Full Speed in Reverse

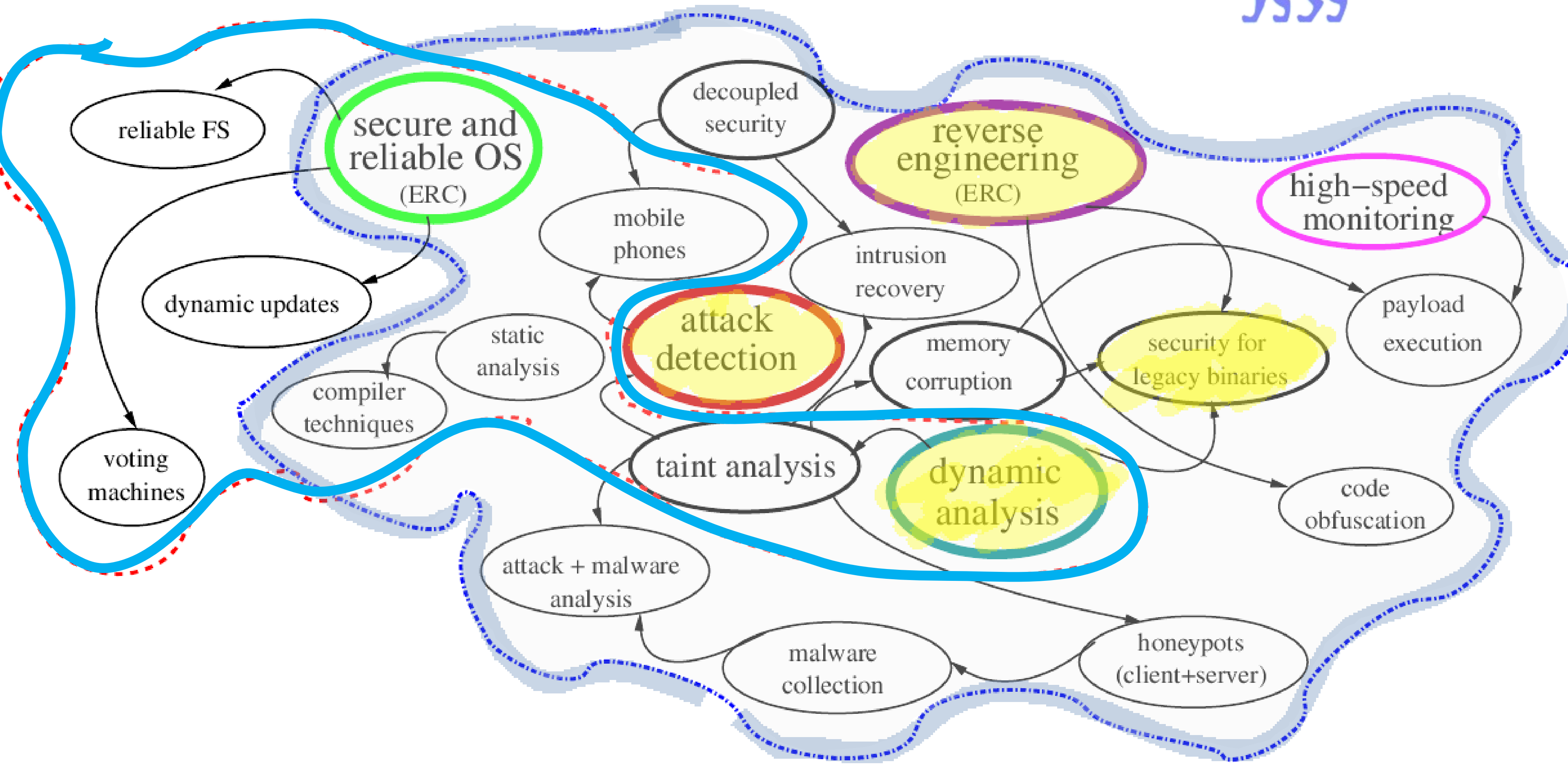
protecting legacy binaries from memory corruption attacks

syssec

Herbert Bos

VU University Amsterdam

Systems Security @ VU



vrije Universiteit

syssec

Grants

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

CAN WE WIN?

vrije Universiteit

syssec

Can we stop advanced malware?

- Like Stuxnet, Duqu, Zeus, TDL4, Flame(?)

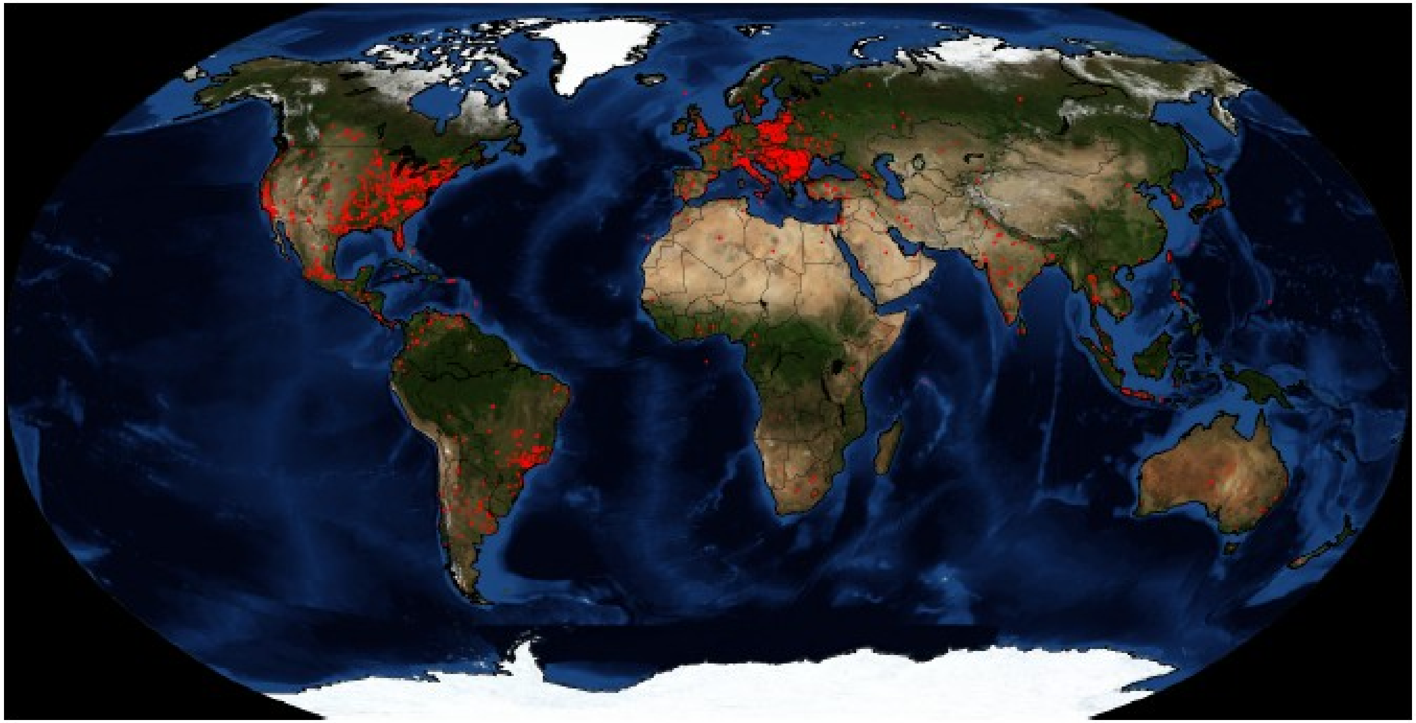
Can we stop advanced malware?

- Zeus is a banking trojan that has been around since 2007
- Sold as a DIY toolkit, anyone can create their own botnet for $4000
- In 2010, the FBI discovered an organized crime network which stole over $70m using Zeus

Can we stop advanced malware?

Zeus v3

- In May 2011, the source code of Zeus v2 was leaked
- The leaked code evolved into a new Zeus P2P variant around October
- The Zeus P2P network is currently estimated to consist of 150.000 – 200.000 peers worldwide



Can we stop advanced malware?

- We tried
  – based on the work by Dennis Andriess

vrije Universiteit

syssec

Can we stop advanced malware?

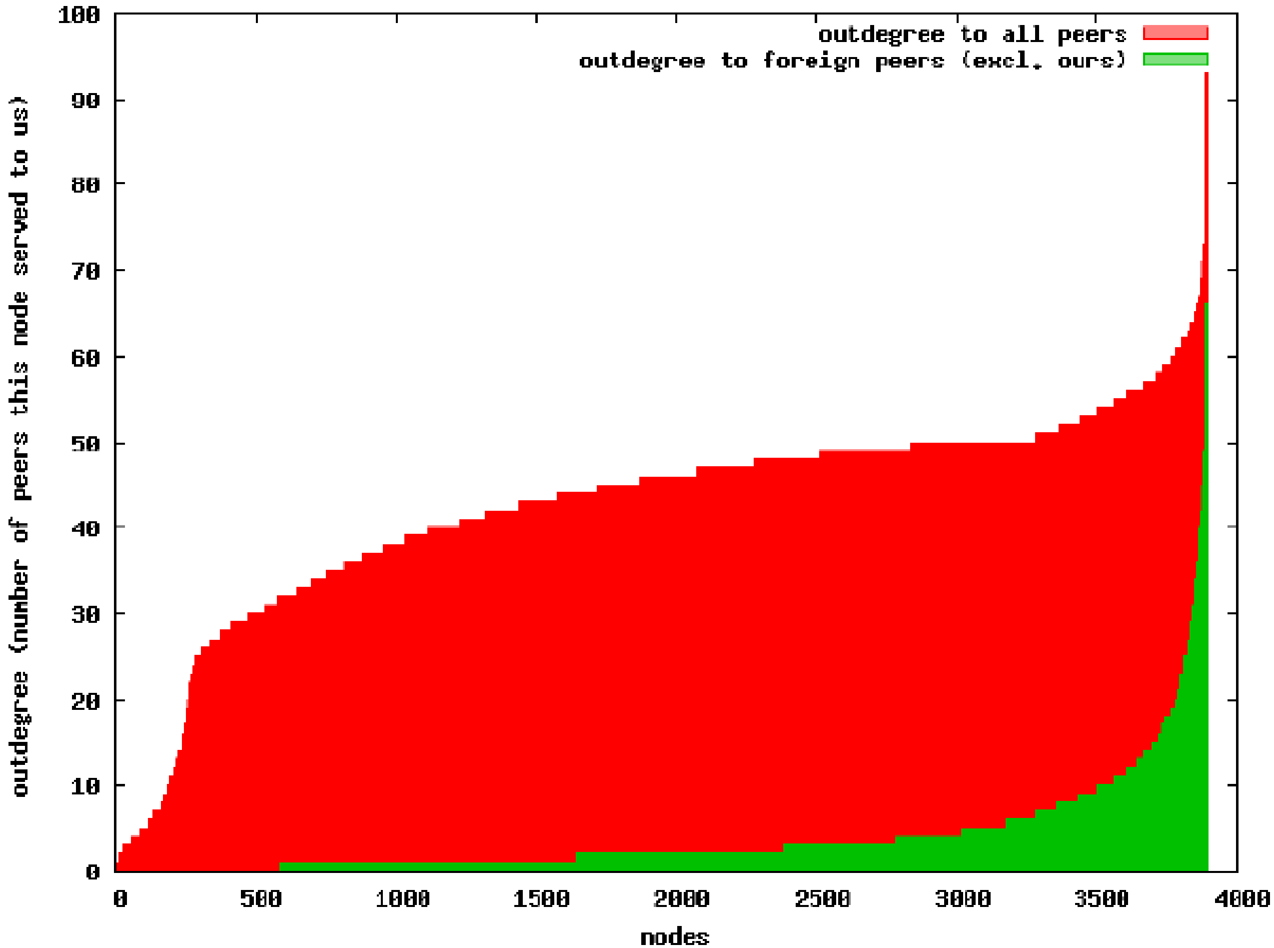
- Plan:
  - Reverse engineer relevant parts
  - Poison the botnet
-
- Reversing is hard
  - several layers of encryption and obfuscation
  -
- We dump the code *after* unpacking
  - and get 268.000 lines of assembly ☹

vrije Universiteit

syssec

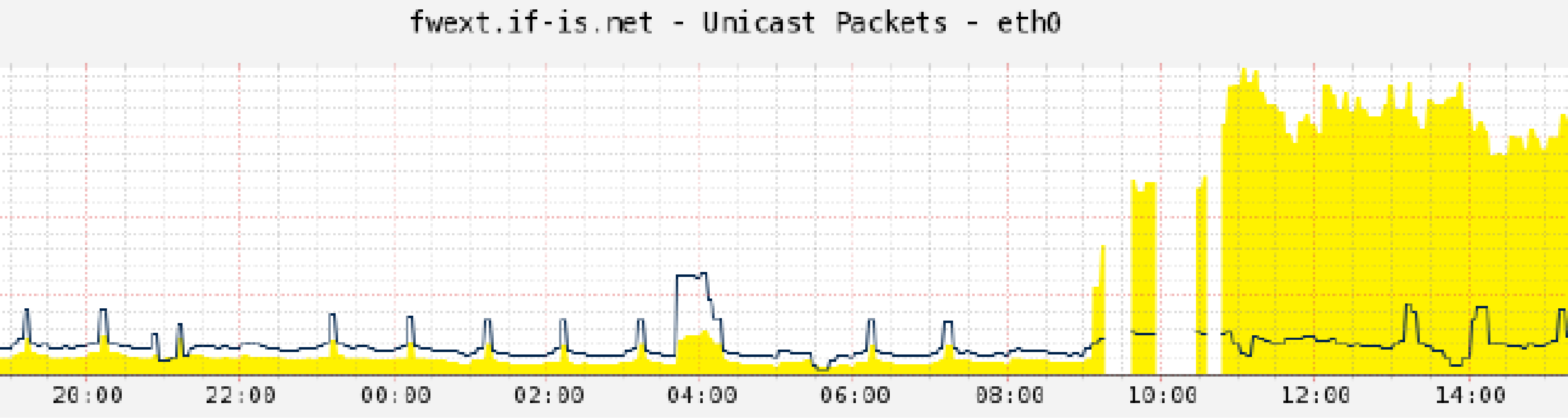
Can we stop advanced malware?

- By May 5th, a significant part of the peerlist entries in the Zeus network points to us



Can we stop advanced malware?

- Zeus strikes back
  - DDoS
  - Update that blacklists our IP addresses



vrije Universiteit

syssec

Can we stop advanced malware?

- We managed to take out 25% of the nodes
-
- In the end, we failed!
-
- Trivial to make even more resilient botnets
-

vrije Universiteit

syssec

Perhaps we need better protection

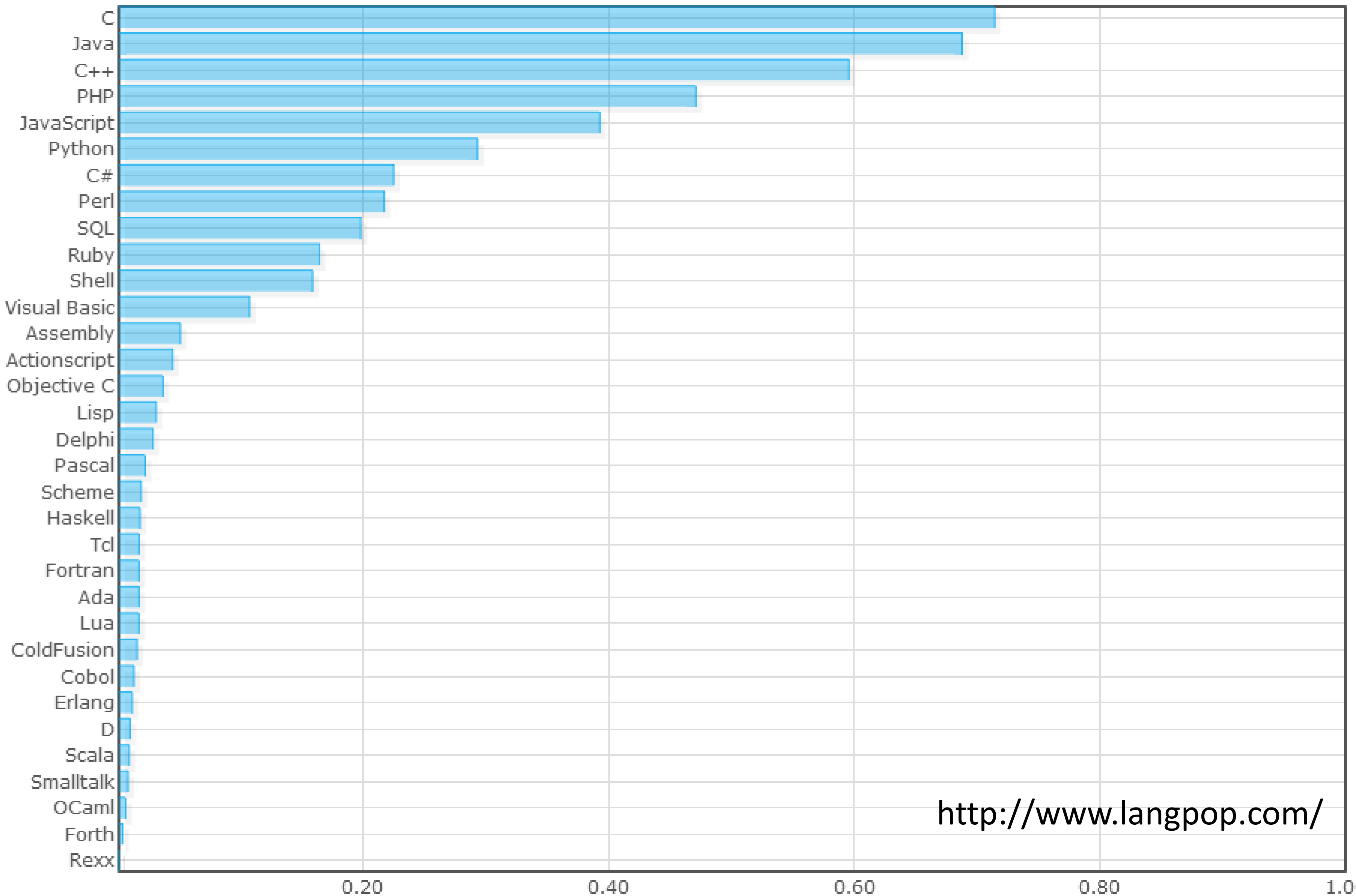
Next few slides are based on

Memory Errors: Past, Present, and Future (RAID'12)

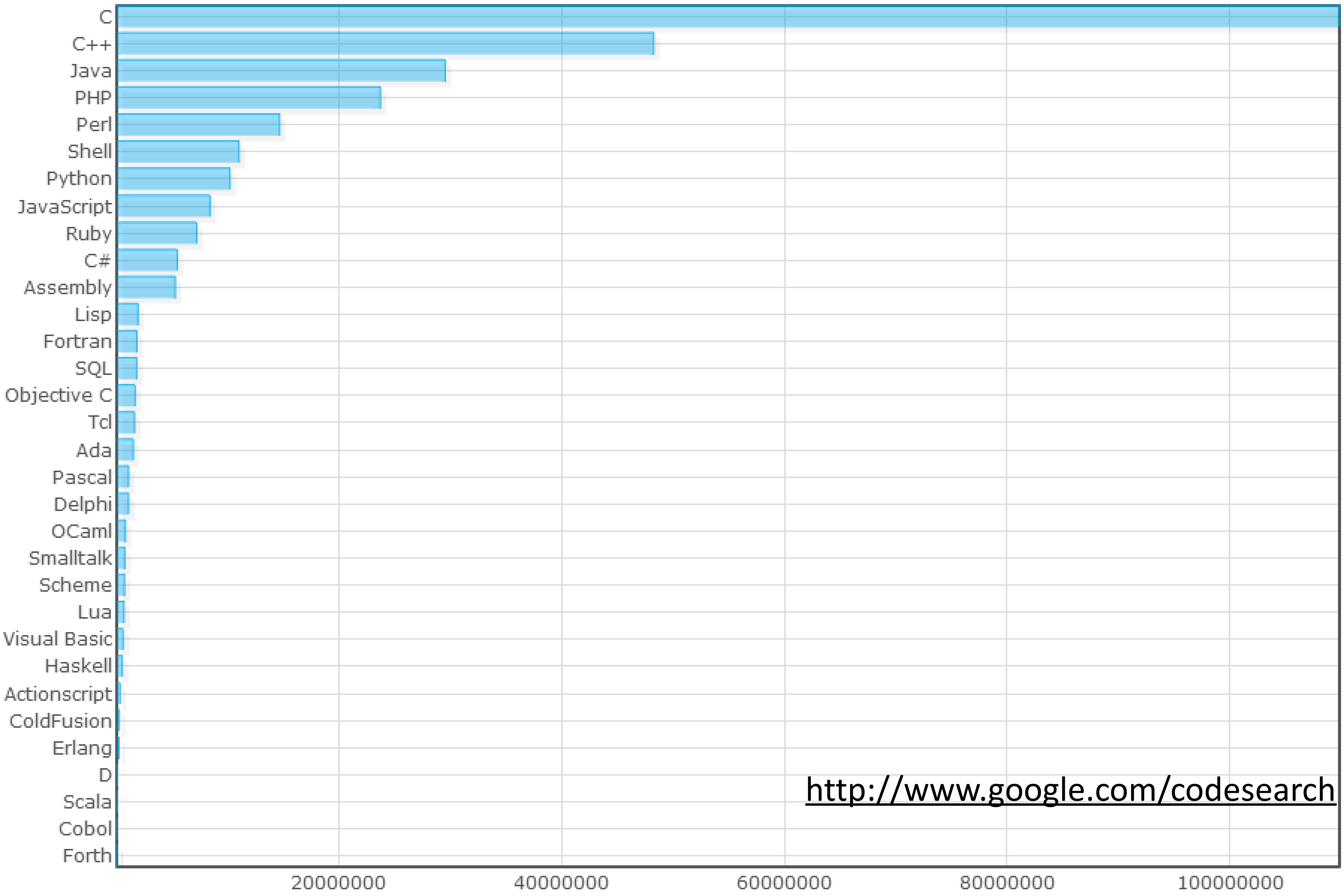
- Victor van der Veen, Nitish Sharma, Lorenzo Cavallaro, Herbert Bos



The most popular language in the world

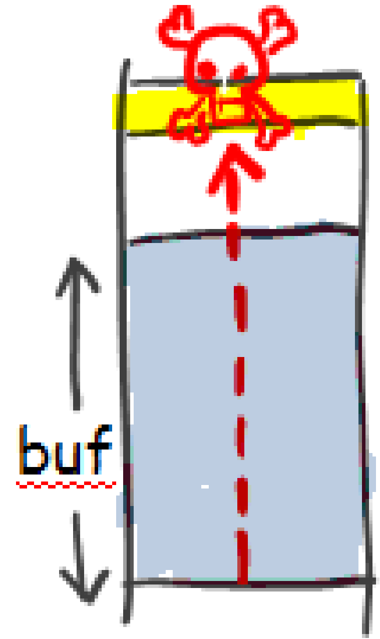


The most popular language in the world



Buffer overflows

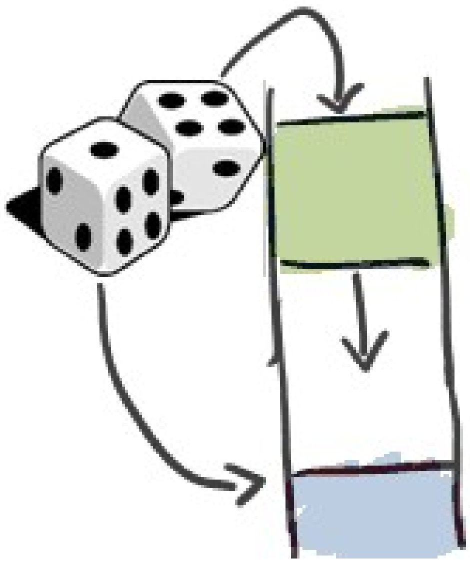
- Perpetual top-3 threat
  - SANS CWE Top 25 Most dangerous programming errors
- Most drive-by-downloads
  - infect browser, download malware
  -



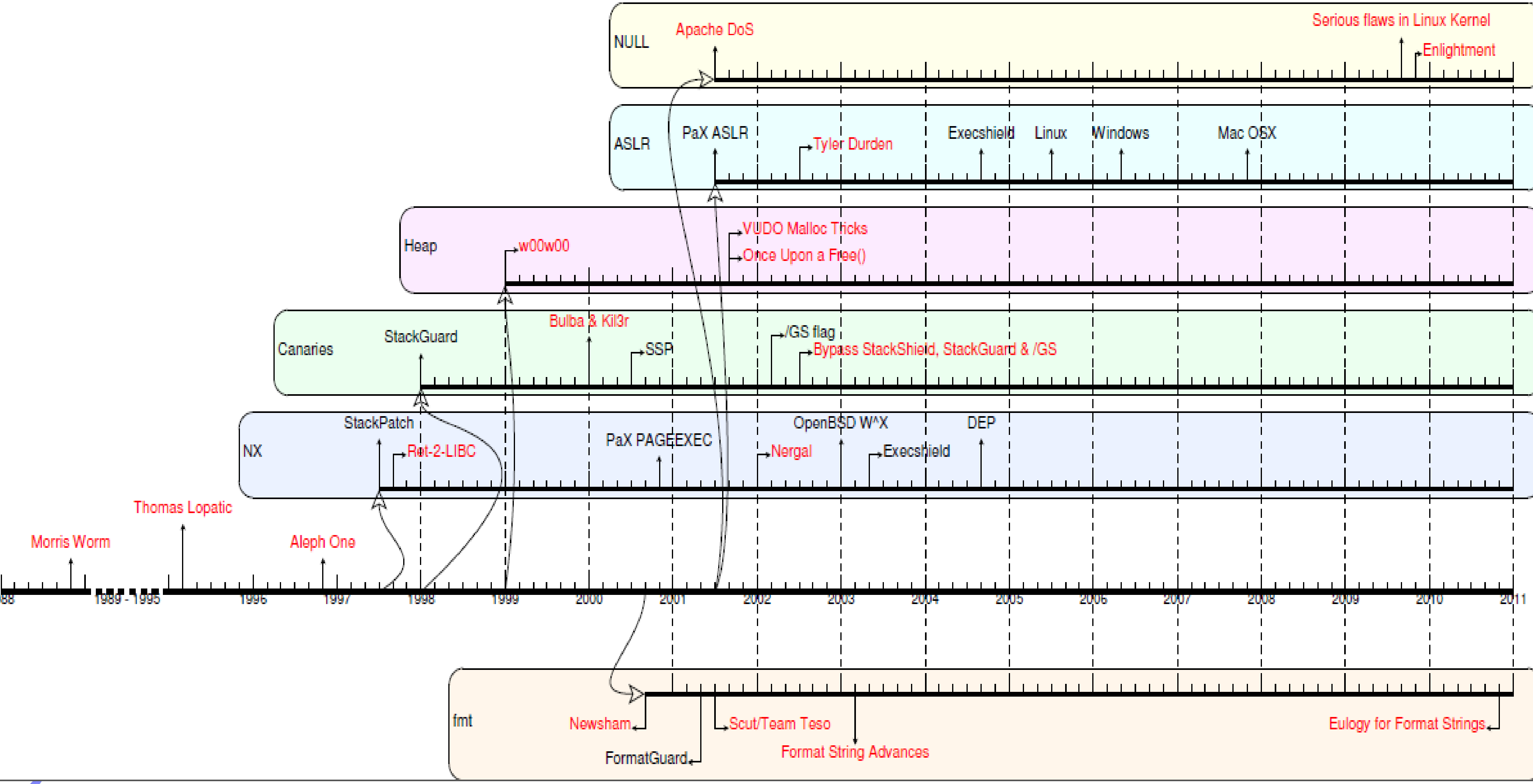
Many defensive measures

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

ProPolice



Evolution at work



vrije Universiteit

syssec

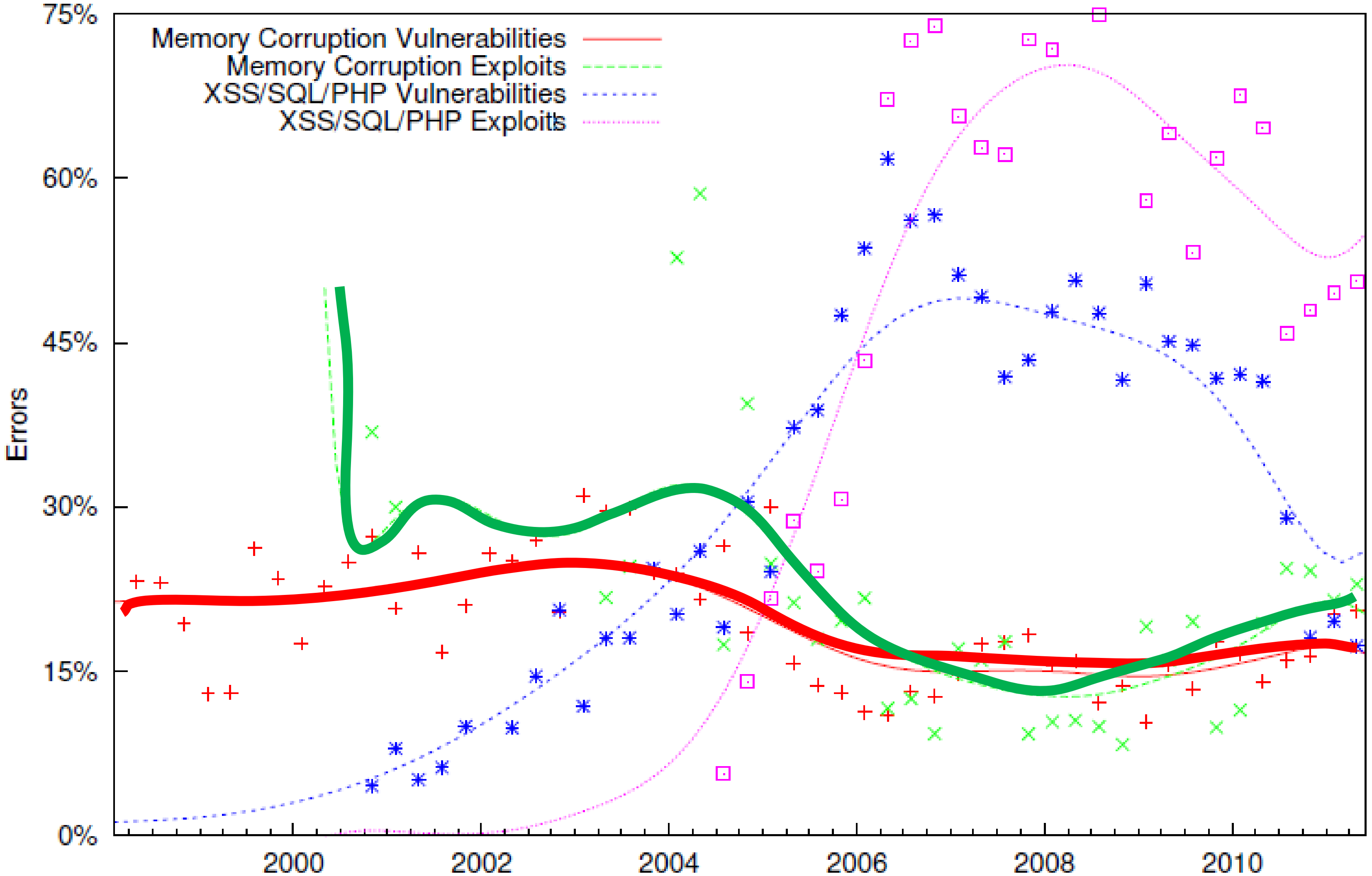
Still they come

vrije Universiteit

syssec

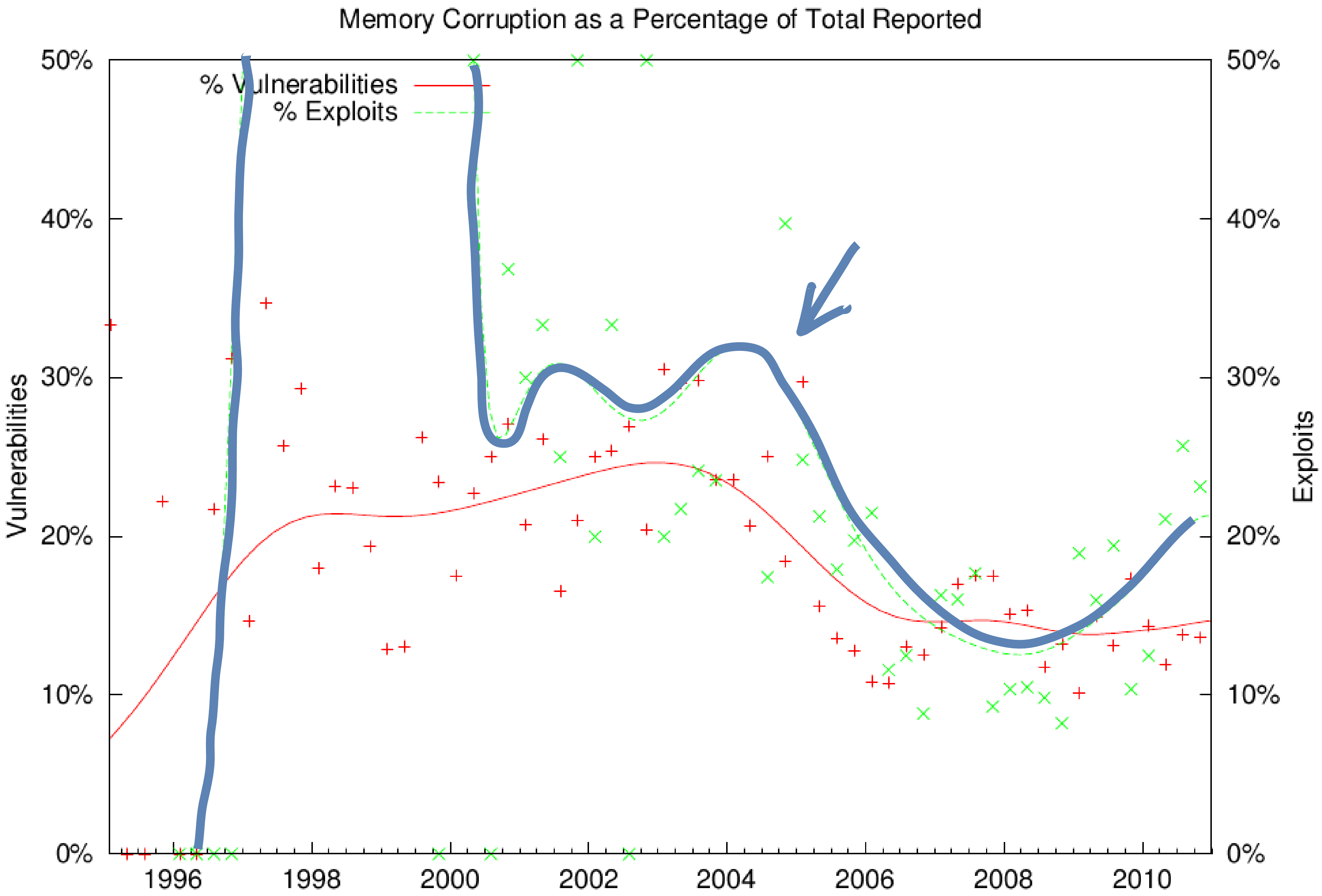
Vulnerabilities and exploits

(as percentage of total)



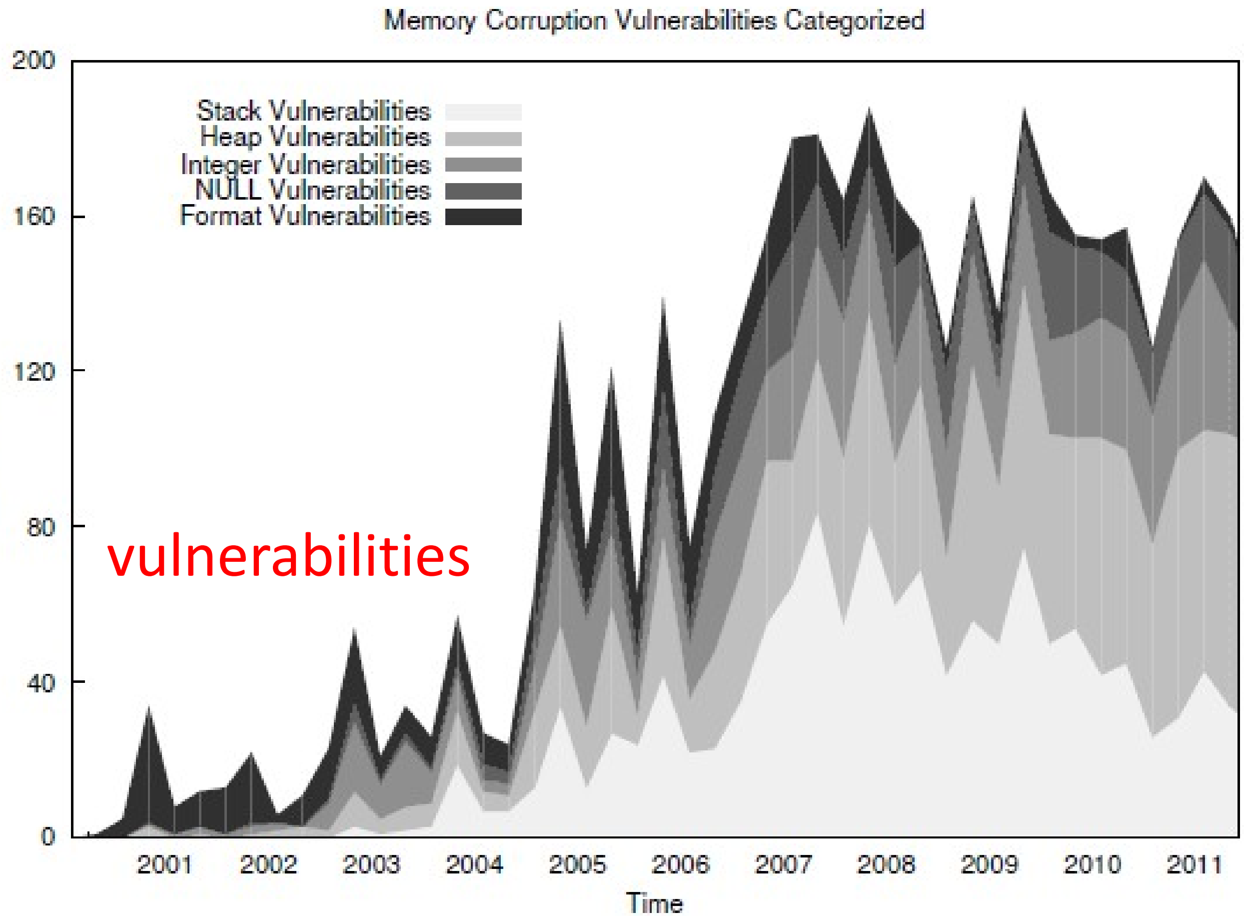
vrije Universiteit

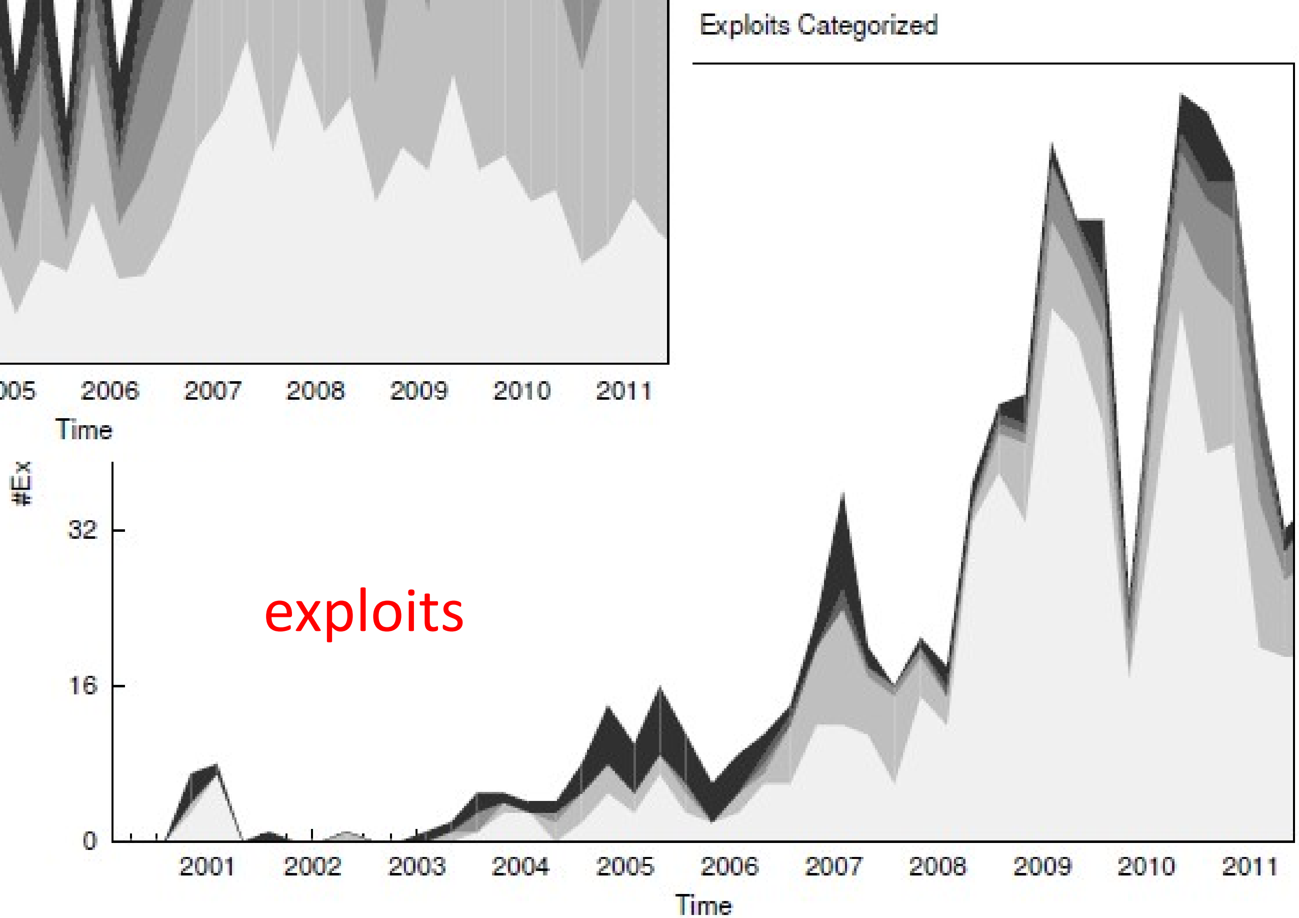
syssec



vrije Universiteit

syssec





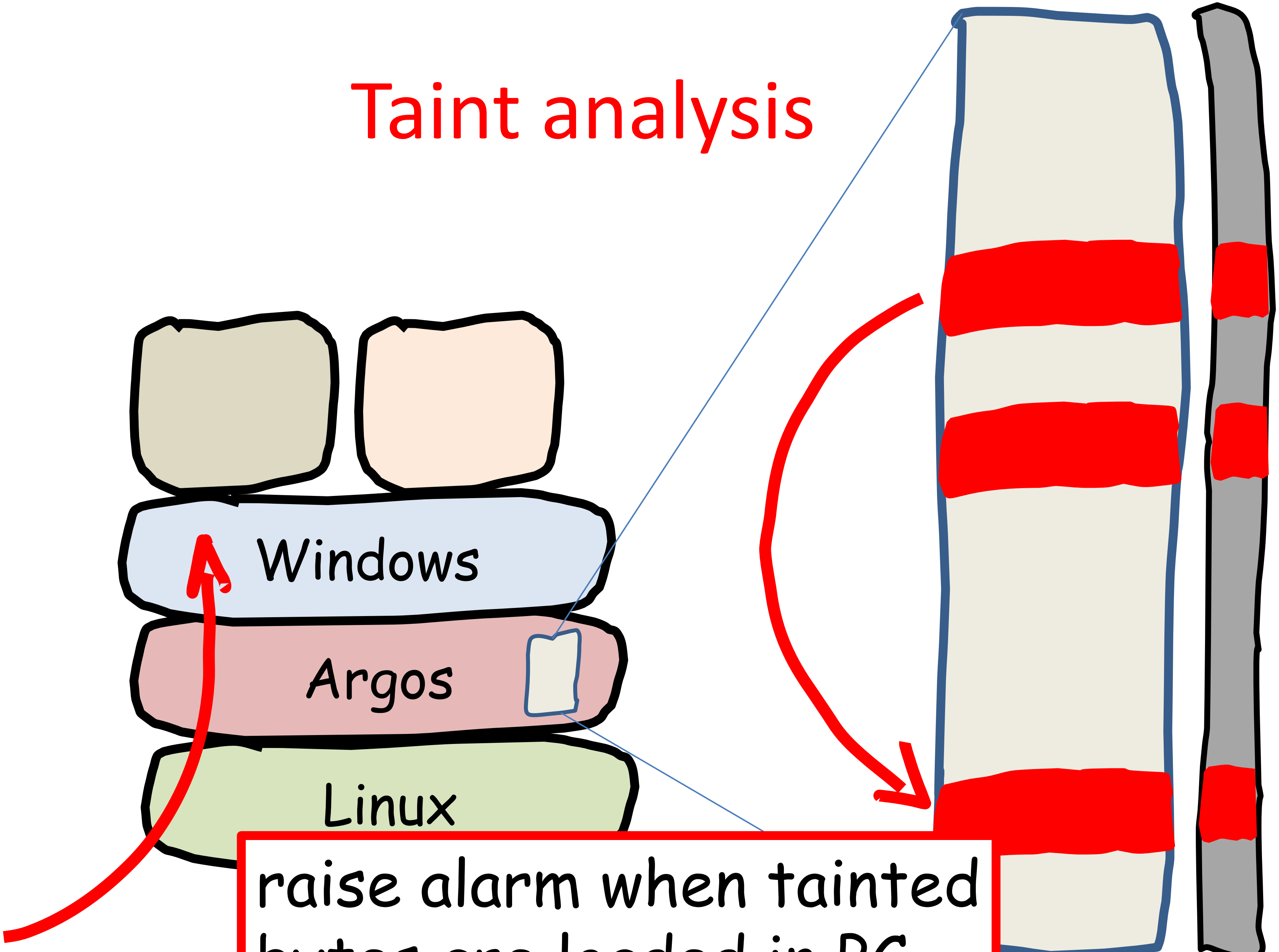
vrije Universiteit

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



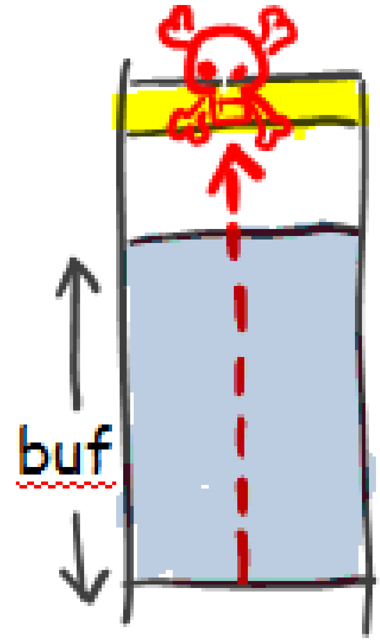
raise alarm when tainted bytes are loaded in PC

Taint tracking: useful, but slow

vrije Universiteit

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

just missed it!



...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
fo s

This talk is based on two papers

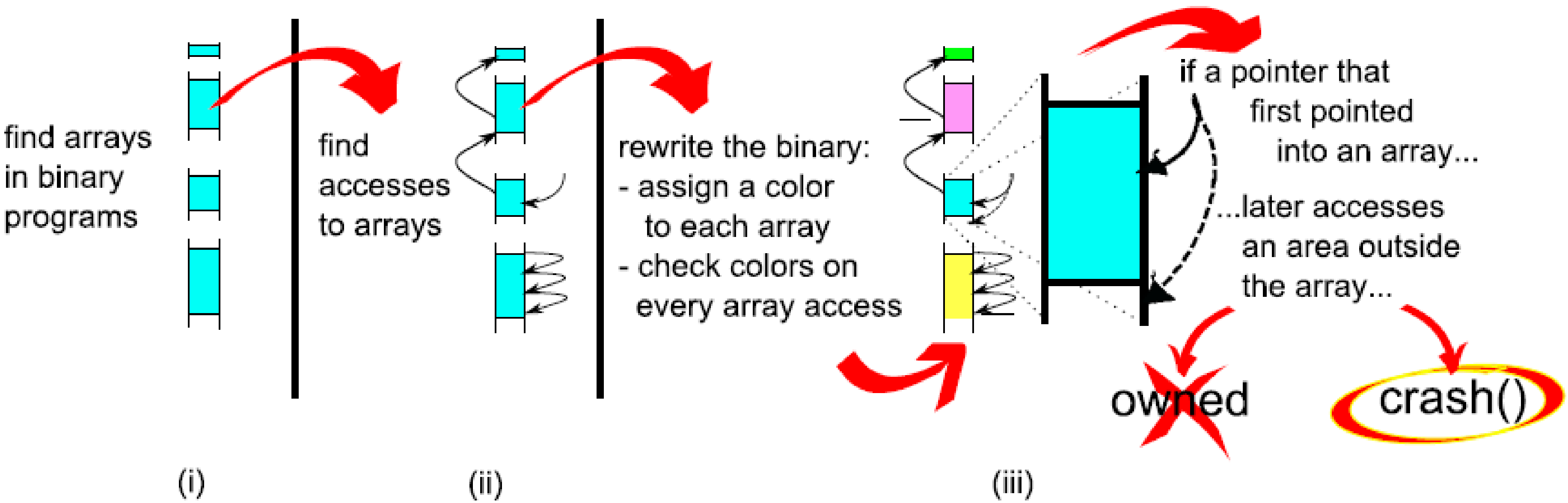
- Asia Slowinska, Traian Stancescu, Herbert Bos
  Howard: a dynamic excavator for reverse engineering data structures (NDSS'11)
- Asia Slowinska, Traian Stancescu, Herbert Bos
  Body armor for binaries: preventing buffer overflows without recompilation (USENIX'12)
-
-

vrije Universiteit

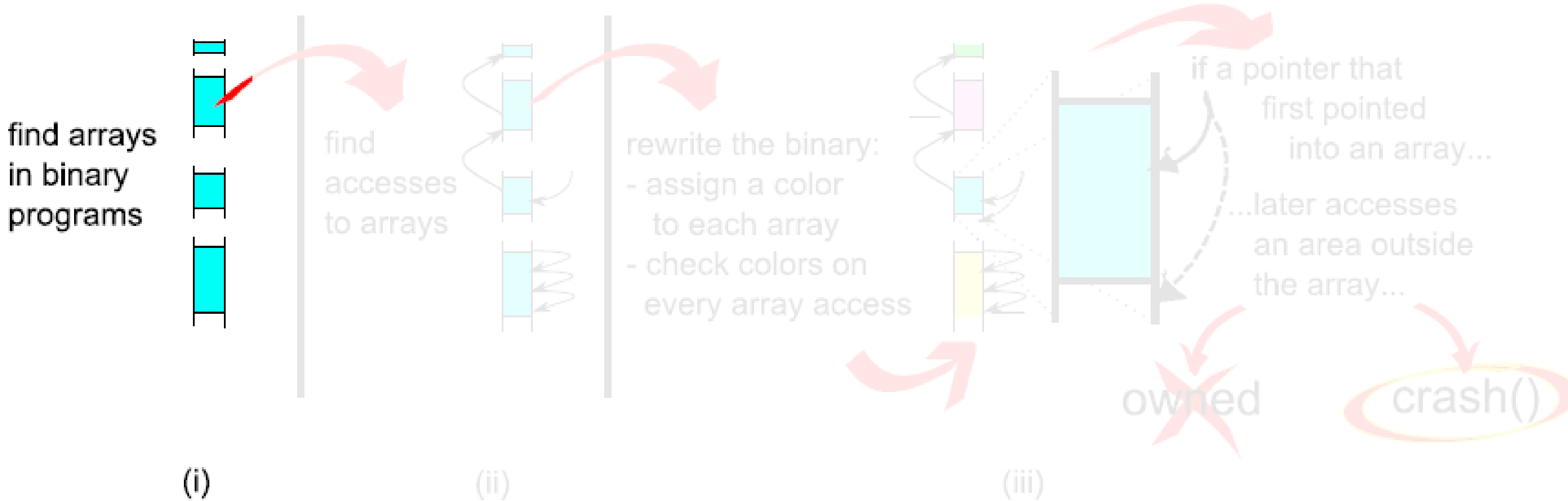
syssec

no source
no symbols
no clue?

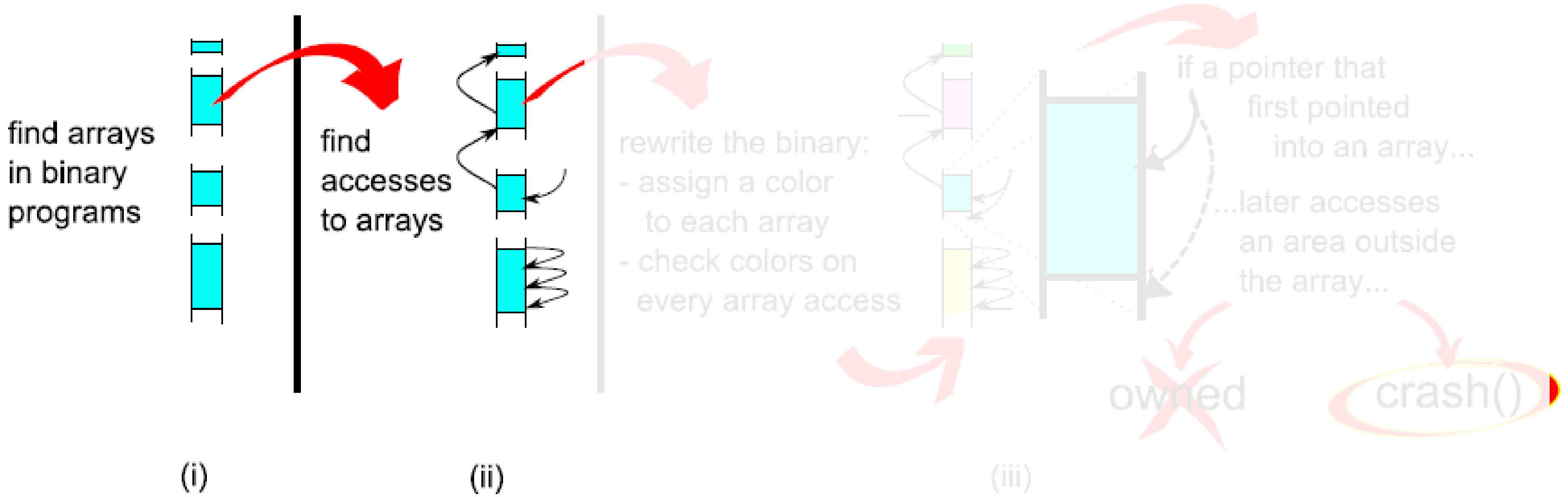
In a nutshell...



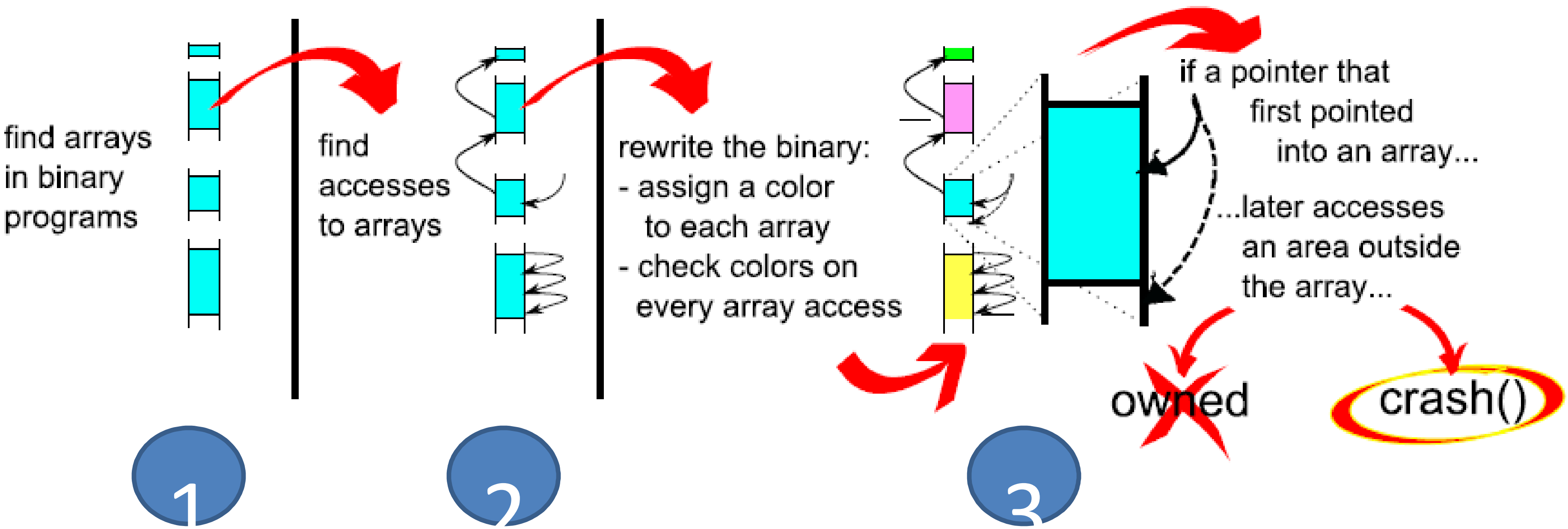
In a nutshell...



In a nutshell…

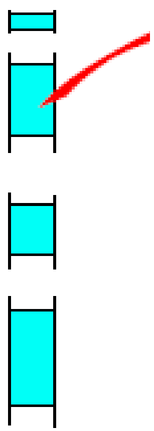


In a nutshell…



1

Step 1: extract the arrays



Two possibilities

– symbol tables

– stripped

➔ reverse engineering

let’s assume the latter

1

Problem

vrije Universiteit

syssec

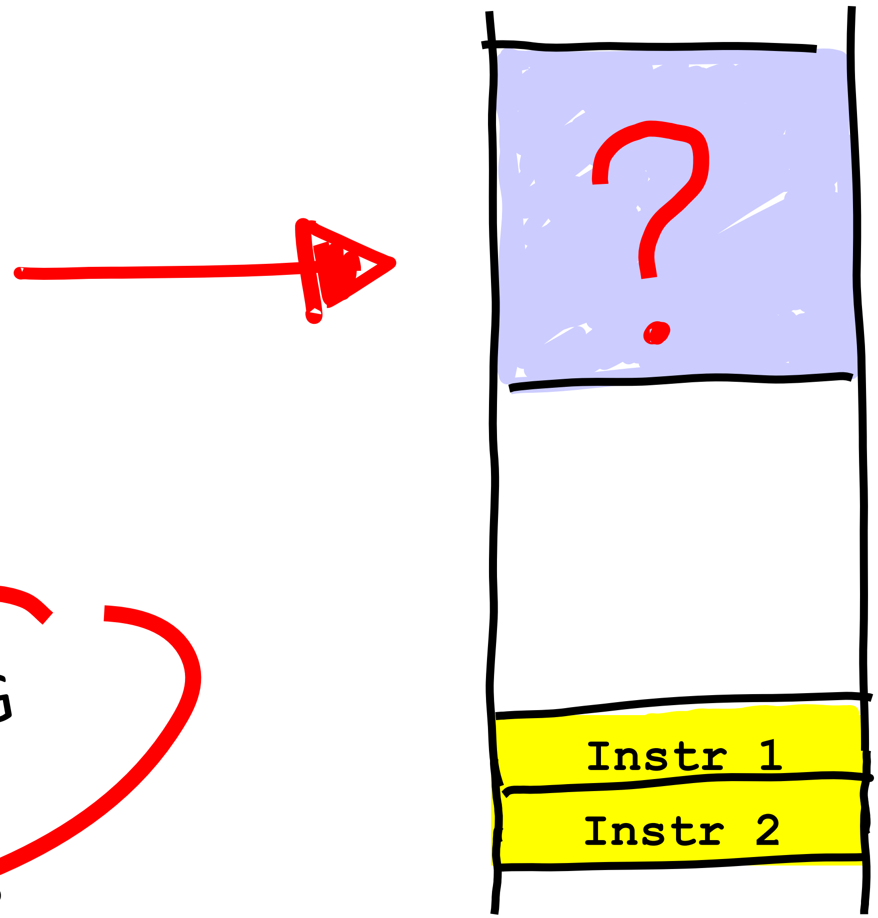
1

Why is it difficult?

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

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

Why is it difficult?

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

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



MISSING

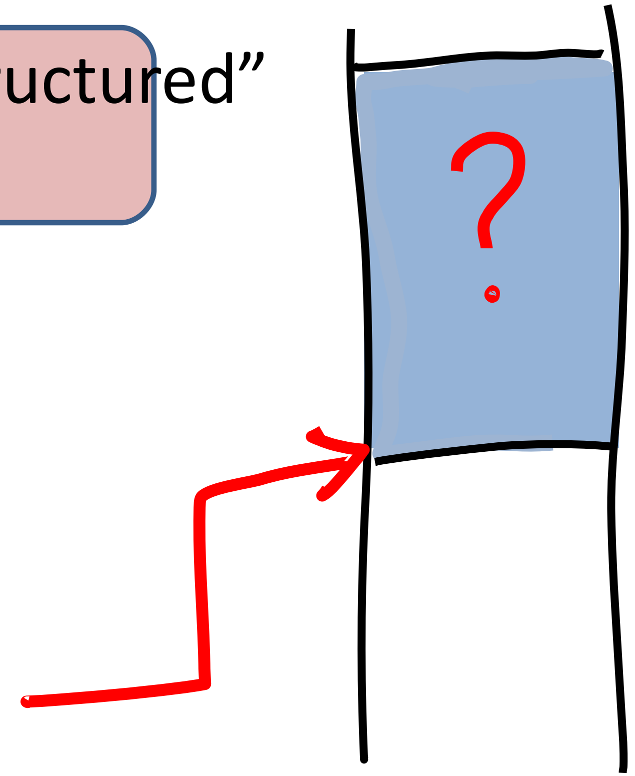
- Data structures
-

1

Data structures: key insight

Yes, data is “apparently unstructured”

But usage is not!



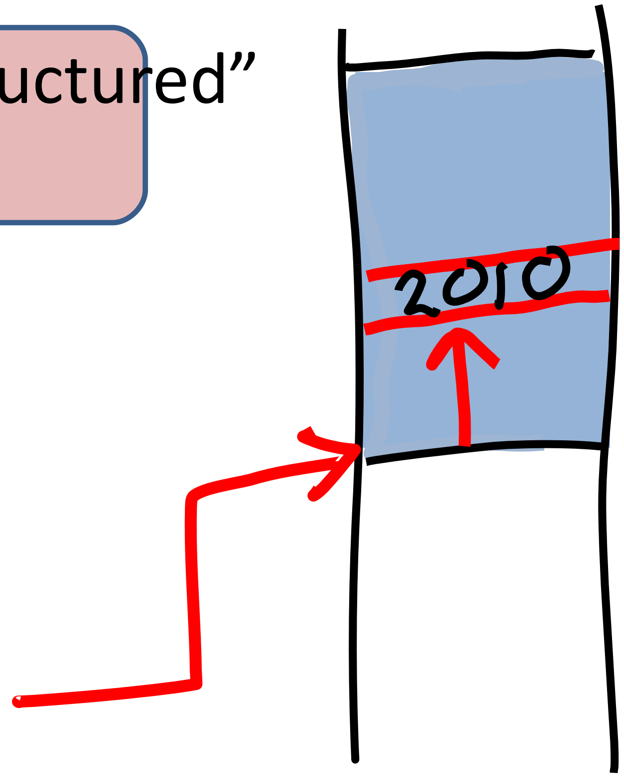
Data structures: key insight

Yes, data is "apparently unstructured"

But usage is not!

–

–



Data structures: key insight

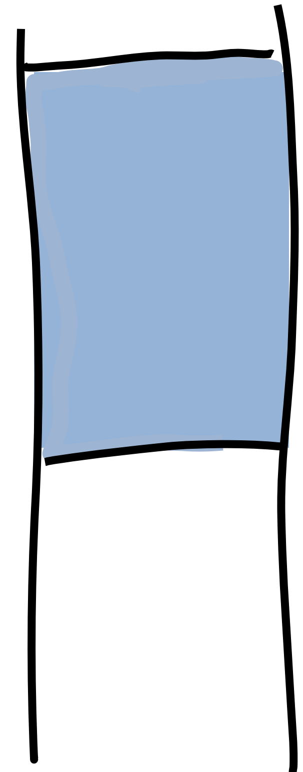
Yes, data is “apparently unstructured”

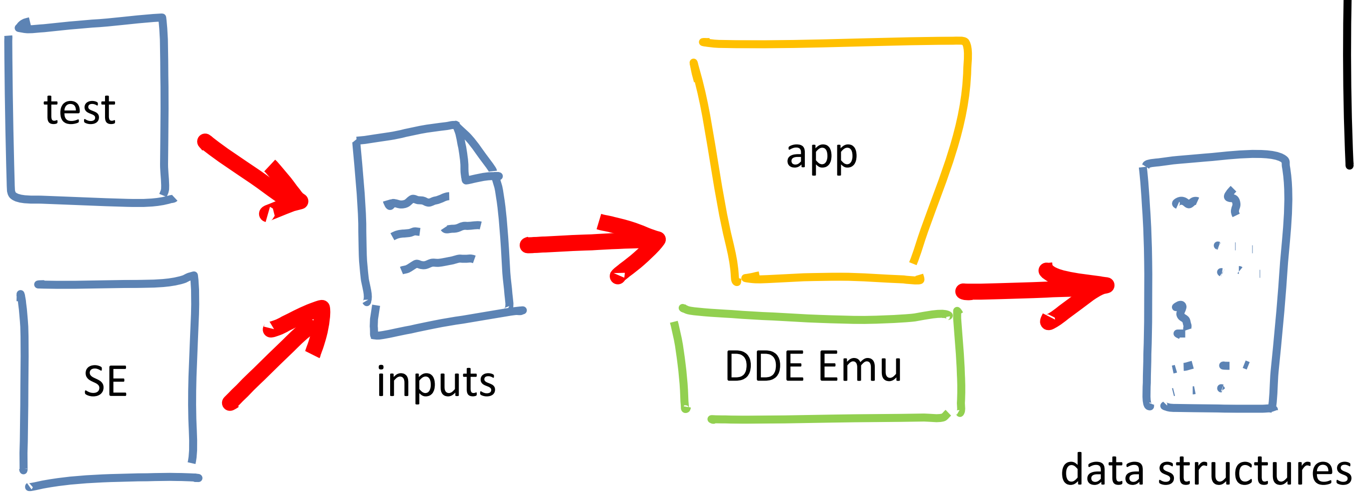
But usage is not!

–

–

Analyse dynamically

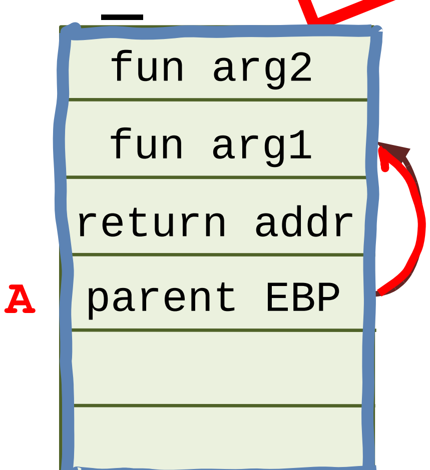




Intuition

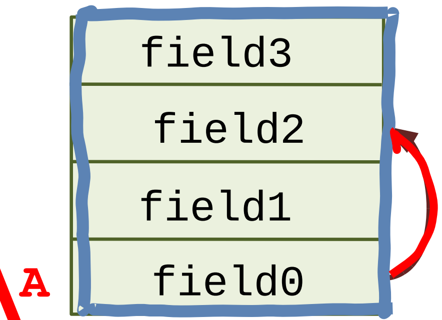
- Observe how memory is *used* at runtime to detect data structures
- E.g., if A is a point

1. and A is a functi then *(A + 8) function argu

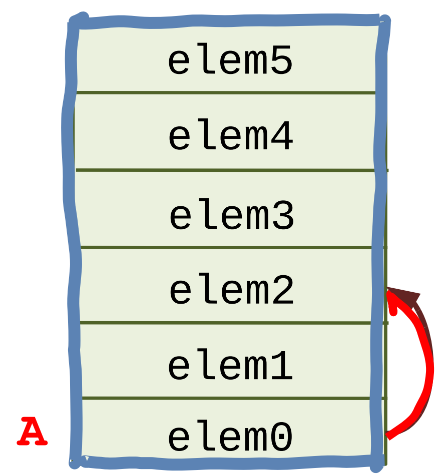


Track pointers

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

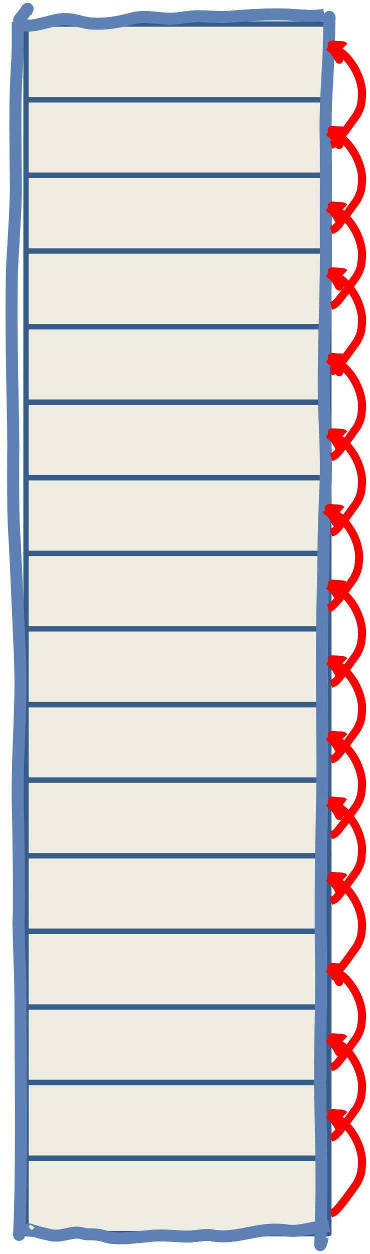


Approach

- 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()`

Arrays are tricky

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

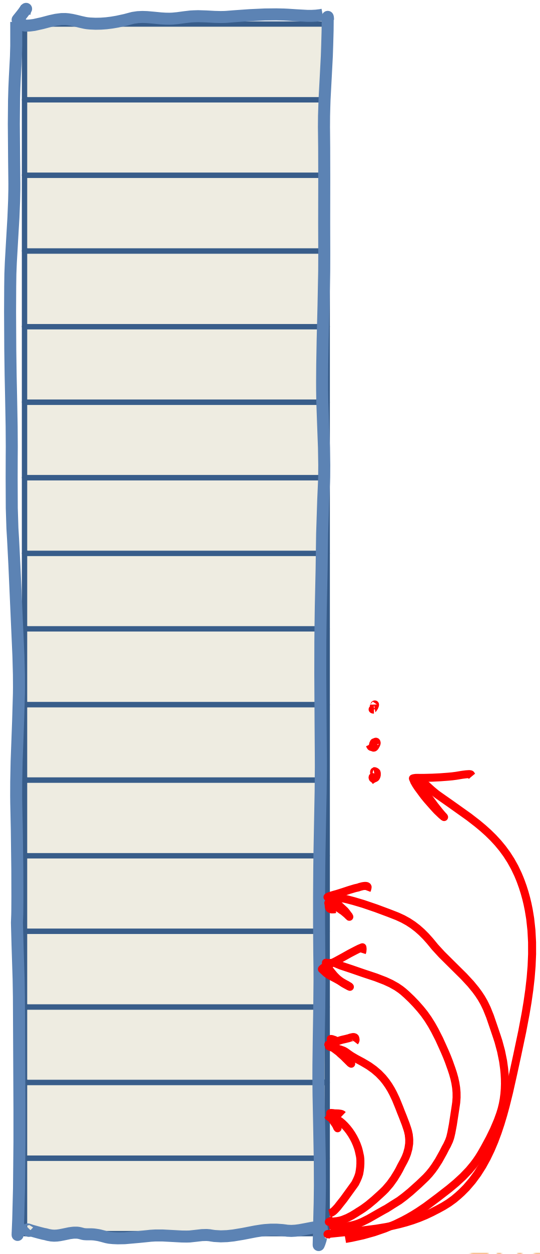


Arrays are tricky

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

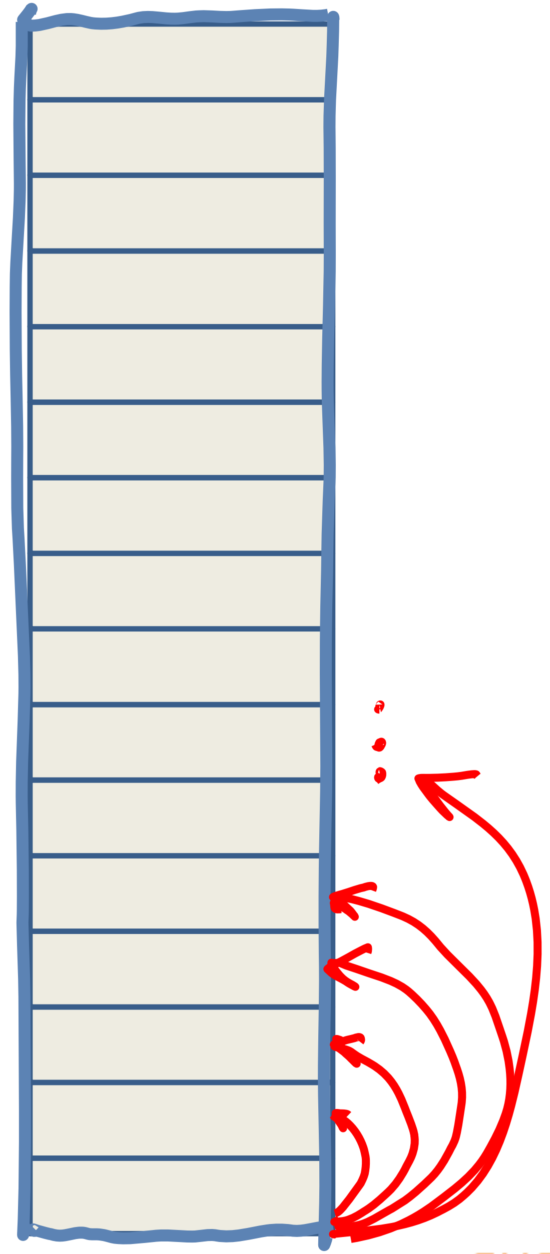
Arrays are tricky

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



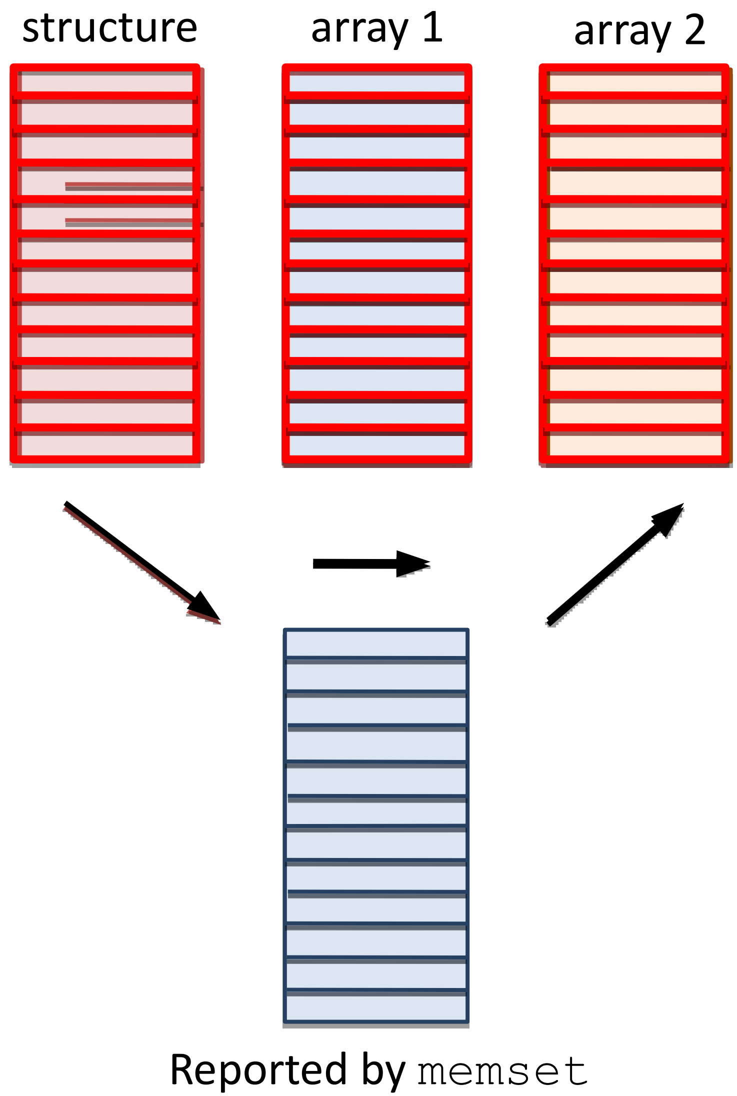
Arrays are tricky

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



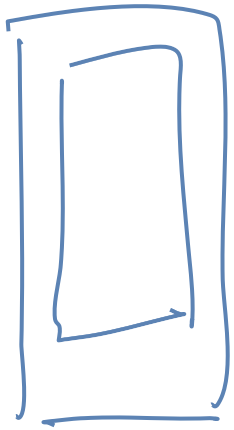
Interesting challenges

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



Further Challenges

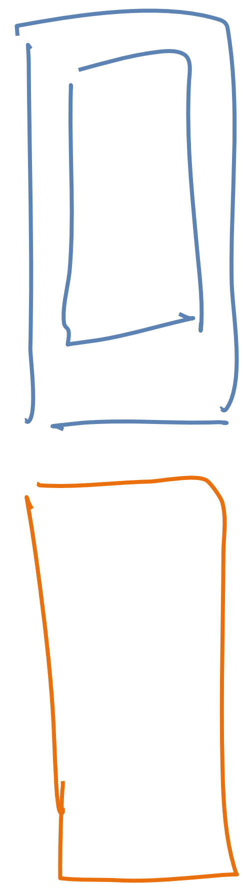
- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements
  -
    -



vrije Universiteit

Further Challenges

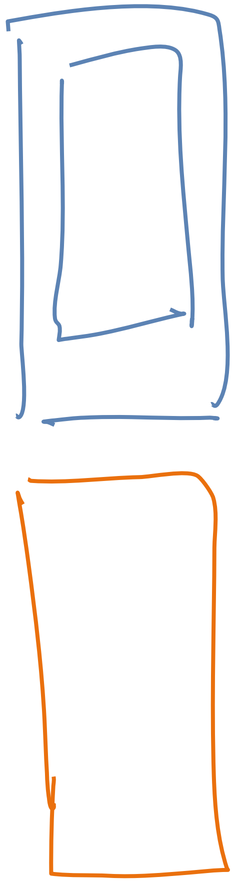
- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements
  -
    -

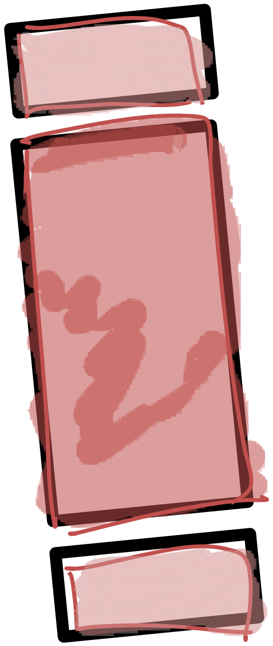


1

Further Challenges

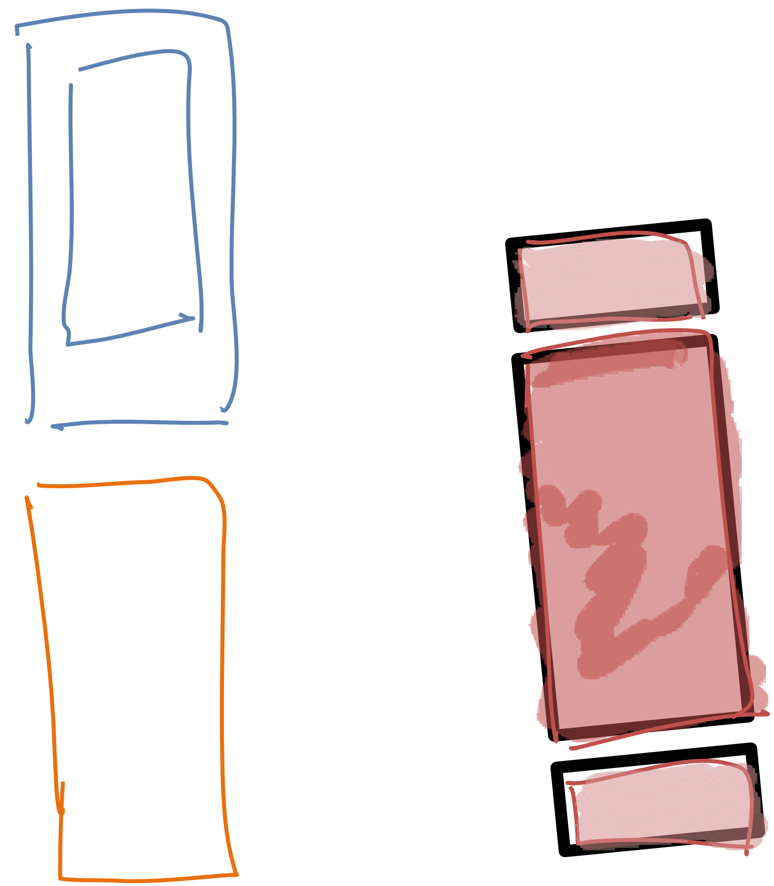
- Arrays
  - Nested loops
  - Consecutive loops
  - Boundary elements





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

vrije Universiteit

syssec

Key insight 2

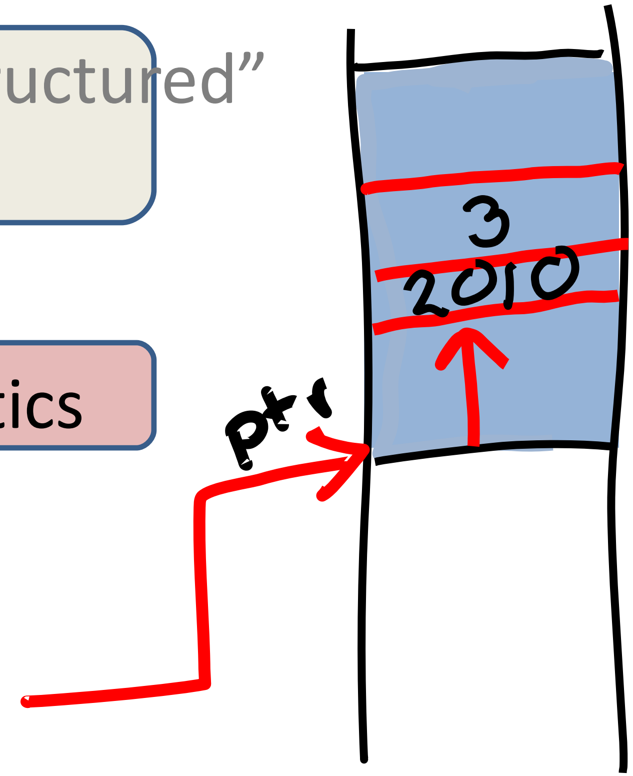
Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics

–

–



Key insight 2

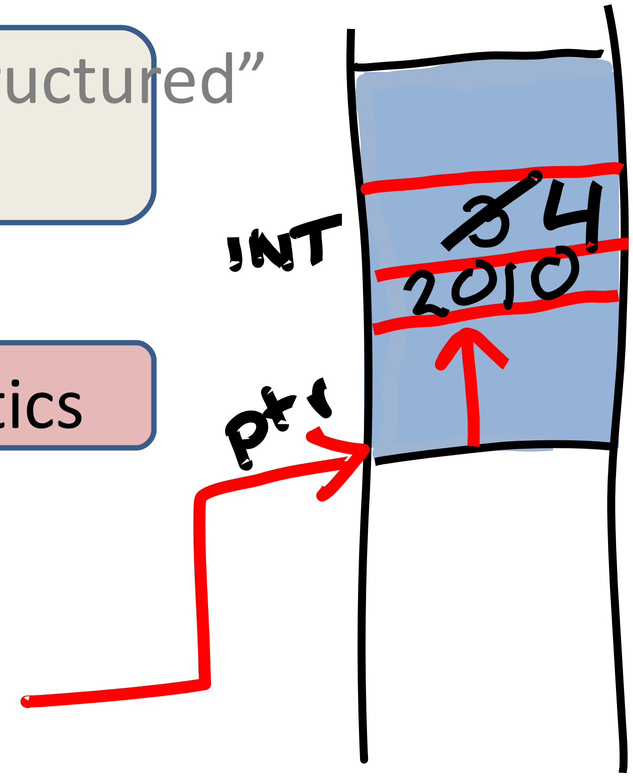
Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics

–

–



vrije Universiteit

syssec

Semantics: key insights

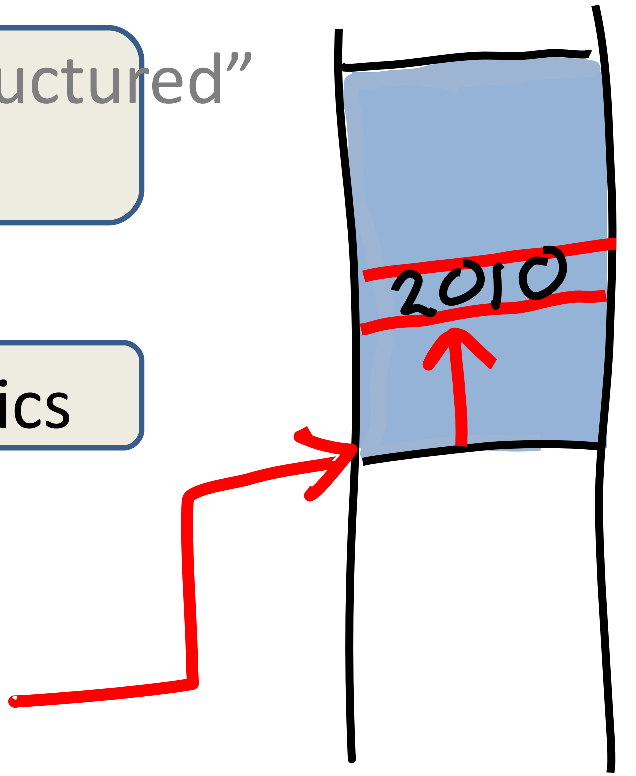
Yes, data is “apparently unstructured”

But usage is not!

Usage (again) reveals semantics

–

–

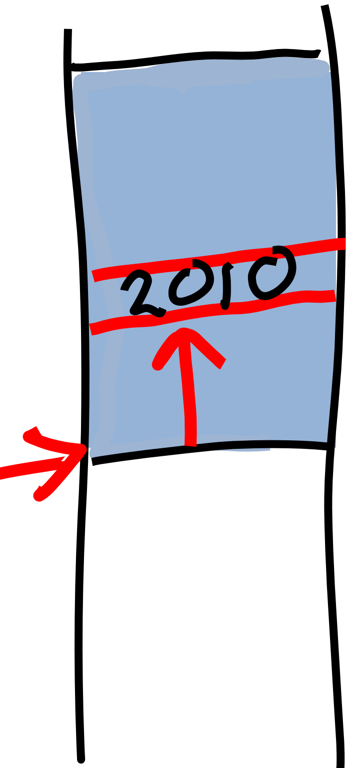


1

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!



Propagate types from sources + sinks

–

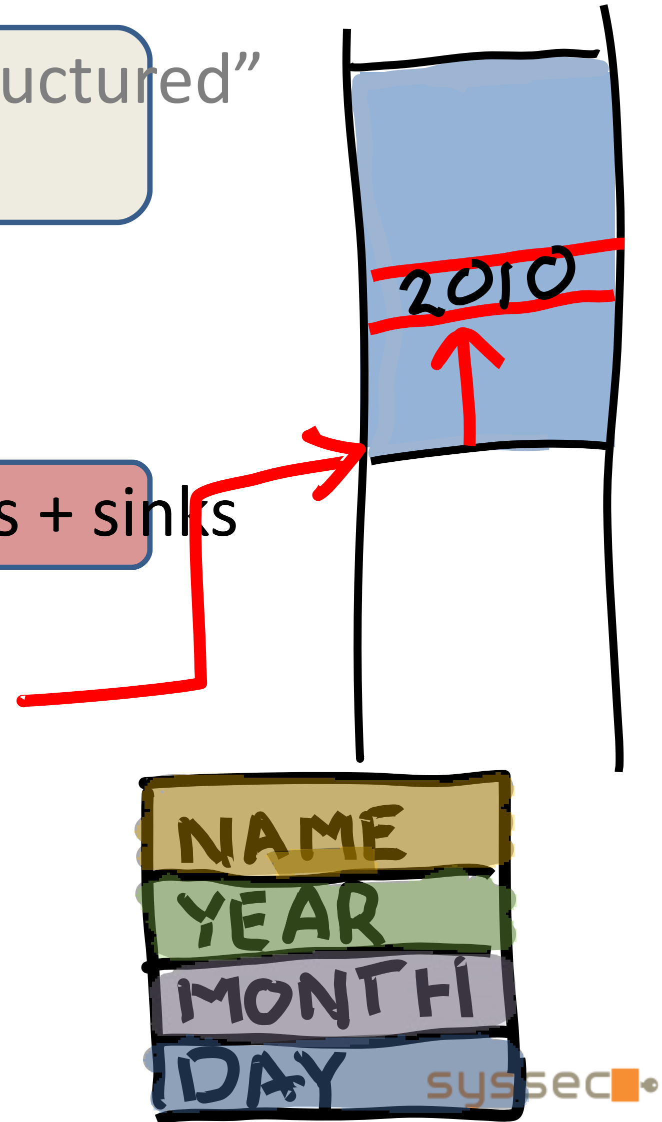
–

Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks



–

–

Key insight 3

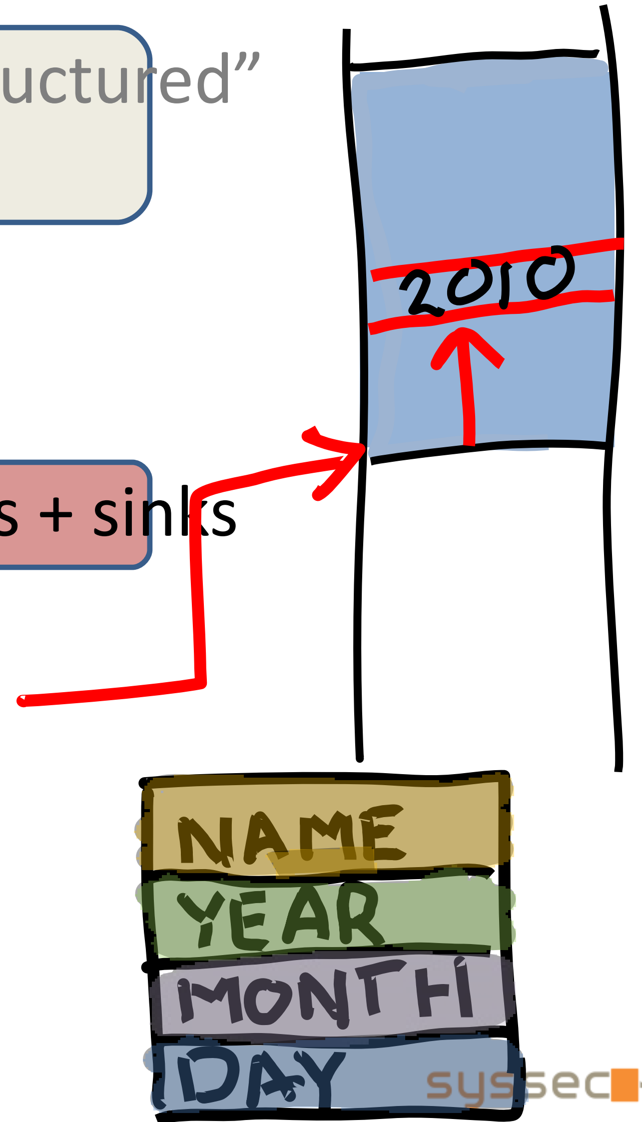
Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks

–

–

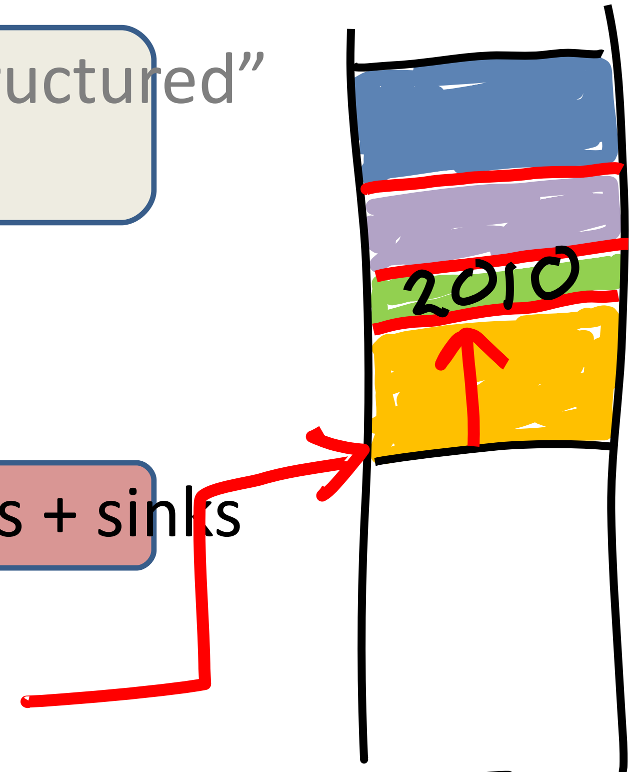


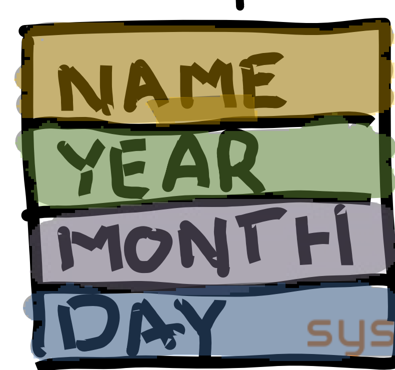
Key insight 3

Yes, data is "apparently unstructured"

But usage is not!

Propagate types from sources + sinks



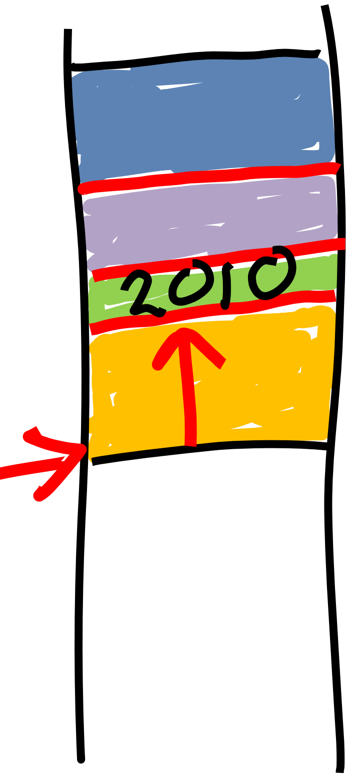


Key insight 3

Yes, data is “apparently unstructured”

But usage is not!

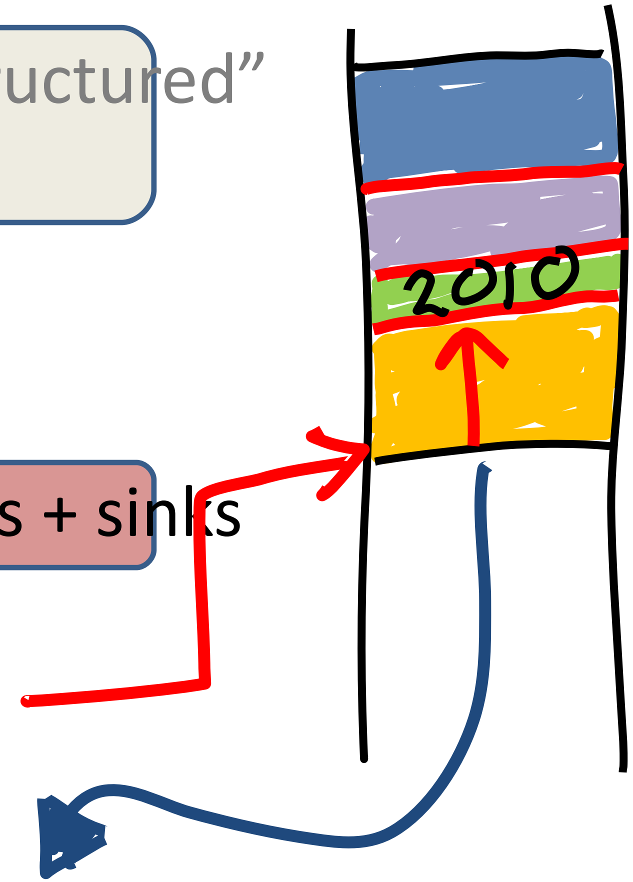
Propagate types from sources + sinks



Key insight 3

Yes, data is “apparently unstructured”

But usage is not!



Propagate types from sources + sinks

open (“Herbert.doc”, R_ONLY)

1

Key insight 3

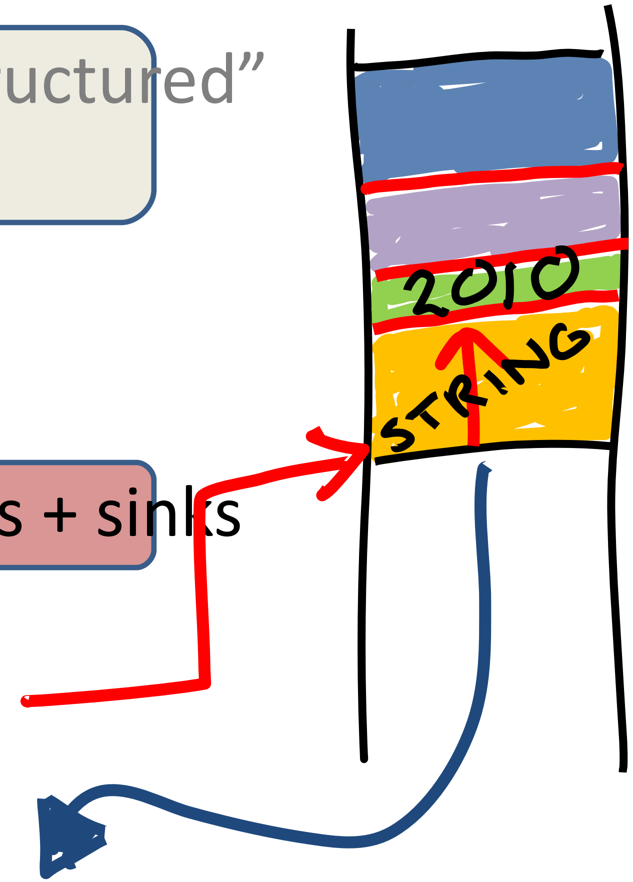
Yes, data is “apparently unstructured”

But usage is not!

Propagate types from sources + sinks

–

–



open (“Herbert.doc”, R_ONLY)

1

Results

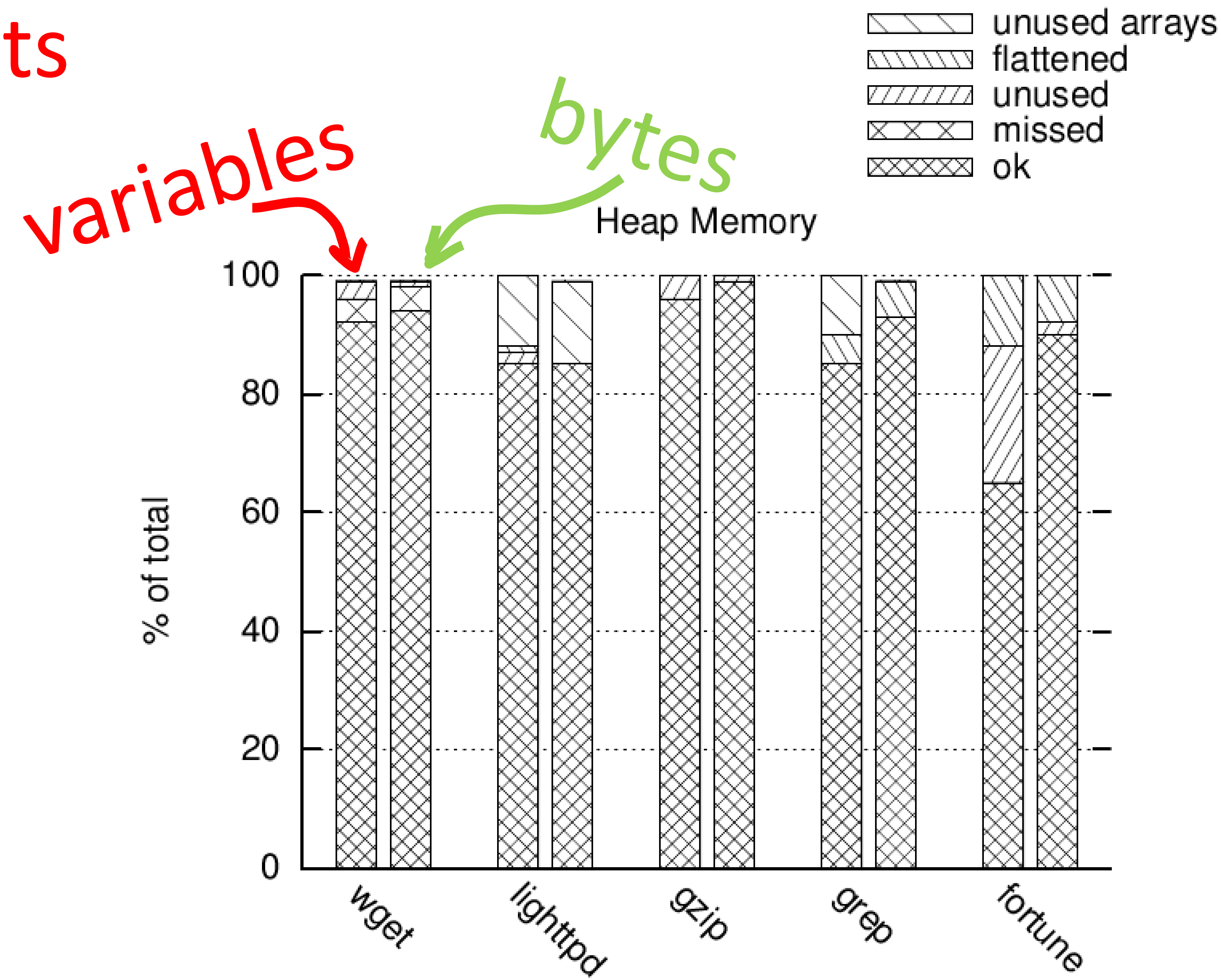
vrije Universiteit

syssec

1

Results

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

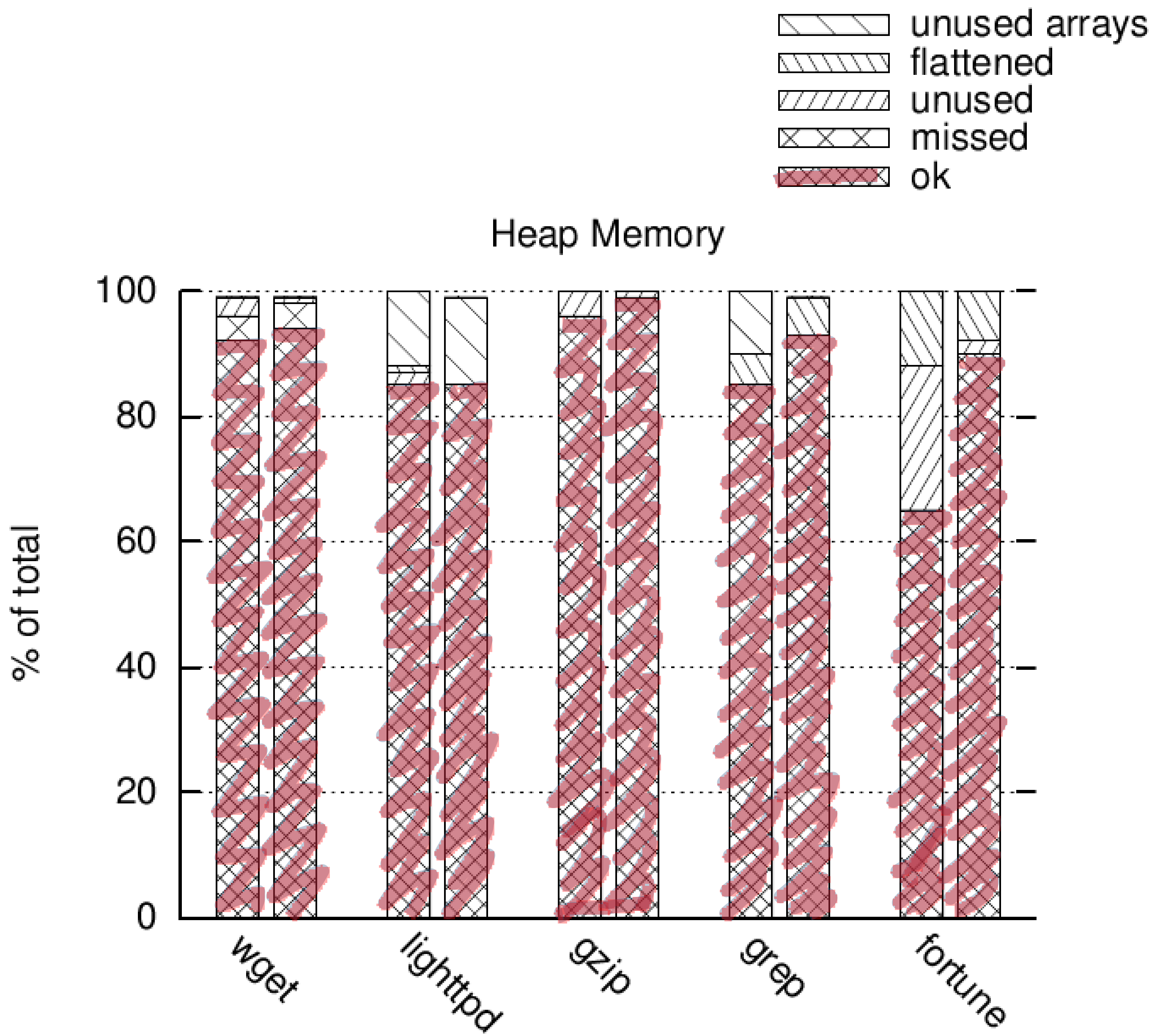


vrije Universiteit

syssec

Results

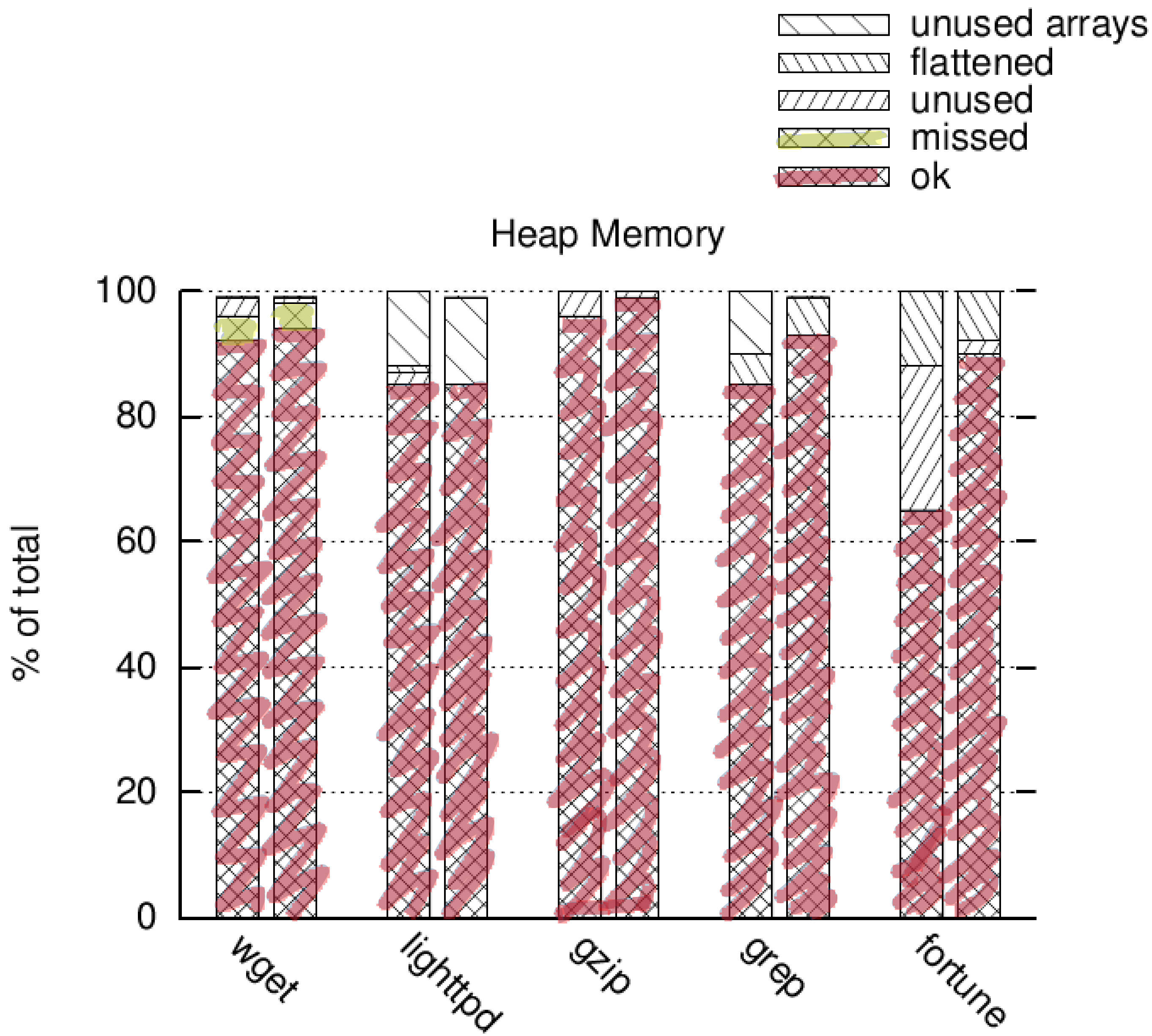
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

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



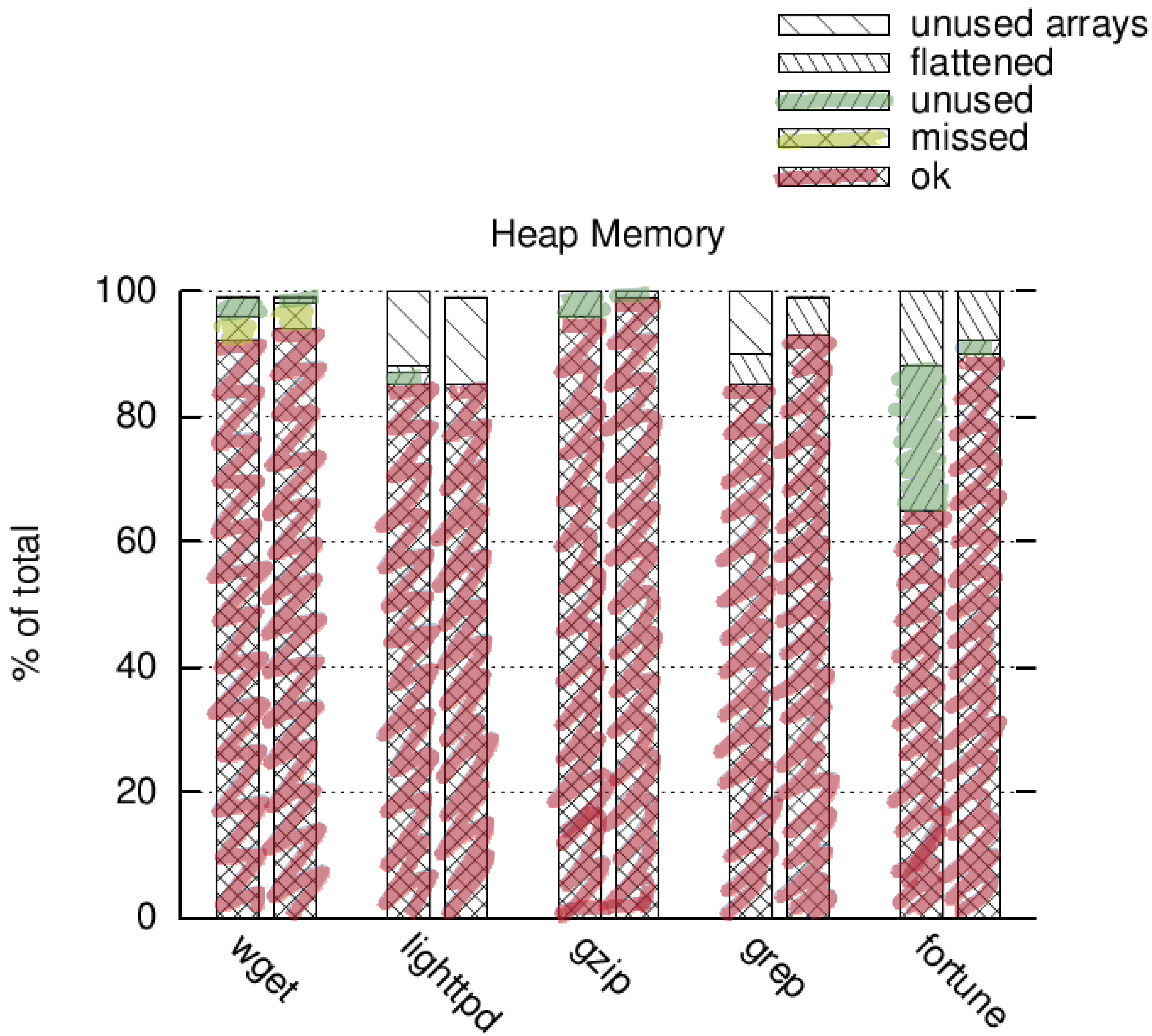
vrije Universiteit

syssec

1

Results

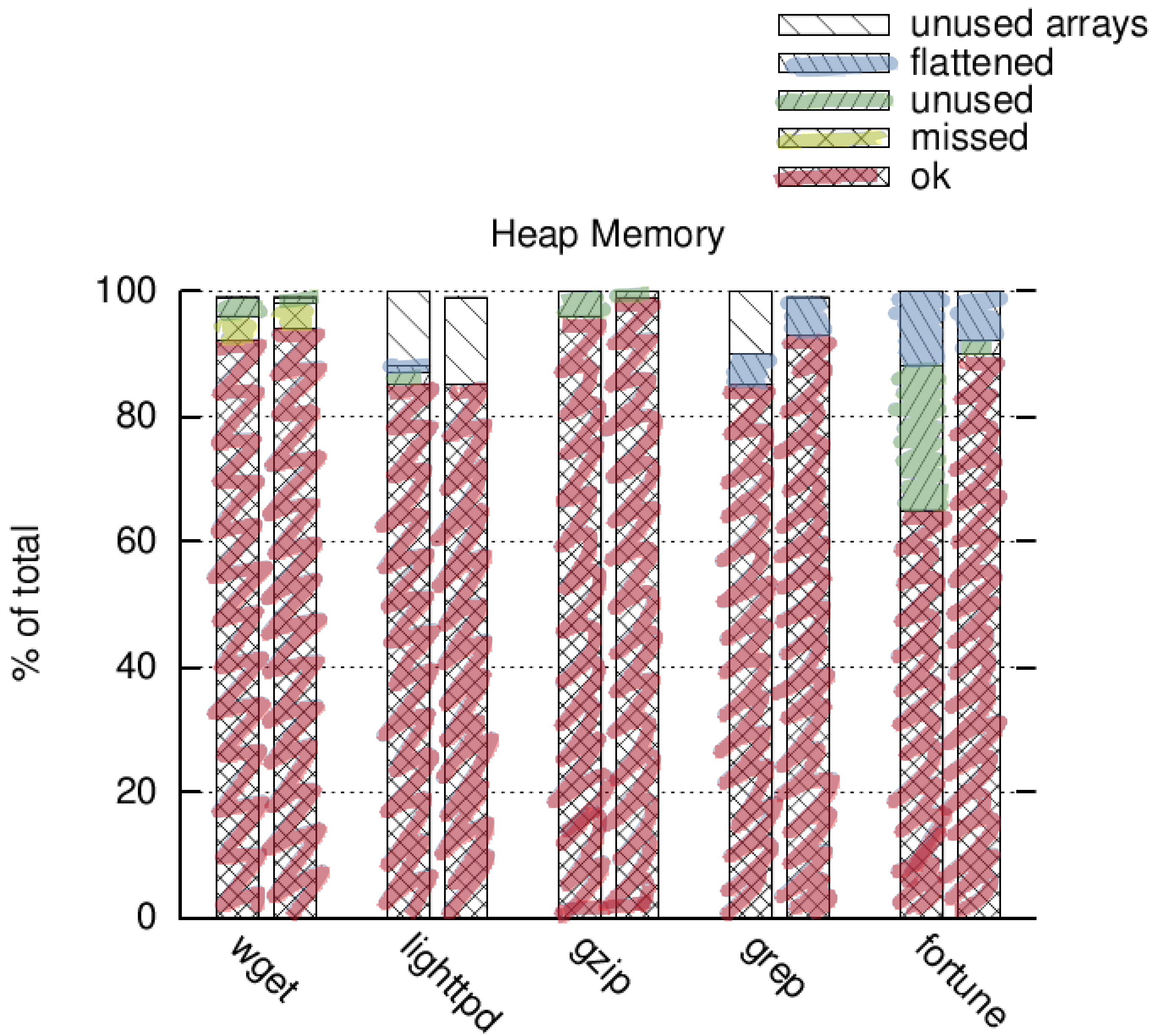
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

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



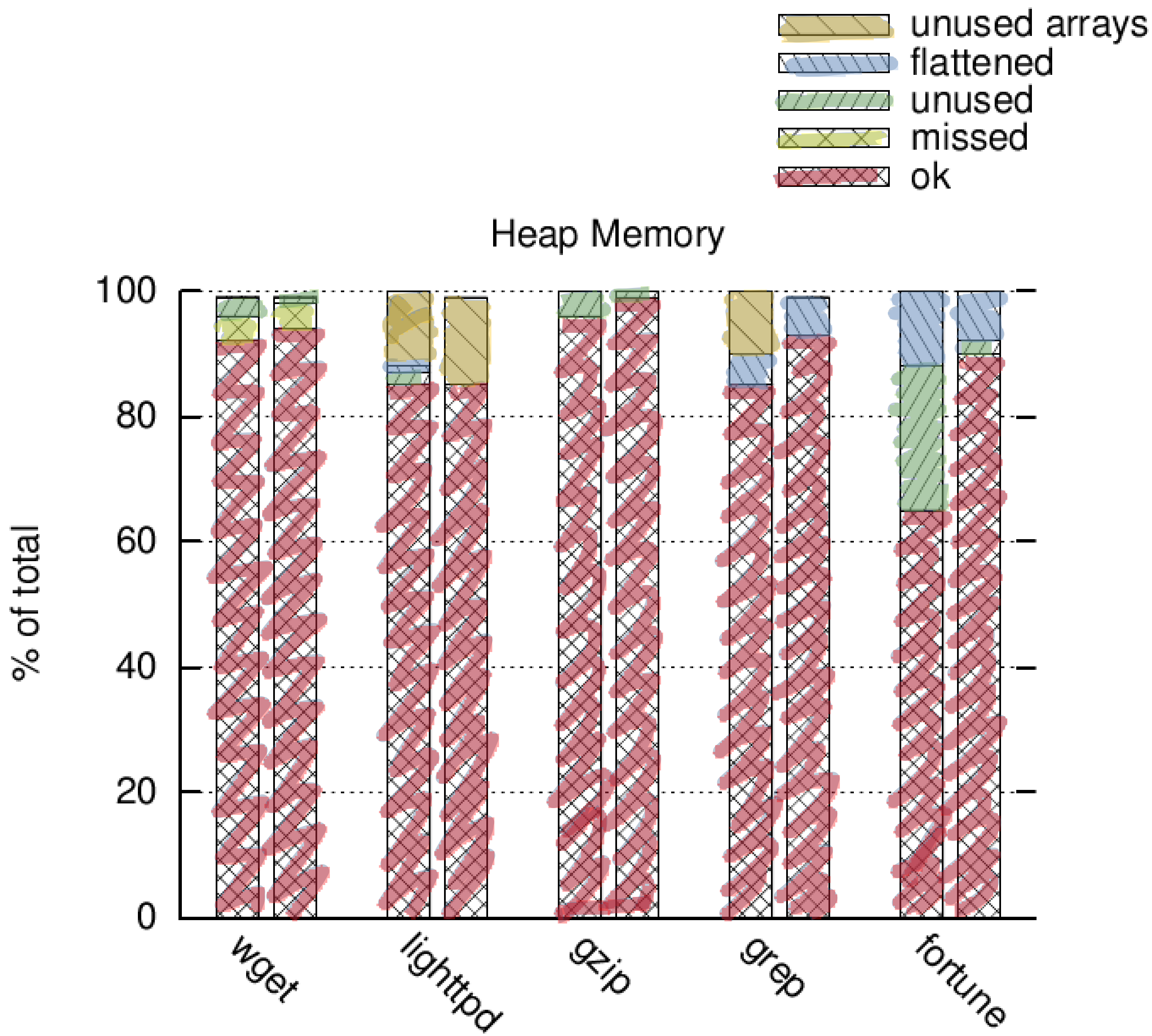
vrije Universiteit

syssec

1

Results

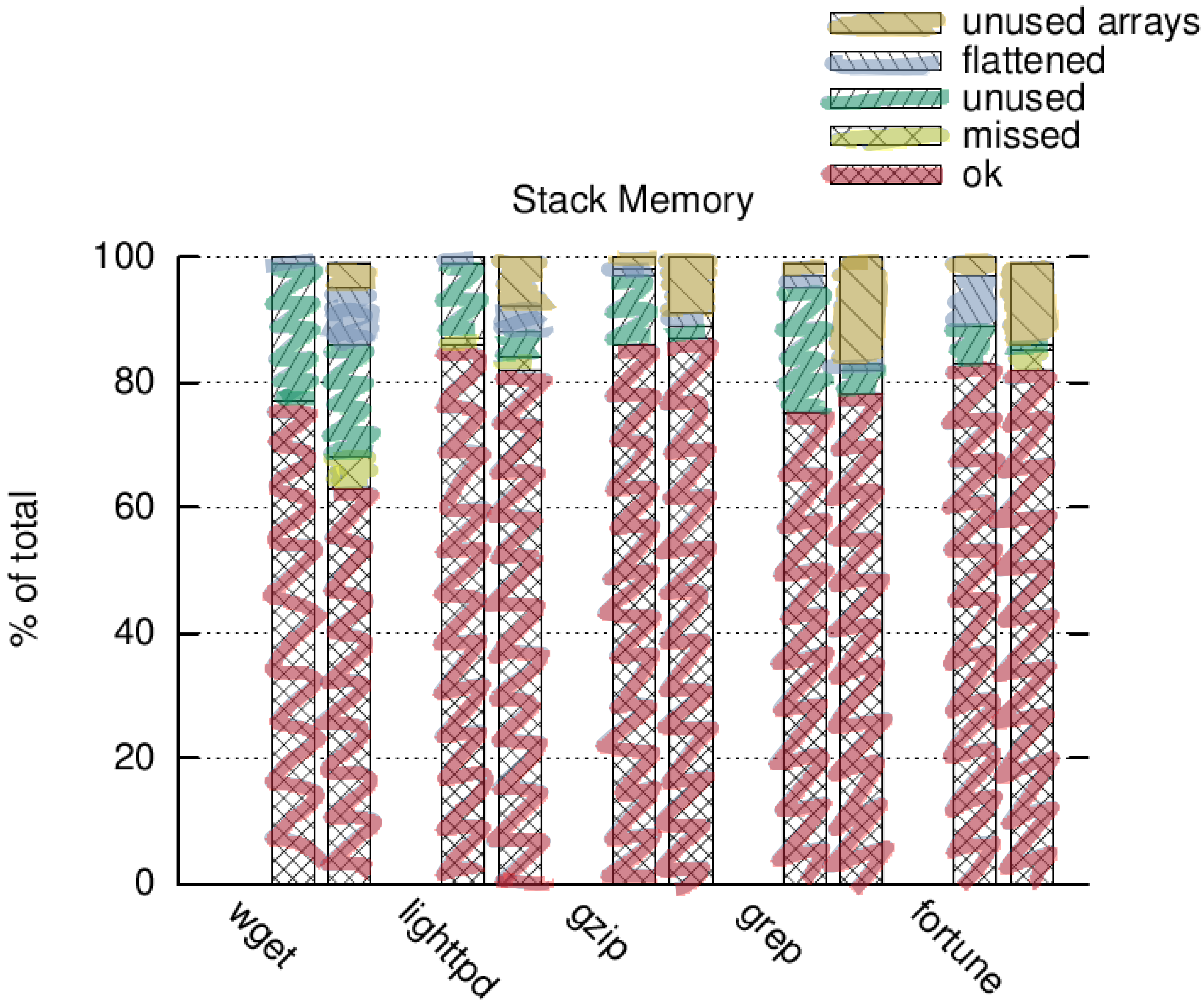
Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



1

Results

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

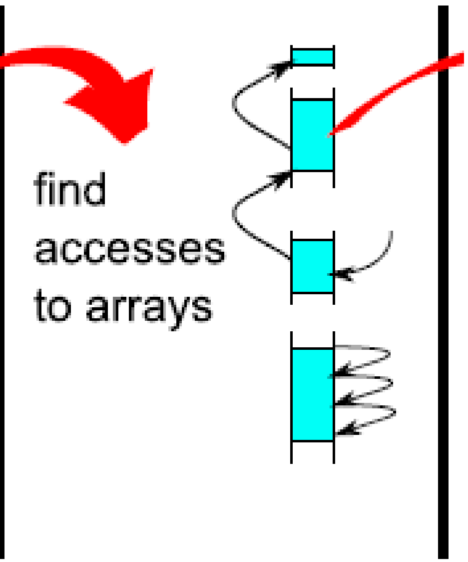


vrije Universiteit

syssec

Demo?

Step 2: find array accesses



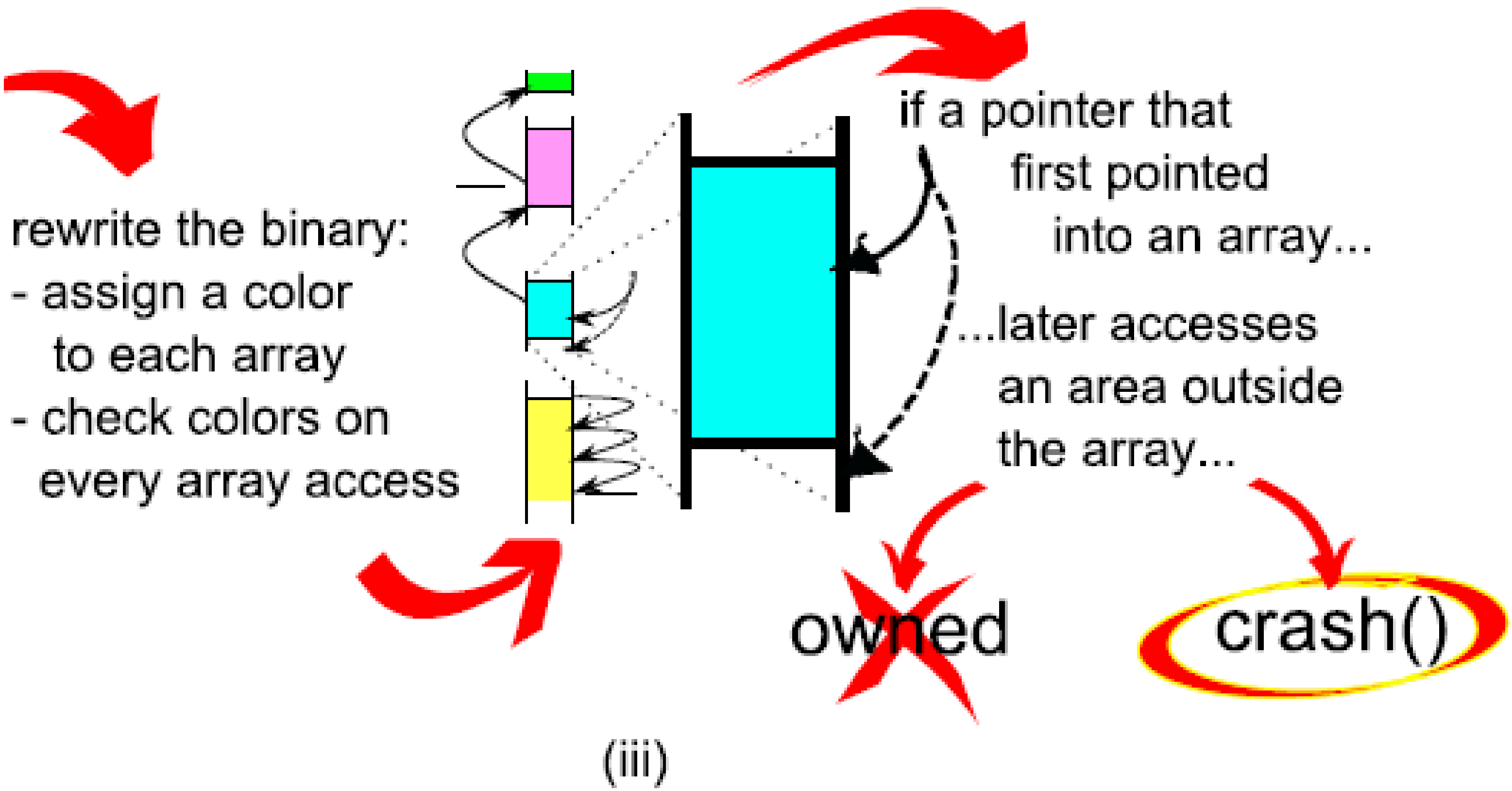
(ii)

In principle: very simple

– detect array accesses at runtime
– remember the instructions
–

Note: not complete

Step 3: rewrite the binary



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

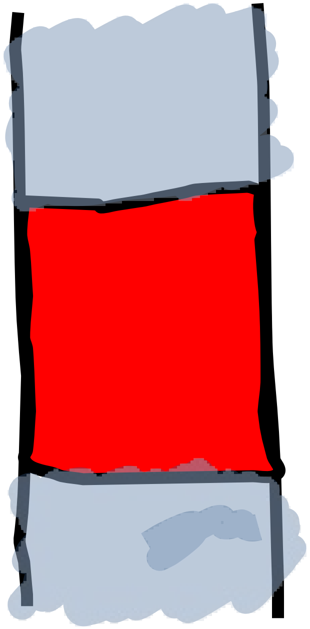
Focuses on the latter

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++;
}
```



vrije Universiteit

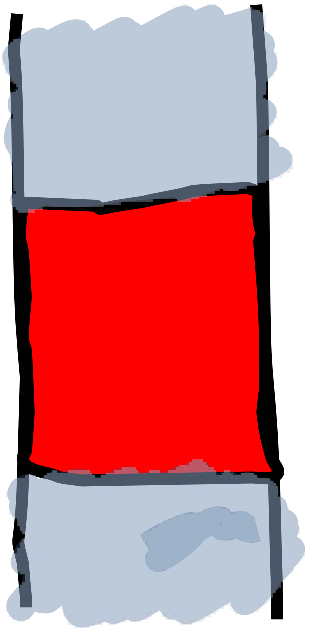
syssec

3

A colourful protection

- give all arrays a unique colour

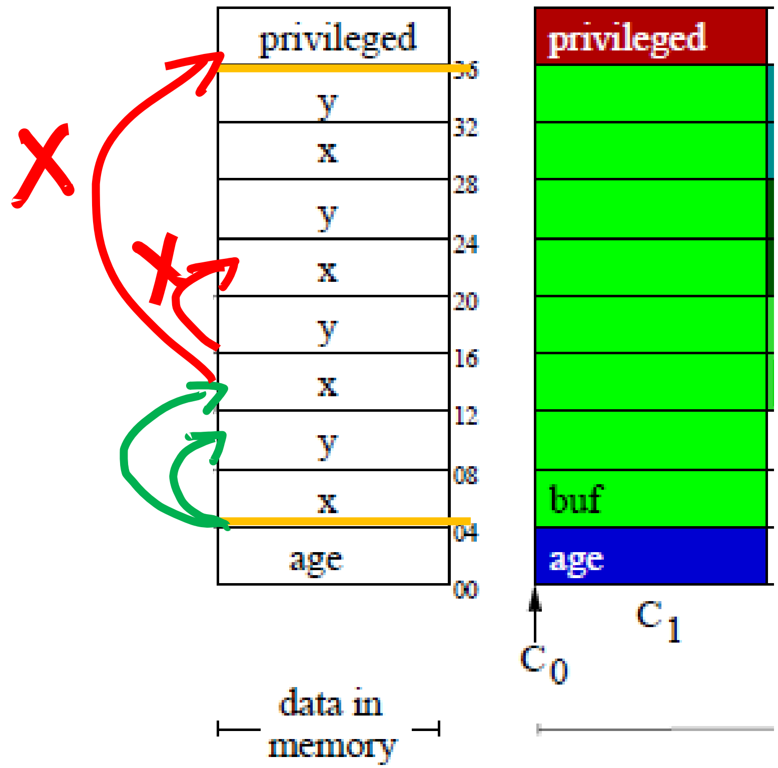
```
p = array;
ASSIGN pointer a colour
col(p) = RED
i = 0;
while(!stop)
{
   *(p + i) = 0;
   CHECK if colours match:
    mem_col(p+i) == col(p)?
   i++;
}
```



3

Reality requires subtle shades

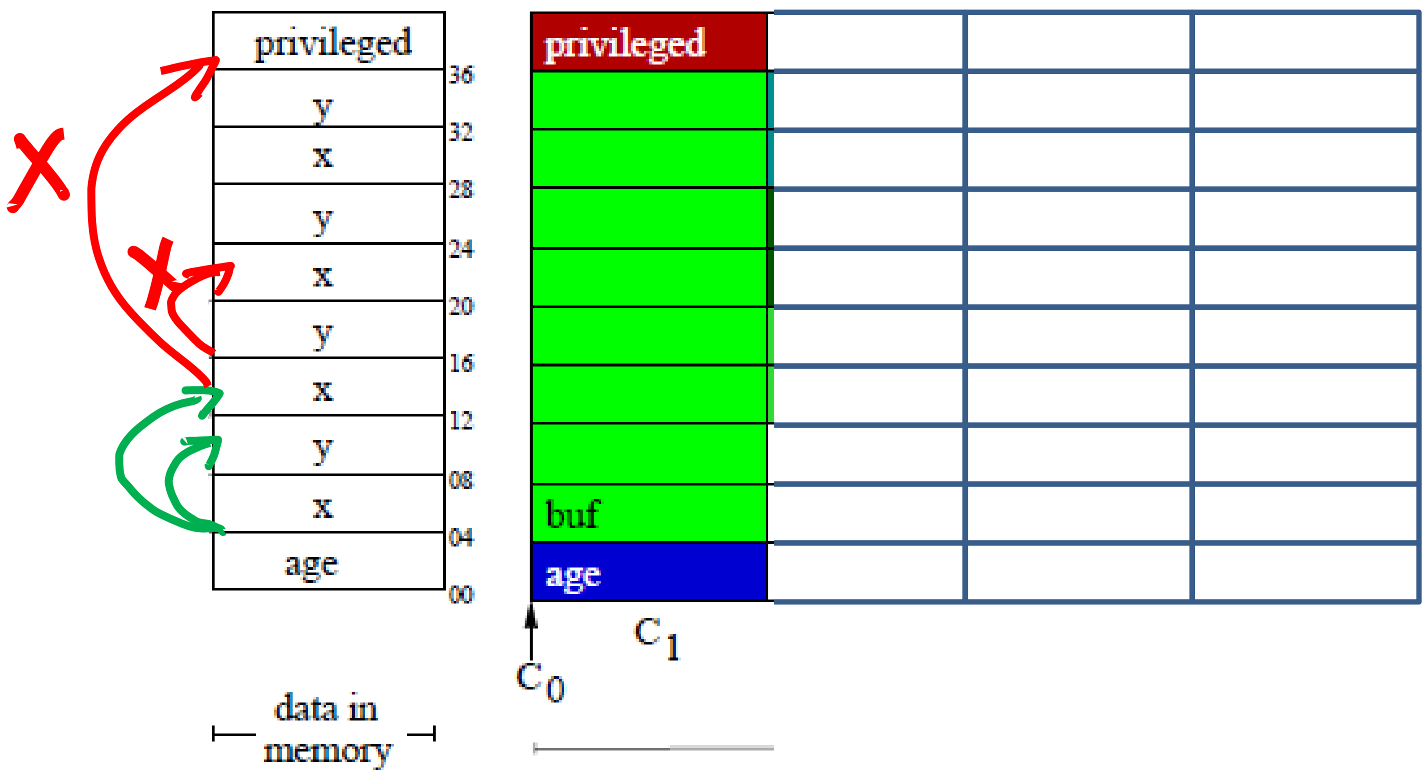
```
typedef struct pair {
    int x;
    int y;
} pair_t;

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



3

Reality requires subtle shades

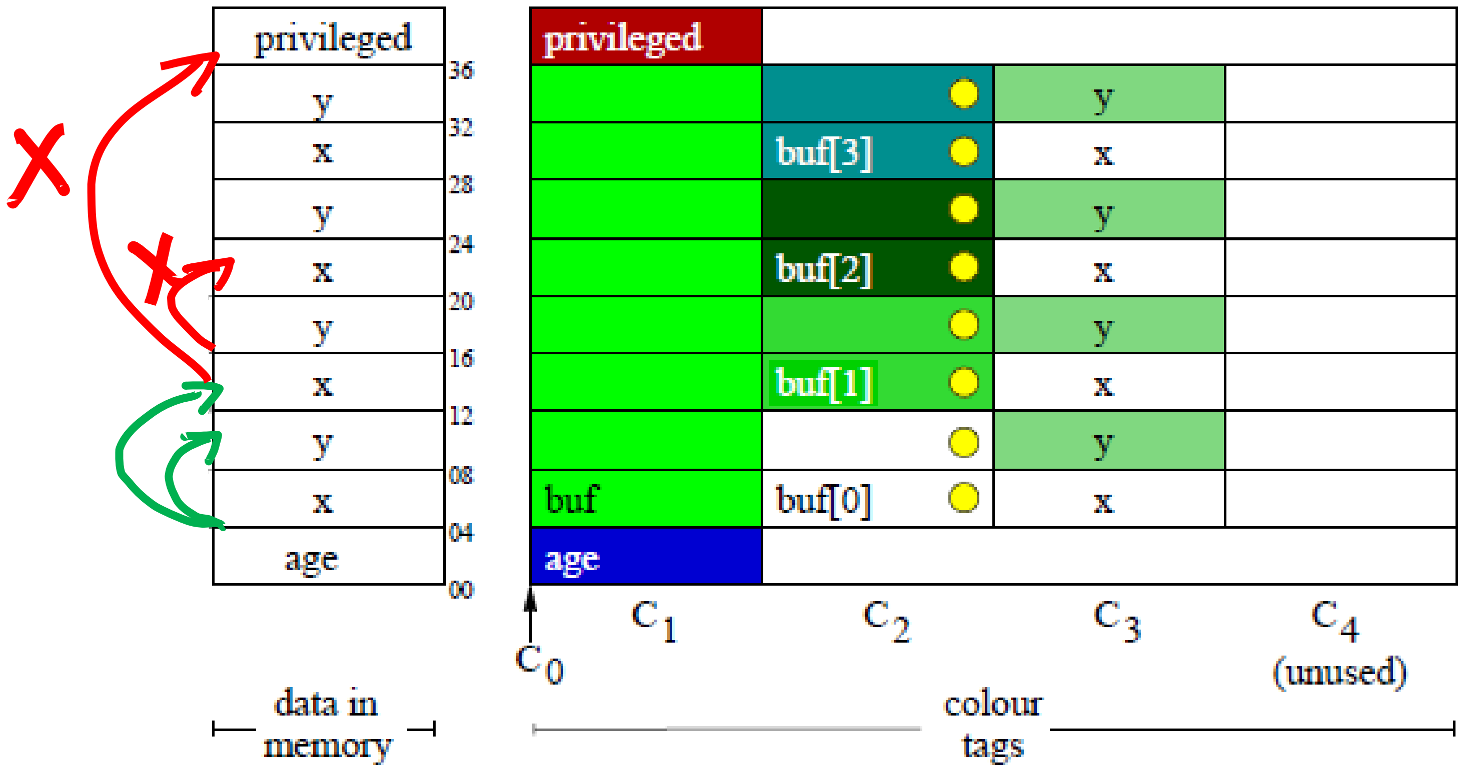
```
typedef struct pair {
    int x;
    int y;
} pair_t;

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



3

Reality requires subtle shades

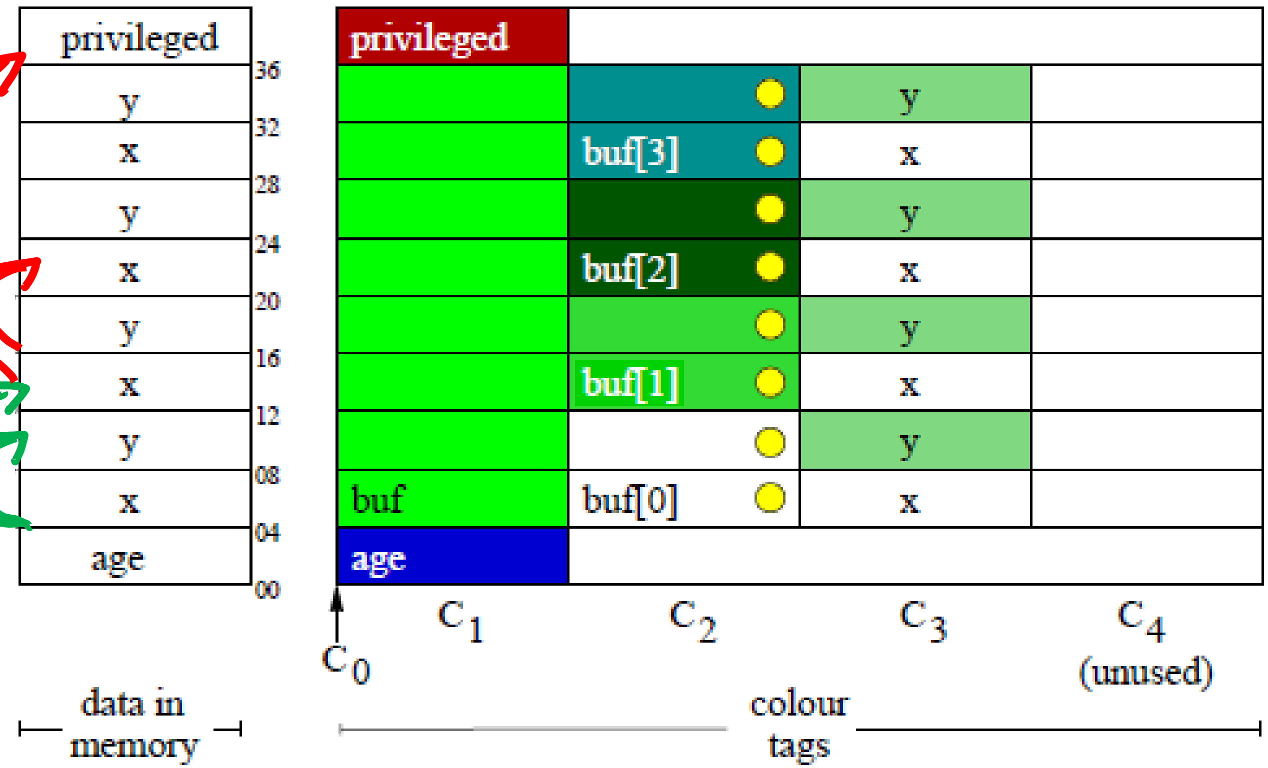
```
typedef struct pair {
    int x;
    int y;
} pair_t;
struct s {
    int  age;
    pair_t buf[4];
    int  privileged;
};
```



3

In reality

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

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

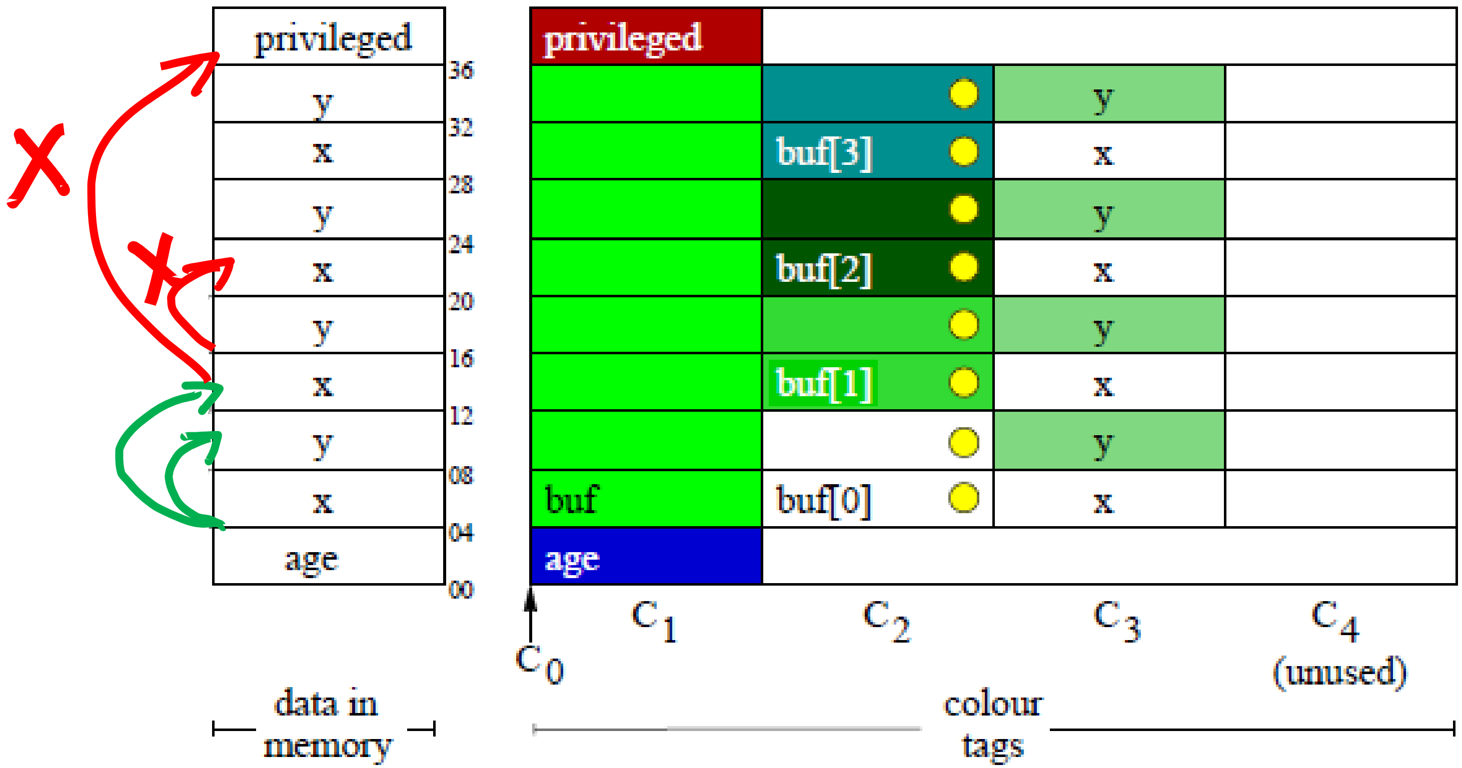


Check: does the pointer colour match that of the location pointed to?
(left to right, in all shades, with blanks serving as wild cards)

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;
};
```



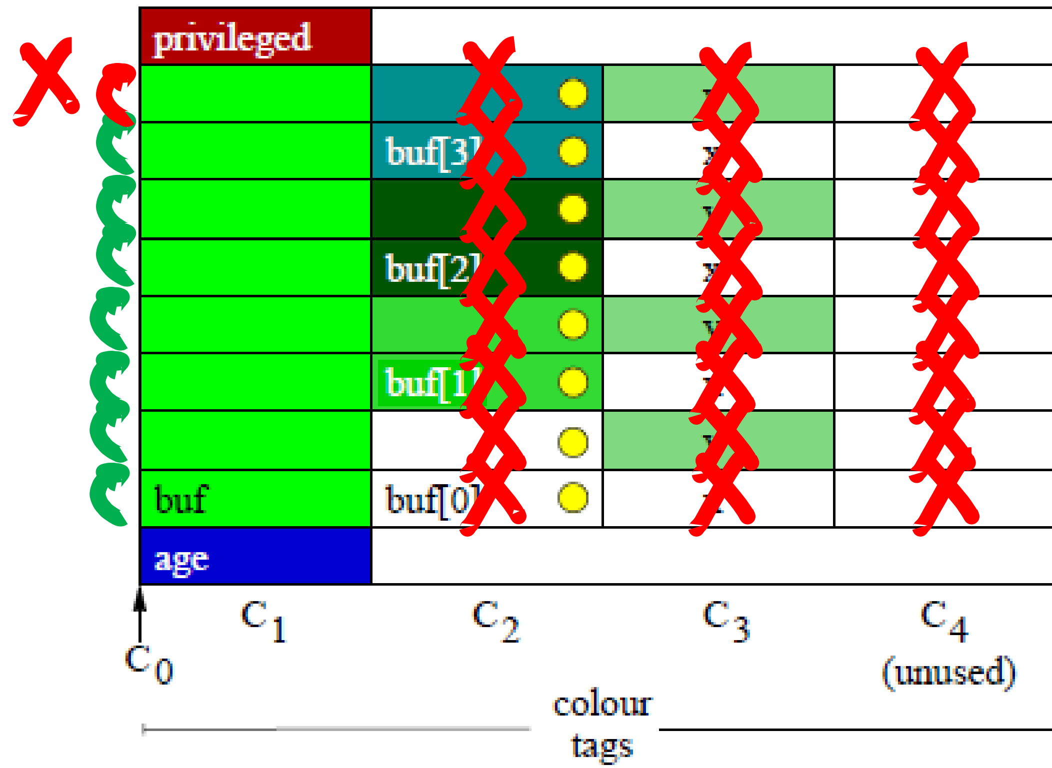

```
int *p;
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```

vrije Universiteit

syssec

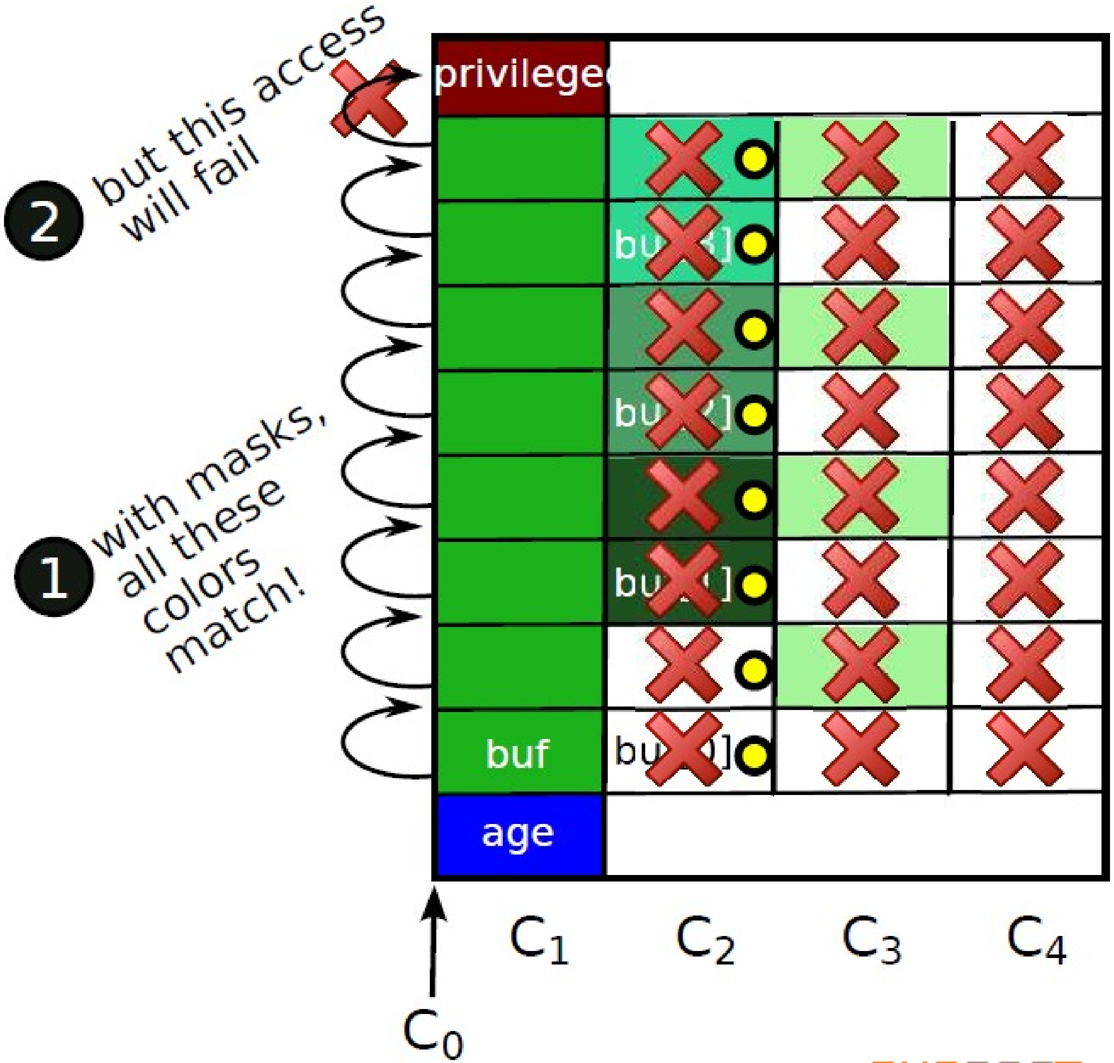
So we mask some shades

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

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

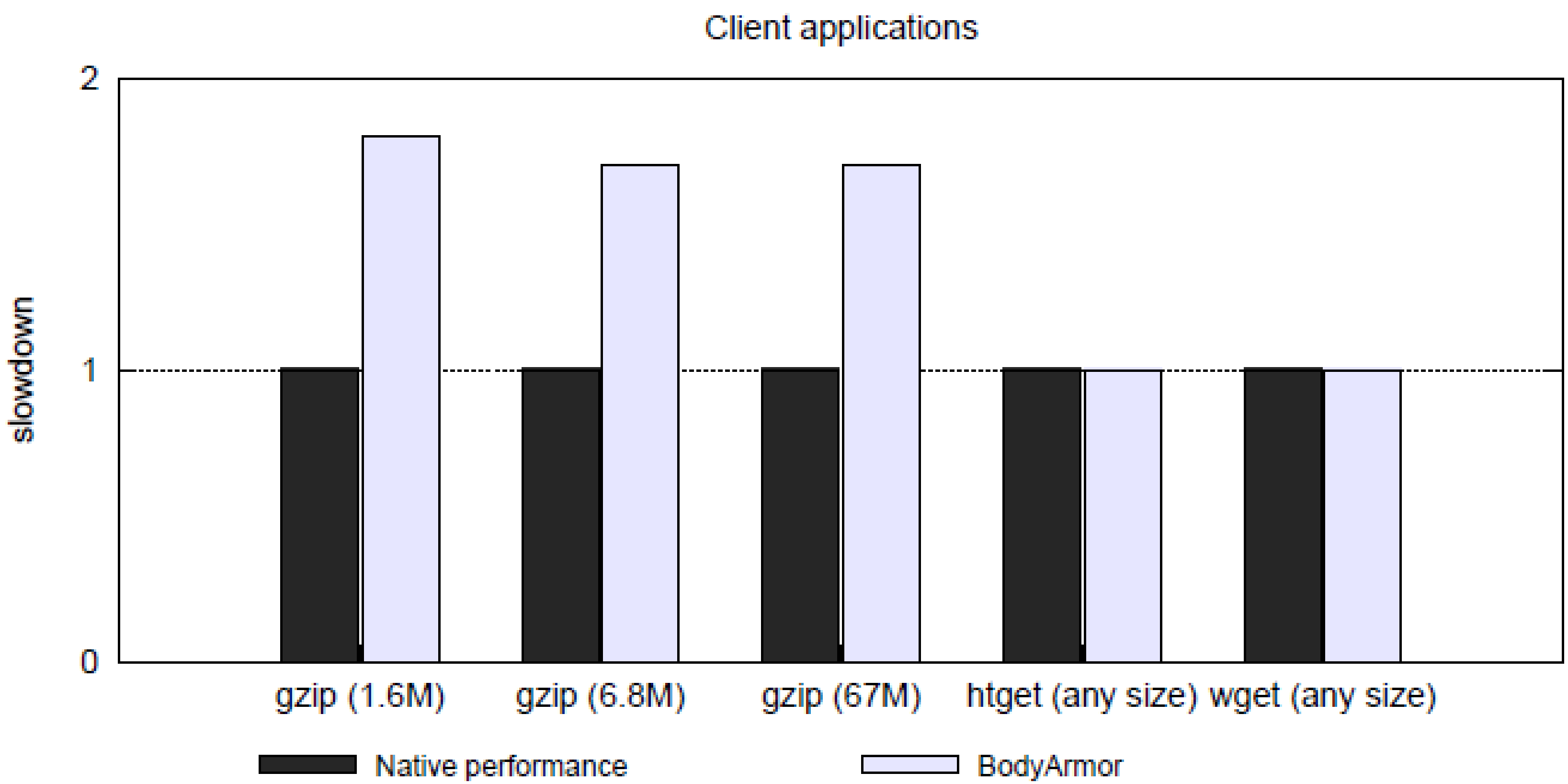


So we mask some shades

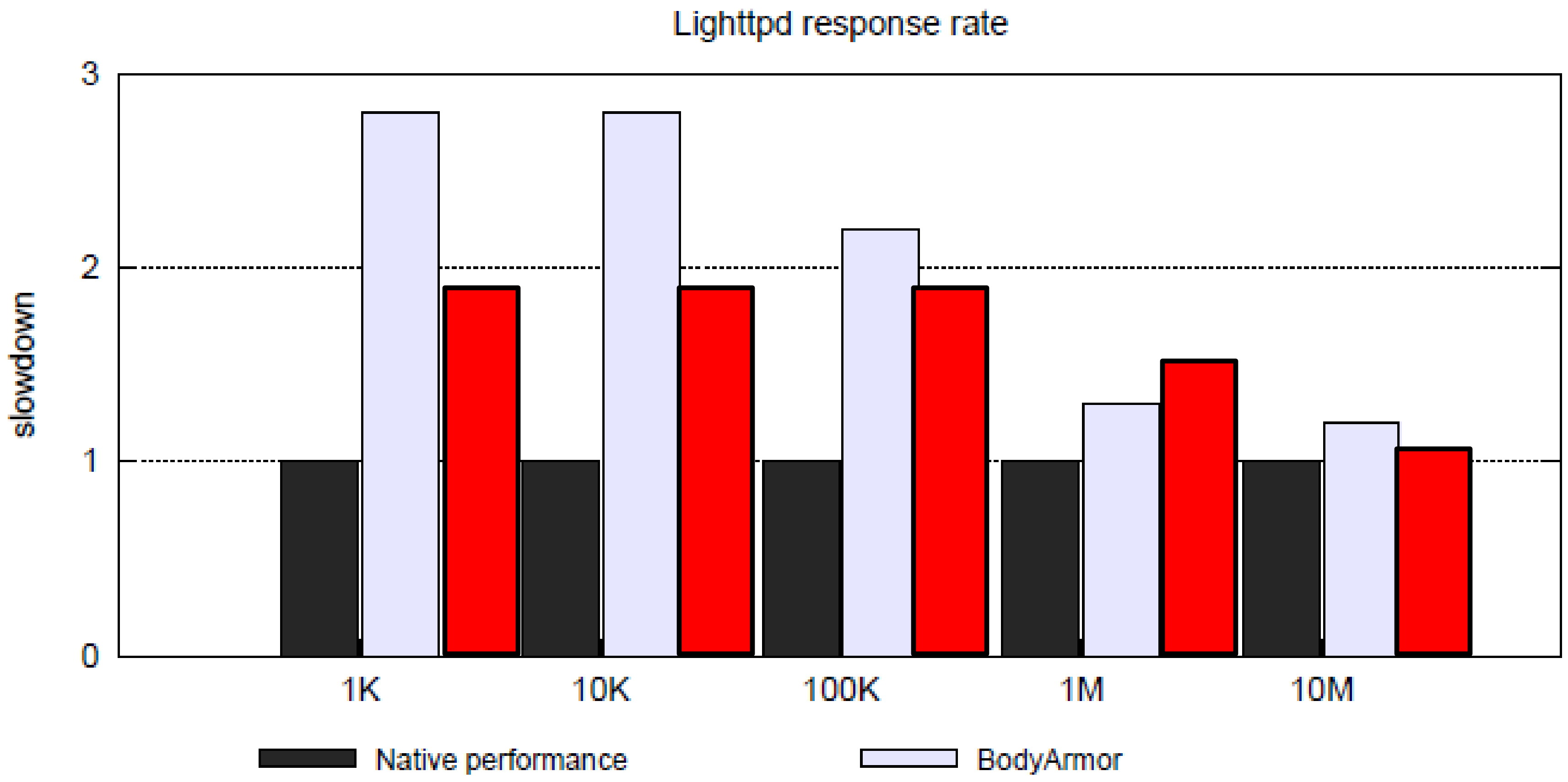
```
/* initialize the buffer
int *p;
int len = 4; //buf length

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



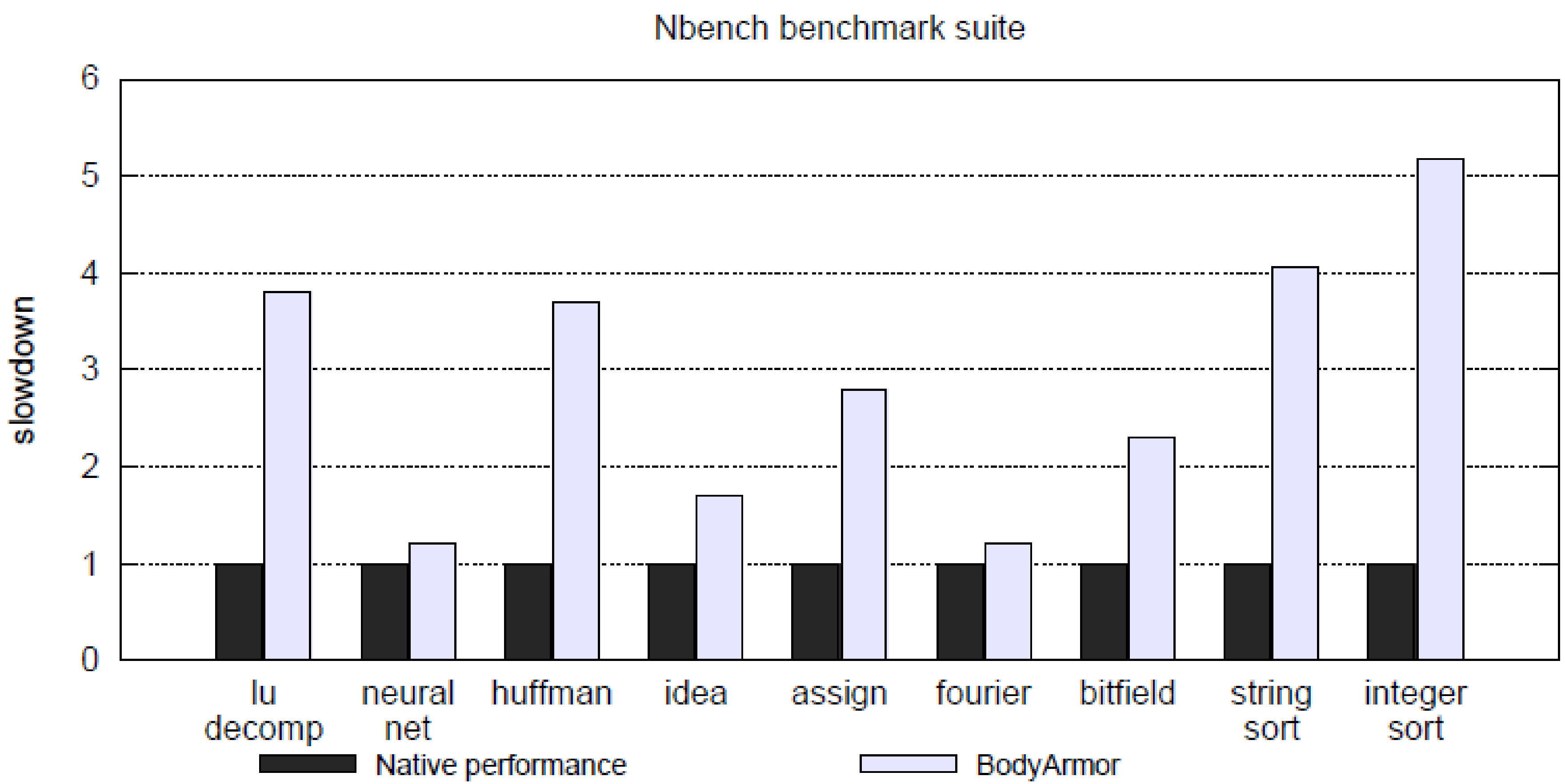
Performance?



Performance?



Performance?



overall: 2.9

1 2 3

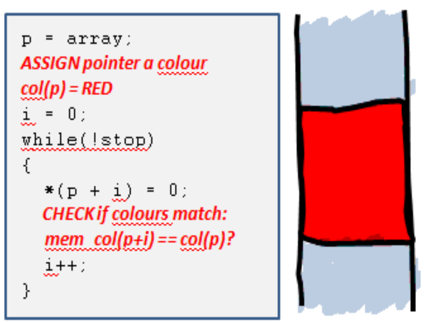
Effectiveness?

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

- Memory errors
  - are not going to go away
- BinArmor
  - protect against attacks on non-control data
  - few (if any) FPs
  - expensive
  - not fully optimised yet!



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

vrije Universiteit

syssec