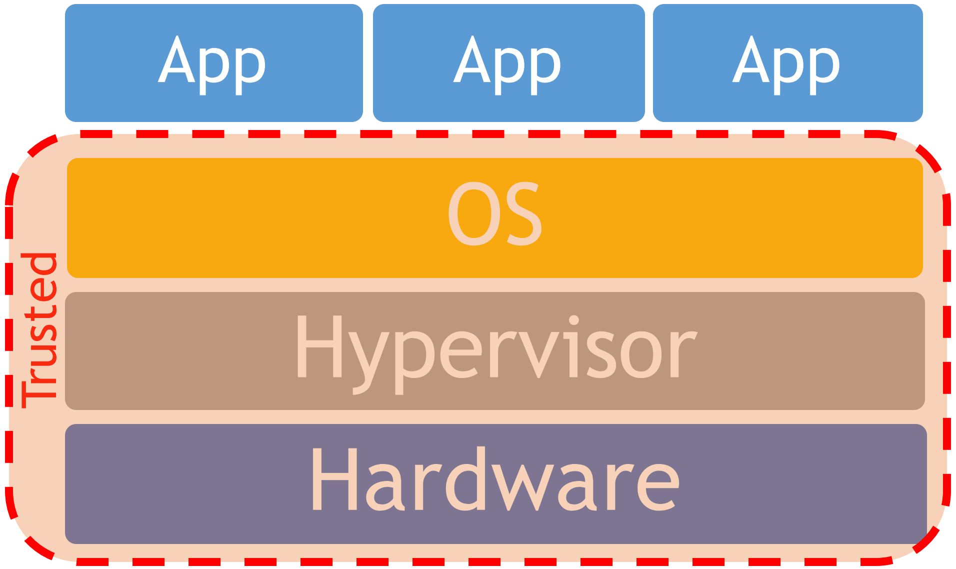
CopyCat: Controlled Instruction-Level Attacks on Enclaves

- **Daniel Moghimi**
- Jo Van Bulck
- Nadia Heninger
- Frank Piessens
- Berk Sunar

Intel Labs – Sept. 10 2020

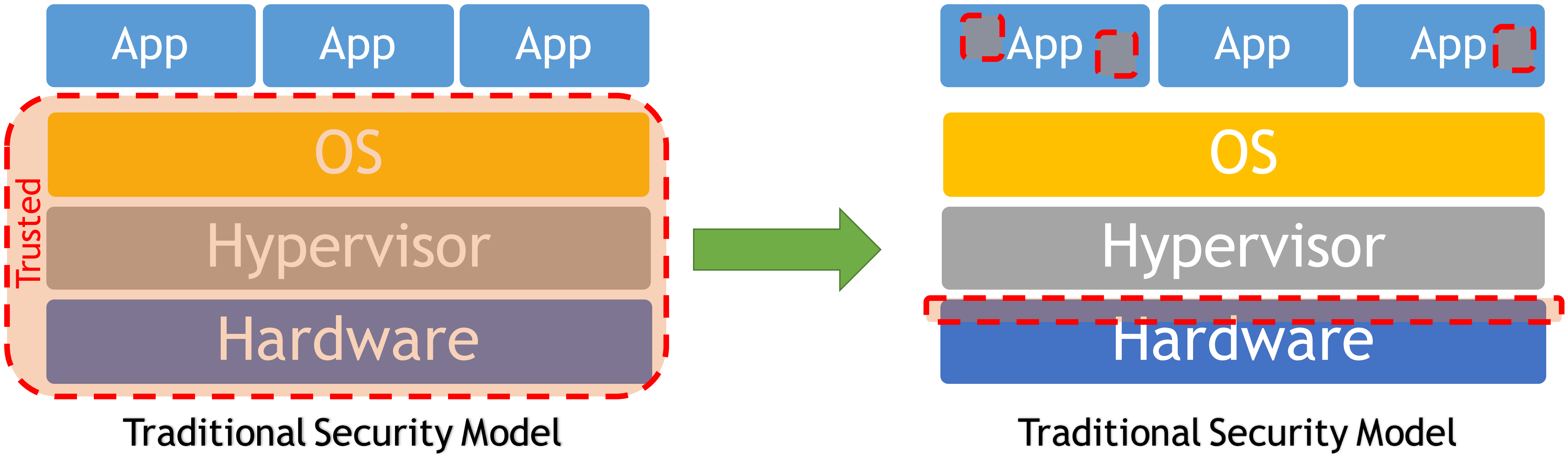
OS/Hypervisor Security Model



Traditional Security Model

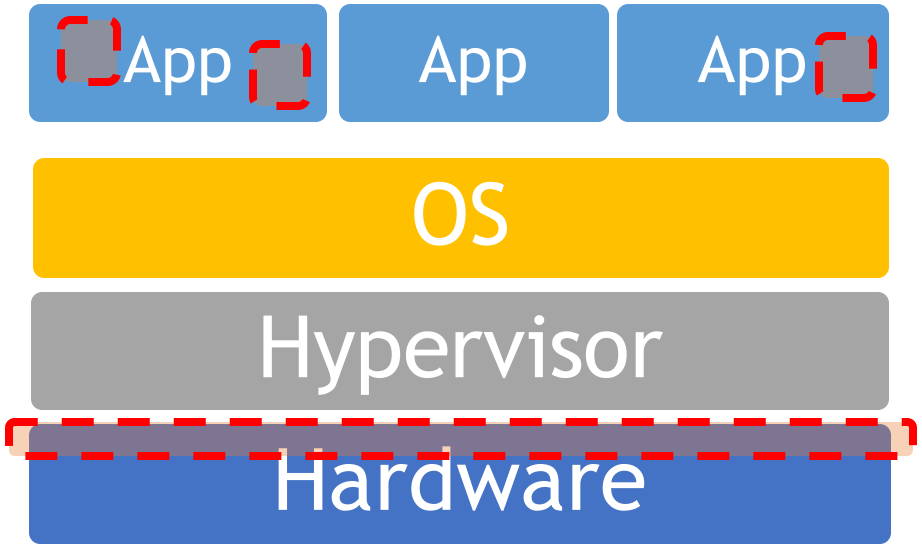
Trusted Execution Environment (TEE) – Intel SGX

- Intel Software Guard eXtensions (SGX)



Trusted Execution Environment (TEE) – Intel SGX

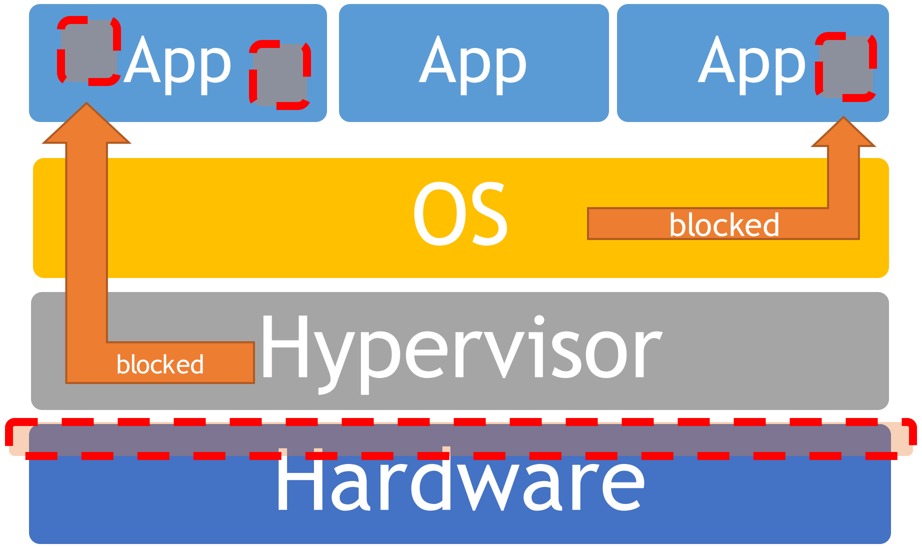
- Intel Software Guard eXtensions (SGX)
- **Enclave:** Hardware protected user-level software module
  - Mapped by the Operating System
  - Loaded by the user program
  - Authenticated and Encrypted by CPU



Traditional Security Model

Trusted Execution Environment (TEE) – Intel SGX

- Intel Software Guard eXtensions (SGX)
- **Enclave:** Hardware protected user-level software module
  - Mapped by the Operating System
  - Loaded by the user program
  - Authenticated and Encrypted by CPU
- Protects against system level adversary



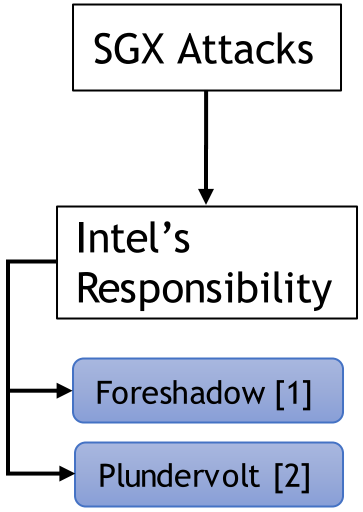
Traditional Security Model

New Attacker Model:

Attacker gets full control over OS

Intel SGX Attack Taxonomy

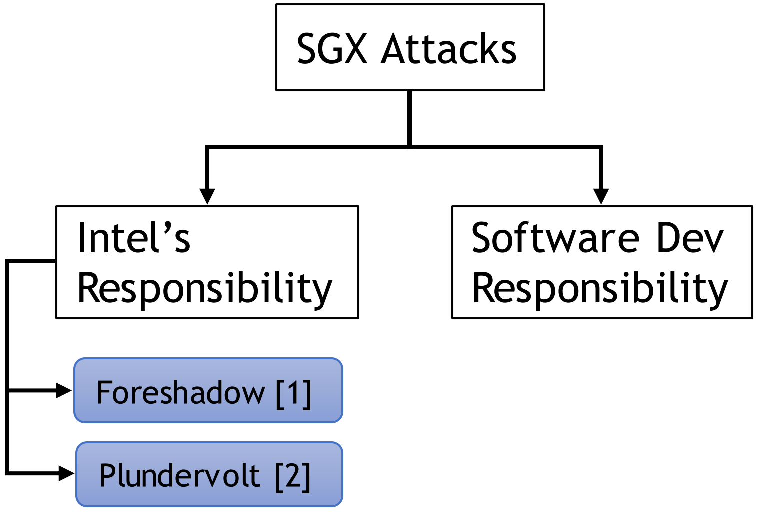
- Intel’s Responsibility
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning



[1] Van Bulck et al. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution." USENIX Security 2018.
[2] Murdock et al. "Plundervolt: Software-based fault injection attacks against Intel SGX." IEEE S&P 2020.

Intel SGX Attack Taxonomy

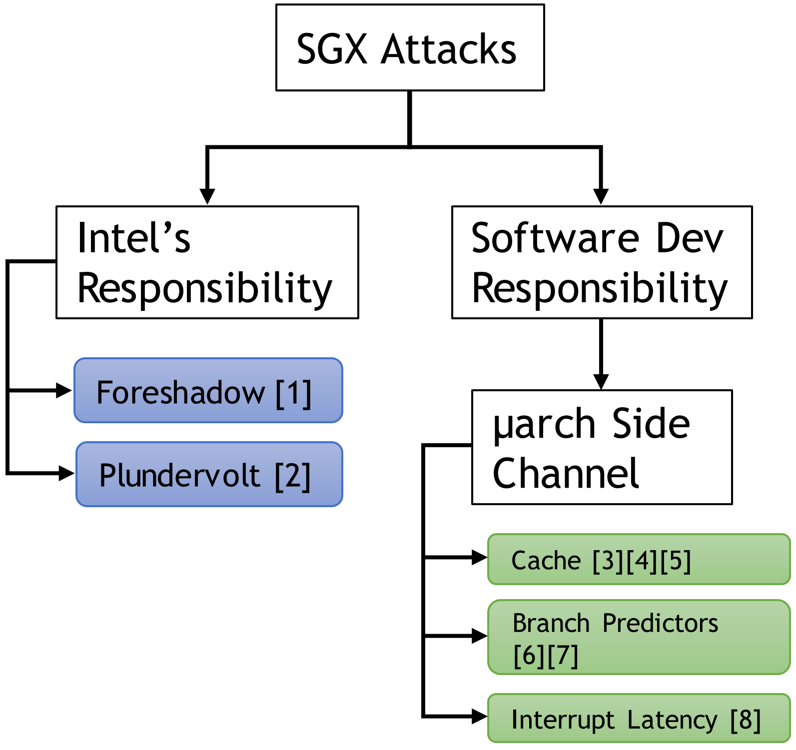
- Intel’s Responsibility
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning



[1] Van Bulck et al. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution." USENIX Security 2018.
[2] Murdock et al. "Plundervolt: Software-based fault injection attacks against Intel SGX." IEEE S&P 2020.

Intel SGX Attack Taxonomy

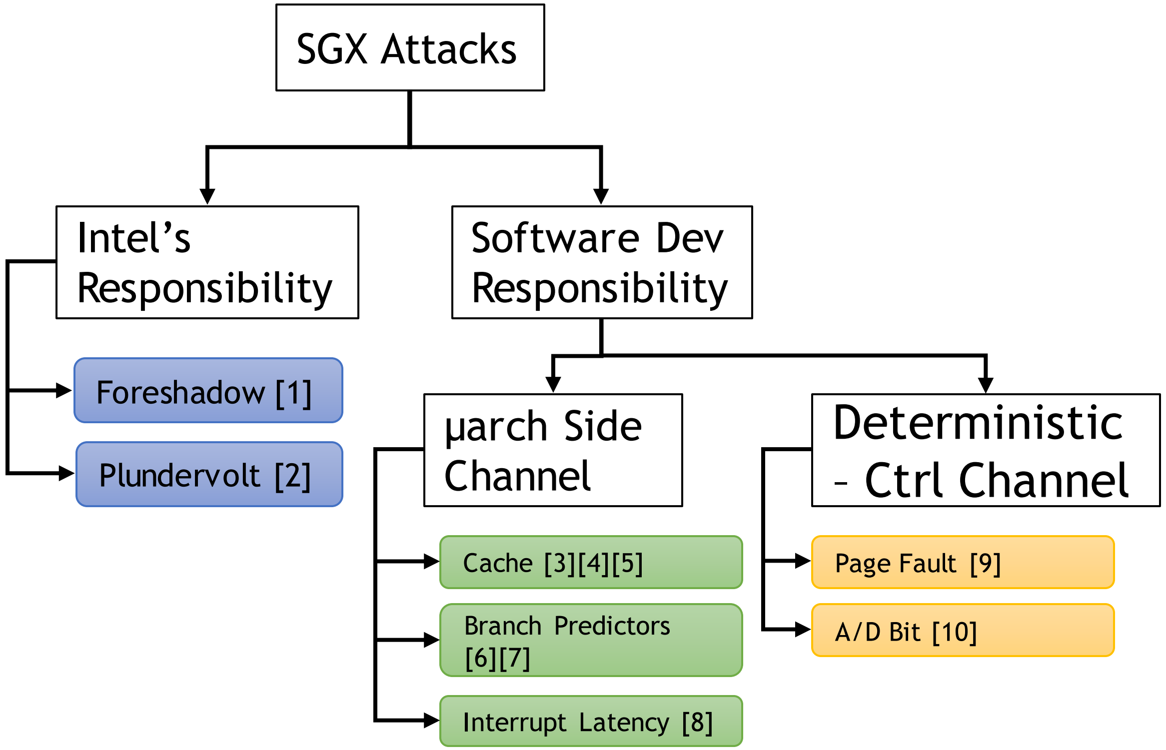
- Intel's Responsibility
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning
- µarch Side Channel
  - Constant-time Coding
  - Flushing and Isolating buffers
  - Probabilistic



[1] Van Bulck et al. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution." USENIX Security 2018.
[2] Murdock et al. "Plundervolt: Software-based fault injection attacks against Intel SGX." IEEE S&P 2020.
[3] Moghimi et al. "Cachezoom: How SGX amplifies the power of cache attacks." CHES 2017.
[4] Brasser et al. "Software grand exposure:{SGX} cache attacks are practical." USENIX WOOT 2017.
[5] Schwarz et al. "Malware guard extension: Using SGX to conceal cache attacks." DIMVA 2017.
[6] Evtyushkin, Dmitry, et al. "Branchscope: A new side-channel attack on directional branch predictor." ACM SIGPLAN 2018.
[7] Lee, Sangho, et al. "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing." USENIX Security 2017.
[8] Van Bulck et al. "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic." ACM CCS 2018.

Intel SGX Attack Taxonomy

- Intel's Responsibility
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning
- µarch Side Channel
  - Constant-time Coding
  - Flushing and Isolating buffers
  - Probabilistic
- Deterministic Attacks
  - Page Fault, A/D Bit, etc. (4kB Granularity)

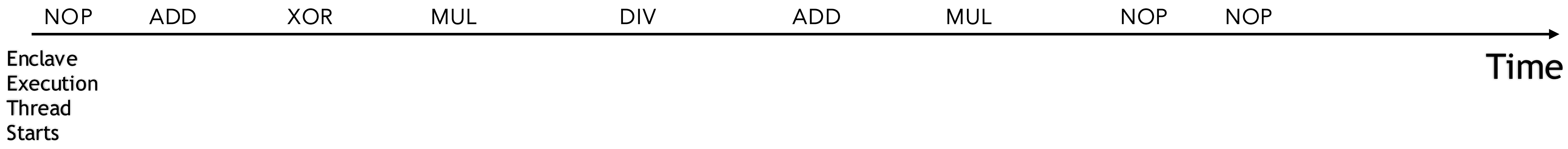


[1] Van Bulck et al. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution." USENIX Security 2018.
[2] Murdock et al. "Plundervolt: Software-based fault injection attacks against Intel SGX." IEEE S&P 2020.
[3] Moghimi et al. "Cachezoom: How SGX amplifies the power of cache attacks." CHES 2017.
[4] Brasser et al. "Software grand exposure:{SGX} cache attacks are practical." USENIX WOOT 2017.
[5] Schwarz et al. "Malware guard extension: Using SGX to conceal cache attacks." DIMVA 2017.
[6] Evtyushkin, Dmitry, et al. "Branchscope: A new side-channel attack on directional branch predictor." ACM SIGPLAN 2018.
[7] Lee, Sangho, et al. "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing." USENIX Security 2017.
[8] Van Bulck et al. "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic." ACM CCS 2018.
[9] Xu et al. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems." IEEE S&P 2015.
[10] Wang, Wenhao, et al. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX." ACM CCS 2017.

CopyCat Attack

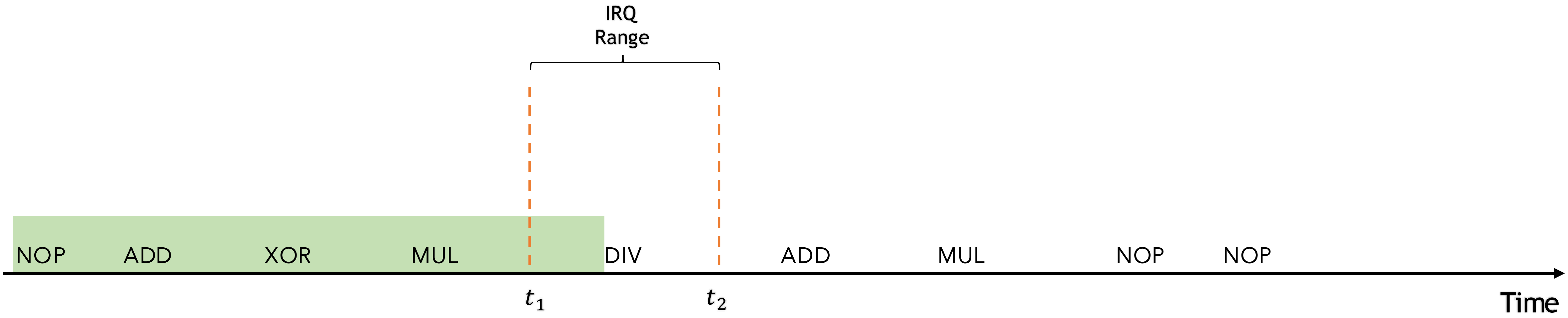
CopyCat Attack

- Malicious OS controls the interrupt handler



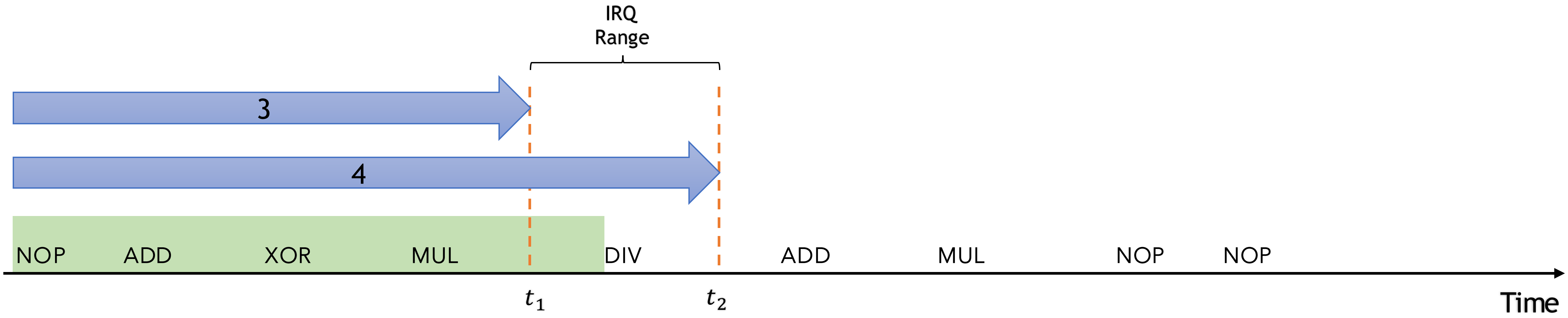
CopyCat Attack

- Malicious OS controls the interrupt handler



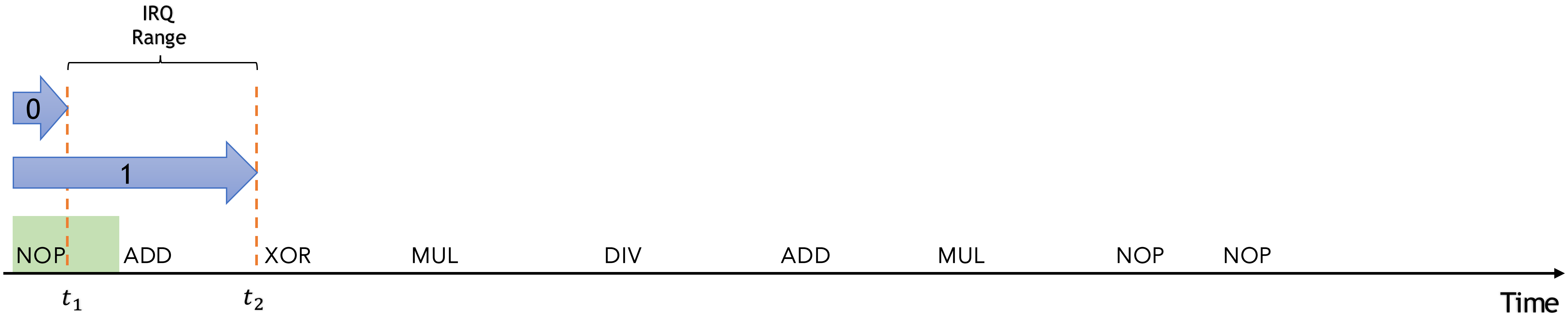
CopyCat Attack

- Malicious OS controls the interrupt handler



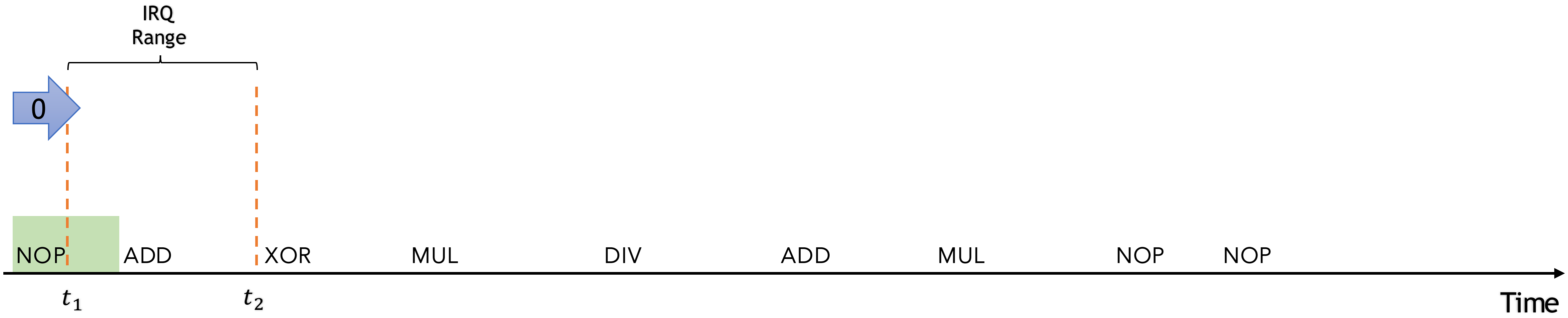
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



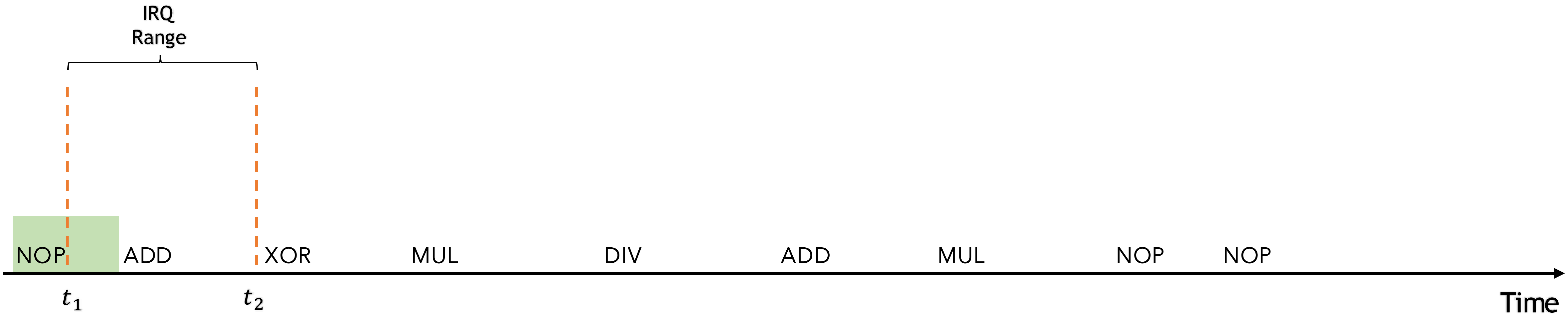
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



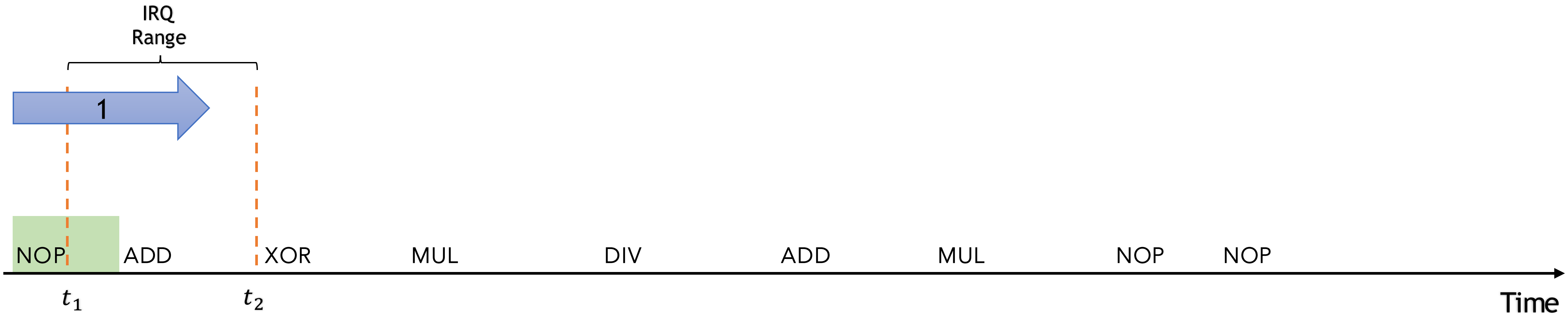
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



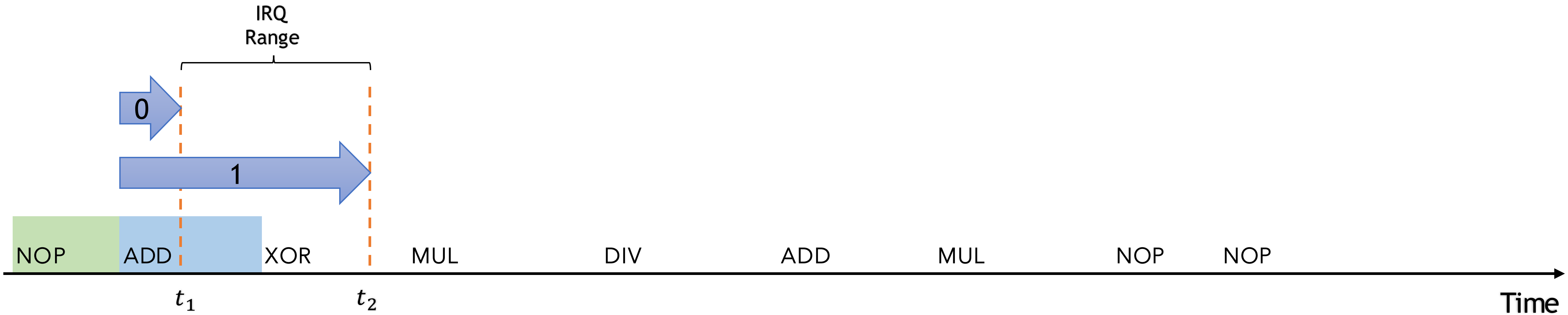
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



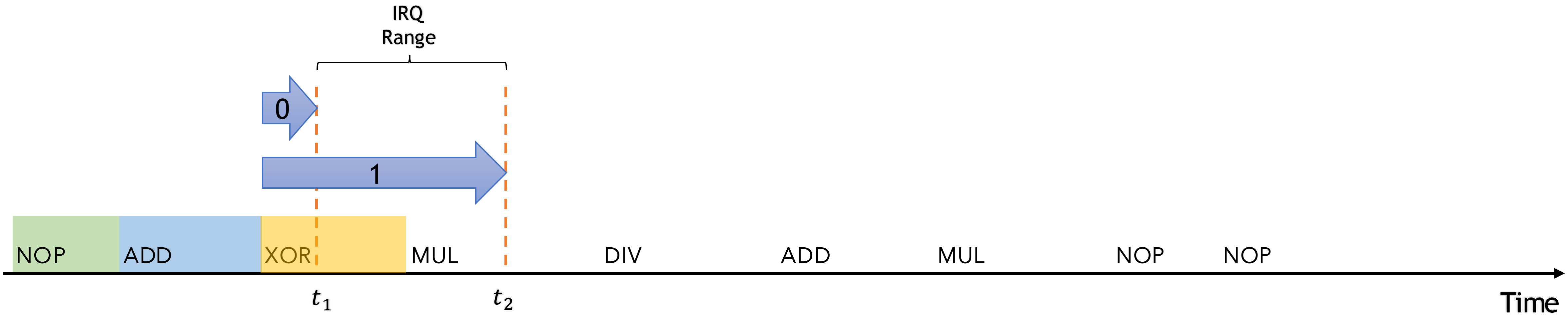
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



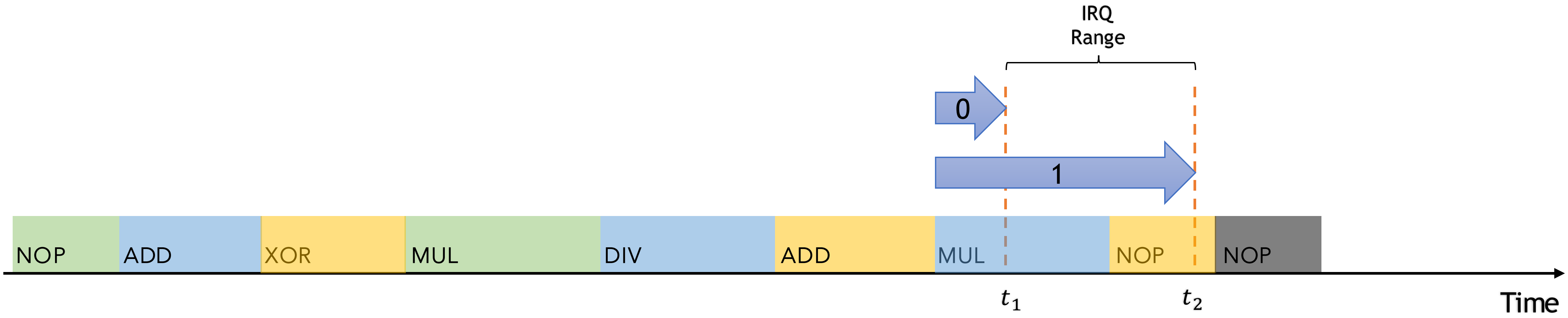
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



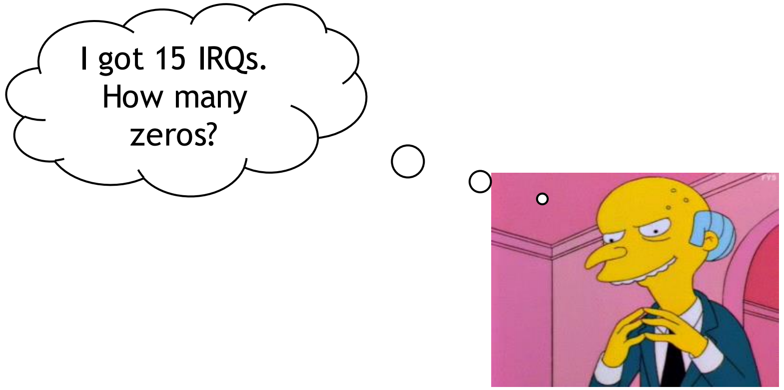
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



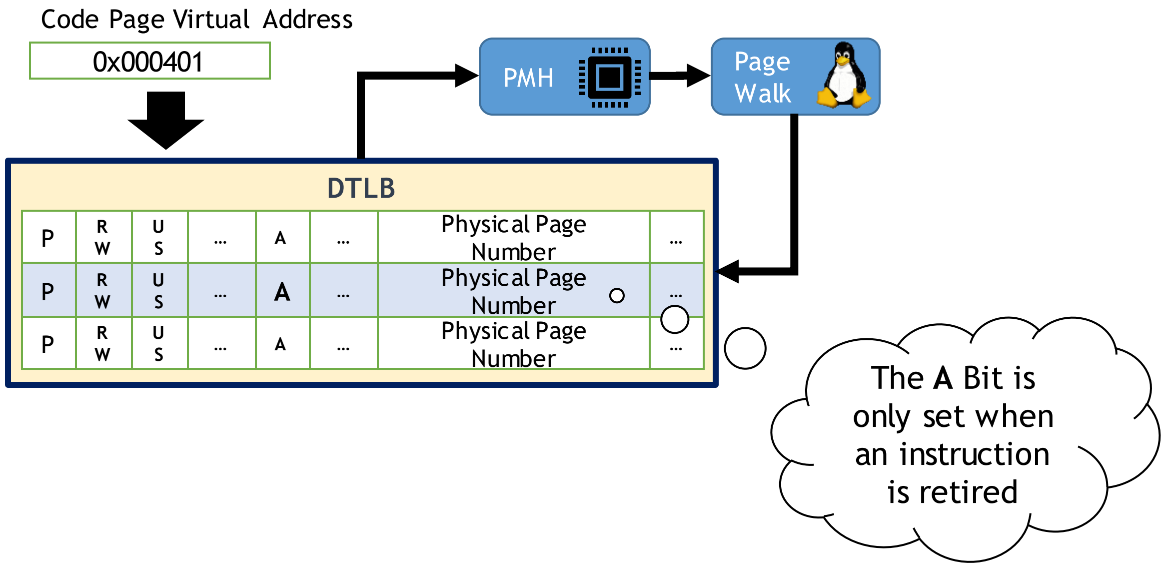
CopyCat Attack

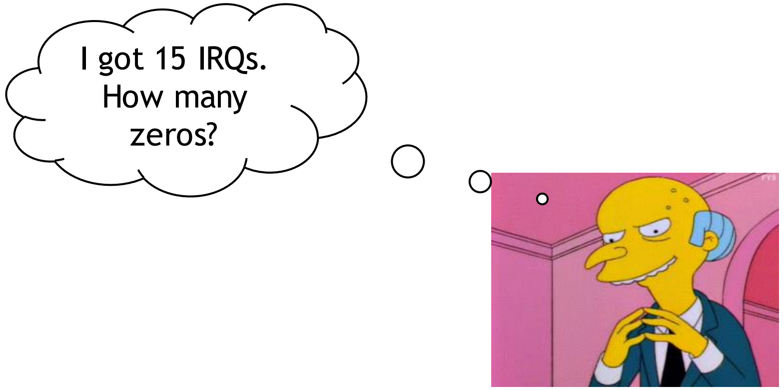
- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions



CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the **A** bit before, Check the **A** bit after



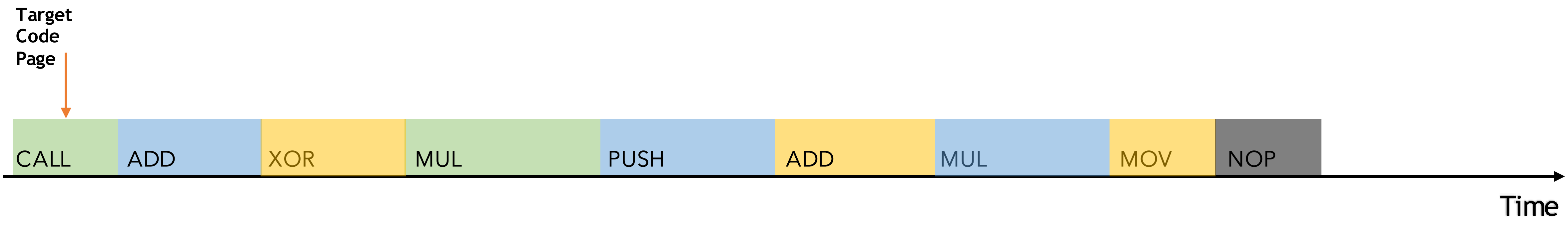


CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the **A** bit before, Check the **A** bit after
- Deterministic Instruction Counting

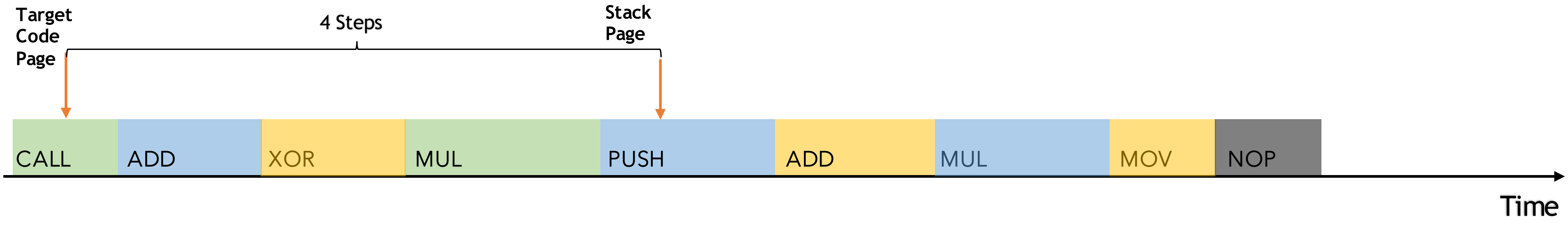
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the **A** bit before, Check the **A** bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle



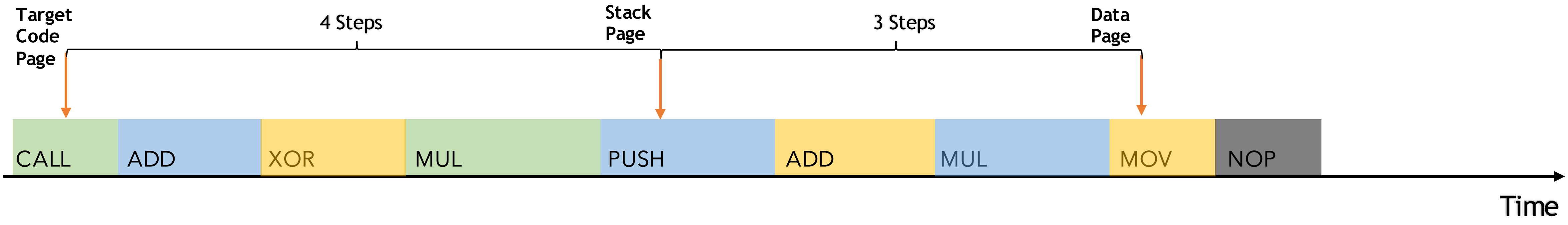
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the **A** bit before, Check the **A** bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle



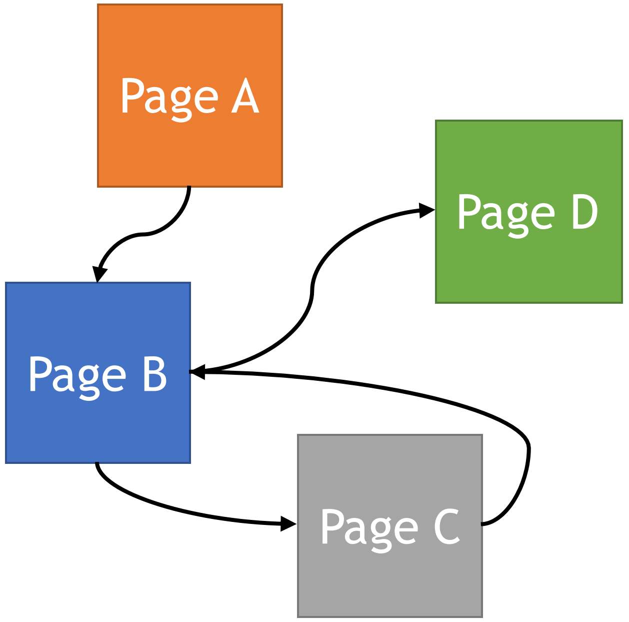
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the **A** bit before, Check the **A** bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle



CopyCat Attack

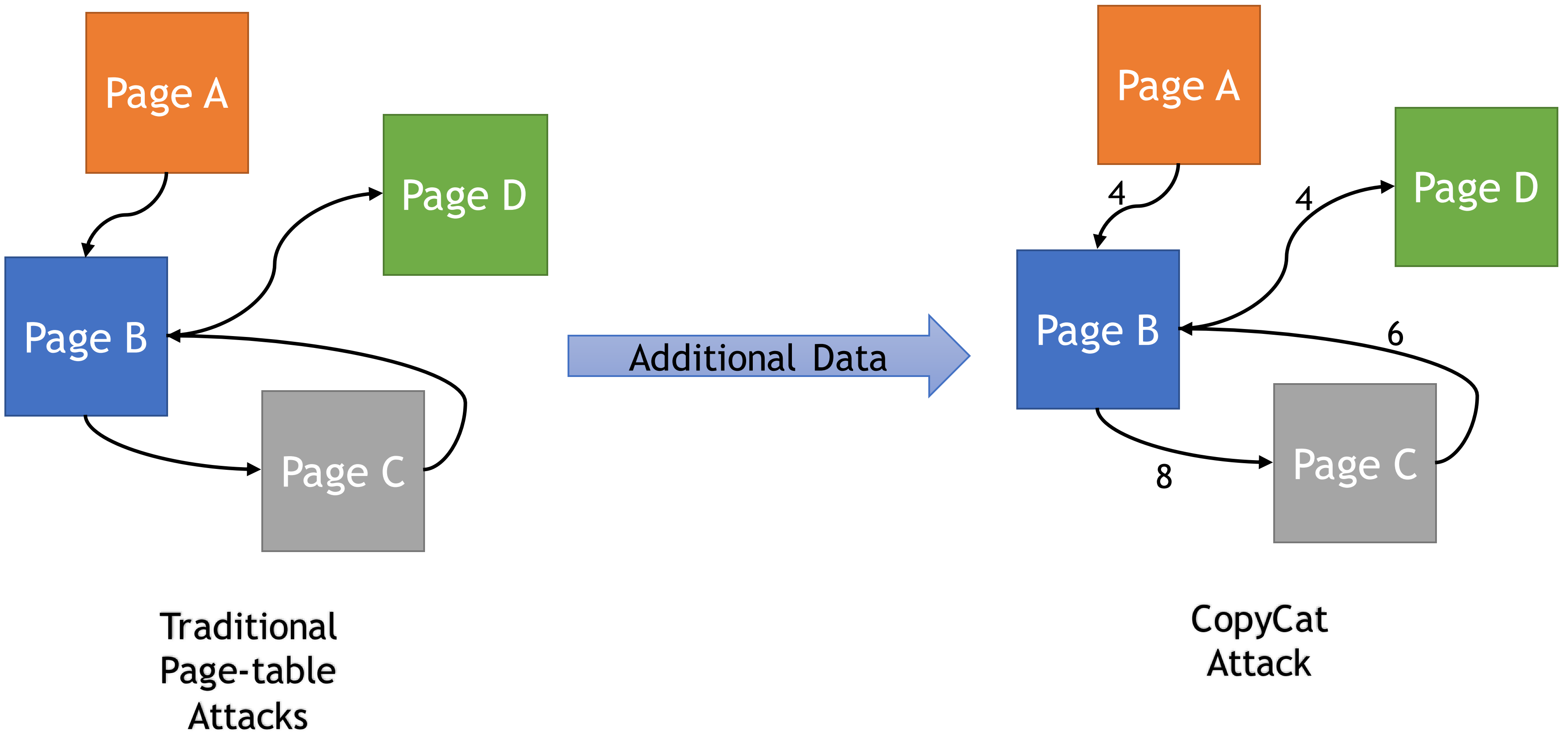
- Previous Controlled Channel attacks leak Page Access Patterns



Traditional Page-table Attacks

CopyCat Attack

- Previous Controlled Channel attacks leak Page Access Patterns
- CopyCat additionally leaks number of instructions per page



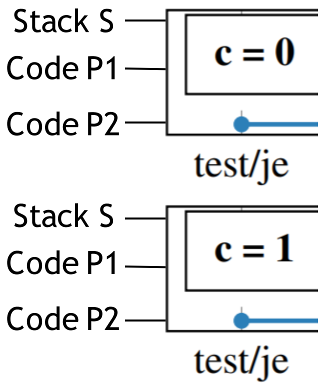
CopyCat – Leaking Branches

```c
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

Compile

```
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```



CopyCat – Leaking Branches

```c
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

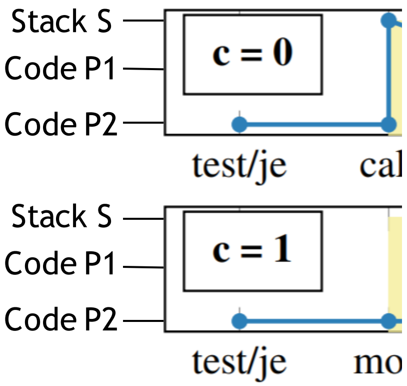
CopyCat – Leaking Branches

```c
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

Compile

```
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```



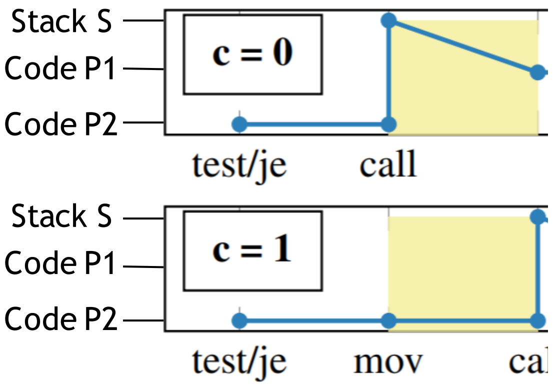
CopyCat – Leaking Branches

```c
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

Compile

```
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```



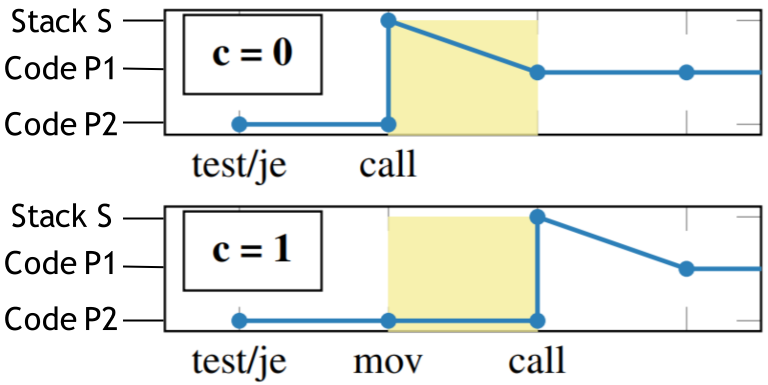
CopyCat – Leaking Branches

```
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

Compile

```
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```



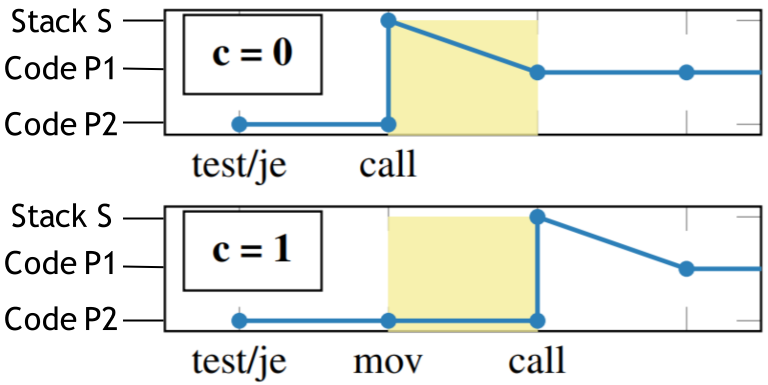
CopyCat – Leaking Branches

```c
if(c == 0) {
  r = add(r, d);
}
else {
  r = add(r, s);
}
```

C Code

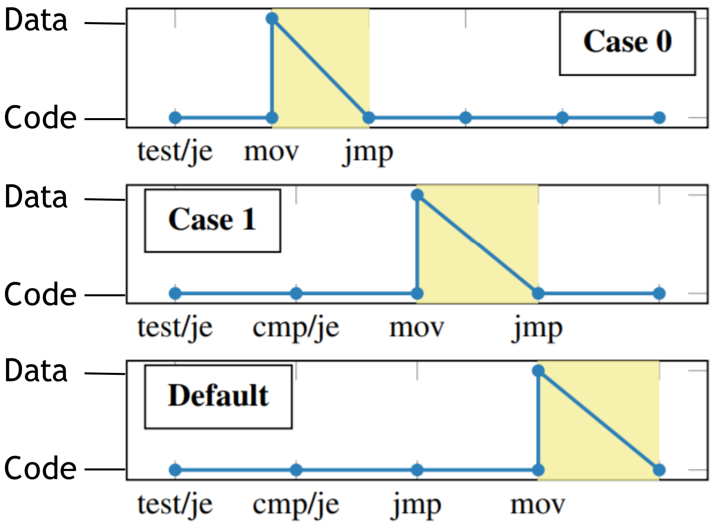
Compile

```
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```




```c
switch (c){
case 0:
  r = 0xbeef;
  break;
case 1:
  r = 0xcafe;
  break;
default:
  r = 0;
}
```

C Code



Crypto means Crpyptoattacks

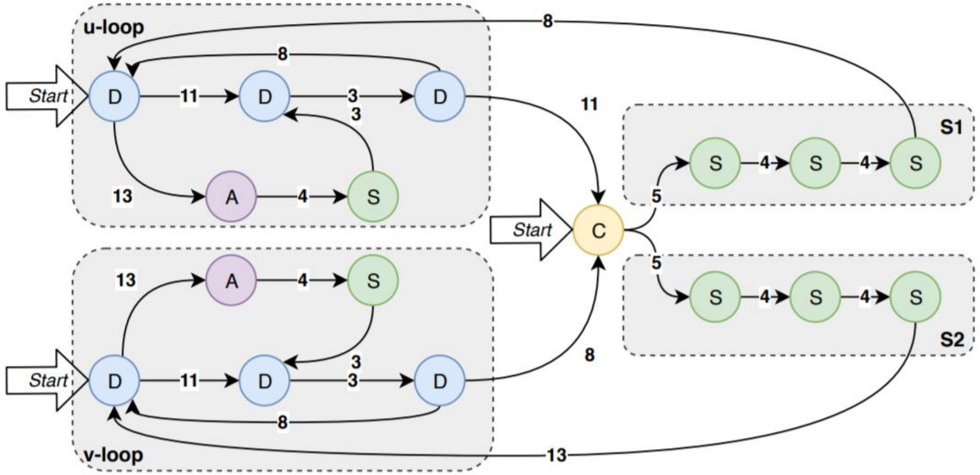
Binary Extended Euclidean Algorithm (BEEA)

- Previous attacks only leak some of the branches w/ some noise

procedure $\textsc{ModInv}(u, \text{modulus } v)$
    $b_i \leftarrow 0\ d_i \leftarrow 1, u_i \leftarrow u, v_i = v,$
    while $isEven(u_i)$ **do**
        $u_i \leftarrow u_i/2$
        if $isOdd(b_i)$ **then**
            $b_i \leftarrow b_i - u$
        $b_i \leftarrow b_i/2$
    while $isEven(v_i)$ **do**
        $v_i \leftarrow v_i/2$
        if $isOdd(d_i)$ **then**
            $d_i \leftarrow d_i - u$
        $d_i \leftarrow d_i/2$
    if $u_i > v_i$ **then**
        $u_i \leftarrow u_i - v_i,\ b_i \leftarrow b_i - d_i$
    else
        $v_i \leftarrow v_i - u_i,\ d_i \leftarrow d_i - b_i$
    return d_i

Binary Extended Euclidean Algorithm

- Previous attacks only leak some of the branches w/ some noise
- CopyCat synchronously leaks all the branches wo/ any noise



procedure MODINV(u, modulus v)

  $b_i \leftarrow 0\ d_i \leftarrow 1, u_i \leftarrow u, v_i = v,$

  while $isEven(u_i)$ **do**

    $u_i \leftarrow u_i/2$

    if $isOdd(b_i)$ **then**

      $b_i \leftarrow b_i - u$

    $b_i \leftarrow b_i/2$

  while $isEven(v_i)$ **do**

    $v_i \leftarrow v_i/2$

    if $isOdd(d_i)$ **then**

      $d_i \leftarrow d_i - u$

    $d_i \leftarrow d_i/2$

  if $u_i > v_i$ **then**

    $u_i \leftarrow u_i - v_i,\ b_i \leftarrow b_i - d_i$

  else

    $v_i \leftarrow v_i - u_i,\ d_i \leftarrow d_i - b_i$

  return d_i

CopyCat on WolfSSL

- Translate instruction Counts to Basic Block Transitions

```
11,3,8,5,4,4,13,11,3,8,5,4,4,8,11,3,8,11,3,8,13,4,3,3,8,11,3,11,5,4,4
```

CopyCat on WolfSSL

- Translate instruction Counts to Basic Block Transitions

11,3,8,5,4,4,13,11,3,8,5,4,4,8,11,3,8,11,3,8,13,4,3,3,8,11,3,11,5,4,4

$\boxed{\text{DDD}}$ 8 $\boxed{\text{CSSS}}$ 13 $\boxed{\text{DDD}}$ 8 $\boxed{\text{CSSS}}$ 8 $\boxed{\text{DDD}}$ 8 $\boxed{\text{DDD}}$ 8 $\boxed{\text{DASDD}}$ 8 $\boxed{\text{DDD}}$ 11 $\boxed{\text{CSSS}}$

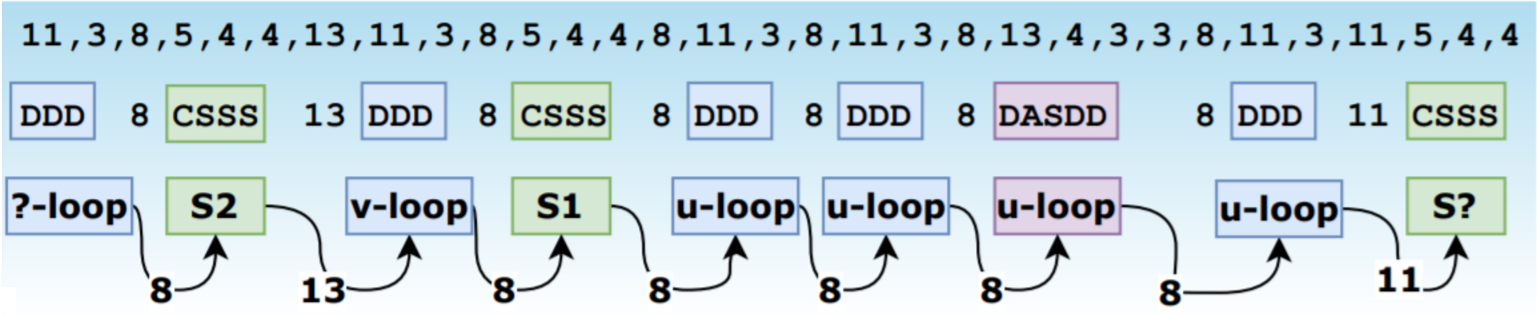
Rule 1: $? \xrightarrow{11} ? \xrightarrow{3} ? = D \rightarrow D \rightarrow D.$

Rule 2: $? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \rightarrow A \rightarrow S \rightarrow D \rightarrow D.$

Rule 3: $? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \rightarrow S \rightarrow S \rightarrow S.$

CopyCat on WolfSSL

- Translate instruction Counts to Basic Block Transitions



Rule 1: $? \xrightarrow{11} ? \xrightarrow{3} ? = D \rightarrow D \rightarrow D.$

Rule 2: $? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \rightarrow A \rightarrow S \rightarrow D \rightarrow D.$

Rule 3: $? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \rightarrow S \rightarrow S \rightarrow S.$

Rule 4: $S? \xrightarrow{13} ? = S2 \rightarrow v\text{-}loop.$

Rule 5: $S? \xrightarrow{8} ? = S1 \rightarrow u\text{-}loop.$

CopyCat on WolfSSL - Cryptanalysis

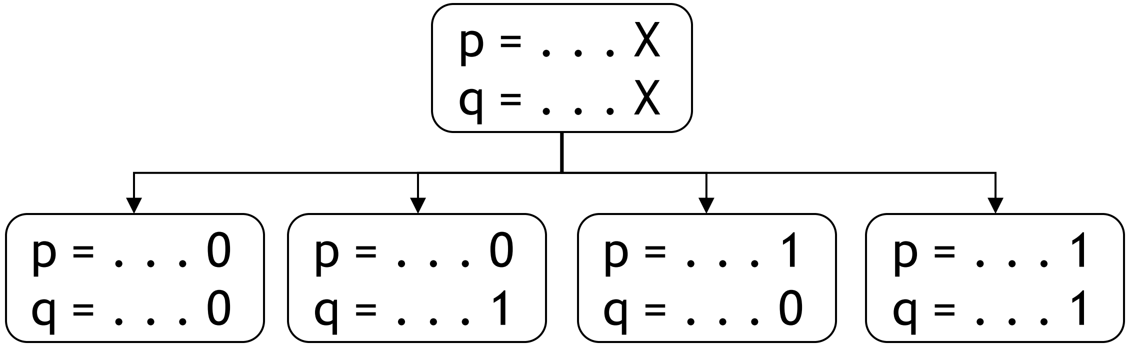
- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x

CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1}\ mod\ n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1 . x) mod\ n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1}\ mod\ p$
  - We know that $\mathrm{p}.\mathrm{q} = \mathrm{N}$

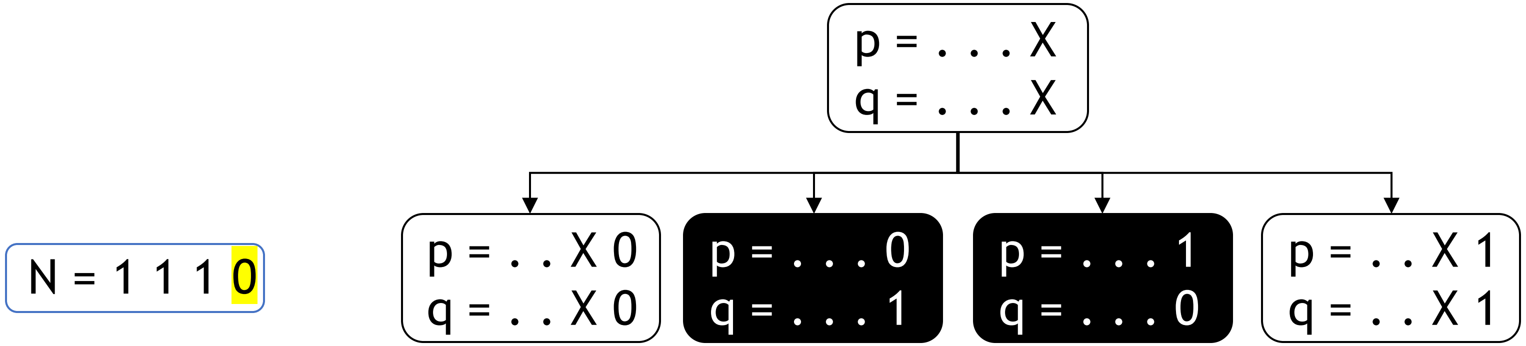
CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p}.\mathrm{q} = \mathrm{N}$
  - Branch and prune Algorithm with the help of the recovered trace



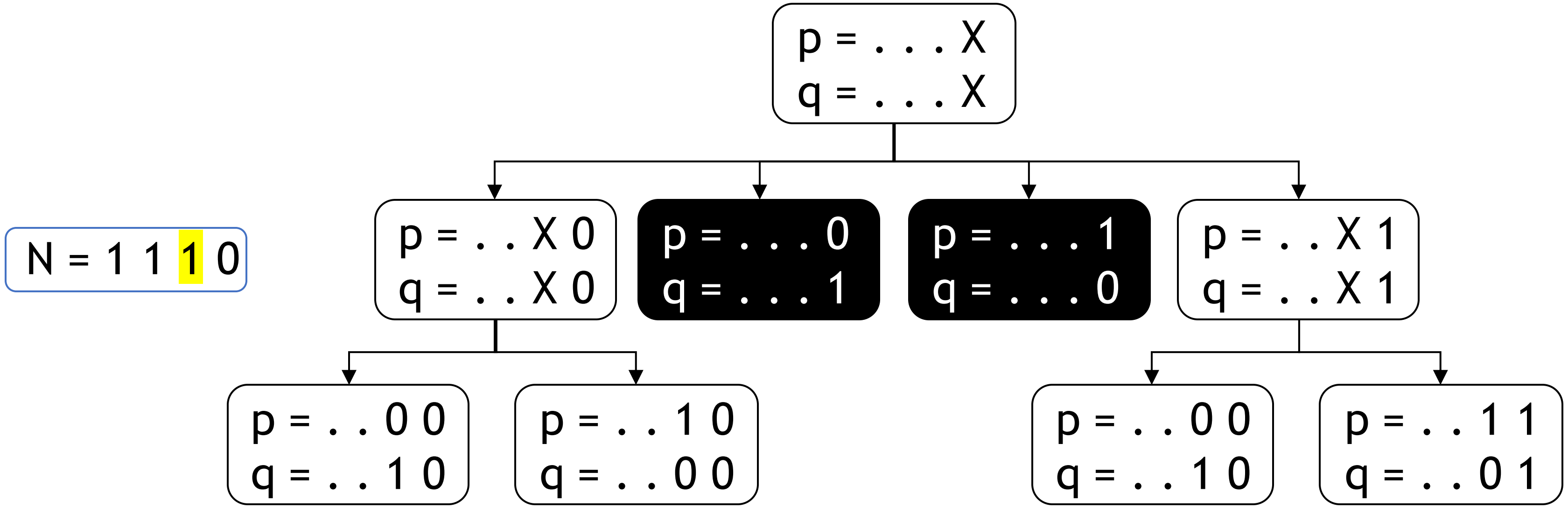
CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1 . x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p} . \mathrm{q} = \mathrm{N}$, and N is public
  - Branch and prune Algorithm with the help of the recovered trace



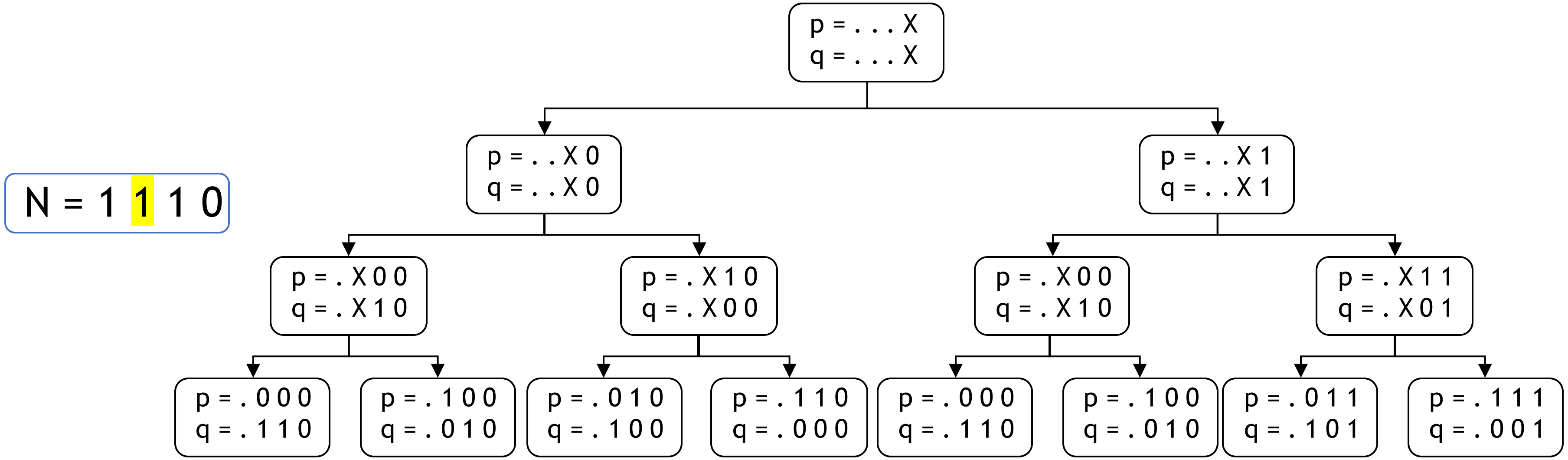
CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p.q} = \mathrm{N}$, and N is public
  - Branch and prune Algorithm with the help of the recovered trace



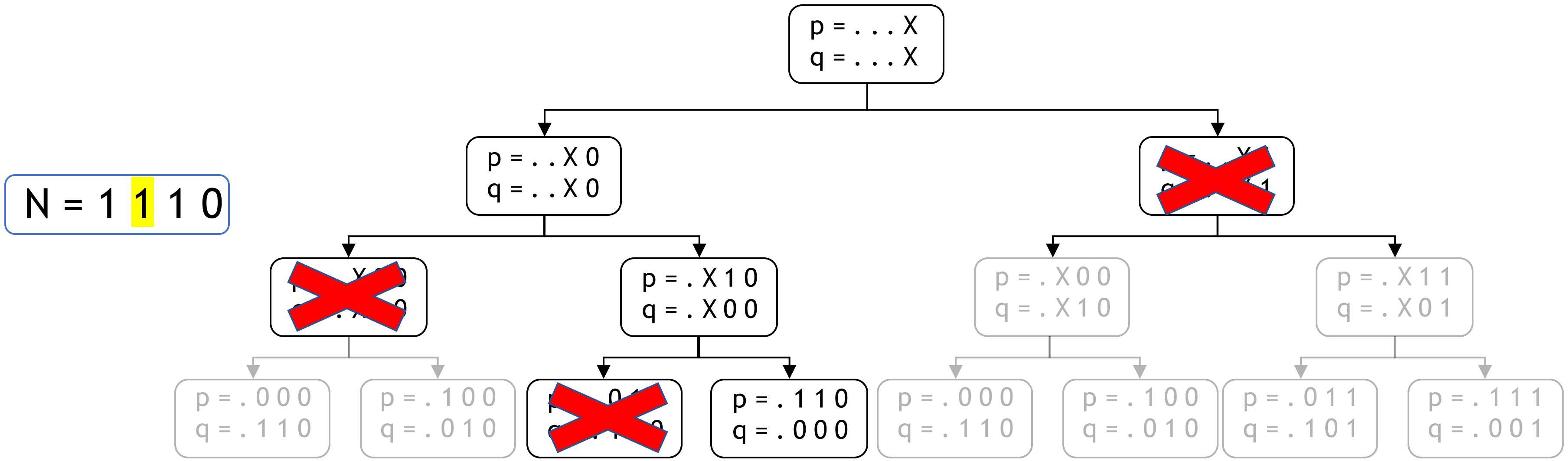
CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p.q} = \mathrm{N}$, and N is public
  - Branch and prune Algorithm with the help of the recovered trace



CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p.q} = \mathrm{N}$, and N is public
  - Branch and prune Algorithm with the help of the recovered trace



CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \, mod \, n$
  - Iterative over the entire recovered trace with n as input → k_{inv}
  - Plug k_{inv} in $s_1 = k_1^{-1}(h - r_1.x) mod \, n$ → get private key x
- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \, mod \, p$
  - We know that $\mathrm{p.q} = \mathrm{N}$, and N is public
  - Branch and prune Algorithm with the help of the recovered trace
- Single-trace Attack during RSA Key Generation: $d = e^{-1} \, mod \, \lambda(N)$
  - Similar attack but instead use $\lambda(N) = \frac{(p-1)(q-1)}{2^i}$
  - Only 81% of the keys have the above property
  - It works even on a hardcoded and big value for e, i.e. $e \neq 65537$

CopyCat on WolfSSL – Cryptanalysis Results

- Executed each attack 100 times.
- DSA $k^{-1}\ mod\ n$
  - Average 22,000 IRQs
  - 75 ms to iterate over an average of 6,320 steps
- RSA $q^{-1}\ mod\ p$
  - Average 106490 IRQs
  - 365 ms to iterate over an average of 39,400 steps
- RSA $e^{-1}\ mod\ \lambda(N)$
  - $e^{-1}\ mod\ \lambda(N)$
  - Average 230,050 IRQs
  - 800ms to iterate over an average of 81,090 steps
- Experimental traces always match the leakage model in all experiments
  → Successful single-trace key recovery

CopyCat – Bypassing ECDSA Timing Countermeasure

```
int wc_ecc_mulmod_ex(mp_int* k, ecc_point *G, ecc_point *R, mp_int* a, mp_int
        * modulus, int map, void* heap) { ...
for (;;) {
if (−−bitcnt == 0) { /* grab next digit as required */
 if (digidx == −1) {
    break;
 }
 buf = get_digit(k, digidx);
 bitcnt = (int)DIGIT_BIT;
 −−digidx;
}
i = (buf >> (DIGIT_BIT − 1)) & 1; /* grab the next msb from the multiplicand */
buf <<= 1;
if (mode == 0) {
 mode = i; /* timing resistant − dummy operations */
 err = ecc_projective_add_point(M[1], M[2], M[2], a, modulus, mp);...
 err = ecc_projective_dbl_point(M[2], M[3], a, modulus, mp);...
}...
err = ecc_projective_add_point(M[0], M[1], M[i^1], a, modulus, mp);...
err = ecc_projective_dbl_point(M[2], M[2], a, modulus, mp);...
} /* end for */...}
```

Table 2: Minimum number of signature samples for each bias class to reach 100% recovery success for the lattice-based key recovery on `wc_ecc_mulmod_ex` of ECDSA, with lattice reduction time L-TIME and trace collection time T-TIME.

LZBs	DIM	L-TIME	SIGNATURES	IRQs	T-TIME
4	75	30 sec	1,200	3.9M	13.3 sec
5	58	5 sec	1,856	6.0M	20.4 sec
6	46	3 sec	2,944	9.6M	33.7 sec
7	42	2 sec	5,376	17.5M	1 min

How about other Crypto libraries?

- Libgcrypt uses a variant of BEEA
  - Single trace attack on DSA, Elgamal, ECDSA, RSA Key generation
- OpenSSL uses BEEA for computing GCD
  - Single trace attack on RSA Key generation when computing $\gcd(q-1, p-1)$
- There is still lots of other cases of micro leakages due to usage of branches, e.g. Intel IPP Crypto lehmer's GCD with optimizations

	Operation (Subroutine)	Implementation	Secret Branch	Exploitable	Computation → Vulnerable Callers	Single-Trace Attack
WolfSSL	Scalar Multiply (wc_ecc_mulmod_ex)	Montgomery Ladder w/ Branches	✔	✔	$(k \times G) \rightarrow$ wc_ecc_sign_hash	✘
	Greatest Common Divisor (fp_gcd)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (fp_invmod)	BEEA	✔	✔	$(k^{-1} \bmod n) \rightarrow$ wc_DsaSign	✔
					$(q^{-1} \bmod p) \rightarrow$ wc_MakeRsaKey	✔
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ wc_MakeRsaKey	✔
Libgcrypt	Greatest Common Divisor (mpi_gcd)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (mpi_invm)	Modified BEEA [43, Vol II, §4.5.2]	✔	✔	$(k^{-1} \bmod n) \rightarrow$ {dsa,elgamal}.c::sign,_gcry_ecc_ecdsa_sign	✔
					$(q^{-1} \bmod p) \rightarrow$ generate_{std,fips,x931}	✔
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ generate_{std,fips,x931}	✔
OpenSSL	Greatest Common Divisor (BN_gcd)	BEEA	✔	✔	$gcd(q-1, p-1) \rightarrow$ RSA_X931_derive_ex	✔
	Modular Inverse (BN_mod_inverse_no_branch)	BEEA w/ Branches	✘	N/A	N/A	N/A
IPP Crypto	Greatest Common Divisor (ippsGcd_BN)	Modified Lehmer's GCD	✔	?	$gcd(q-1, e) \rightarrow$ cpIsCoPrime	N/A
				?	$gcd(p-1, q-1) \rightarrow$ isValidPriv1_rsa	N/A
	Modular Inverse (cpModInv_BNU)	Euclidean (Divisions)	✔	✘	N/A	N/A

Responsible Disclosure

- WolfSSL fixed the issues in 4.3.0 and 4.4.0
  - Blinding for $k^{-1}\ mod\ n$ and $e^{-1}\ mod\ \lambda(N)$
  - Alternate formulation for $q^{-1}\ mod\ p$: $q^{p-2}\ mod\ p$
  - Using a constant-time (branchless) modular inverse [11]
- Libgcrypt fixed the issues in 1.8.6
  - Using a constant-time (branchless) modular inverse [11]
- OpenSSL fixed the issue in 1.1.1e
  - Using a constant-time (branchless) GCD algorithm [11]

[11] Bernstein, Daniel J., and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion." *CHES 2019*.

Interrupt Driven Attacks and Single Stepping

- Amplifying Transient Execution Attacks
  - Foreshadow, ZombieLoad, LVI, CrossTalk
- Amplifying Microarchitectural Side Channels
  - CacheZoom, BranchScope, Branch Shadowing, Bluethunder, etc.
- Interrupt Latency as a Side Channel
  - Nemesis, Frontal Attack

- **CopyCat: Deterministic Instruction Counting as a Side Channel**

SGX-Step

Title	Publication details	Source code	SGX-Step features used
CrossTalk: Speculative Data Leaks Across Cores Are Real	S&P21	-	Single-stepping, page fault
Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend	arXiv20	-	Single-stepping interrupt latency, PTE A/D
From A to Z: Projective coordinates leakage in the wild	CHES20	-	Page fault
LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection	S&P20	link	Single-stepping, page-table manipulation
CopyCat: Controlled Instruction-Level Attacks on Enclaves	USEC20	-	Single-stepping, page fault, PTE A/D
When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA	CHES20	-	Single-stepping, page fault, PTE A/D
Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations	USEC20	-	Page fault
Plundervolt: Software-based Fault Injection Attacks against Intel SGX	S&P20	link	Privileged interrupt/call gates, MSR
Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX	CHES20	-	Single-stepping
Fallout: Leaking Data on Meltdown-resistant CPUs	CCS19	-	PTE A/D
A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes	CCS19	link	Single-stepping, page fault, PTE A/D
ZombieLoad: Cross-Privilege-Boundary Data Sampling	CCS19	link	Single-stepping, zero-stepping, page-table manipulation
SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks	USEC19	-	Single-stepping interrupt latency
Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic	CCS18	link	Single-stepping interrupt latency, page fault, PTE A/D
Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution	USEC18	link	Single-stepping, zero-stepping, page-table manipulation
Single Trace Attack Against RSA Key Generation in Intel SGX SSL	AsiaCCS18	-	Page fault
Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution	ESSoS18	link	Single-stepping, IA32 segmentation, page fault
SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control	SysTEX17	link	Single-stepping, page fault, PTE A/D

Comparison to other Attacks

	Attack	Code/Data	Granularity	Noise
μ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✘ Low (32 KiB)	~ Low
	COPYCAT	**Code**	✔ **High (instruction)**	✔ **Deterministic**

Comparison to other Attacks

- Some do not work when hyper-threadnig is disabled (Strong TCB of Intel SGX)

	Attack	Code/Data	Granularity	Noise
μ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✘ Low (32 KiB)	~ Low
	COPYCAT	**Code**	**✔ High (instruction)**	**✔ Deterministic**

Comparison to other Attacks

- Some do not work when hyper-threadnig is disabled (Strong TCB of Intel SGX)
- Some can be mitigated by flushing/isolating microarchitectural buffers.

	Attack	Code/Data	Granularity	Noise
μ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✘ Low (32 KiB)	~ Low
	COPYCAT	**Code**	**✔ High (instruction)**	**✔ Deterministic**

Comparison to other Attacks

- Some do not work when hyper-threadnig is disabled (Strong TCB of Intel SGX)
- Some can be mitigated by flushing/isolating microarchitectural buffers.
- Some only apply to legacy enclave (32-bit)

	Attack	Code/Data	Granularity	Noise
µ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✘ Low (32 KiB)	~ Low
	COPYCAT	**Code**	**✔ High (instruction)**	**✔ Deterministic**

Comparison to other Attacks

- Some do not work when hyper-threadnig is disabled (Strong TCB of Intel SGX)
- Some can be mitigated by flushing/isolating microarchitectural buffers.
- Some only apply to legacy enclave (32-bit)
- Some are limited to be applied synchronously.

	Attack	Code/Data	Granularity	Noise
µ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	PRIME+PROBE cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table FLUSH+RELOAD [72]	Code + data	✘ Low (32 KiB)	~ Low
	COPYCAT	**Code**	**✔ High (instruction)**	**✔ Deterministic**

CopyCat and Macro-fusion

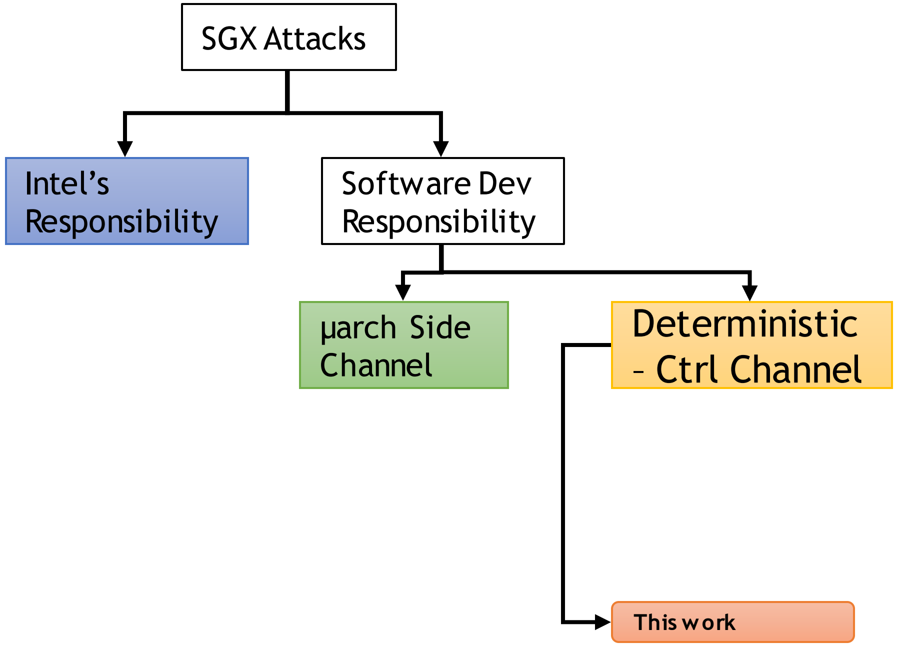
- Fused instructions are counted as one.
- Confirm/RE of the behavior of macro-fusion on Intel CPUs
- Macro-fusion is dependent on the program layout → deterministic
  - The offset of a *cmp+branch* within a cache line
  - True when hyperthreading is disabled (Intel SGX TCB)

Macro-Fusibility							
Instruction	**TEST**	**CMP**	**AND**	**ADD**	**SUB**	**INC**	**DEC**
JO/JNO	✓	✗	✓	✗	✗	✗	✗
JC/JB/JAE/JNB	✓	✓	✓	✓	✓	✗	✗
JE/JZ/JNE/JNZ	✓	✓	✓	✓	✓	✓	✓
JNA/JBE/JA/JNBE	✓	✓	✓	✓	✓	✗	✗
JS/JNS/JP/JPE/JNP/JPO	✓	✗	✓	✗	✗	✗	✗
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	✓	✓	✓	✓	✓	✓	✓

https://en.wikichip.org/wiki/macro-operation_fusion

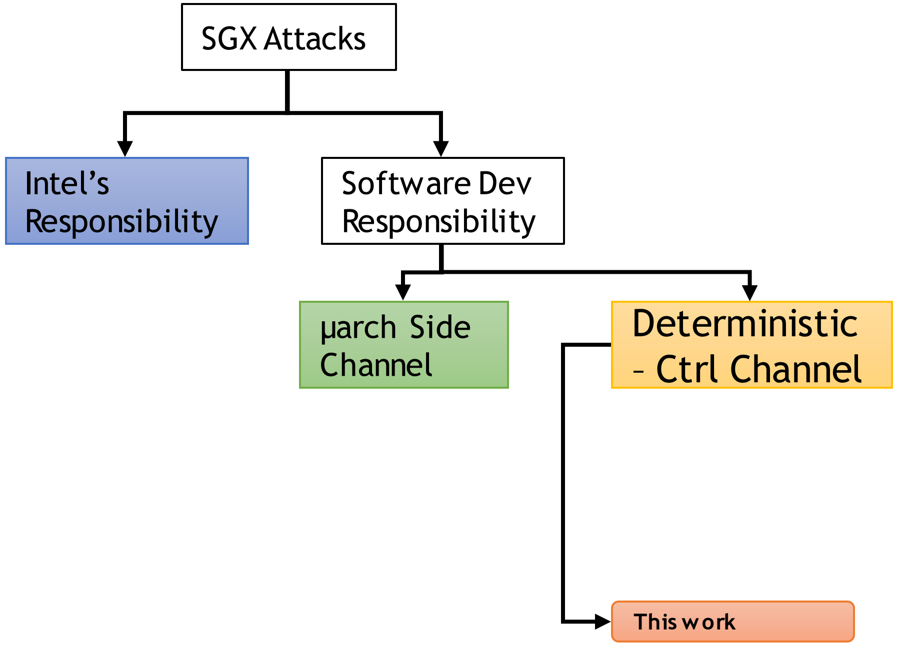
Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches



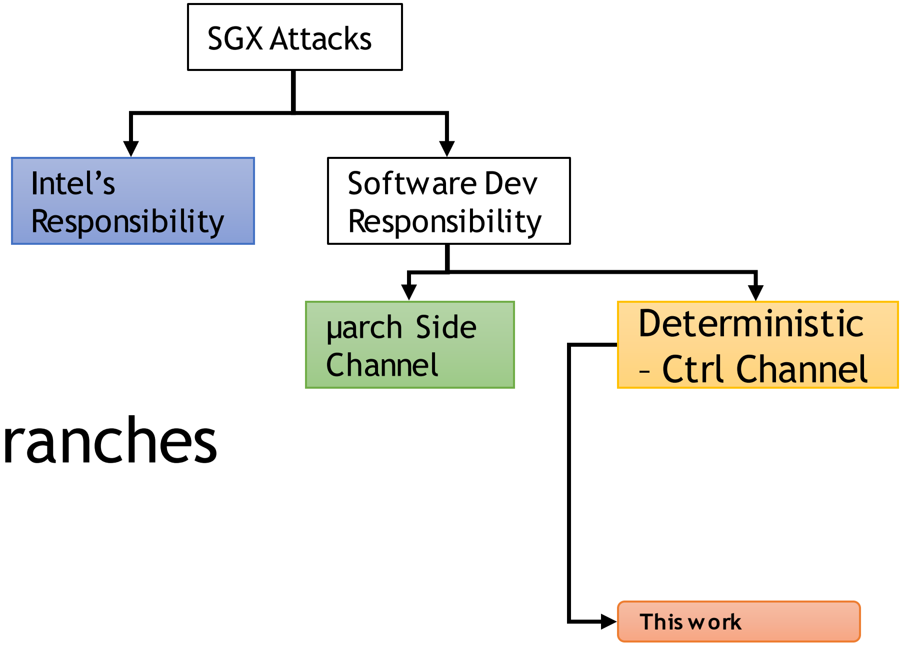
Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches
- Fully Deterministic and reliable
  - Millions of instructions tested
  - Attacks match the exact leakage model



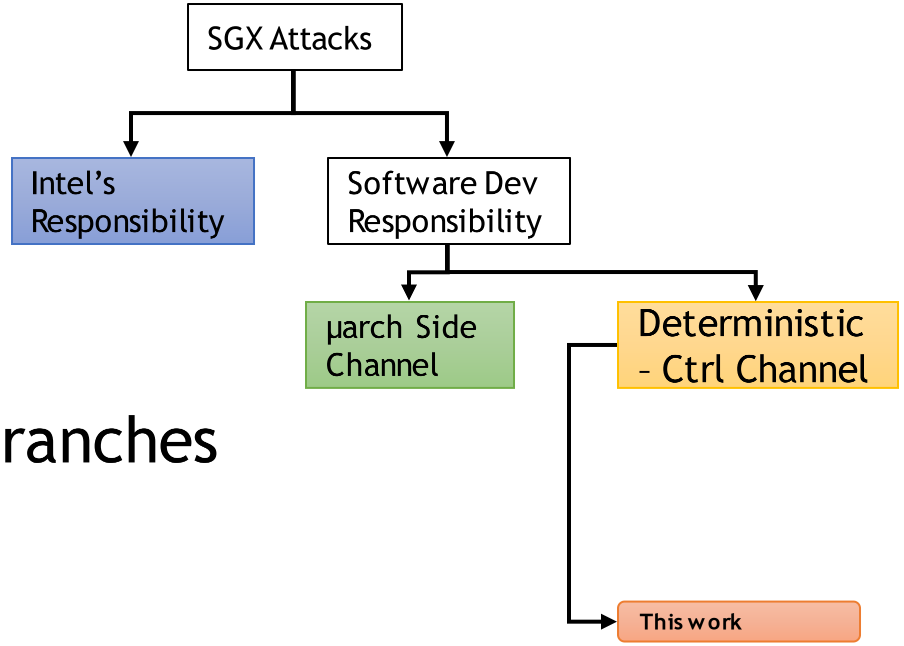
Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches
- Fully Deterministic and reliable
  - Millions of instructions tested
  - Attacks match the exact leakage model of branches
- Easy to scale and replicate
  - No reverse engineering of branches and microarchitectural components
  - Tracking all the branches synchronously



Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches
- Fully Deterministic and reliable
  - Millions of instructions tested
  - Attacks match the exact leakage model of branches
- Easy to scale and replicate
  - No reverse engineering of branches and microarchitectural components
  - Tracking all the branches synchronously
- Branchless programming is hard!

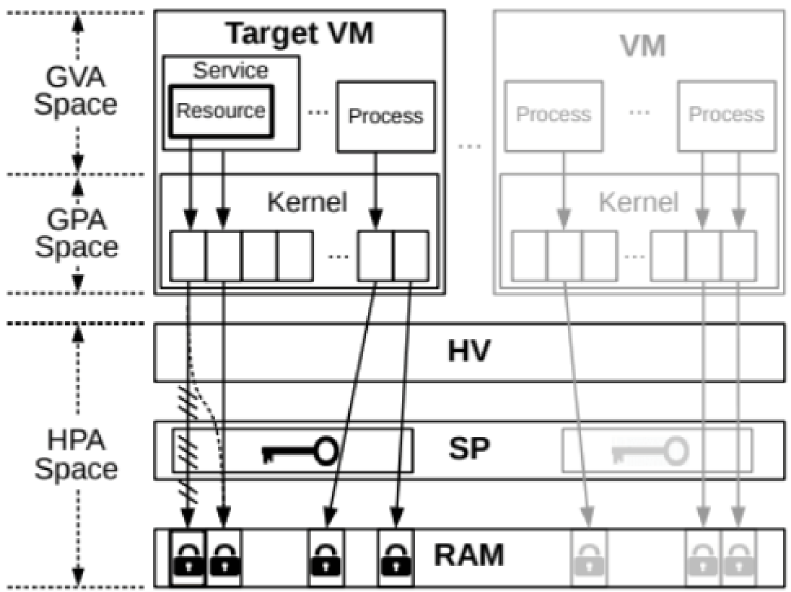


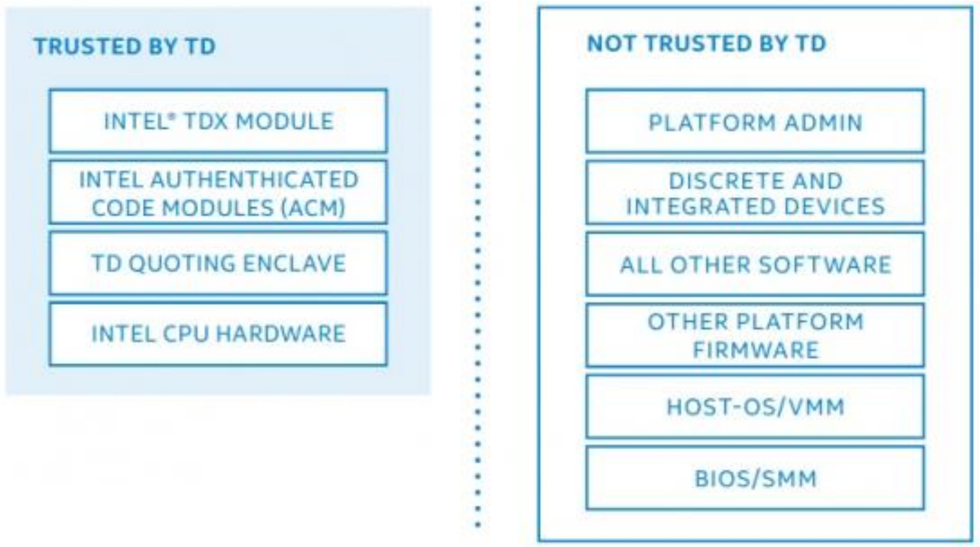
Future Directions – Other TEE Models

- Virtual Machine TEE
  - AMD SEV
  - Intel TDX

- What are other ways to interrupt a TEE in the above models?

- What is the impact?
  - Guest OSS
  - Cryptographic Services
  - Other Applications
  - …

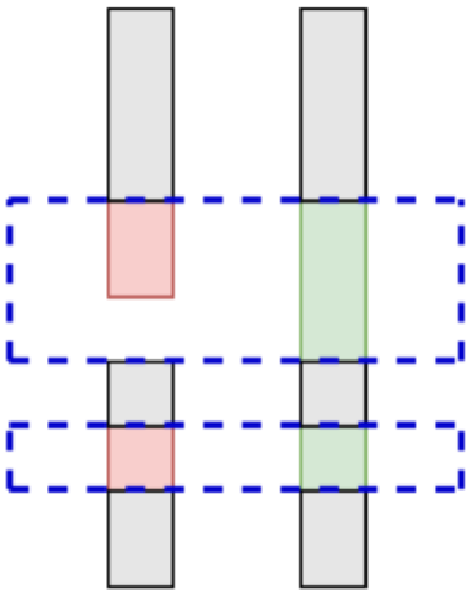




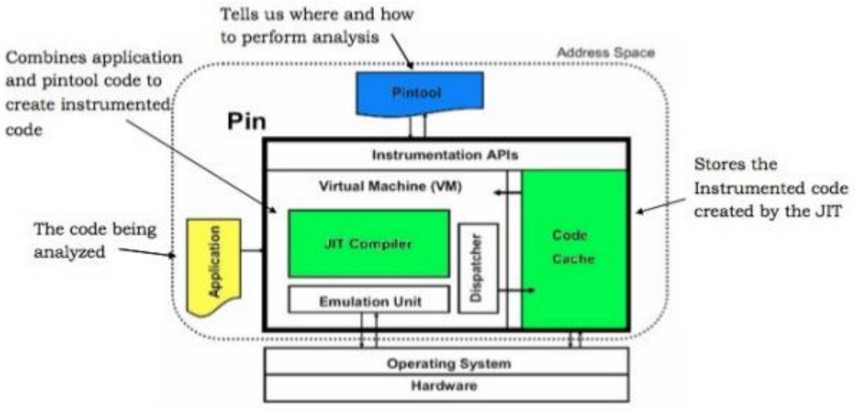
Future Directions – Non-cryptographic Application of Enclaves

- Data-dependent secret-processing applications
  - Confidential Deep Learning (
  - Trusted Database (EnclaveDB)

- Automated Leakage Analysis and Exploit Generation
  - Fuzzing and Taint Analysis
  - Dynamic Analysis



Pin ?



27

Future Directions – Mitigation

- Compiler-based Solutions
  - Balancing secret-dependent branches with dummy instructions

- System-level Mitigation
  - Self-paging Enclave (Autarky)

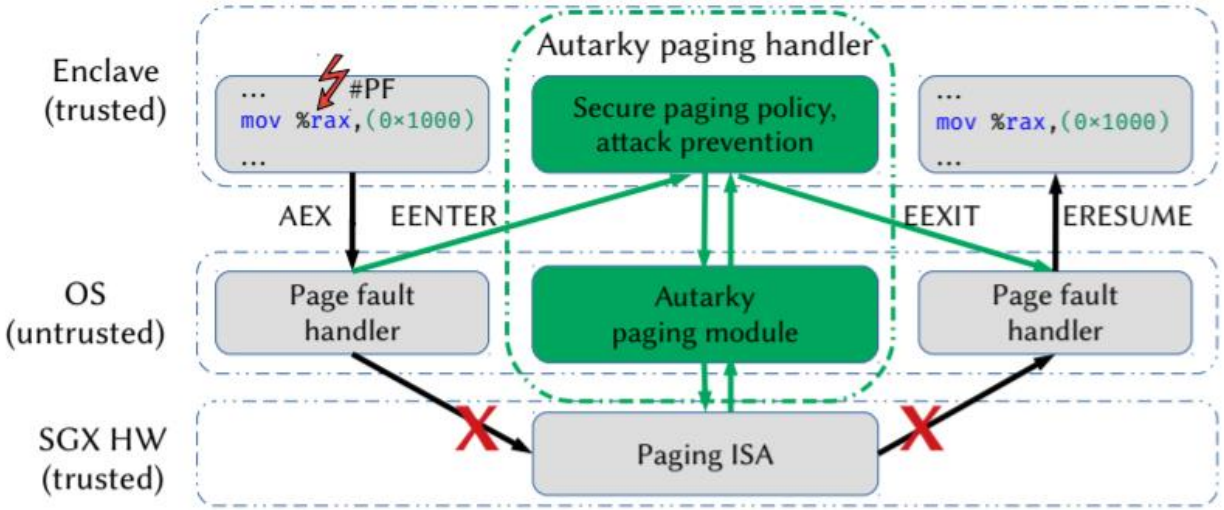
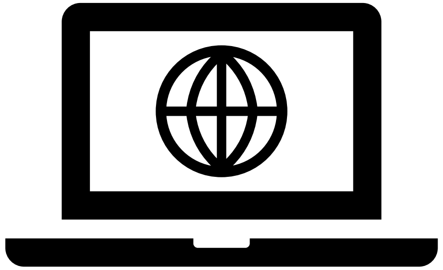


Figure 2. Autarky enforces invocation of an enclave's self-paging handler on each page fault.

Questions?!

Daniel Moghimi
@danielmgmi



https://github.com/jovanbulck/sgx-step