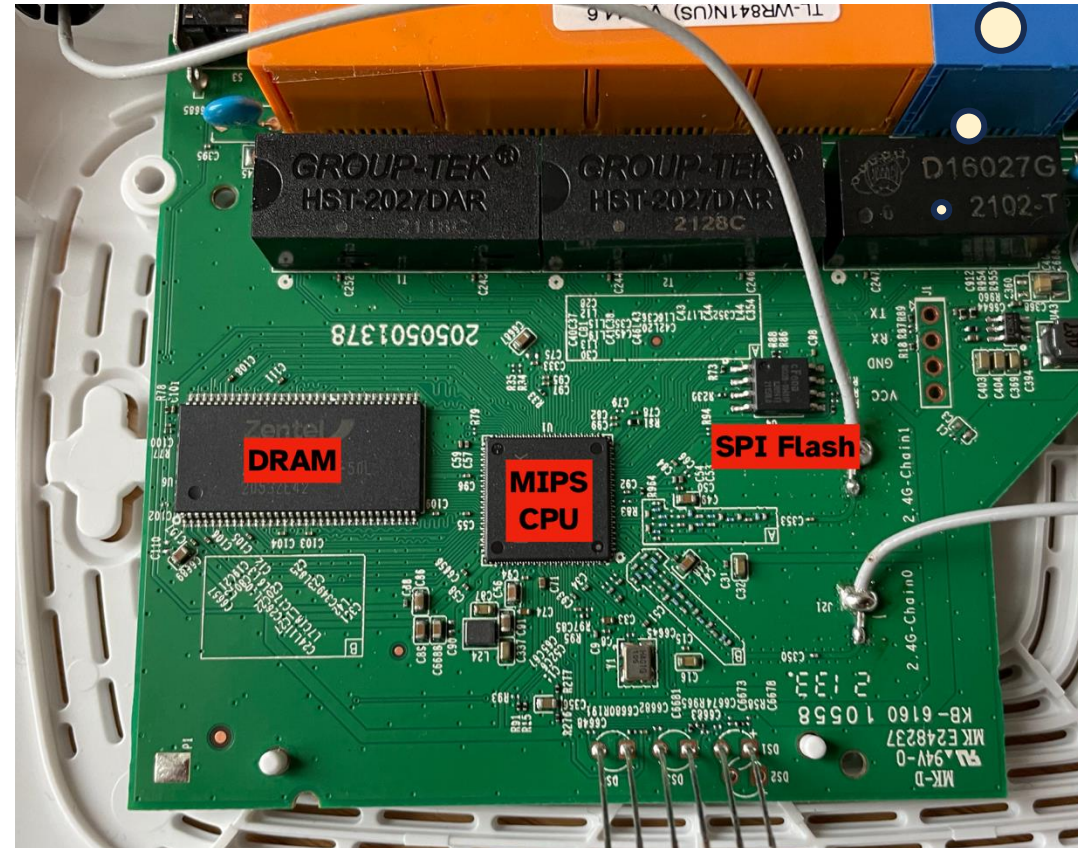
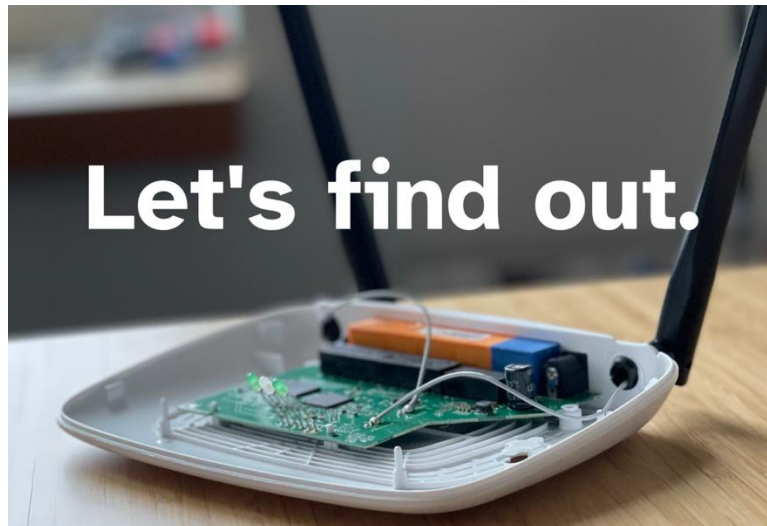
Spring 2025

Based on slides from Prof. Mengjia Yan

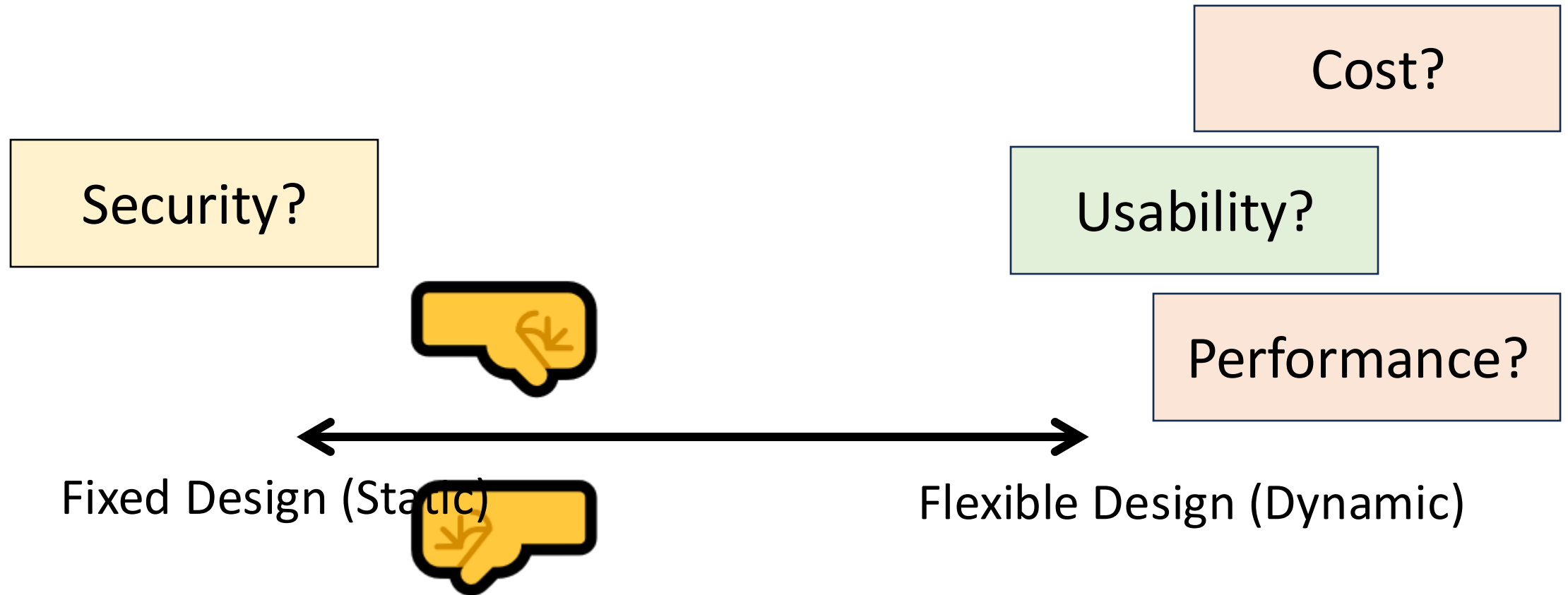


Recap Physical Attacks

Why is the debug port so easily accessible?



Mitigation Design Considerations



Physical Attack Mitigation Case Study

- IBM 4758
- Satisfy FIPS 140-1 Level 4

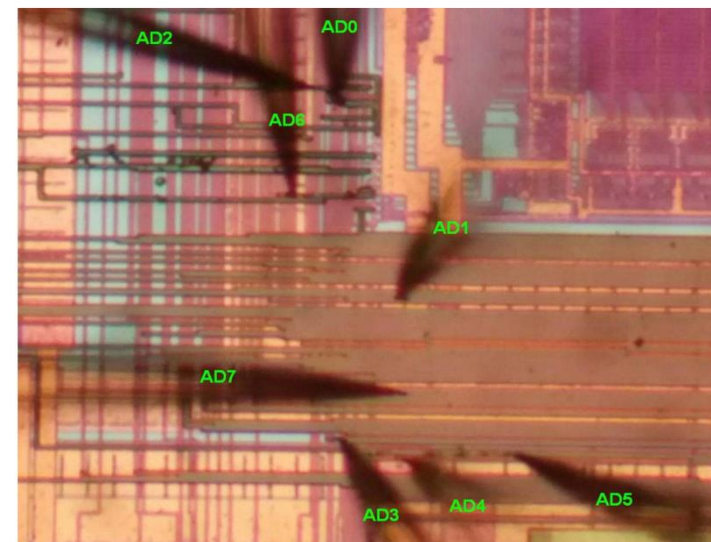
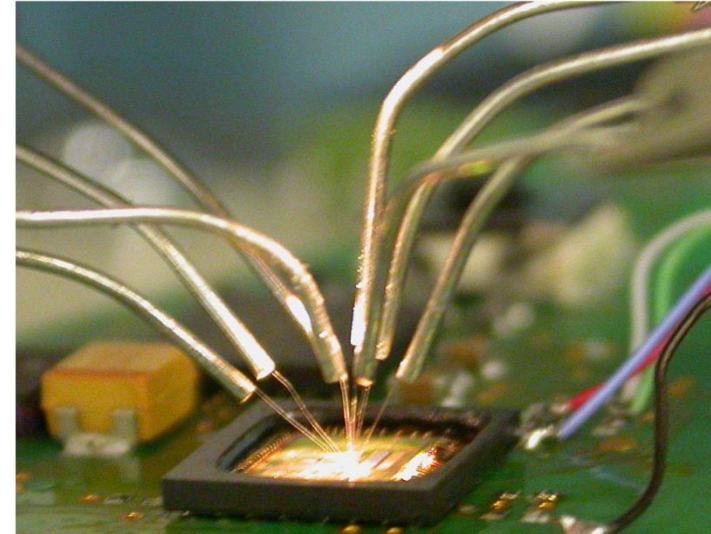
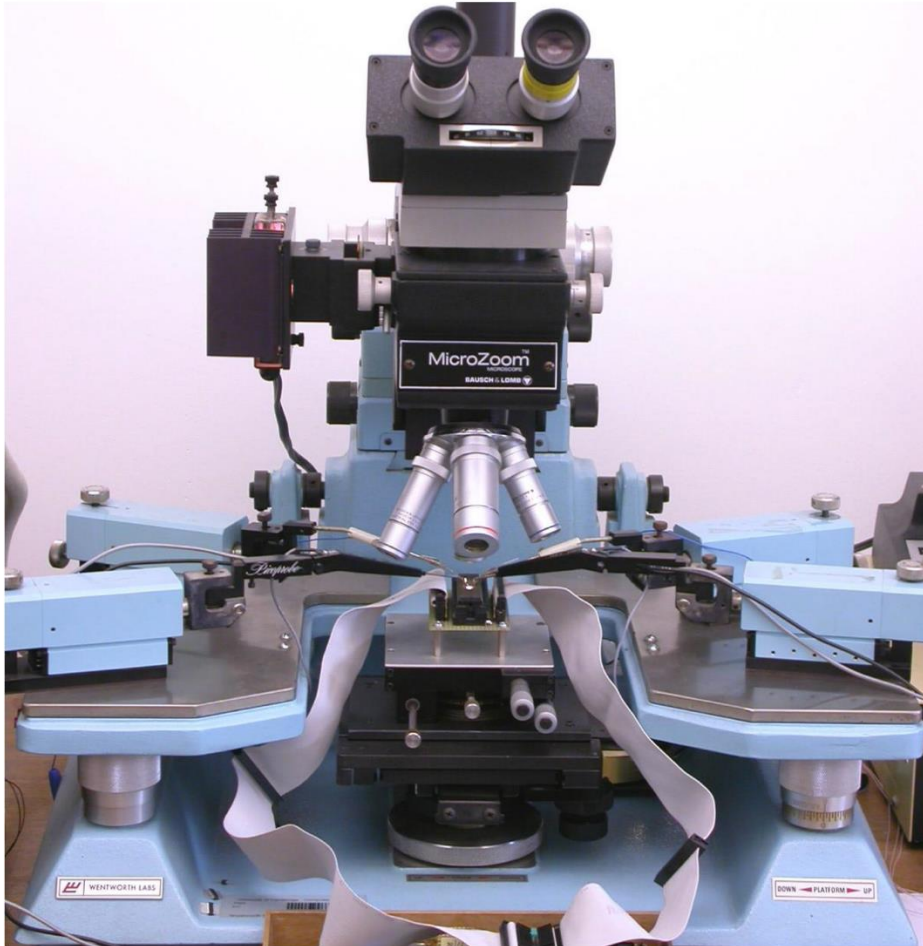
1.4 Security Level 4

Security Level 4 provides the highest level of security. Although most existing products do not meet this level of security, some products are commercially available which meet many of the Level 4 requirements. Level 4 physical security provides an envelope of protection around the cryptographic module. Whereas the tamper detection circuits of lower level modules may be bypassed, the intent of Level 4 protection is to detect a penetration of the device from any direction. For example, if one attempts to cut through the enclosure of the cryptographic module, the attempt should be detected and all critical security parameters should be zeroized. Level 4 devices are particularly useful for operation in a physically unprotected environment where an intruder could possibly tamper with the device.



Photo of IBM 4758 Cryptographic Coprocessor (courtesy of Steve Weingart) from <https://www.cl.cam.ac.uk/~rnc1/descrack/ibm4758.html>

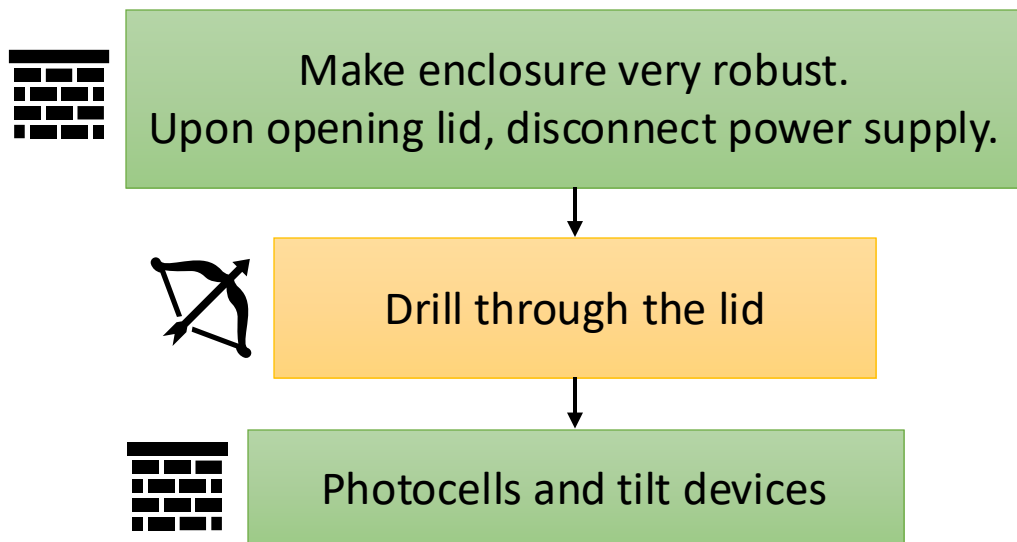
A Dedicated Attacker Can Mount Invasive Attacks!



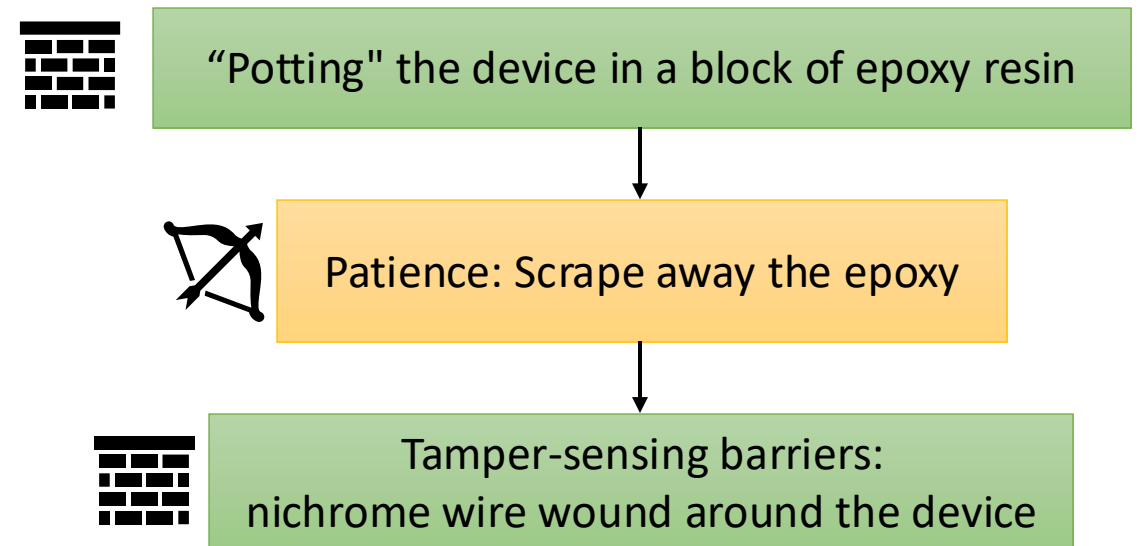
Physical Tamper Resistance

- Make it difficult for the attackers to get access to the chip...

Tamper Detection



Tampering Evident



IBM 4758 Secure Co-Processor

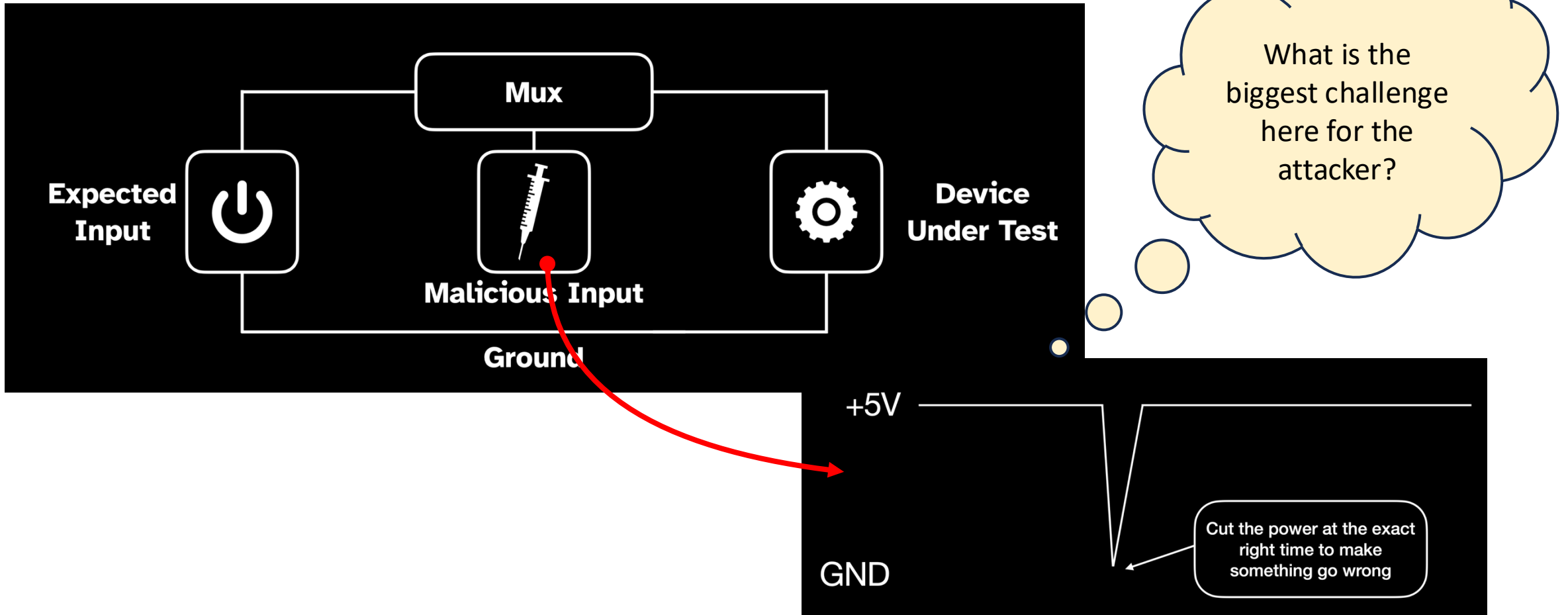
- Clock Glitching
 - Use phase locked loops and independently generated internal clocks
- Voltage Glitching:
 - Add detection and monitor circuits to watch for voltage changes
- X-ray fault injection:
 - A radiation sensor
- Electromagnetic side channels:
 - Solid aluminium shielding and a low-pass filter (a Faraday cage)



Photo of IBM 4758 Cryptographic Coprocessor (courtesy of Steve Weingart) from <https://www.cl.cam.ac.uk/~rnc1/descrack/ibm4758.html>

Expensive. Other secure processors only focus on a limited set of physical attacks.

Recap Fault Injection Attacks



Make Fault Injection Difficult

- Attacker's challenges:
 - Having control over the timing and the location of the fault
- What can be the high-level attack strategies?
 - Approach 1: Randomization to make the control more difficult
 - Approach 2: Detect anomalous behaviors of the system and block it (e.g., ECC)

Slides adapted from:

Miles Dai <milesdai@zerorisc.com>
zeroRISC Inc.

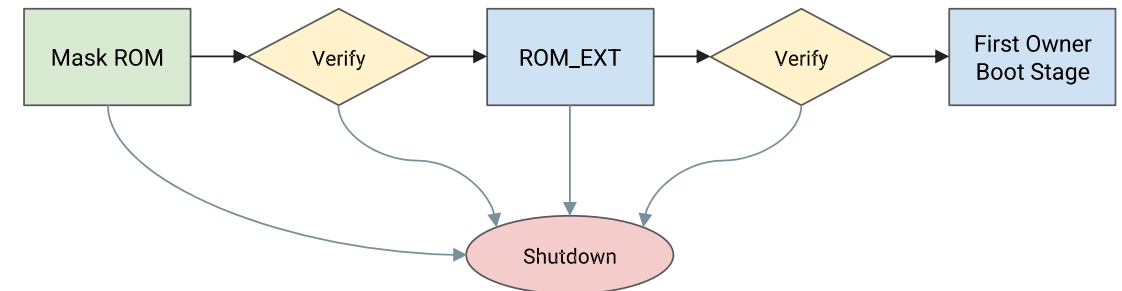
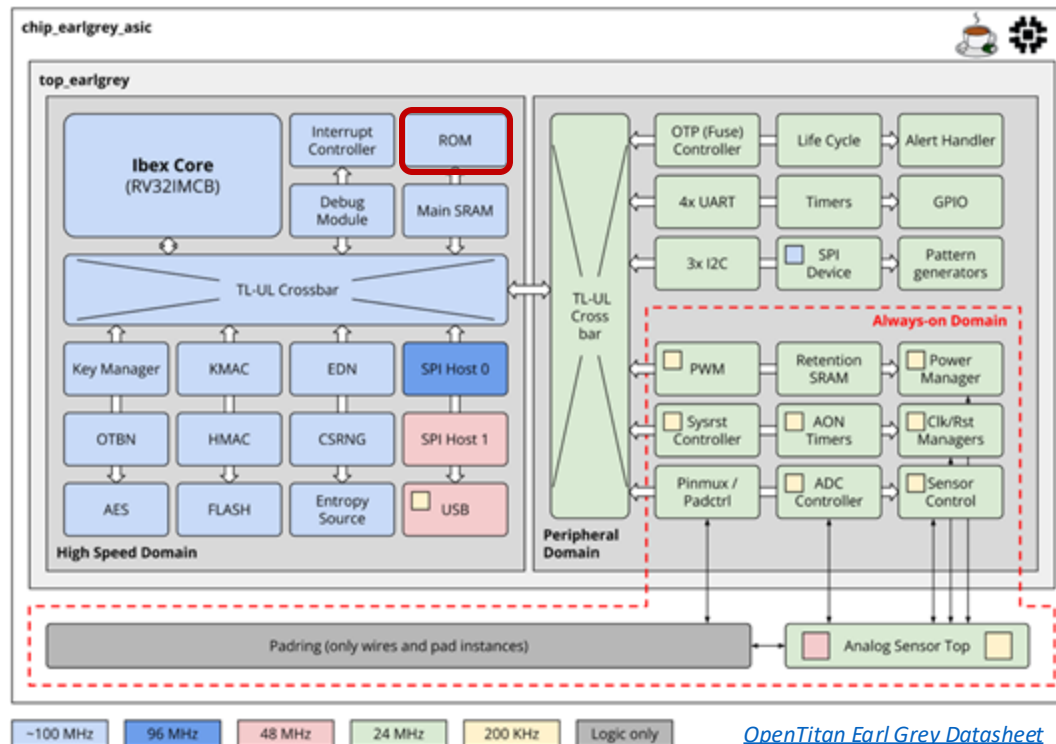
Arun Thomas
zeroRISC Inc., VP Engineering

Dominic Rizzo
zeroRISC Inc., CEO
OpenTitan Project Director

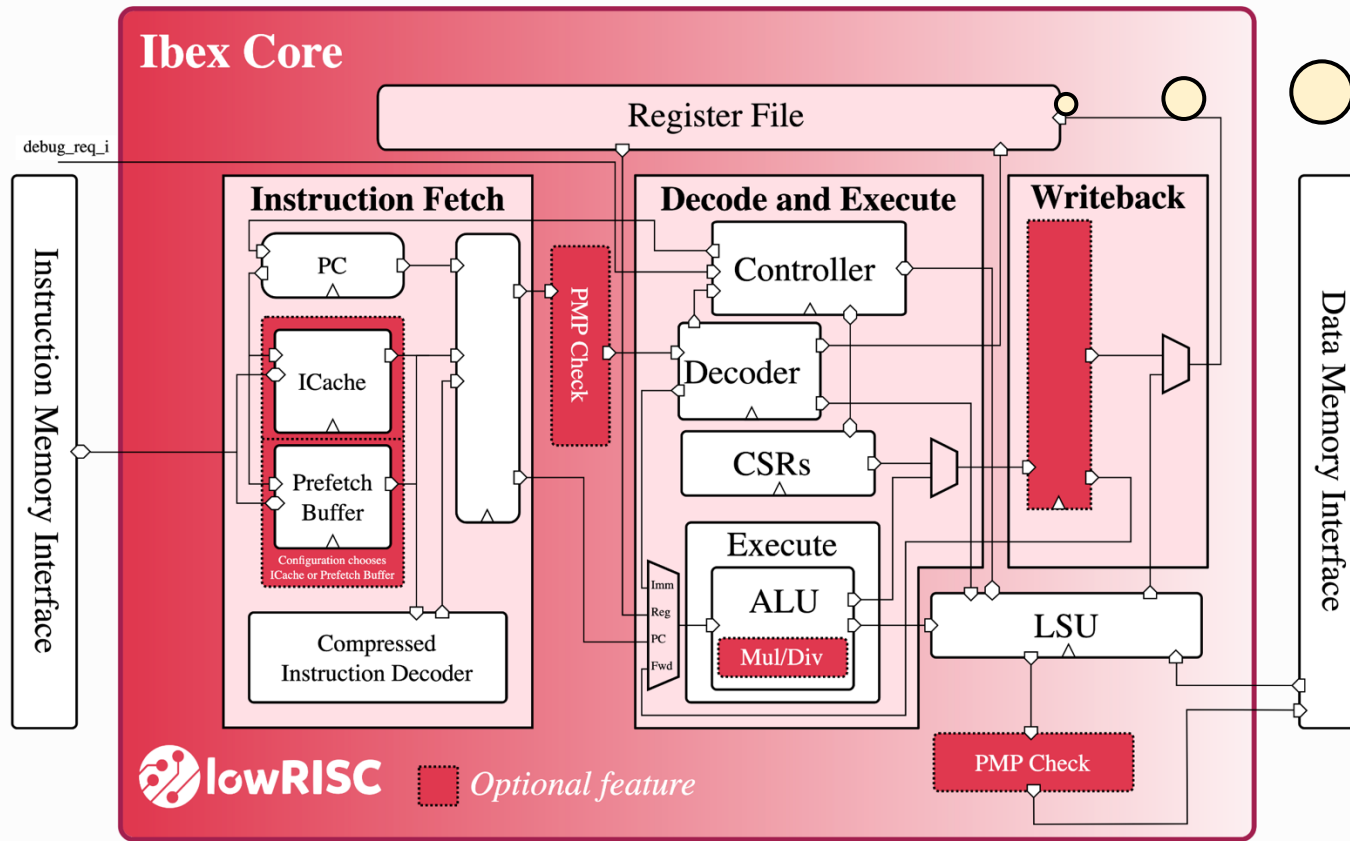
Software-Based Approaches: A collection of lessons from OpenTitan projects

OpenTitan Overview

- Goal: Establish Root of Trust, validate platform integrity (similar to TPM)
- Boot ROM: Configure critical hardware and verify next boot stage
 - Hardened C code



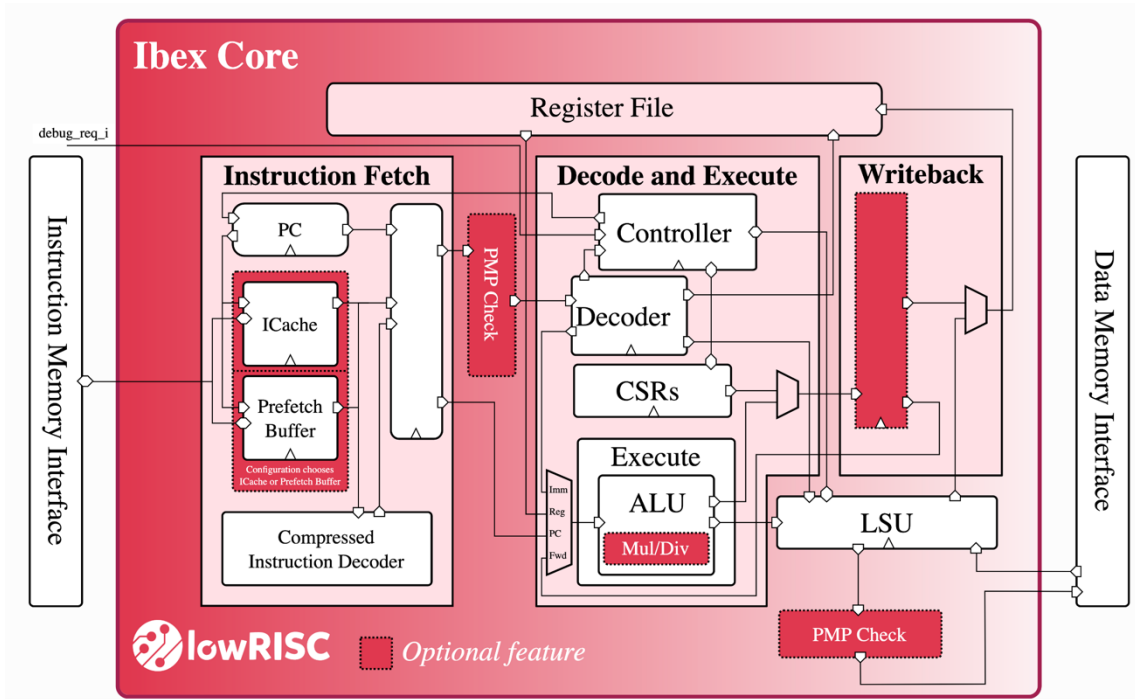
Hardware-Assisted Fault Detection



How can we protect ourselves against bit-flips in the register file?

- We can add an ECC to the register file to detect when a value has changed.
- What are the costs associated with ECC?

Challenge: When should we check ECC?



- We can't verify the entire contents of the ECC-protected register file every cycle...
- **Idea:** Verify the ECC immediately before reading a register.
- **Challenge:** Pipeline signaling is complicated, and a "register read" doesn't always come from the register file!
- **CVE-2024-57037:** Pipeline forwarding signal is misread, leading to ECC checks being skipped.

What can feasibly be done via fault injection?

The OpenTitan team has identified that some attacks are easier to perform than others!

- Easy
 - Skip one instruction
 - Glitch a register to all 0's or 1's
- Hard
 - Set a register to a specific value
 - Multiple precisely-timed glitches
 - Skipping a precise number of instructions

Example 1: Multi-bit Encodings

Make the attacker's life more difficult:

Instead of requiring the attacker to glitch a register to all 0's or 1's, force them to set a register to a specific value...

Multi-bit (MUBI) Encodings

```
enum lifecycle_state {  
    // Unlocked test state with debug functions.  
    kLcStateTest,  
  
    // Production life cycle state.  
    kLcStateProd,  
  
    // RMA life cycle state.  
    kLcStateRma,  
};
```



```
enum lifecycle_state {  
    // Unlocked test state with debug functions.  
    kLcStateTest = 0xb2865fbb,  
  
    // Production life cycle state.  
    kLcStateProd = 0x65f2520f,  
  
    // RMA life cycle state.  
    kLcStateRma = 0xcf8cfaab,  
};
```

What integers do we use under the hood?

Multi-bit (MUBI) Encodings

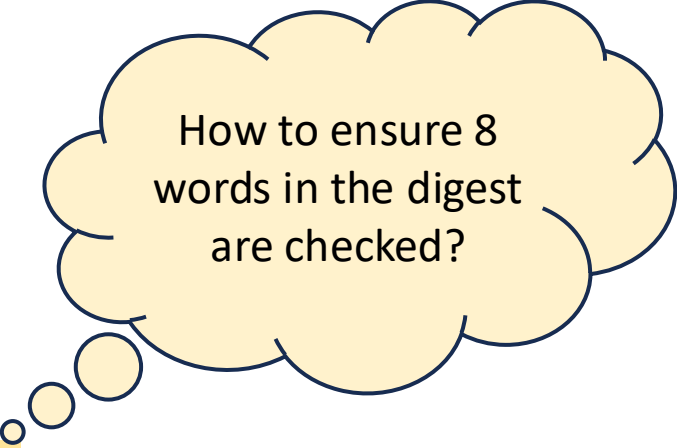
```
/**
 * Lifecycle states.
 *
 * This is a condensed version of the 24 possible life cycle states where
 * TEST_UNLOCKED_* states are mapped to `kLcStateTest` and invalid states where
 * CPU execution is disabled are omitted.
 */
*
* Encoding generated with
* $ ./util/design/sparse-fsm-encode.py -d 6 -m 5 -n 32 \
*   -s 2447090565 --language=c
*
* Minimum Hamming distance: 13
* Maximum Hamming distance: 19
* Minimum Hamming weight: 15
* Maximum Hamming weight: 20
*/
typedef enum lifecycle_state {
  /**
   * Unlocked test state where debug functions are enabled.
   */
  kLcStateTest = 0xb2865fbb,
  /**
   * Development life cycle state where limited debug functionality is
   * available.
   */
  kLcStateDev = 0x0b5a75e0,
  ...
} lifecycle_state_t;
```

Example 2: Hash Checking

Hash Checking

```
typedef struct hmac_digest {  
    uint32_t digest[8];  
} hmac_digest_t;
```

```
typedef struct boot_data {  
    hmac_digest_t digest; // SHA-256 digest of boot data.  
    uint32_t min_security_version_rom_ext;  
    uint32_t min_security_version_b10;  
} boot_data_t;
```



How to ensure 8 words in the digest are checked?

Hash Checking

```
static const uint32_t shares[8] = {  
    0xe021e1a9, 0xf81e8365, 0xbf8322db, 0xc7a37080,  
    0xdd8ce33f, 0x7585d574, 0x951777af, 0x271a933f,  
};
```

← Pre-compute shares

Hash Checking

```
static const uint32_t shares[8] = {  
    0xe021e1a9, 0xf81e8365, 0xbf8322db, 0xc7a37080,  
    0xdd8ce33f, 0x7585d574, 0x951777af, 0x271a933f,  
};
```

← Pre-compute shares

```
bool check_digest(const boot_data_t *boot_data) {  
    rom_error_t error = 0;  
    hmac_digest_t act_digest;  
    boot_data_digest_compute(boot_data, &act_digest);
```

← Compute the digest

```
}
```

Hash Checking

```
static const uint32_t shares[8] = {  
    0xe021e1a9, 0xf81e8365, 0xbf8322db, 0xc7a37080,  
    0xdd8ce33f, 0x7585d574, 0x951777af, 0x271a933f,  
};
```

← Pre-compute shares

```
bool check_digest(const boot_data_t *boot_data) {  
    rom_error_t error = 0;  
    hmac_digest_t act_digest;  
    boot_data_digest_compute(boot_data, &act_digest);
```

← Compute the digest

```
    for (size_t i = 0; i < 8; ++i) {  
        error ^= boot_data->digest[i] ^ act_digest[i] ^ shares[i];  
    }  
    return error == kErrorOk; //kErrorOk is the xored result of the shares  
}
```

← Generate the valid error value from the shares.

Hash Checking

```
static const uint32_t shares[8] = {  
    0xe021e1a9, 0xf81e8365, 0xbf8322db, 0xc7a37080,  
    0xdd8ce33f, 0x7585d574, 0x951777af, 0x271a933f,  
};
```

```
bool check_digest(const boot_data_t *boot_data) {  
    rom_error_t error = 0;  
    hmac_digest_t act_digest;  
    boot_data_digest_compute(boot_data, &act_digest);
```

```
    for (size_t i = 0; i < 8; ++i) {  
        error ^= boot_data->digest[i] ^ act_digest[i] ^ shares[i];  
    }  
    return error == kErrorOk; //kErrorOk is the xored result of the shares  
}
```

Any additional
hardening
opportunities?

Pre-compute shares

Compute the digest

Generate the valid
error value from the
shares.

Example 3: Redundant Condition Checks

Force the attacker:

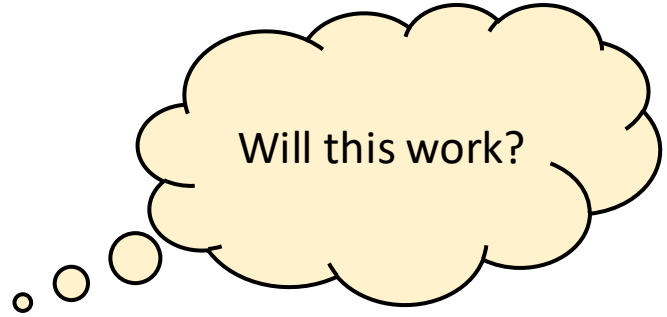
Skip one instruction → Skipping a precise number of instructions

Redundant Condition Checks

```
if (lc_state != kLcStateProd) {  
    assert();  
}
```

Redundant Condition Checks – First Attempt

```
if (lc_state != kLcStateProd) {  
    assert();  
}  
assert(lc_state == kLcStateProd);
```



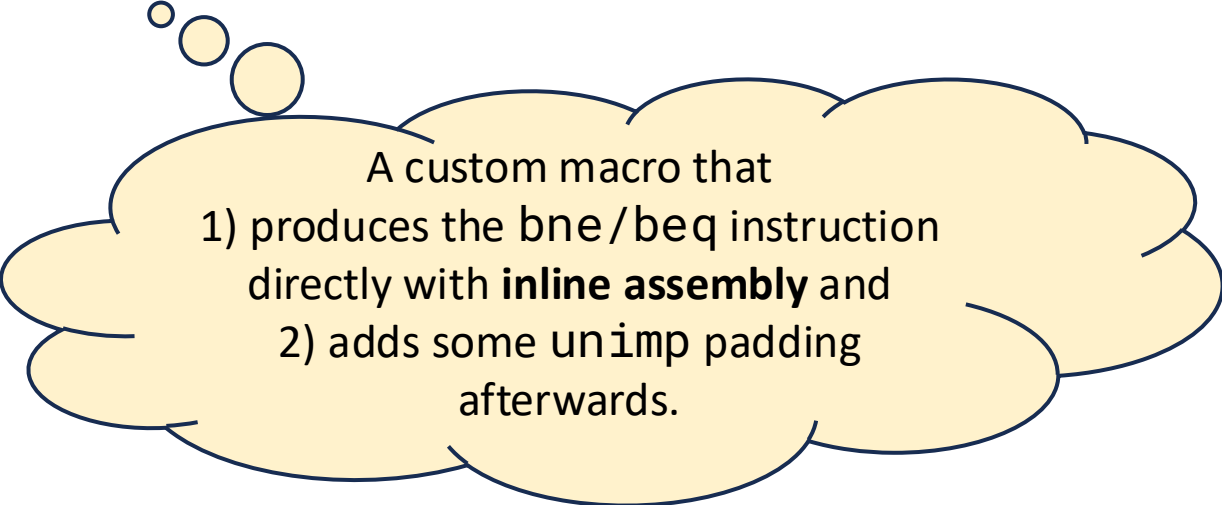
Will this work?

Redundant Condition Checks - launder32

```
inline uint32_t launder32(uint32_t val) {  
    asm volatile("" : "+r"(val));  
    return val;  
}
```

Redundant Condition Checks

```
if (launder32(lc_state) != LcStateProd) {  
    assert();  
}  
HARDENED_CHECK_EQ(lc_state, LcStateProd);
```



A custom macro that
1) produces the bne/beq instruction directly with **inline assembly** and
2) adds some unimp padding afterwards.

Redundant Condition Checks

C	Assembly
<pre>if (launder32(lc_state_check) != lc_state) { HARDENED_TRAP(); } HARDENED_CHECK_EQ(lc_state_check, lc_state);</pre>	

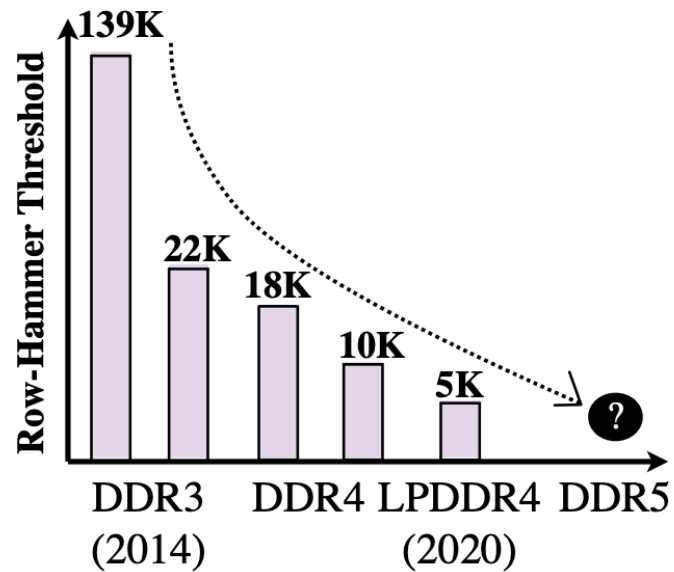
Redundant Condition Checks

C	Assembly
<pre>if (launder32(lc_state_check) != lc_state) { HARDENED_TRAP(); } HARDENED_CHECK_EQ(lc_state_check, lc_state);</pre>	<pre>/proc/self/cwd/sw/device/silicon_creator/rom/rom.c:306 if (launder32(lc_state_check) != lc_state) { 91b0: lw a2, -390(s1) 91b4: beq a1, a2, 91c4 /proc/self/cwd/sw/device/silicon_creator/rom/rom.c:307 HARDENED_TRAP(); 91b8: unimp 91ba: unimp 91bc: unimp 91be: unimp</pre>

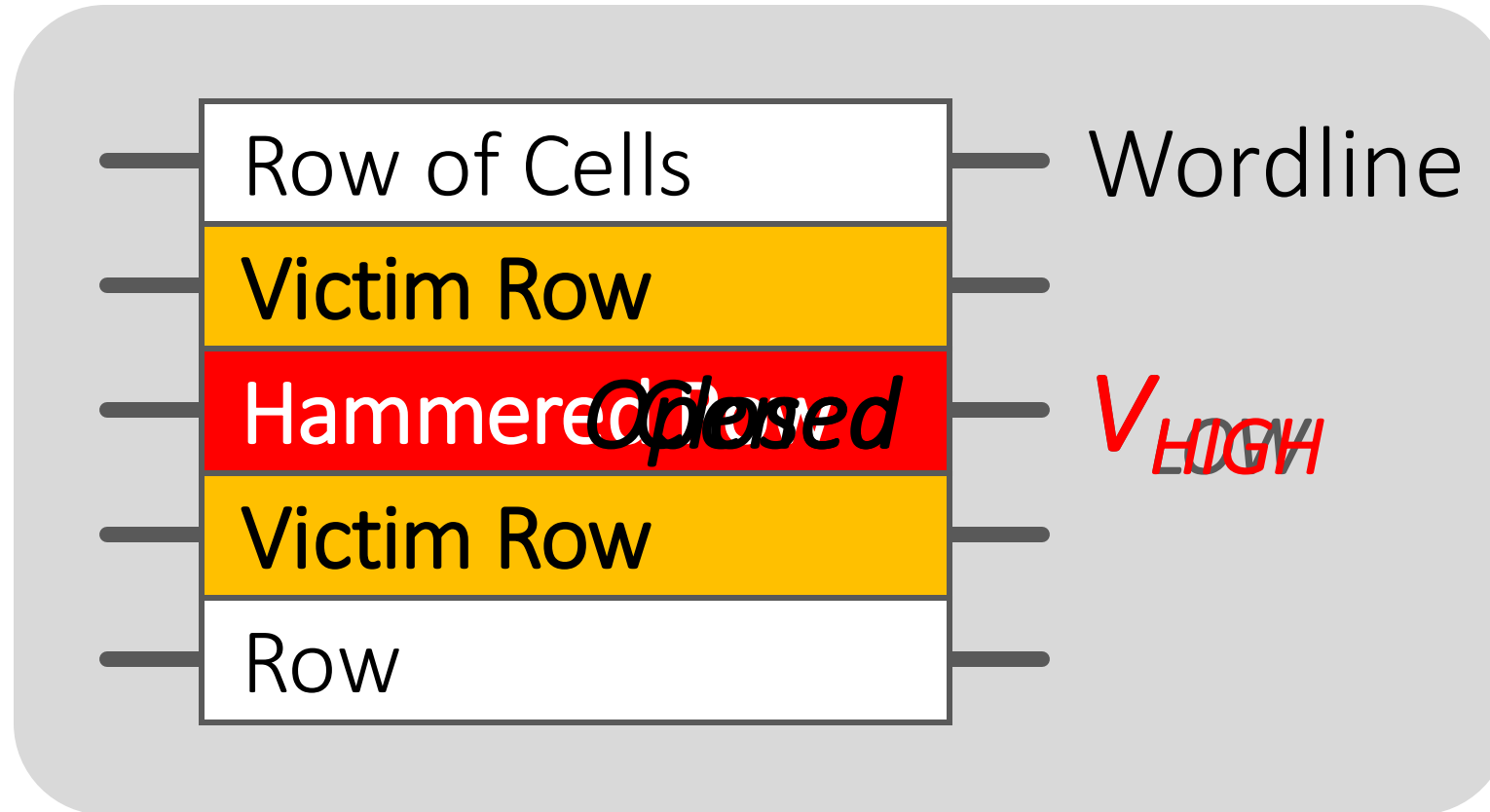
Redundant Condition Checks

C	Assembly
<pre>if (launder32(lc_state_check) != lc_state) { HARDENED_TRAP(); } HARDENED_CHECK_EQ(lc_state_check, lc_state);</pre>	<pre>/proc/self/cwd/sw/device/silicon_creator/rom/rom.c:306 if (launder32(lc_state_check) != lc_state) { 91b0: lw a2, -390(s1) 91b4: beq a1, a2, 91c4 /proc/self/cwd/sw/device/silicon_creator/rom/rom.c:307 HARDENED_TRAP(); 91b8: unimp 91ba: unimp 91bc: unimp 91be: unimp /proc/self/cwd/sw/device/silicon_creator/rom/rom.c:309 HARDENED_CHECK_EQ(lc_state_check, lc_state); 91c0: lw a1, -390(s1) 91c4: beq a0, a1, 91d0 91c8: unimp 91ca: unimp 91cc: unimp 91ce: unimp</pre>

RowHammer Mitigations: A Numbers Game



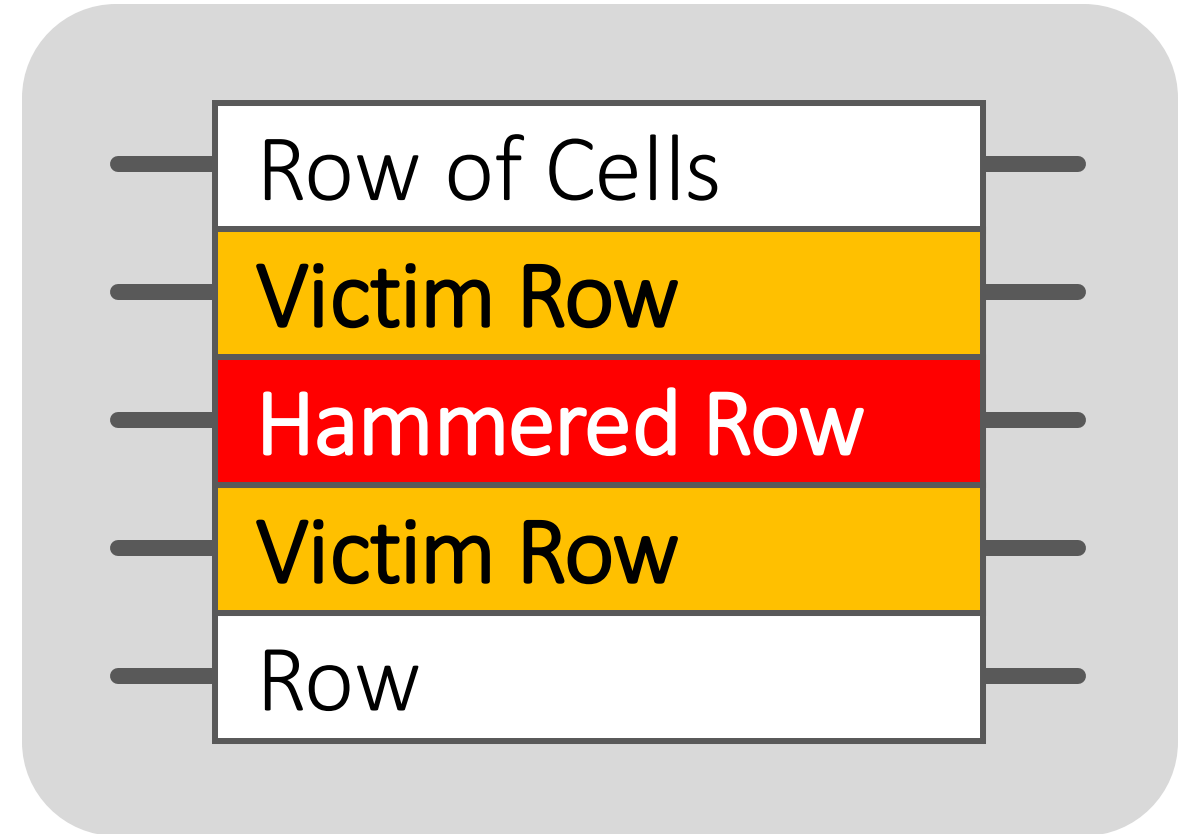
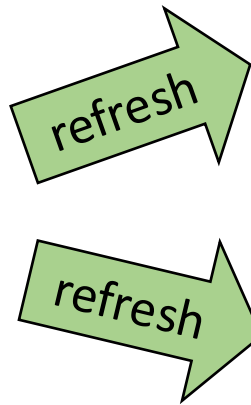
Recap RowHammer



Observation: Repeatedly accessing a row enough times **between refreshes** can cause disturbance errors in nearby rows

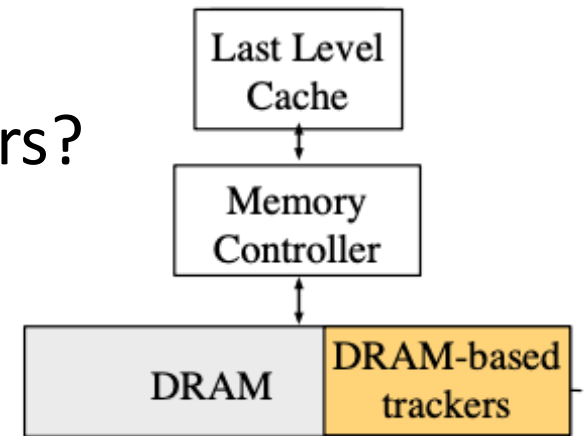
Probabilistic Row Activation

- Pick a probability “ p ”
- Question: how to pick “ p ”?
What is the consequence?



Counter-based Row Activation

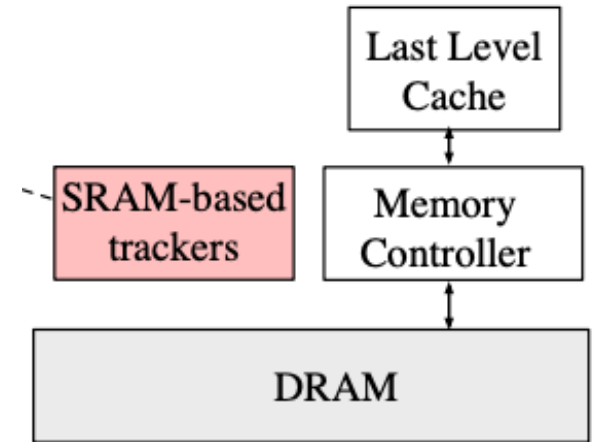
- Maintain a counter to track the number of accesses per row
 - Increment the counter when accessing a row
 - When reaching a threshold, activate the neighboring rows
 - After activating, reset the counter
- How much storage overhead for the row-access counters?
 - Example: 8GB memory with 1M rows, each counter 2 bytes
 - Answer?
- What factors affect the performance overhead?



SRAM-based Trackers

- Naïve: one counter per row
 - What is the problem?
- Do it smartly: using the Misra-Gries Algorithm
 - The Rowhammer tracking problem is very similar to the frequent elements problem
 - Given a stream of W items, the algorithm identifies **all the items** that appear more than T times, as long as

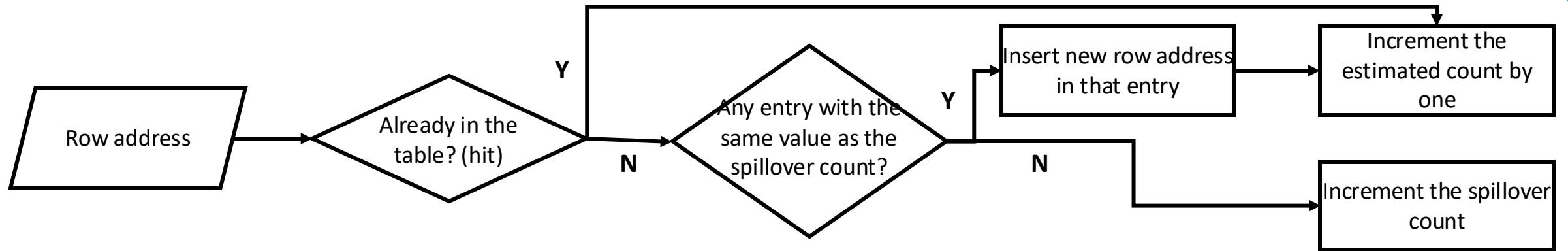
$$N_{entry} > W/T - 1$$



Row Address	Count
0x1010	5
0x2020	7
...	...
Spillover Count	2

N_{entry}

Graphene Aggressor Tracking



Row Address	Count
0x1010	5
0x2020	7
0x3030	3

Row Address	Count
0x1010	6
0x2020	7
0x3030	3

Row Address	Count
0x1010	6
0x2020	7
0x3030	3

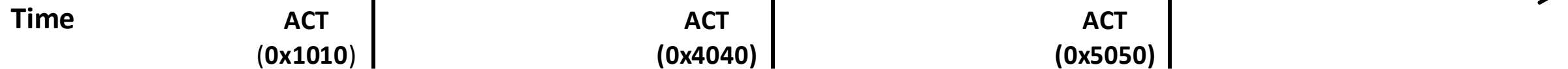
Row Address	Count
0x1010	6
0x2020	7
0x5050	4

Spillover Count	Count
Spillover Count	2

Spillover Count	Count
Spillover Count	2

Spillover Count	Count
Spillover Count	3

Spillover Count	Count
Spillover Count	3



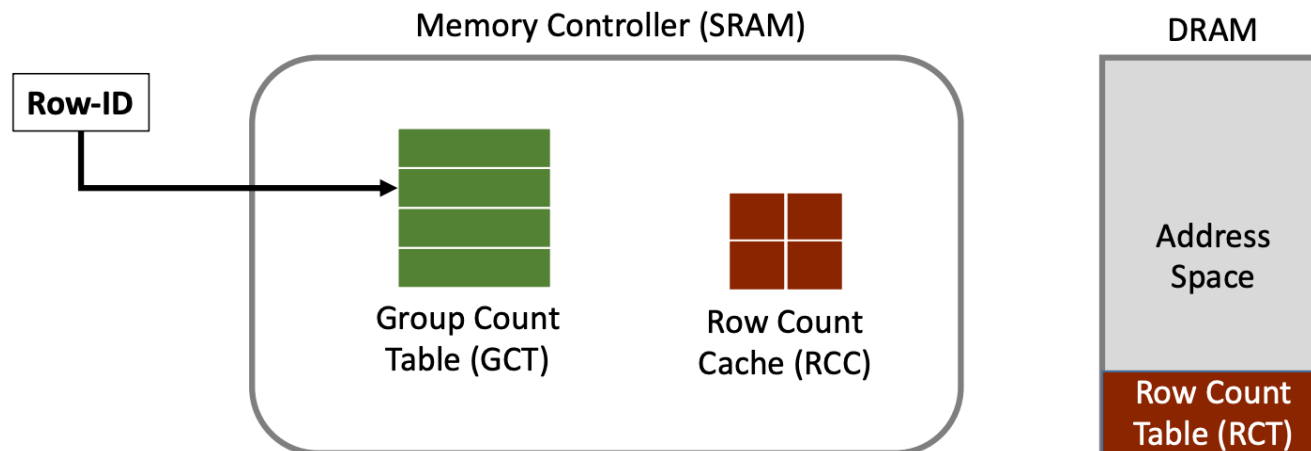
Graphene Analysis

$$N_{entry} > W/T - 1$$

- In the original paper (2020)
 - W Max number of ACTs in a refresh window: 1,360K
 - T Threshold for aggressor tracking: 12.5K (actual threshold = 25K)
 - N_{entry} Number of table entries: 108
 - Each entry: 16 bits for row address; 15 bits for counting value up to T
 - Memory type: CAM
- In a recent paper, assuming 16GB memory (2022)
 - T Threshold for aggressor tracking: 250 (actual threshold = 500)
 - N_{entry} Number of table entries: 5440 (50x more)

More Ideas: Hydra

- Profile a lot of applications and find *Rowhammer is a race against time*
 - Access many rows few times ✓
 - Access few rows many times ✓
 - Access many rows many times ✗



Key idea: use DRAM to get scalable tracking, and SRAM to avoid performance overheads

Mitigation Design Considerations

