Fuzzing Away Speculative Execution Attacks

Mark Silberstein

Joint work with
Oleksii Oleksenko and Christof Fezer
To appear in USENIX Security 2020

Big Picture in One Slide

Problem:

Speculative attacks cannot be mitigated in hardware

Research question:

How to validate that a program is not vulnerable?

Challenge:

Modern runtime verification tools are helpless

New concept:

Simulate mis-speculation in software at runtime

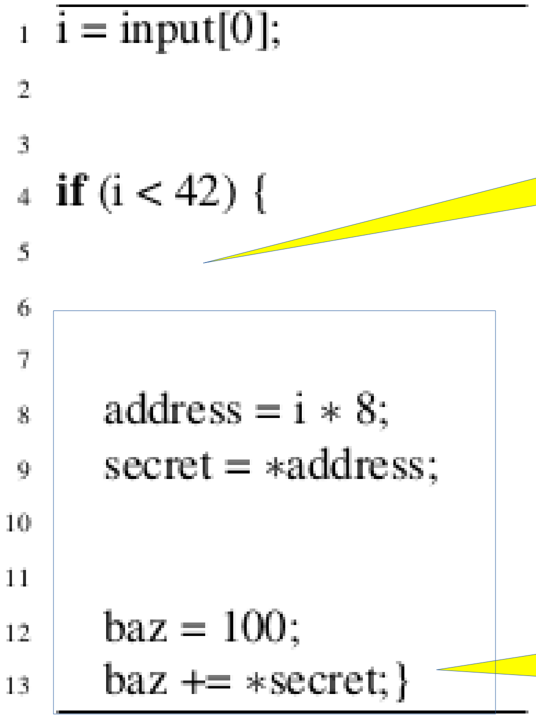
Practical implications:

Faster mitigation, new vulnerabilities found

Today

- Background
- Problem: overheads of Spectre V1 defenses
- Speculation exposure
- SpecFuzz
- Ample opportunities for future research

Spectre V1 requires **software** mitigation



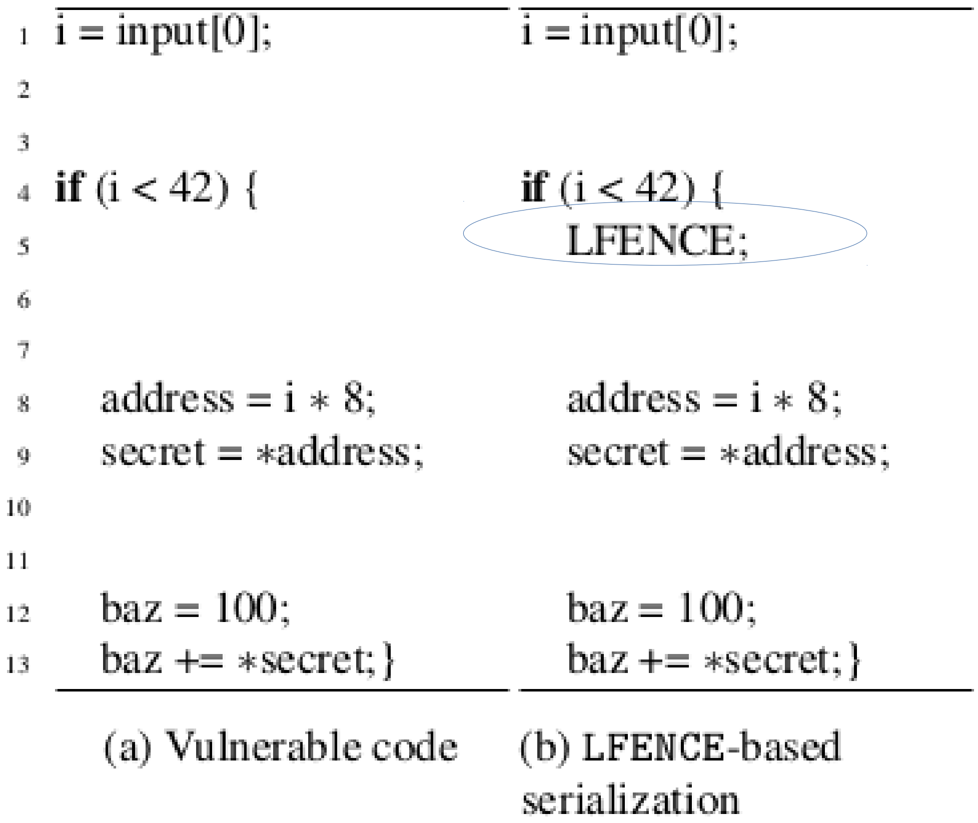
```
i = input[0];


if (i < 42) {



  address = i * 8;
  secret = *address;


  baz = 100;
  baz += *secret;}
```

(a) Vulnerable code

Speculation occurs
here due to branch misprediction!

Access to process address space
is **architecturally legal**
but
violates program semantics

Simple solution: stop speculation in all conditional branches



```
i = input[0];


if (i < 42) {




    address = i ∗ 8;
    secret = ∗address;


    baz = 100;
    baz += ∗secret;}
```

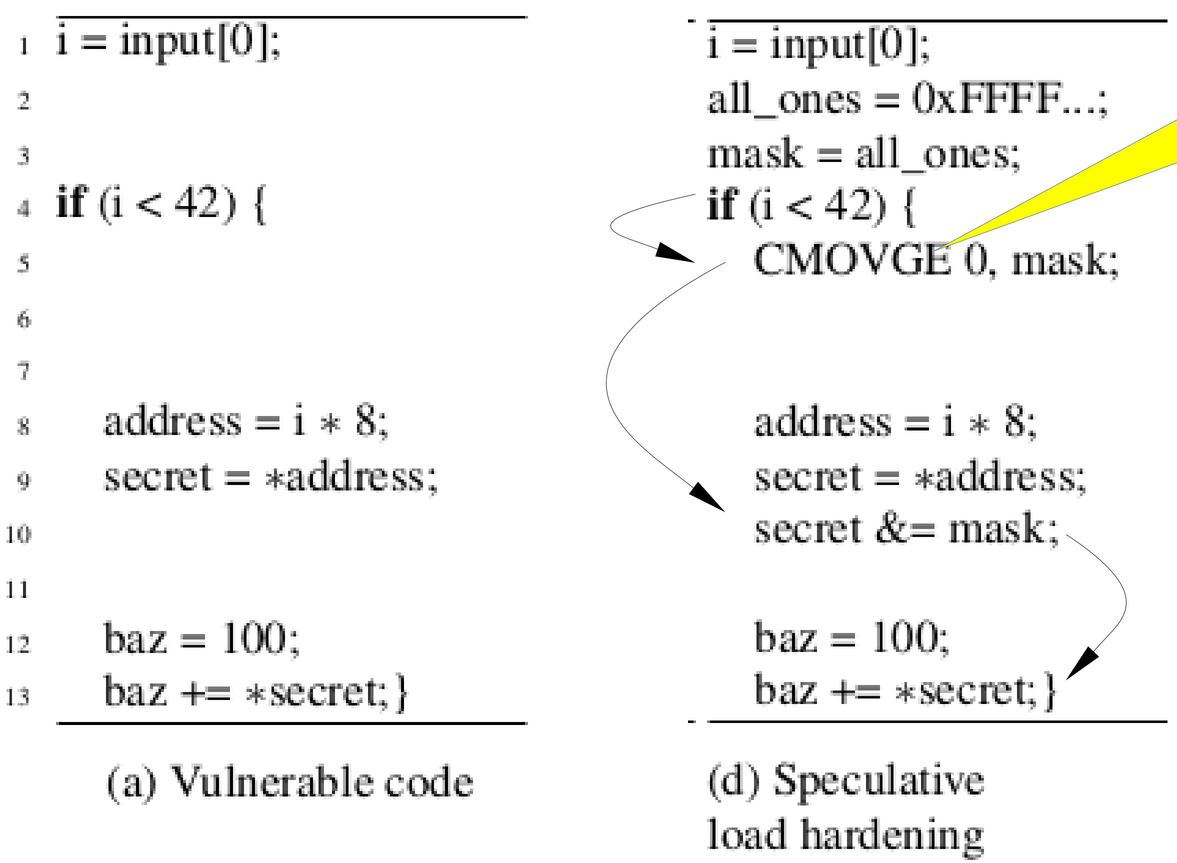
(a) Vulnerable code

```
i = input[0];


if (i < 42) {
    LFENCE;


    address = i ∗ 8;
    secret = ∗address;


    baz = 100;
    baz += ∗secret;}
```

(b) LFENCE-based serialization

Problems?

A (possibly) better idea: destroy values in the speculative path (since LLVM 8.0)



```
i = input[0];


if (i < 42) {



   address = i * 8;
   secret = *address;


   baz = 100;
   baz += *secret;}
```

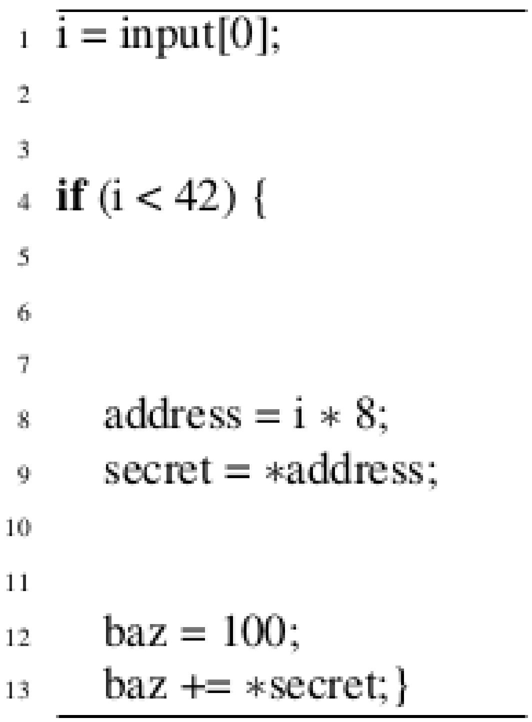
(a) Vulnerable code

```
i = input[0];
all_ones = 0xFFFF...;
mask = all_ones;
if (i < 42) {
   CMOVGE 0, mask;


   address = i * 8;
   secret = *address;
   secret &= mask;

   baz = 100;
   baz += *secret;}
```

(d) Speculative load hardening

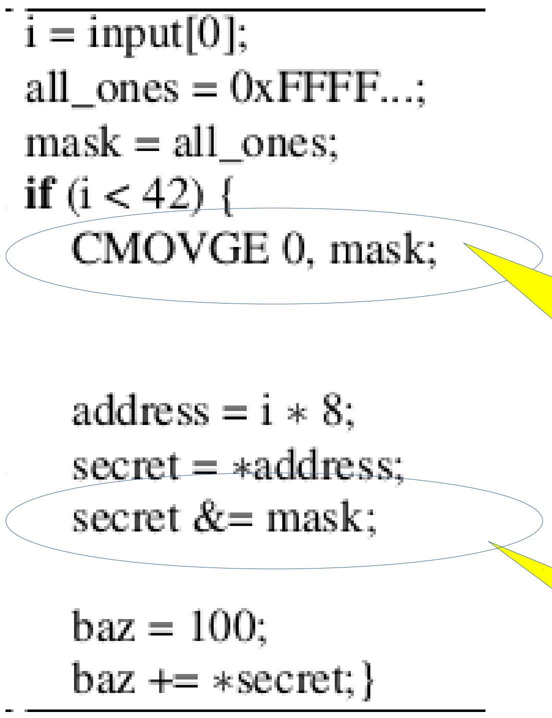
Data dependency on condition evaluation

A (possibly) better idea: destroy values in the speculative path (since LLVM 8.0)



```
i = input[0];


if (i < 42) {



    address = i * 8;
    secret = *address;


    baz = 100;
    baz += *secret;}
```

(a) Vulnerable code



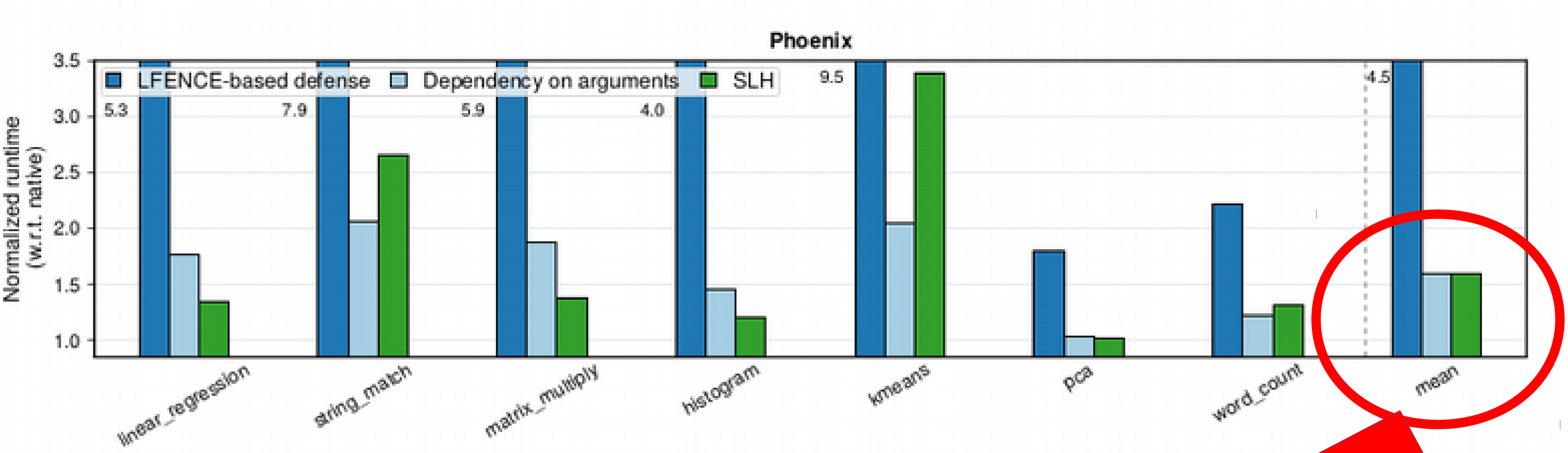
```
i = input[0];
all_ones = 0xFFFF...;
mask = all_ones;
if (i < 42) {
    CMOVGE 0, mask;


    address = i * 8;
    secret = *address;
    secret &= mask;

    baz = 100;
    baz += *secret;}
```

(d) Speculative load hardening

only in the speculative path mask=0

only in the speculative path AND-ed with 0

Performance loss due to mitigation



60% on average!

Why do we instrument all branches?

- Static analysis is inefficient:
  - MS Visual Studio missed 12 out of 13 tests engineered to evade detection
- A single vulnerability leaves the whole memory exposed

Can we elide instrumentation without compromising security?

How can we know that the branch is secure?

Fuzzing: background

- Finds security and correctness bugs
- **Fuzzing drivers** invoke with many (random) inputs
- **Coverage:** explore (as many as possible) branches
- Combined with **buffer overflow checkers** to catch bugs

Why can't we use fuzzing to catch Spectre vulnerabilities?

- Mis-speculation results are architecturally invisible by design!
- The architectural state remains unchanged
- Invalid accesses are “silenced”

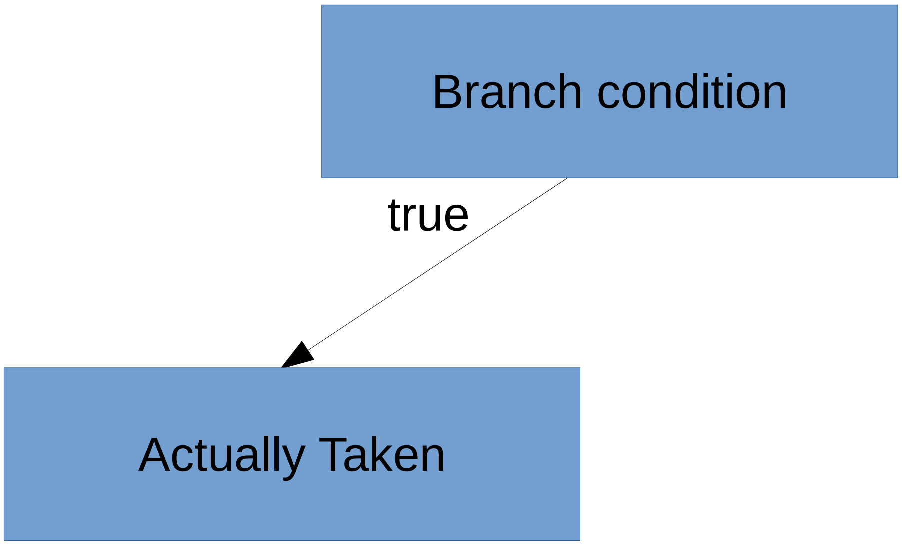
Why can't we use fuzzing to catch Spectre vulnerabilities?

- Mis-speculation results are architecturally invisible by design!
- The architectural state remains unchanged
- Invalid accesses are “silenced”

How can we make Spectre vulnerabilities visible for fuzzers?

Idea: *Speculation Exposure (SE)*

- **Simulate** mis-speculation and run it as part of the execution



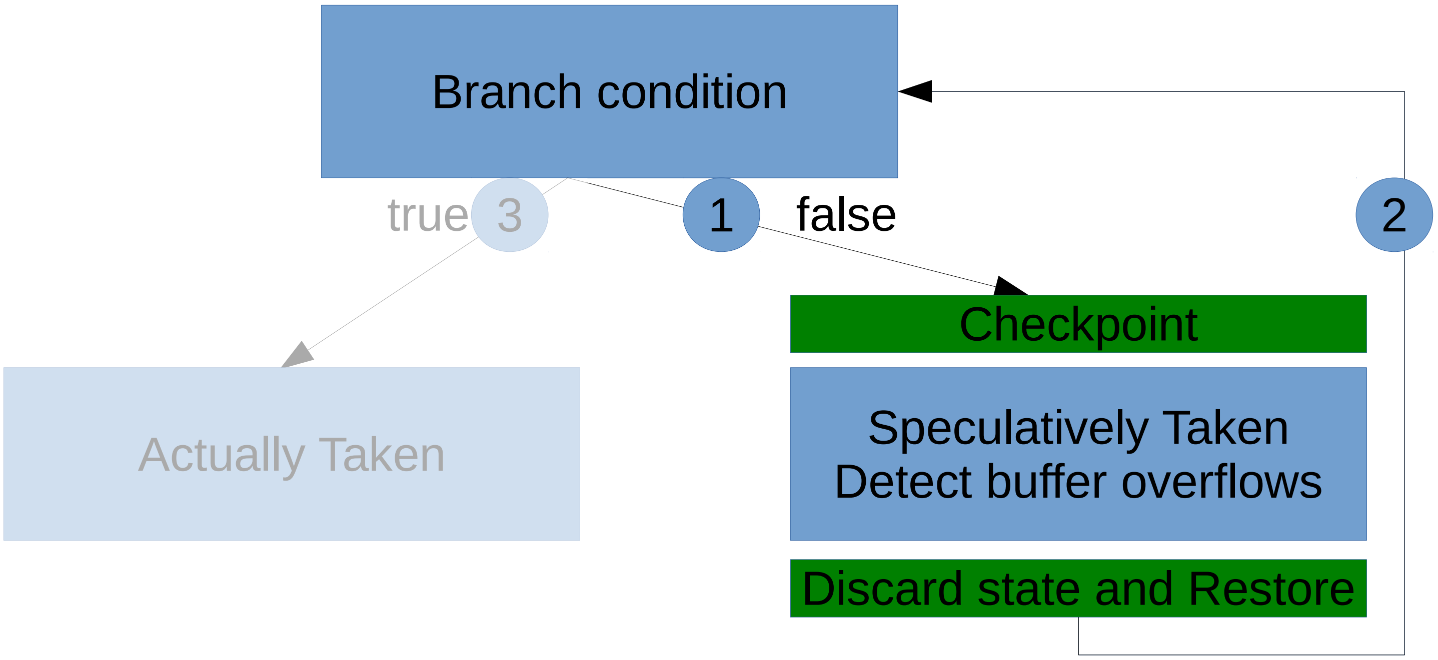
Branch condition

true

Actually Taken

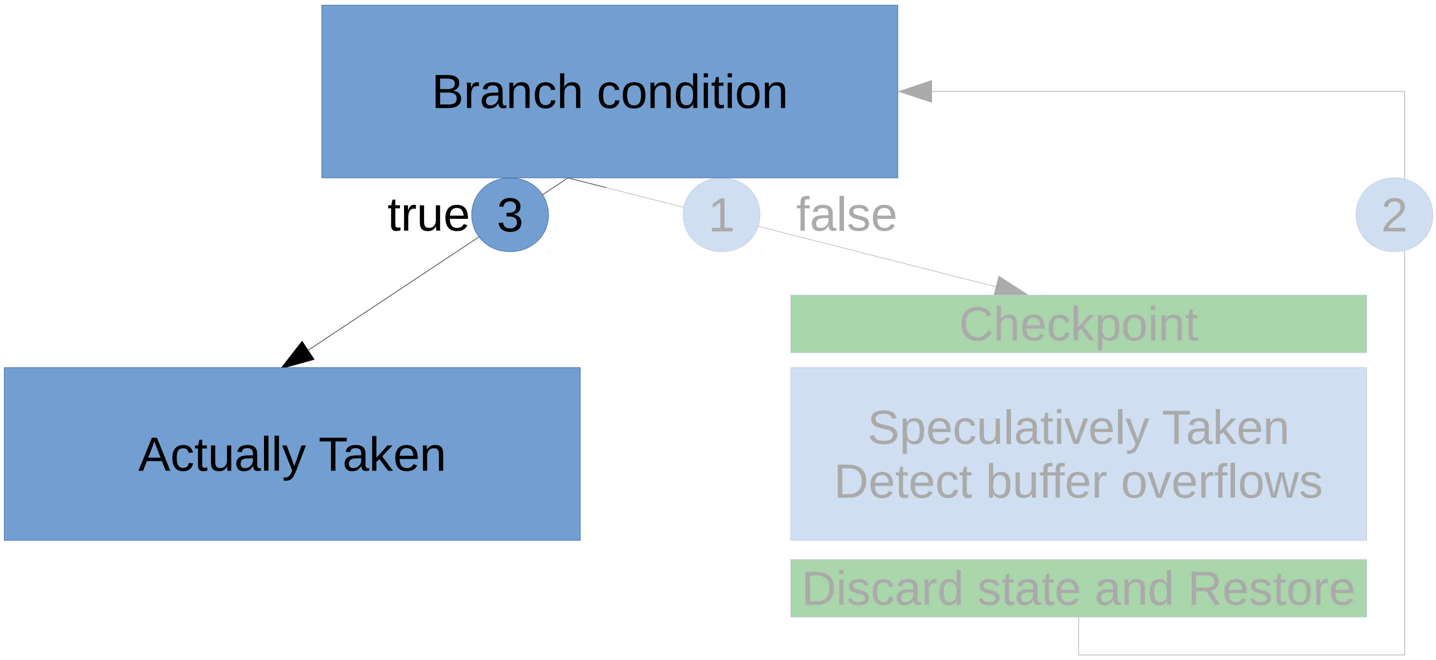
Idea: *Speculation Exposure (SE)*

- **Simulate** mis-speculation and run it as part of the execution



Idea: *Speculation Exposure (SE)*

- **Simulate** mis-speculation and run it as part of the execution



SE: how it works

- Instrument each branch with:
  - Check-point
  - Forced (simulation) execution of a mispredicted path
  - Detection/logging of vulnerabilities
  - Termination of the simulation (worst case – ROB size)
  - Restart of the normal path

How do we know a branch is secure?

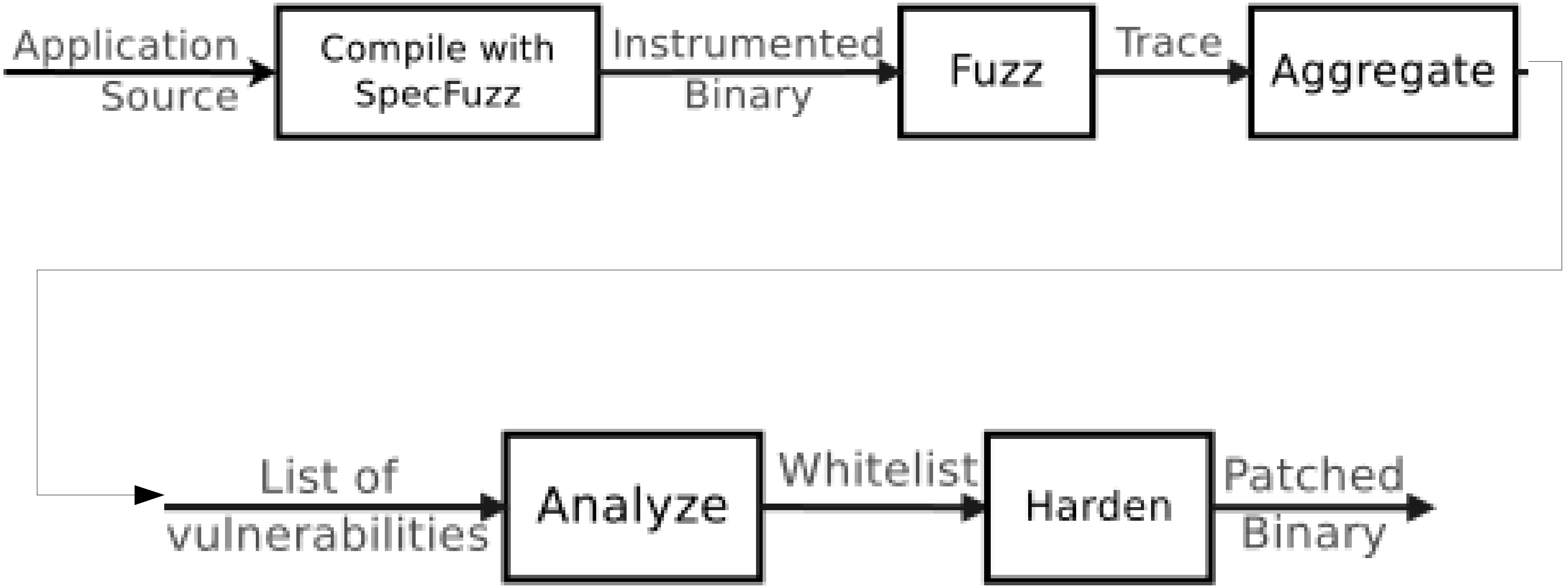
- We do not know for sure... But with high probability
- Apply fuzzing with SE
- Classify buffer overflows occurring in SE
  - Benign (input-independent)
  - Potential vulnerabilities (input-dependent)

How do we know a branch is secure?

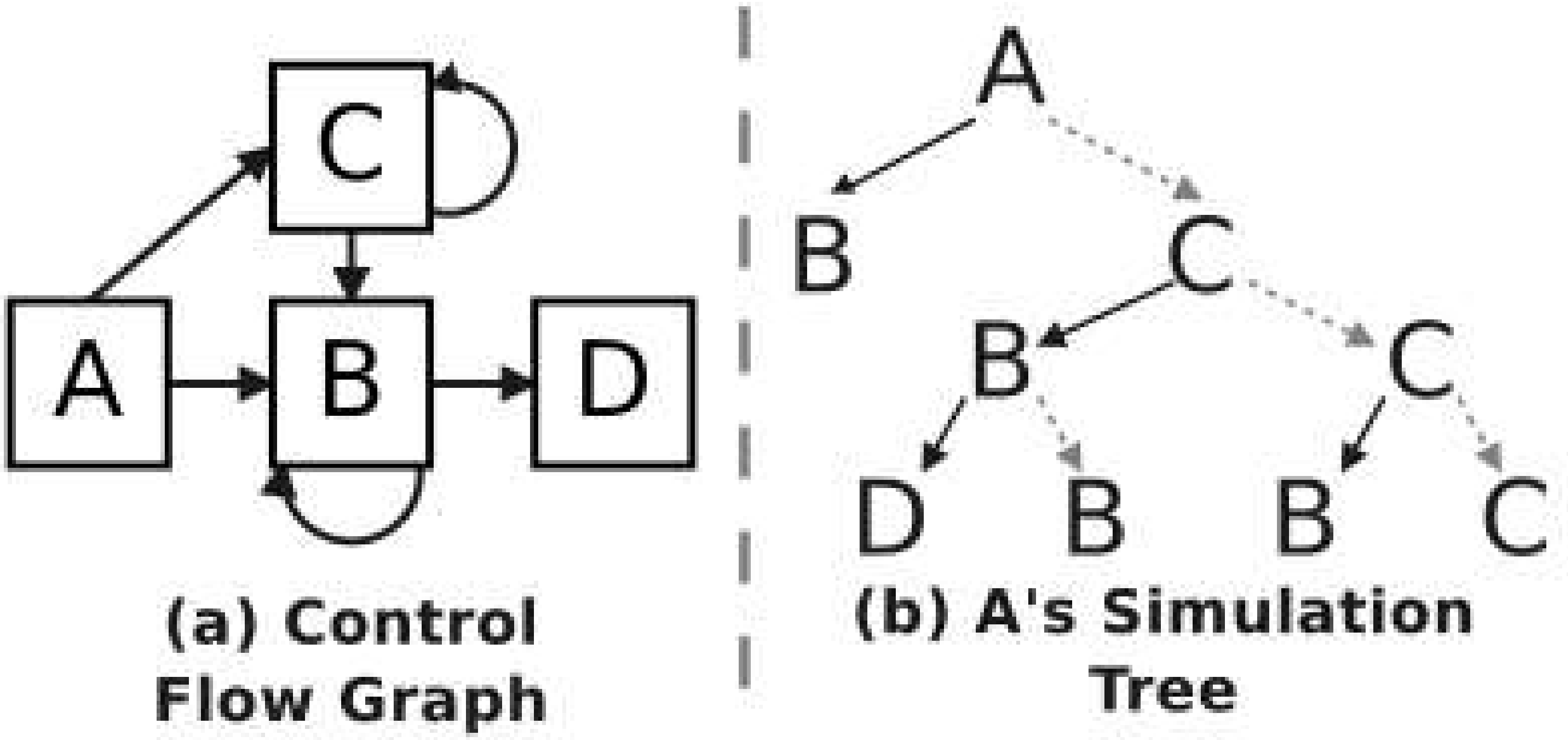
- We do not know for sure... But with high probability
- Apply fuzzing with SE
- Classify buffer overflows occurring in SE
  - Benign (input-independent)
  - Potential vulnerabilities (input-dependent)

We remove serialization instructions in branches with benign overflow

Putting it all together



Nested speculation

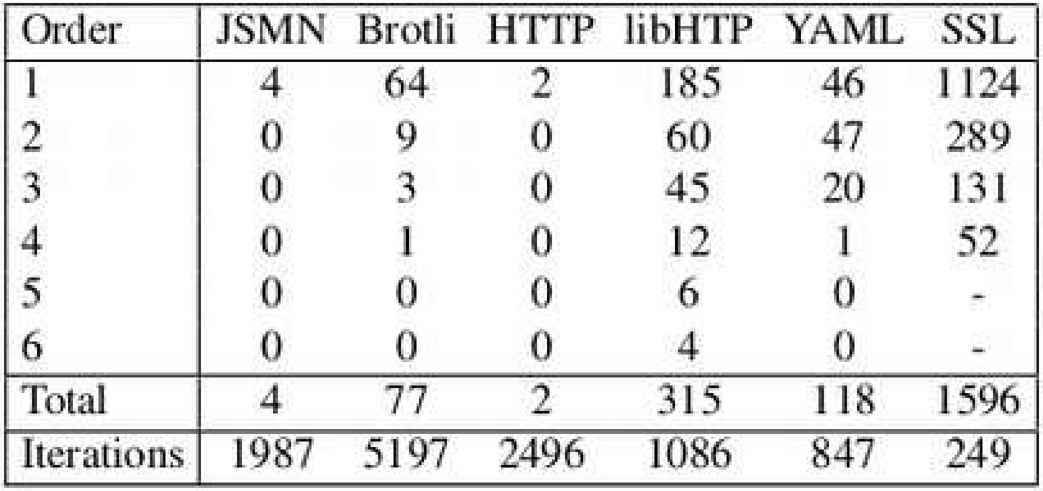


(a) Control Flow Graph

(b) A's Simulation Tree

Nested speculation: exhaustive is too slow

- It is necessary!



Order	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1	4	64	2	185	46	1124
2	0	9	0	60	47	289
3	0	3	0	45	20	131
4	0	1	0	12	1	52
5	0	0	0	6	0	-
6	0	0	0	4	0	-
Total	4	77	2	315	118	1596
Iterations	1987	5197	2496	1086	847	249

- Exponential number of branches to be simulated
- Fuzzing becomes too slow – coverage is affected

Prioritized nested fuzzing

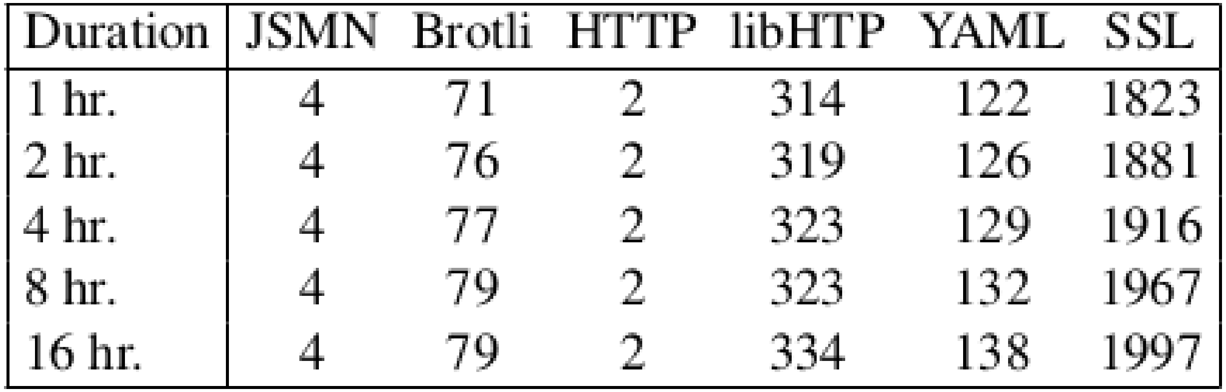
- Deeper nesting levels are tested with exponentially smaller number of fuzzing inputs
- For a given branch
  - Nested level 1: each input
  - Nested level 2: every 2nd input
  - Nested level 3: every 4th input
  - Nested level log(n)+1: every nth input

External calls/callbacks

- Non-instrumented code cannot be checked
- If a function is instrumented – the simulation continues
- Otherwise – considered a serialization point
- Instrumented callbacks from non-instrumented functions are not supported

Results

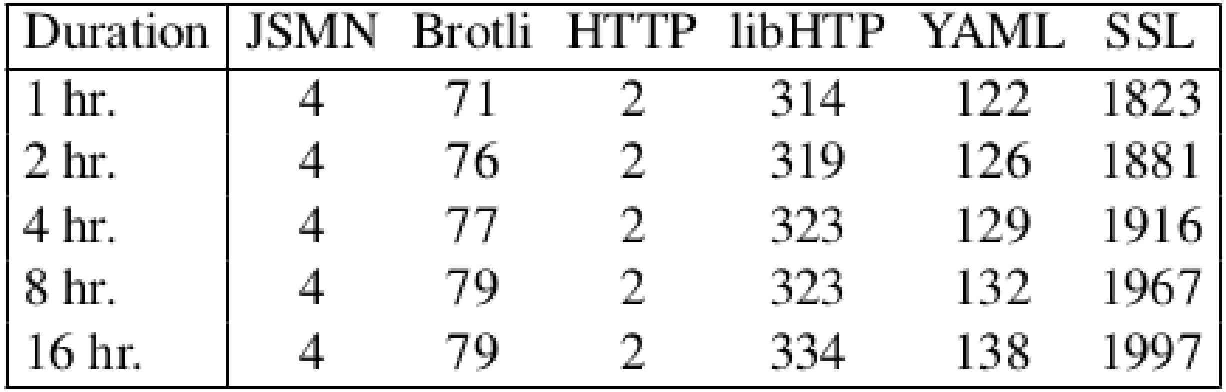
- Total potential vulnerabilities



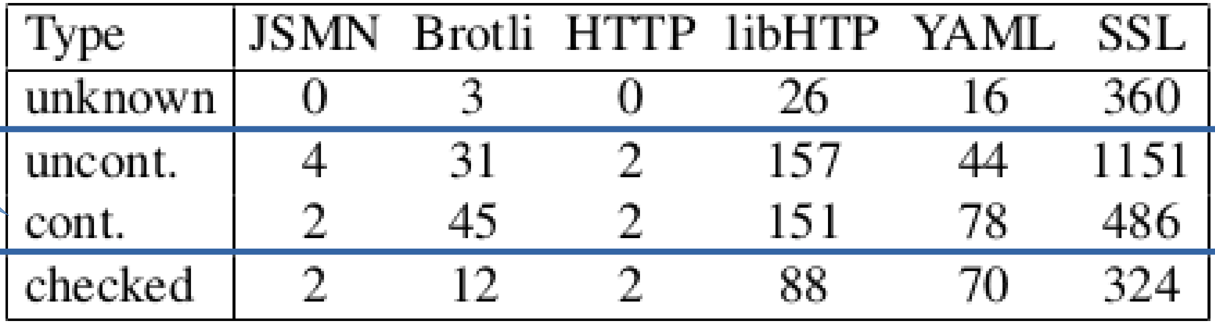
Duration	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1 hr.	4	71	2	314	122	1823
2 hr.	4	76	2	319	126	1881
4 hr.	4	77	2	323	129	1916
8 hr.	4	79	2	323	132	1967
16 hr.	4	79	2	334	138	1997

Results

- Total potential vulnerabilities



Duration	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1 hr.	4	71	2	314	122	1823
2 hr.	4	76	2	319	126	1881
4 hr.	4	77	2	323	129	1916
8 hr.	4	79	2	323	132	1967
16 hr.	4	79	2	334	138	1997

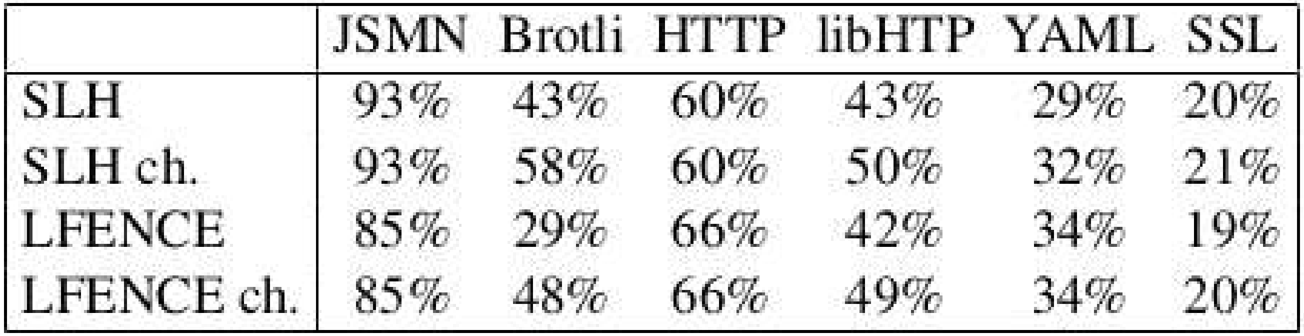


Type	JSMN	Brotli	HTTP	libHTP	YAML	SSL
unknown	0	3	0	26	16	360
uncont.	4	31	2	157	44	1151
cont.	2	45	2	151	78	486
checked	2	12	2	88	70	324

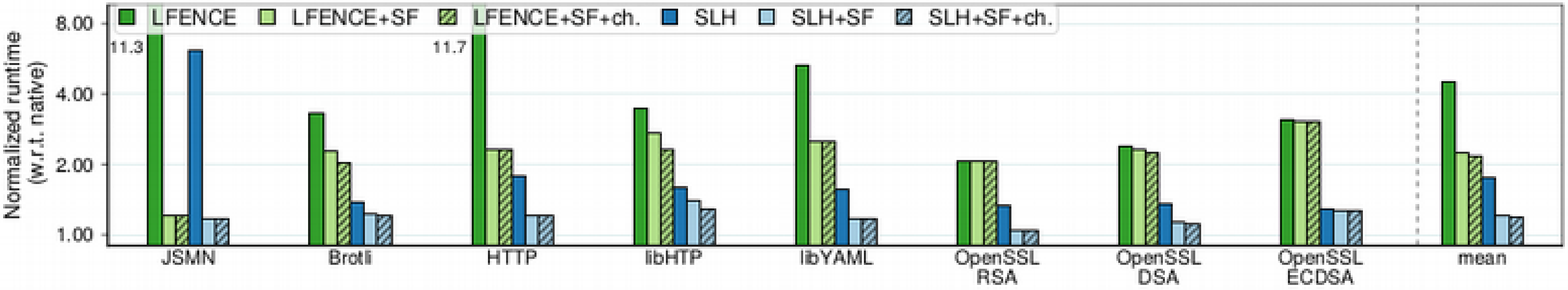
Definitely vulnerable!

over 55% elided

Performance improvements



	JSMN	Brotli	HTTP	libHTP	YAML	SSL
SLH	93%	43%	60%	43%	29%	20%
SLH ch.	93%	58%	60%	50%	32%	21%
LFENCE	85%	29%	66%	42%	34%	19%
LFENCE ch.	85%	48%	66%	49%	34%	20%



Future work

- Other type of Spectre attacks?
  - Removing V2 mitigations will improve OS performance!
- How to get rid of the source code requirement?
  - Important for third-party libraries
- Can we ***prove*** that the branch is benign?
- Can we provide a good metric of coverage?
- What is the right way to specify speculation simulation?
- ...