Body Armour for Binaries

protecting legacy binaries from memory corruption attacks

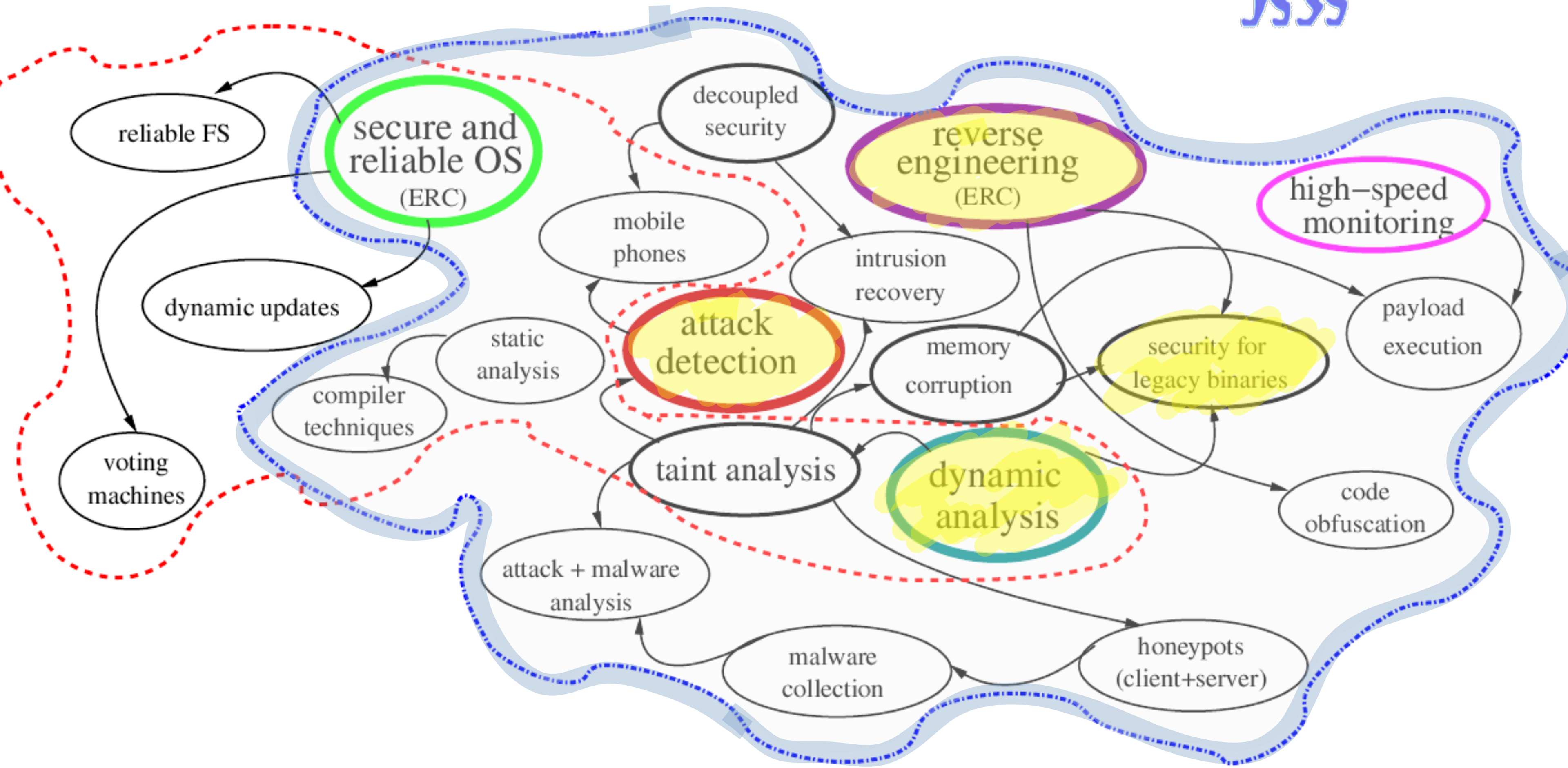
Herbert Bos

VU University Amsterdam

Grants

- ERC StG “Rosetta”
- EU FP 7 Syssec
- DG Home iCode

Systems Security @ VU



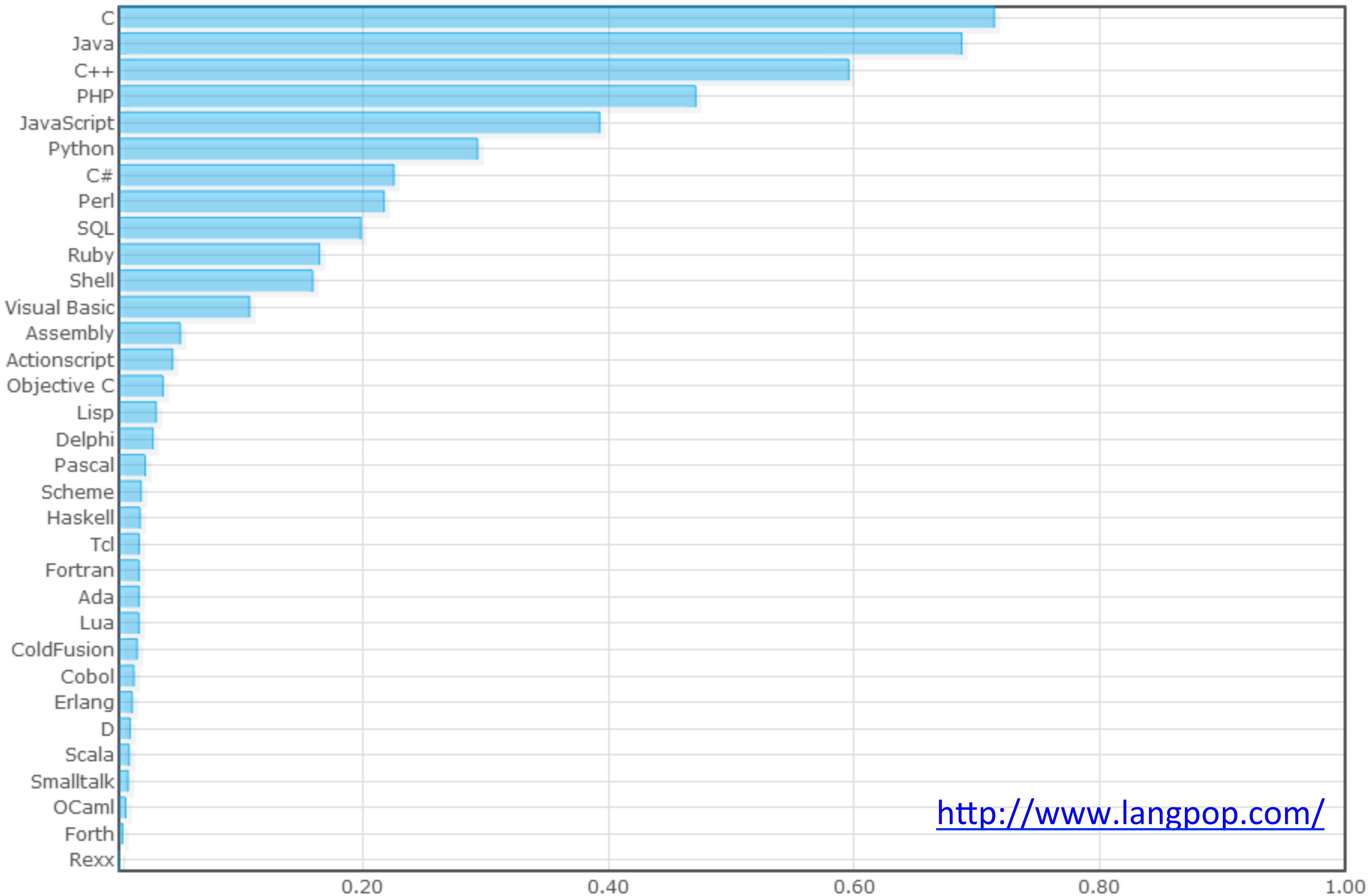
This talk is based on two papers

- Asia Slowinska, Traian Stancescu, Herbert Bos
  Howard: a dynamic excavator for reverse engineering data structures (NDSS'11)
- Asia Slowinska, Traian Stancescu, Herbert Bos
  Body armor for binaries: preventing buffer overflows without recompilation (USENIX'12)

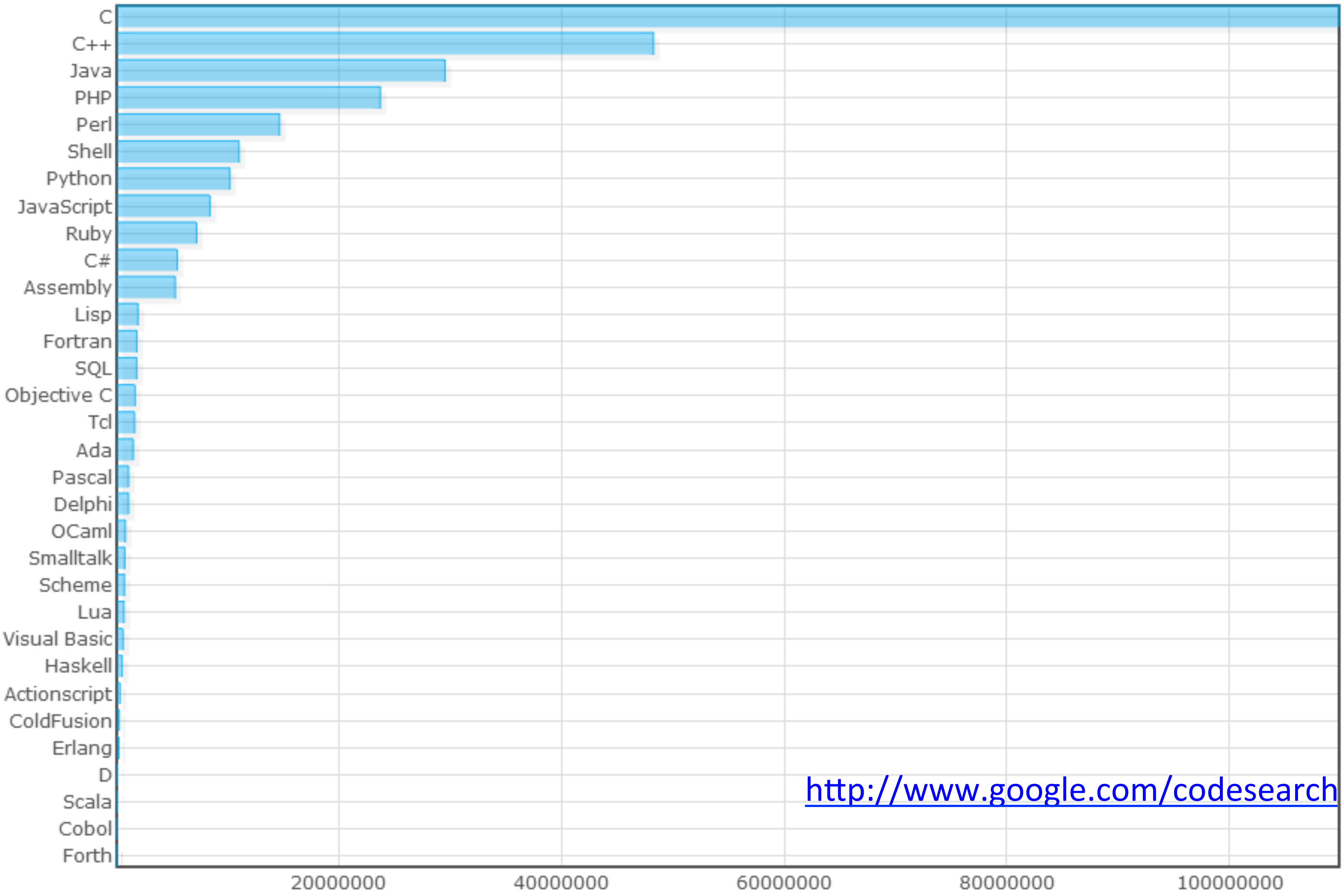
vrije Universiteit

syssec

The most popular language in the world



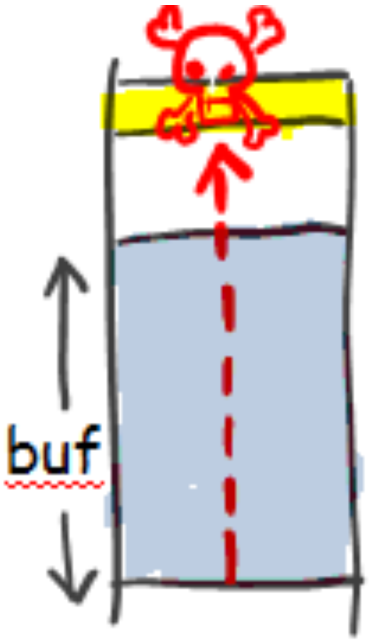
The most popular language in the world



http://www.google.com/codesearch

Buffer overflows

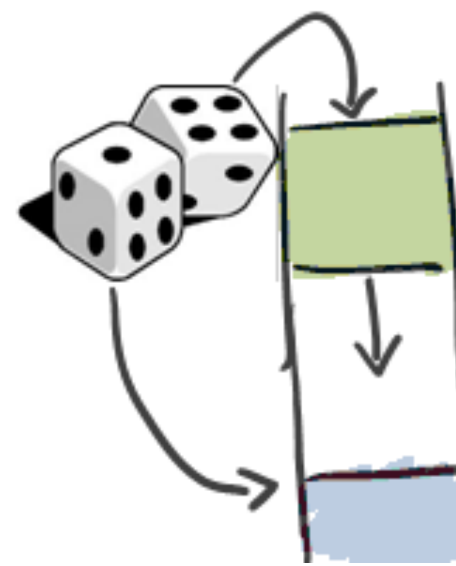
- Perpetual top-3 threat
  - SANS CWE Top 25 Most dangerous programming errors
- Most drive-by-downloads
  - infect browser, download malware



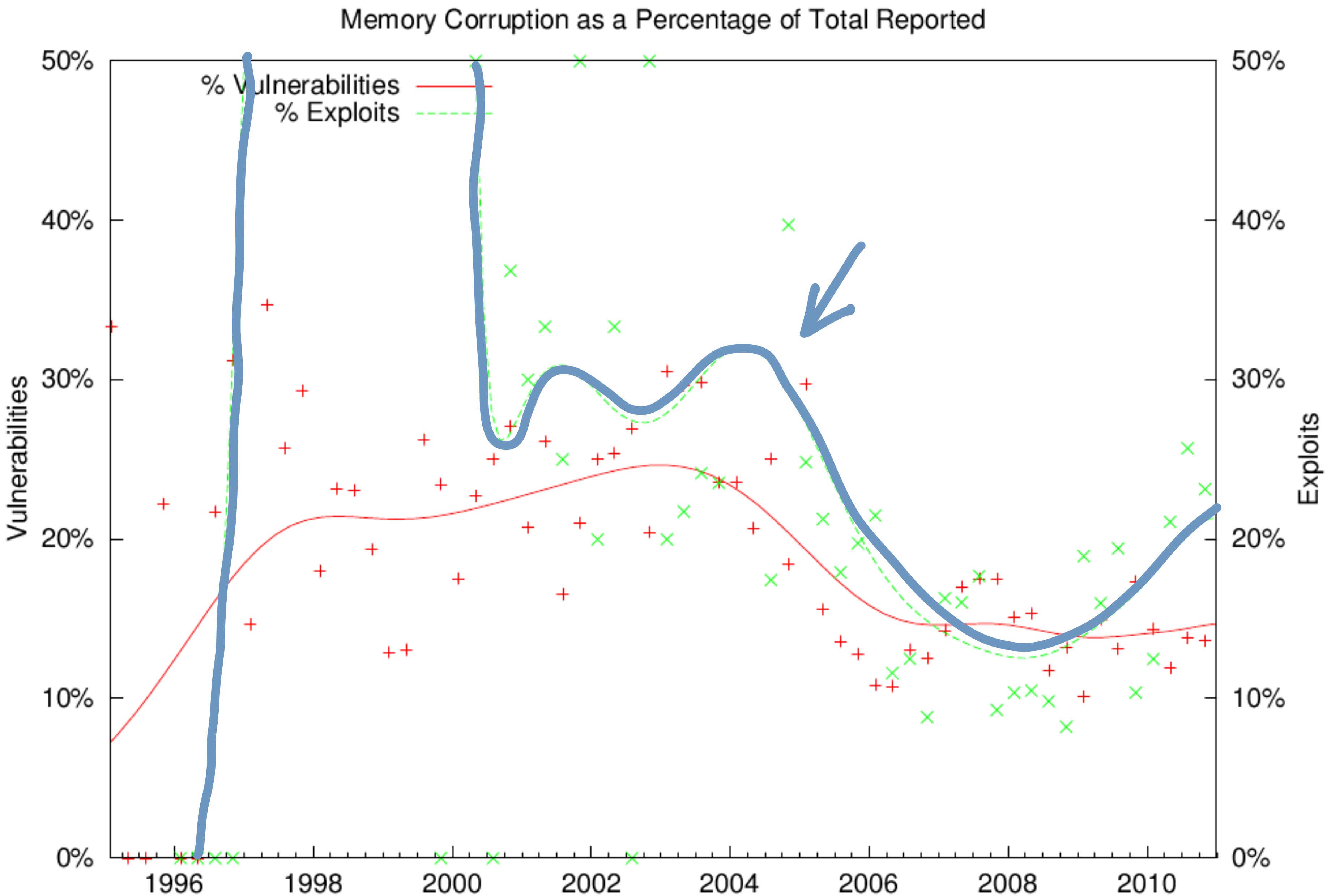
Many defensive measures

- Canaries (StackGuard and friends)
- NX bit / W⊕X
- ASLR

ProPolice



Still they come



vrije Universiteit

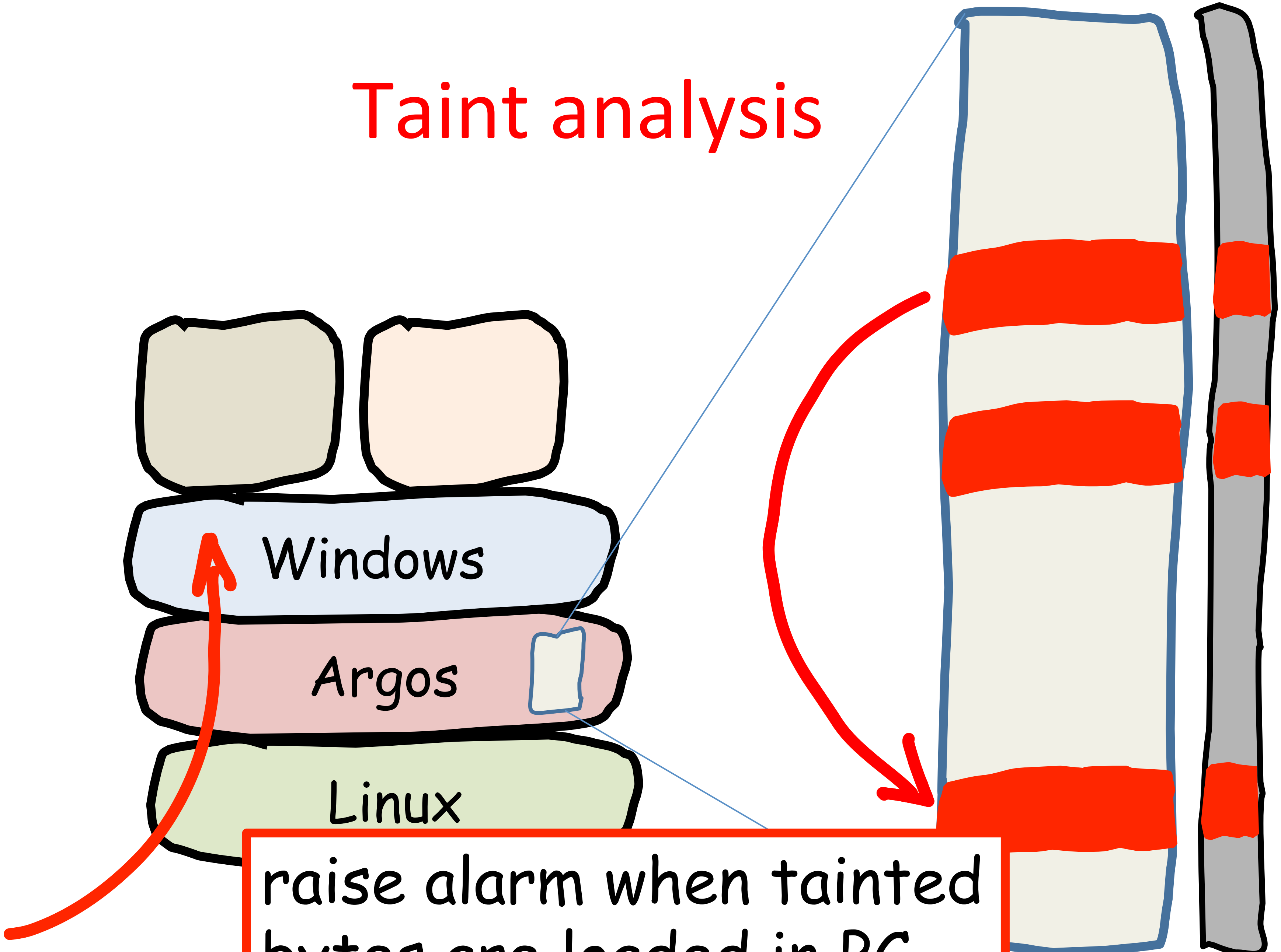
syssec

And legacy code?

- we do not have source code
  - we probably do not even have symbols
- we cannot recompile
  - most protective measures require recompilation
- we cannot protect

Taint Analysis?

Taint analysis

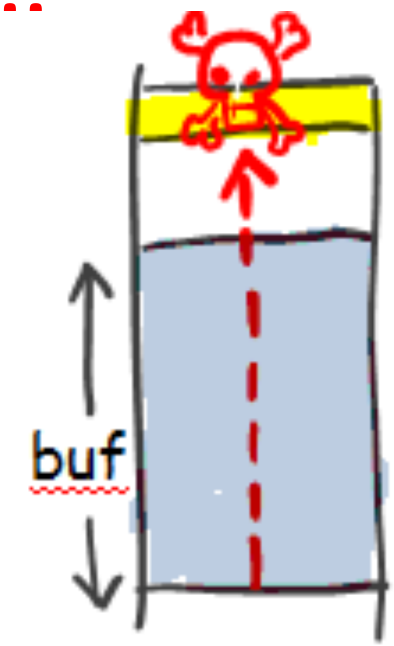


raise alarm when tainted bytes are loaded in PC

Taint tracking: useful, but slow

...and detects not the attack, but its manifestation...

just missed it!



vrije Universiteit

syssec

...and does not detect attacks on non-control data at all!

```
void get_private_medical_data (int uid) {
  int c,i=0;
  int authorized = check(uid); // result=0 for attacker
  char patientid[8];

  printf ("Type patientid, followed by the '#' key\n");
  while (((c=getchar())!='#') patientid[i++] = c;

  if (authorized) print_medical_data (patientid);
  else printf ("sorry, you are not authorized\n");
}
```

- trivially exploitable
- not prevented by ASLR, NX, or StackGuard

BinArmor

A Body Armour for Binaries

vrije Universiteit

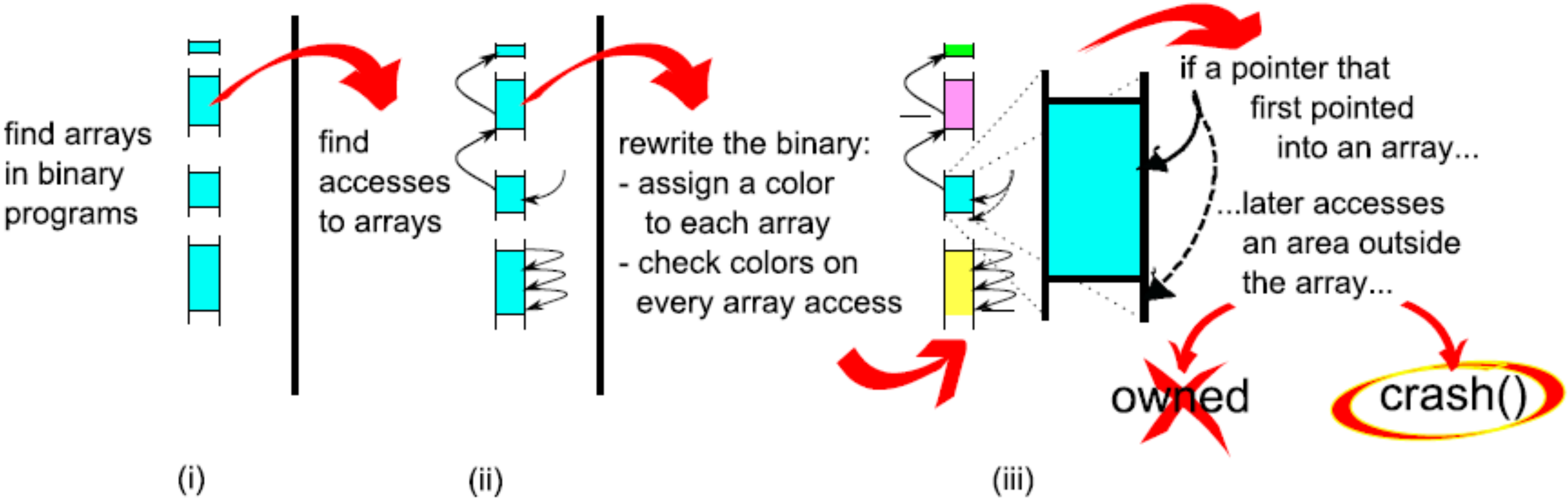
syssec

no source

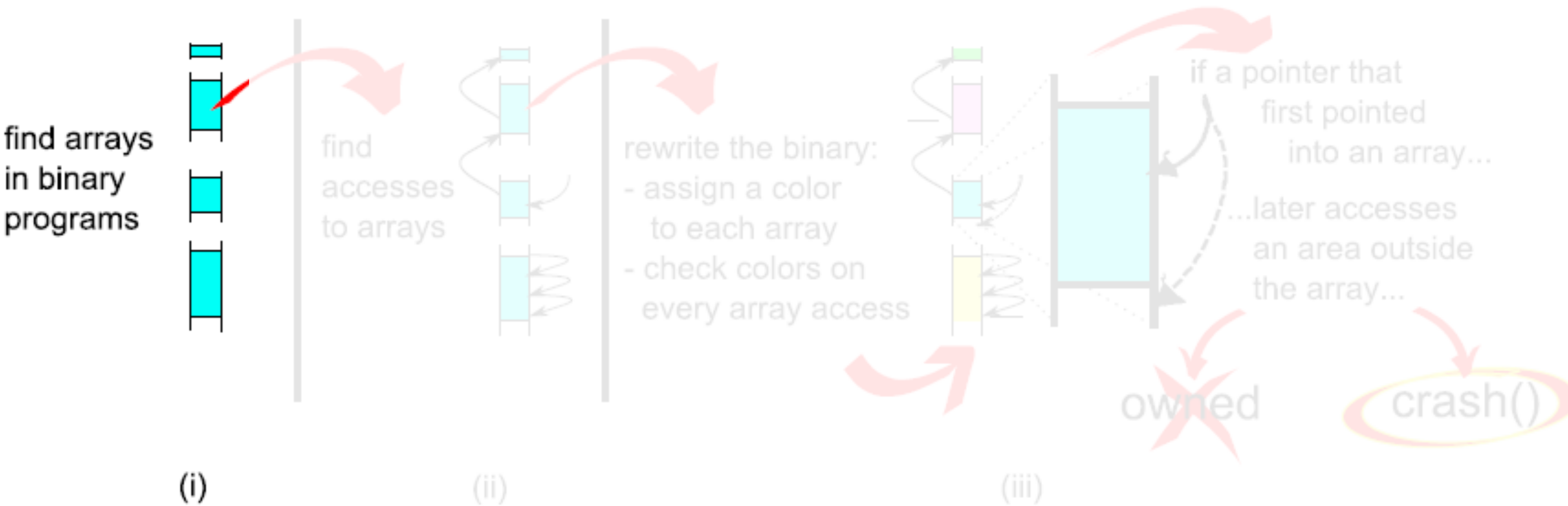
no symbols

no clue?

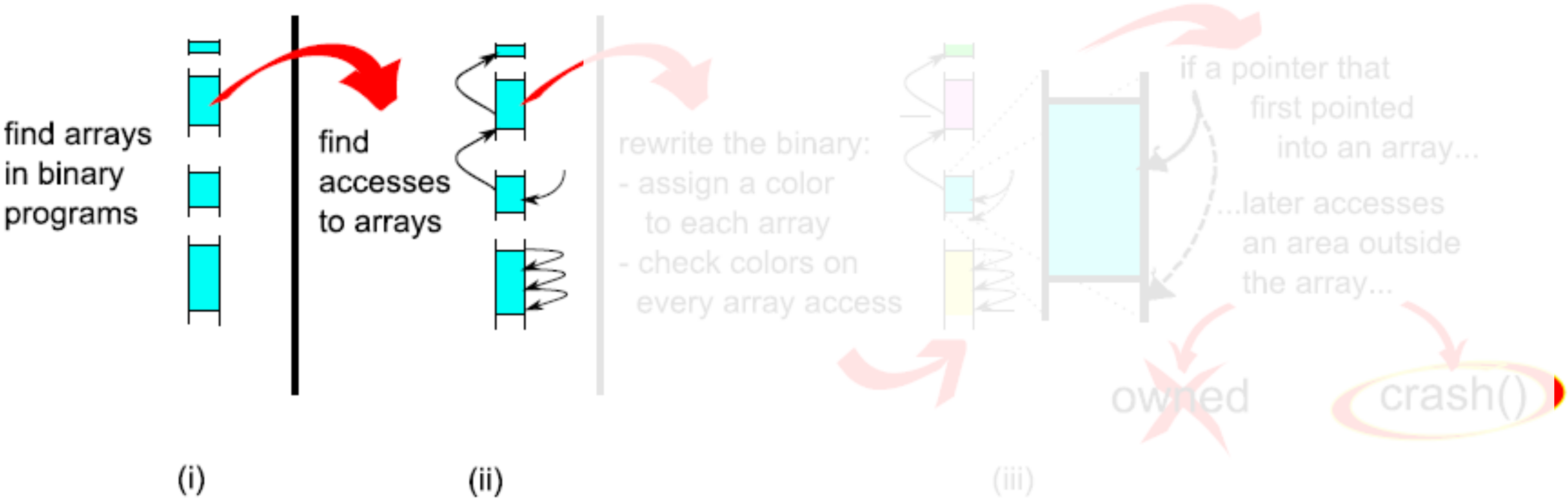
In a nutshell…



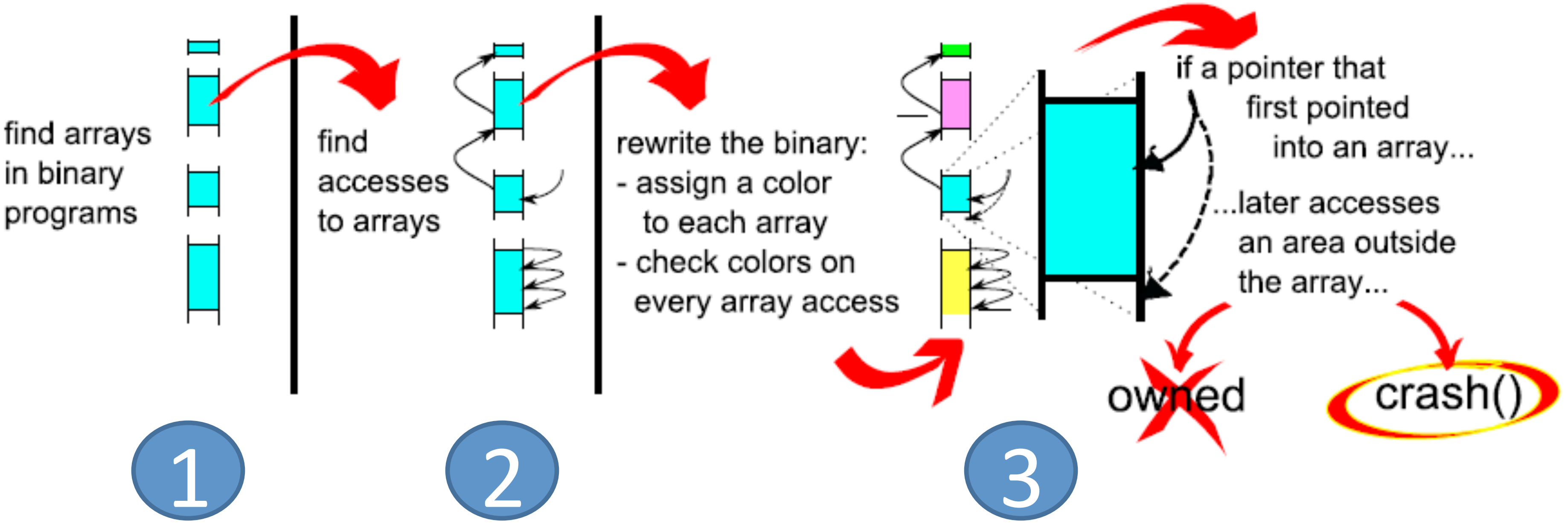
In a nutshell…



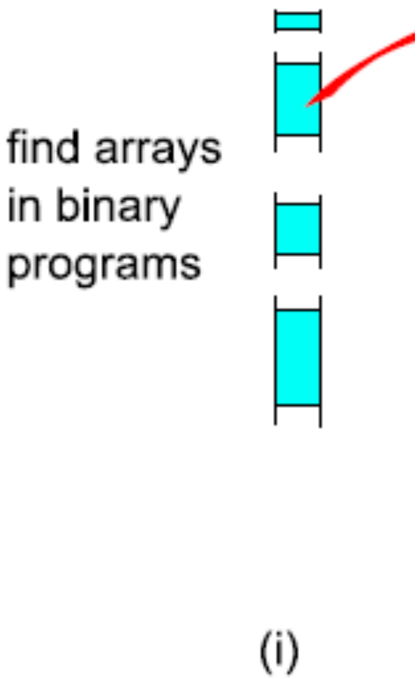
In a nutshell...



In a nutshell…



Step 1: extract the arrays



Two possibilities

- symbol tables
- stripped

➔ reverse engineering

let’s assume the latter

Problem

1

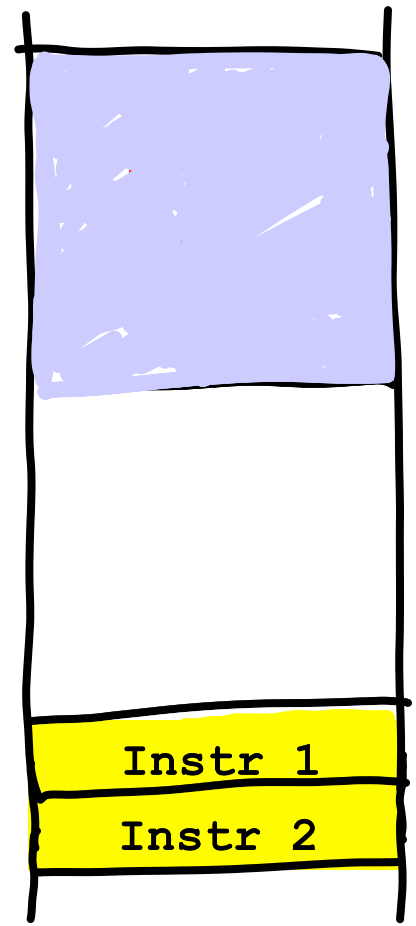
Why is it difficult?

```c
struct employee {
        char name[128];
        int year;
        int month;
        int day
};

struct employee e;
e.year = 2010;
```

1

Why is it difficult?

```c
struct employee {
        char name[128];
        int year;
        int month;
        int day
};

struct employee e;
e.year = 2010;
```



MISSING

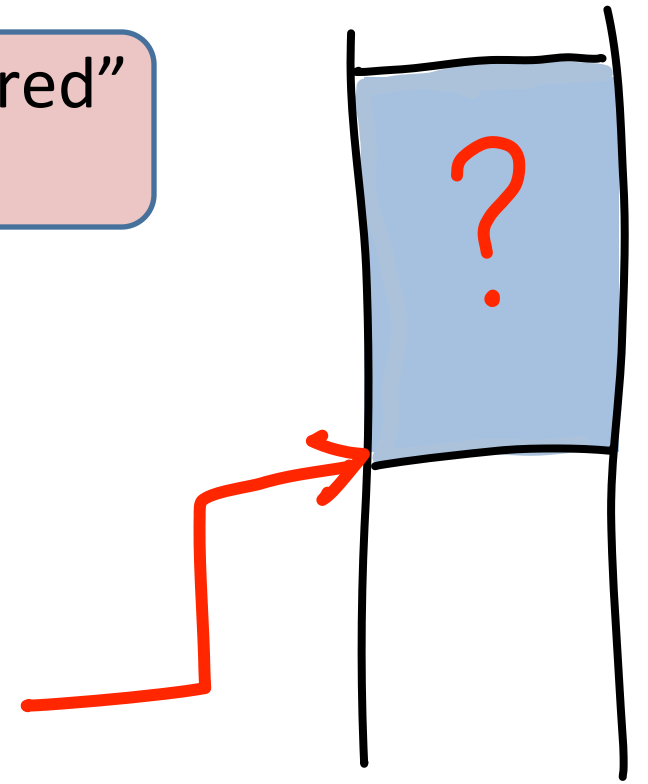
- Data structures

1

Data structures: key insight

Yes, data is "apparently unstructured"

But usage is not!



vrije Universiteit

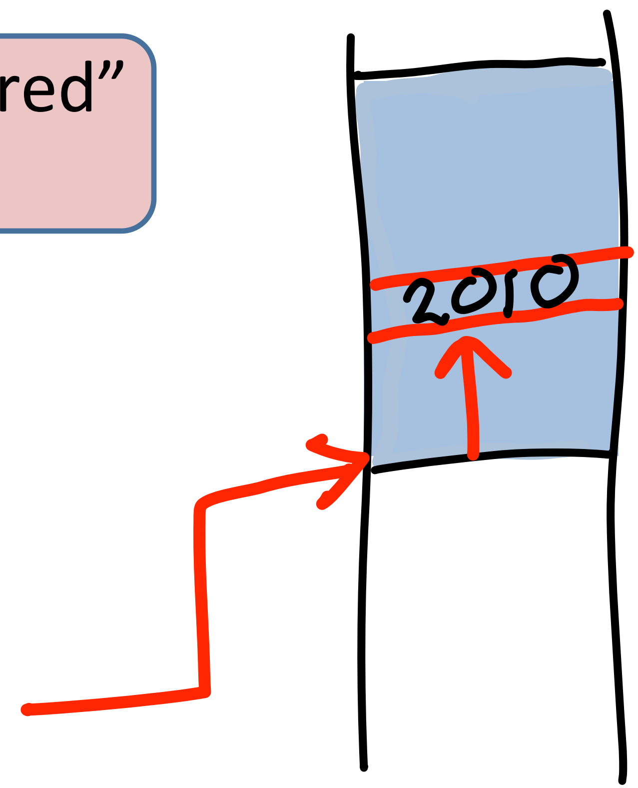
syssec

1

Data structures: key insight

Yes, data is "apparently unstructured"

But usage is not!



vrije Universiteit

syssec

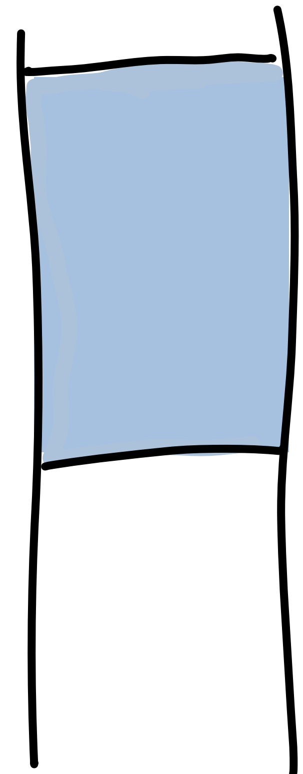
1

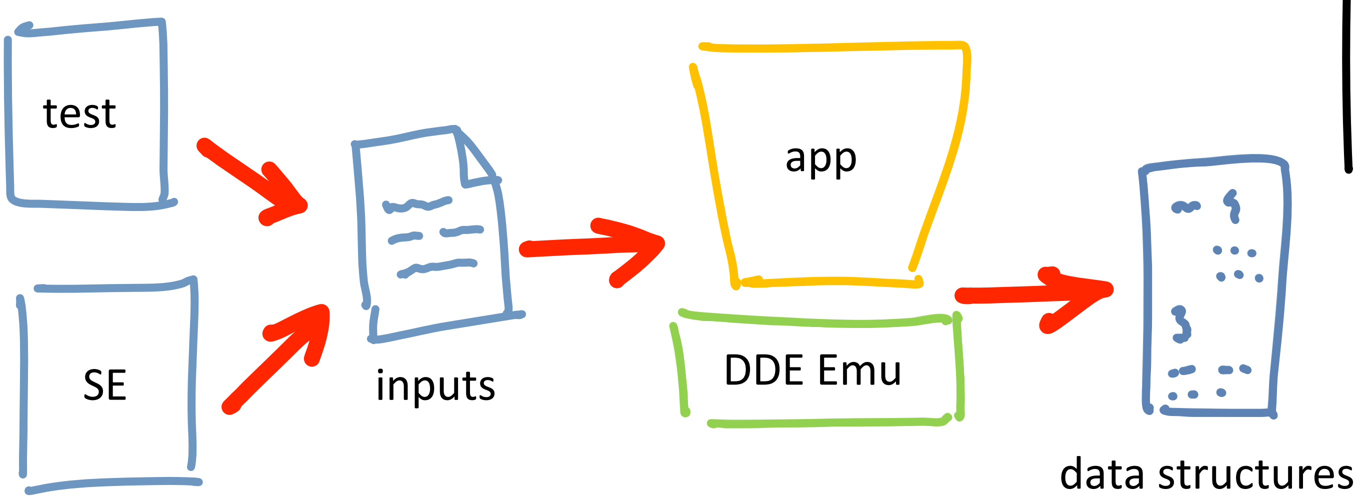
Data structures: key insight

Yes, data is “apparently unstructured”

But usage is not!

Analyse dynamically



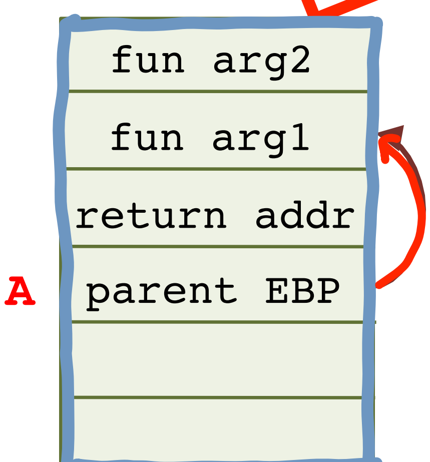


1

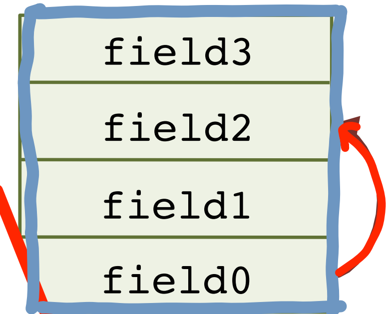
Intuition

- Observe how memory is *used* at runtime to detect data structures
- E.g., if A is a pointer…

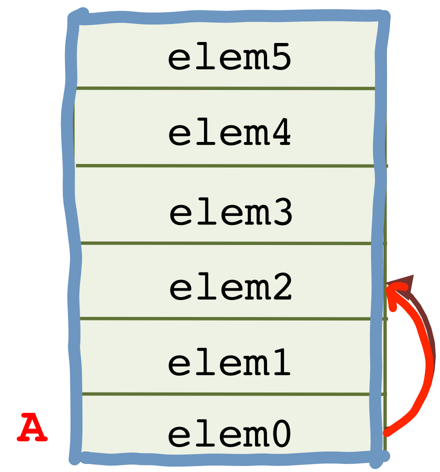
1. and A is a functi… then *(A + 8) is p… …on argument



2. and A is an address of a structure, then *(A + 8) is perhaps a field in this structure



3. and A is an address of an array, then *(A + 8) is perhaps an element of this array



Track pointers

1

Approach

- Track pointers
  - find root pointers
  - track how pointers derive from each other
    - for any address B=A+8, we need to know A.
- Challenges:
  - missing base pointers
    - for instance, a field of a `struct` on the stack may be updated using EBP rather than a pointer to the struct
  - multiple base pointers
    - e.g., normal access and `memset()`

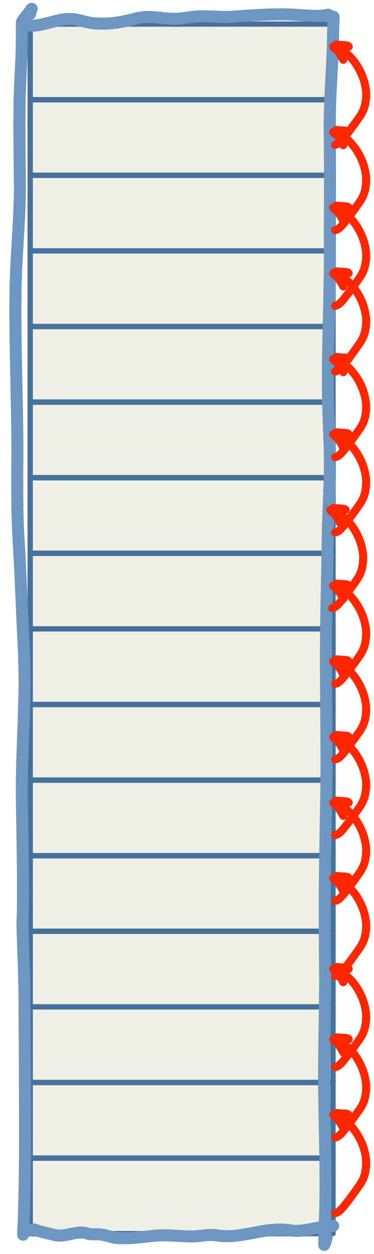
vrije Universiteit

syssec

1

Arrays are tricky

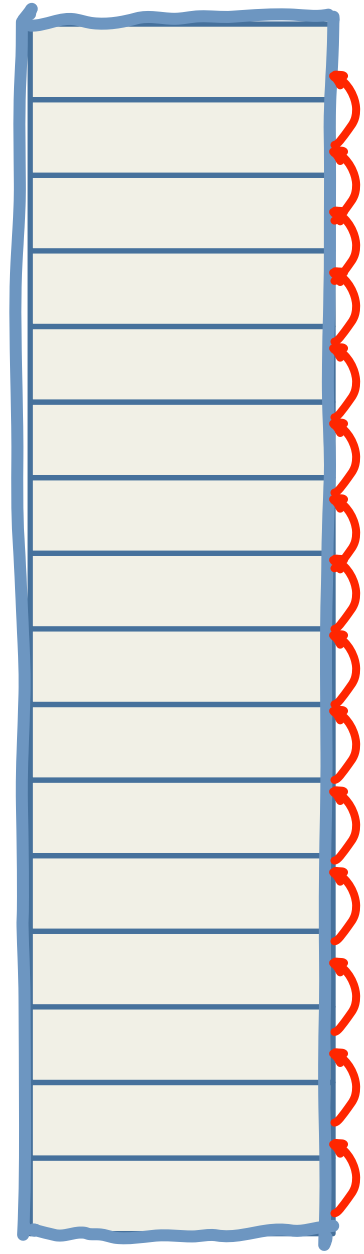
- Detection:
  - looks for chains of accesses in a loop



1

Arrays are tricky

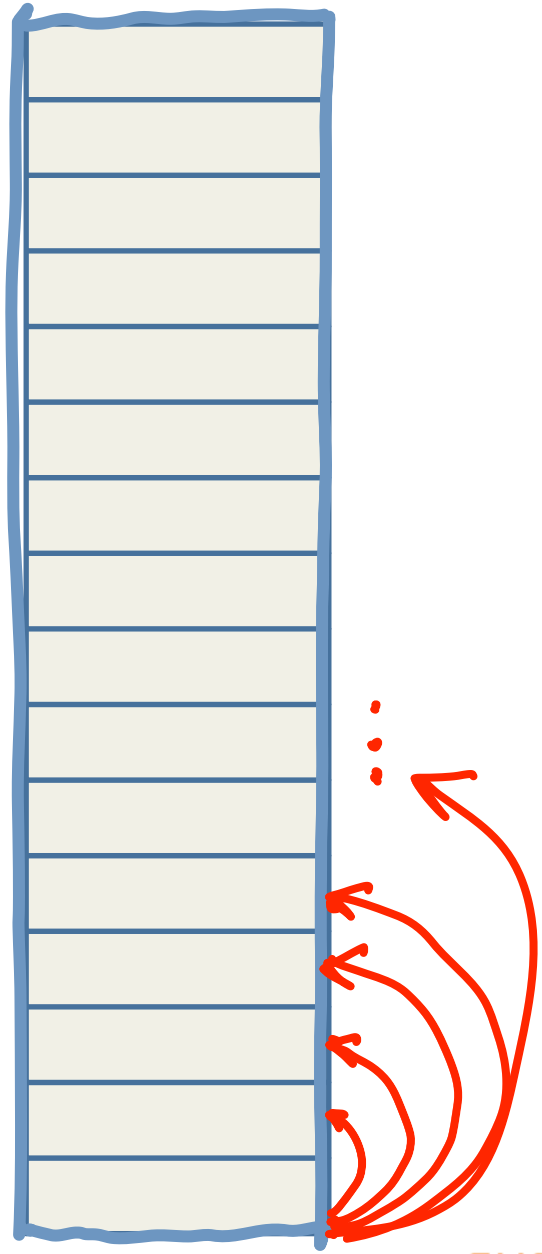
- Detection:
  - looks for chains of accesses in a loop



1

Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop



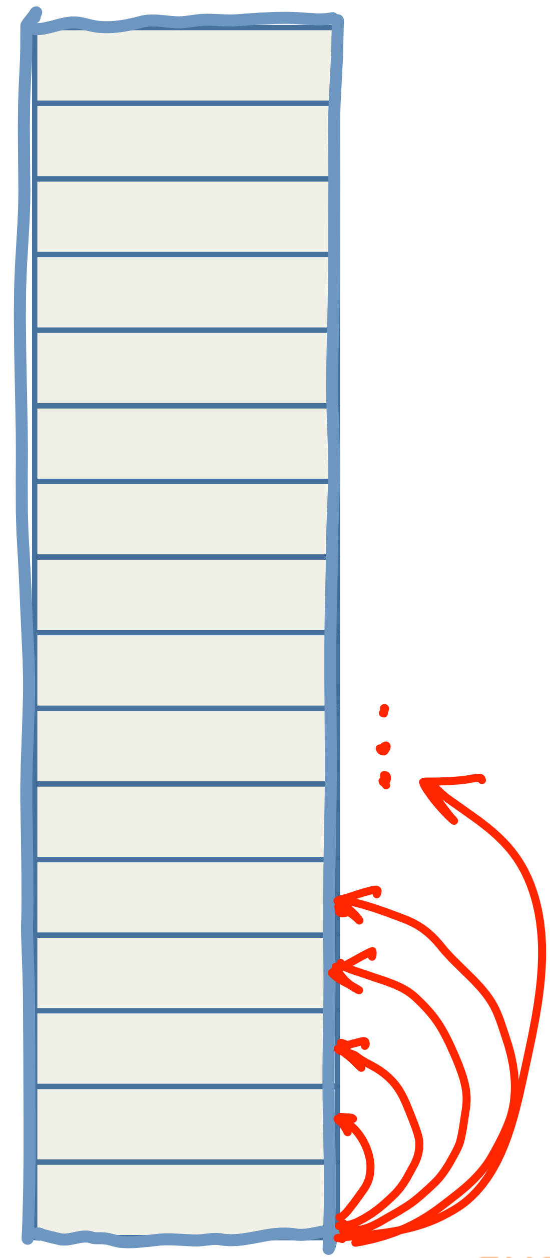
vrije Universiteit

syssec

1

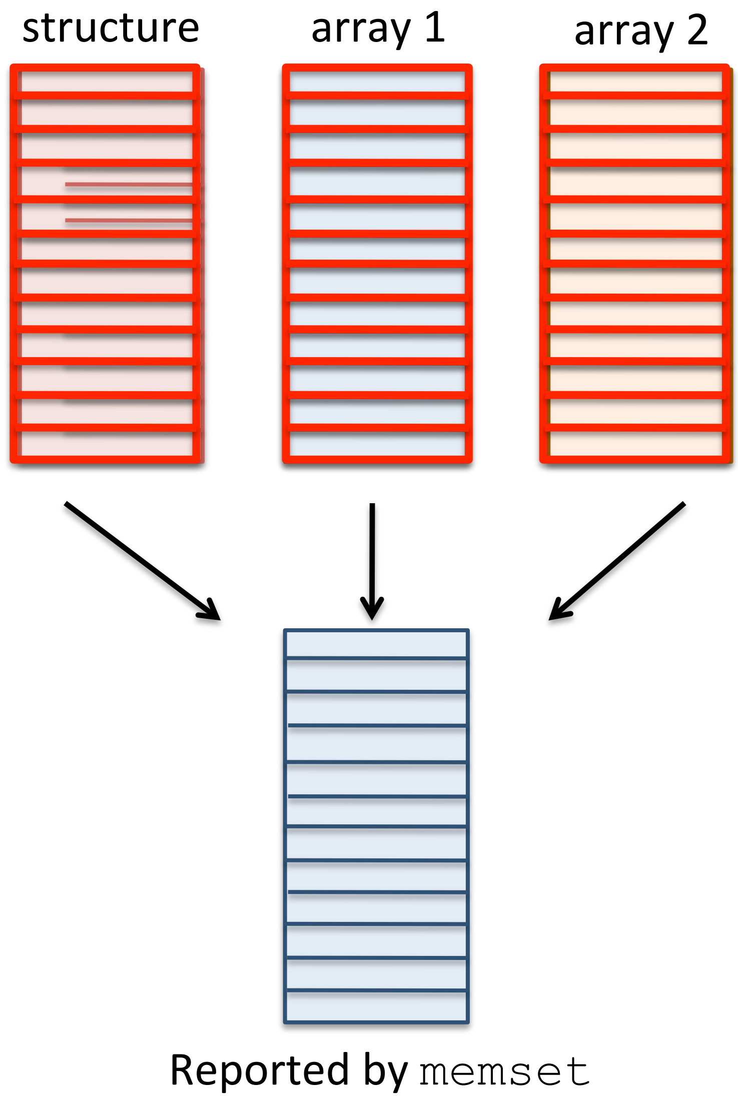
Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop
  - and sets of accesses with same base in linear space



Interesting challenges

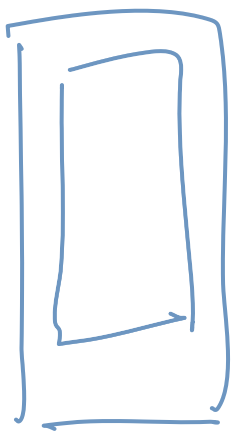
- Example:
  - Decide which accesses are relevant
    - Problems caused by e.g., `memset`-like functions



1

Further Challenges

- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements

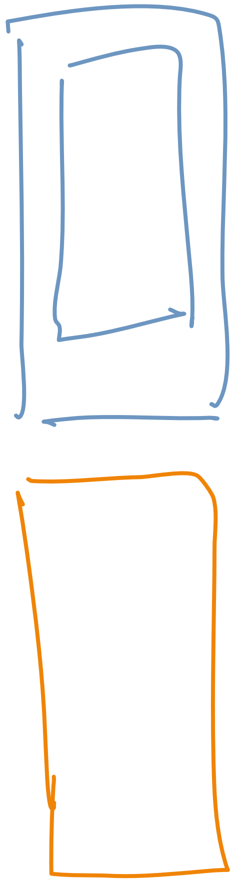


vrije Universiteit

1

Further Challenges

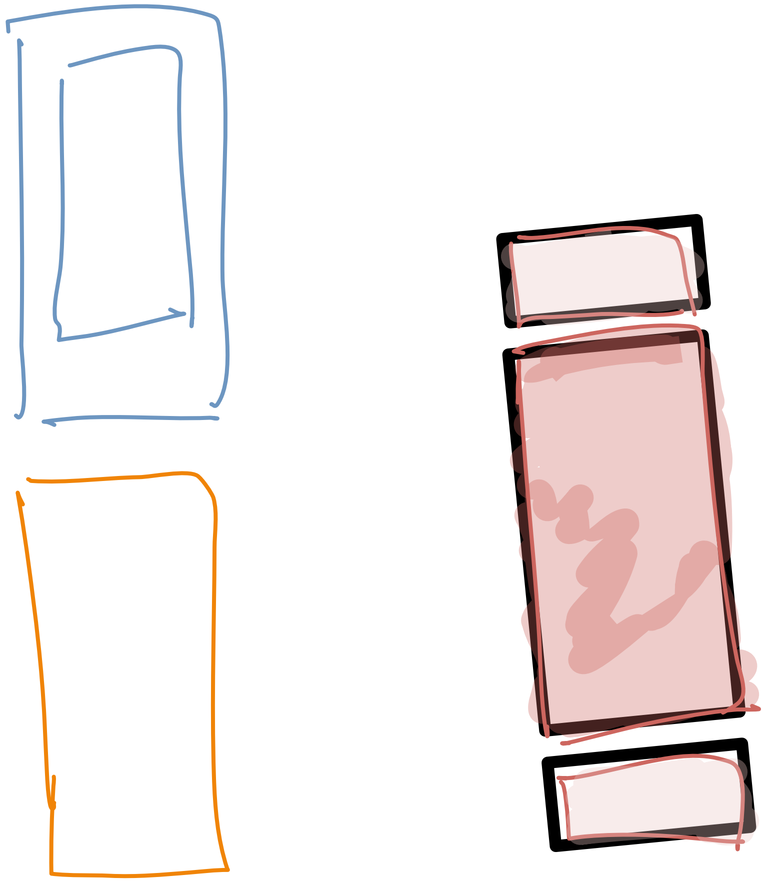
- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements



1

Further Challenges

- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements



1

Final mapping

- map access patterns to data structures
  - static memory : on program exit
  - heap memory : on free
  - stack frames : on return

vrije Universiteit

syssec

1

Also: not everything is hidden

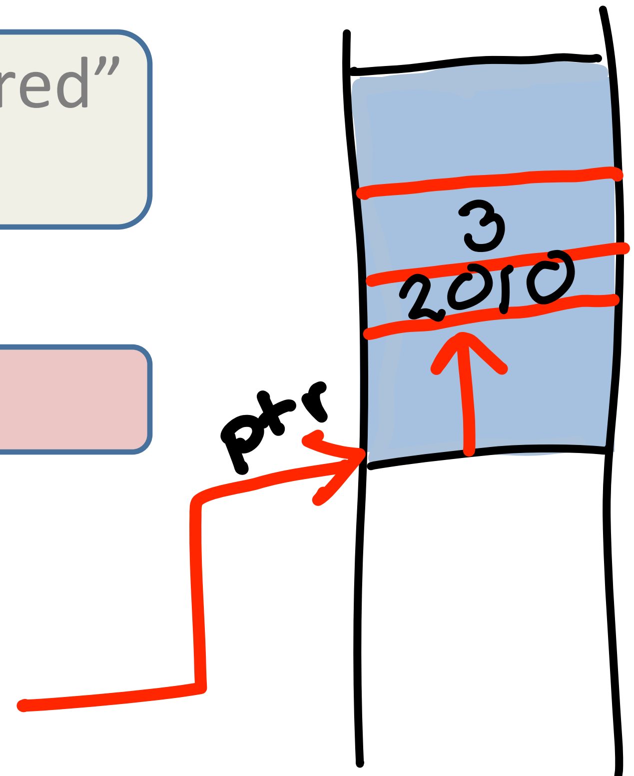
1

Key insight 2

Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics



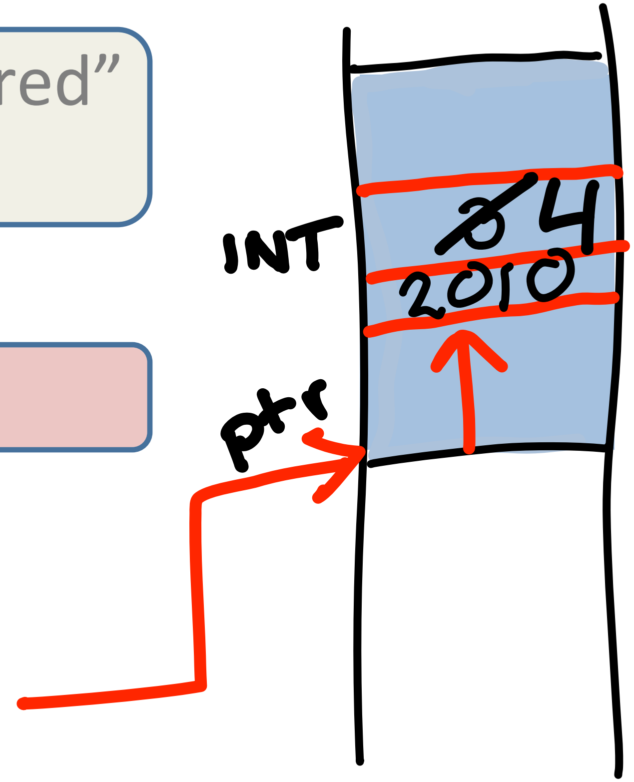
1

Key insight 2

Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics



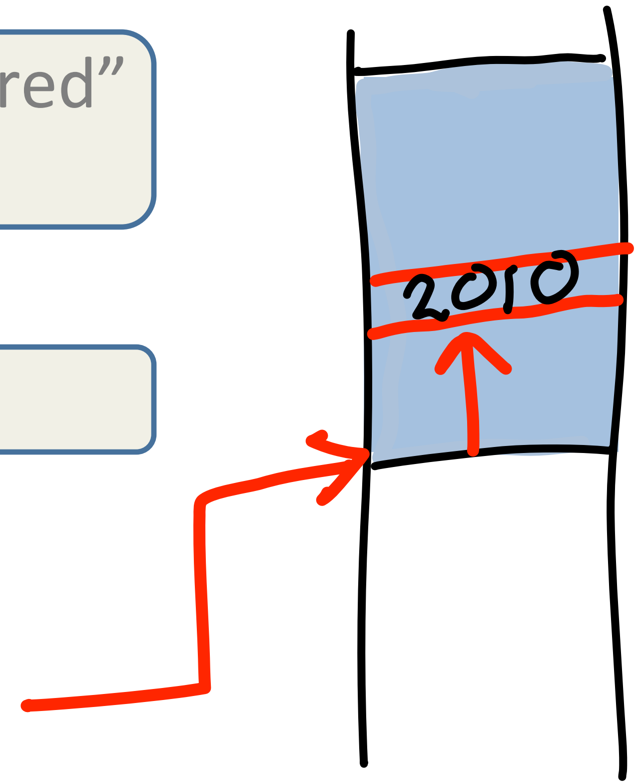
1

Semantics: key insights

Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics



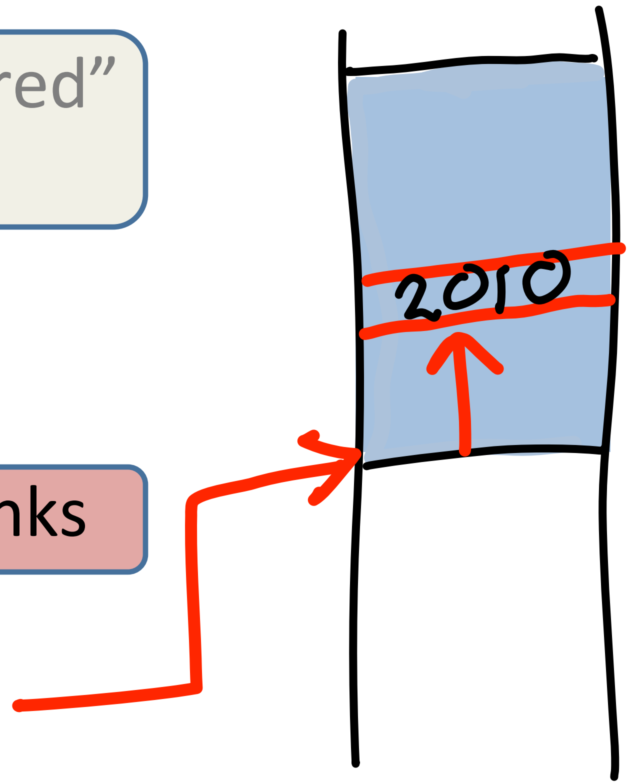
1

Key insight 3

Yes, data is "apparently unstructured"

But usage is not!

Propagate types from sources + sinks



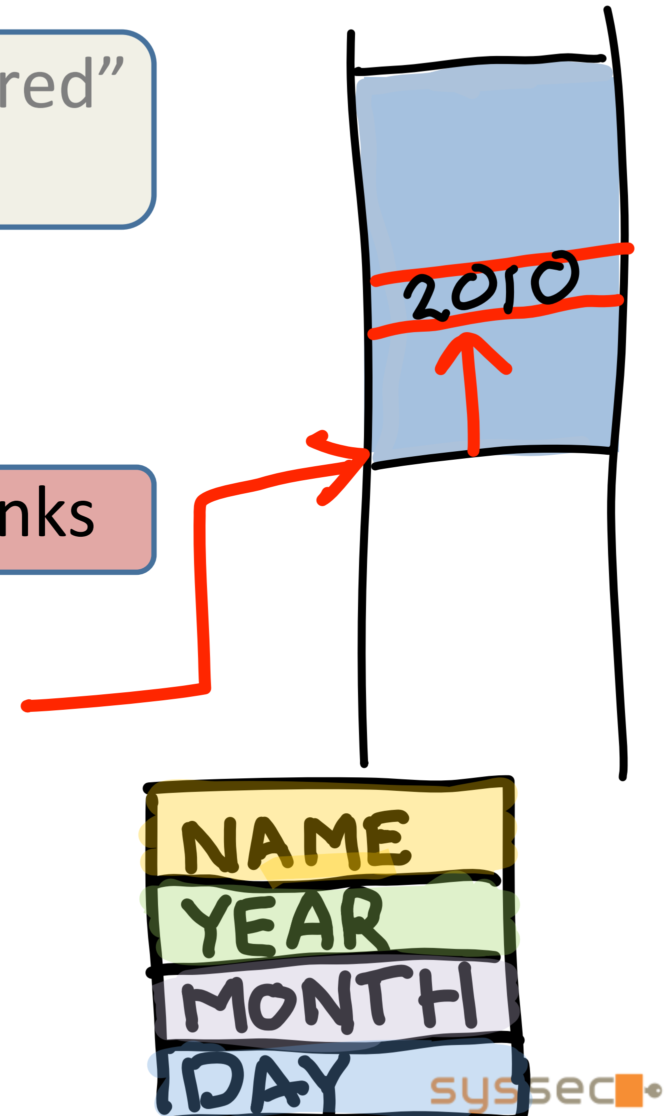
1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



vrije Universiteit

syssec

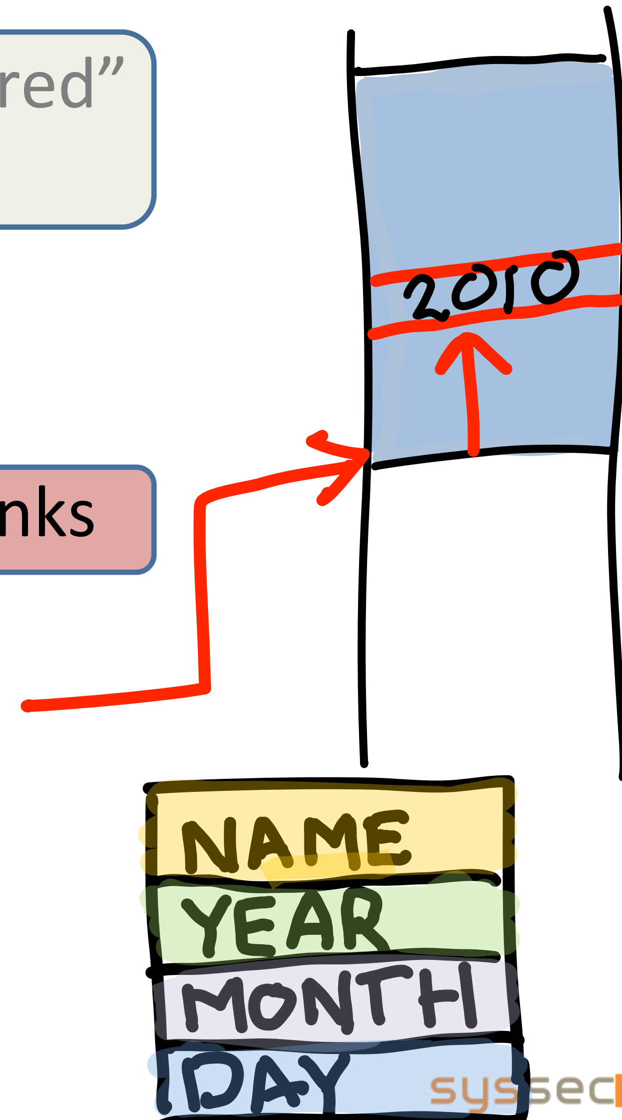
1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



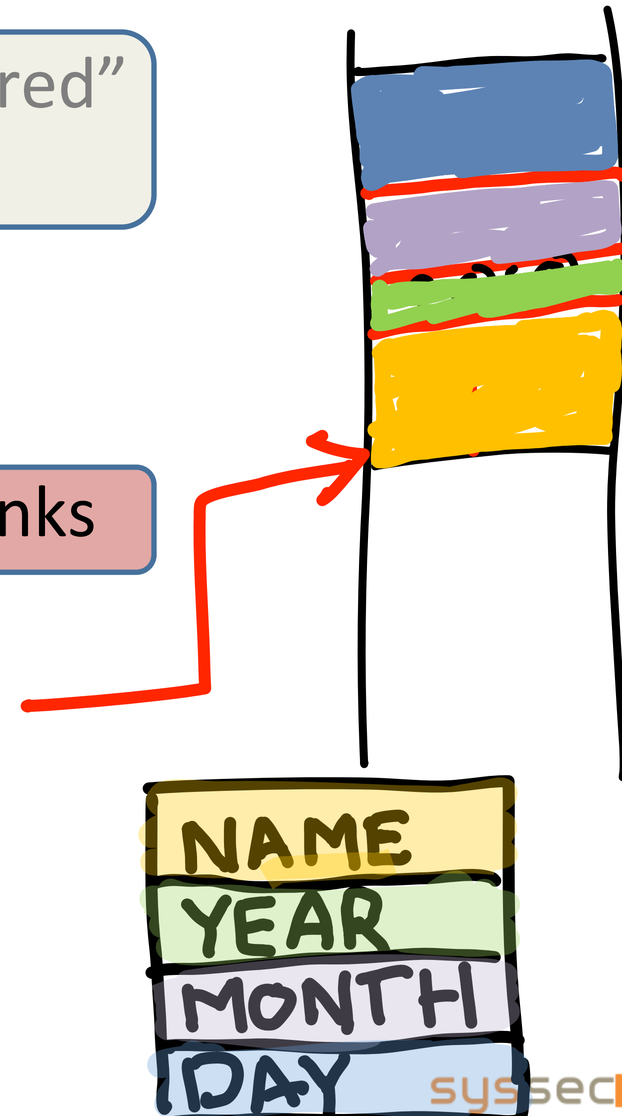
1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



vrije Universiteit

syssec

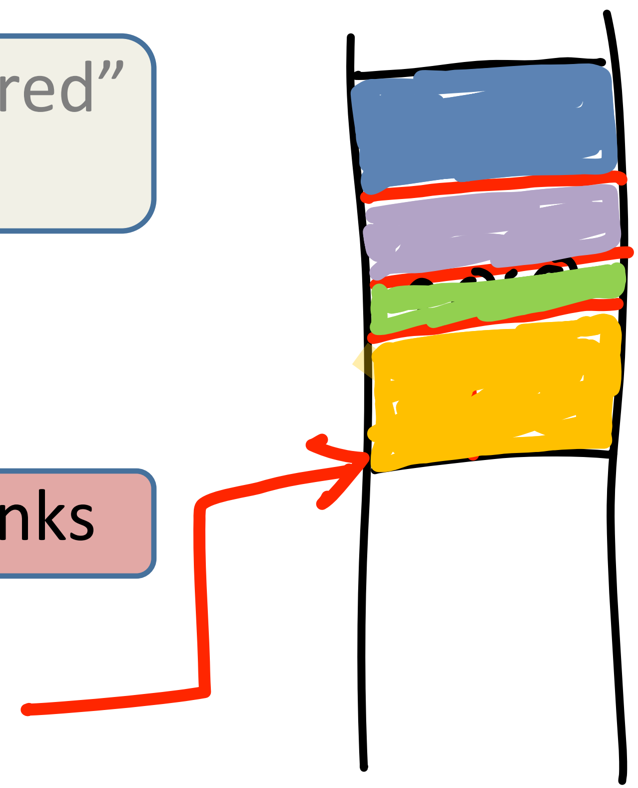
1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



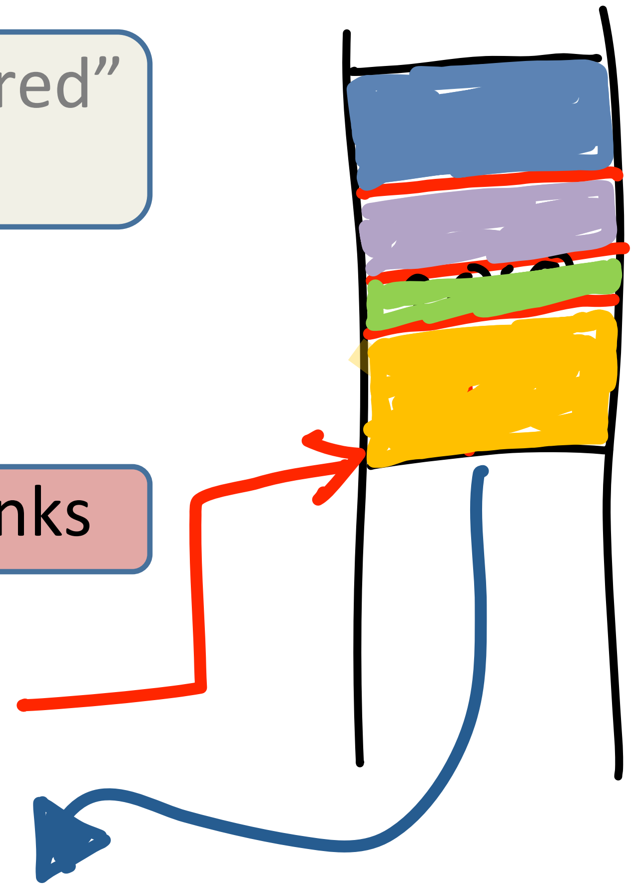
1

Key insight 3

Yes, data is "apparently unstructured"

But usage is not!

Propagate types from sources + sinks



open ("Herbert.doc", R_ONLY)

vrije Universiteit

syssec

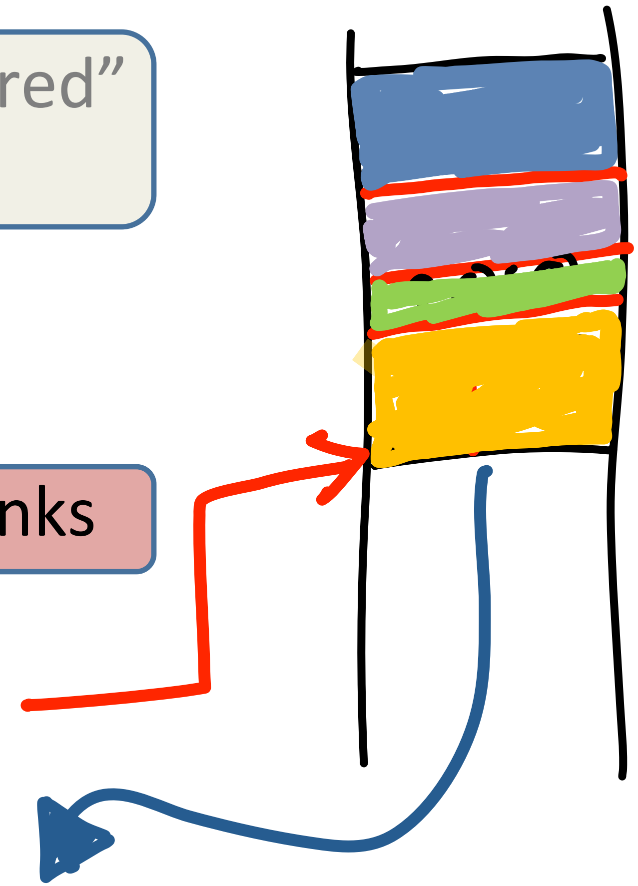
1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



open (“Herbert.doc”, R_ONLY)

1

Results

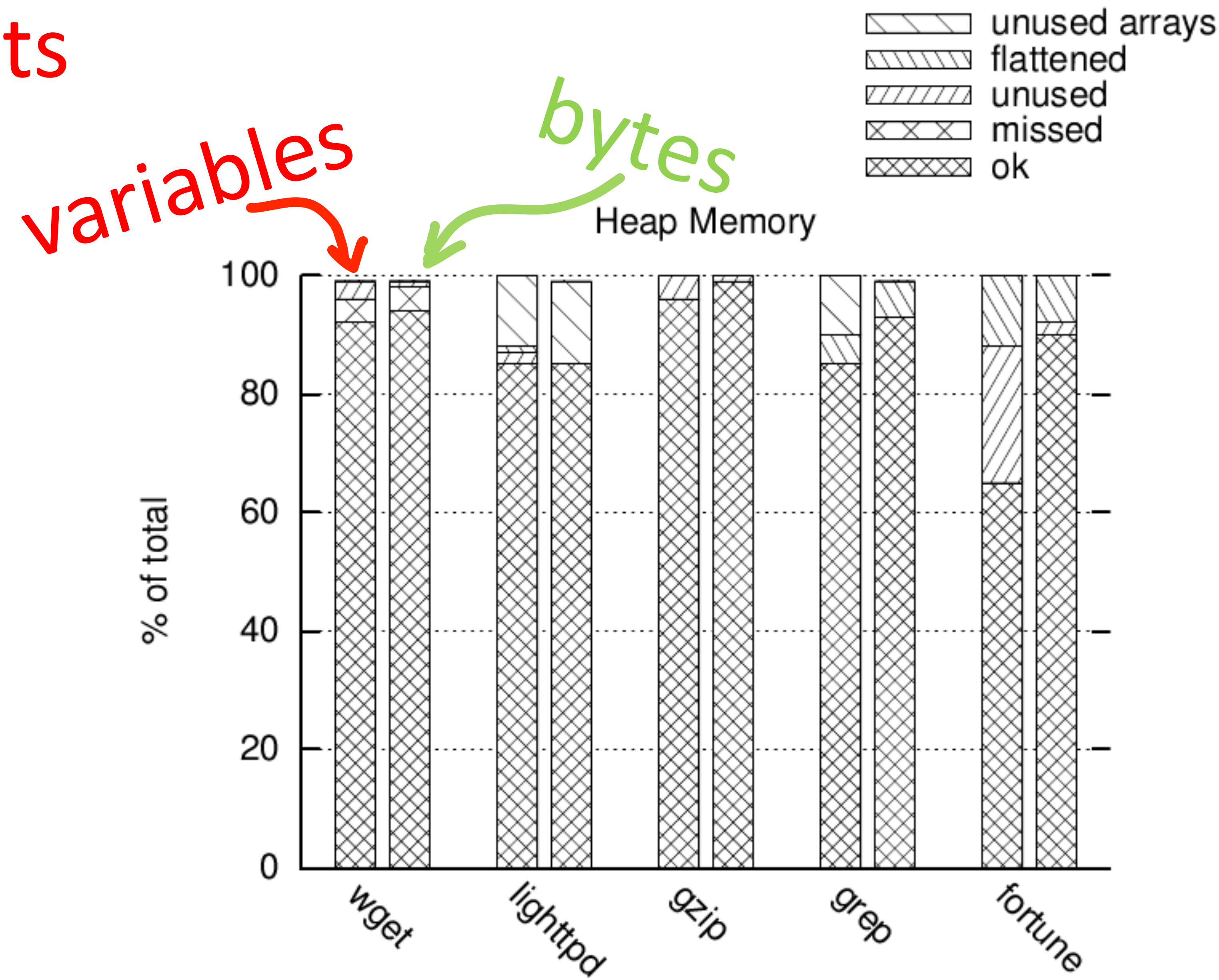
vrije Universiteit

syssec

1

Results

Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



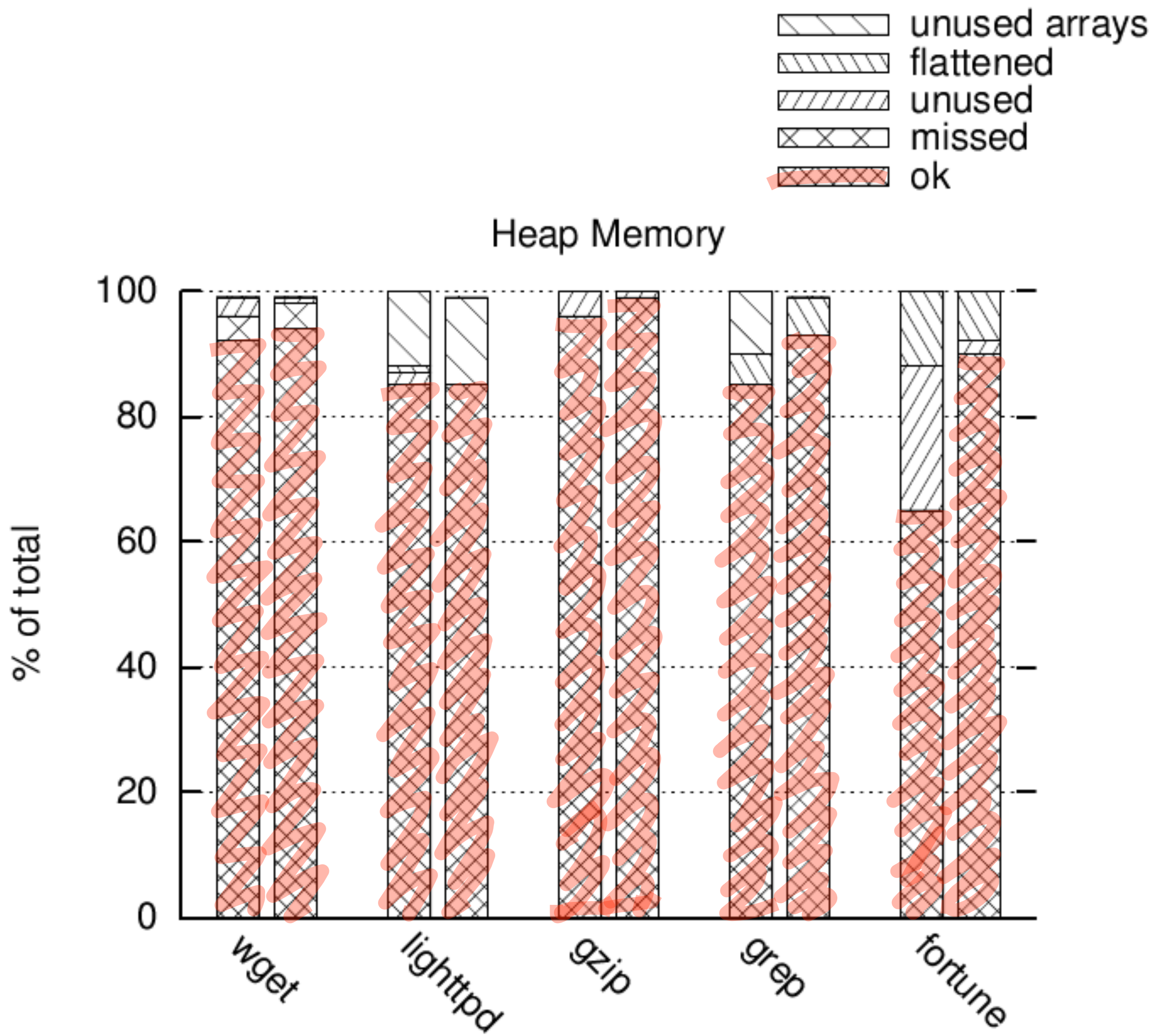
vrije Universiteit

syssec

1

Results

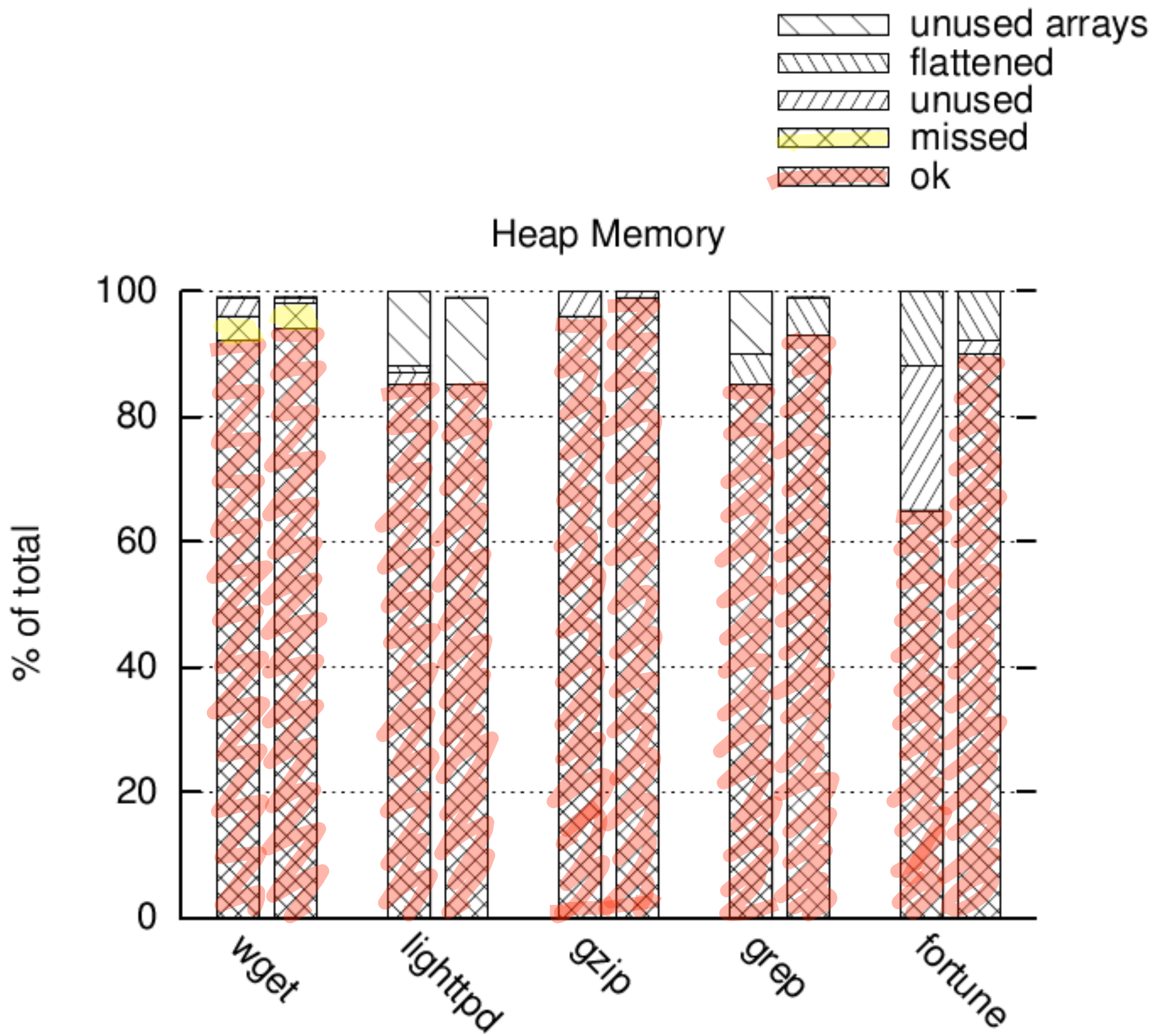
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K

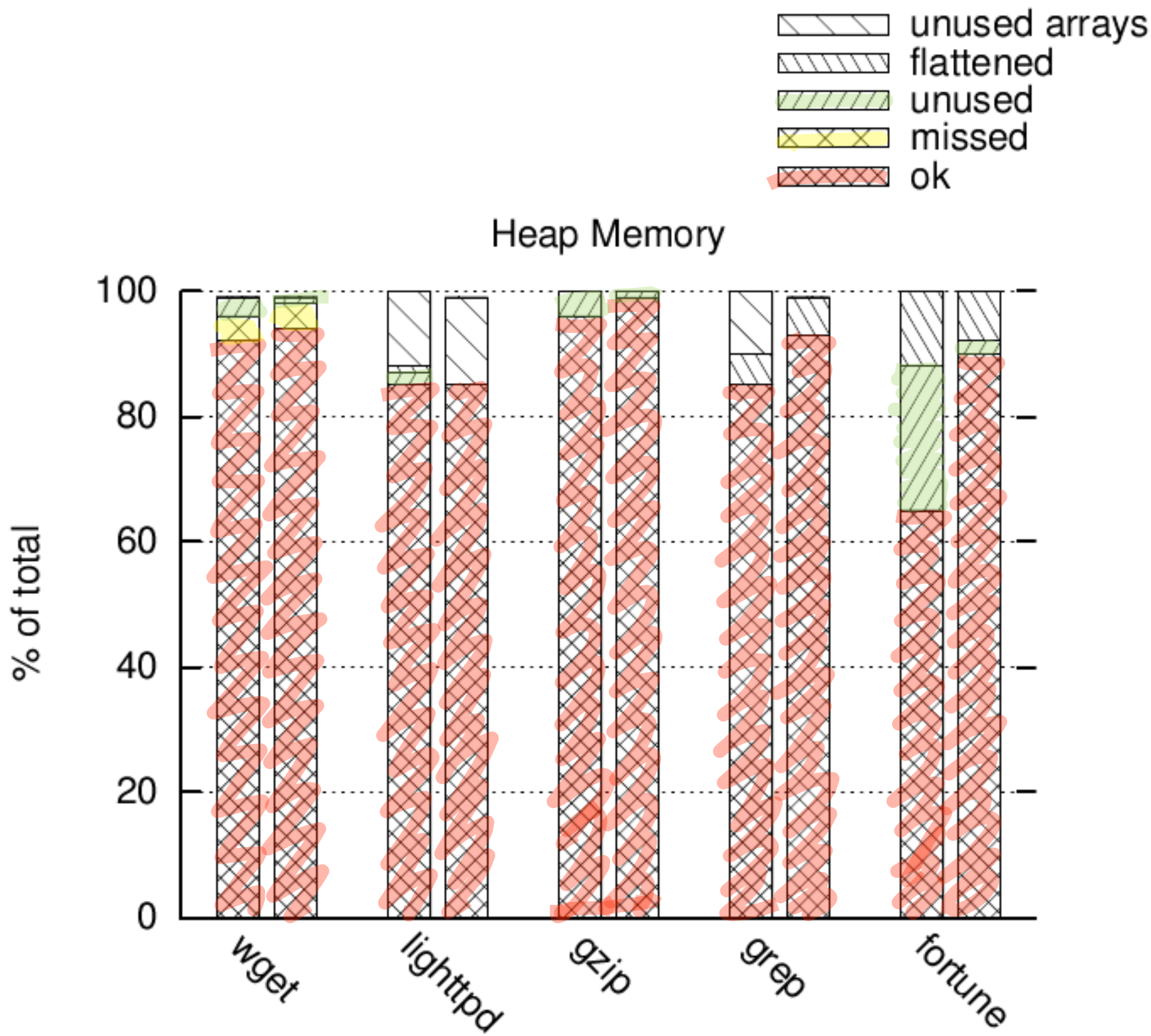


vrije Universiteit

syssec

Results

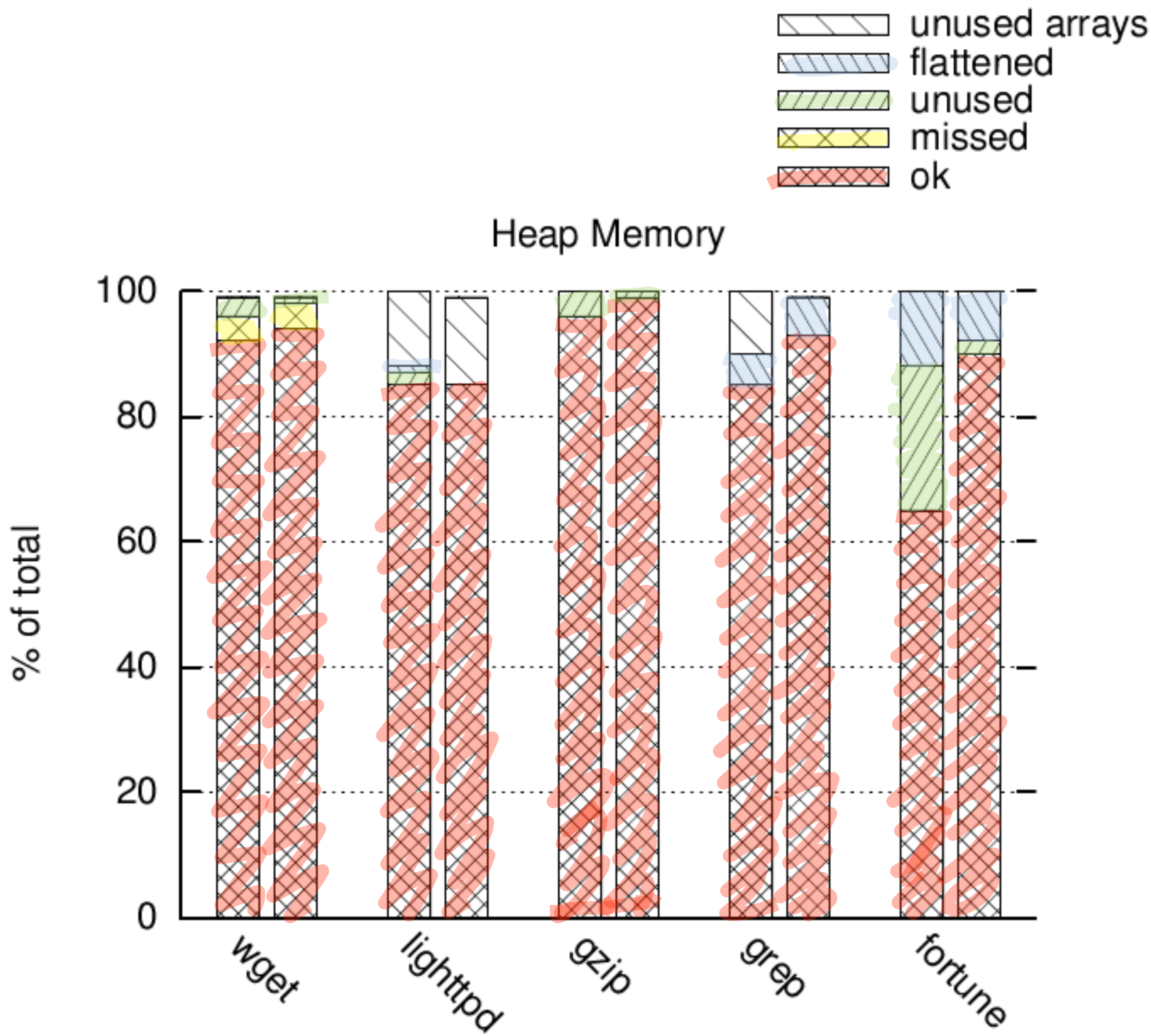
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

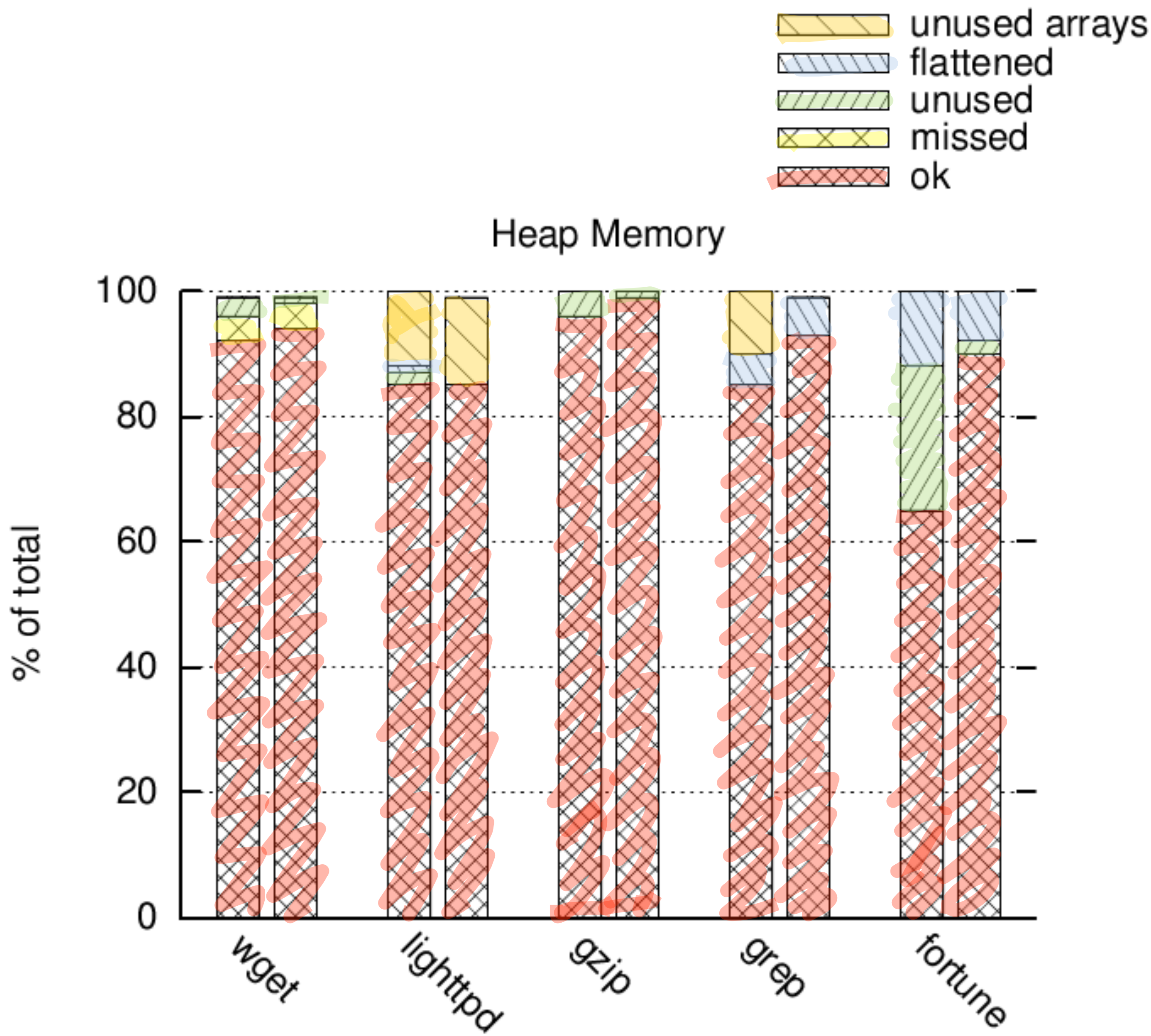
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



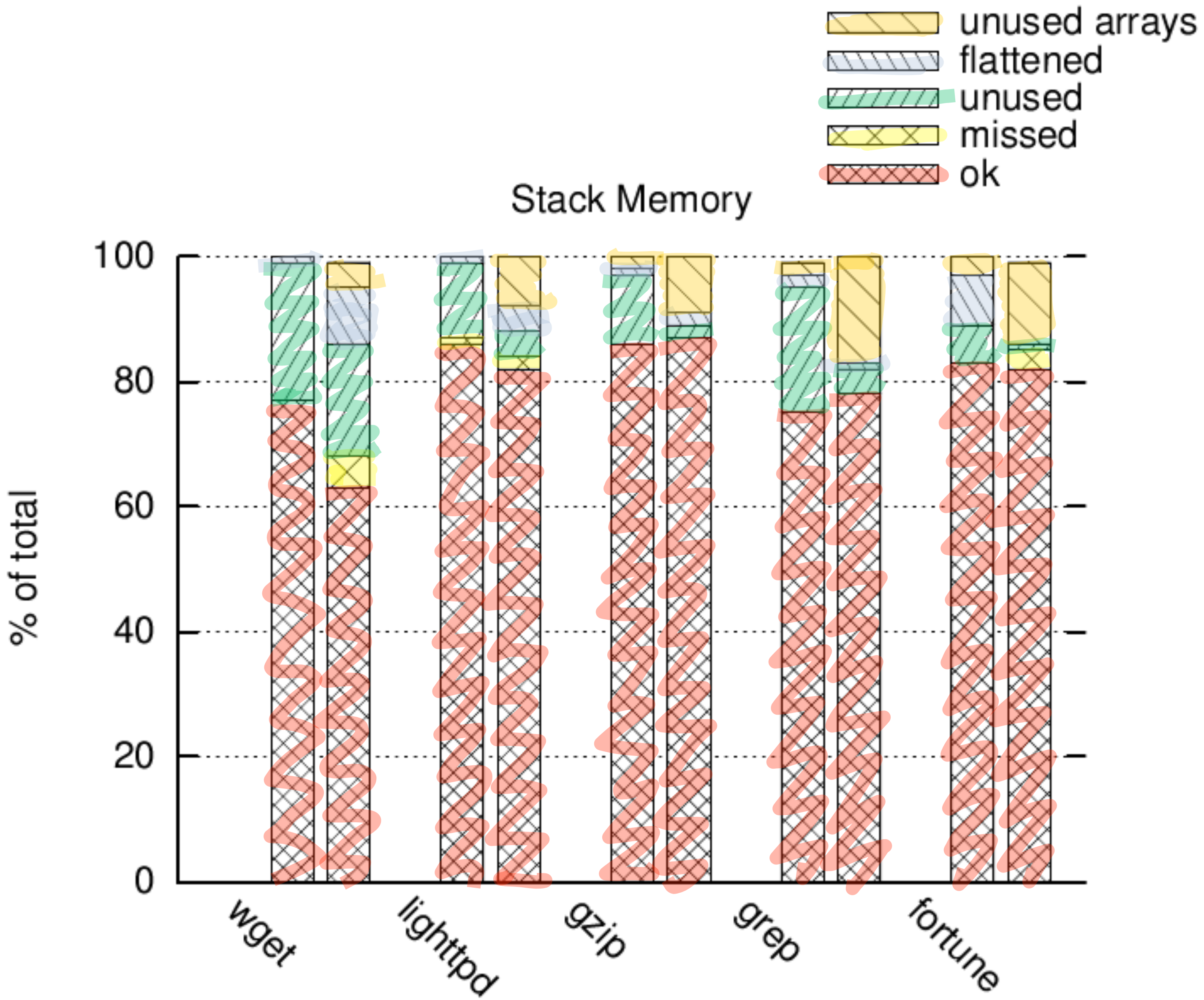
vrije Universiteit

syssec

1

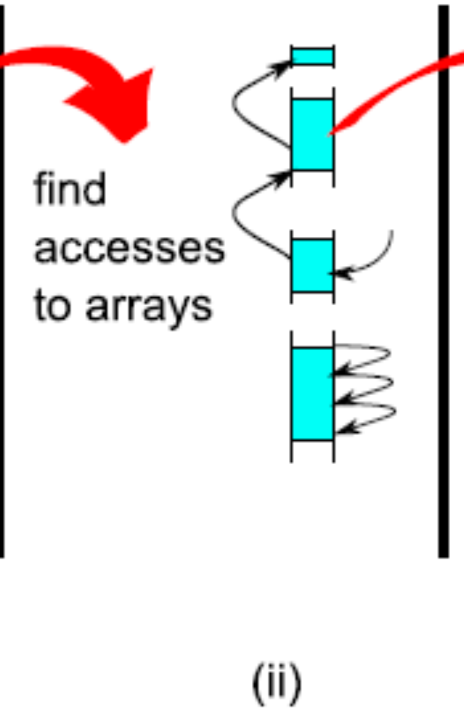
Results

Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



Demo?

Step 2: find array accesses



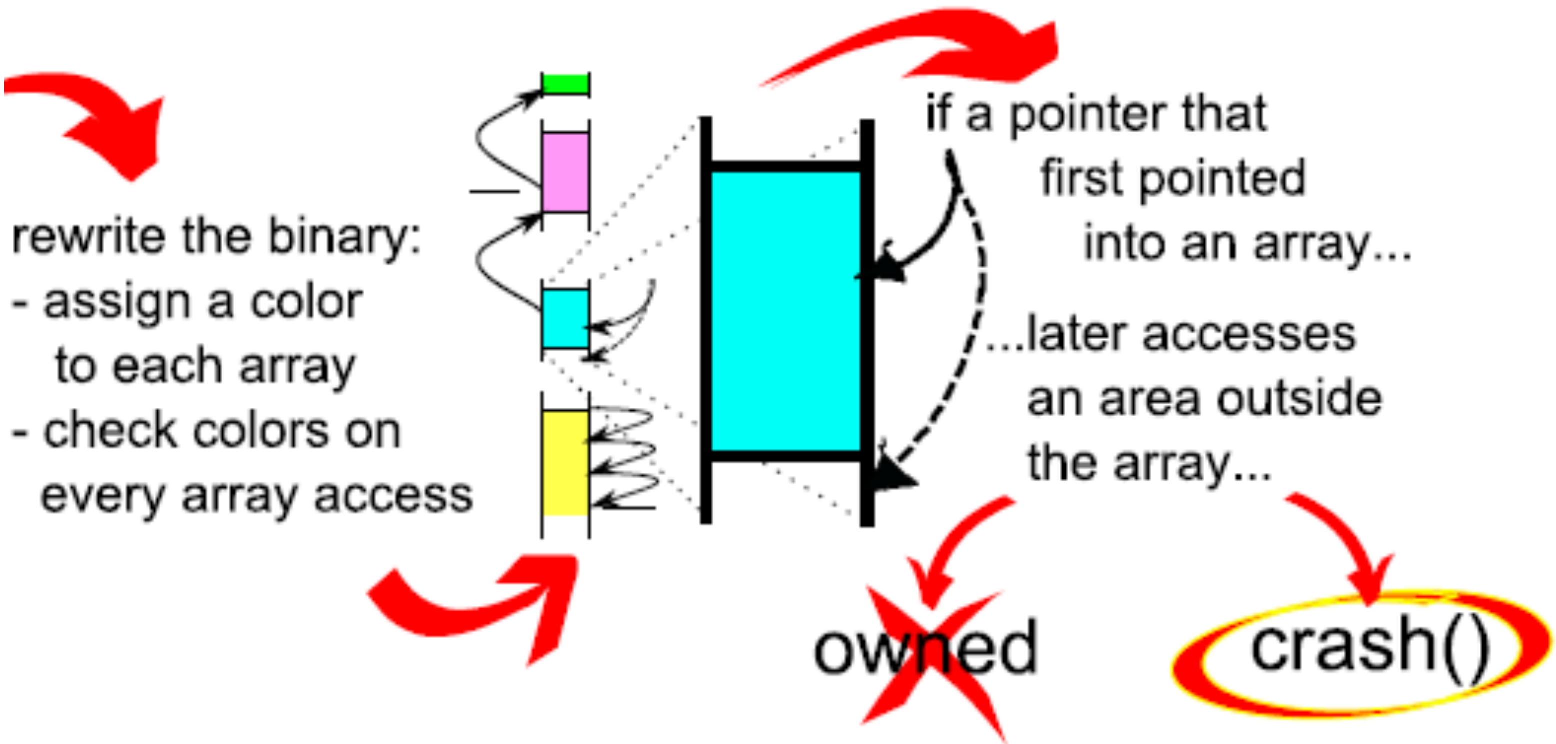
In principle: very simple

- detect array accesses at runtime
- remember the instructions

Note: not complete

3

Step 3: rewrite the binary



(iii)

Two Modes

- Protect at object level (like WIT, BBC)
  - given symbols: zero false positives
- Protect at subfield granularity (like no-one else)
  - no false positives seen in practice (but no guarantees)

THIS TALK

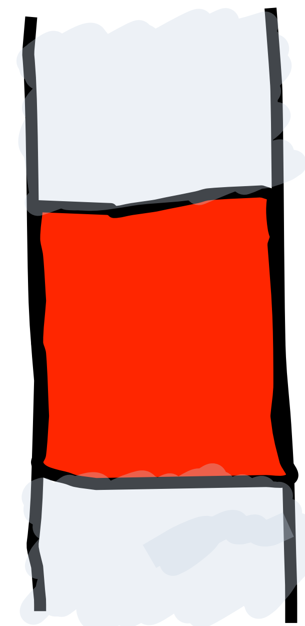
Focuses on the latter

3

A colourful protection

- give all arrays a unique colour

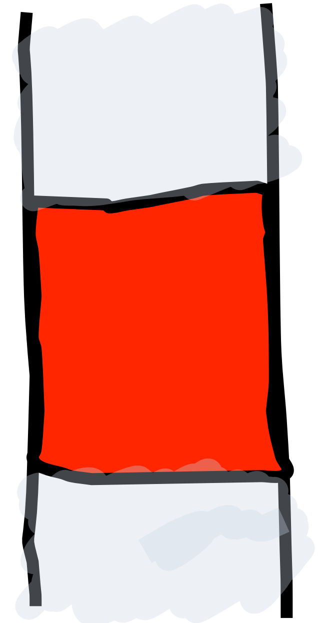
```
p = array;
ASSIGN pointer a colour
col(p) = RED
i = 0;
while(!stop)
{
   *(p + i) = 0;


   i++;
}
```



A colourful protection

- give all arrays a unique colour

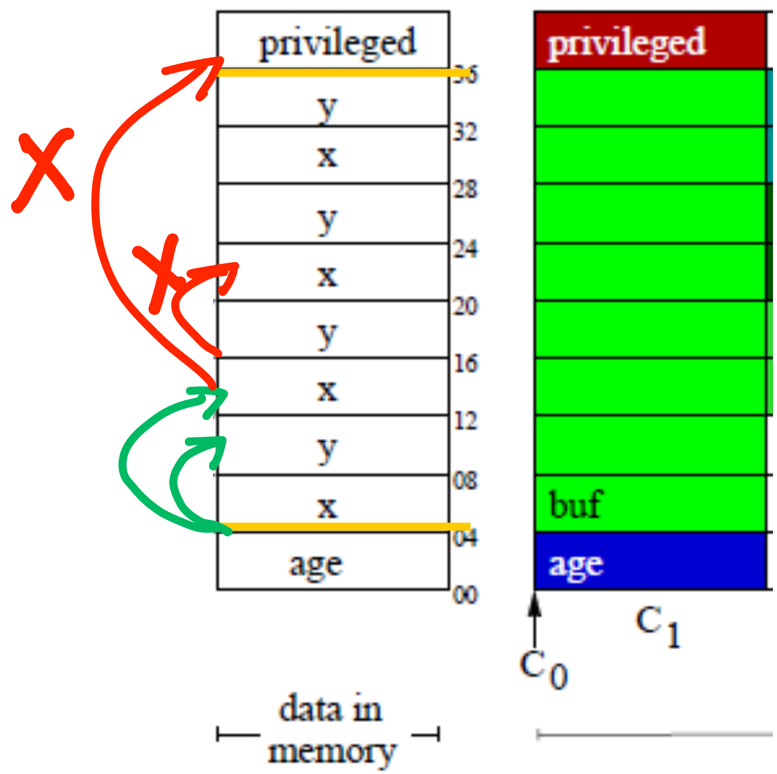
```
p = array;
ASSIGN pointer a colour
col(p) = RED
i = 0;
while(!stop)
{
   *(p + i) = 0;
   CHECK if colours match:
    mem_col(p+i) == col(p)?
   i++;
}
```



3

Reality requires subtle shades

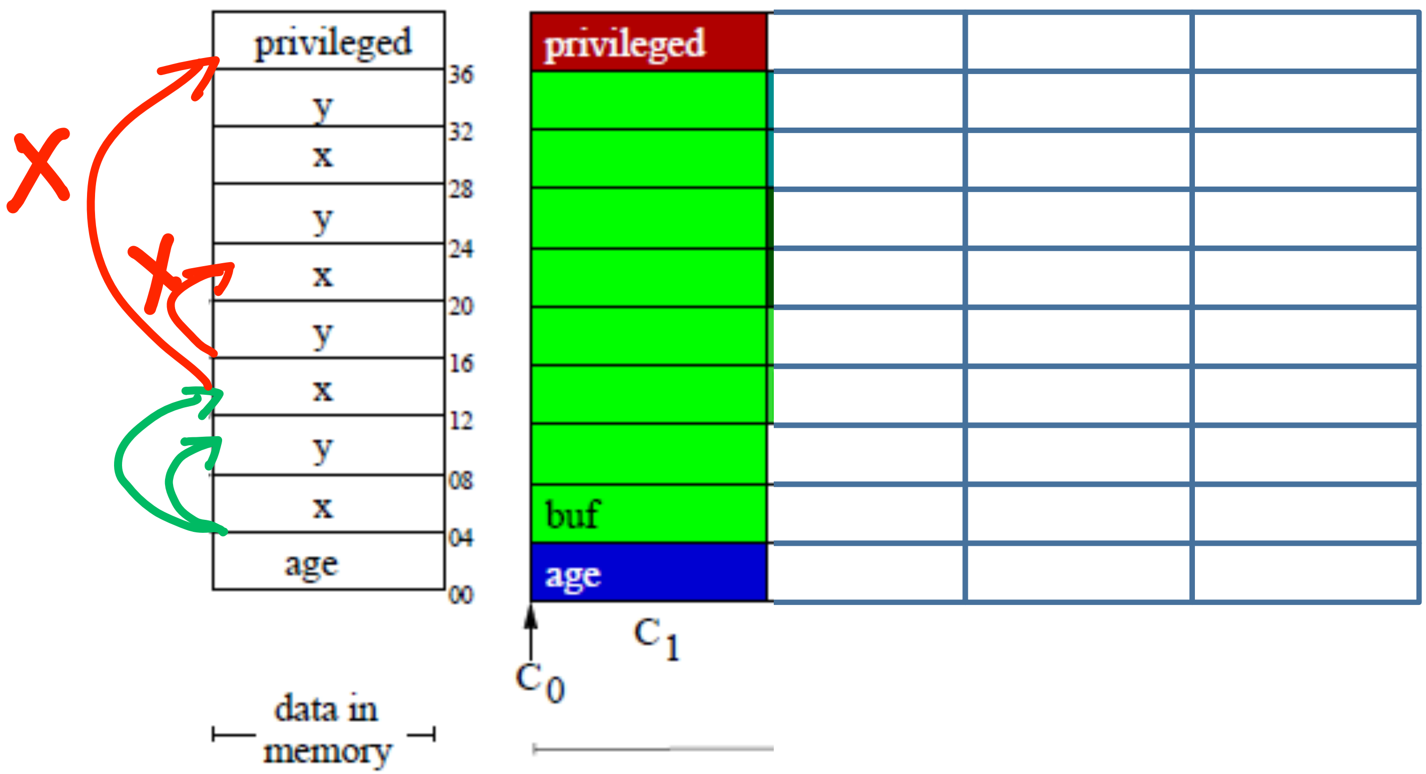
```
typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int  age;
    pair_t buf[4];
    int  privileged;
};
```



3

Reality requires subtle shades

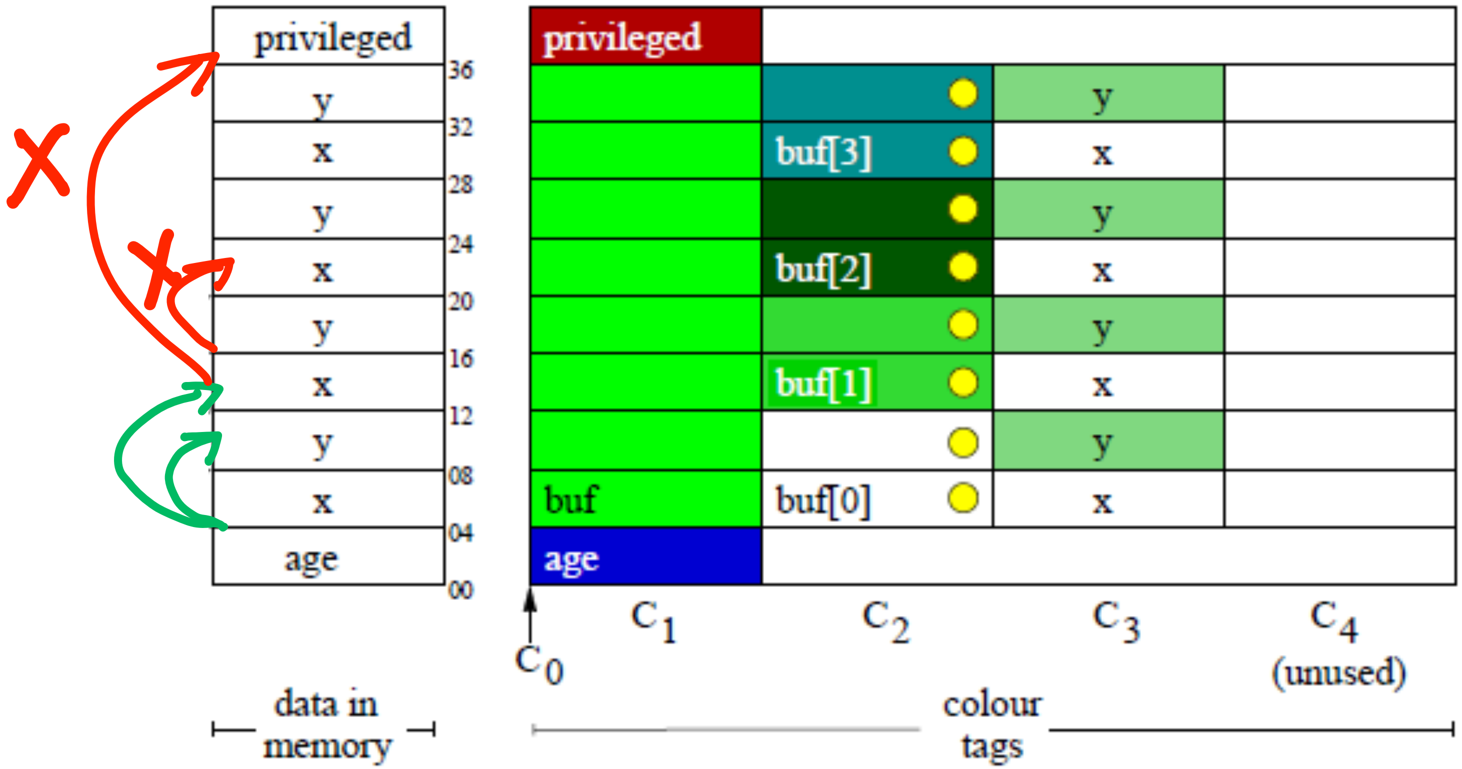
```
typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int  age;
    pair_t buf[4];
    int  privileged;
};
```



3

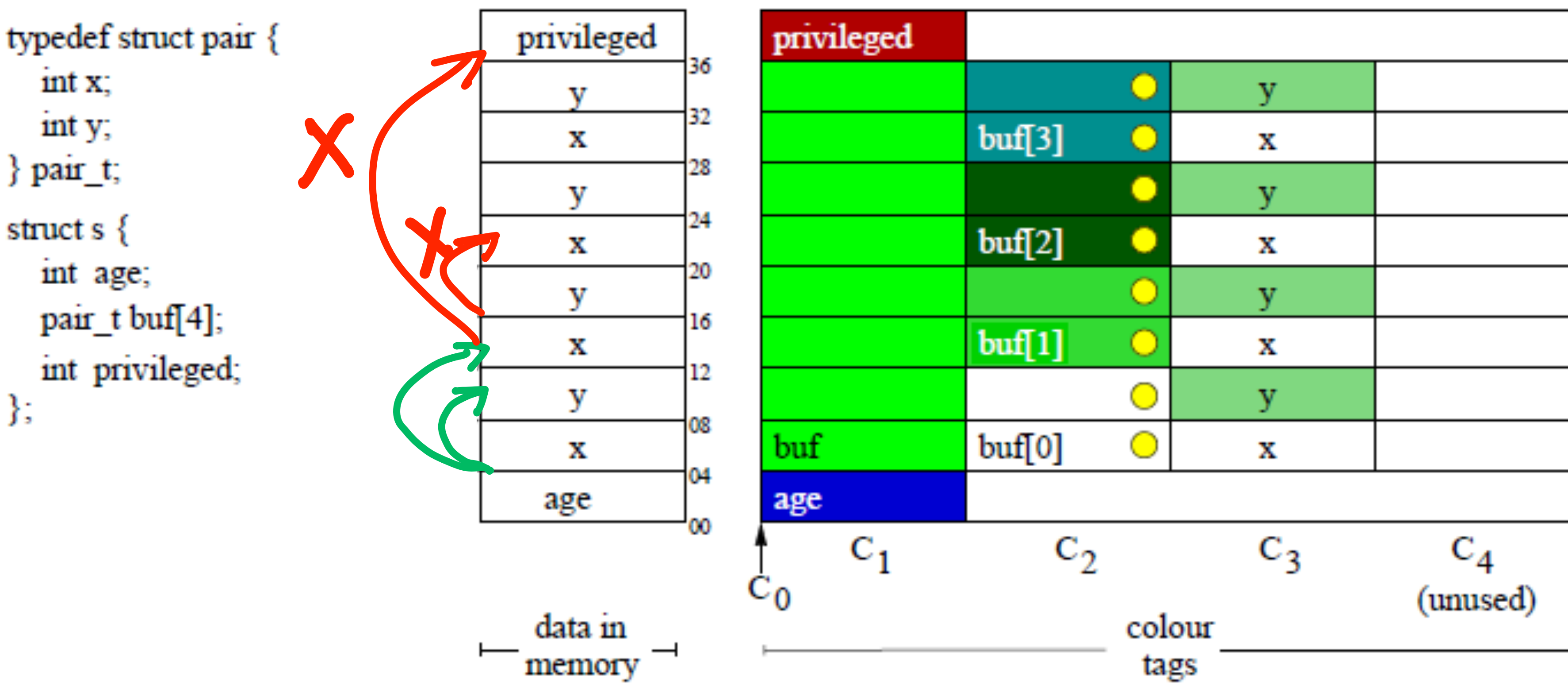
Reality requires subtle shades

```
typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int  age;
    pair_t buf[4];
    int  privileged;
};
```



3

In reality

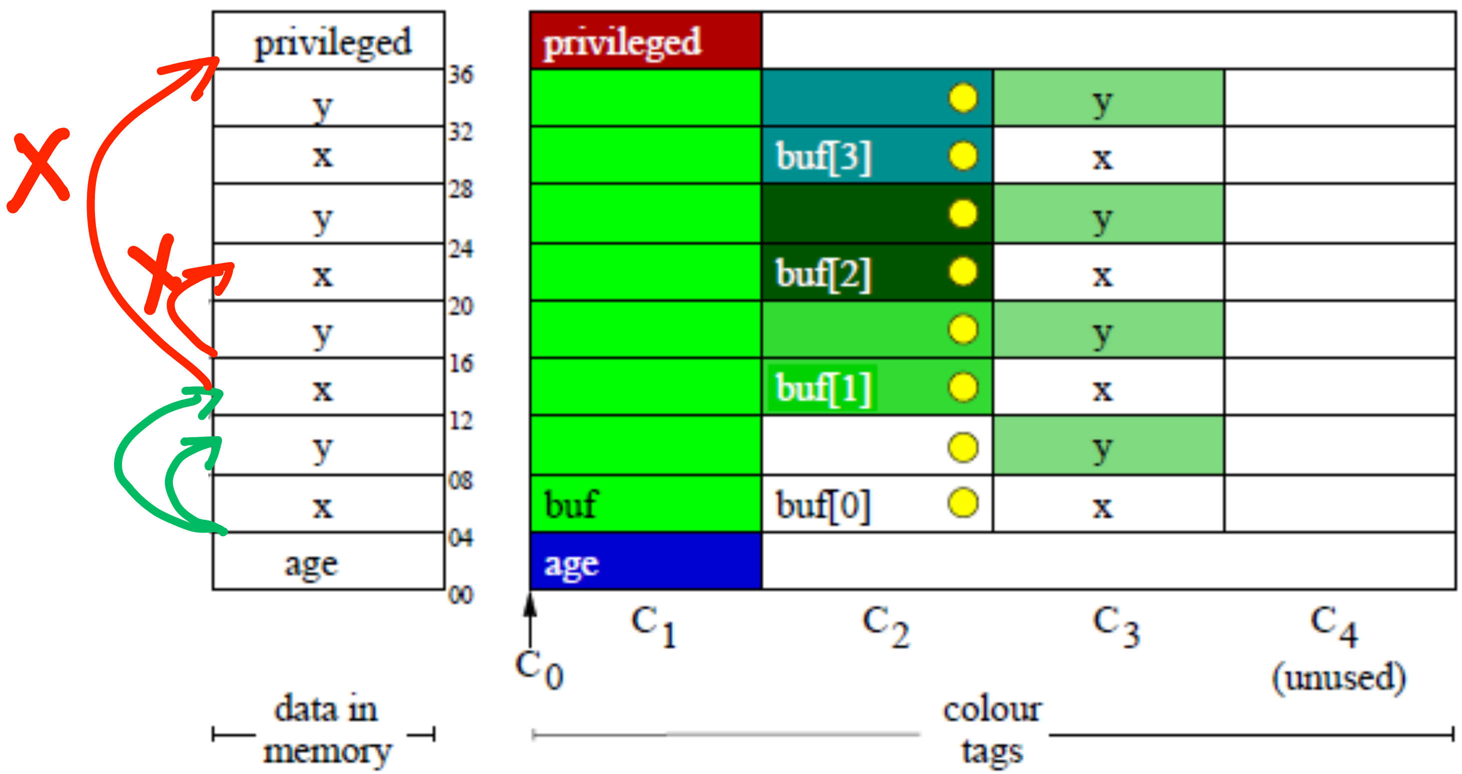


Check: does the pointer colour match that of the location pointed to?
(left to right, in all shades, with blanks serving as wild cards)

3

Unfortunately, some code is colour blind!

```
typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int  age;
    pair_t buf[4];
    int  privileged;
};
```




```
int *p;
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```

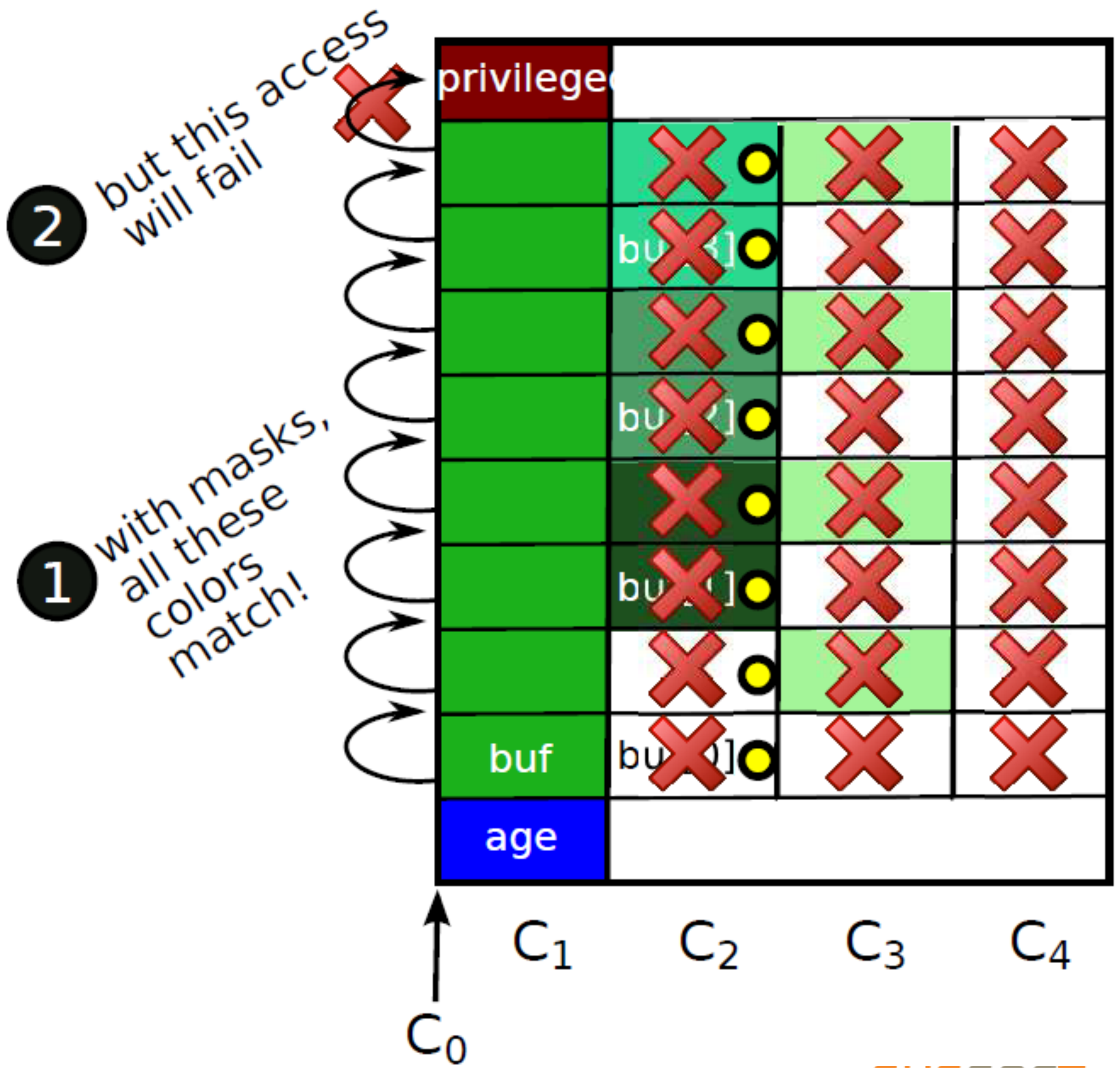
vrije Universiteit

syssec

3

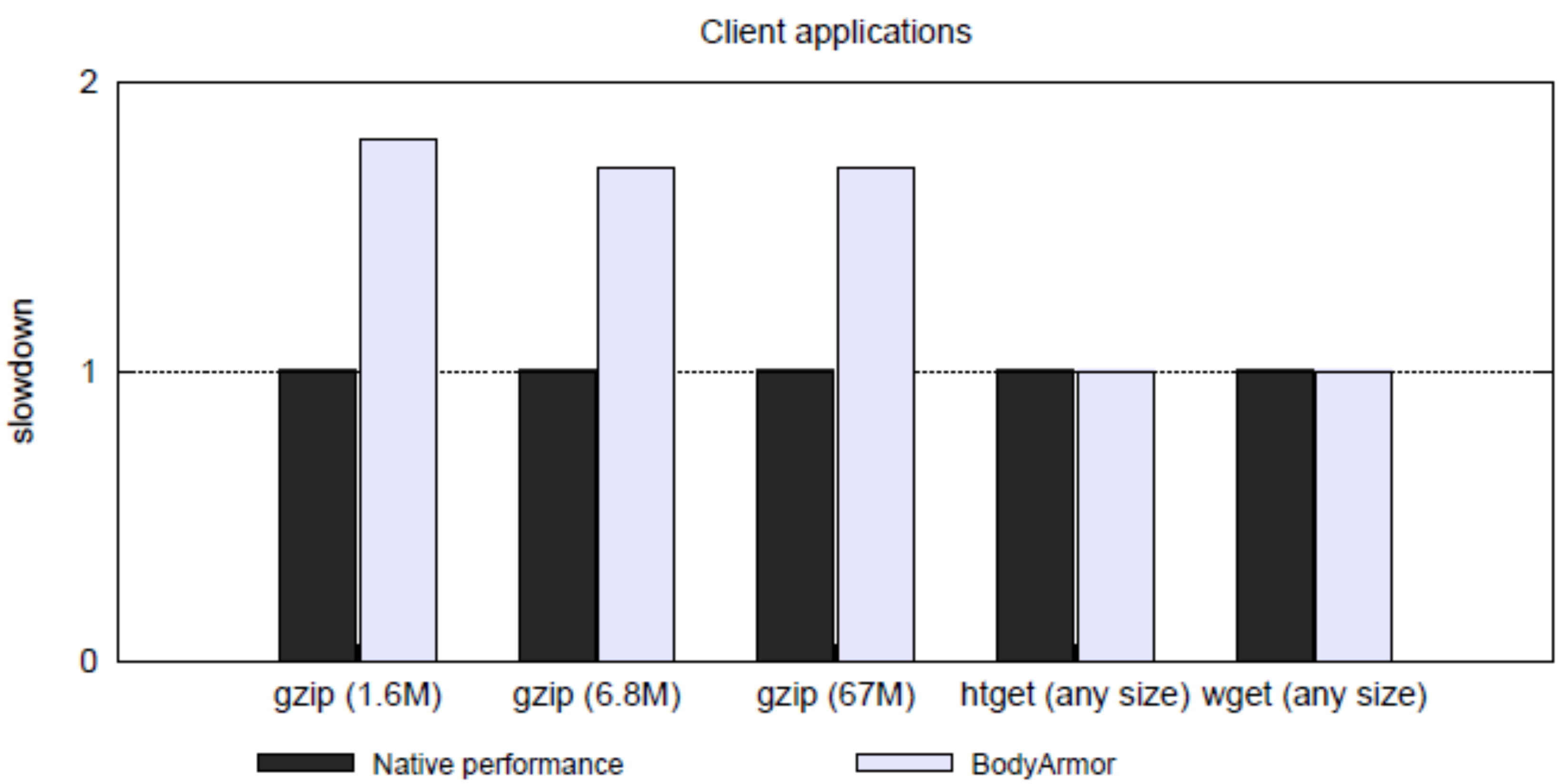
So we mask some shades

```
/* initialize the buffer
int *p;
int len = 4; //buf length

for(p = mystruct.buf;
  p < mystruct.buf+len;
  p++)
{
    *p = 0;
}
```



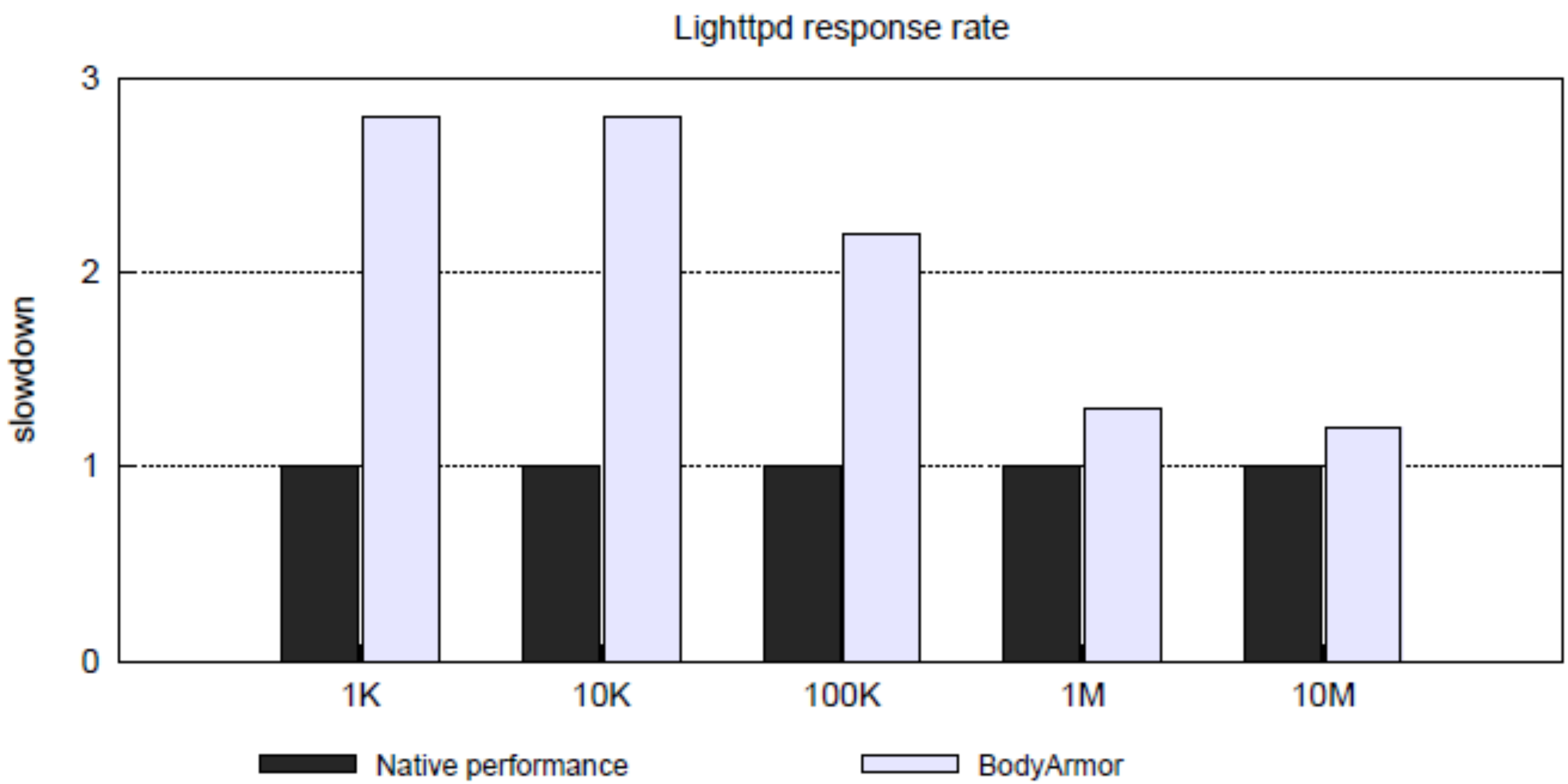
1 2 3

Performance?



1 2 3

Performance?

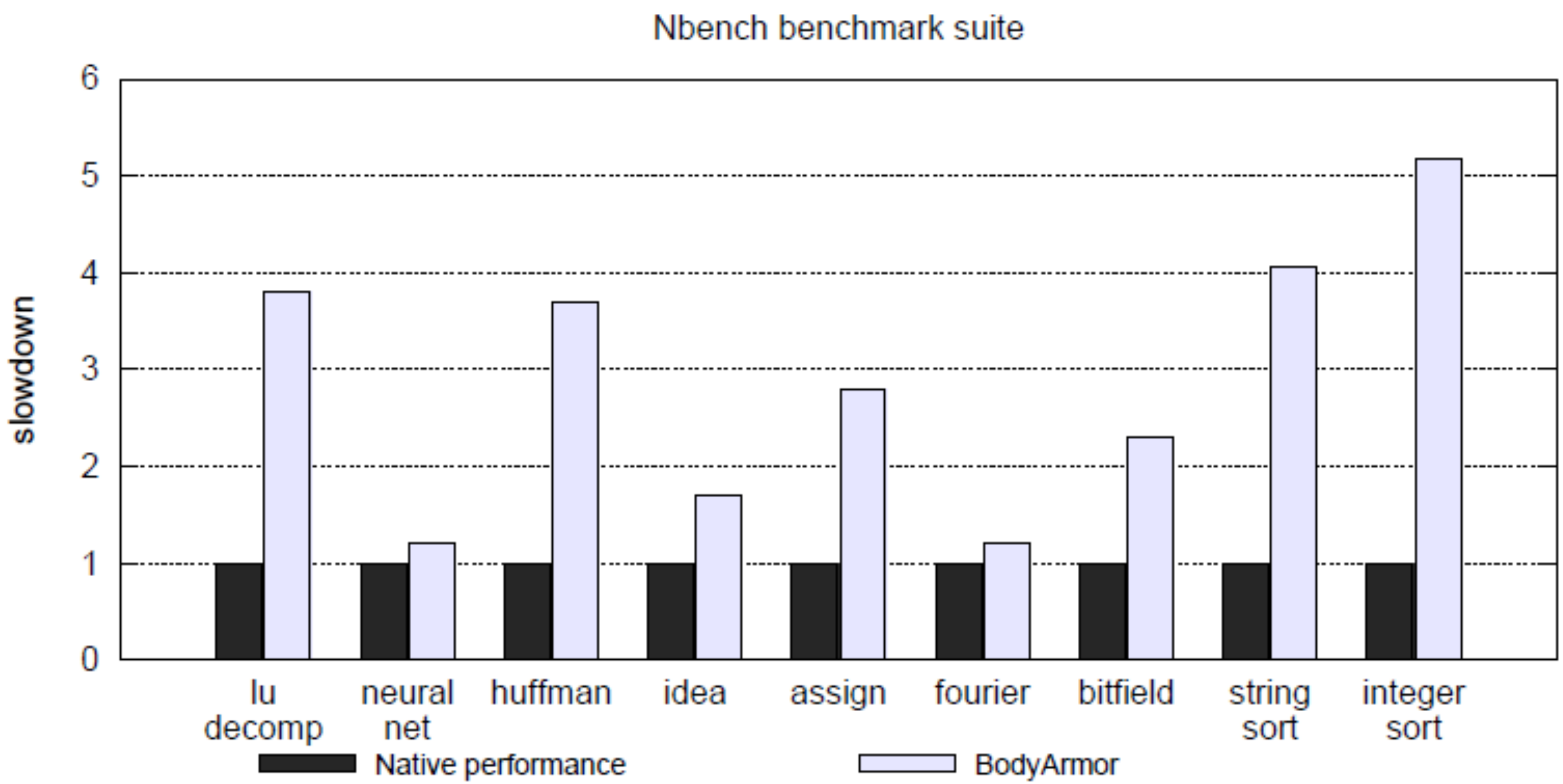


vrije Universiteit

syssec

1 2 3

Performance?



overall: 2.9

1 2 3

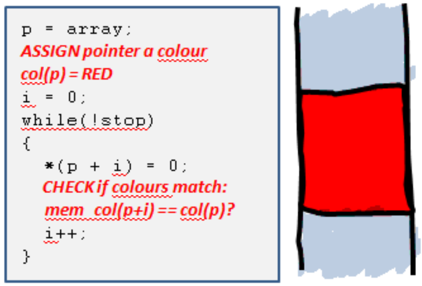
Effectiveness?

Application	Type of vulnerability	Security advisory
Proftpd 1.3.3a	Stack overflow	CVE-2010-4221
Htget 0.93 (1)	Stack overflow	CVE-2004-0852
Htget 0.93 (2)	Stack overflow	
Aspell 0.50.5	Stack overflow	CVE-2004-0548
Iwconfig v.26	Stack overflow	CVE-2003-0947
Aeon 0.2a	Stack overflow	CVE-2005-1019

Application	Type of vulnerability	Security advisory
Exim 4.41	Heap overflow, non-control data	CVE-2010-4344
bc-1.06 (1)	Heap overflow	Bugbench [27]
bc-1.06 (2)	Heap overflow	Bugbench [27]
Nullhttpd-0.5.1	Heap overflow, reproduced	CVE-2002-1496
Squid-2.3	Heap overflow, reproduced	Bugbench [27]
Ncompress 4.2.4	Stack overflow	CVE-2001-1413

Conclusions

- BinArmor
  - protect against attacks on non-control data
  - few (if any) FPs
  - efficient compared to DTA
  - not fully optimised yet!



http://www.cs.vu.nl/~herbertb/