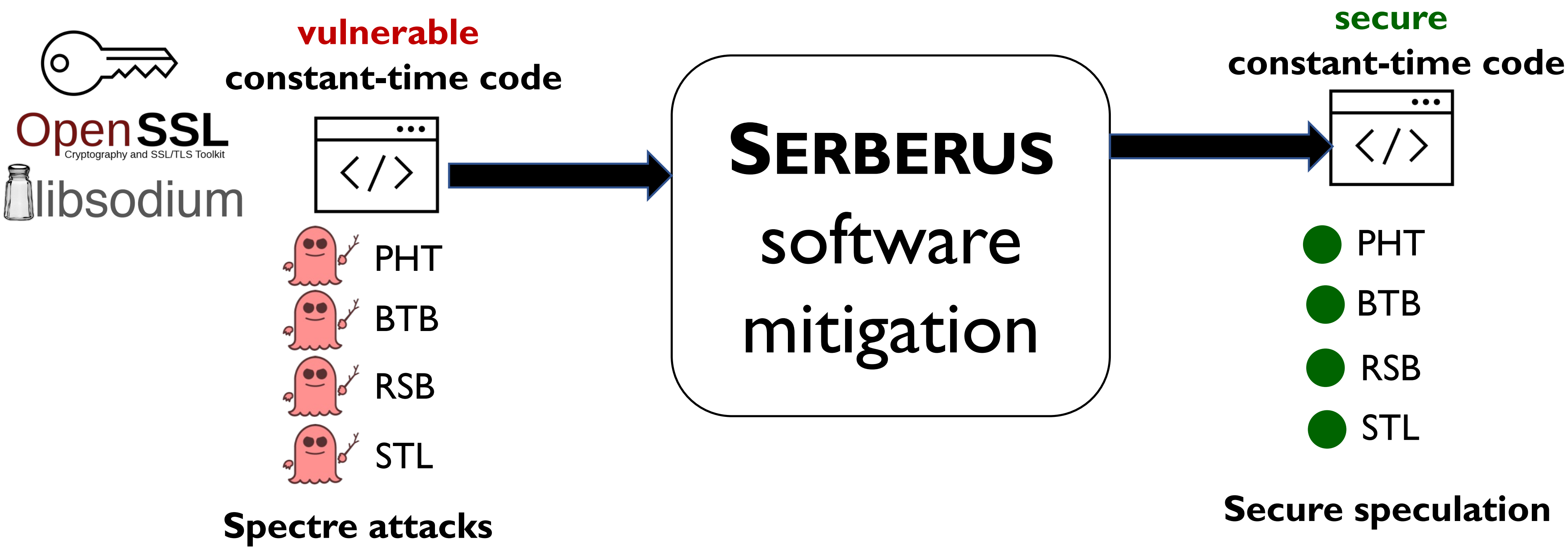
SERBERUS: Comprehensive Spectre Mitigations for Constant-Time Code

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

April 13, 2023 • Stanford Security Workshop

[1]Stanford University

[2]CISPA Helmholtz Center for Information Security



OpenSSL
Cryptography and SSL/TLS Toolkit

libsodium

vulnerable
constant-time code

PHT
BTB
RSB
STL

Spectre attacks

SERBERUS
software
mitigation

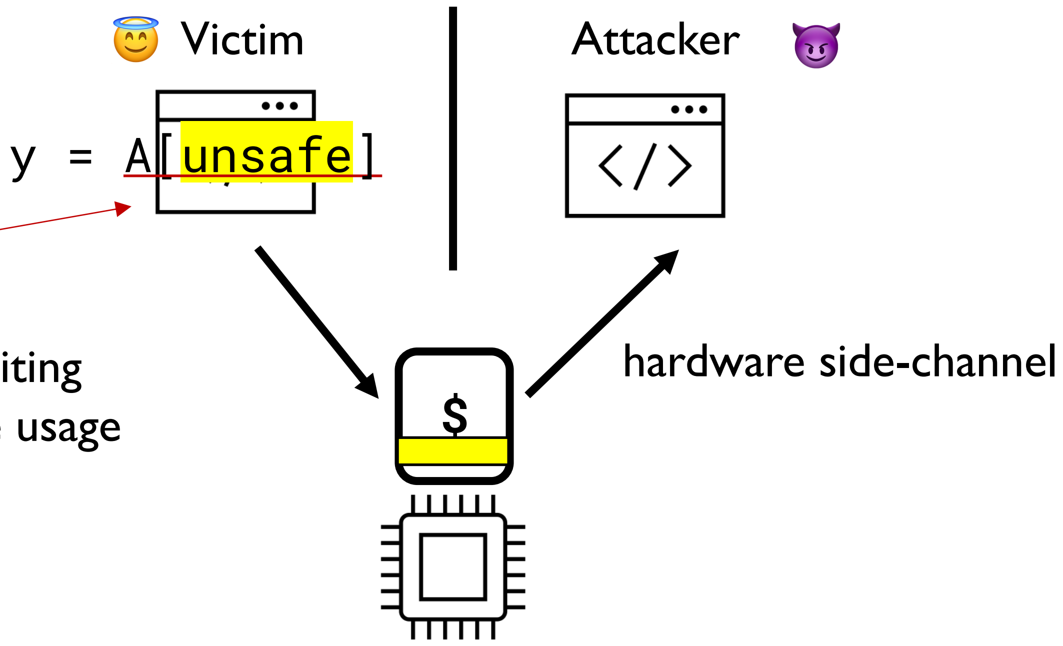
secure
constant-time code

● PHT
● BTB
● RSB
● STL

Secure speculation

✓ First comprehensive software mitigation for PHT, BTB, RSB, STL speculation primitives
✓ Proven correct

Hardware Side-Channel Attacks



`y = A[unsafe]`

hardware side-channel

transmitter:
unsafe instruction that exhibiting
operand-dependent resource usage

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

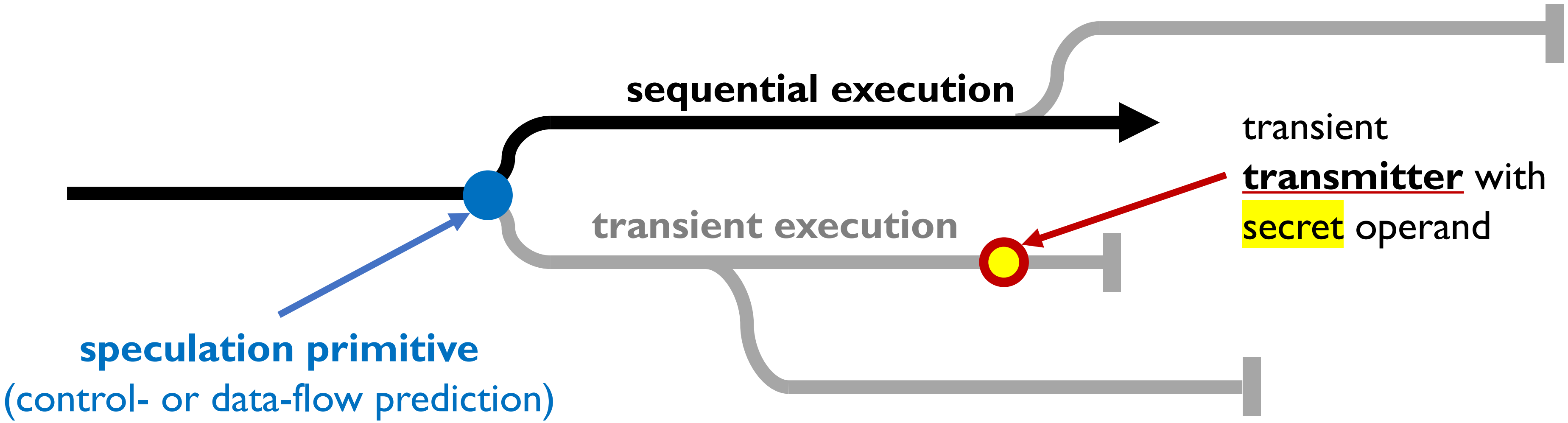
☑

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

transient
(instruction does not commit)

Constant-time programs are sequentially secure but transiently insecure

Spectre Terminology



sequential execution

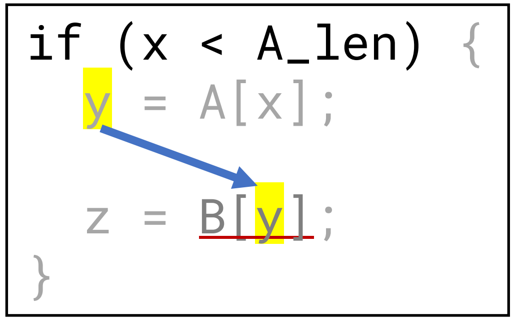
transient execution

transient
transmitter with
secret operand

speculation primitive
(control- or data-flow prediction)

Speculation Primitives

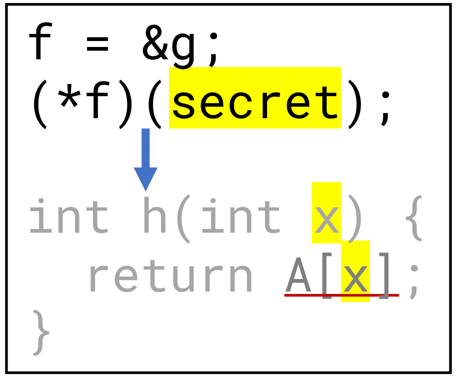
control-flow
speculation primitives



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

  z = B[y];
}
```

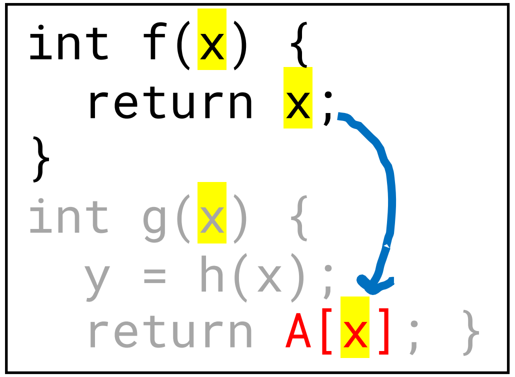
PHT
conditional branch



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

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

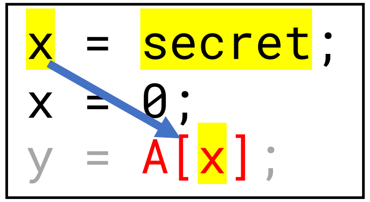
BTB
indirect branch prediction



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

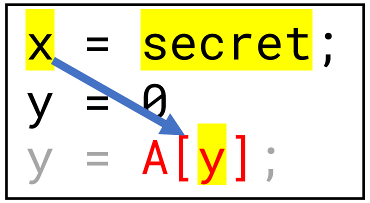
RSB
return address prediction

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

Mitigating all Spectre leakage due to ***any combination of*** {PHT, BTB, RSB, STL, PSF} is ***easy***.

Doing so ***efficiently*** 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)

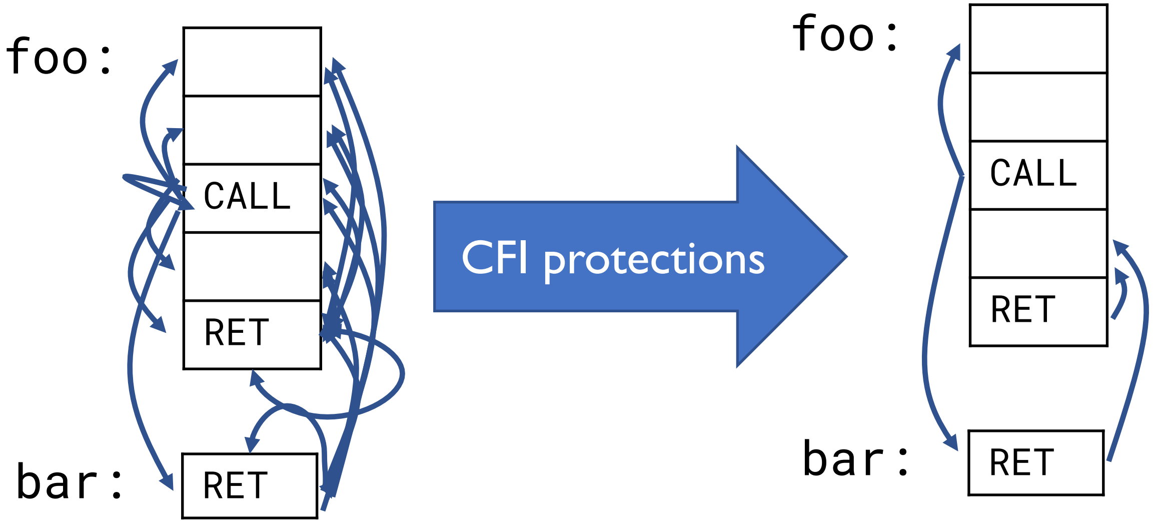
Mitigation	**Leakage**	**Proof**	PHT	BTB	RSB	STL	PSF
INTEL-LFENCE [29]	-	-	⌫	-	-	-	-
LLVM-SLH [30]	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	-	-	-	-
RETPOLINE [31]	-	-	-	⌫	↑	-	-
IPREDD [32]	-	-	-	⌫	-	-	-
SSBD [33]	-	-	-	-	-	⌫	⌫
PSFD [34]	-	-	-	-	-	-	⌫
F+RETP+SSBD	-	-	⌫	⌫	-	⌫	⌫
S+RETP+SSBD	$[\![\cdot]\!]_{\text{arch}}$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

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**.

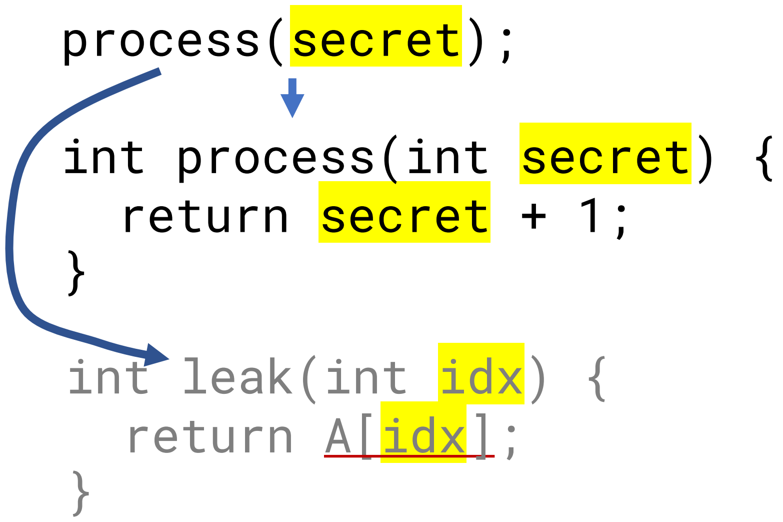
① Latent CT violations

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

Underlying issue: *transmitter's sensitive operand* is statically dependent on a *secretly-typed value*

SERBERUS' Software Requirements: CT Limitations

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 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
- **Four classes** of taint primitives in CTS programs
- Spectre leakage in CTS programs occurs when a ***taint primitive*** *passes its result to a* ***transmitter***
- Suggests novel Spectre mitigation approach:
  - ⌫ *Eliminate* taint primitive
  - ● *Prevent* taint-primitive-dependent transmitters

SERBERUS uses **both strategies**

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];
```

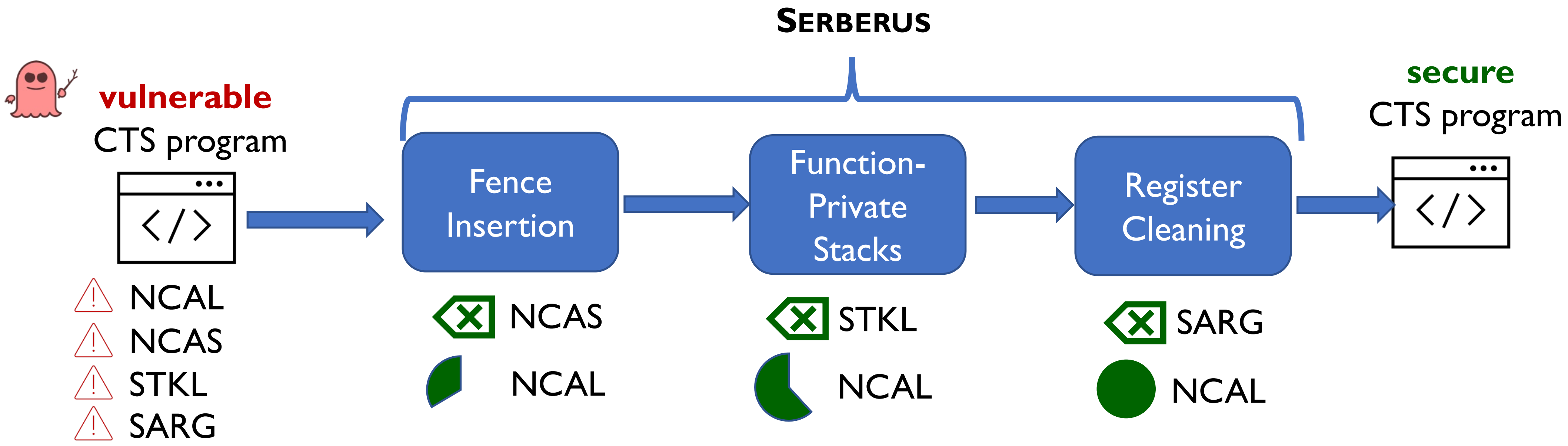
SARG

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

vulnerable CTS program

⚠ NCAL
⚠ NCAS
⚠ STKL
⚠ SARG

Fence Insertion

☒ NCAS
◔ NCAL

Function-Private Stacks

☒ STKL
◕ NCAL

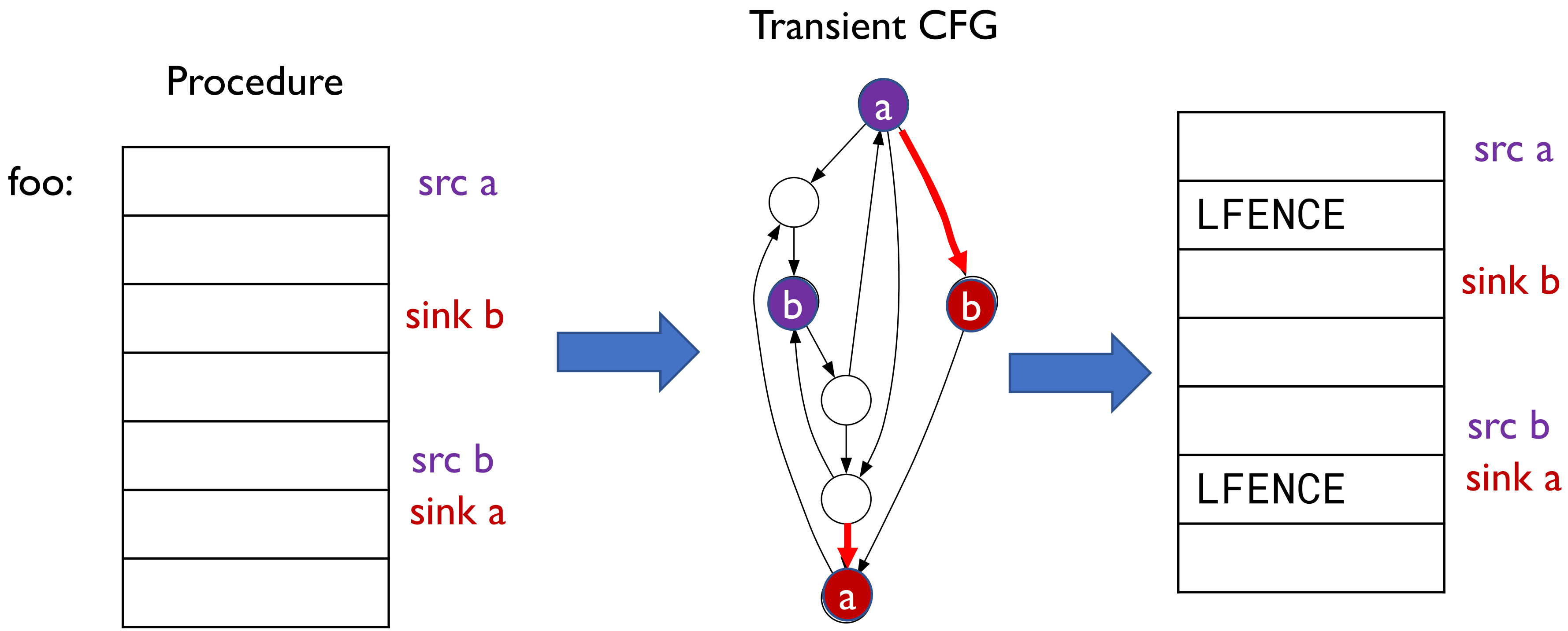
Register Cleaning

☒ SARG
● NCAL

secure CTS program

SERBERUS’ **Fence Insertion** Pass

- Frames 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

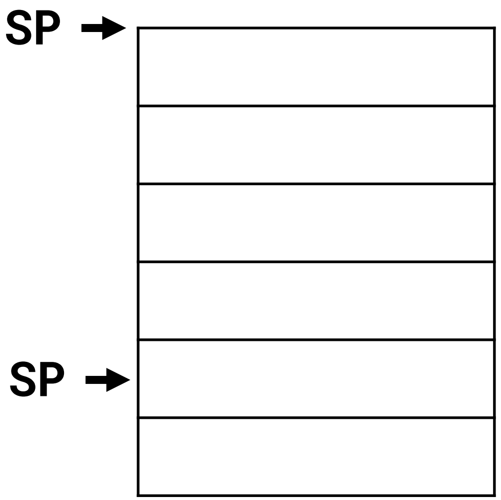
STKL

uninitialized stack load

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

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

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



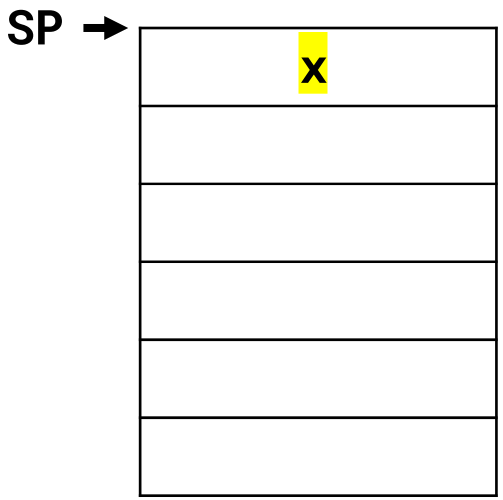
SERBERUS' **Function-Private Stacks** Pass

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
```

prologue

```
  + 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
    ...
```

callsite

```
    CALL r1
  + LD [ZR+PSP_F],SP // load private SP
    ...
```

epilogue

```
  + LD [SP+0],ZR     // probe for underflow
    ADD SP,SP,k      // frame deallocation
  + ST [ZR+PSP_F],SP // store private SP
    RET
```

SERBERUS' **Register Cleaning** Pass

SARG

unexpectedly secret argument

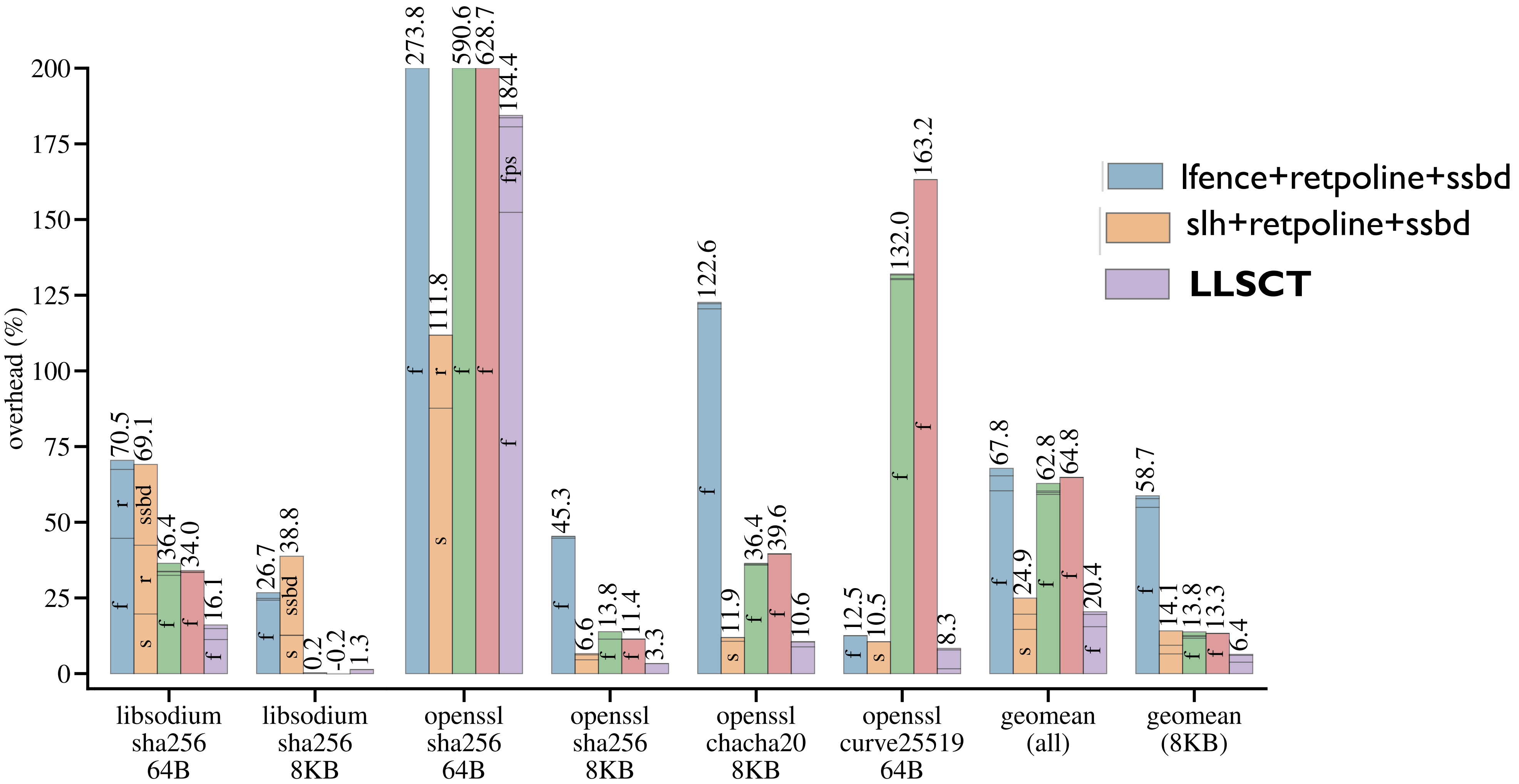
```
foo(int x):
  y = A[x];
```

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*



f = *LFENCE*

r = *retpoline*

slh = *speculative load hardening*

ssbd = *STL disable*

fps = *function-private stacks*

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