YARRA

Modular Protection against Non-control Data Attacks

COLE SCHLESINGER

JUNE 28TH, 2011

Adviser: David Walker

Joint work with Karthik Pattabiraman, Nikhil Swamy, David Walker, and Ben Zorn.

CSF 2011

Report from the Front Lines

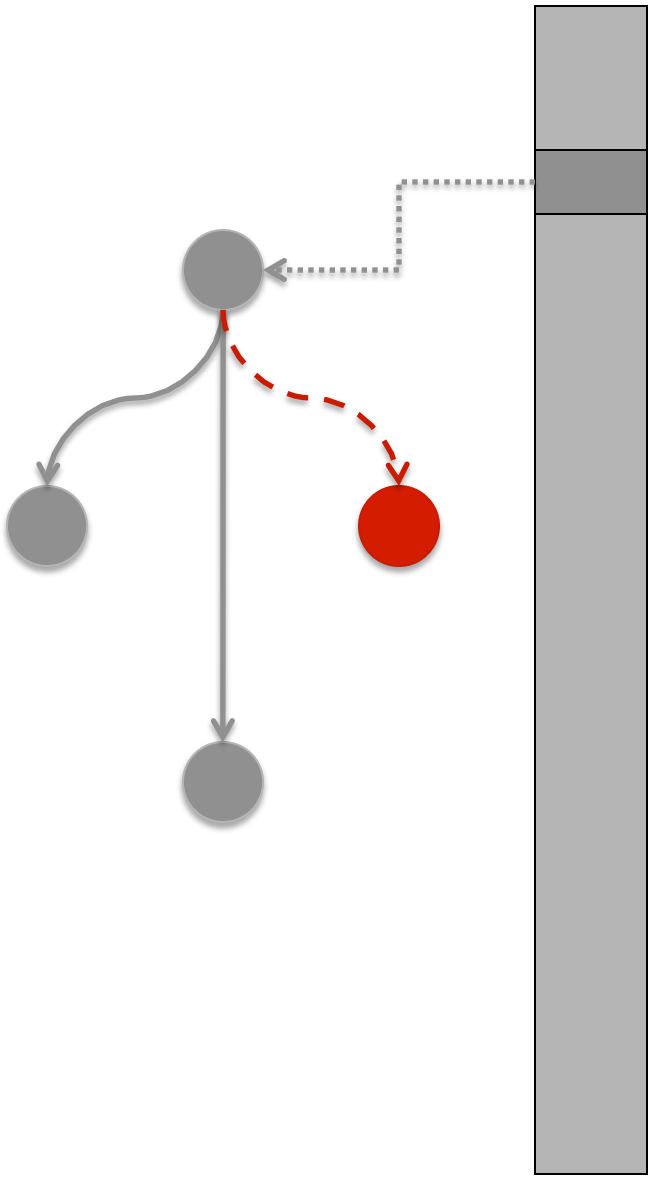
The battle:

Attackers vs. *C, C++ programmers*

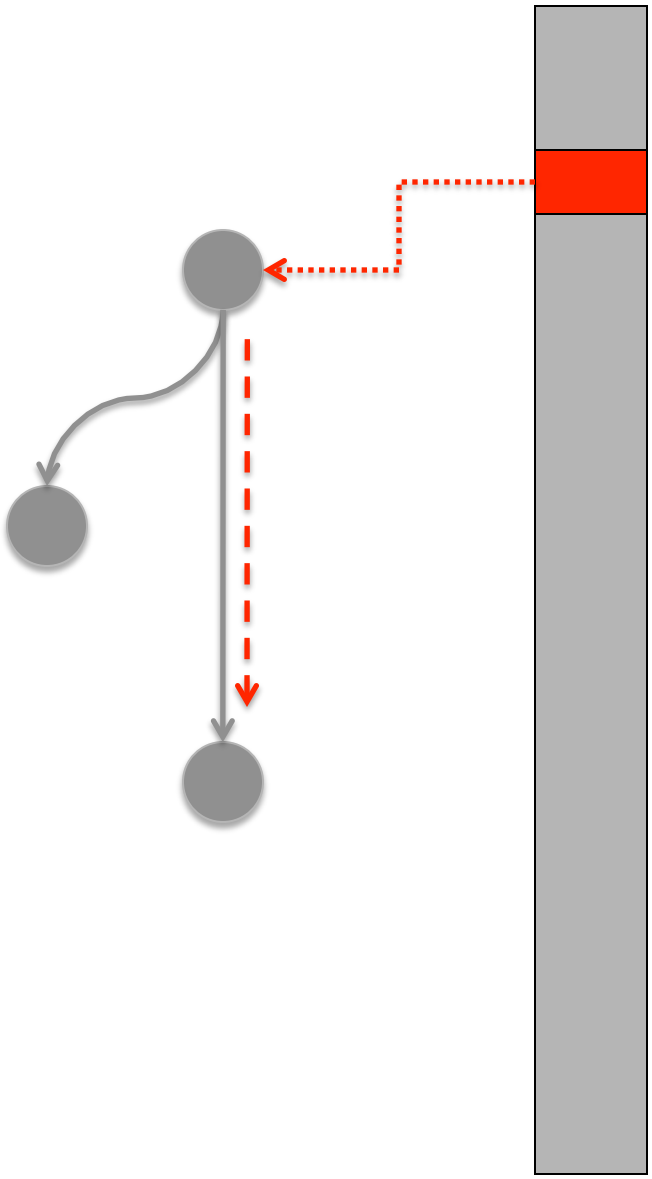
The battleground:

Legacy code, new projects, and *new components*

- **Control-flow attack:** alters control data to execute malicious code or out-of-context library code.
  - stack-smashing, return-to-libc attacks, etc.
  - many protections, including control flow integrity



- **Non-control-data attack:** alters non-control data to break program invariants.
  - configuration data
  - user input
  - user identity data
  - decision-making data



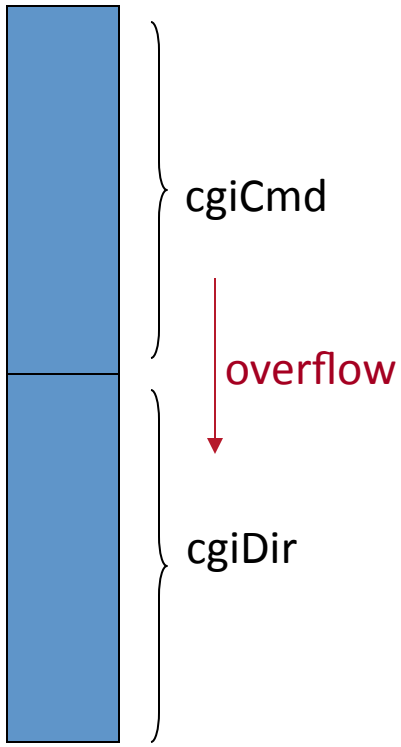
A non-control data attack

[source: Akritidis et al.; inspired by true nullhttpd attack]

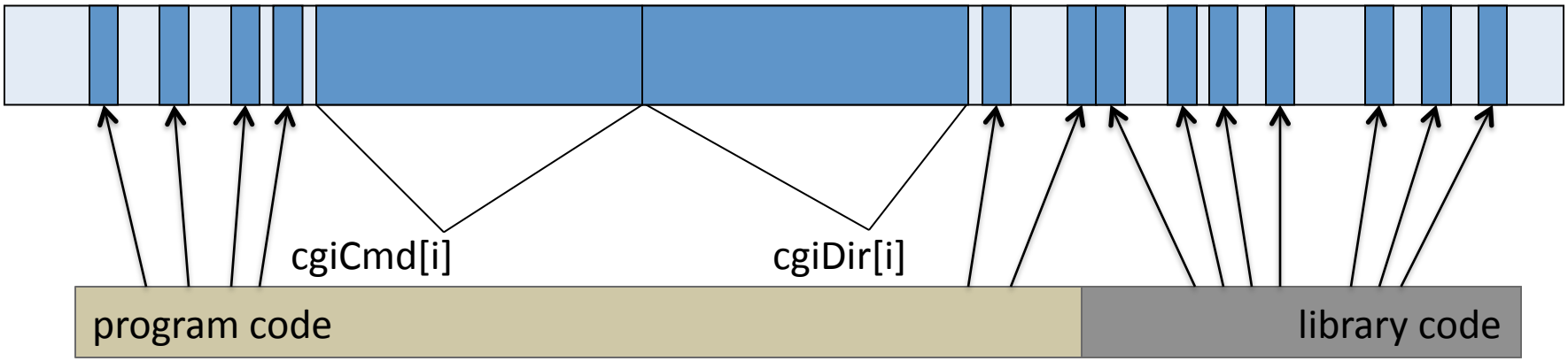
Web Server Code:

```
1  static char cgiCmd[1024];
2  static char cgiDir[1024];
3  void ProcessCGIRequest(char* msg, int sz) {
4      int flag, i=0;
5      while (i < sz) {
6          cgiCmd[i] = msg[i];
7          i++;
8      }
9    flag = CheckRequest(cgiCmd);
10   if (flag) {
11     Log(" . . . ");
12     ExecuteRequest(cgiDir, cgiCmd);
13   }}
```

~~dir contains approved executables~~



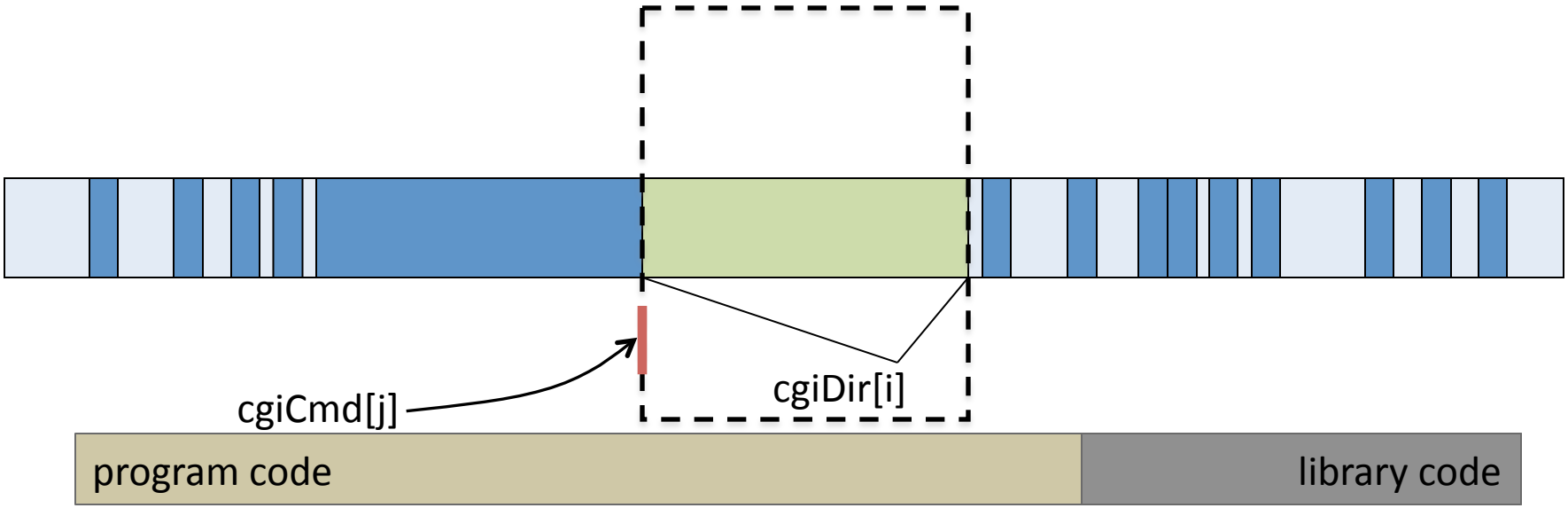
Array Bounds Checking



- Must check *every* indexing operation
  - even on **non-critical data**
  - and **inside libraries**

Three Goals

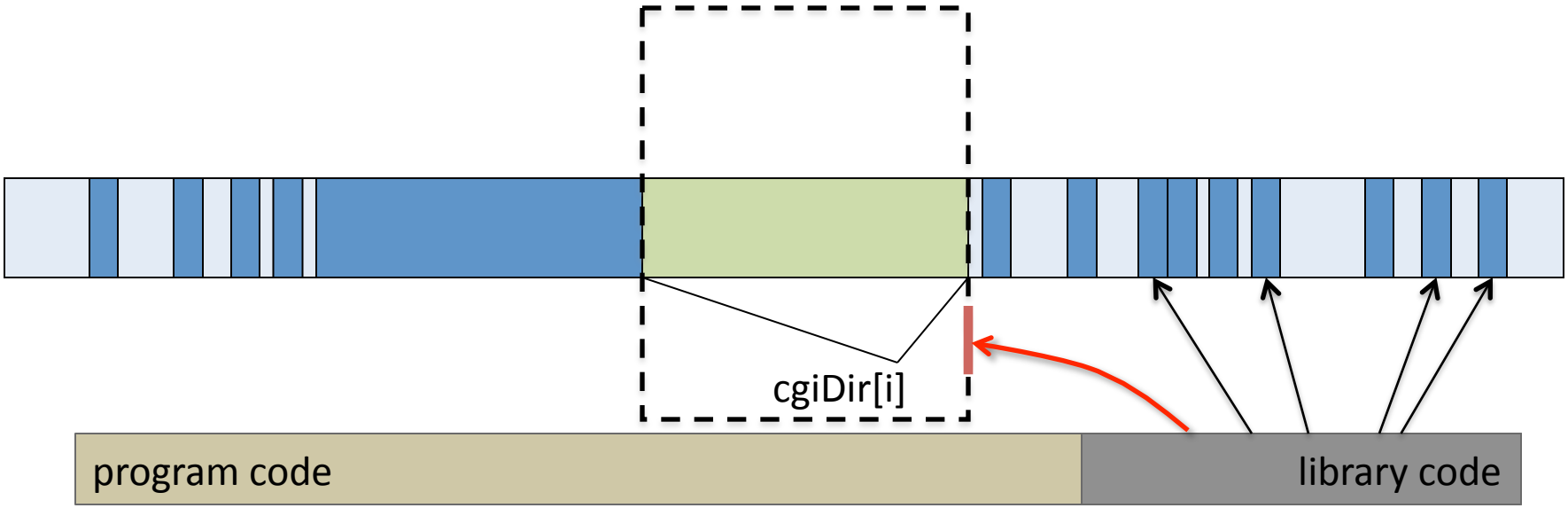
1: Targeted Protection



Protect *critical* data

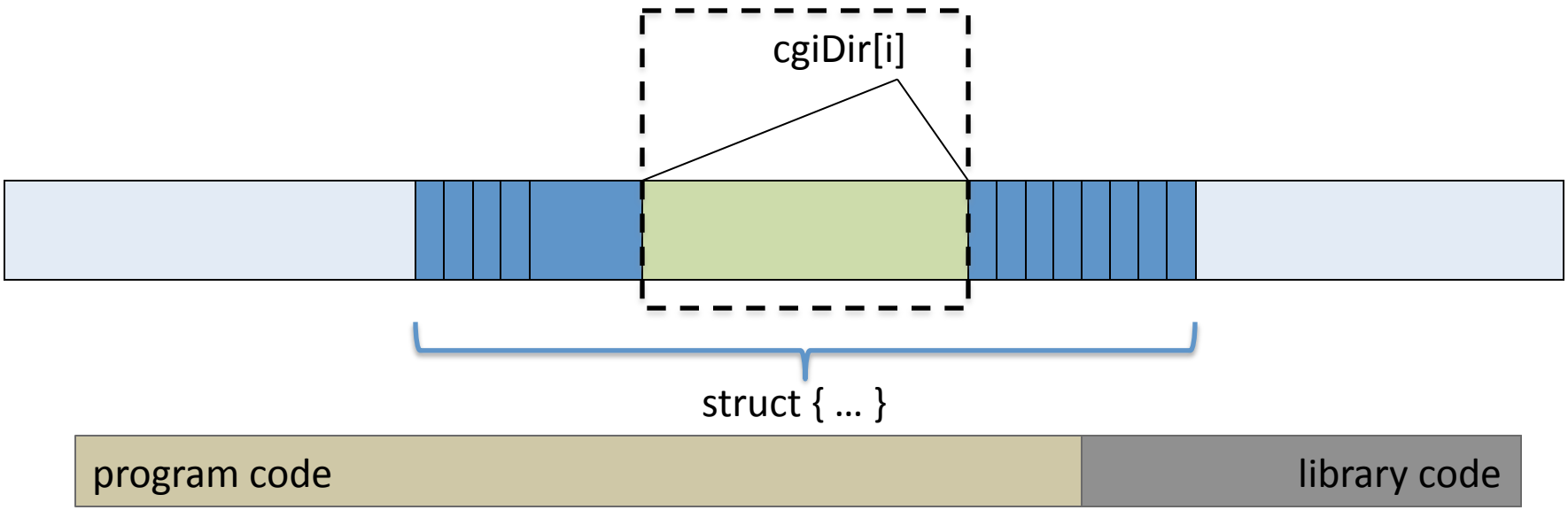
(without protecting *all* data)

2: Modular Protection



without checking *all* the code

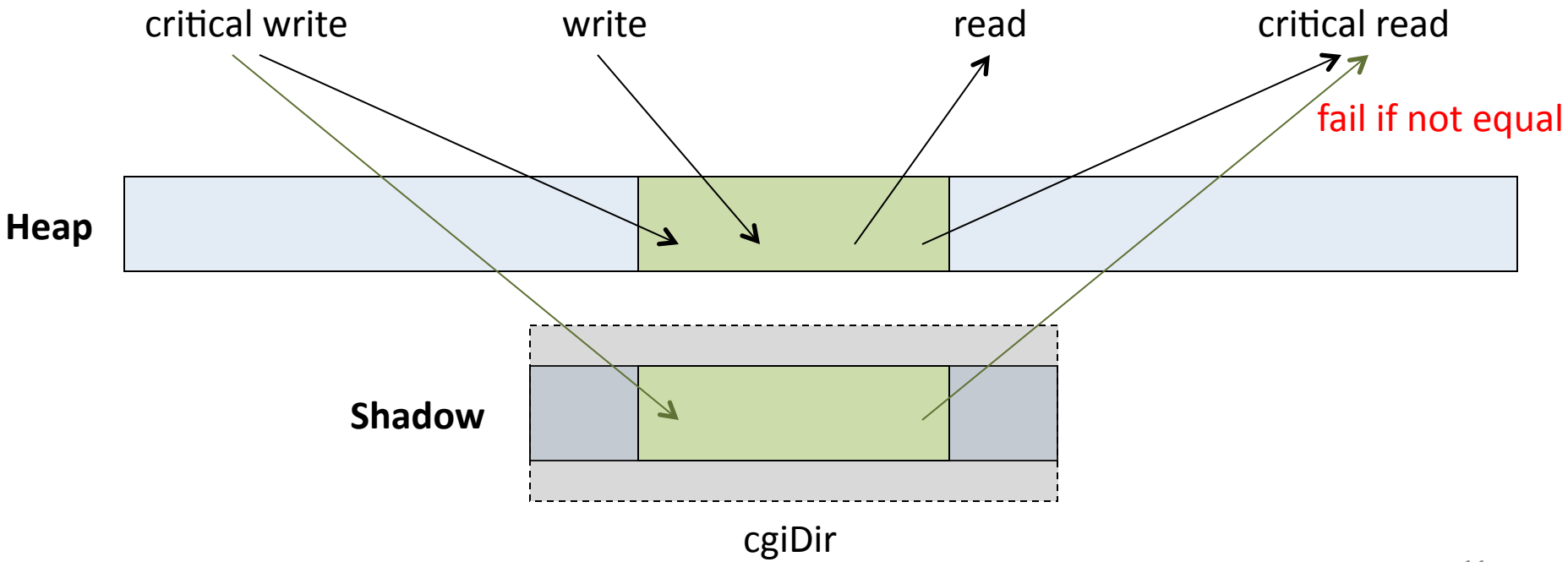
3: Format Preservation



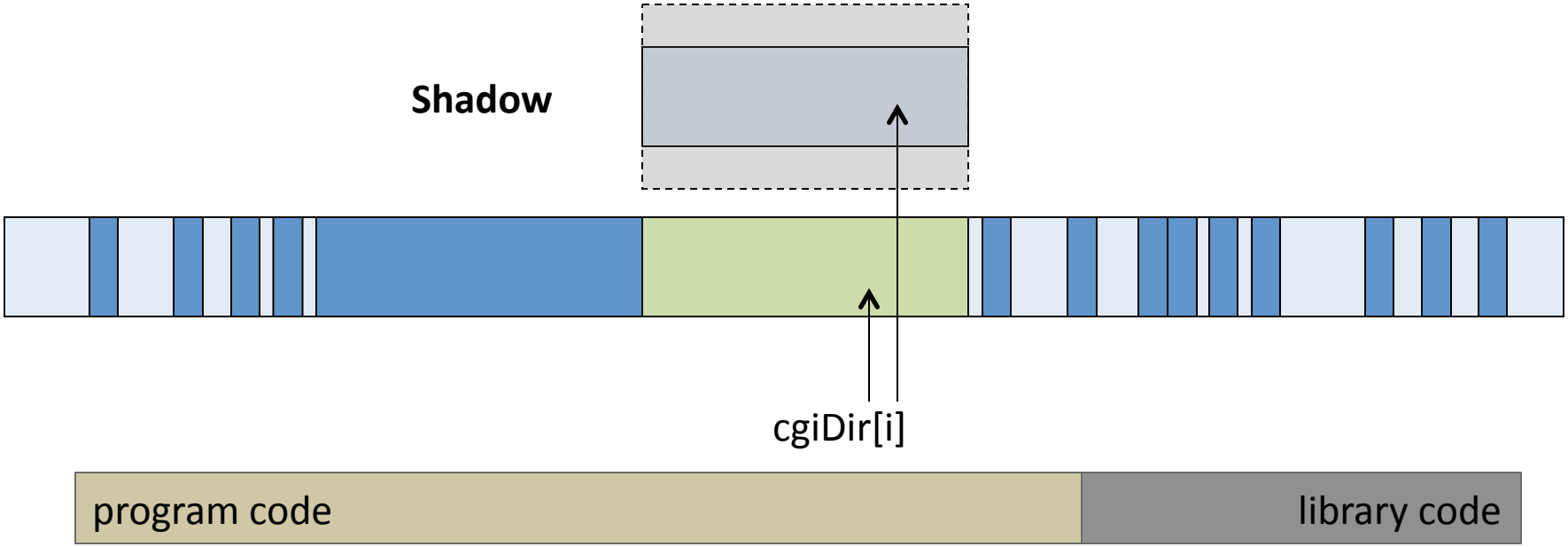
and without changing/padding objects in memory.

YARRA: An Extension to C

Critical memory model → formal basis for partial memory safety!



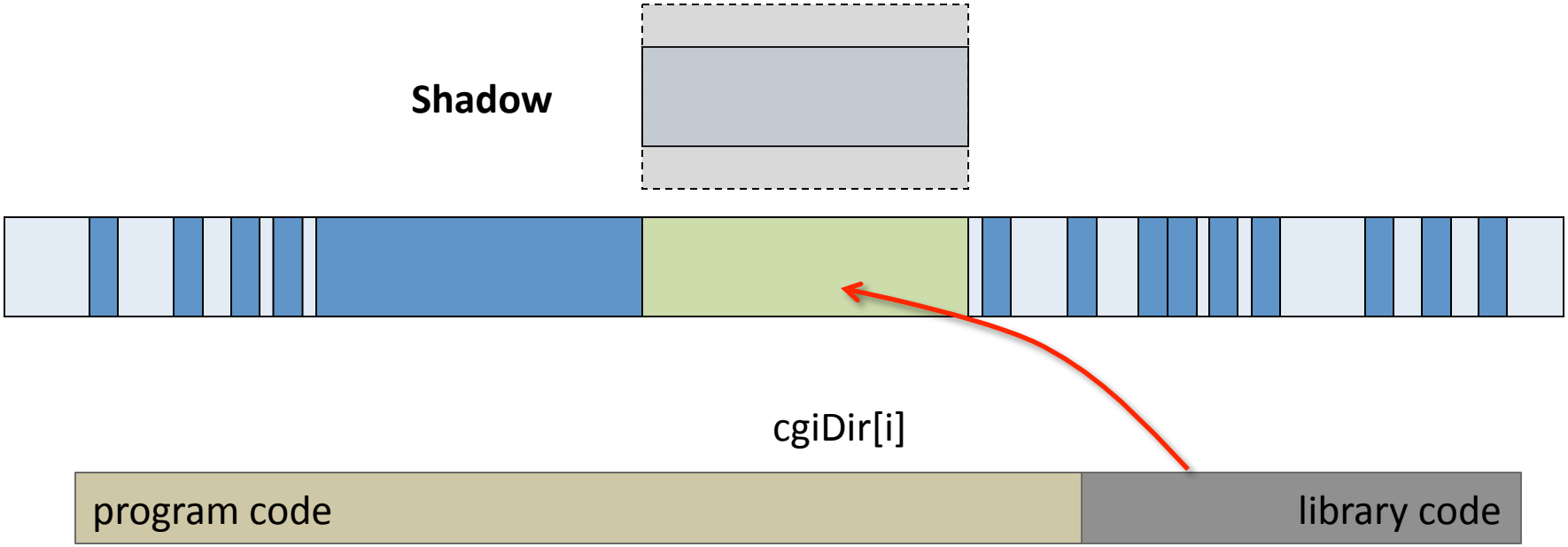
1: Targeted Protection



Protect *critical* data

(without protecting *all* data)

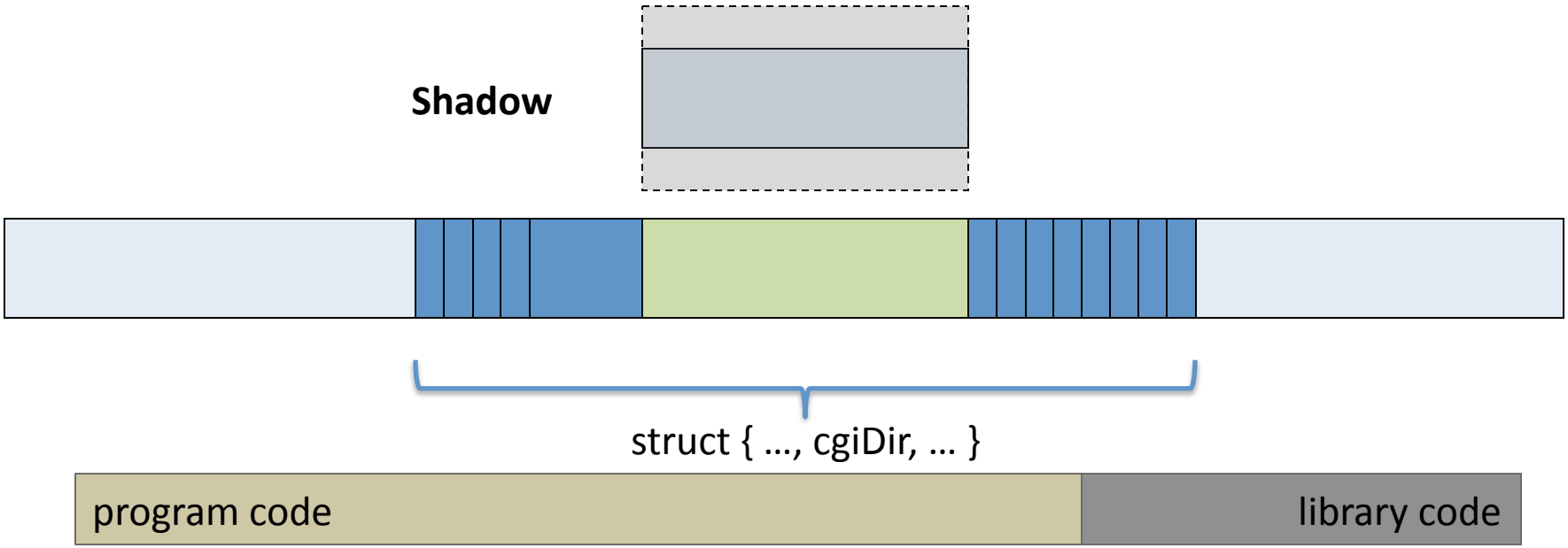
2: Modular Protection



Protect *critical* data

(without protecting *all* data)

3: Format Preservation



and without changing/padding objects in memory.

YARRA: An Extension to C

Inverse array bounds checking – YARRA = ARRAY^{-1}

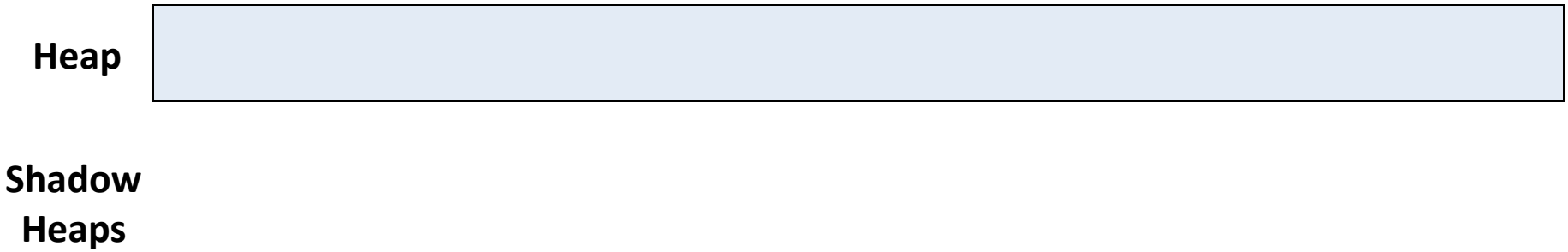
Formalization

- Language design based on the abstraction of critical data and heap partitions.
- Program logic + a **frame rule** for modular reasoning and partial memory safety.
- Formal protection against non-control data attacks.

Implementation

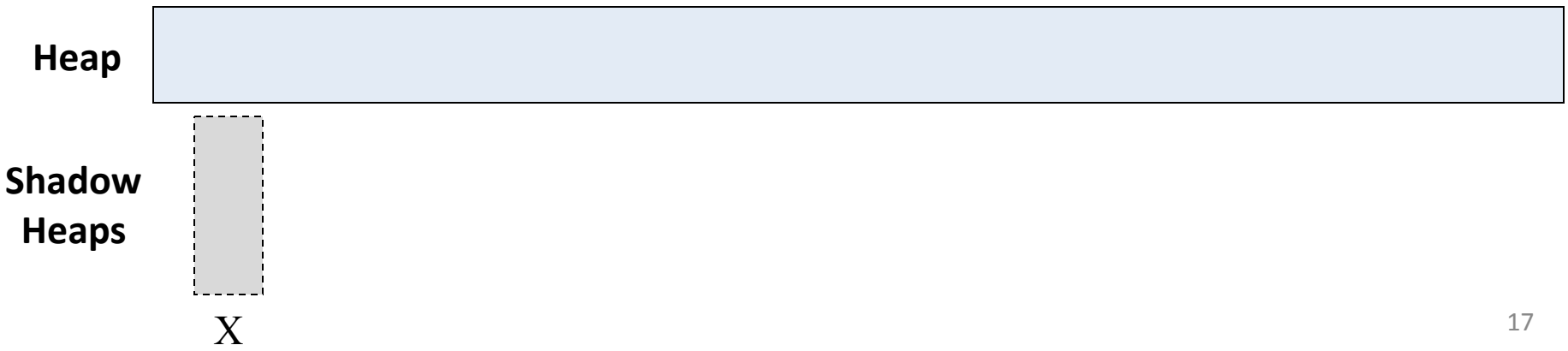
- Compiler + runtime system implementing YARRA semantics in **two different ways**.
- Evaluation on four open source programs with known non-control data vulnerabilities.
- Negligible end-to-end overhead.

Language Extensions



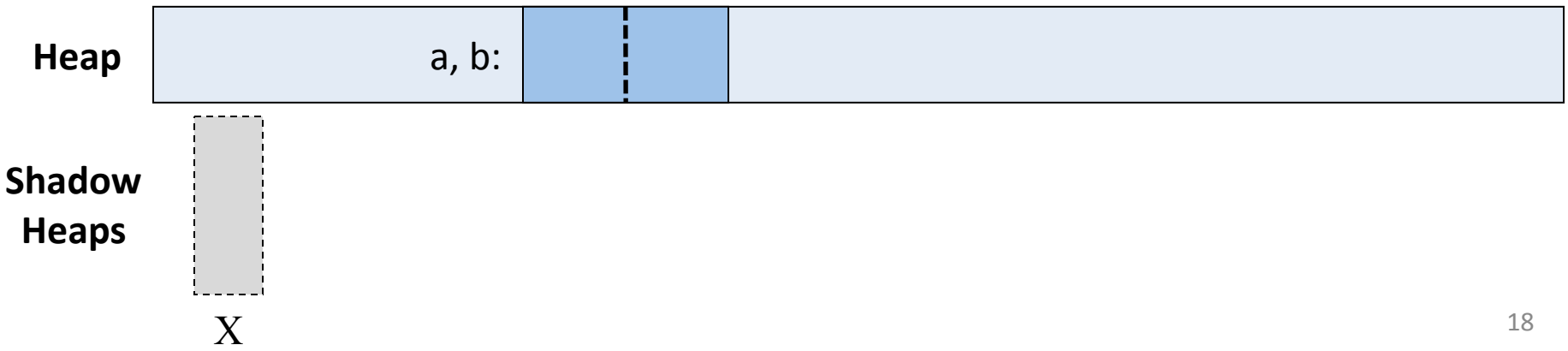
Language Extensions

```
yarra struct {int a; int b;} X;
```



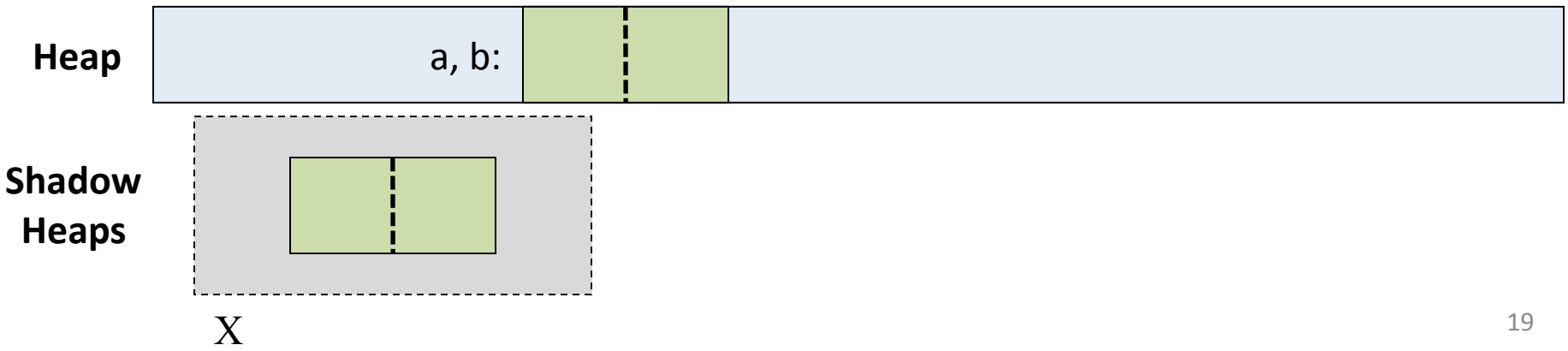
Language Extensions

```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
```



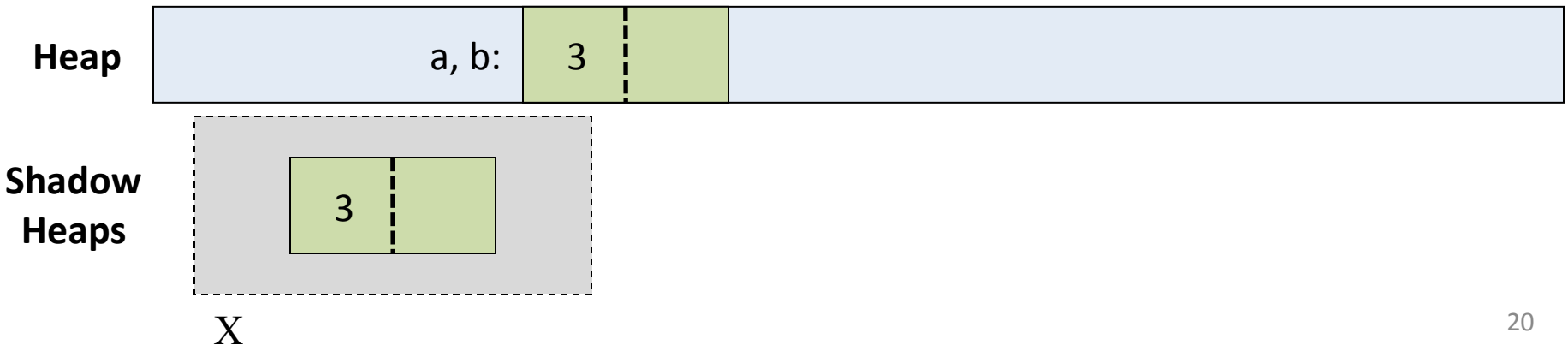
Language Extensions

```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
bless(X, px);
```



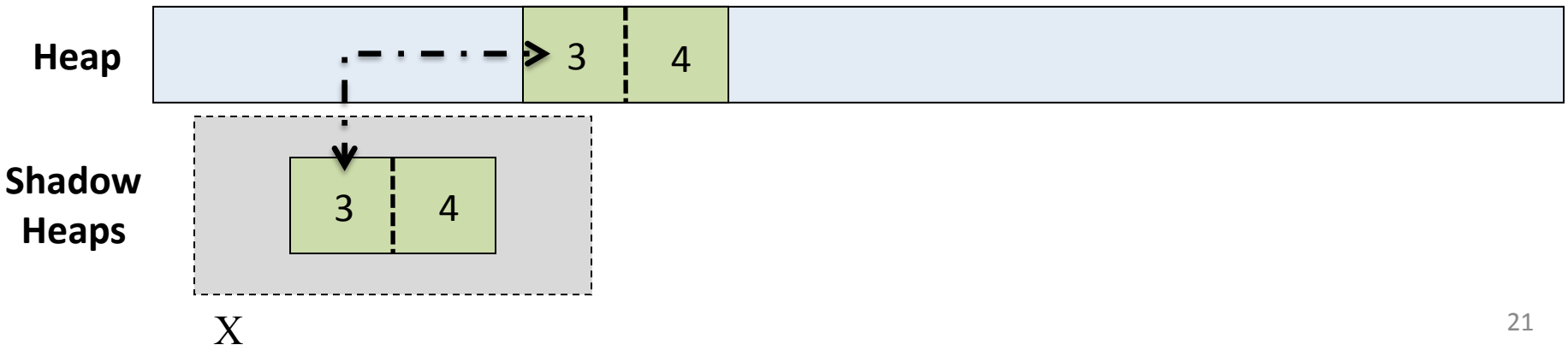
Language Extensions

```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
bless(X, px);
px->a = 3;
```



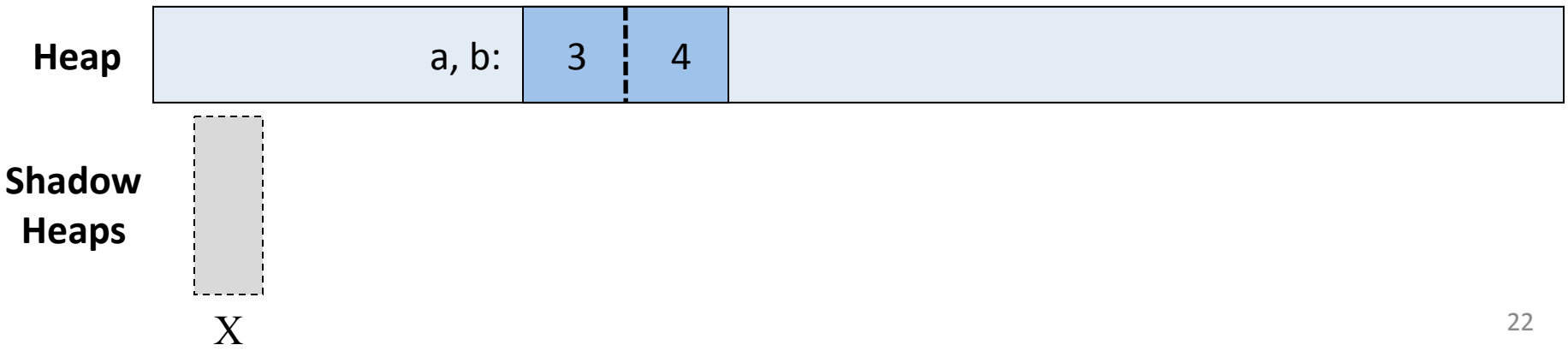
Language Extensions

```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
bless(X, px);
px->a = 3;
px->b = px->a + 1;
```



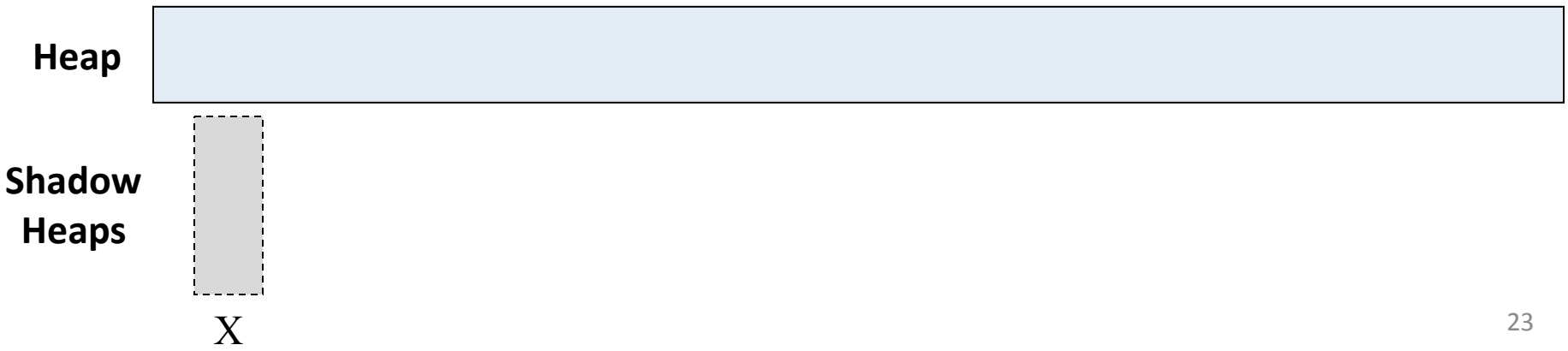
Language Extensions

```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
bless(X, px);
px->a = 3;
px->b = px->a + 1;
unbless(X, px);
```



Language Extensions

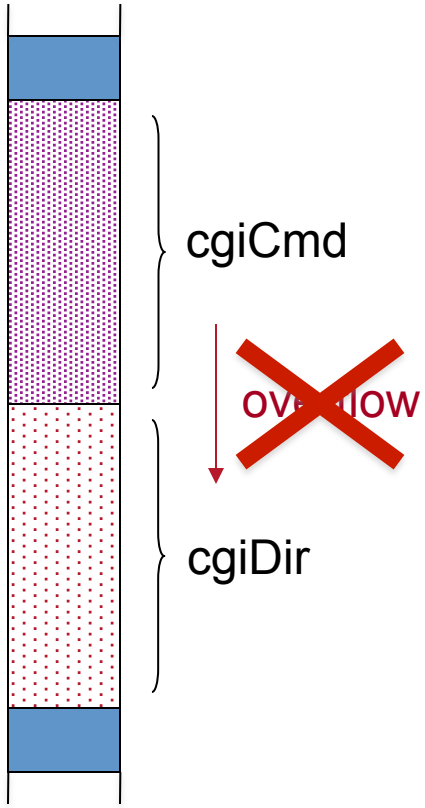
```
yarra struct {int a; int b;} X;
X *px = malloc(sizeof(X));
bless(X, px);
px->a = 3;
px->b = px->a + 1;
unbless(X, px);
free(px);
```



type declarations for data with high integrity

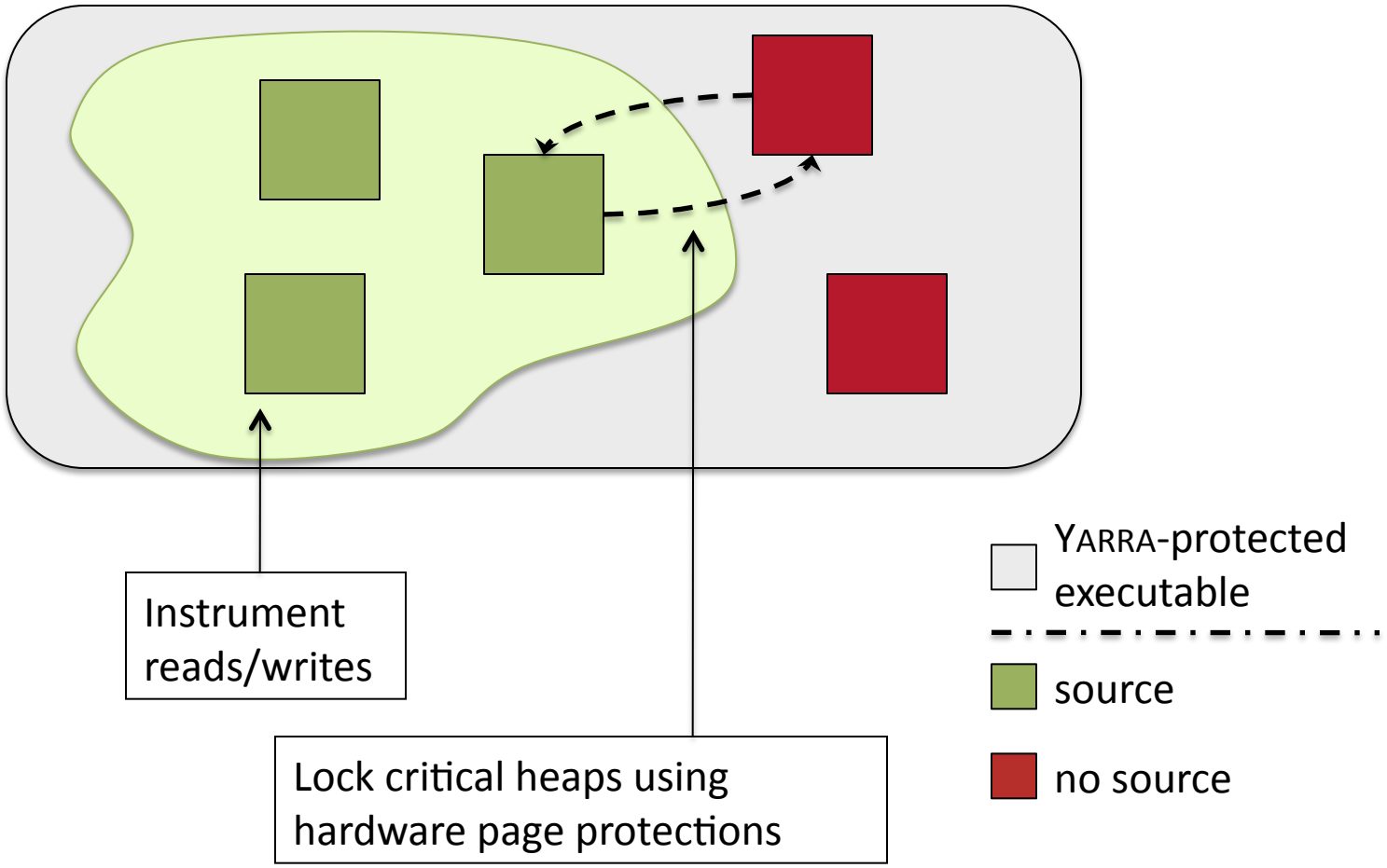
high integrity data structures protected by run-time system

```
1  yarra struct {char cc;} cchar;
2  yarra struct {char dc;} dchar;
3  static cchar cgiCmd[1024];
4  static dchar cgiDir[1024];
5  void ProcessCGIRequest(char* msg, int sz) {
6     int flag, i=0;
7     while (i < sz) {
8           cgiCmd[i].cc = msg[i];
9           i++;
10    }
11    flag = CheckRequest(cgiCmd);
12    if (flag) {
13       Log(" . . . ");
14       ExecuteRequest(cgiDir, cgiCmd);
15    }}
```



on overflow, access pointer has type cchar[ ] but memory written to has type dchar[]

Implementation



Program Logic

Classical Hoare-style program logic:

$$\Gamma; \Delta \vdash \{P\}\ s\ \{Q\}$$

Δ	modified set
P	precondition
s	statement
Q	postcondition

The Frame Rule

$$\frac{\Gamma;\Delta \setminus FV(F) \vdash \{P\}\ s\ \{Q\}}{\Gamma;\Delta \vdash \{P \wedge F\}\ s\ \{Q \wedge F\}}$$

F is preserved across s if s does not modify the free variables of F.

Key technical idea:

- A partitioned model of the heap
- Non-critical data resides in the normal heap H
- Values of critical type Y reside in a separate heap region named Y.

Invariants on Y are *preserved* over modifications to H.

$$\frac{\Gamma; H \vdash \{H(\ell_1) = 3\}\ s\ \{\mathit{True}\}}{\Gamma; H \vdash \{H(\ell_1) = 3 \wedge Y(\ell_2) = 4\}\ s\ \{\mathit{True} \wedge Y(\ell_2) = 4\}}\ \text{T-Frame}$$

Defining an Attack Model

Formally See the paper.

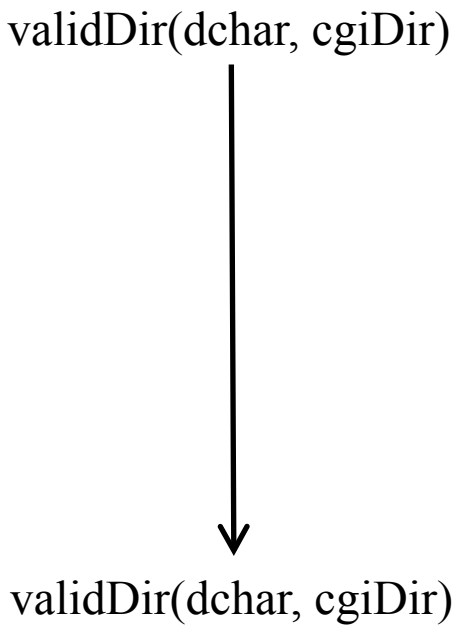
Informally An attacker is a program that is free to make arbitrary changes in the heap H.

(Trivial) Attack Specification

$$H \vdash \{True\}\ s\ \{True\}$$

The Frame Rule in Action

```
1  yarra struct {char cc;} cchar;
2  yarra struct {char dc;} dchar;
3  static cchar cgiCmd[1024];
4  static dchar cgiDir[1024];
5  void ProcessCGIRequest(char* msg, int sz) {
6     int flag, i=0;
7     while (i < sz) {
8           cgiCmd[i].cc = msg[i];
9           i++;
10    }
11    flag = CheckRequest(cgiCmd);
12    if (flag) {
13       Log(" . . . ");
14       ExecuteRequest(cgiDir, cgiCmd);
15    }}
```



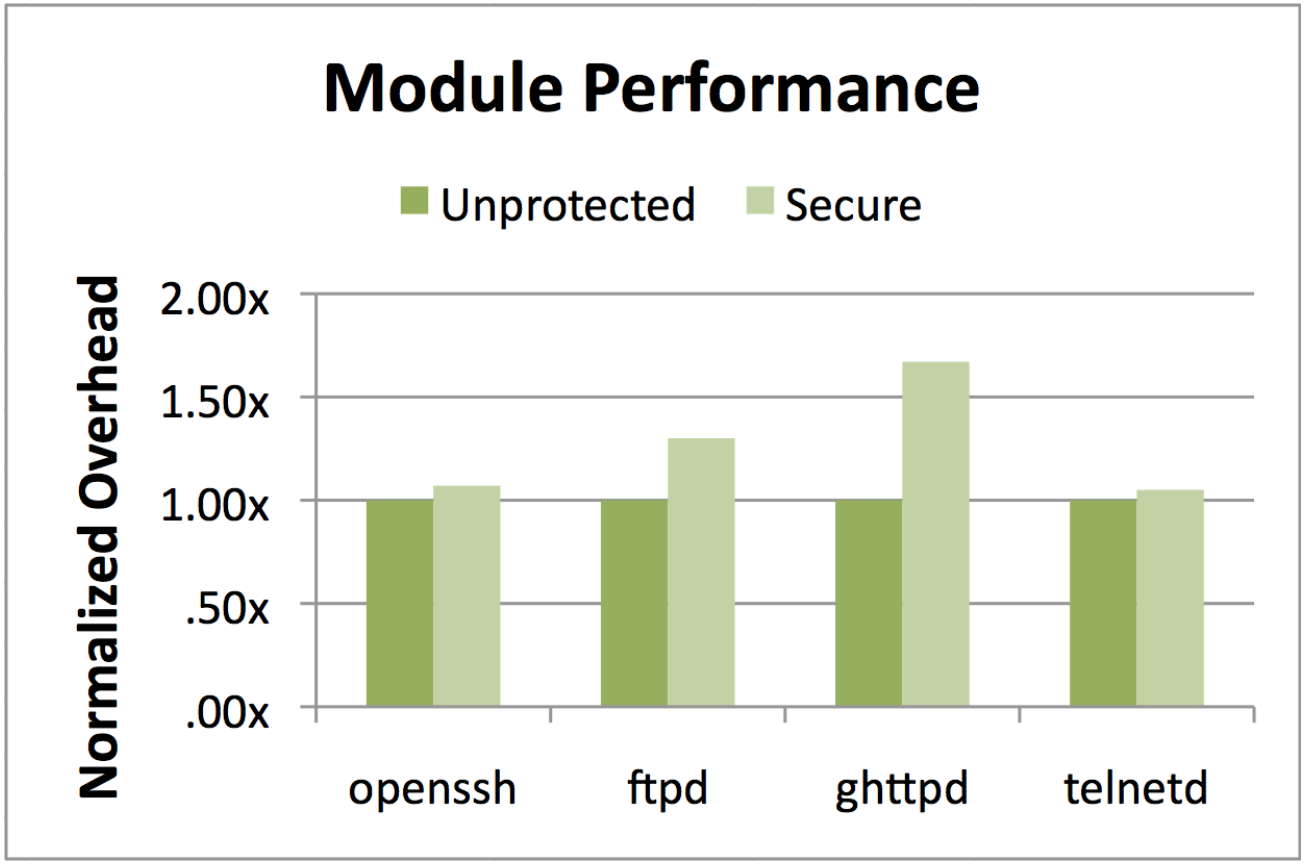
Evaluation & Results

Protecting Security-critical Data

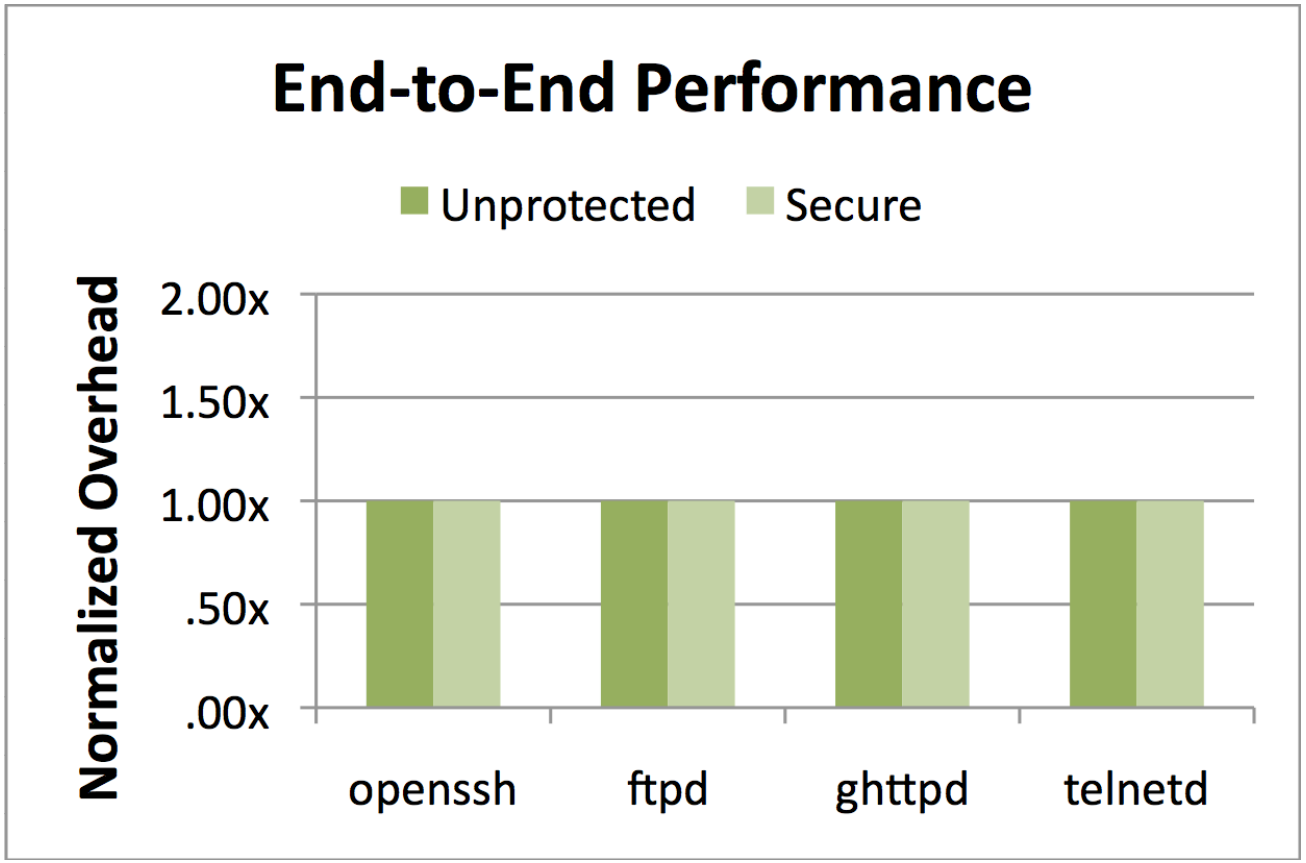
Program	Original LOC	Modified LOC	Modified %
sshd	60,148	497	0.8%
wu-ftpd	17,993	262	1.5%
telnetd	3,962	63	1.6%
ghttpd	514	69	13%

- **SSHD:** OpenSSH daemon
- **WU-FTPD:** ftp server
- **TelnetD:** telnet server
- **Ghttpd:** web server

Protecting Security-critical Data



Protecting Security-critical Data



Summary

- YARRA characterizes **partial memory safety** in an unsafe context.
- The program logic admits a powerful **type-based frame rule** for modular reasoning.
- The language extension is minimal and easy to use, and we have **two implementations** of the semantics.
- We can harden **real non-control data vulnerabilities** with **negligible performance cost**.

Looking Ahead

- **YARRA for static verification**
  - Right now: **VCC + YARRA**
  - Managed/unmanaged language interaction
- **YARRA with other runtime protections**
  - YARRA + CFI, SFI and more.