Hardware-Software Codesign for Mitigating Spectre

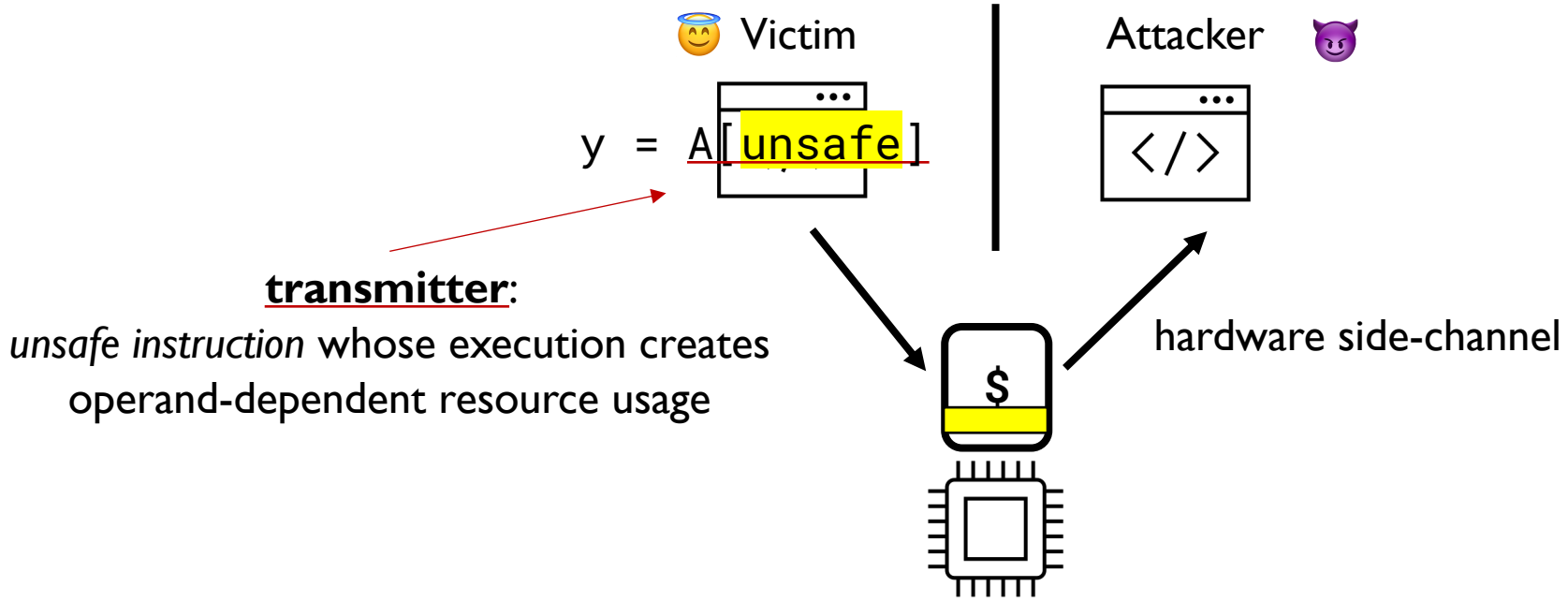
Nicholas Mosier,[1] Kate Eselius,[1] Hamed Nemati,[1,2] John Mitchell,[1] Caroline Trippel[1]

PLARCH'23 • June 17, 2023

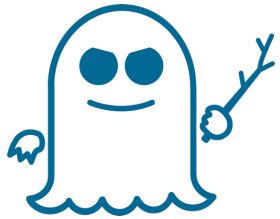
[1]Stanford University

[2]CISPA Helmholtz Center for Information Security

Hardware Side-Channel Attacks



Spectre Attacks



```
1: if (x < len) {
2:   y = A[x];
3:   z = B[y];
4: }
```

speculatively accessed secret → `y`

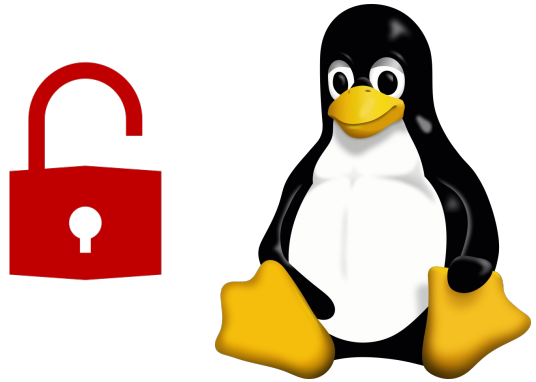
← mispredicted branch introducing speculative execution

← transmitter (load) which leaks secret

Spectre attacks exploit **control- or data-flow mispredictions** in hardware to **speculatively leak sensitive data** via transmitters.

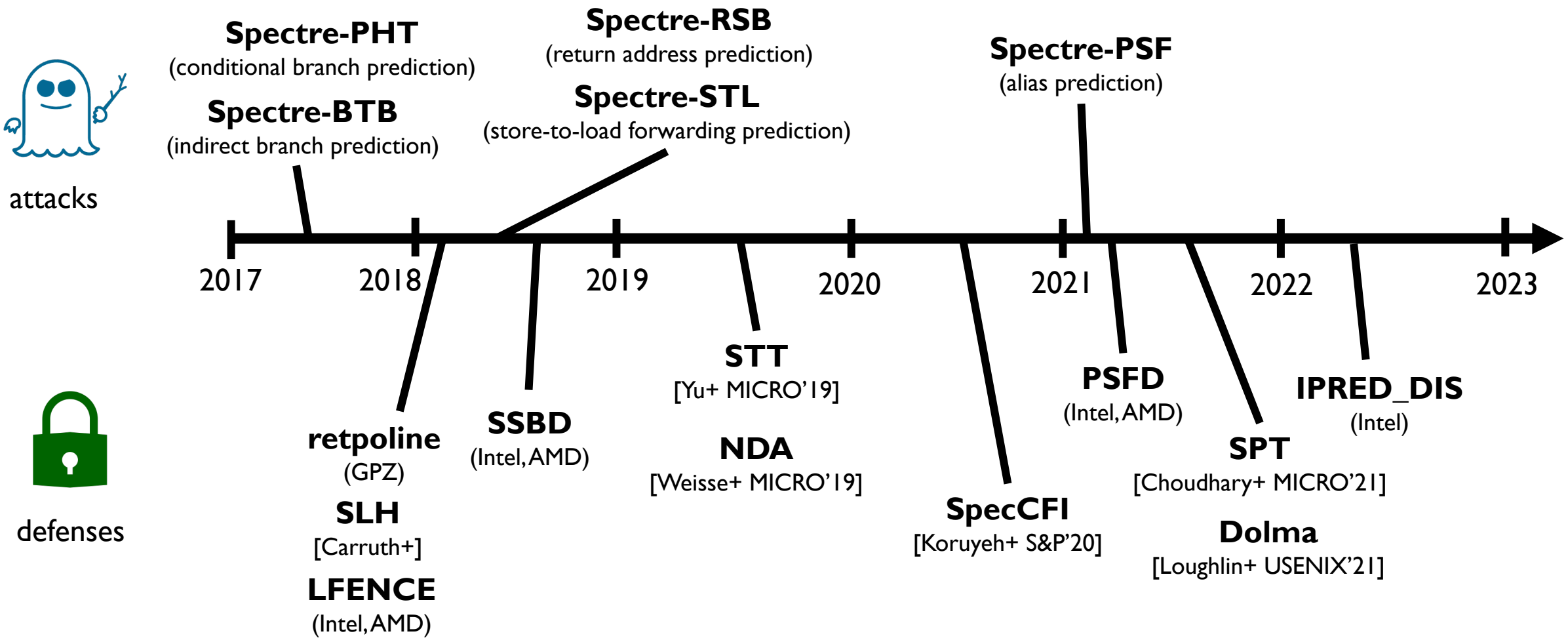
OpenSSL
Cryptography and SSL/TLS Toolkit

Spectre can leak cryptographic keys…



Spectre can leak arbitrary kernel memory, including memory of other processes…

Spectre Attacks and Defenses

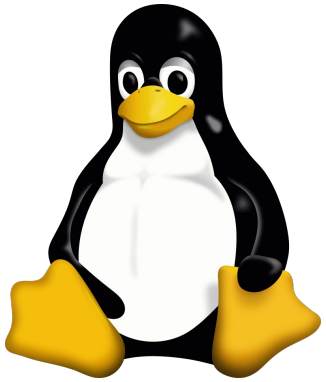


OpenSSL
Cryptography and SSL/TLS Toolkit

Spectre and Meltdown Attacks Against OpenSSL

The OpenSSL Technical Committee (OTC) was recently made aware of several potential attacks against the OpenSSL libraries which might permit information leakage via the Spectre attack.[1] Although there are currently no known exploits for the Spectre attacks identified, it is plausible that some of them might be exploitable.

Local side channel attacks, such as these, are outside the scope of our security policy, however the project generally does introduce mitigations when they are discovered. In this case, the OTC has decided that these attacks will **not** be mitigated by changes to the OpenSSL code base. The full reasoning behind this is given below.

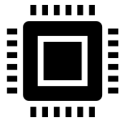


Spectre variant 1

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping [2] to avoid any usable disclosure gadgets. However, it may not cover all attack vectors for Spectre variant 1.

The Spectre Mitigation Challenge

Efficiently mitigating all Spectre leakage due to **any combination** of speculation primitives is **hard**.



Comprehensive hardware mitigations

High complexity

Hardware speculation controls

High overhead; Incomplete

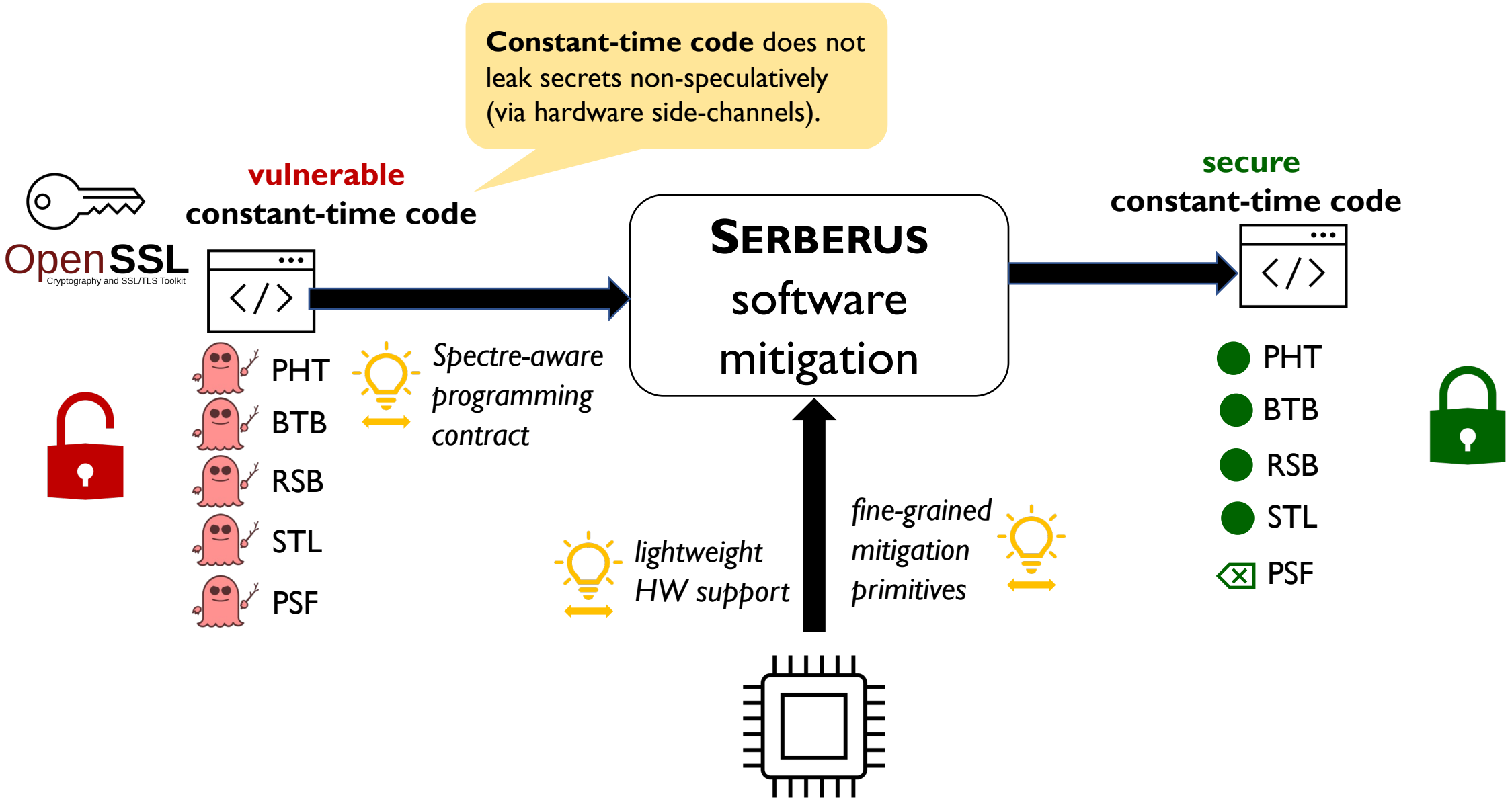
Software-only mitigations

High overhead; Incomplete

	PHT	BTB	RSB	STL	PSF	Deployable?	General?	Overhead
STT [Yu+ MICRO'19]	✓	✓	✓	✓	✓			14.5%***
NDA [Weisse+ MICRO'19]	✓	✓	✓	✓	✓		✓	45%
Dolma [Loughlin+ USENIX'21]	✓	✓	✓	✓	✓			42%
SPT [Choudhary+ MICRO'21]	✓	✓	✓	✓	✓		✓	42%
SSBD (Intel, AMD)				✓	✓	✓	✓	10%***
PSFD (Intel, AMD)					✓	✓	✓	???***
IPRED_DIS (Intel, AMD)		✓				✓	✓	<1%***
Speculative load hardening	✓					✓	✓	~75%
retpoline		✓	+			✓	✓	???***
Naive LFENCE insertion	✓			✓	✓	✓	✓	>300%
Best deployable	✓	✓		✓	✓	✓	✓	>100%***

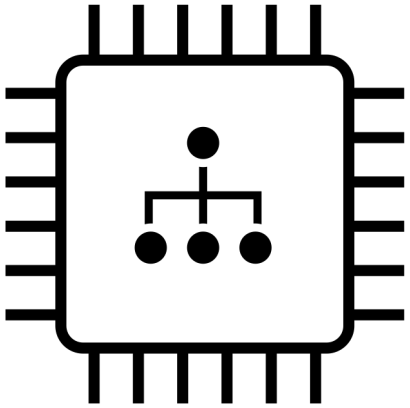
SERBERUS

Comprehensive, Efficient, Proven Spectre Mitigation for Constant-Time Crypto Code


Constant-time code does not leak secrets non-speculatively (via hardware side-channels).
vulnerable
constant-time code
OpenSSL
Cryptography and SSL/TLS Toolkit
PHT
BTB
RSB
STL
PSF
Spectre-aware programming contract
SERBERUS
software
mitigation
lightweight HW support
fine-grained mitigation primitives
secure
constant-time code
PHT
BTB
RSB
STL
PSF

Co-Design Area 1: Constraining Speculation

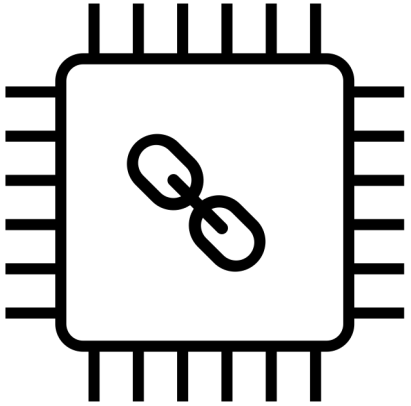
Default speculation model



unconstrained speculative control- and data-flow

Features on existing x86 HW
1. indirect branch tracking
2. shadow stack
3. RRSBA_DIS
4. PSFD

SERBERUS speculation model



constrained speculative control- and data-flow

Precise static program analysis **intractable**!

Precise static program analysis **easy**!

Co-design opportunity: other **low-cost speculation constraints** to make static analyses more precise

Co-Design Area 2: Programming Contract

Constant-time (CT) programming permits **vulnerable code patterns** that inhibit efficient mitigations

1. **Latent CT violations**

```
if (0)
  x = A[secret];
```

2. **Passing secret arguments by value**

```
safe(secret);

int leak(int idx) {
  return A[idx]; }
```

SERBERUS' Solution:
static constant-time (CTS) programming extends constant-time with:

1. require **static security types** of variables

```
if (0)
  x = A[public];
```

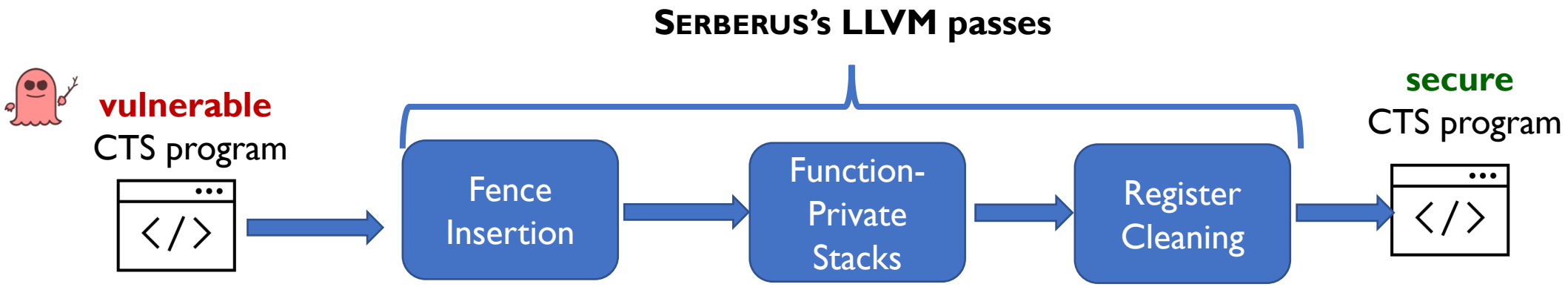
2. **Spectre-aware calling convention** that forbids passing secret arguments by value

```
safe(public);
```

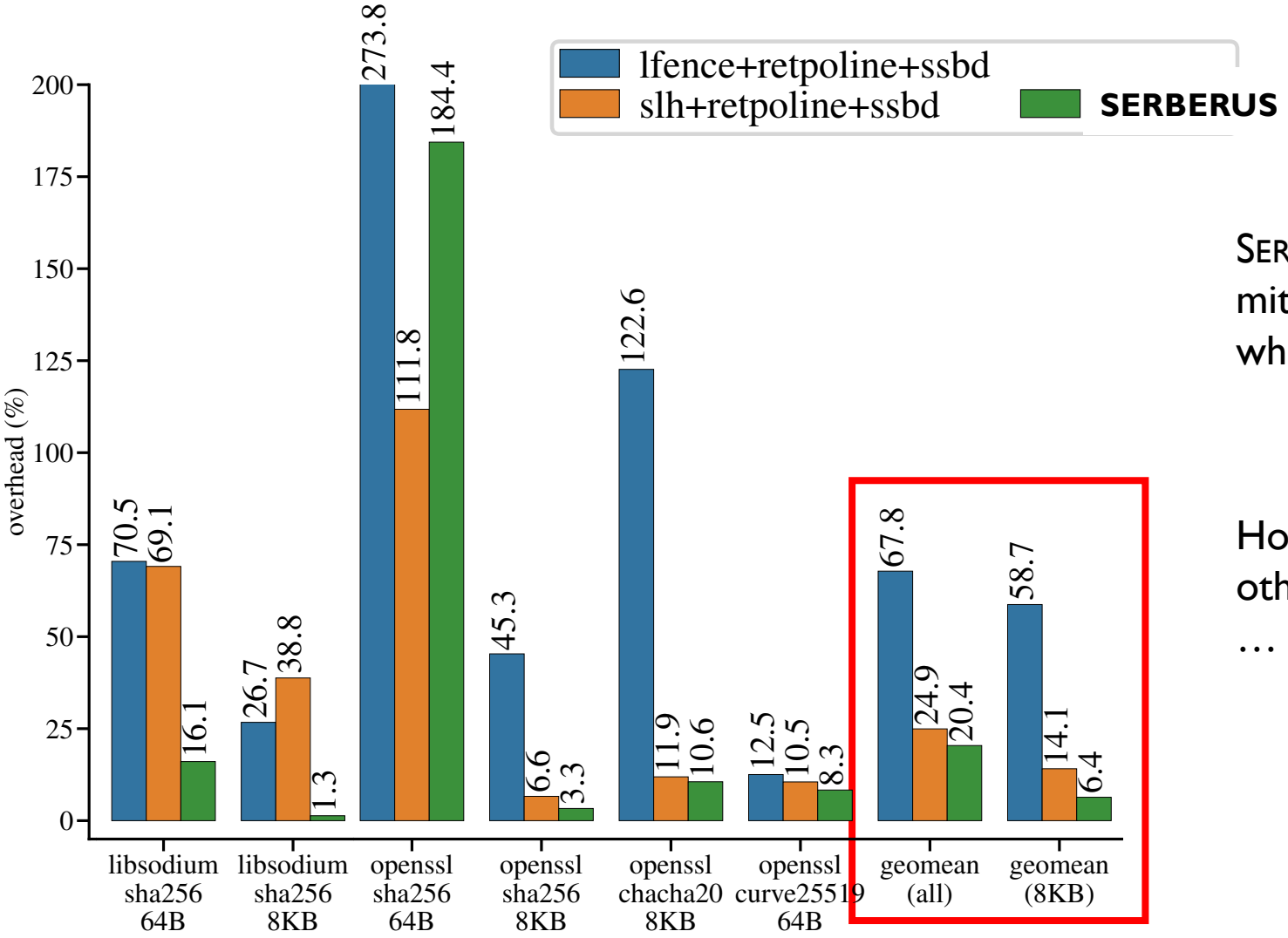
Co-design opportunity: other **modest programming contract requirements** to make static analyses more precise

SERBERUS' Passes

- Consists of **three intraprocedural passes**
  - **Fence Insertion**: inserts LFENCEs into program
  - **Function-Private Stacks**: assigns distinct stacks to each function to prevent Spectre leakage due to stack sharing
  - **Register Cleaning**: zeroes out registers that may hold secrets before leaving the function



SERBERUS' Performance



SERBERUS **outperforms** state-of-the-art mitigations in the crypto primitives we evaluate while offering **stronger security guarantees**

However, SERBERUS incurs high overhead in other application domains…
… e.g., **>300% overhead** for SPEC CPU2017

SERBERUS' **Fence Insertion** Pass

- Frames speculation fence (LFENCE) insertion as a ***minimum directed multicut*** problem over the **transient control-flow graph**
- **Sources** are loads or stores that may access secrets
- **Sinks** are dependent transmitters

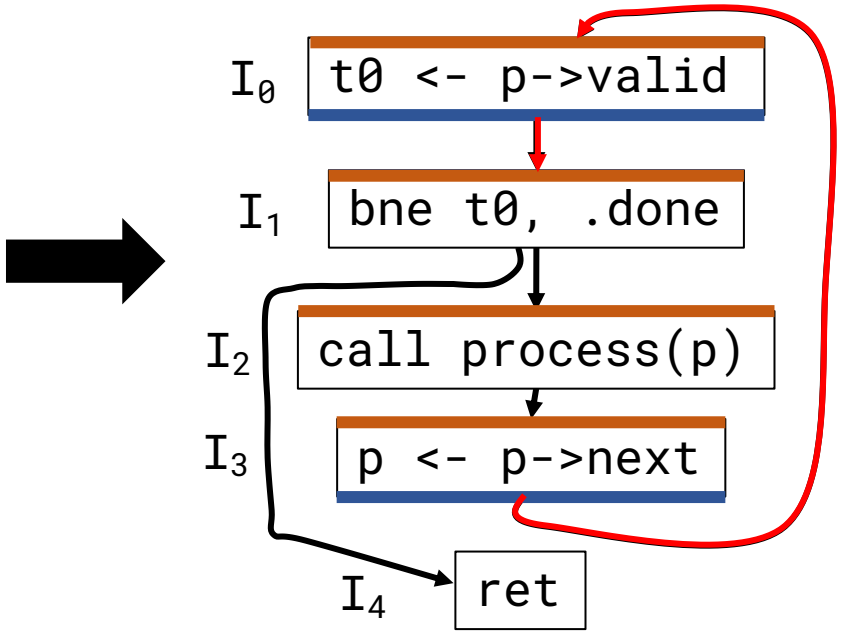
Original Procedure

```
while (p->valid) {
  process(p);
  p = p->next;
}
```

Source-Sink Pairs

(I_0, I_1) (I_3, I_0)

(I_3, I_2)

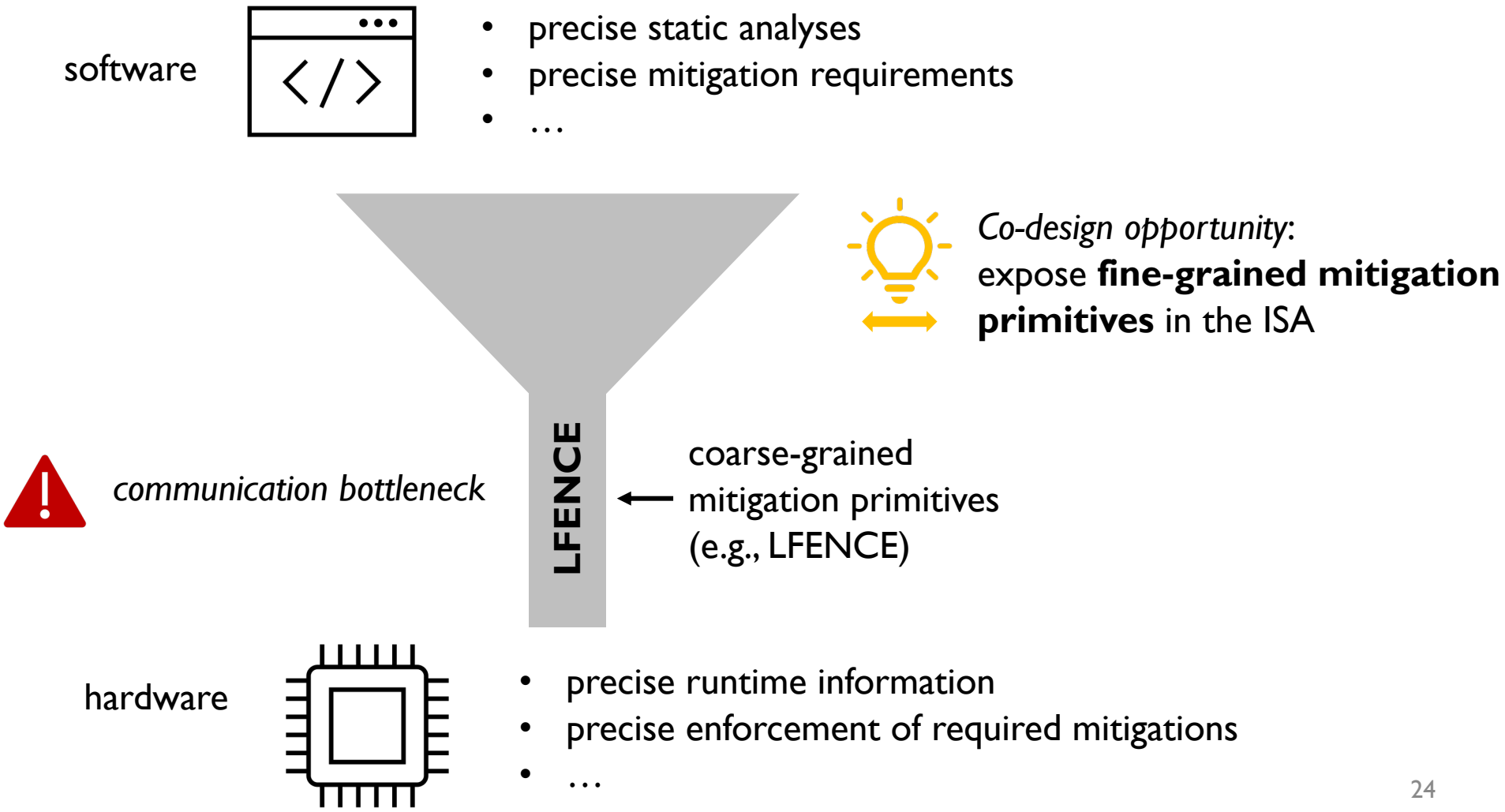


Mitigated Procedure

```
while (t0 = p->valid,
       LFENCE(),
       t0){
  process(p);
  p = p->next;
  LFENCE();
}
```

reduces to **disabling conditional branch prediction**…

Coarse-Grained Mitigation Problem



Example Fine-Grained Mitigation: **NOSPEC**

- **NOSPEC**: instruction flag that delays an instruction's execution until it becomes non-speculative:

```
NOSPEC mov rsi,[rdi]
```

Original Procedure

```
while (p->valid) {
  process(p);
  p = p->next;
}
```

***Coarsely**-Mitigated Procedure*

```
while (t0 = p->valid,
       LFENCE(),
       t0){
  process(p);
  p = p->next;
  LFENCE();
}
```

reduces to **disabling conditional branch prediction**…

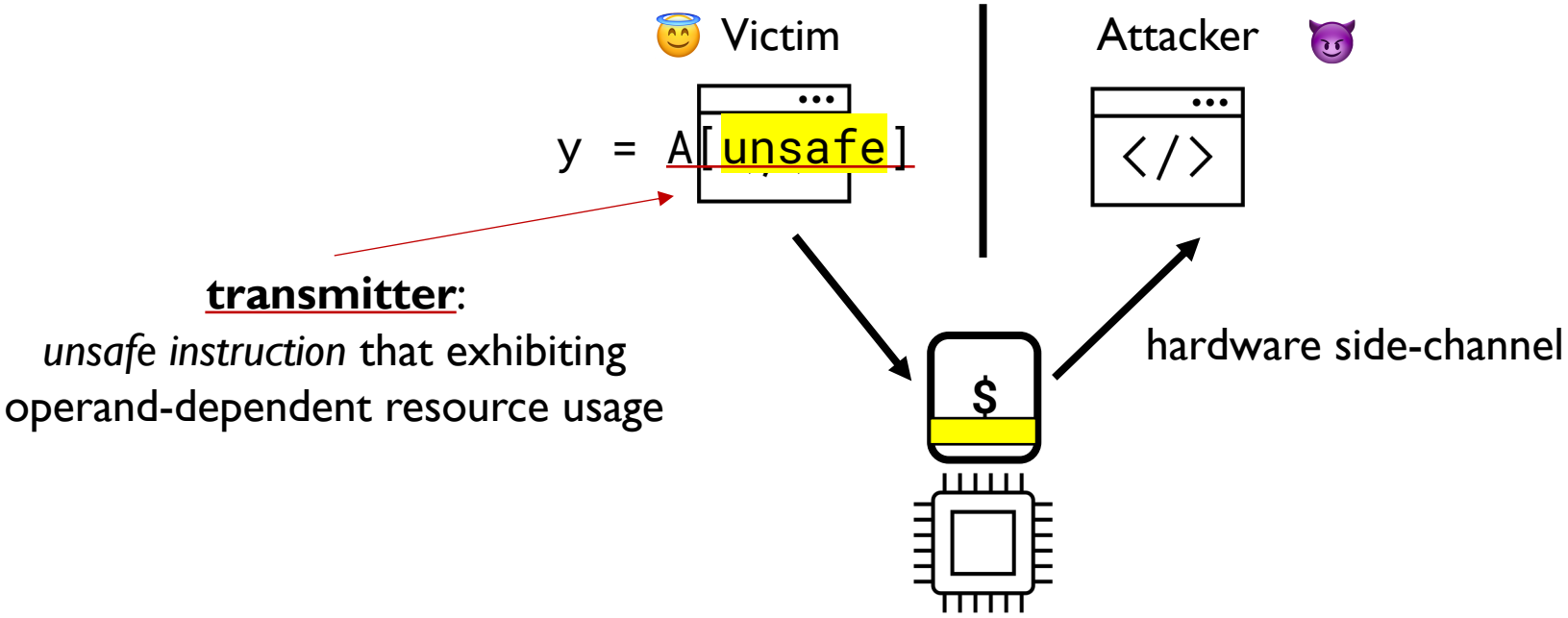
***Finely**-Mitigated Procedure*

```
while (nospec(p->valid)) {
  process(p);
  p = nospec(p->next);
}
```

allows secure speculation to proceed!

END

Hardware Side-Channel Attacks



Constant-Time (CT) Programming

CT programs do not pass **secrets** to sensitive (*unsafe*) **transmitter** operands in any **sequential execution**

forbidden

control-flow ✗ `if (unsafe)`

load ✗ `y = A[unsafe];`

store ✗ `A[unsafe] = y;`

division ✗ `x = a / b;`

Constant-time programs are 

sequentially secure

Spectre Attacks on CT Code

However, **Spectre attacks** can still exploit **transient execution** to steer secrets to transient transmitters

permitted by CT

```
if (x < A_len)
  y = A[x];
  z = B[y];
```

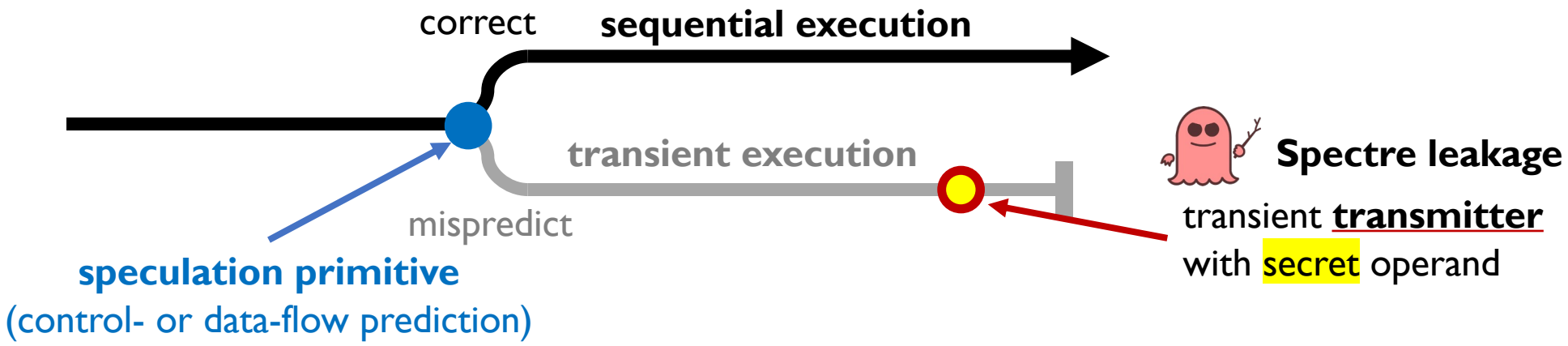
transient
(instruction does not commit)

Constant-time programs are  but 

sequentially secure

transiently insecure

Spectre Terminology



Speculation Primitives

control-flow
speculation primitives

```
if (x < A_len) {
  y = A[x];

  z = B[y];
}
```

PHT
conditional branch

```
f = &g;
(*f)(secret);

int h(int x) {
  return A[x];
}
```

BTB
indirect branch prediction

```
int f(x) {
  return x;
}
int g(x) {
  y = h(x);
  return A[x]; }
```

RSB
return address prediction

data-flow
speculation primitives

```
x = secret;
x = 0;
y = A[x];
```

STL
store-to-load forwarding

```
x = secret;
y = 0
y = A[y];
```

PSF
predictive store forwarding

Mitigating Spectre in Software

Efficiently mitigating all Spectre leakage due to ***any combination of*** {PHT, BTB, RSB, STL, PSF} is ***hard***.

Two approaches:

- ⌫ Disable speculation primitive
- ● Prevent secret-dependent transmitters

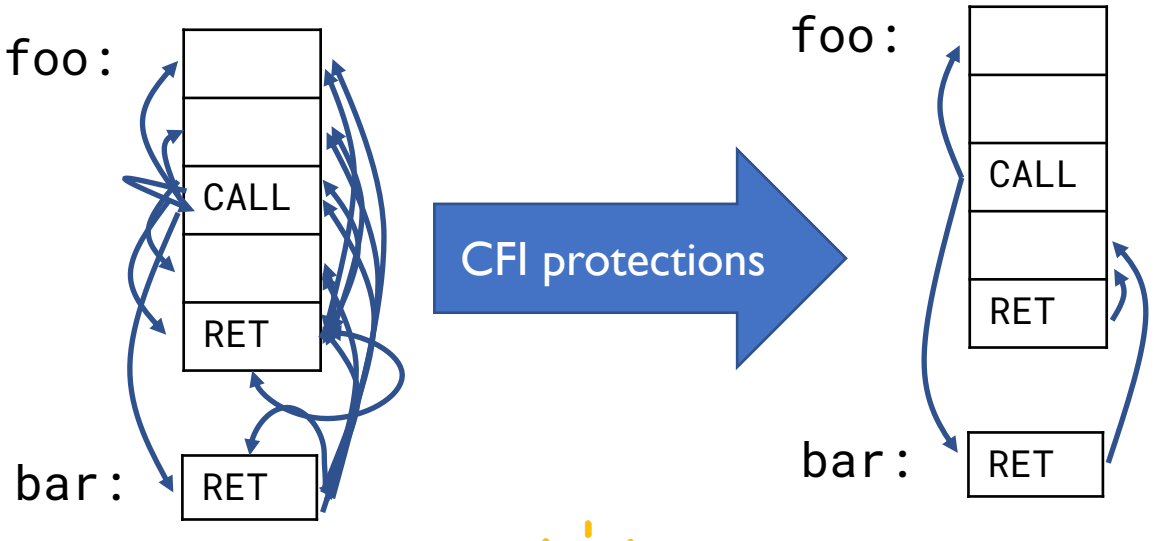
Mitigation	Leakage	Proof	PHT	BTB	RSB	STL	PSF
INTEL-LFENCE	-	-	⌫	-	-	-	-
LLVM-SLH	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	-	-	-	-
RETPOLINE	-	-	-	⌫	↑	-	-
SSBD	-	-	-	-	-	⌫	⌫
PSFD	-	-	-	-	-	-	⌫
BLADE	$[\![\cdot]\!]_{\text{ct}}$	✓	●	-	-	-	-
SWIVEL-CET	$[\![\cdot]\!]_{\text{mem}}$	✗	●	●	●	⌫	⌫
SERBERUS (ours)	$[\![\cdot]\!]_{\text{ct}}$	✓	●	●	●	●	⌫

SERBERUS Insights

1. **Hardware model**: CFI protections enable comprehensive analysis of transient control-flow
2. **Software requirements**: static constant-time (CTS) overcomes unsafe code patterns permitted by CT programming
3. **Leakage characterization:** Spectre leakage is due to four classes of *taint primitives*, which assign secrets to publicly-typed variables

SERBERUS' Hardware Model

Constraining transient control-flow



Unconstrained transient control-flow

Intractable to analyze…

SERBERUS **constrains** transient control-flow with **CFI protections** from Intel CET**:**

- *Indirect branch tracking* (forward-edge)
- *Shadow stack* (backward-edge)

Easy to analyze!

Constraining transient data-flow

PSF

SERBERUS **disables PSF**, since it is intractable to efficiently mitigate in software.

SERBERUS' Software Requirements: CT Limitations

Is CT at least a good starting place for Spectre mitigations? **No.**

Two **unsafe CT code patterns** *almost always* leak secrets transiently and inhibit efficient mitigations:

1. *Latent CT violations*

```
if (0)
  x = A[secret];
```

2. *Spectre-unaware calling convention*

```
process(secret);
```

Constant-Time Limitation 2

② Spectre-unaware calling convention

```
process(secret);

int process(int secret) {
  return secret + 1;
}

int leak(int idx) {
  return A[idx];
}
```

Underlying issue: passing/returning secrets by value is *inherently dangerous*

Solution:

We propose **static constant-time (CTS)**, which extends CT to prohibit these unsafe code patterns ① and ②

Taint Primitives in CTS Programs

- **Taint primitive**: instruction that assigned a **secret** value to a **publicly-typed** variable when executed
- Spectre leakage in CTS programs occurs when a **taint primitive** passes its result to a **transmitter**
- **Four classes** of taint primitives in CTS programs
- Suggests novel Spectre mitigation approach:
  -  *Eliminate* taint primitive
  -  *Prevent* taint-primitive-dependent transmitters

NCAL

non-constant-address load

```
x = *p;
y = A[x];
```

NCAS

non-constant-address store

```
x = 0;
*p = secret;
y = A[x];
```

STKL

uninitialized stack load

```
int x = 0;
y = A[x];
```

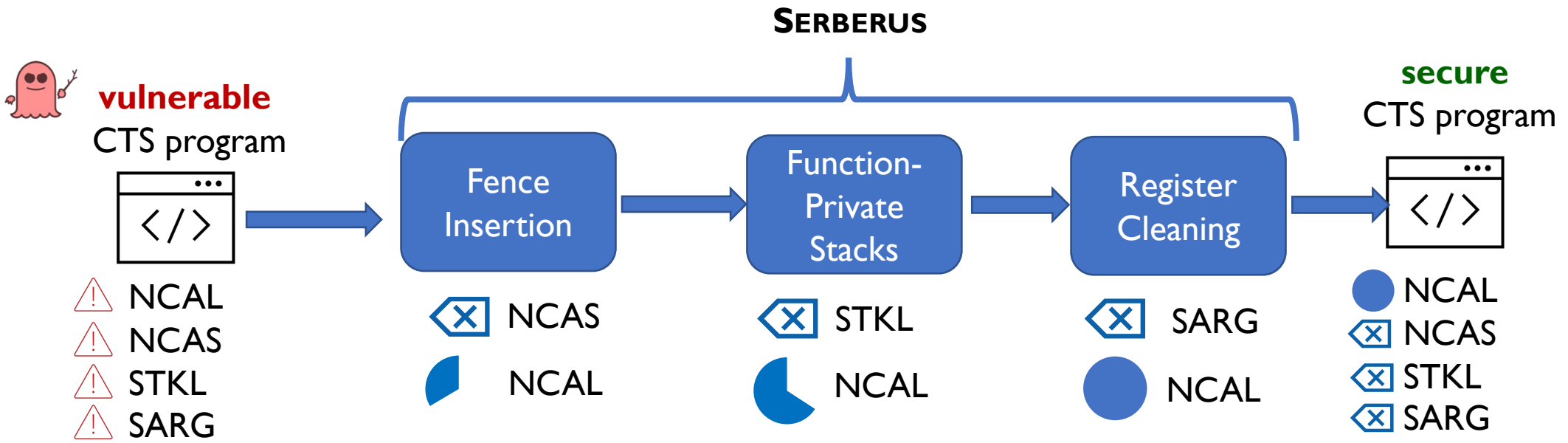
SARG

unexpectedly secret argument

```
foo(int x):
  y = A[x];
```

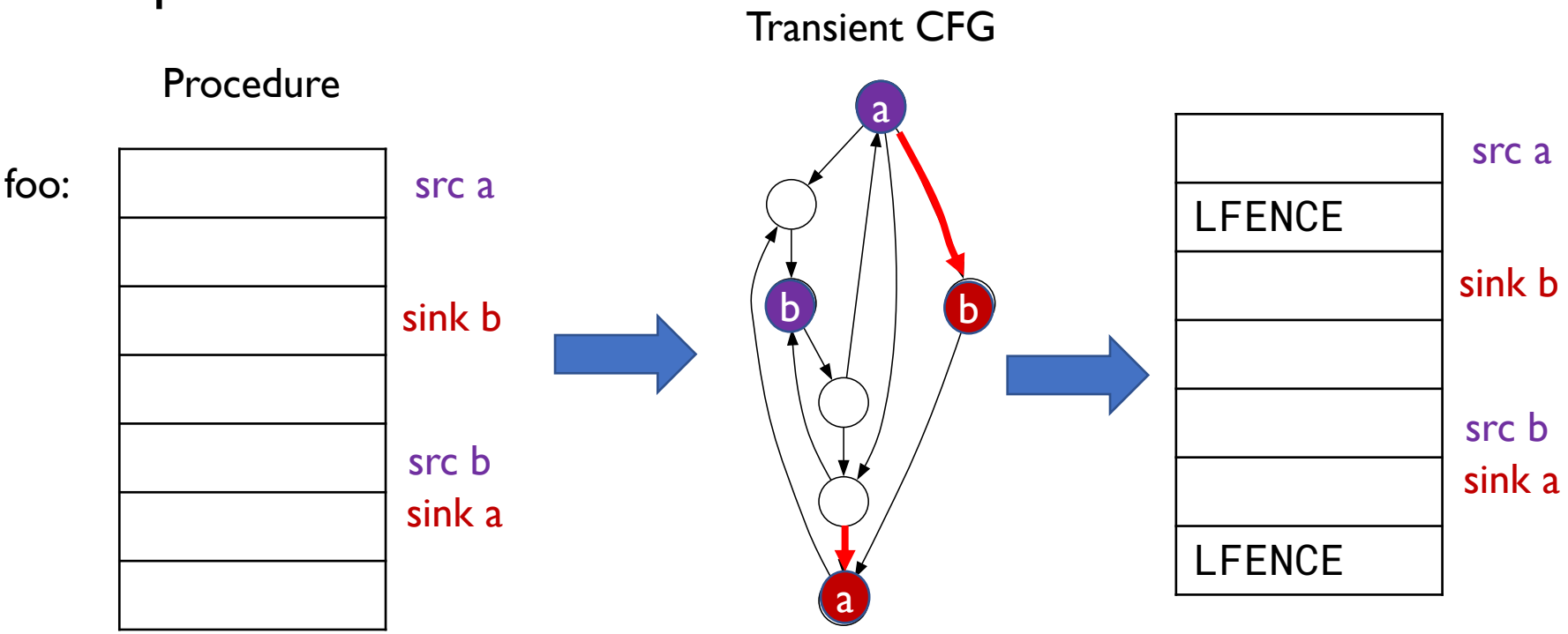
SERBERUS Overview

- SERBERUS eliminates ***all secret leakage*** in CTS programs due to **any combination of** {PHT, BTB, RSB, STL} speculation primitives.
- Consists of **three intraprocedural passes**



SERBERUS' **Fence Insertion** Pass

- Frames speculation fence (LFENCE) insertion as a min-cut problem over the **transient control-flow graph**
- **Sources** are candidate NCAL or NCAS taint primitives
- **Sinks** are dependent transmitters and instructions that may facilitate dependent transmitters



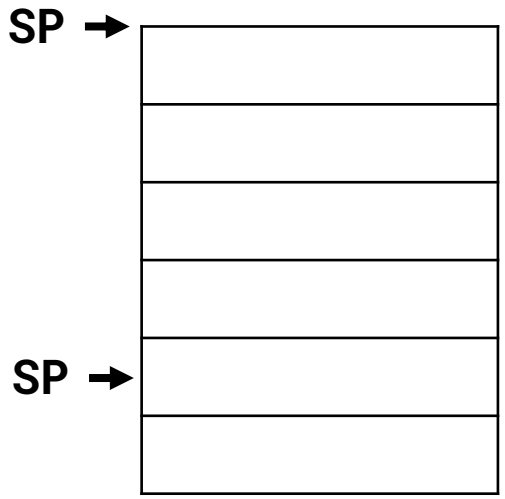
SERBERUS' **Function-Private Stacks** Pass

Stack sharing is the root cause of STKL: a publicly-typed load may read a stale secret from prior procedure's stack frame.

STKL
uninitialized stack load

```
int x = 0;
y = A[x];
```

```
foo() {
  x = secret;
  ...
}
```

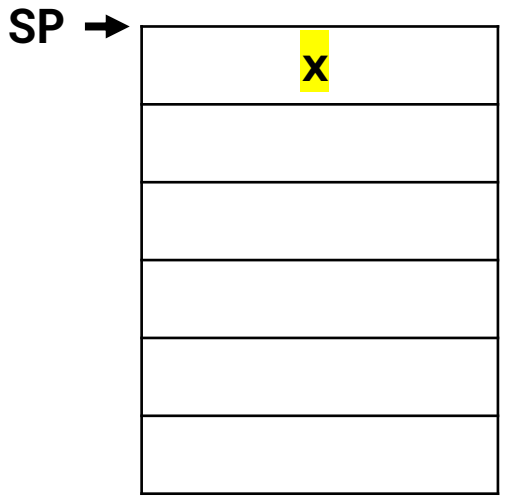


SERBERUS' **Function-Private Stacks** Pass

Stack sharing is the root cause of STKL: a publicly-typed load may read a stale secret from prior procedure's stack frame.

STKL
uninitialized stack load

```
int x = 0;
y = A[x];
```




```
bar() {
  y = 0;
  z = A[y];
}
```

Solution: allocate a **private stack** to each procedure.

```
foo: ENDCALL
  + LD [ZR+PSP_F],SP // load private SP
    SUB SP,SP,k      // frame allocation
  + LD [SP+0],ZR     // probe for overflow
  + ST [ZR+PSP_F],SP // store private SP
    ...
    CALL r1
  + LD [ZR+PSP_F],SP // load private SP
    ...
  + LD [SP+0],ZR     // probe for underflow
    ADD SP,SP,k      // frame deallocation
  + ST [ZR+PSP_F],SP // store private SP
    RET
```

prologue

callsite

epilogue

SERBERUS' **Register Cleaning** Pass

unexpectedly secret argument

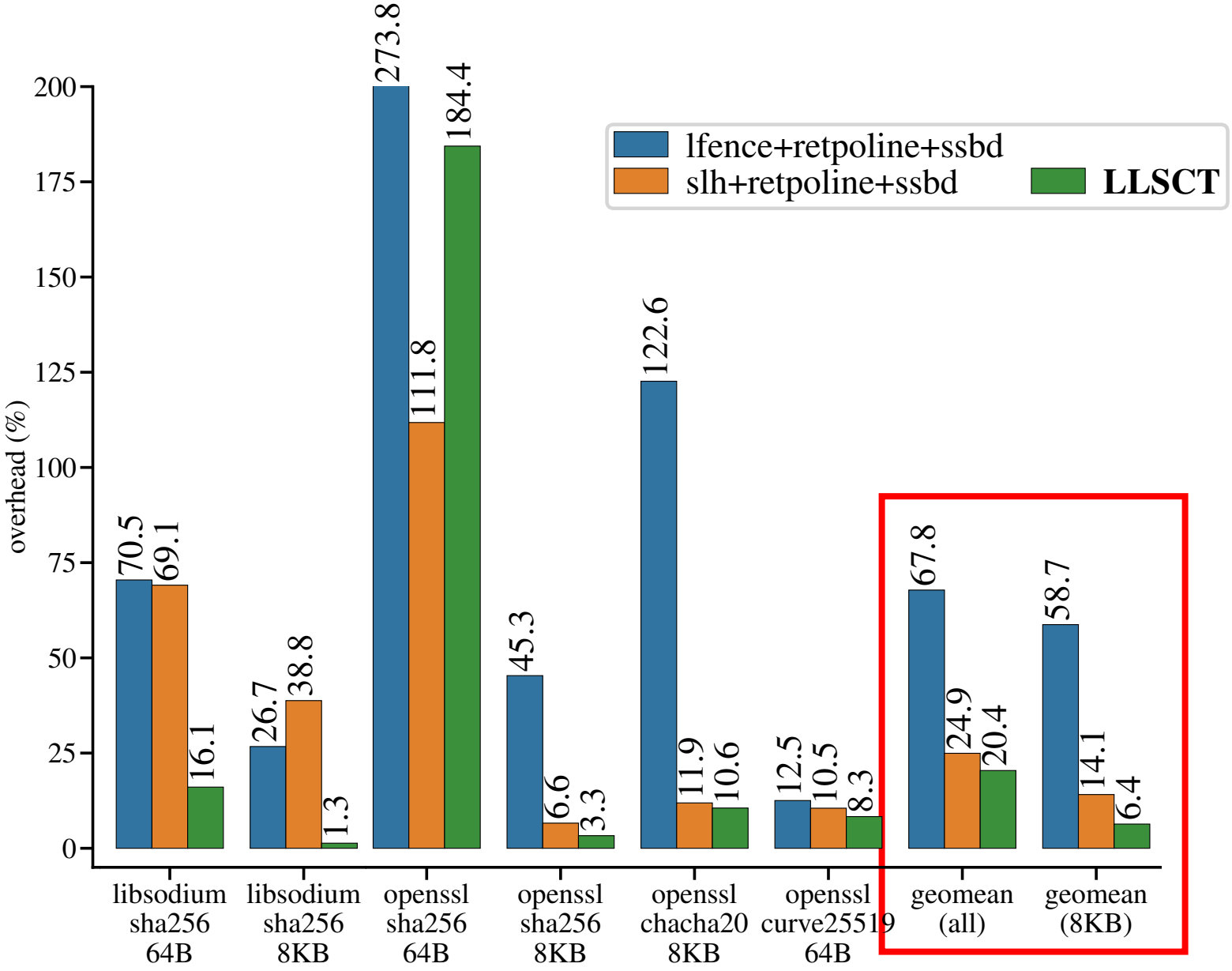
```
foo(int x):
  y = A[x];
```

Zero out non-argument registers before every call/return

```
foo:
  ...
  MOV r2, 0
  MOV r3, 0
  CALL r1
  ...
  MOV r1, 0
  MOV r2, 0
  MOV r3, 0
  RET
```

LLSCT: Implementation of SERBERUS for LLVM

- Implemented as three of LLVM IR and machine passes
- Requires **no user annotations**
- Benchmarked runtime performance overhead over insecure baseline
- Evaluated against state-of-the-art mitigations:
  - **lfence+retpoline+ssbd**
  - **slh+retpoline+ssbd**
- *Testing setup*: Intel 12th-gen Core i9-12900KS processor (supports Intel CET)
- *Workloads*: crypto primitives from OpenSSL, Libsodium, and HACL*

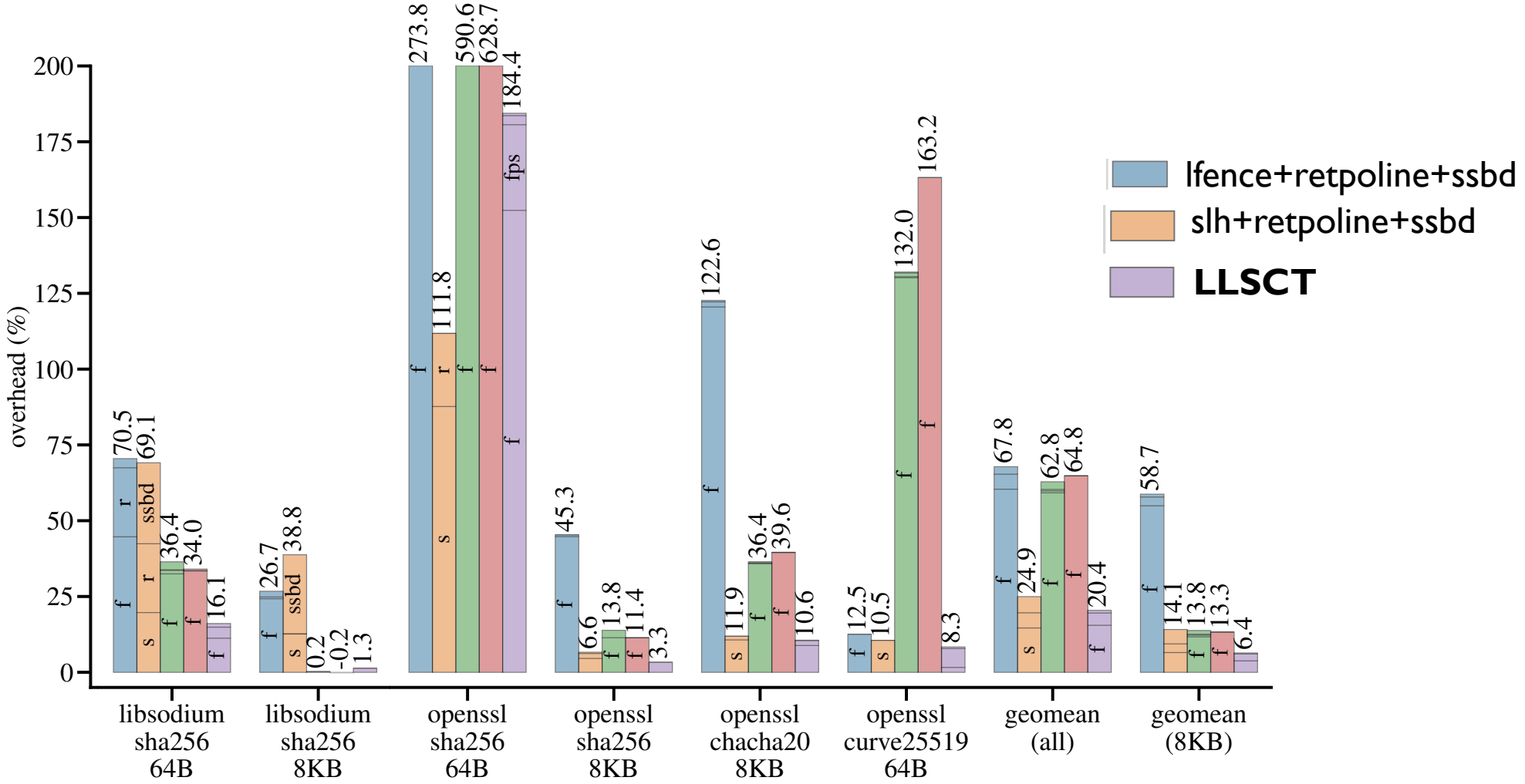

overhead (%)
0
25
50
75
100
125
150
175
200
lfence+retpoline+ssbd
slh+retpoline+ssbd
LLSCT
70.5
69.1
16.1
26.7
38.8
1.3
273.8
111.8
184.4
45.3
6.6
3.3
122.6
11.9
10.6
12.5
10.5
8.3
67.8
24.9
20.4
58.7
14.1
6.4
libsodium sha256 64B
libsodium sha256 8KB
openssl sha256 64B
openssl sha256 8KB
openssl chacha20 8KB
openssl curve25519 64B
geomean (all)
geomean (8KB)

Conclusions and Future Work

- SERBERUS is the first software mitigation for Spectre-PHT/BTB/RSB/STL leakage in CT programs
- LLSCT: implementation of SERBERUS for LLVM
- LLSCT **outperforms** state-of-the-art mitigations in the crypto primitives we evaluate while offering **stronger security guarantees**
- Future work: overcoming performance limitations of applying LLSCT more broadly in non-crypto-code

Questions?

nmosier@stanford.edu



f = *LFENCE*

r = *retpoline*

slh = *speculative load hardening*

ssbd = *STL disable*

fps = *function-private stacks*

Mitigating Spectre in Software

Efficiently mitigating all Spectre leakage due to ***any combination of*** {PHT, BTB, RSB, STL, PSF} is ***hard***.

Two approaches:

- ⌫ Disable speculation primitive
- ● Prevent secret-dependent transmitters

Three tools:

- **Serialization instructions** (e.g., LFENCE)
- **Code rewriting** (e.g., SLH)
- **Speculation controls** (e.g., SSBD)

Mitigation	Leakage	Proof	PHT	BTB	RSB	STL	PSF
INTEL-LFENCE [29]	-	-	⌫	-	-	-	-
LLVM-SLH [30]	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	-	-	-	-
RETPOLINE [31]	-	-	-	⌫	↑	-	-
IPREDD [32]	-	-	-	⌫	-	-	-
SSBD [33]	-	-	-	-	-	⌫	⌫
PSFD [34]	-	-	-	-	-	-	⌫
F+RETP+SSBD	-	-	⌫	⌫	-	⌫	⌫
S+RETP+SSBD	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	⌫	-	⌫	⌫
BLADE [35]	$[\![\cdot]\!]_{\text{ct}}$	✓	●	-	-	-	-
SWIVEL-CET [36]	$[\![\cdot]\!]_{\text{mem}}$	✗	●	●	●	⌫	⌫
SERBERUS (ours)	$[\![\cdot]\!]_{\text{ct}}$	✓	●	●	●	●	⌫