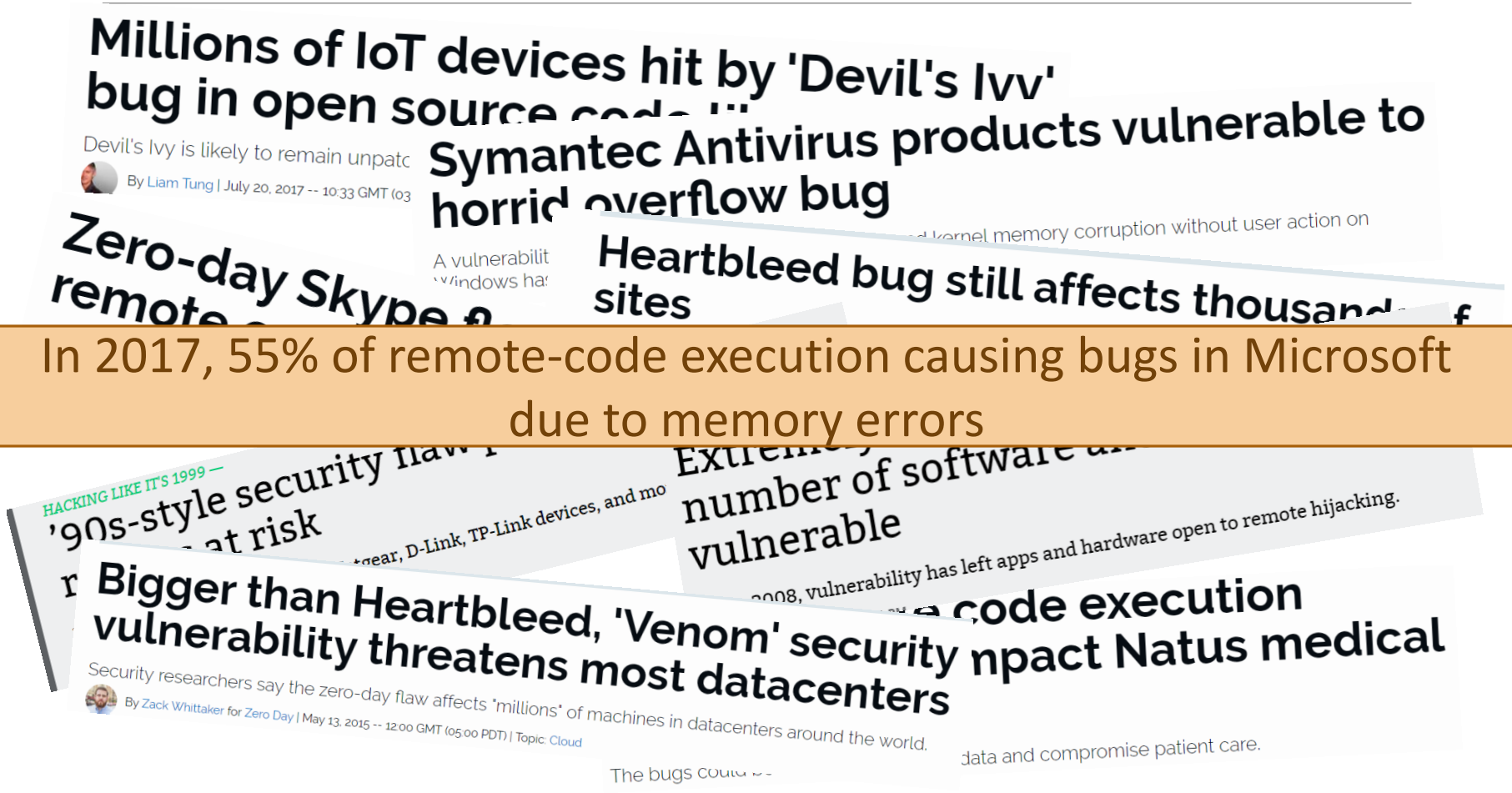
Practical Memory Safety with *REST*

KANAD SINHA & SIMHA SETHUMADHAVAN

COLUMBIA UNIVERSITY

Is memory safety relevant?



In 2017, 55% of remote-code execution causing bugs in Microsoft due to memory errors

Is memory safety relevant?

Yes!

Practical memory safety

Presenting…

Random Embedded Security Tokens or *REST*

Core H/W primitive: Insert known 64B random value (*token*) in program and detect accesses to them.

Practical memory safety

Presenting…

Random Embedded Security Tokens or *REST*

Core H/W primitive: Insert known 64B random value (*token*) in program and detect accesses to them.

```
char *buf = malloc(BUF_LEN);

for (i=0; i<out_of_bounds; i++)
  buf = 0;
```

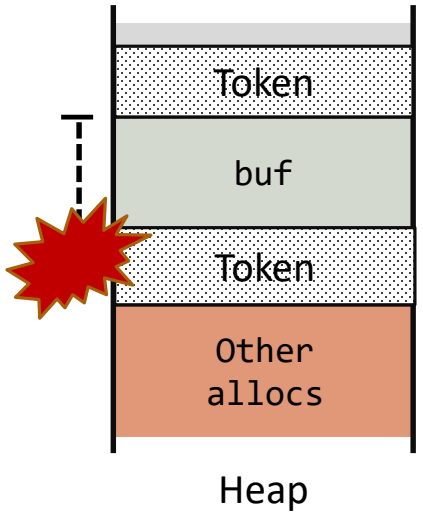
Practical memory safety

Presenting…

Random Embedded Security Tokens or *REST*

Core H/W primitive: Insert known 64B random value (*token*) in program and detect accesses to them.

```
char *buf = malloc(BUF_LEN);

for (i=0; i<out_of_bounds; i++)
  buf = 0;
```



Practical memory safety

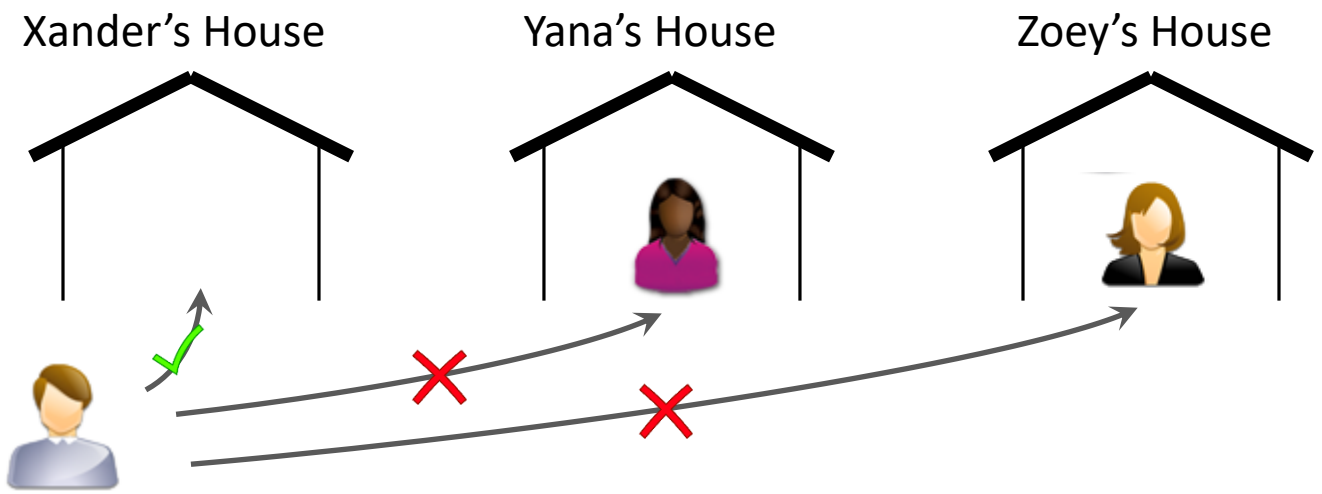
Presenting…

Random Embedded Security Tokens or *REST*

Core H/W primitive: Insert known 64B random value (*token*) in program and detect accesses to them.

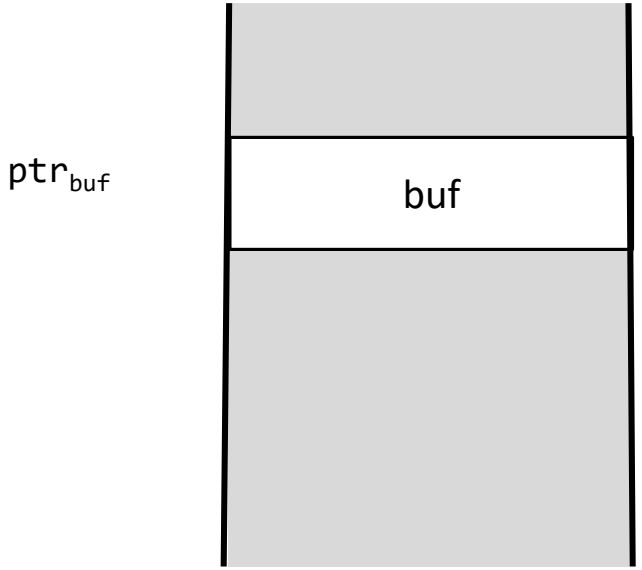
- Trivial hardware implementation
- Software framework based on *AddressSanitizer*
- Provides heap safety for legacy binaries

Background: Spatial Memory Safety



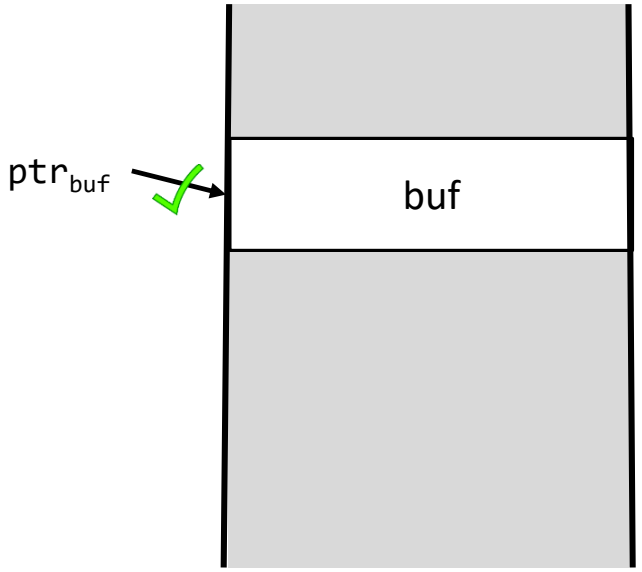
Background: Spatial Memory Safety

```
char *ptr_buf = malloc(BUF_LEN);
…
ptr_buf[in_bounds] = X;
…
ptr_buf[out_of_bounds] = Y;
```



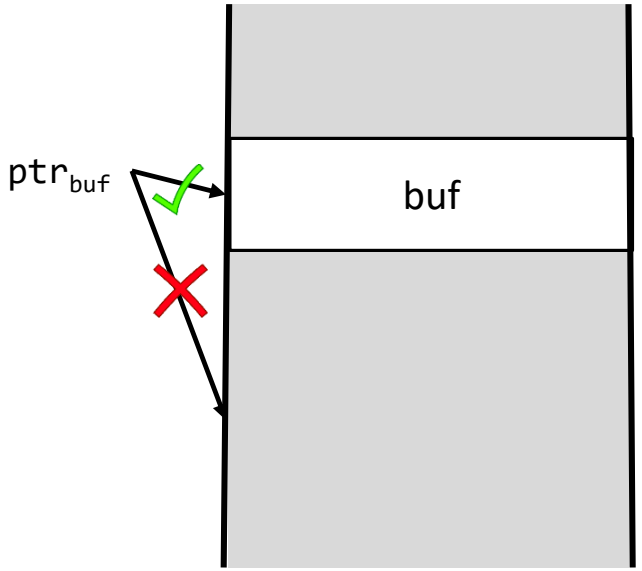
Background: Spatial Memory Safety

```
char *ptr_buf = malloc(BUF_LEN);
…
ptr_buf[in_bounds] = X;
…
ptr_buf[out_of_bounds] = Y;
```



Background: Spatial Memory Safety

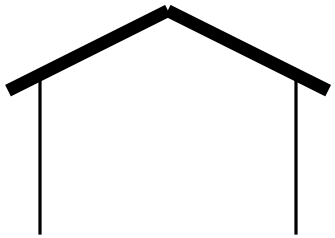
```
char *ptr_buf = malloc(BUF_LEN);
…
ptr_buf[in_bounds] = X;
…
ptr_buf[out_of_bounds] = Y;
```



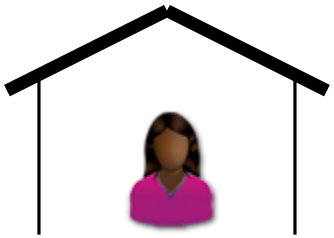
Background: Temporal Memory Safety

Xander moves out, Will moves in

Xander's House



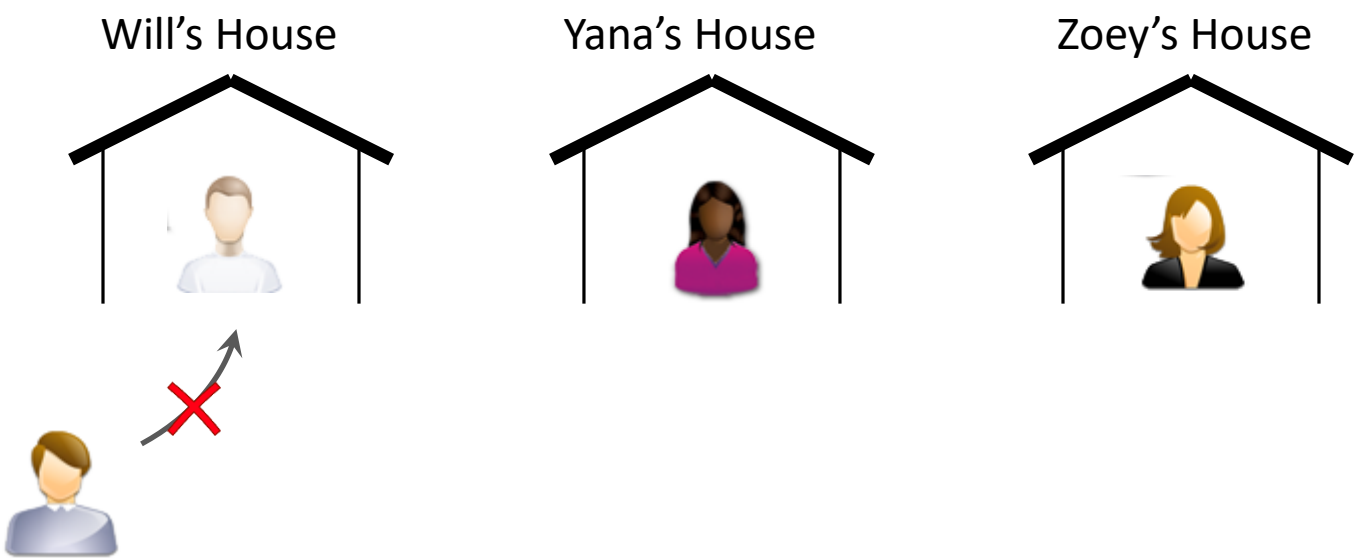
Yana's House



Zoey's House

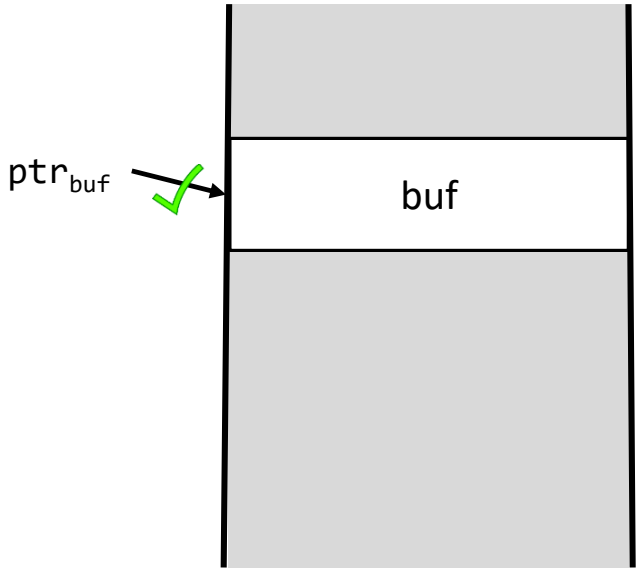
Background: Temporal Memory Safety

Xander moves out, Will moves in



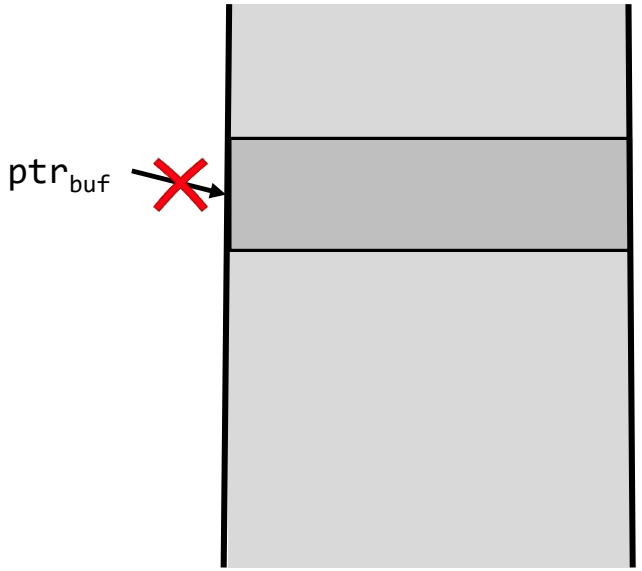
Background: Temporal Memory Safety

```
char *ptr_buf = malloc(BUF_LEN);
ptr_buf[in_bounds] = X;
…
free(ptr_buf);
ptr_buf[in_bounds] = Y;
```



Background: Temporal Memory Safety

```
char *ptr_buf = malloc(BUF_LEN);
ptr_buf[in_bounds] = X;
…
free(ptr_buf);
ptr_buf[in_bounds] = Y;
```



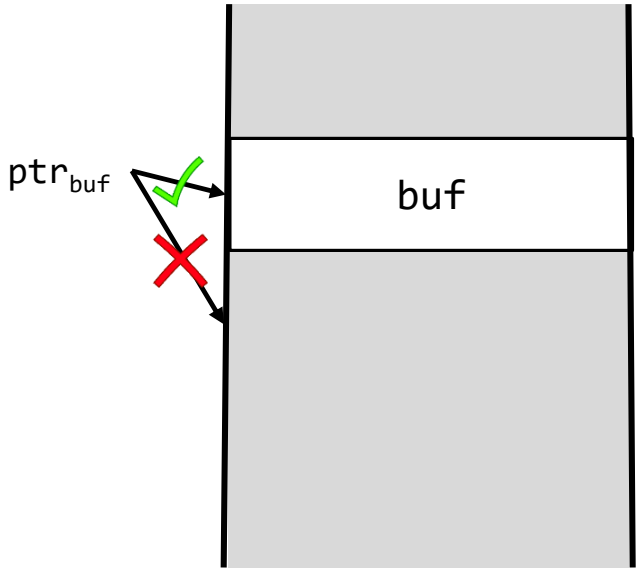
Previous H/W Solutions

Mainly categorizable into 2 types.

Previous H/W Solutions

Mainly categorizable into 2 types.

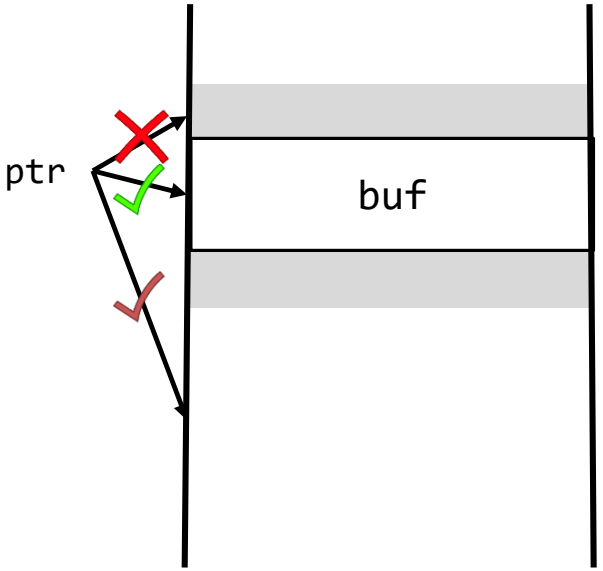
- **Whitelisting:** Pointer based
  - \+ Good coverage
  - \+ Temporal safety (for some)
  - \- Performance overhead
  - \- Implementation overhead
  - \- Imprecise



Previous H/W Solutions

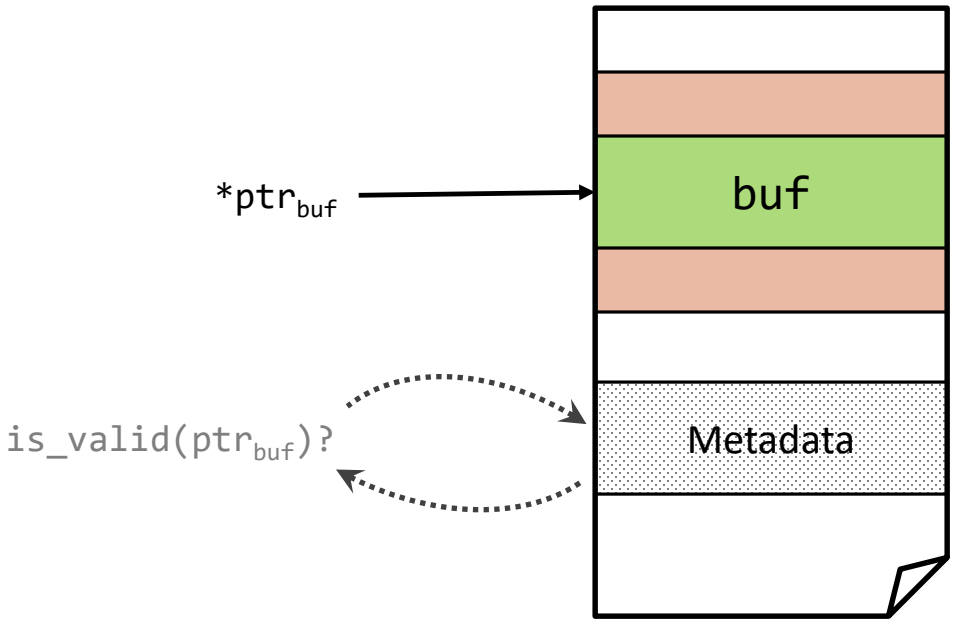
Mainly categorizable into 2 types.

- **Whitelisting:** Pointer based
  - \+ Good coverage
  - \+ Temporal safety (for some)
  - \- Performance overhead
  - \- Implementation overhead
  - \- Imprecise
- **Blacklisting:** Location based
  - \+ Fast
  - \- Weaker coverage (has false negatives)
  - \- Implementation overhead
  - \- No temporal protection



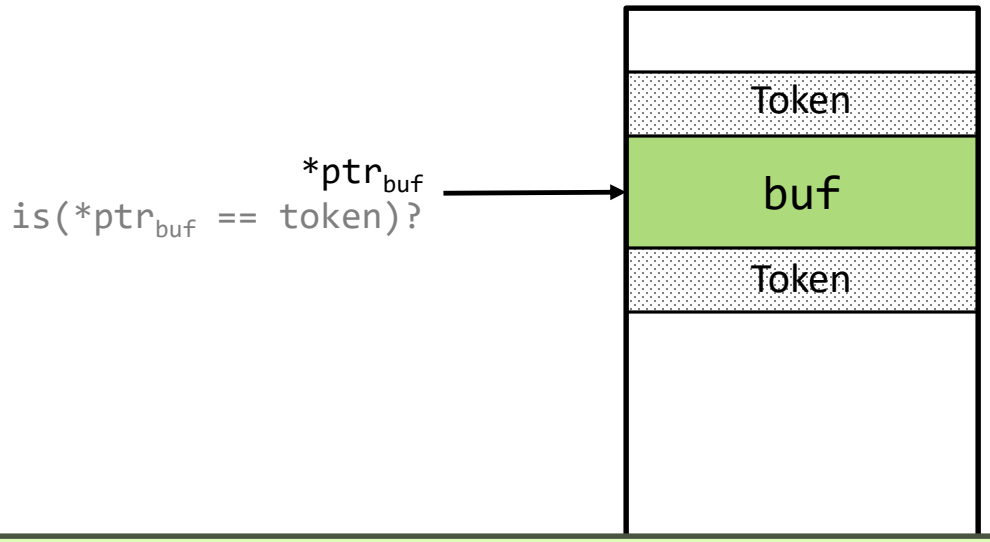
Previous H/W Solutions

Tag-based



REST: Primitive Overview

Content-based blacklisting



REST primitive has trivial complexity, overhead

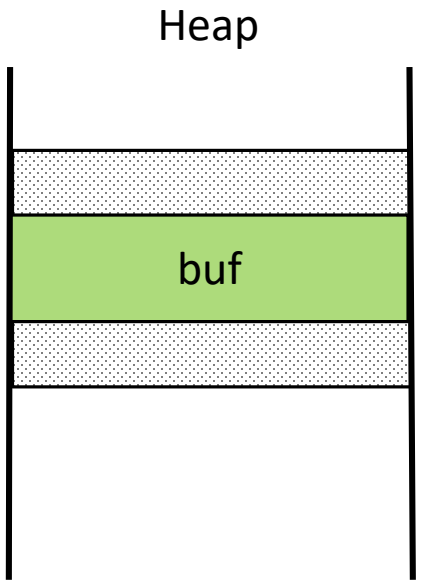
REST: Spatial Memory Safety

REST: Temporal Memory Safety

REST Software

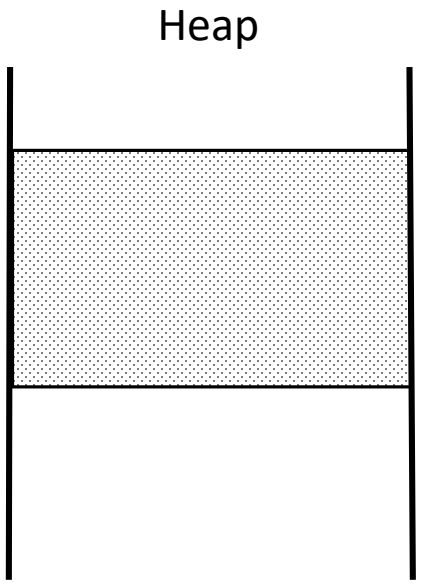
Heap Safety

- Allocate and bookend region, `malloc` to program



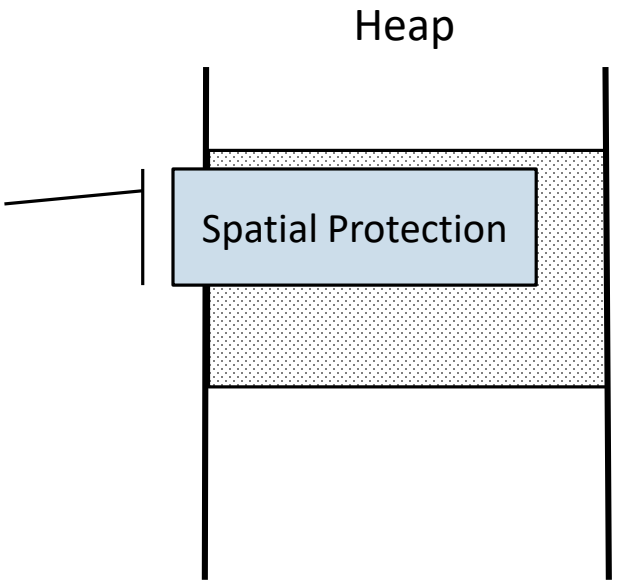
Heap Safety

- Allocate and bookend region, `malloc` to program
- REST'ize at `free`
- Do not reallocate region until heap sufficiently consumed



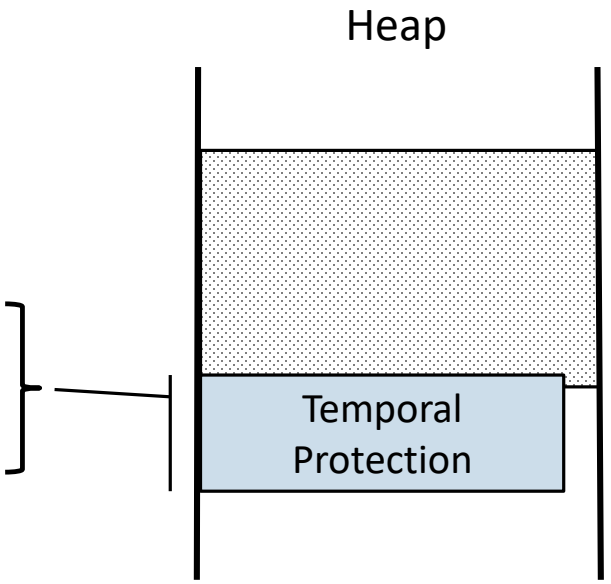
Heap Safety

- Allocate and bookend region, `malloc` to program
- REST'ize at `free`
- Do not reallocate region until heap sufficiently consumed



Heap Safety

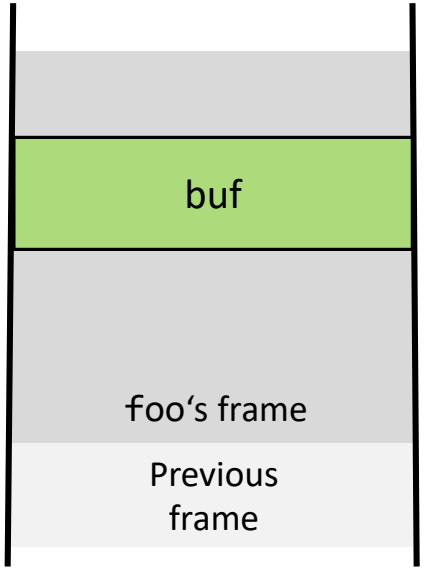
- Allocate and bookend region, `malloc` to program
- REST'ize at `free`
- Do not reallocate region until heap sufficiently consumed



Can be enabled for legacy binaries

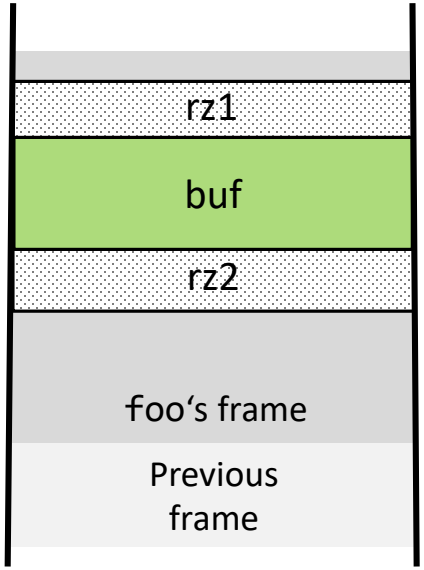
Stack Safety

```
void foo() {
    char buf[64];
    …
    return;
}
```



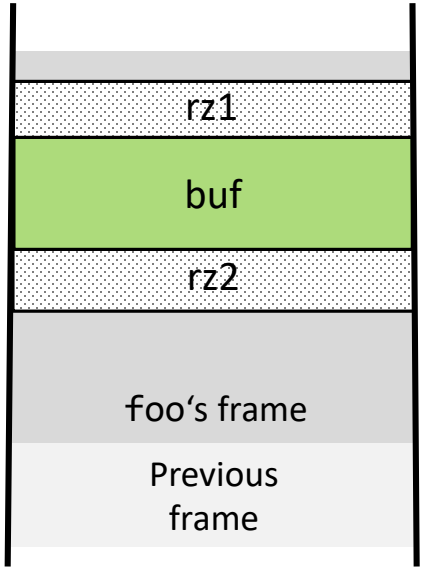
Stack Safety

```
void foo() {
    char rz1[64];
    char buf[64];
    char rz2[64];
    arm(rz1);
    arm(rz2);
    …
    disarm(rz1);
    disarm(rz2);
    return;
}
```



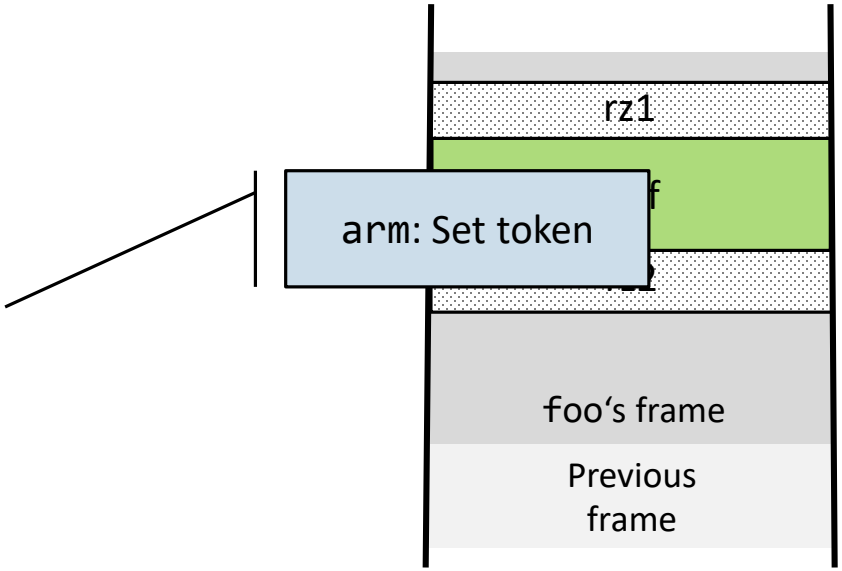
Stack Safety

```
void foo() {
    char rz1[64];
    char buf[64];
    char rz2[64];
    arm(rz1);
    arm(rz2);
    …
    disarm(rz1);
    disarm(rz2);
    return;
}
```



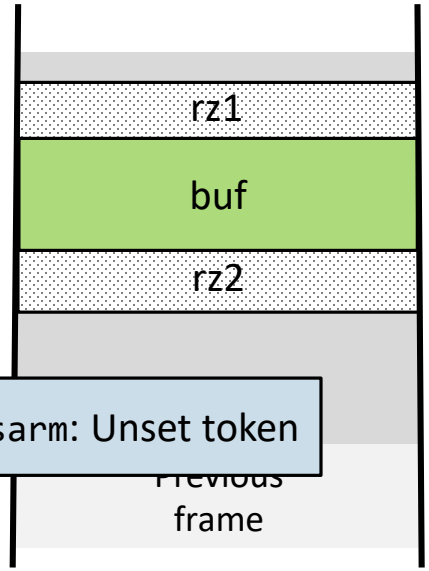
Stack Safety

```
void foo() {
    char rz1[64];
    char buf[64];
    char rz2[64];

    arm rz1;
    arm rz2;

    disarm(rz1);
    disarm(rz2);
    return;
}
```



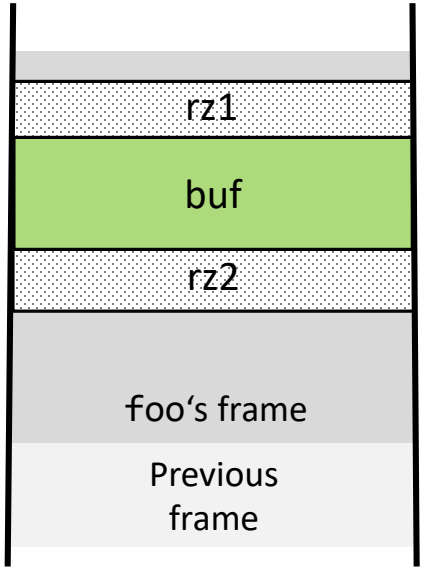
Stack Safety

```
void foo() {
    char rz1[64];
    char buf[64];
    char rz2[64];
    arm(rz1);
    arm(rz2);

    disarm rz1;
    disarm rz2;

    return;
}
```



Stack Safety

```
void foo() {
    char rz1[64];
    char buf[64];
    char rz2[64];
    arm(rz1);
    arm(rz2);
    …
    disarm(rz1);
    disarm(rz2);
    return;
}
```

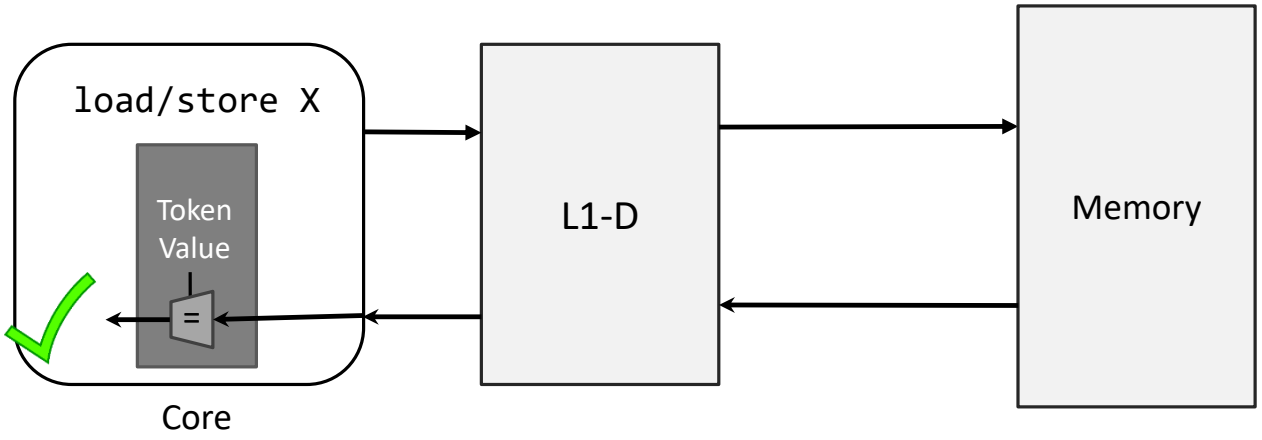


Requires recompilation with *REST* plugin

REST Hardware

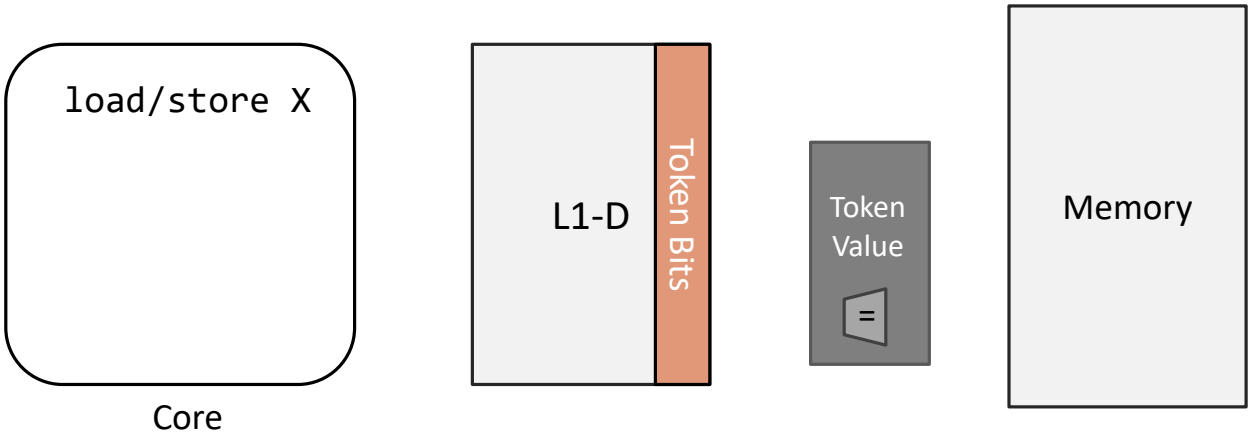
Naïve Design

Every store involves an extra load → Complicated and expensive

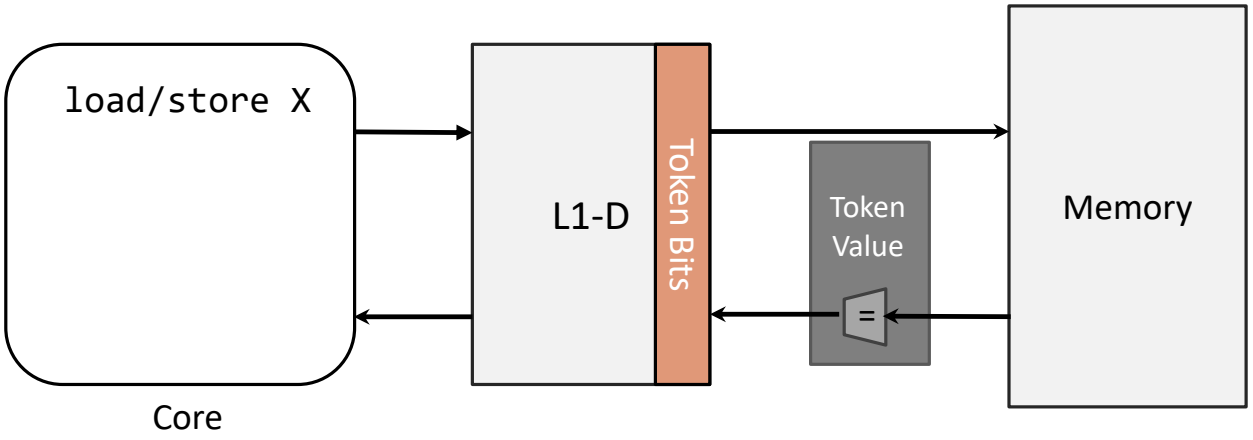


Cache Modifications

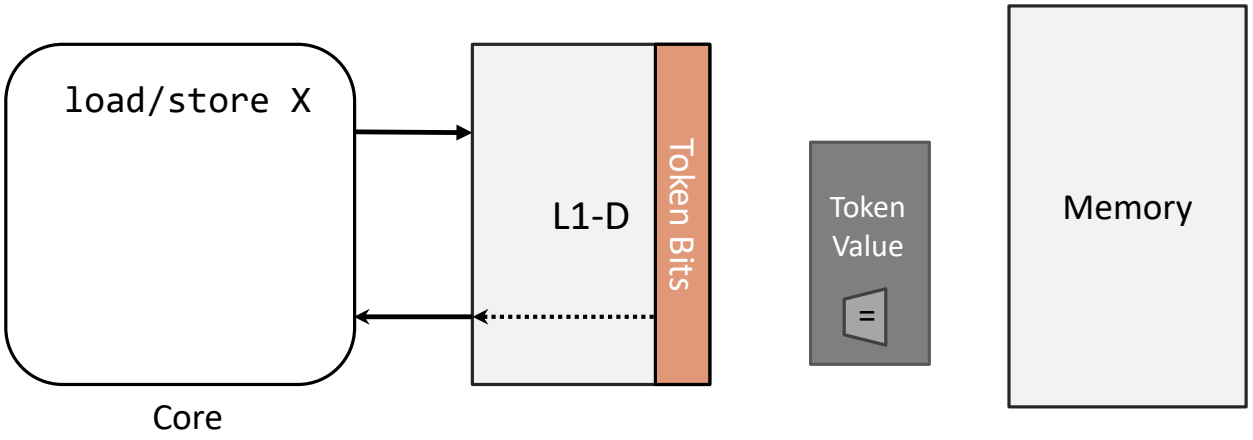
Comparator at L1-D mem interface + 1b per L1-D line



Cache Miss

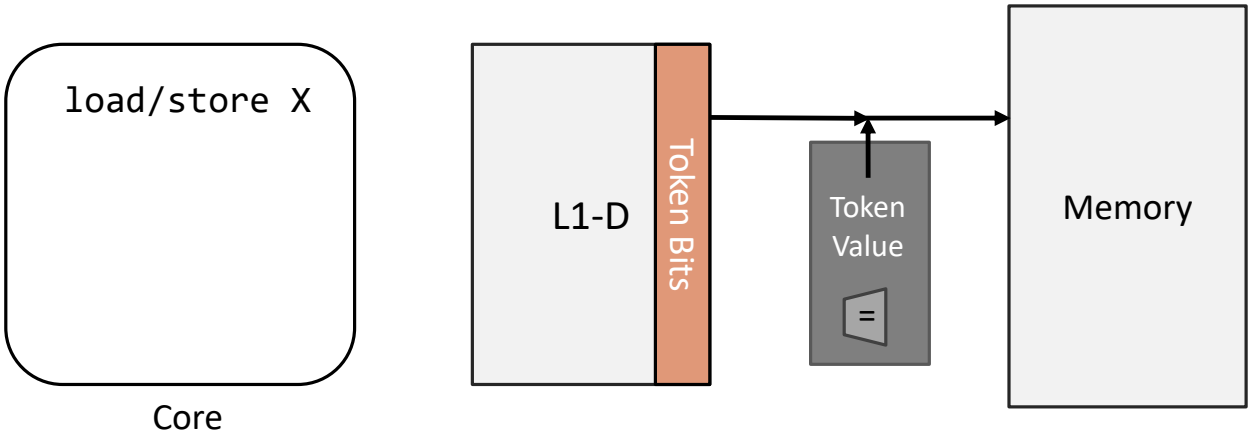


Cache Hit



Cache Eviction

Armed outgoing line filled with token value



What about the core?

TODO: Have to support arms and disarms

- 512b writes
- Special semantics: can only touch token with `disarm`

LSQ design concerns:

- Forwarding would break semantics
- 512b data entries
- How to match unaligned token access?

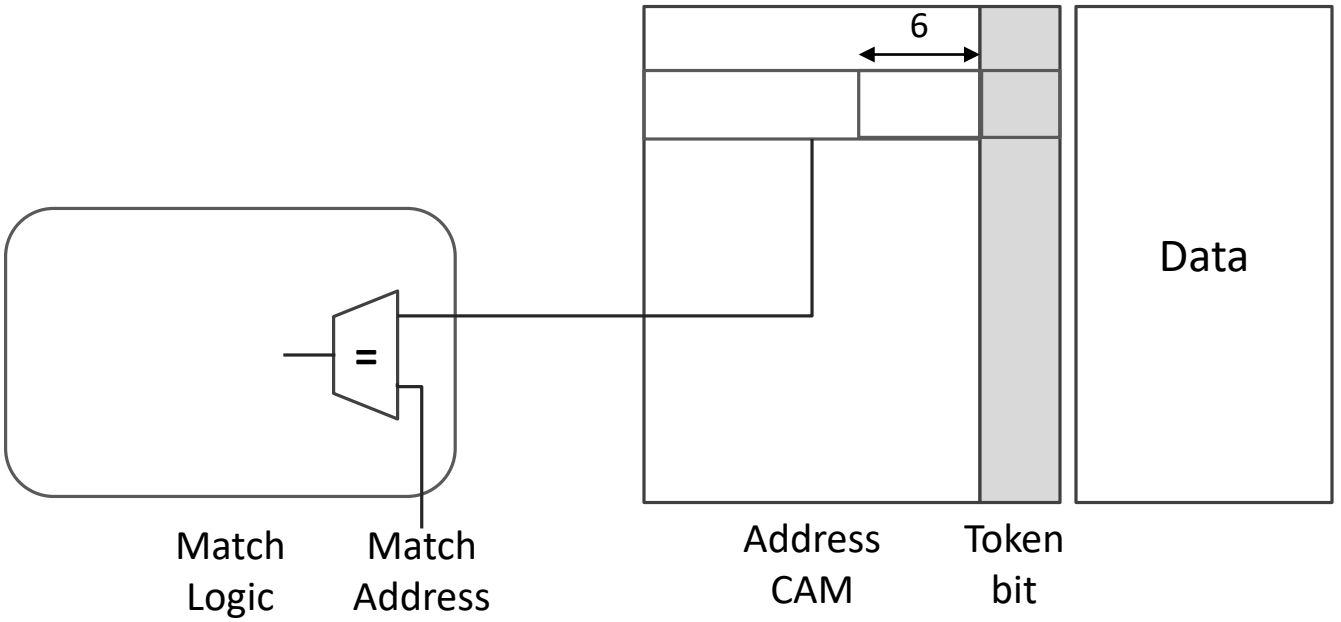
Load-Store Queue

- Forwarding breaks semantics
- 512b data entries
- Detecting unaligned token access

Add 1b tag

Only update token bit

Split regular match logic



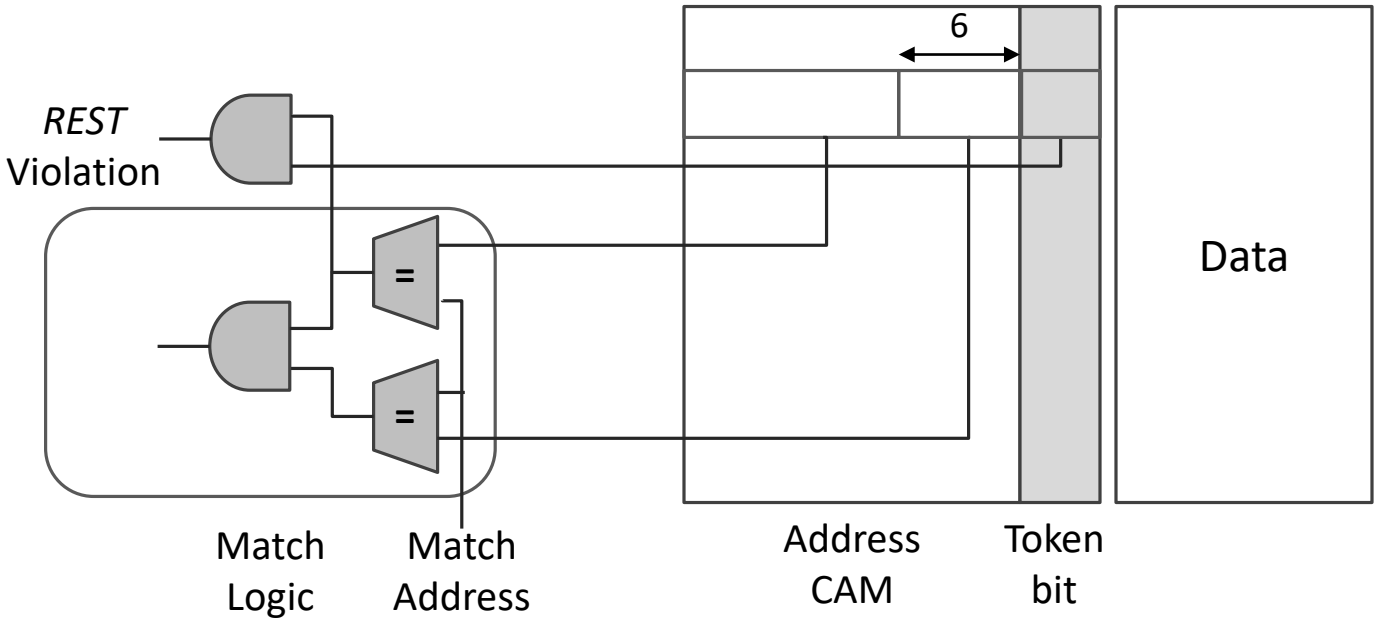
Load-Store Queue

- Forwarding breaks semantics
- 512b data entries
- Detecting unaligned token access

Add 1b tag

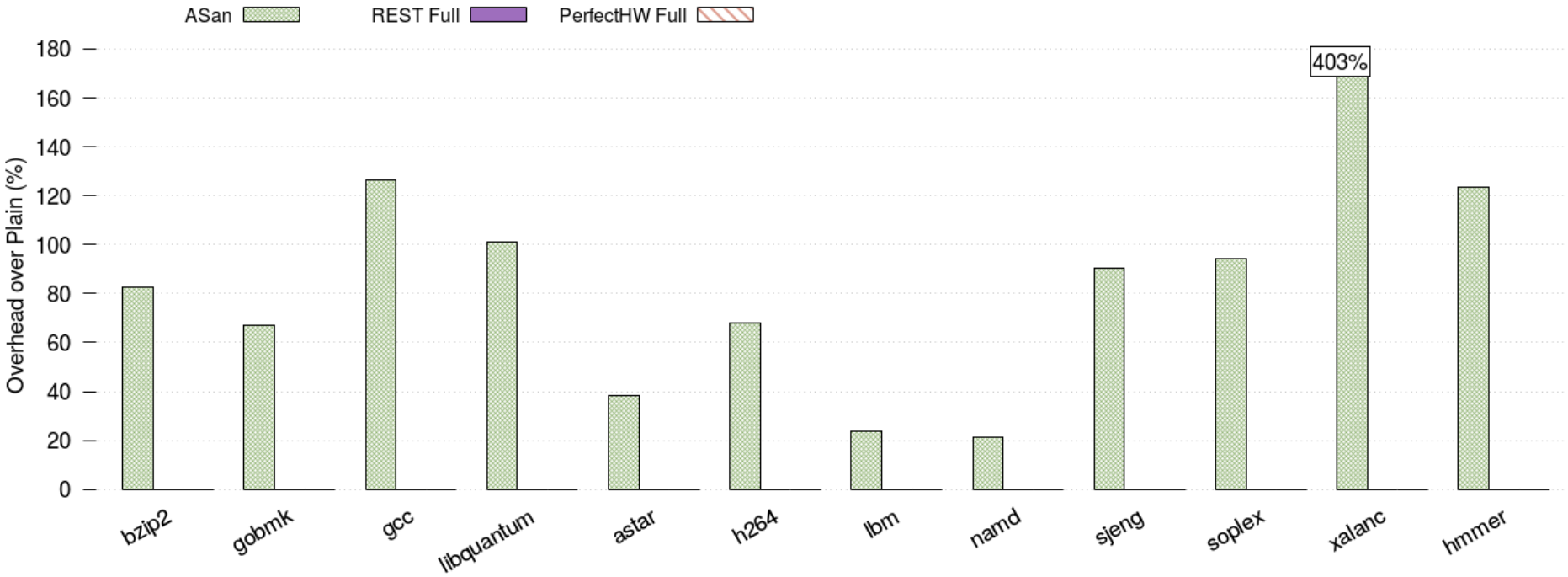
Only update token bit

Split regular match logic

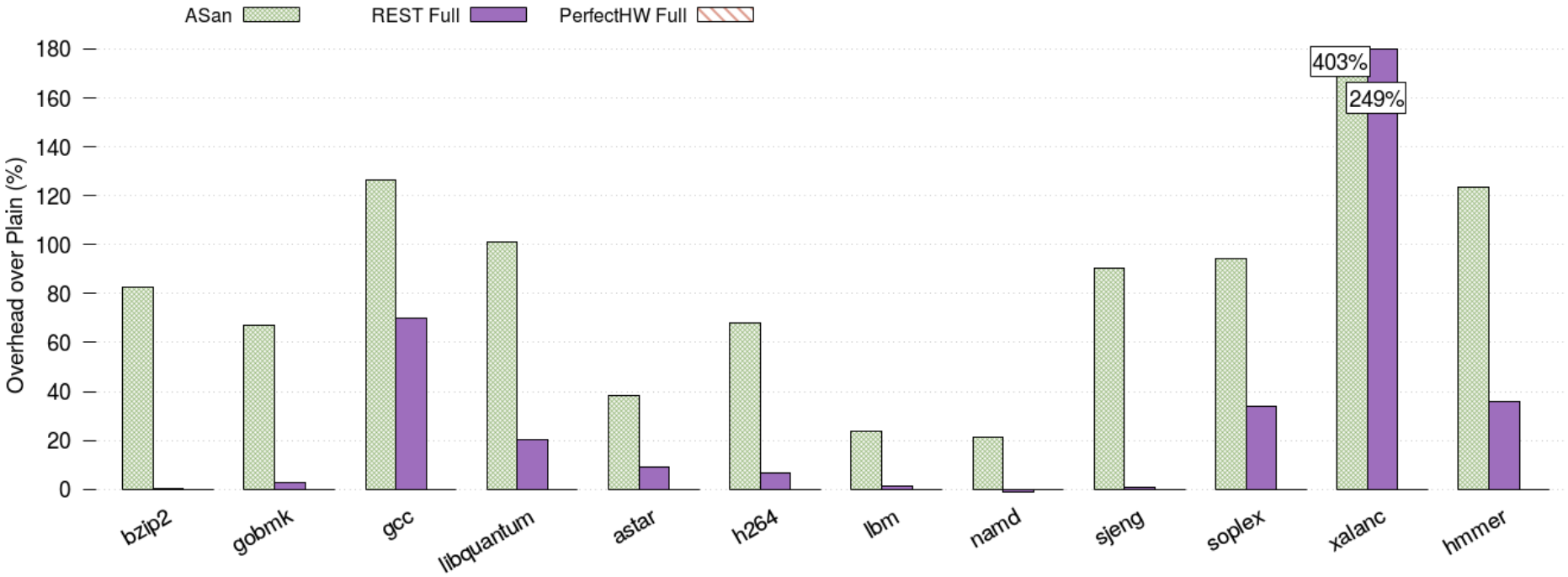


REST Overhead

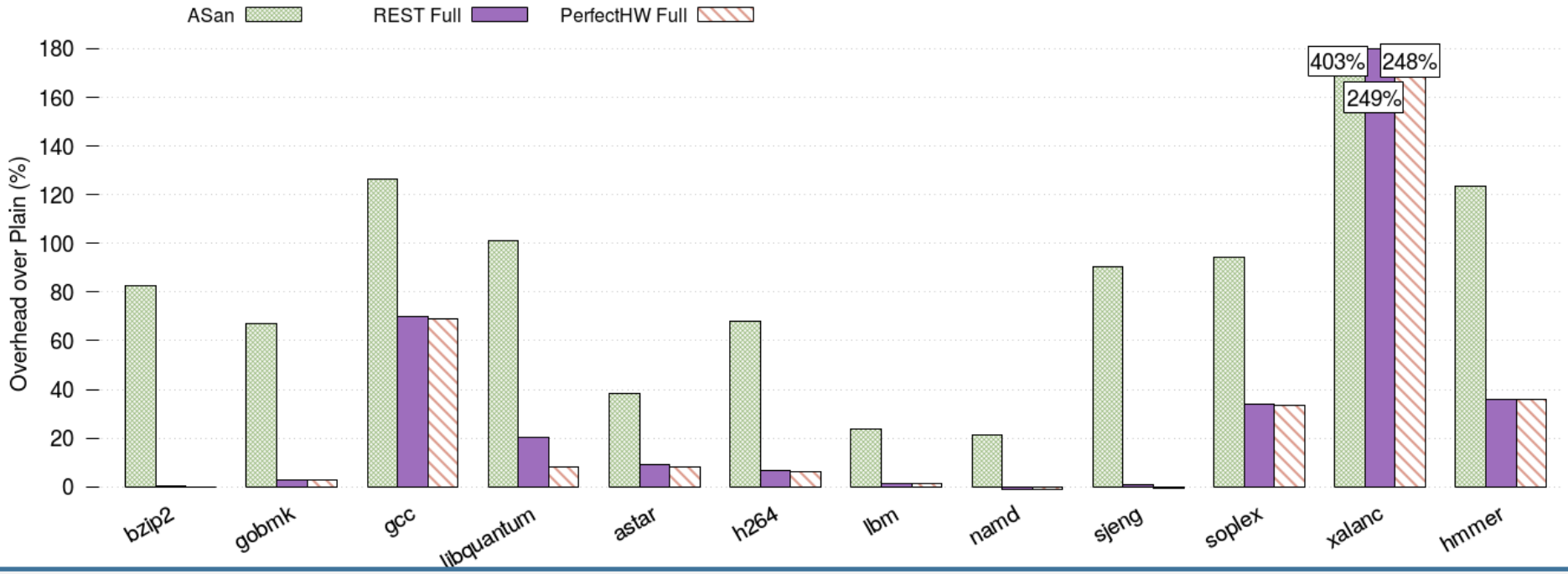
REST Performance



REST Performance



REST Performance



REST primitive overhead near-zero.

Software overhead mostly from allocator.

To conclude…

REST: Hardware/software mechanism to detect common memory safety errors

- Low overhead, low complexity hardware implementation
- Heap safety for legacy binaries

22-90% faster than comparable software solution on SPEC CPU

Questions?