# Effective Memory Protection Using Dynamic Tainting

James Clause
Alessandro Orso
(software)

and

Ioanis Doudalis
Milos Prvulovic
(hardware)

College of Computing
Georgia Institute of Technology

Supported in part by:
NSF awards CCF-0541080 and CCR-0205422 to Georgia Tech,
DHS and US Air Force Contract No. FA8750-05-2-0214

# Illegal memory accesses (IMA)

```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```

Memory

# Illegal memory accesses (IMA)

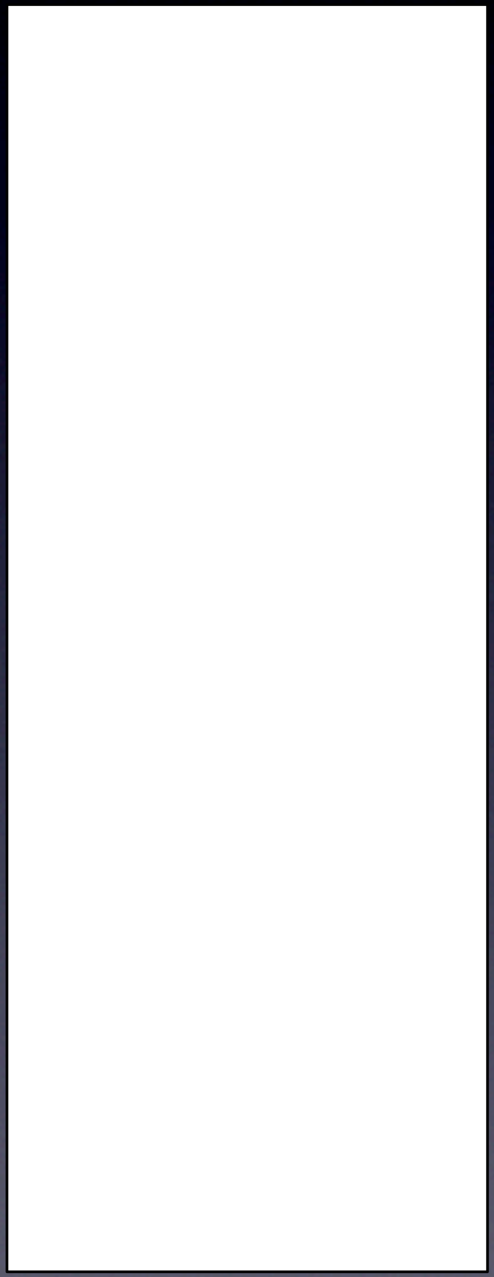
```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```

Memory

# Illegal memory accesses (IMA)

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```

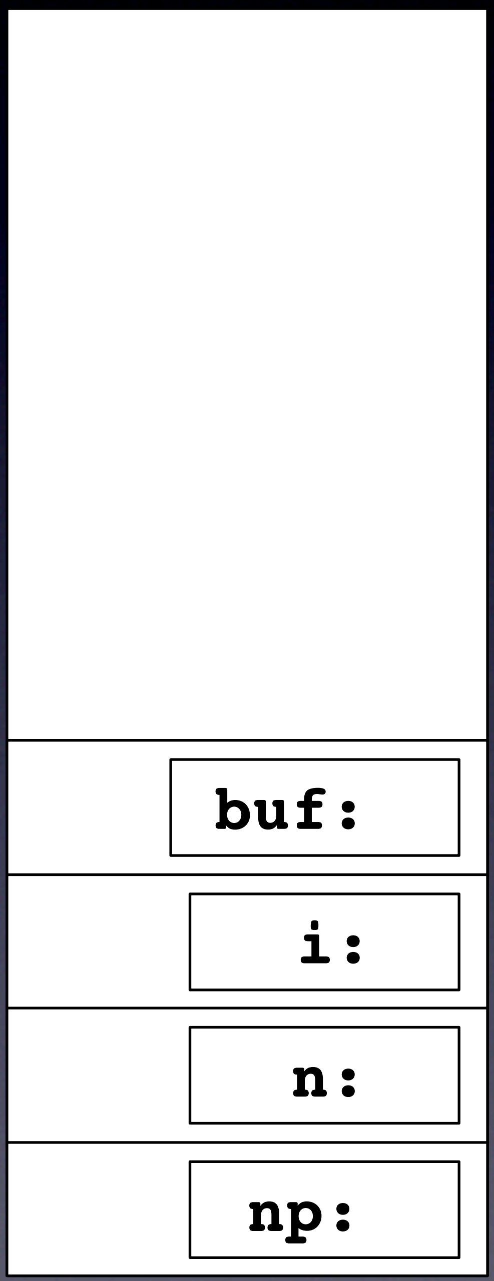
Memory



# Illegal memory accesses (IMA)

```
void main() {
int *np, n, i, *buf;

np = &n;

printf(“Enter size: “);
scanf(“%d”, np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

Memory



# Illegal memory accesses (IMA)

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```

Memory

# Illegal memory accesses (IMA)

```c
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

# Illegal memory accesses (IMA)

```
   void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```

Memory

# Illegal memory accesses (IMA)

```c
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

Memory

# Illegal memory accesses (IMA)

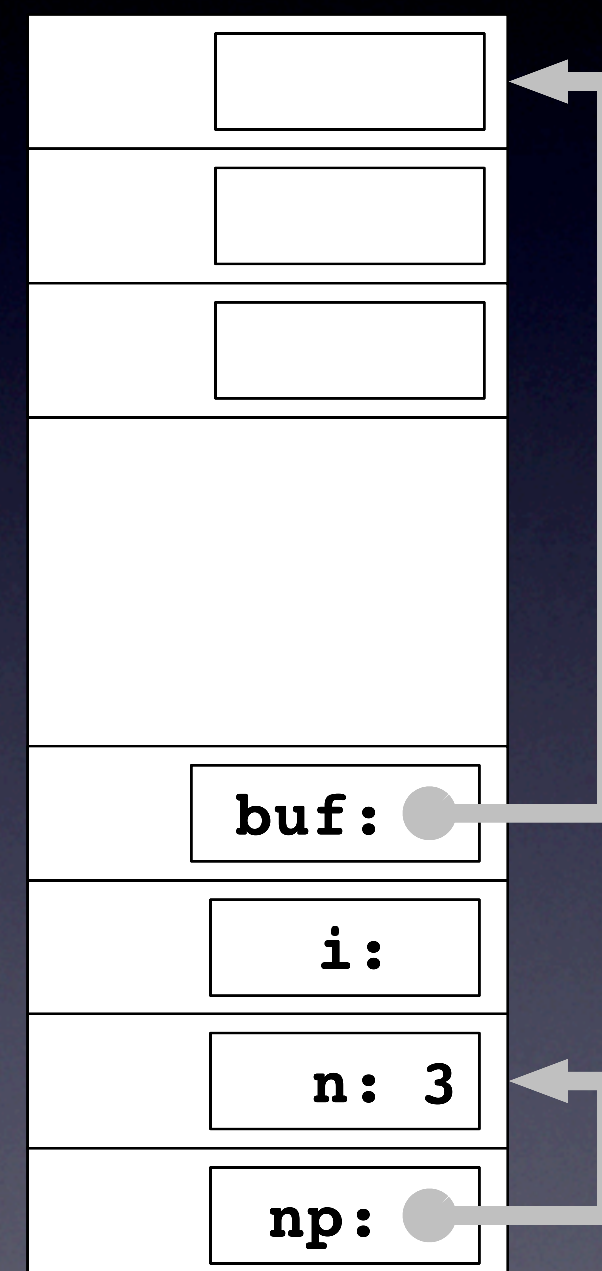
```c
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```

Memory

# Illegal memory accesses (IMA)

```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```

Memory



# Illegal memory accesses (IMA)

```
void main() {
int *np, n, i, *buf;

np = &n;

printf(“Enter size: “);
scanf(“%d”, np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

# Illegal memory accesses (IMA)

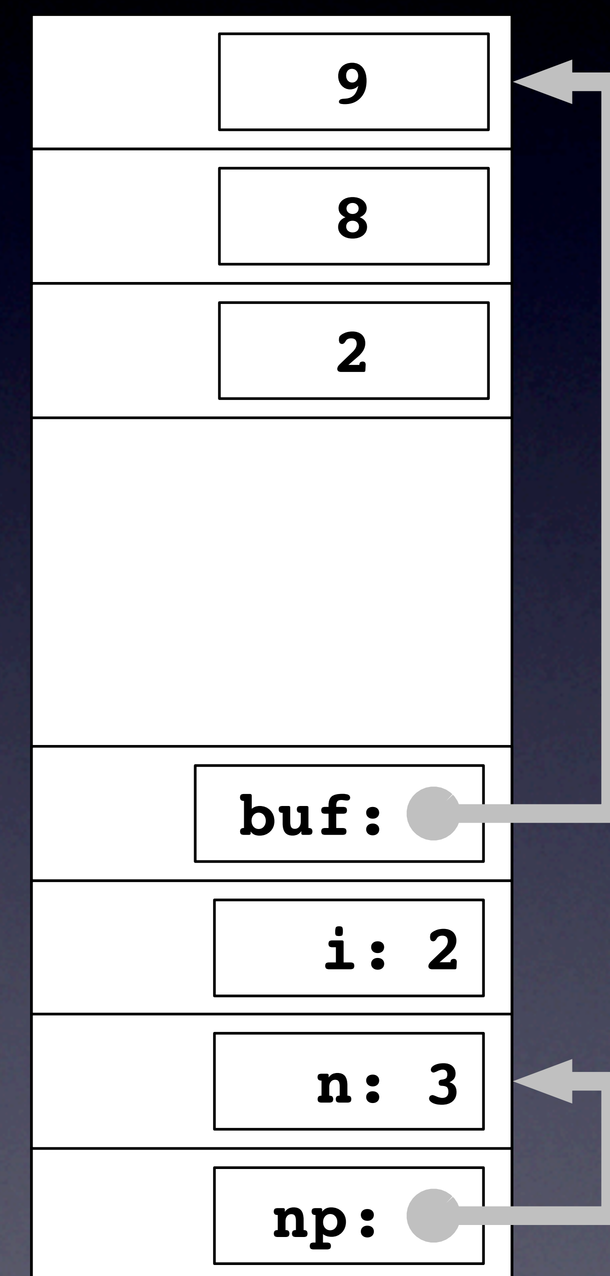
```
  void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
   *(buf + i) = rand()%10;
   ...
}
```

Memory

# Illegal memory accesses (IMA)

```
void main() {
int *np, n, i, *buf;

np = &n;

printf(“Enter size: “);
scanf(“%d”, np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

Memory

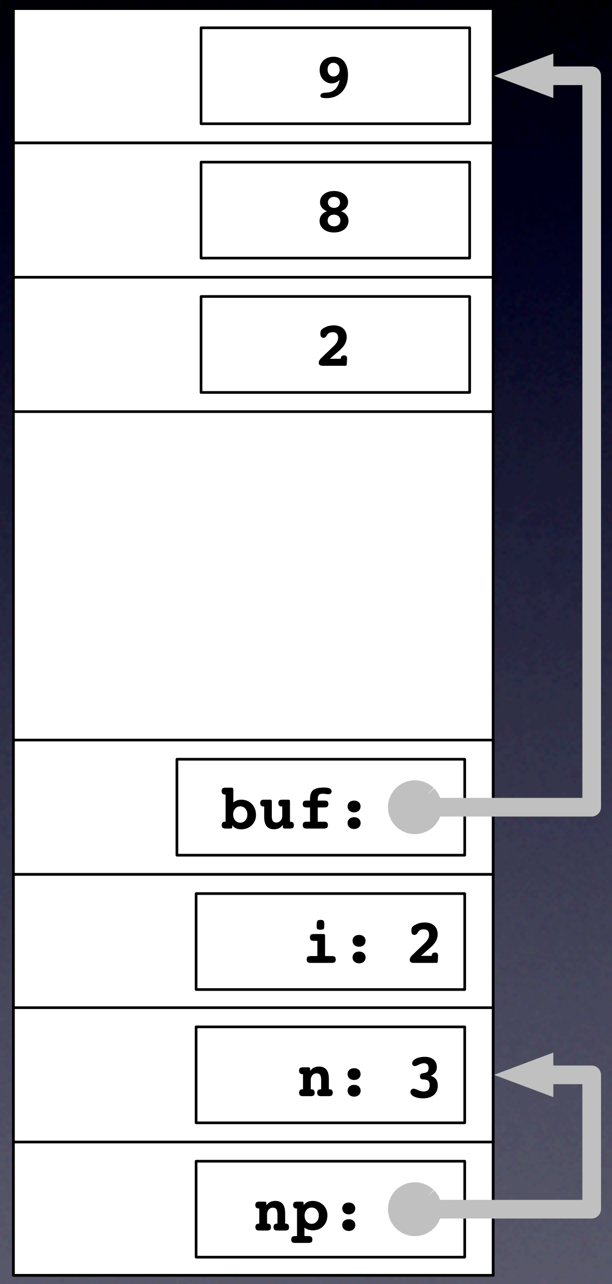


# Illegal memory accesses (IMA)

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

i <= n ➜ i < n

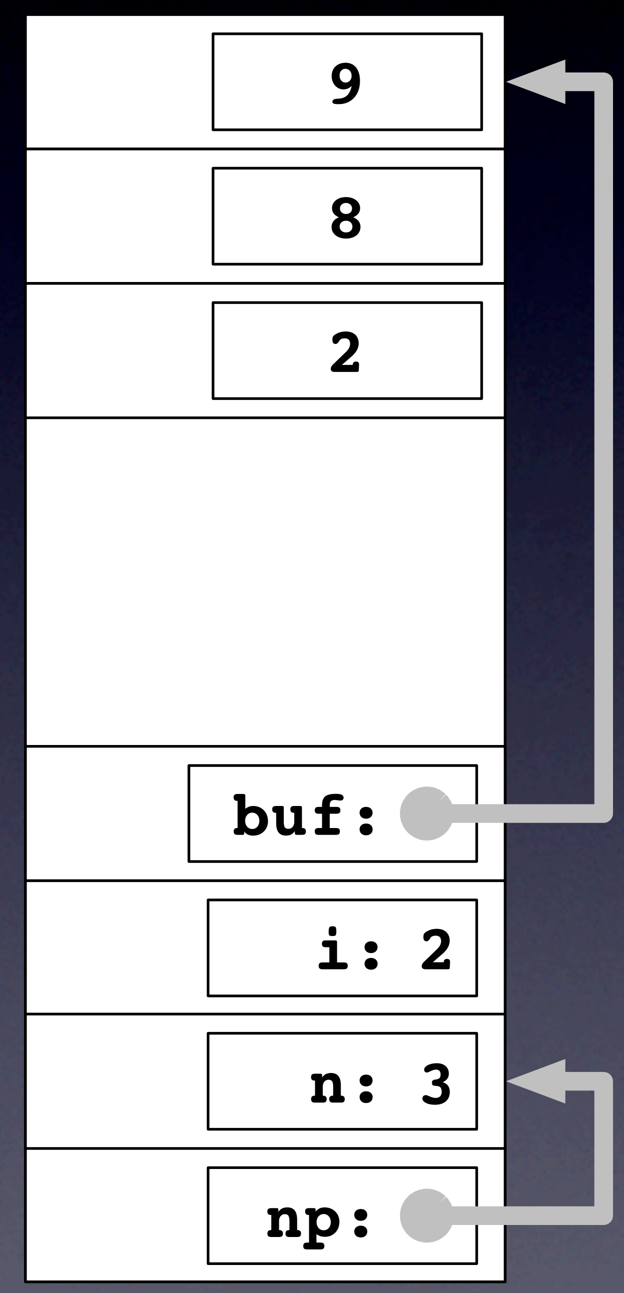
Memory



# Illegal memory accesses (IMA)

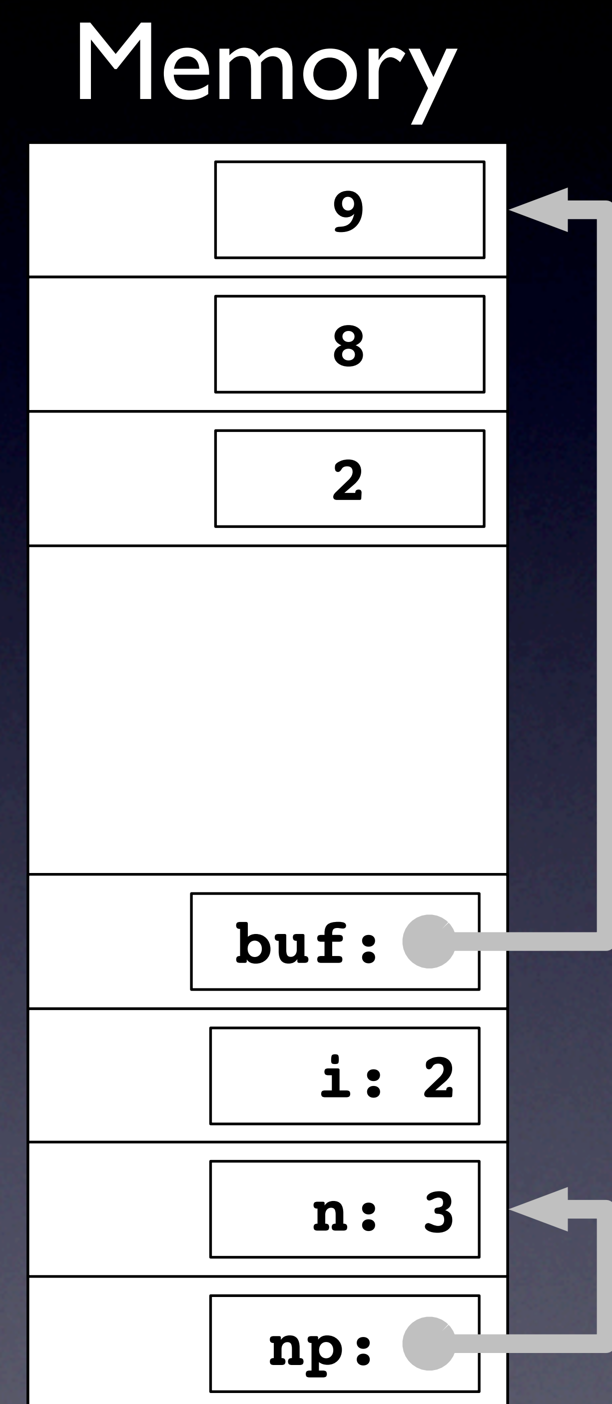
```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```

Memory



# Illegal memory accesses (IMA)

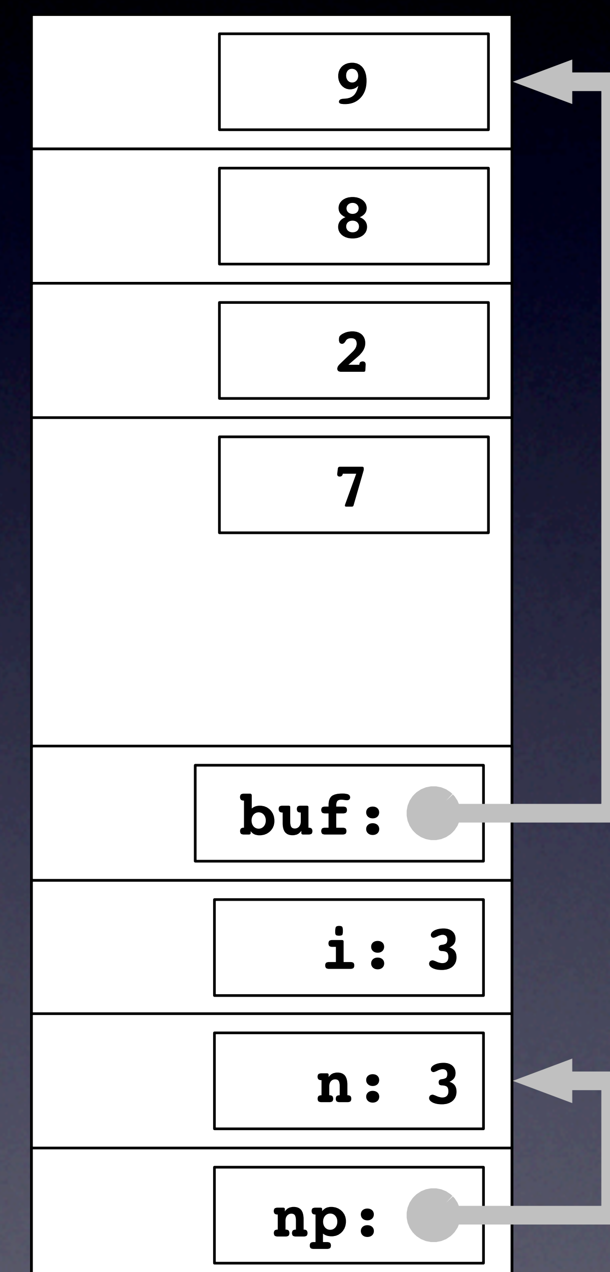
```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```



# Illegal memory accesses (IMA)

```c
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

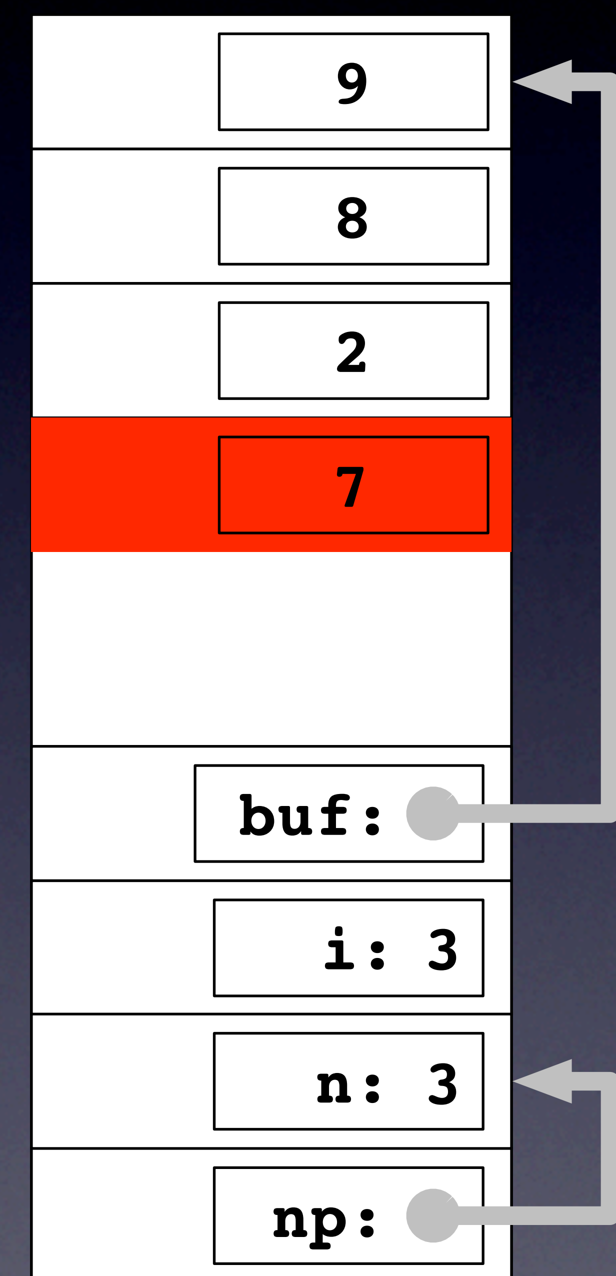
Memory



# Illegal memory accesses (IMA)

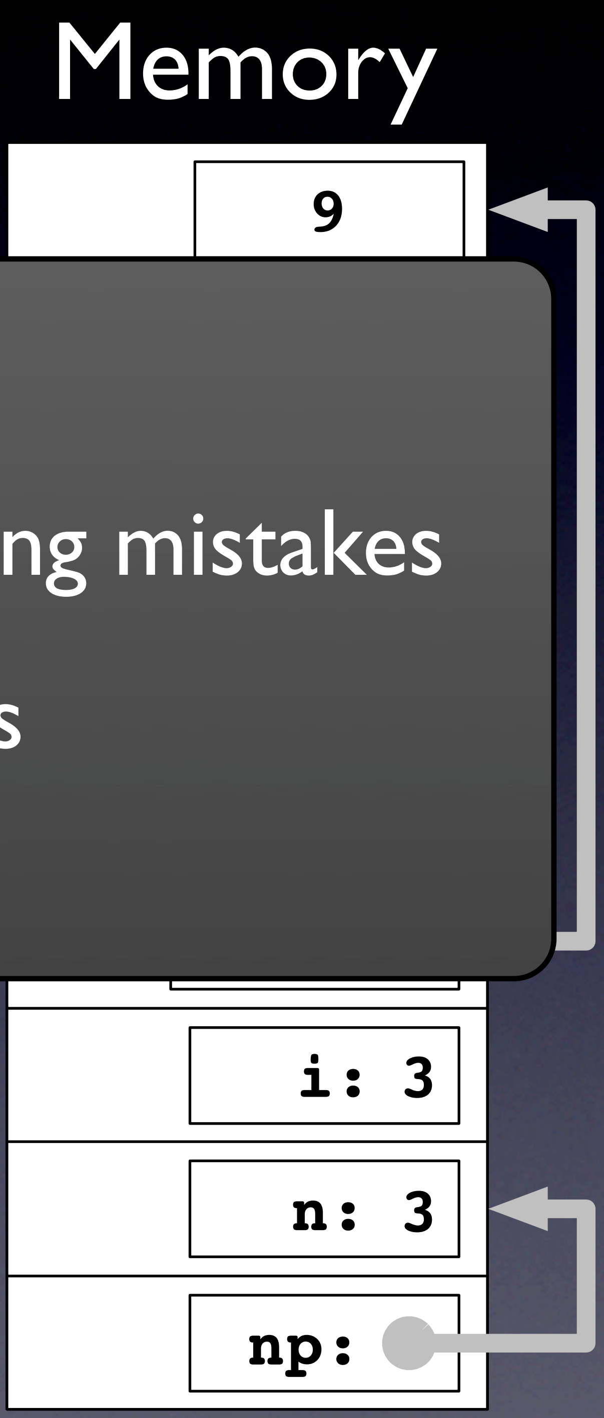
```c
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

Memory



# Illegal memory accesses (IMA)

```
voi
i
n
p
s
b
for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```




## Illegal memory accesses

- Caused by common programming mistakes
- Cause non-deterministic failures
- Cause security vulnerabilities

# Previous work

## Static techniques

- Language based

  e.g., Jim et al. 02, Necula et al. 05

- Analysis based

  e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

## Dynamic techniques

- Analysis based

  e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

- Hardware based

  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

# Previous work

## Static techniques

- Language based

  e.g., Jim et al. 02, Necula et al. 05

- Analysis based

  e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

} Require source code

## Dynamic techniques

- Analysis based

  e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

- Hardware based

  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

# Previous work

## Static techniques

- Language based

  e.g., Jim et al. 02, Necula et al. 05

- Analysis based

  e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Require source code

## Dynamic techniques

- Analysis based

  e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

Unacceptable overhead

- Hardware based

  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

# Previous work

## Static techniques

- Language based

  e.g., Jim et al. 02, Necula et al. 05

- Analysis based

  e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Require source code

## Dynamic techniques

- Analysis based

  e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

Unacceptable overhead

- Hardware based

  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

Extensive modification

# Previous work

Static techniques

We define our approach to overcome these limitations

- Operate at the binary level
- Use hardware to reduce overhead
- Minimal, practical modifications

- Hardware based

  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

Extensive modification

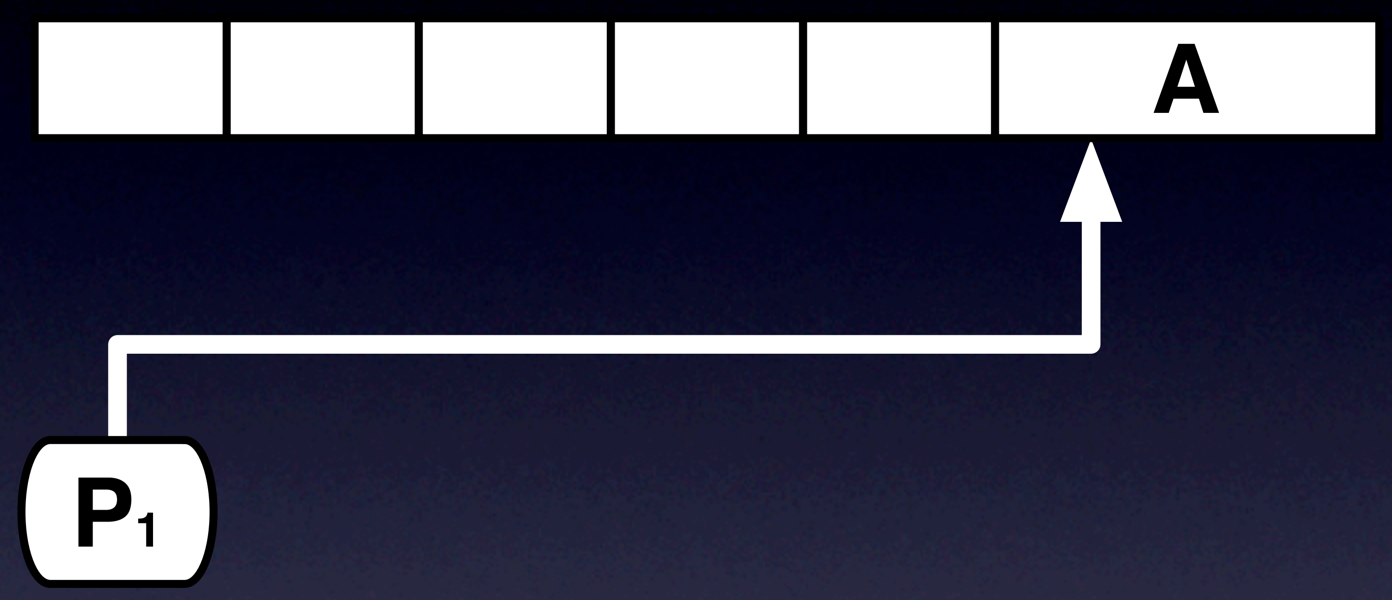
# Approach overview

# Approach overview

1. **Assign taint marks**

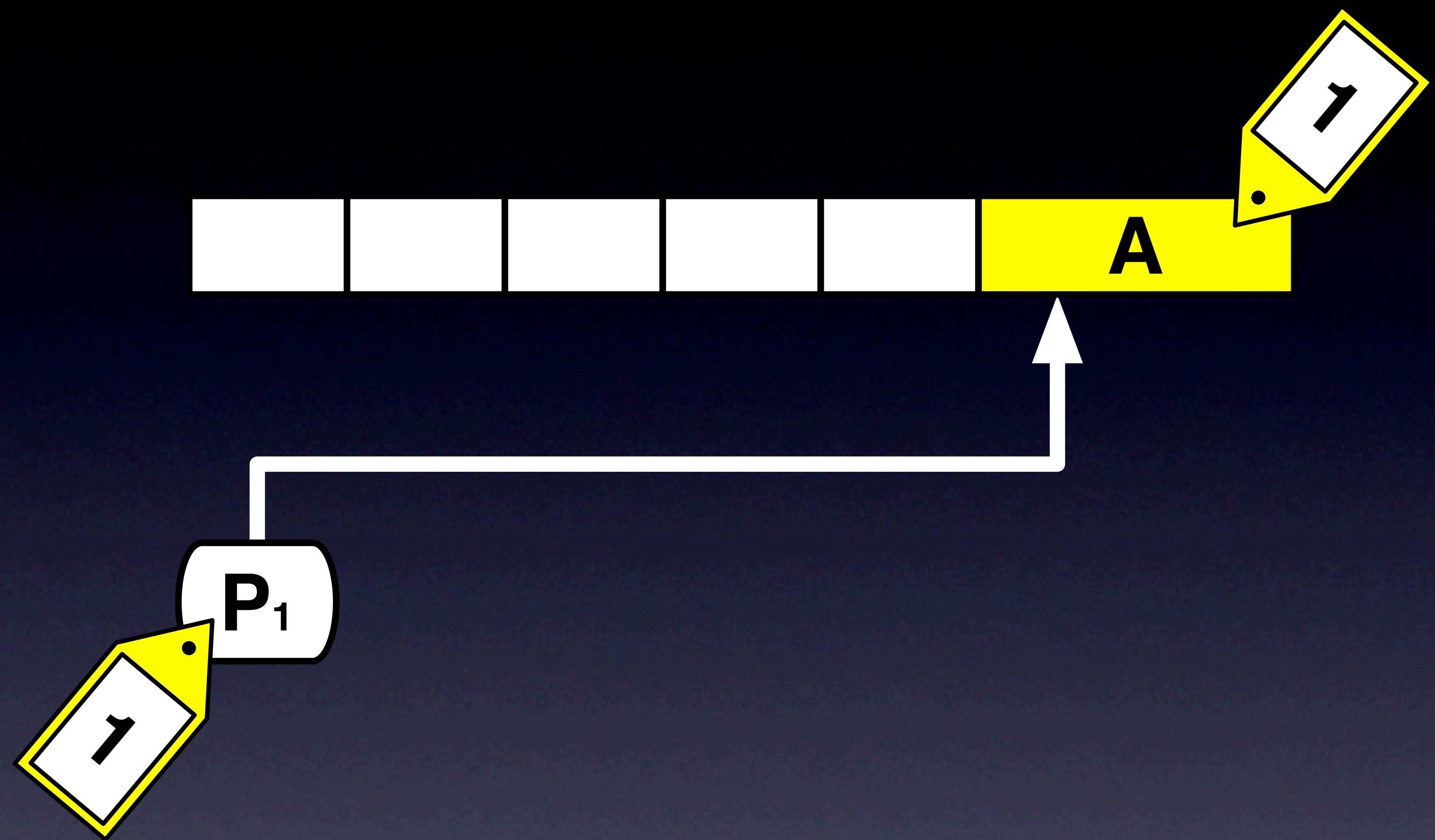
# Approach overview

1 **Assign taint marks**



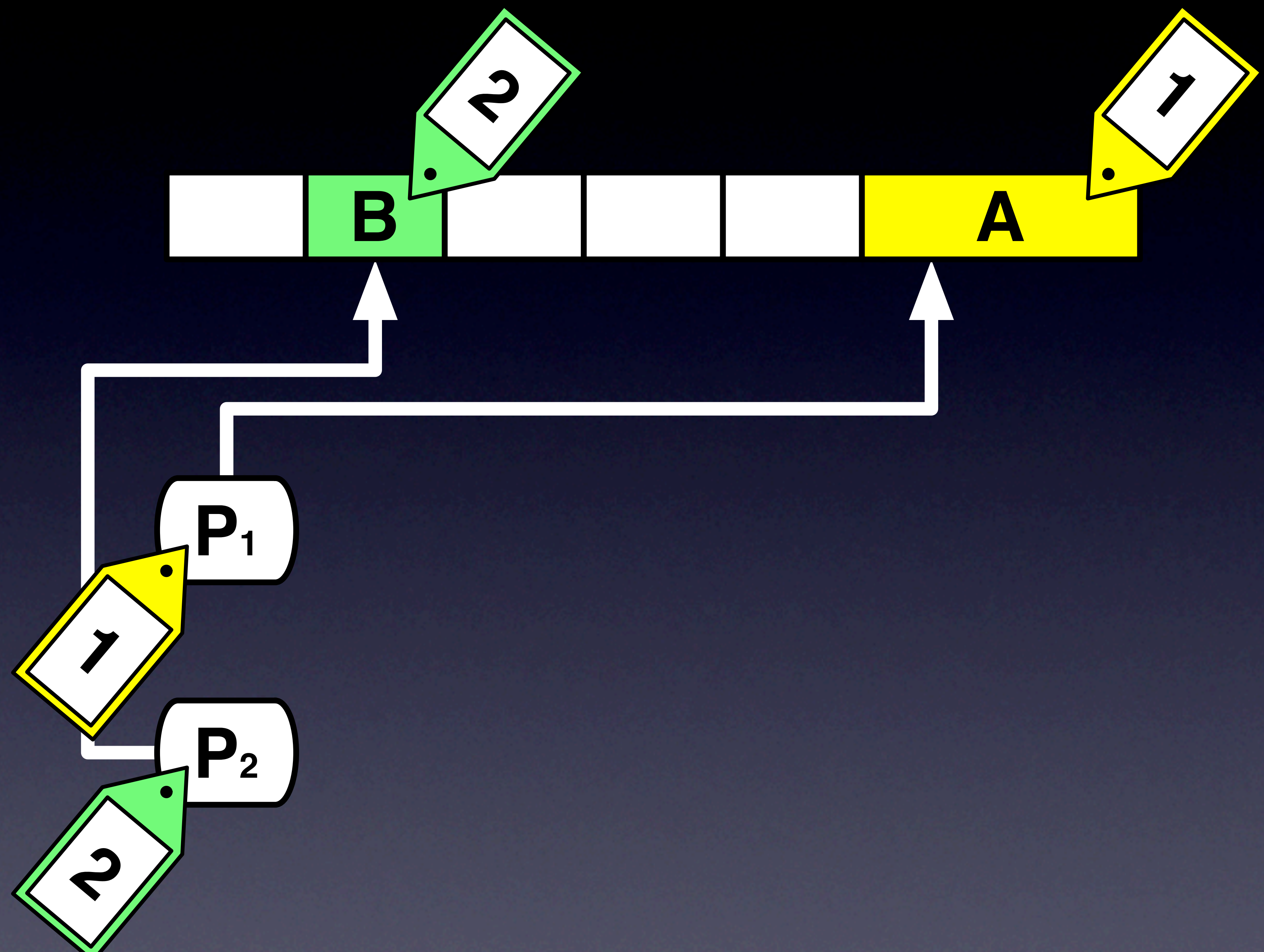
# Approach overview

1 **Assign taint marks**



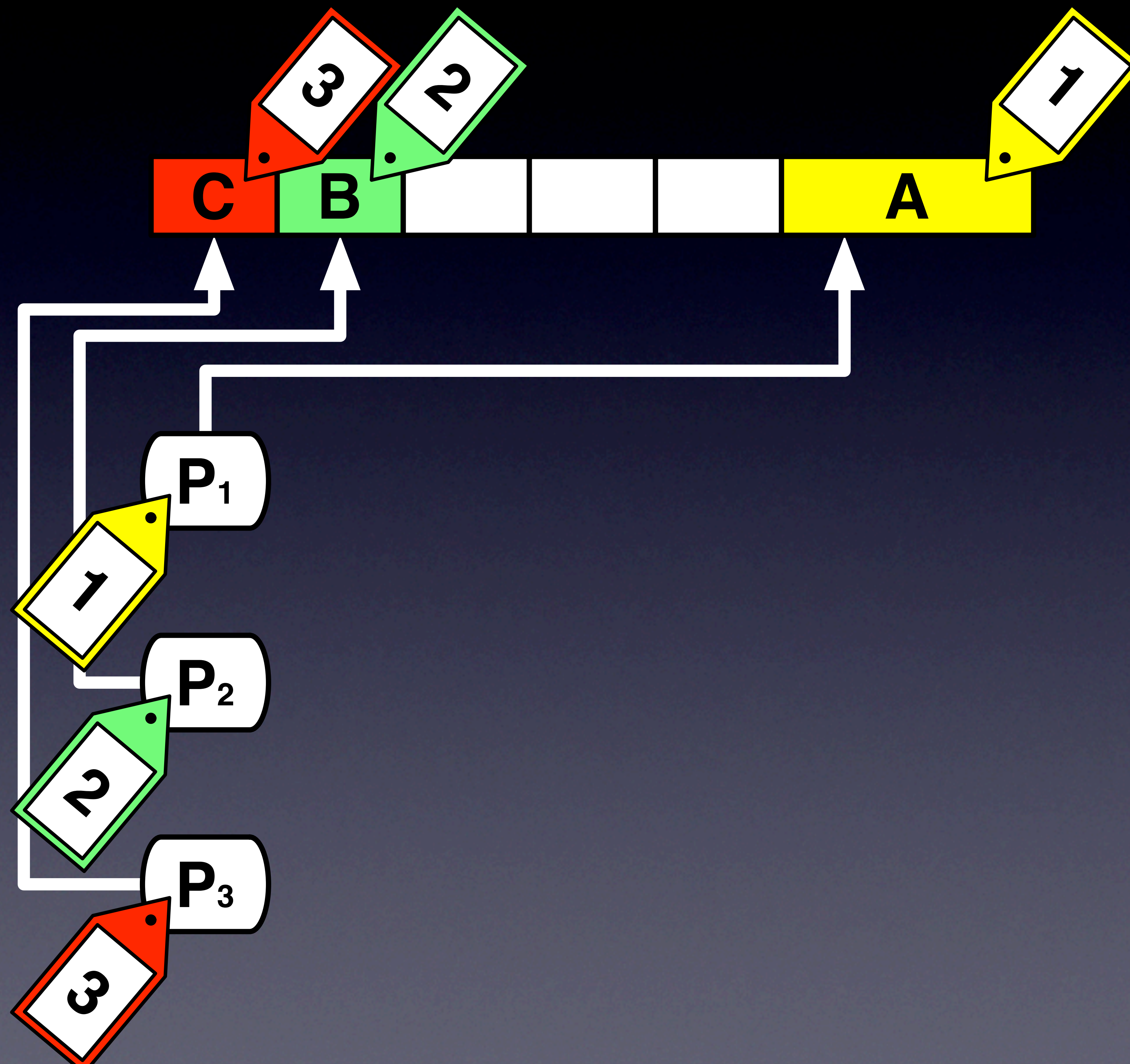
# Approach overview

1 **Assign taint marks**



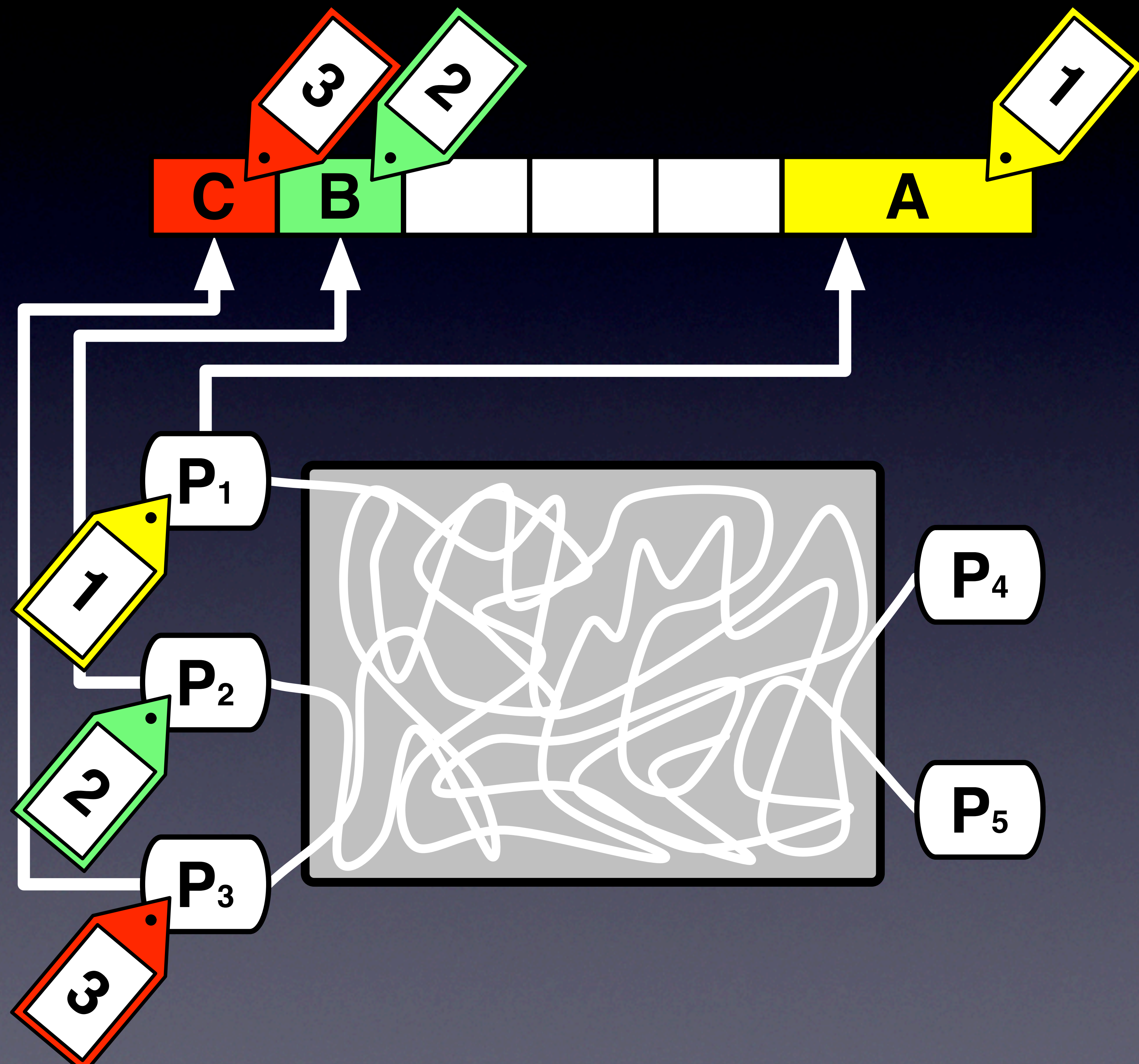
# Approach overview

1 **Assign taint marks**



# Approach overview

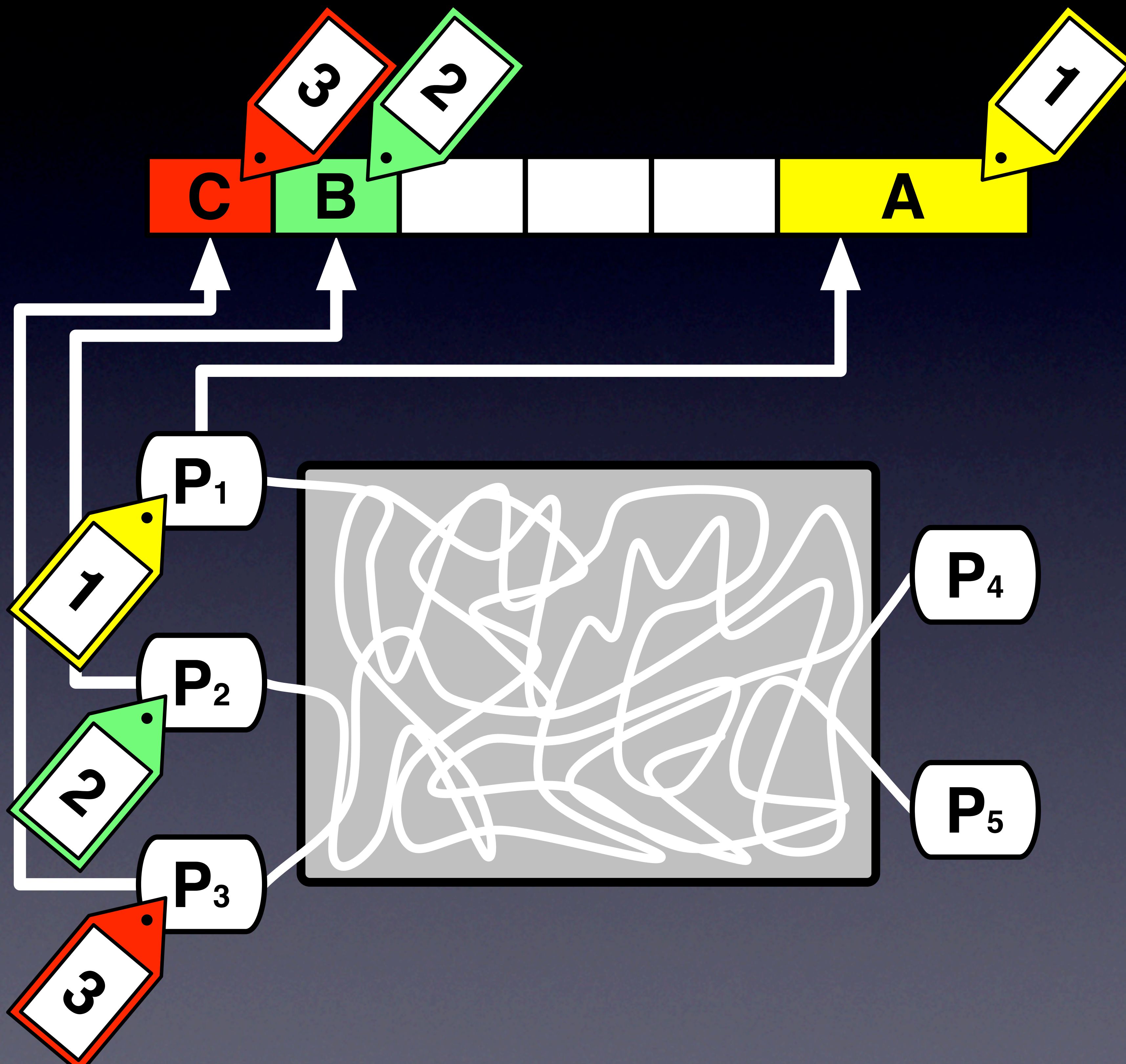
**1 Assign taint marks**



# Approach overview

1 **Assign taint marks**

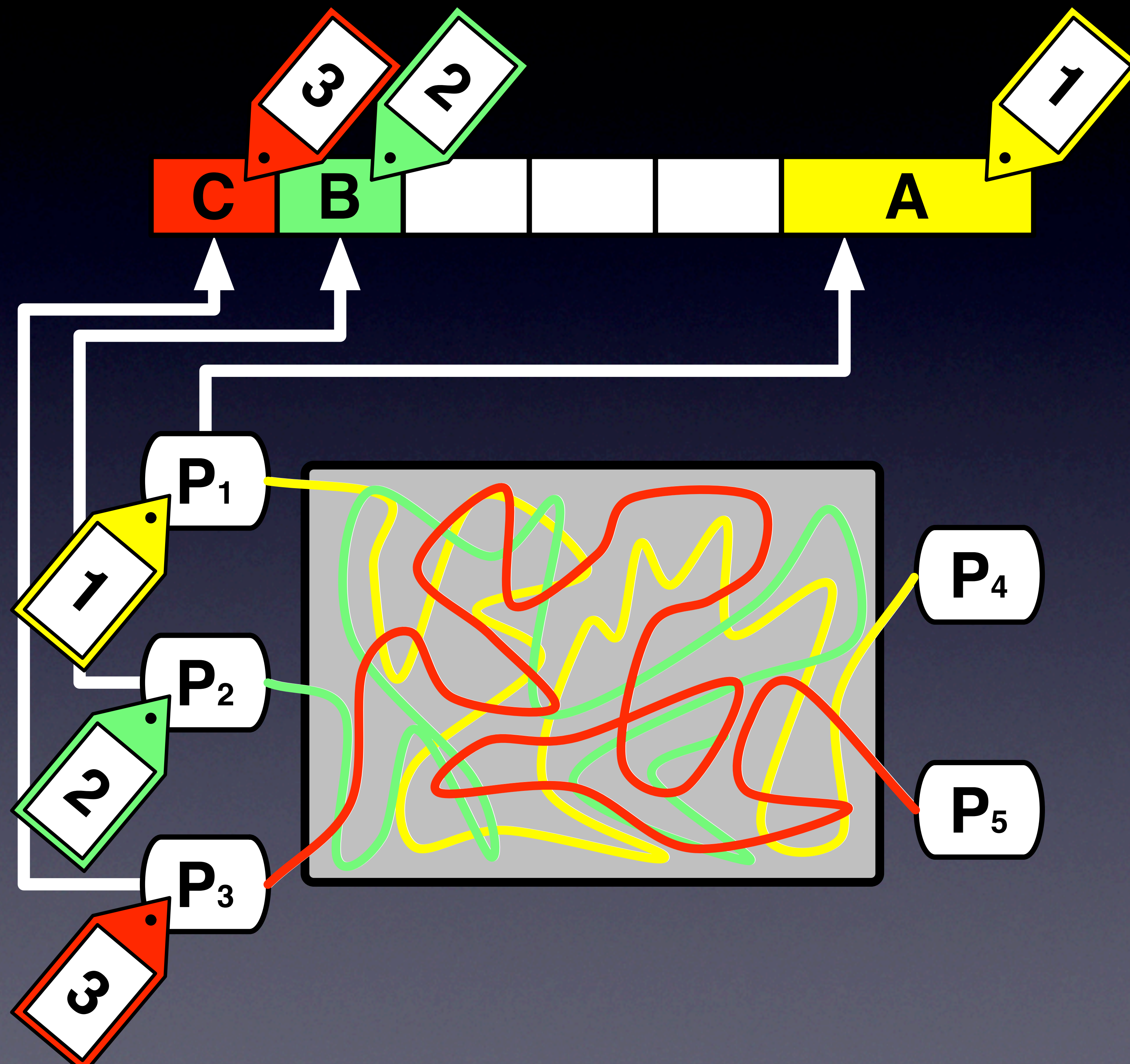
2 **Propagate taint marks**



# Approach overview

1 **Assign taint marks**

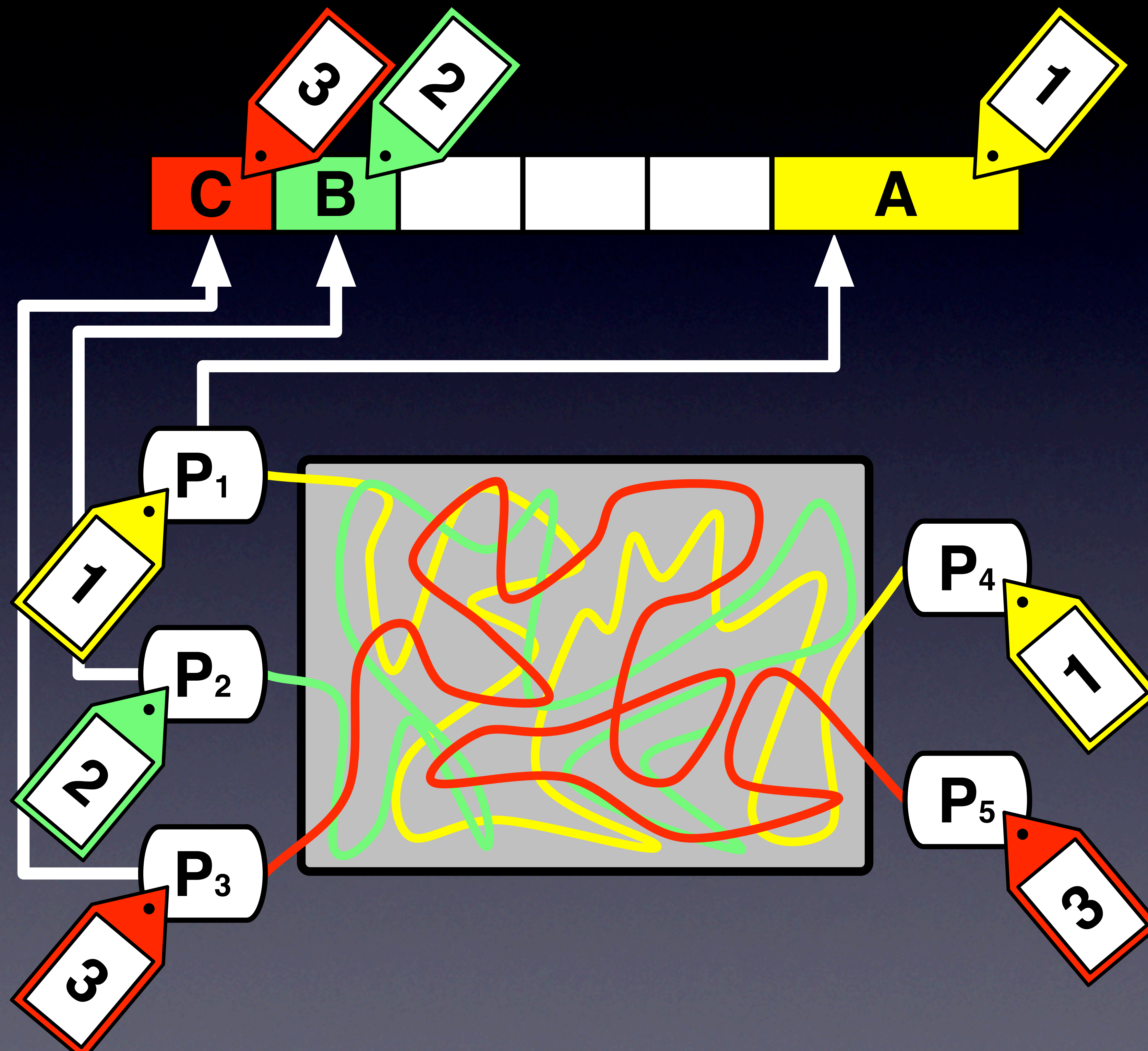
2 **Propagate taint marks**



# Approach overview

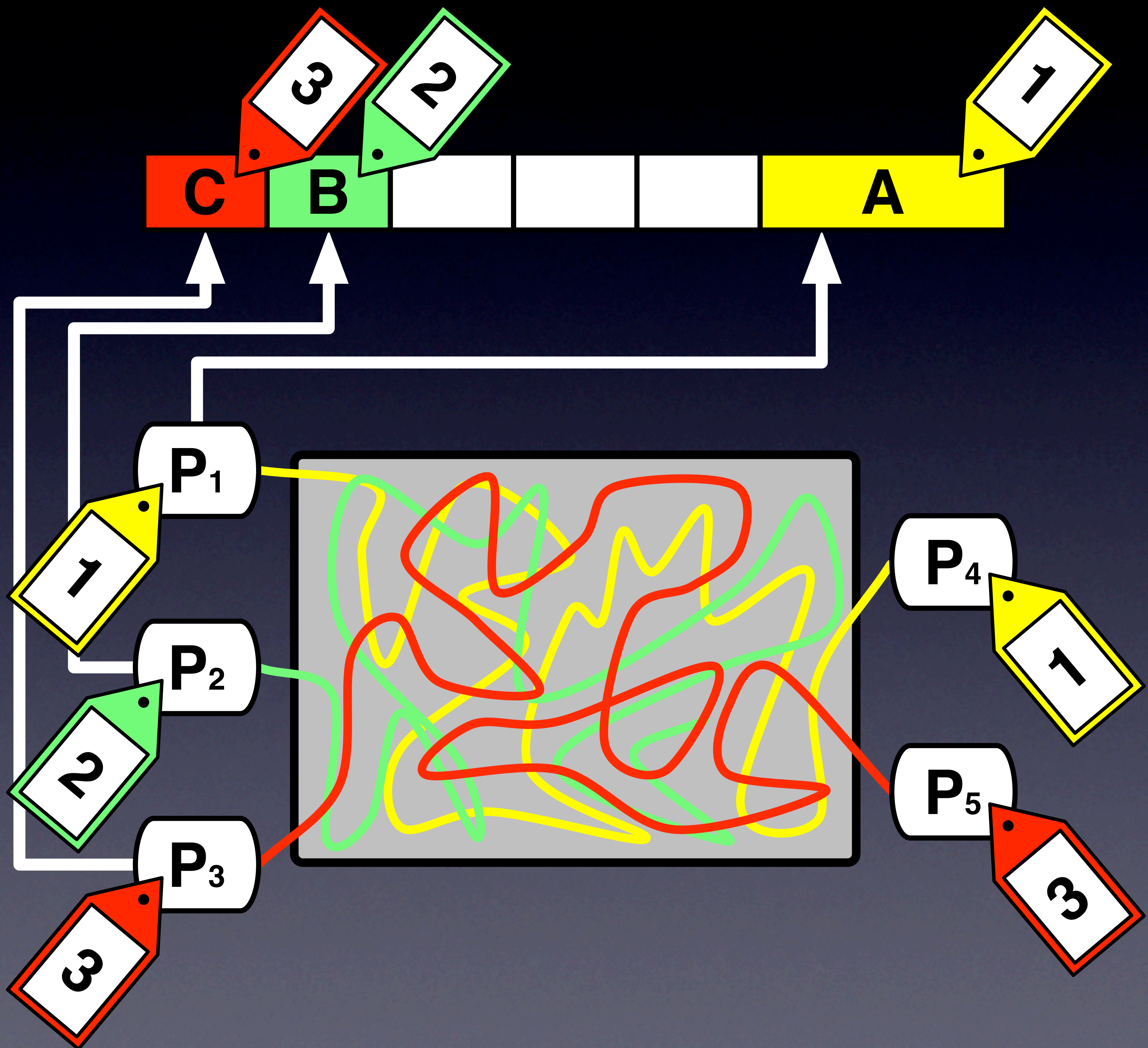
1 **Assign taint marks**

2 **Propagate taint marks**



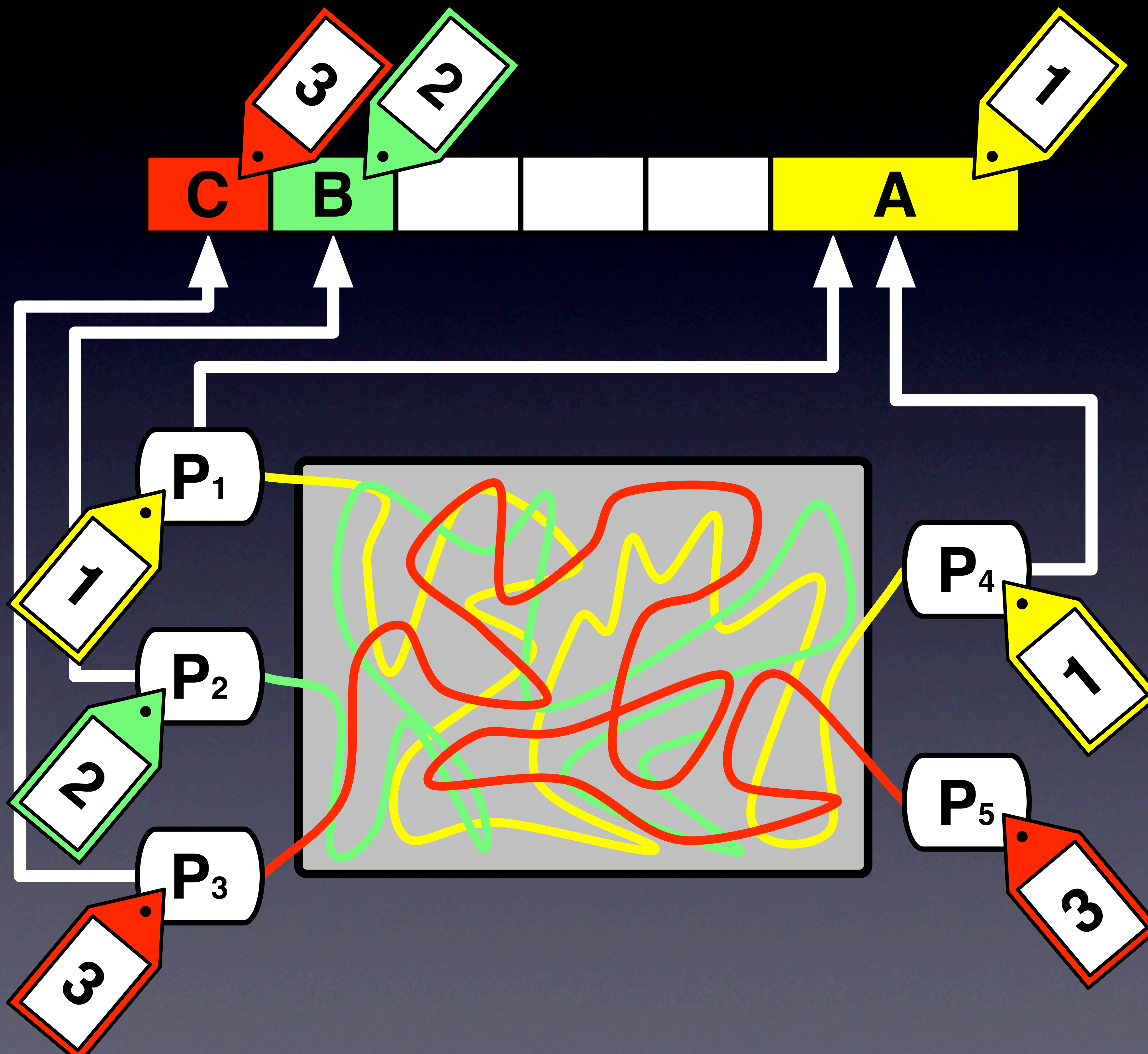
# Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks



# Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks

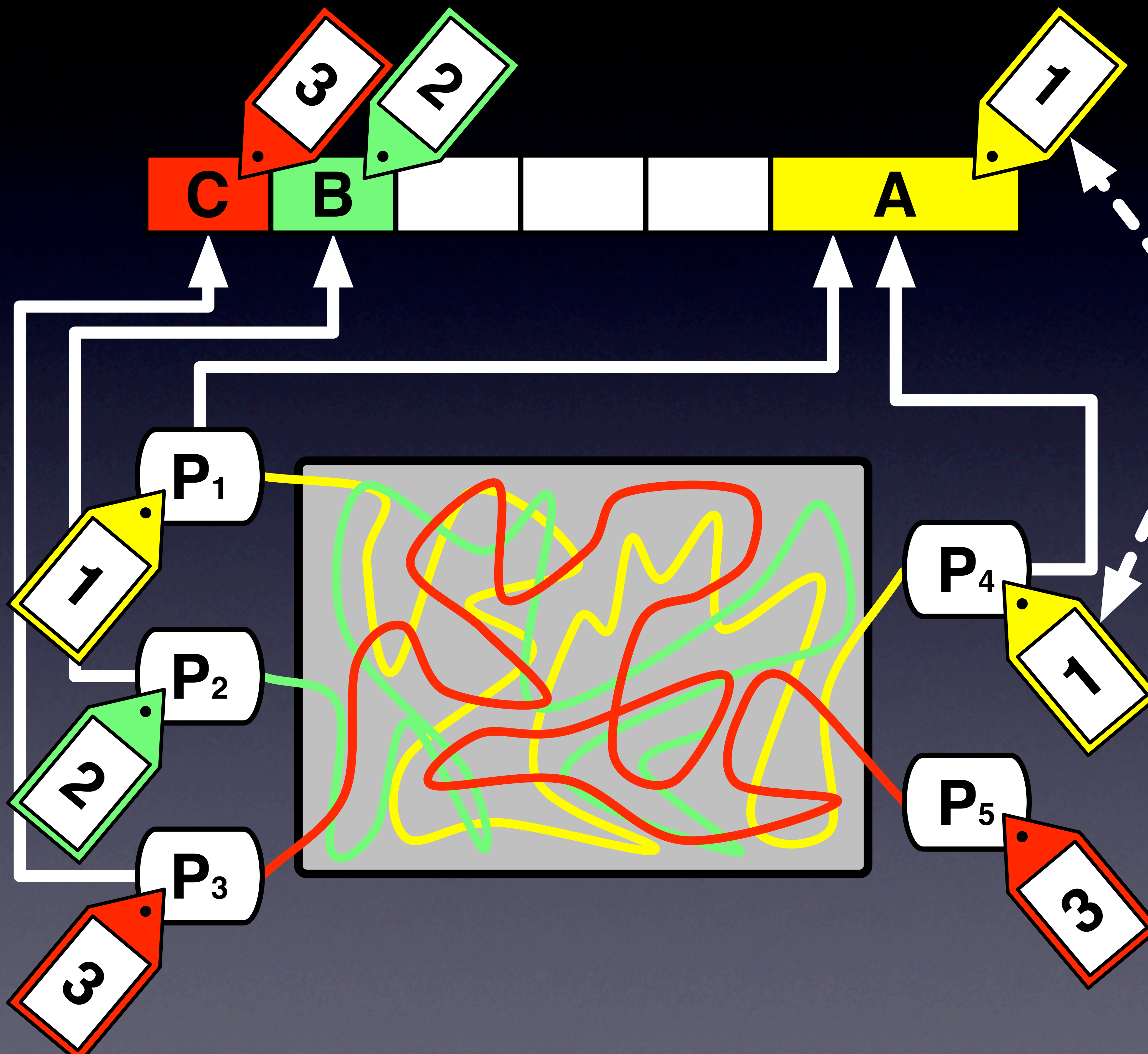


# Approach overview

1 **Assign taint marks**

2 **Propagate taint marks**

3 **Check taint marks**

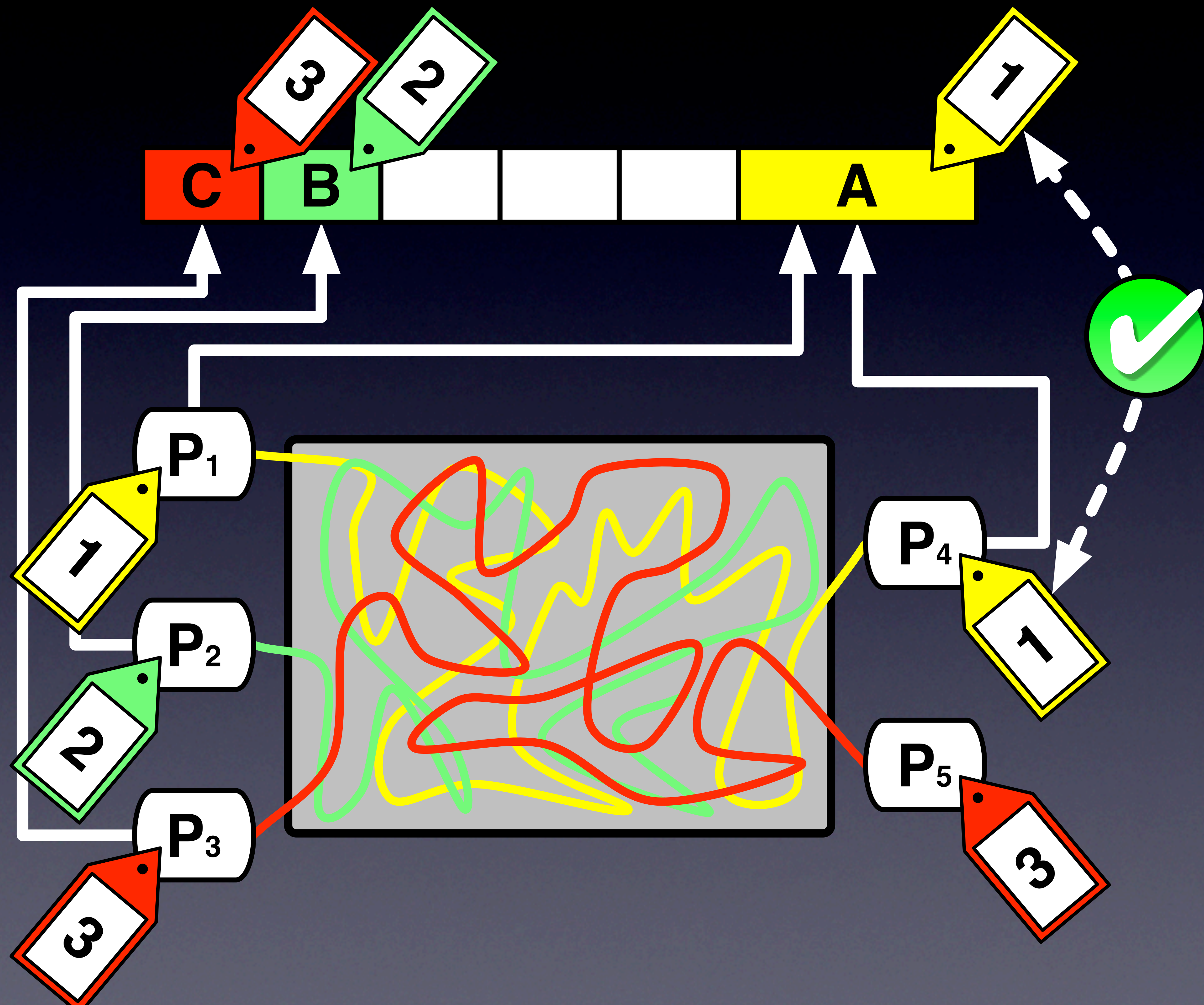


# Approach overview

1 Assign taint marks

2 Propagate taint marks

3 Check taint marks

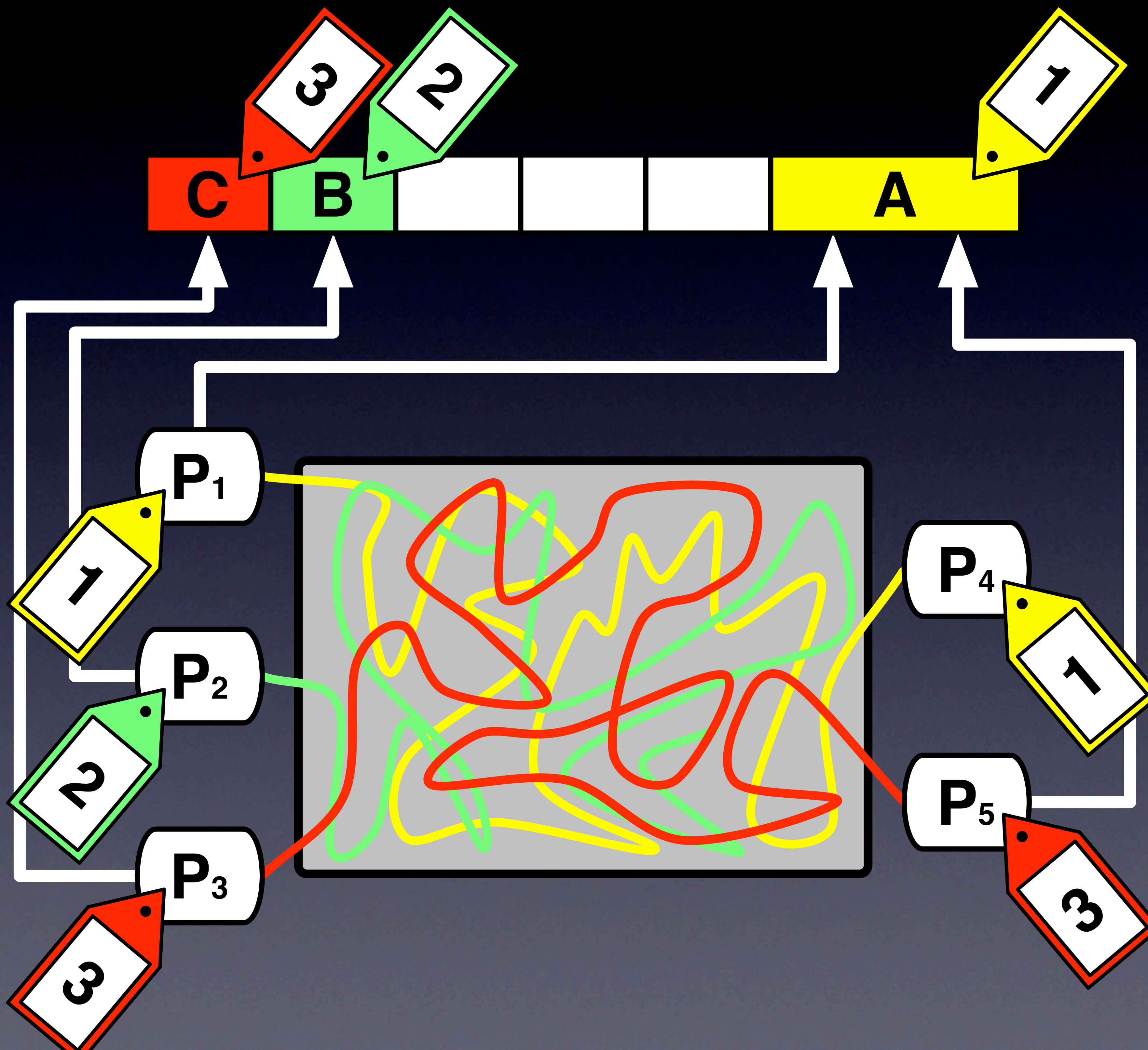


# Approach overview

1 **Assign taint marks**

2 **Propagate taint marks**

3 **Check taint marks**

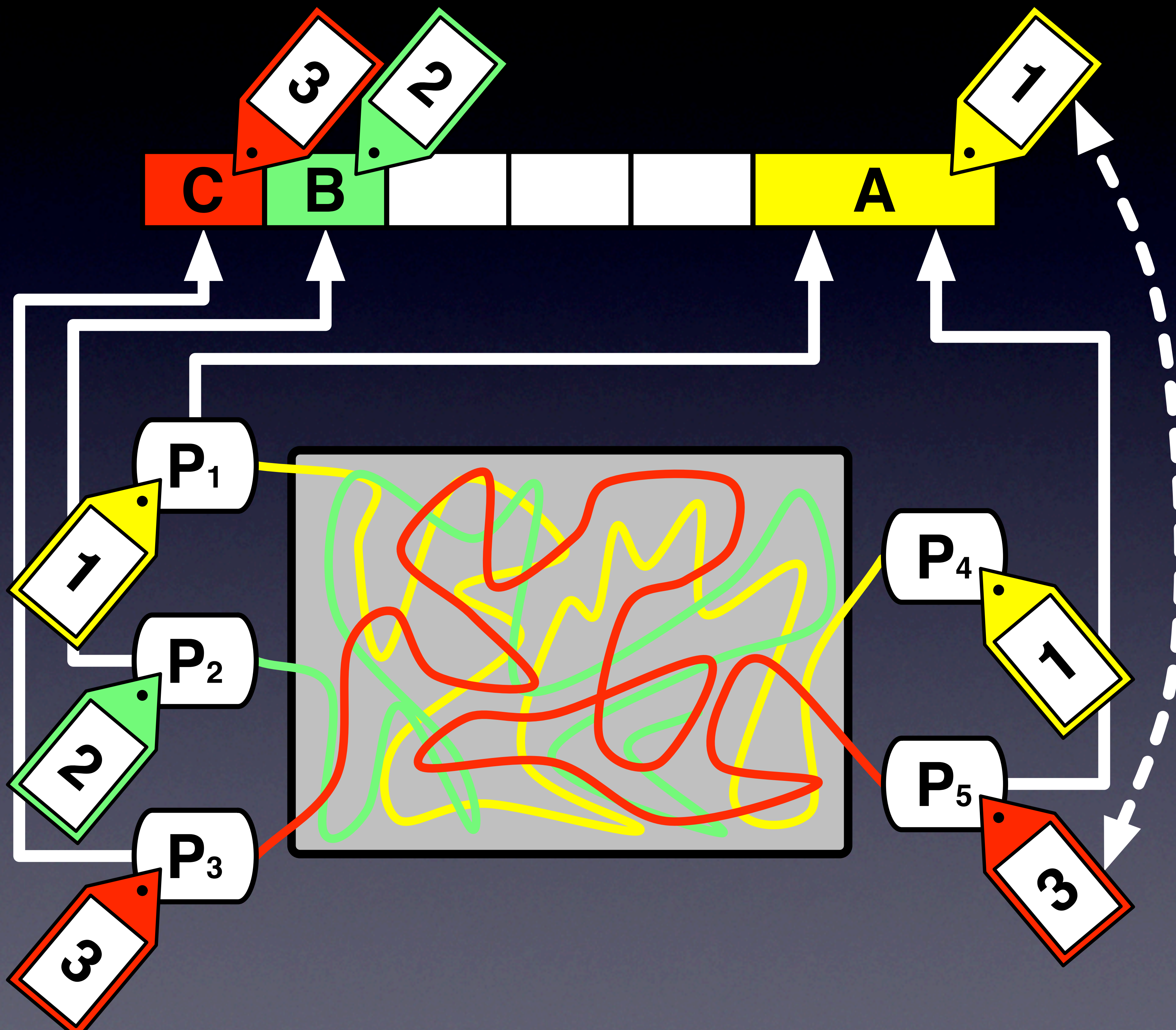


# Approach overview

1 Assign taint marks

2 Propagate taint marks

3 Check taint marks

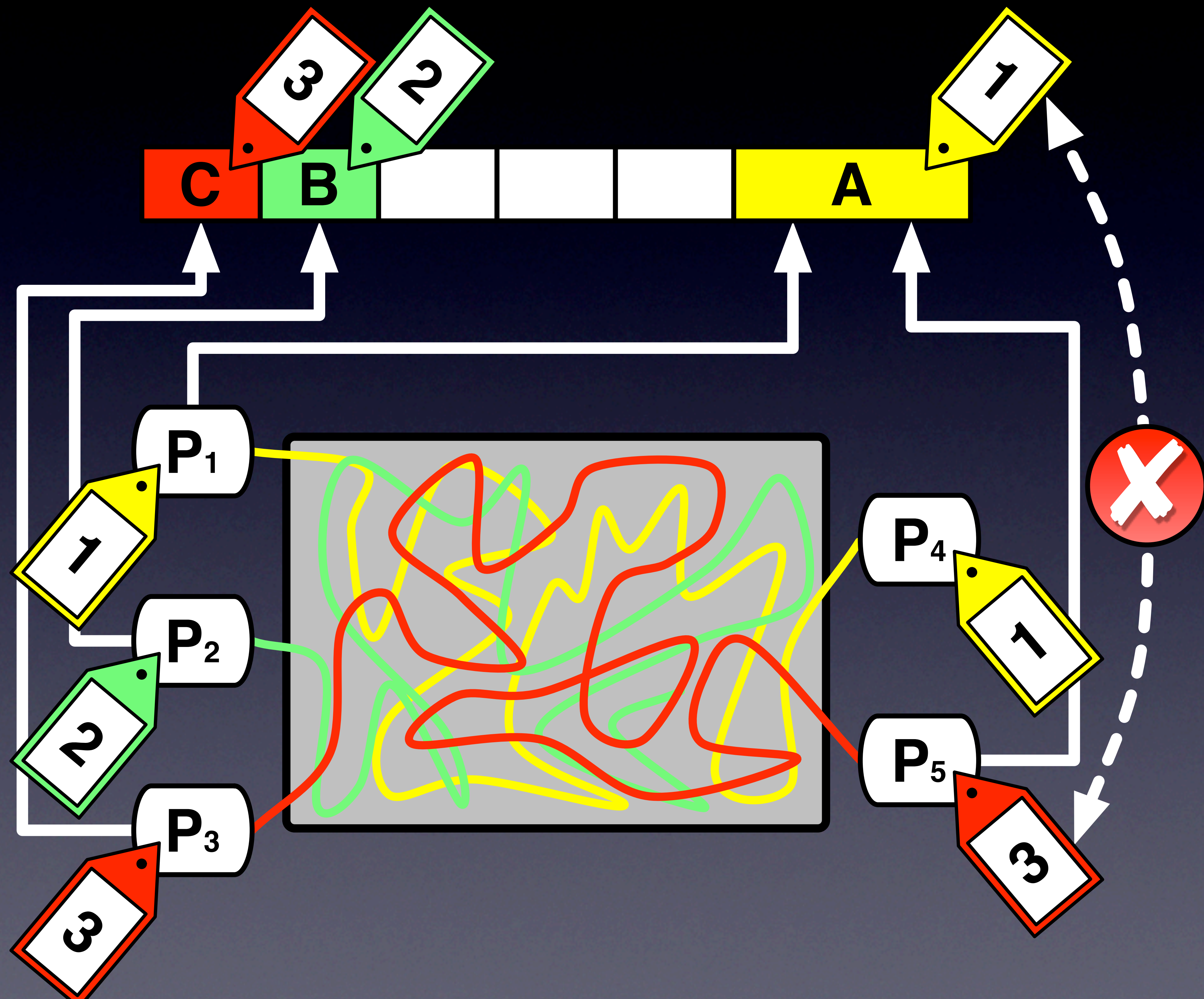


# Approach overview

1 Assign taint marks

2 Propagate taint marks

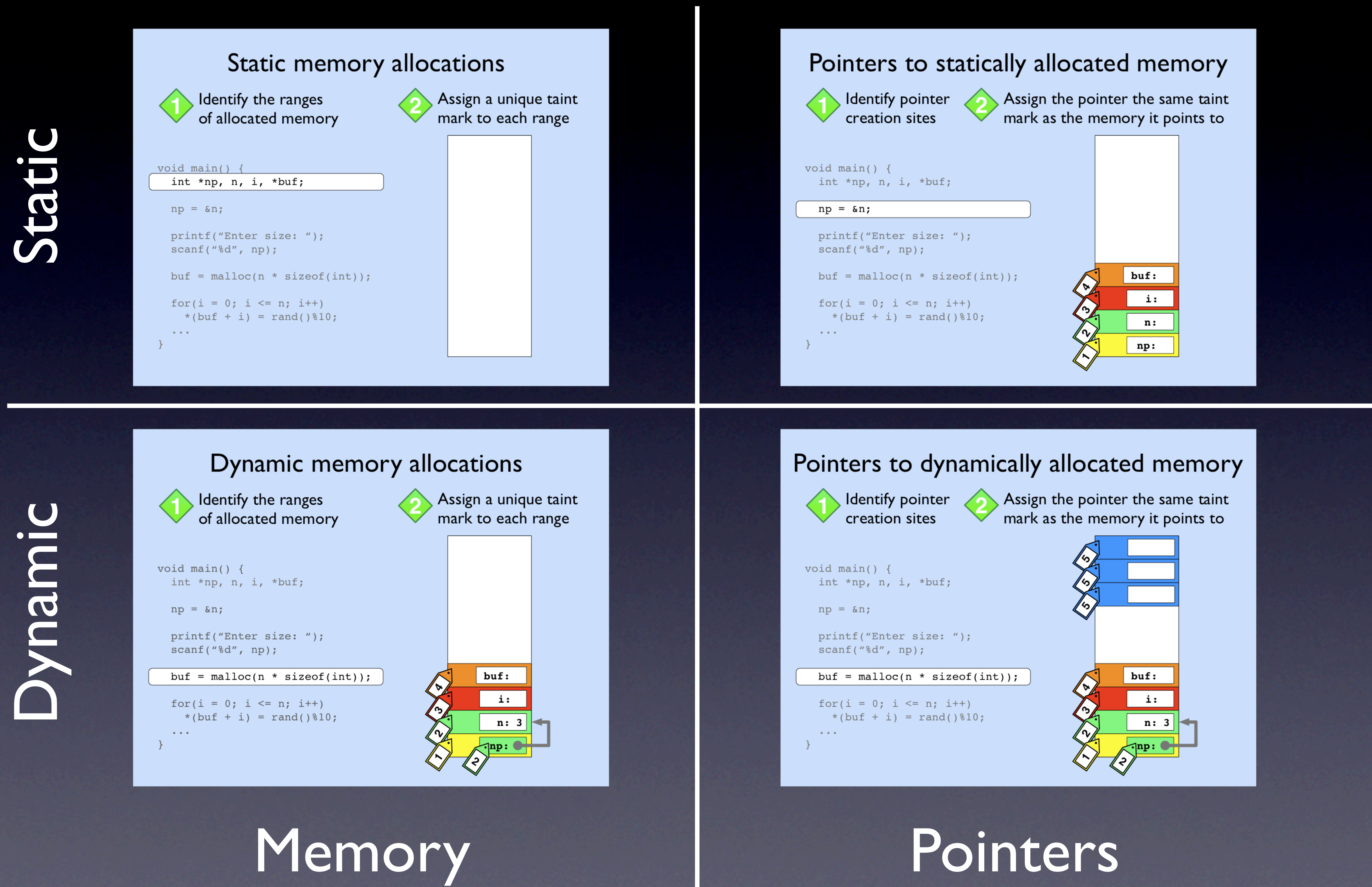
3 Check taint marks



# Outline

- Our approach
    1. Assigning taint marks
    2. Propagating taint marks
    3. Checking taint marks
- Empirical evaluation
- Conclusions

# 1 Assigning taint marks



# Static memory allocations

1 Identify the ranges of allocated memory

2 Assign a unique taint mark to each range

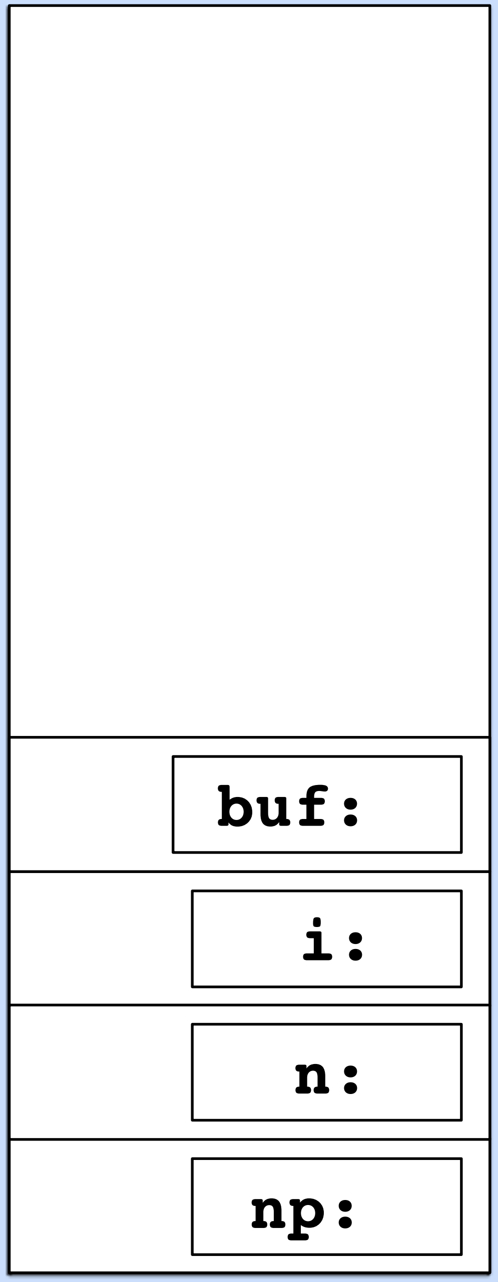
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

# Static memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



# Static memory allocations

1 Identify the ranges of allocated memory

2 Assign a unique taint mark to each range

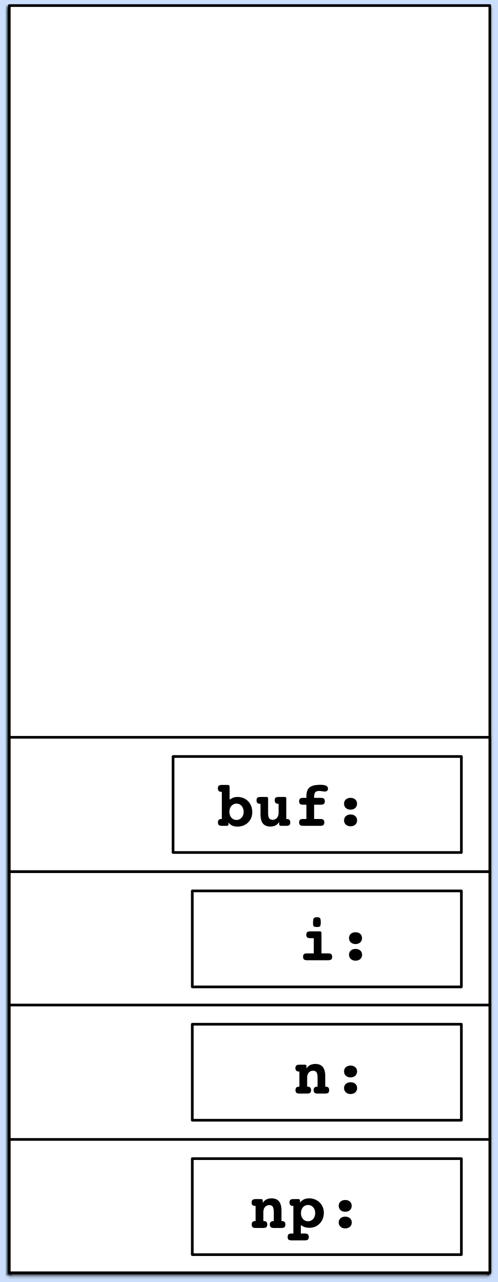
```
[&np, &np + sizeof(int *))
```

```
void main() {
   int *np, n, i, *buf;

   np = &n;

   printf("Enter size: ");
   scanf("%d", np);

   buf = malloc(n * sizeof(int));

   for(i = 0; i <= n; i++)
     *(buf + i) = rand()%10;
   ...
}
```

# Static memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

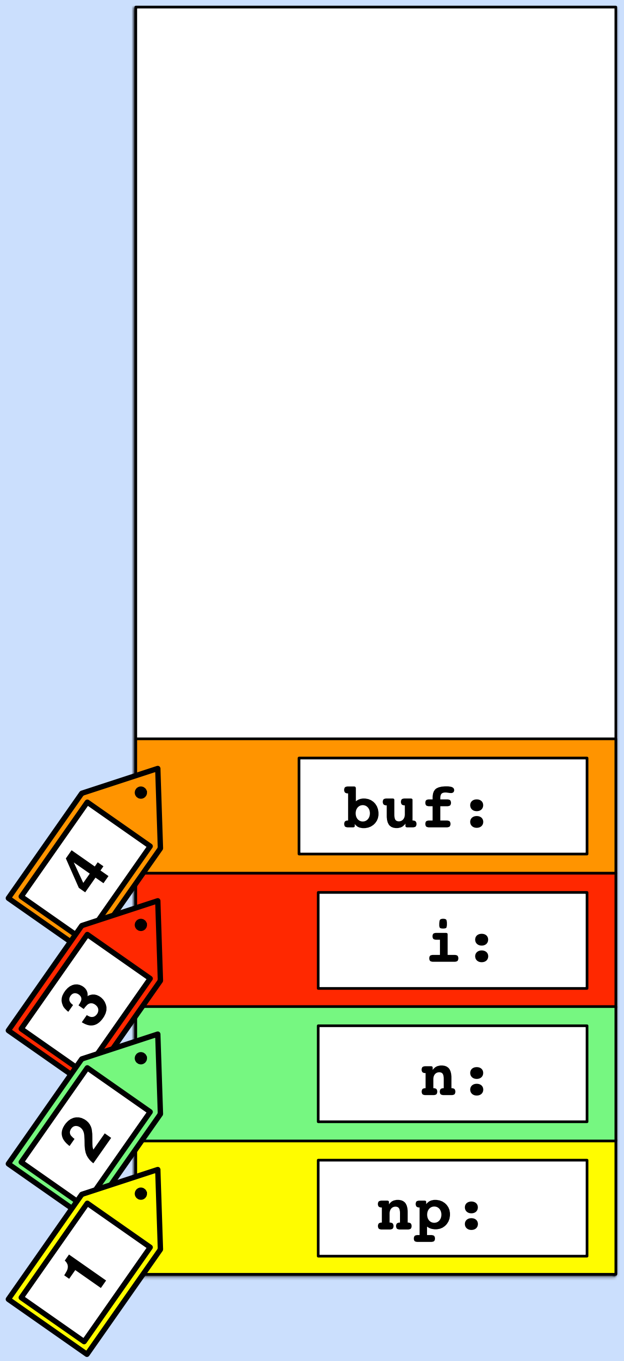


# Static memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



# 1 Assigning taint marks

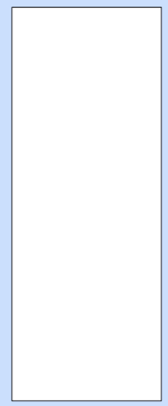
Static

Dynamic

## Static memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

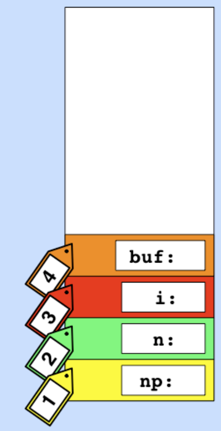
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



## Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

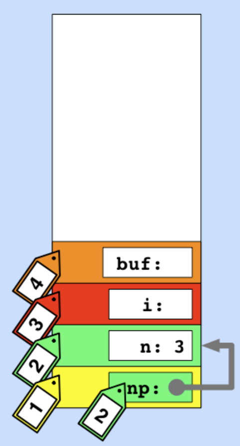
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

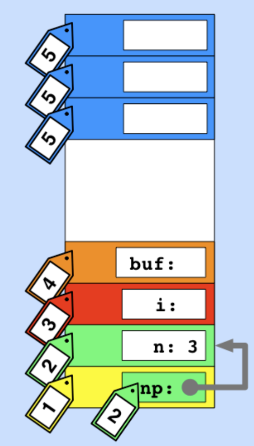
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



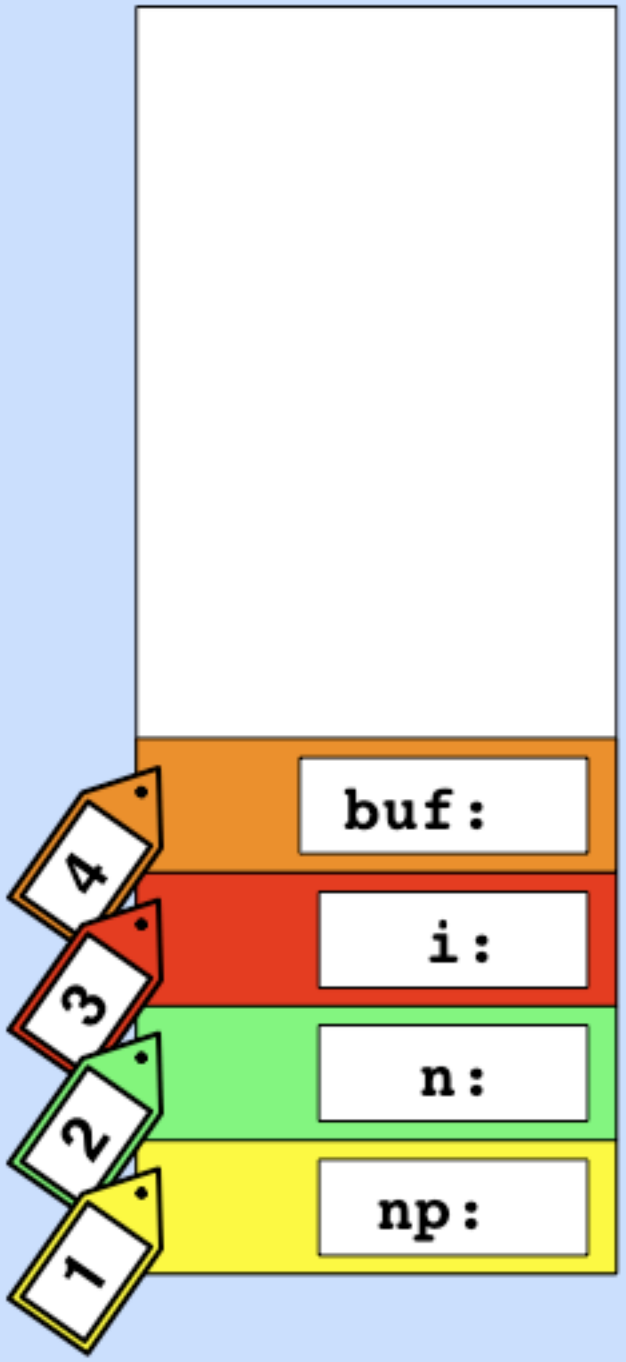

Memory

Pointers

# Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

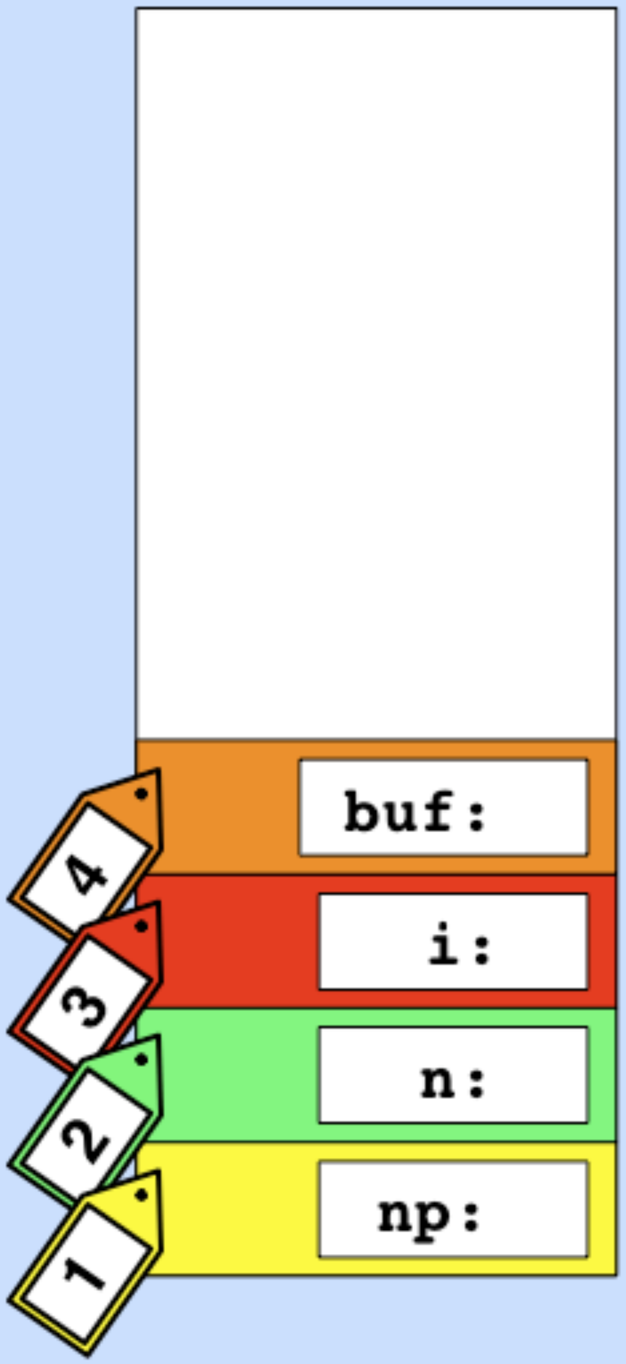


# Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int
    np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

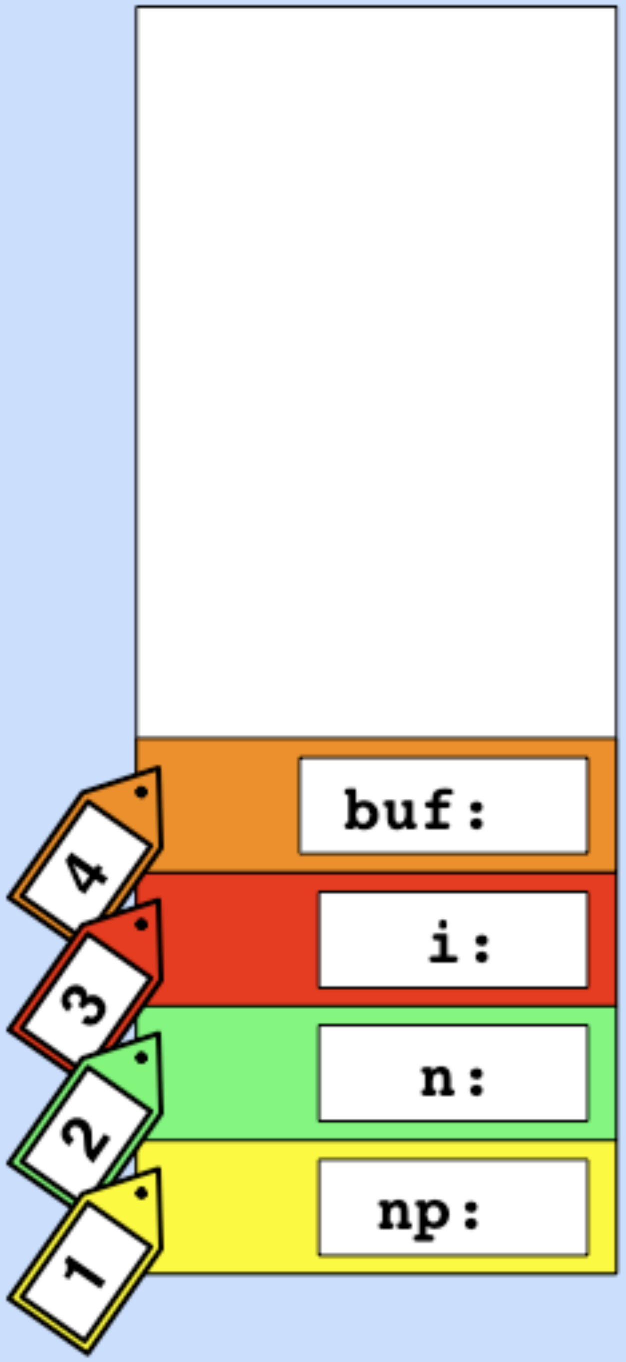
address-of operator (& )



# Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

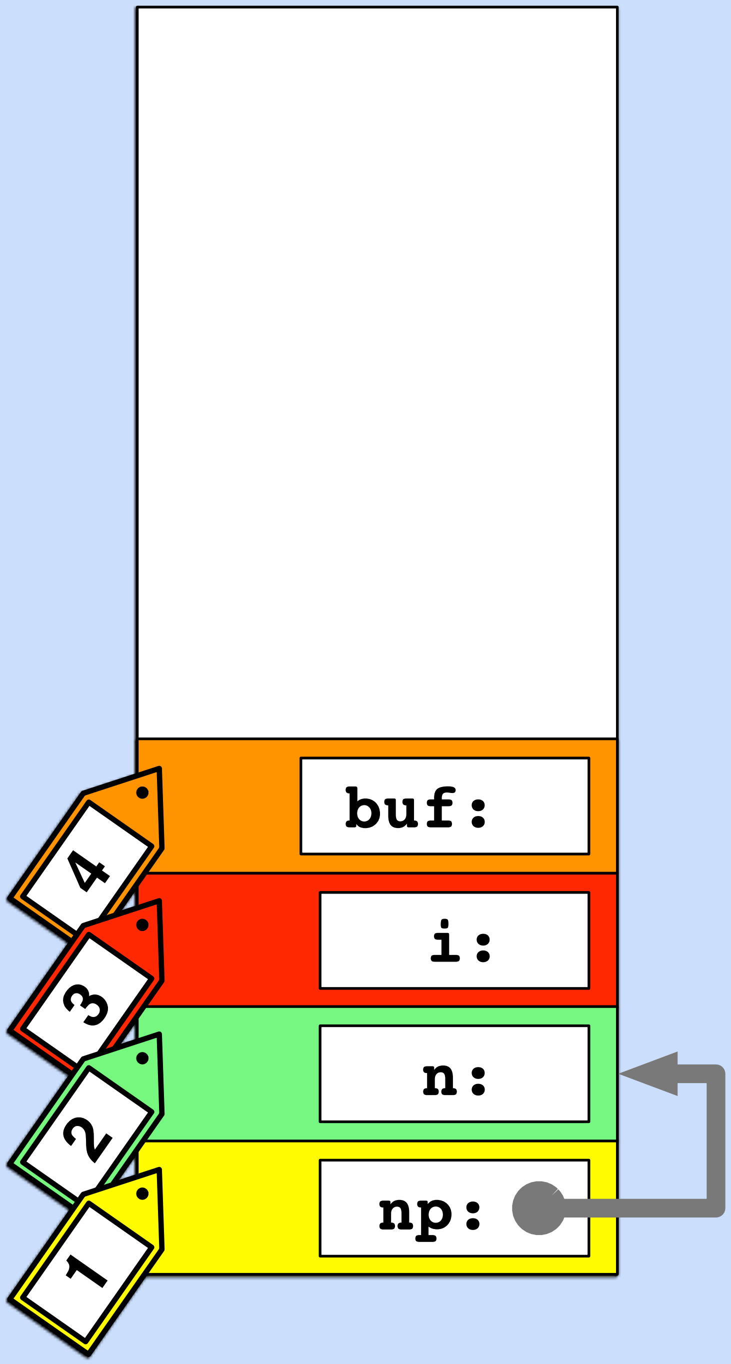


# Pointers to statically allocated memory

1 Identify pointer creation sites

2 Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

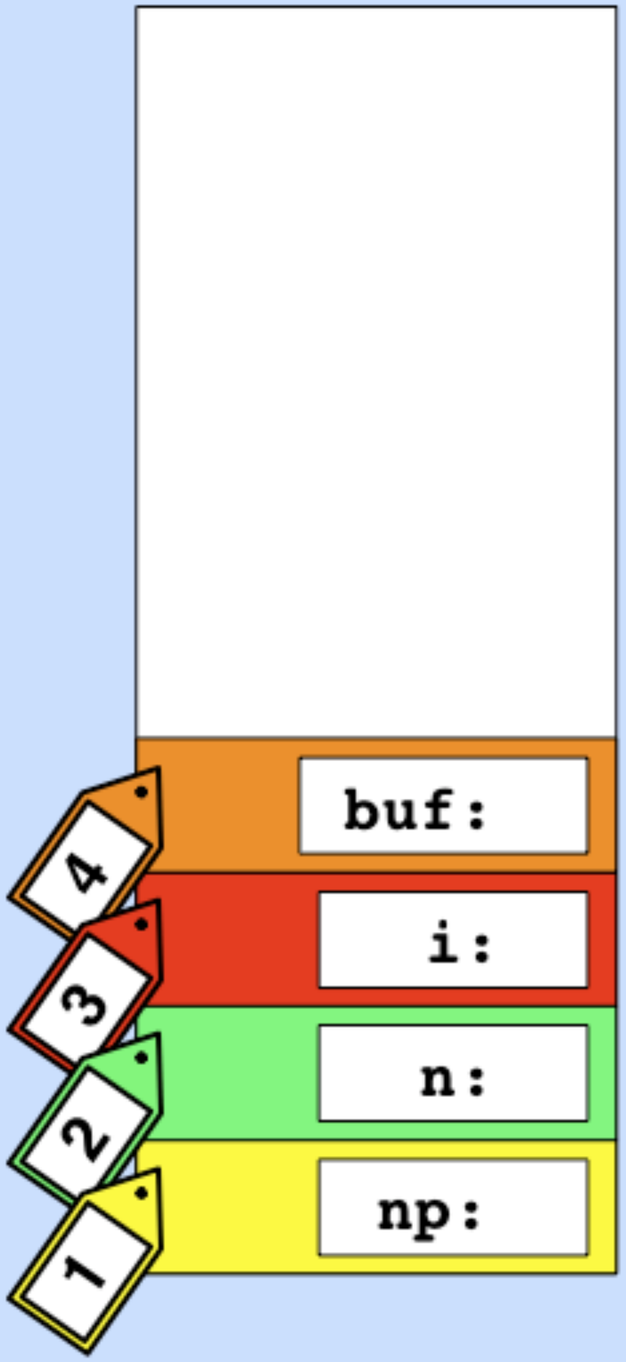


# Pointers to statically allocated memory

1 Identify pointer creation sites

2 Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



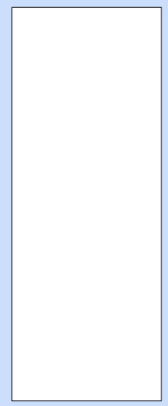
# 1 Assigning taint marks

Static

## Static memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

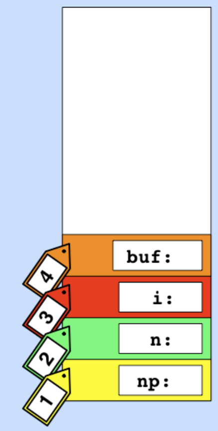
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



## Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

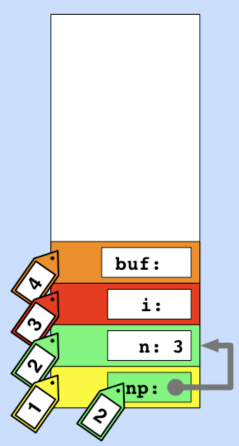



Dynamic

## Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

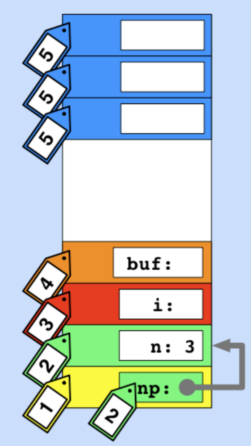
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




Memory

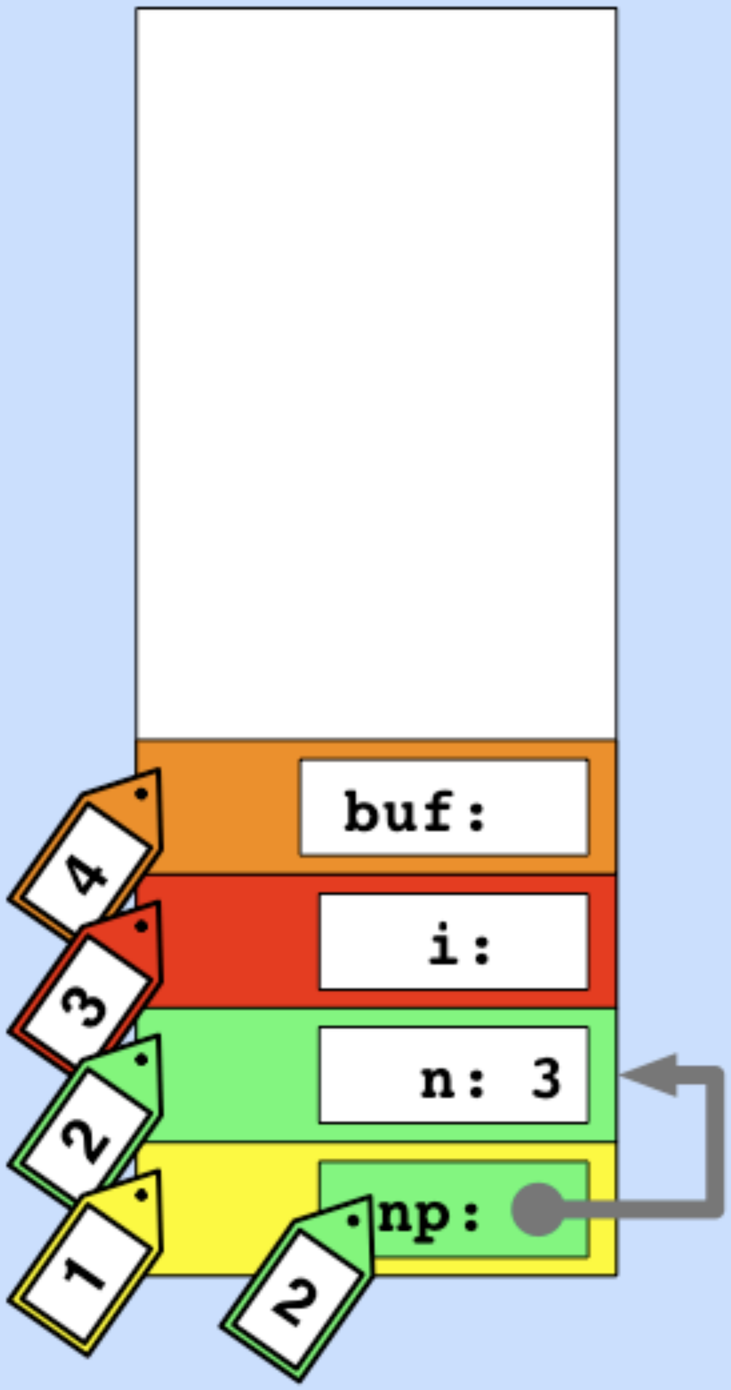
Pointers

# Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

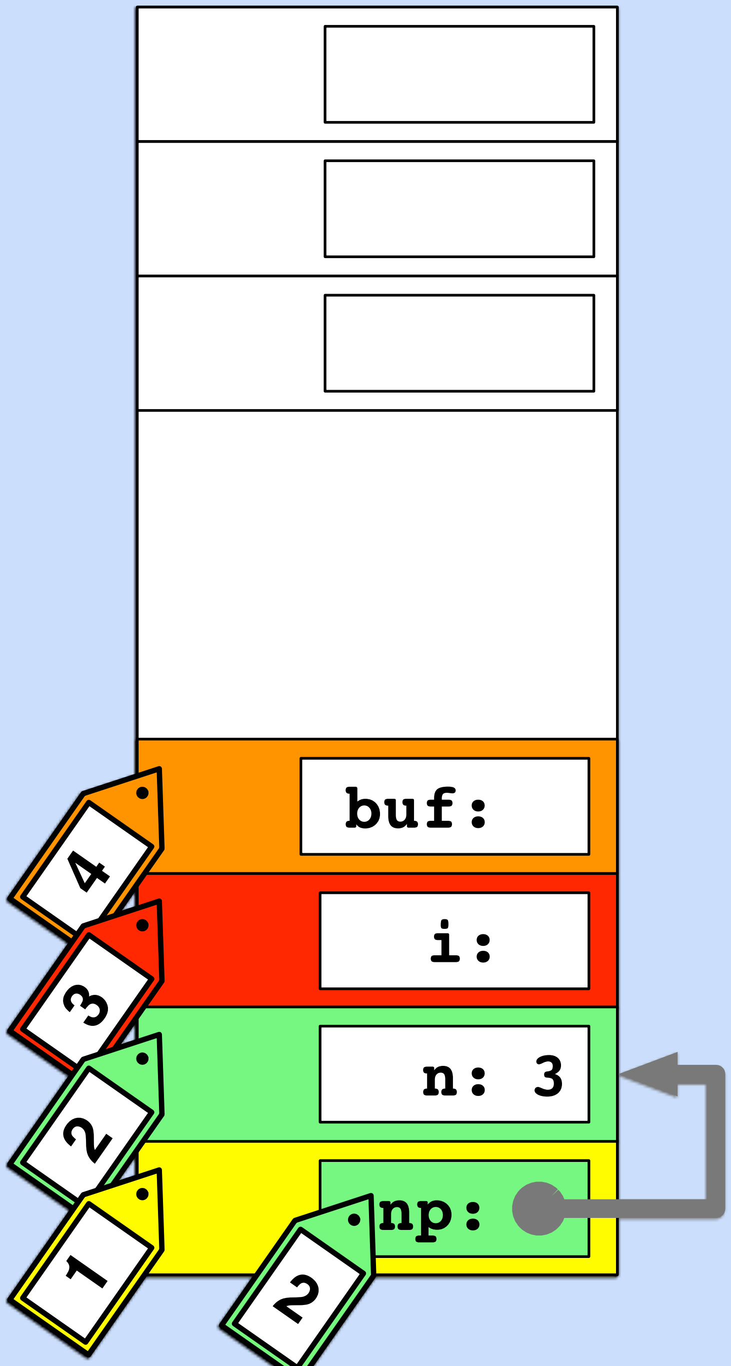


# Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



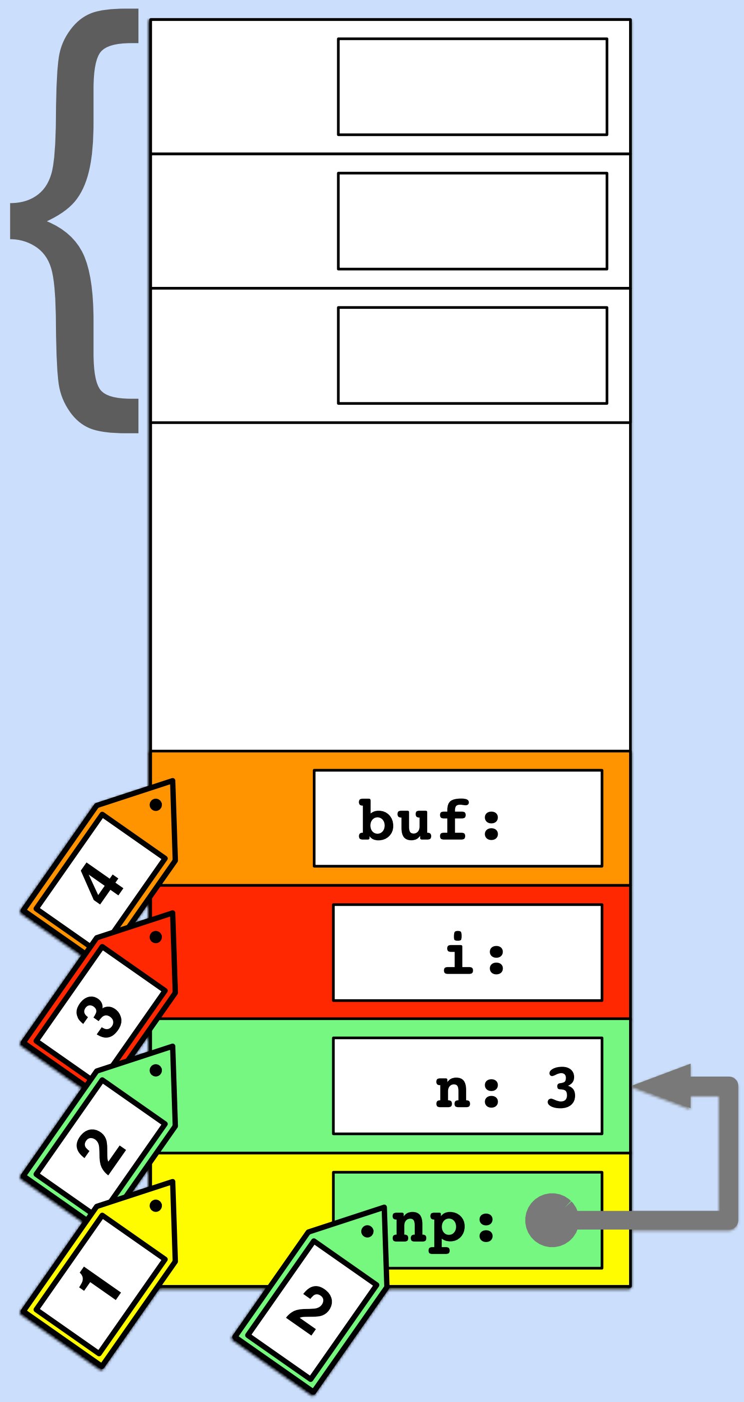
# Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf(
  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

[ret, ret + arg0)

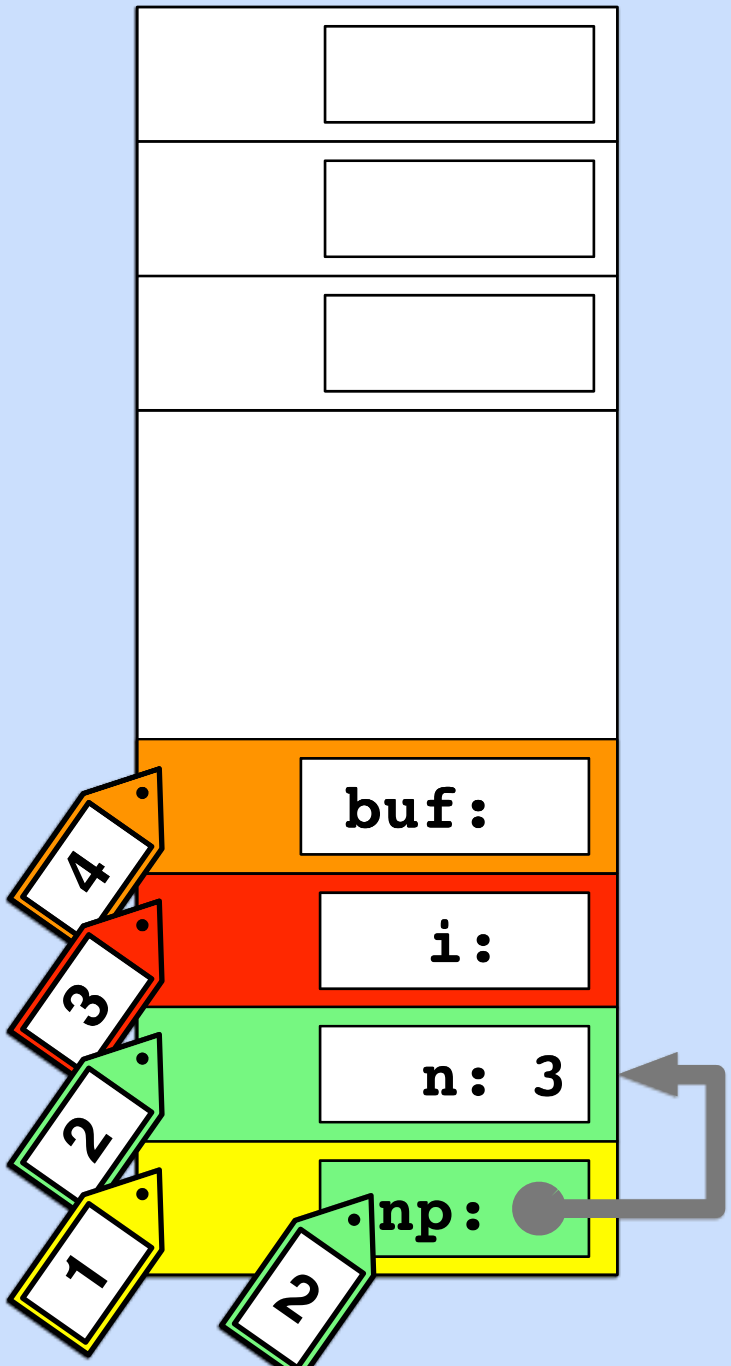


# Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

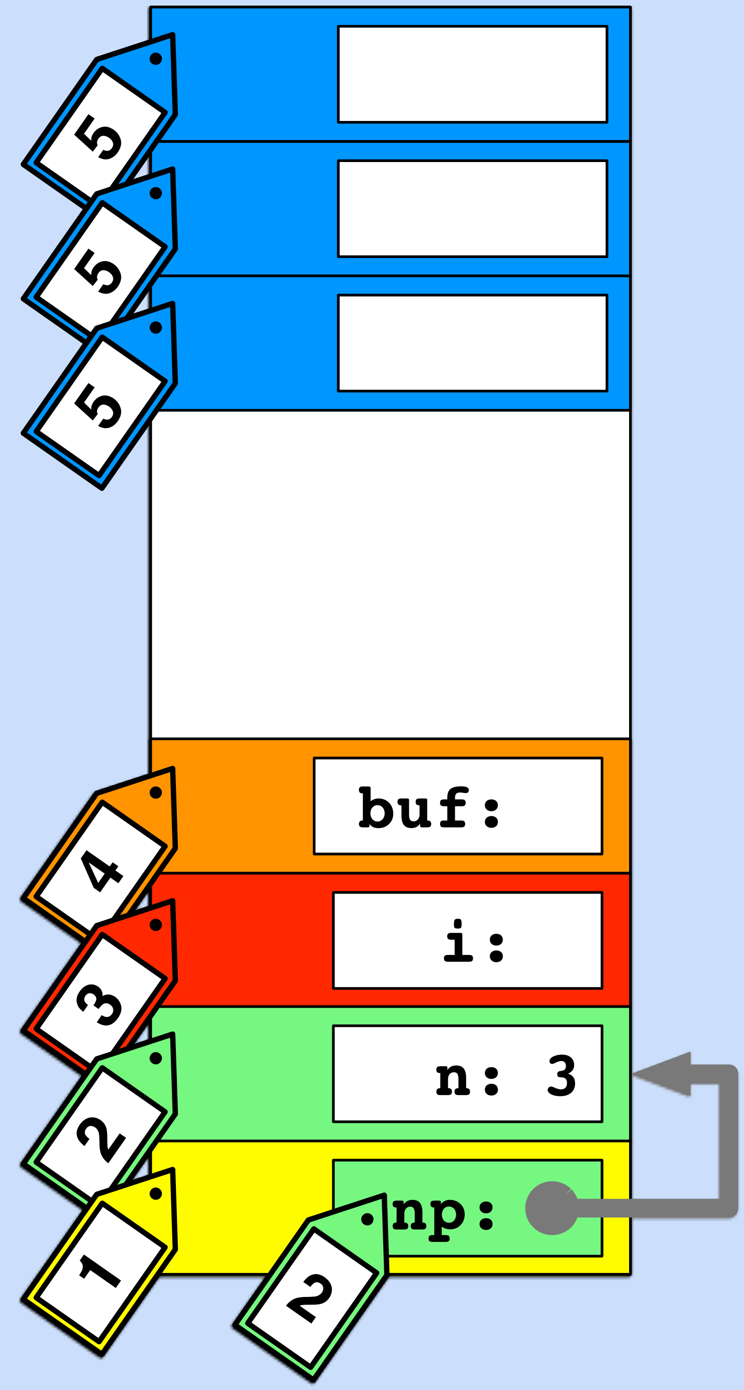


# Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



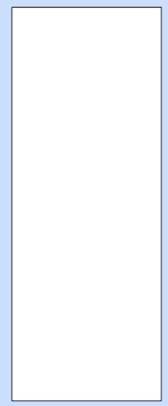
# 1 Assigning taint marks

Static

## Static memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

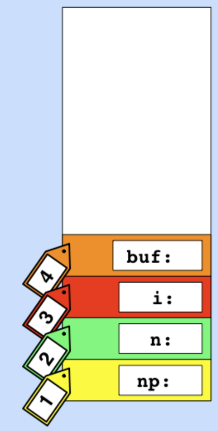
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



## Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

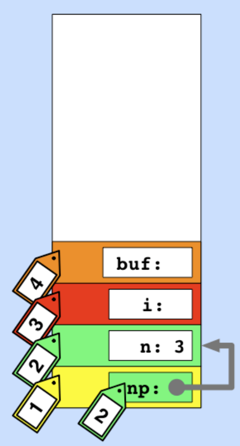



Dynamic

## Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

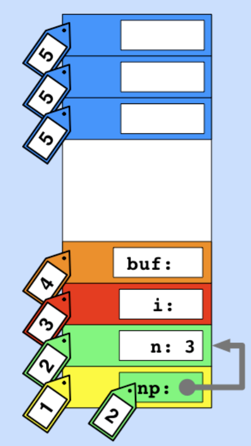
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



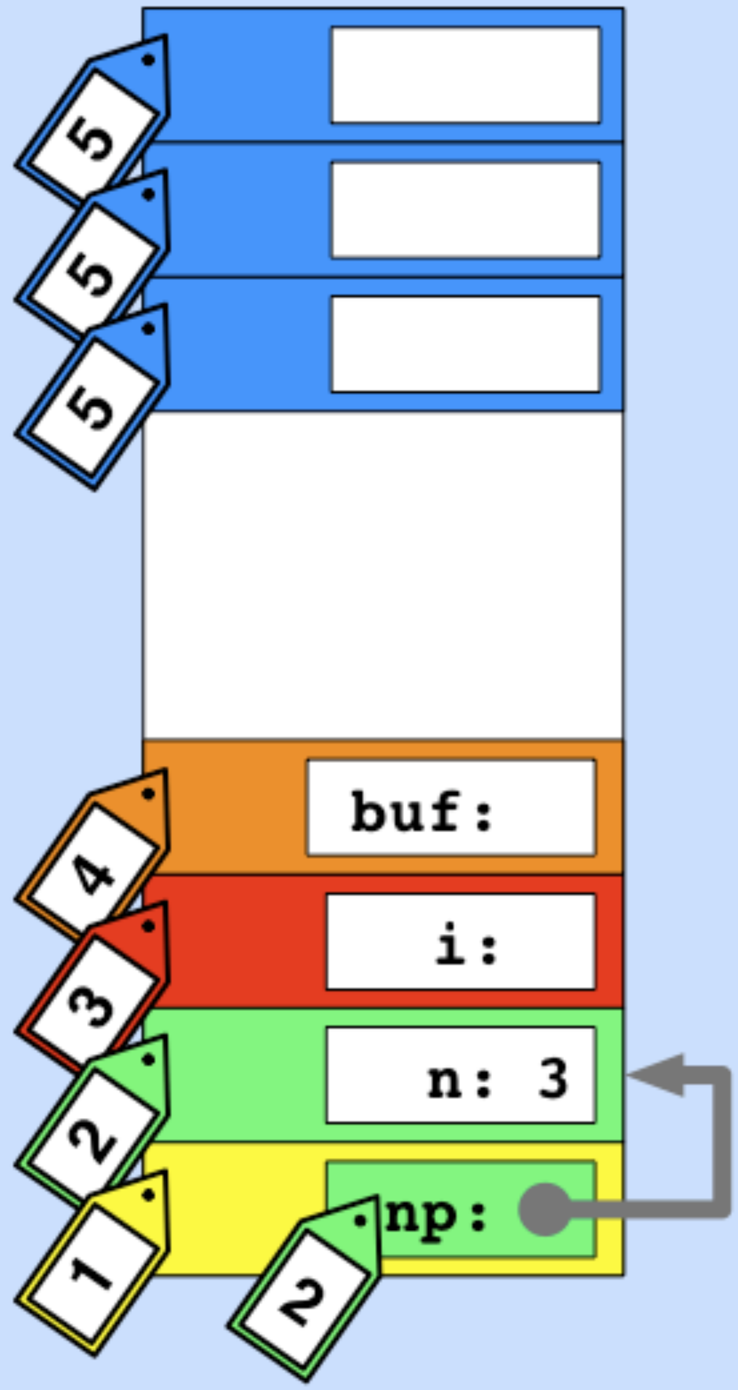

Memory

Pointers

# Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

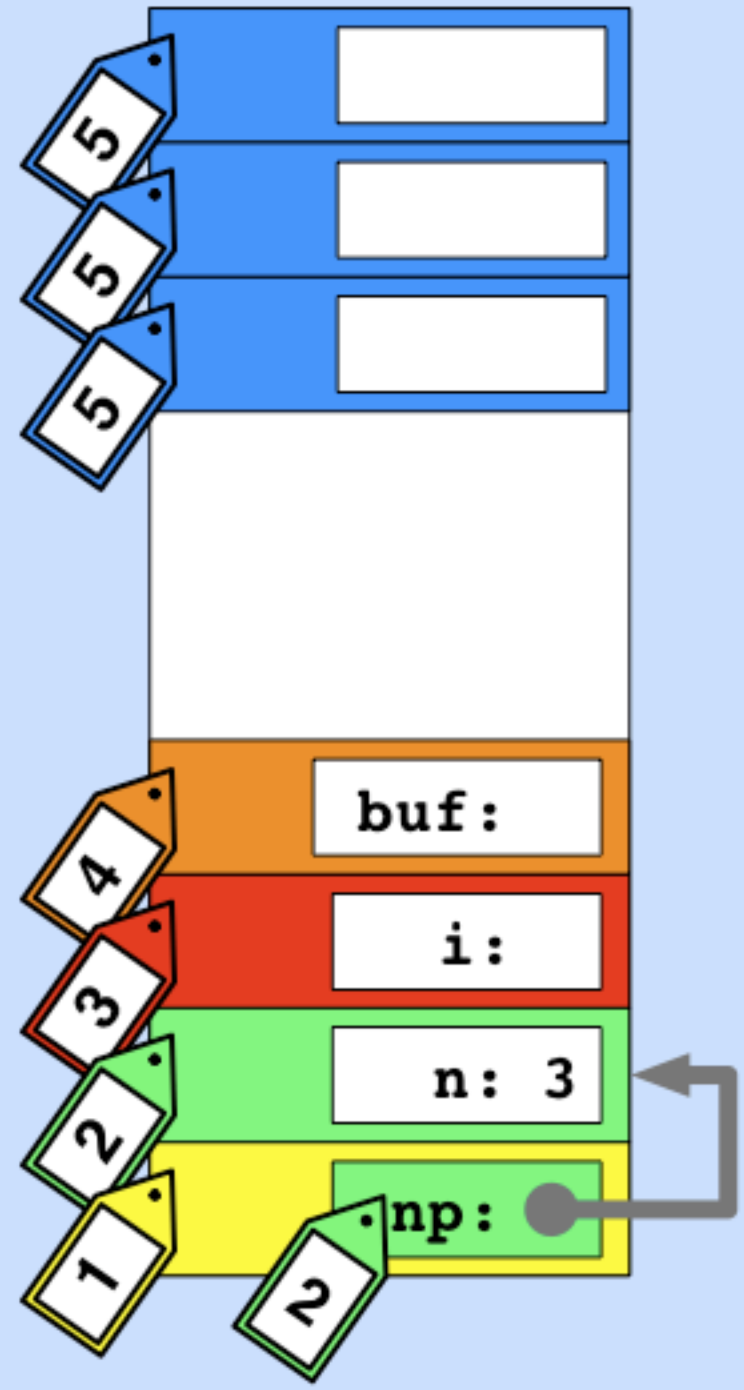


# Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf(

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```

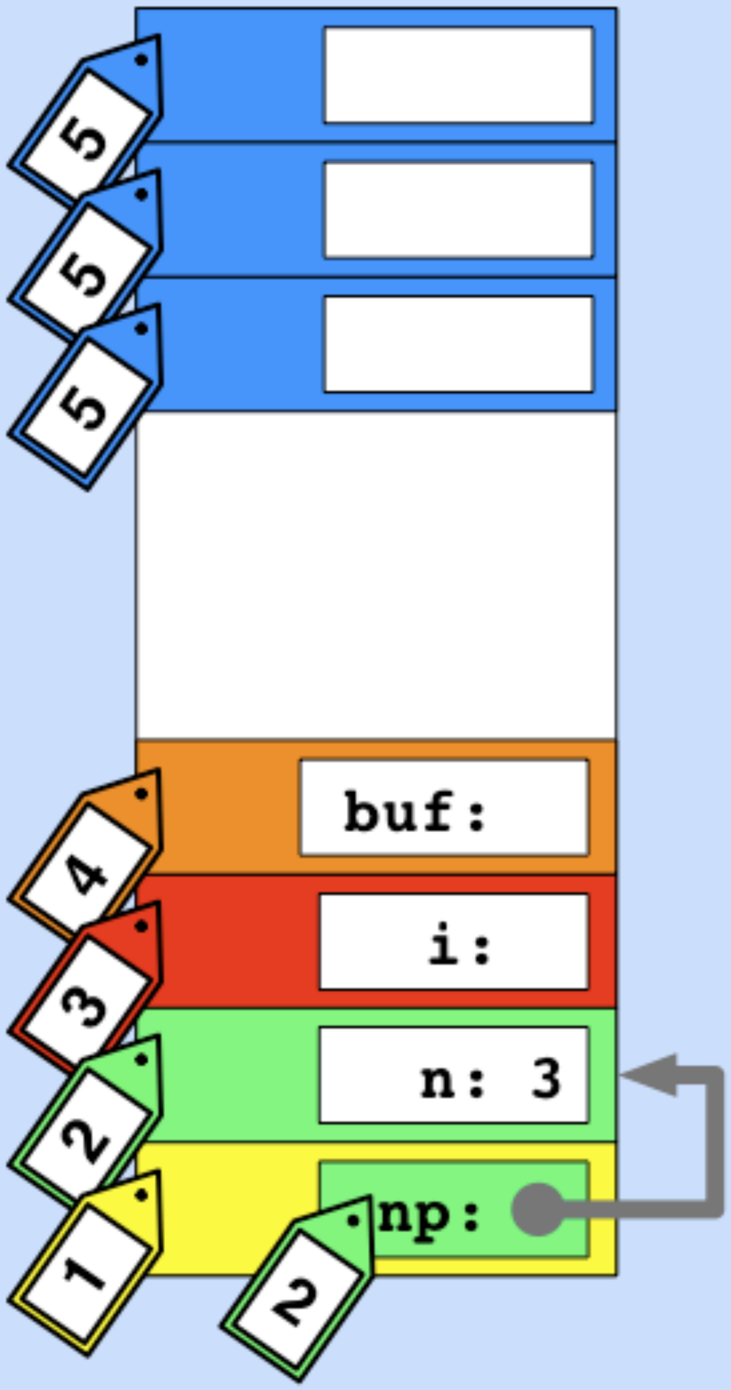
return value of malloc



# Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

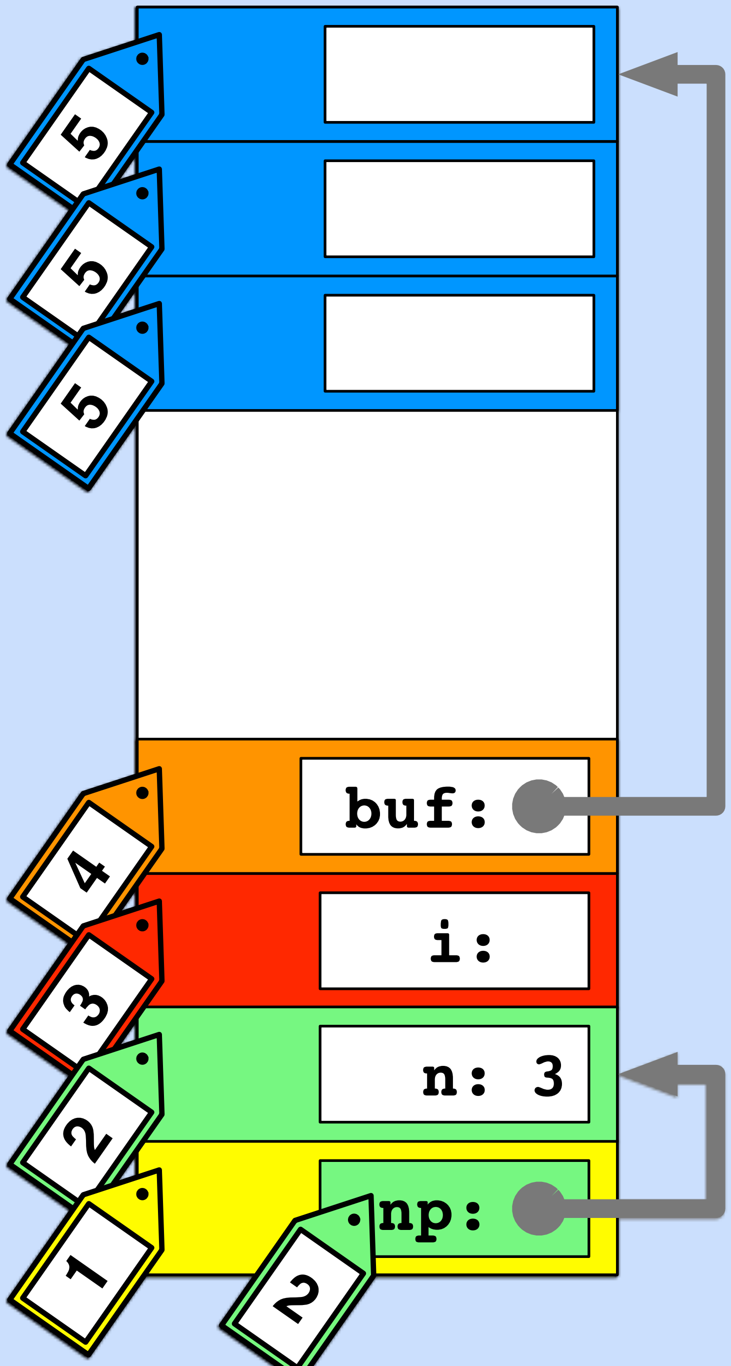
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



# Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

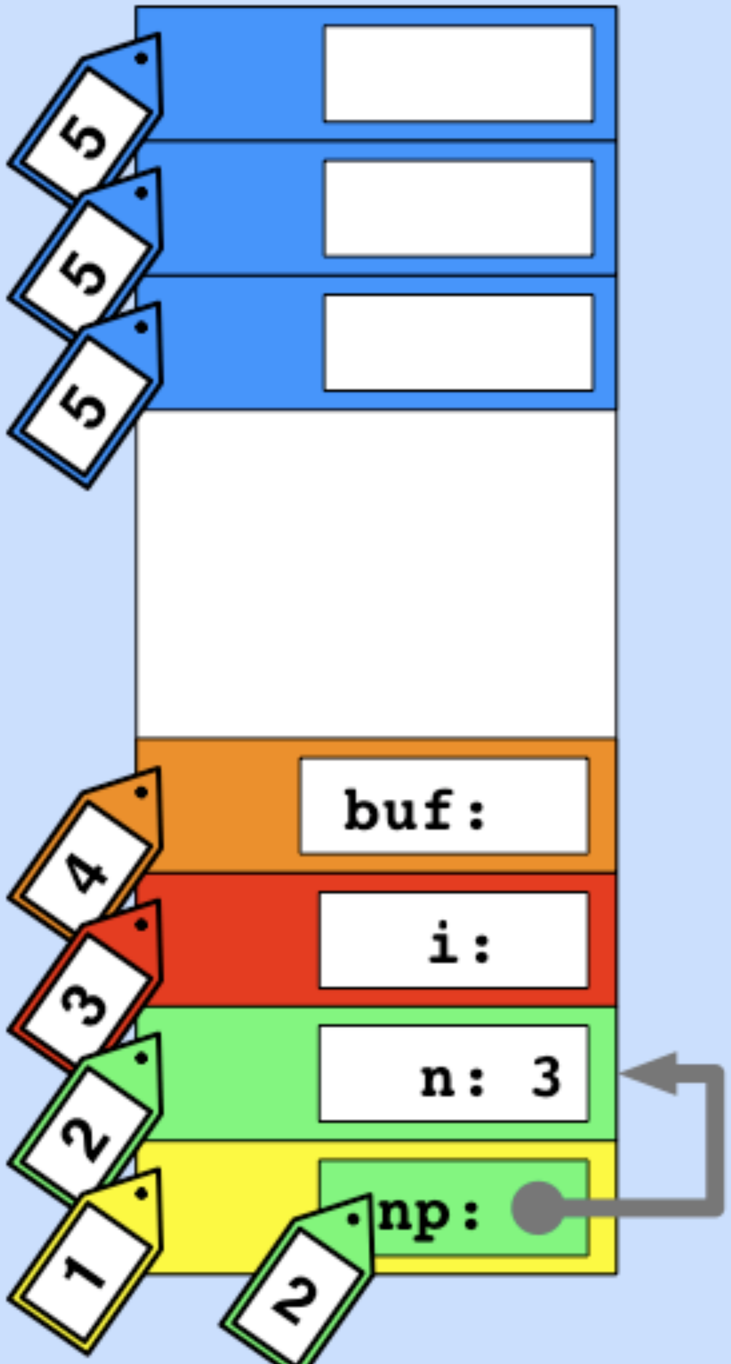
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



# Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



# 1 Assigning taint marks

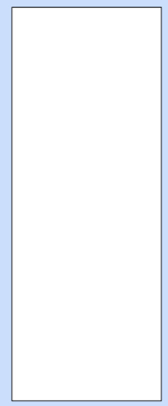
Static

Dynamic

## Static memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

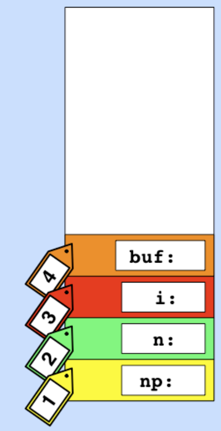
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```



## Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

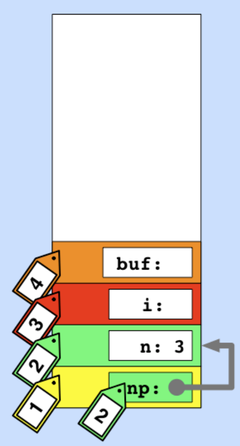
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

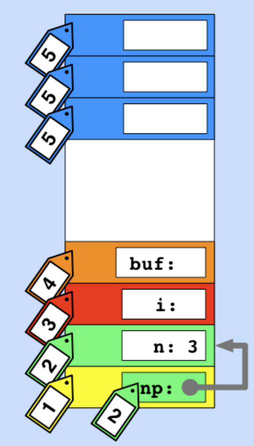
```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




## Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```
void main() {
  int *np, n, i, *buf;

  np = &n;

  printf(“Enter size: “);
  scanf(“%d”, np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
    *(buf + i) = rand()%10;
  ...
}
```




Memory

Pointers

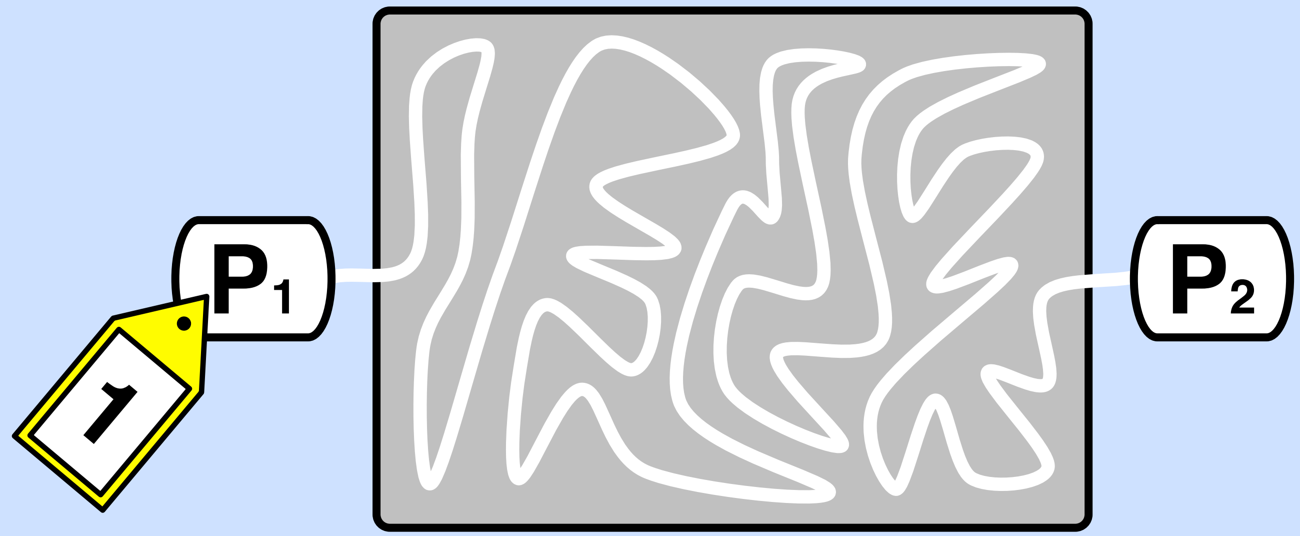
# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division,
OR, XOR



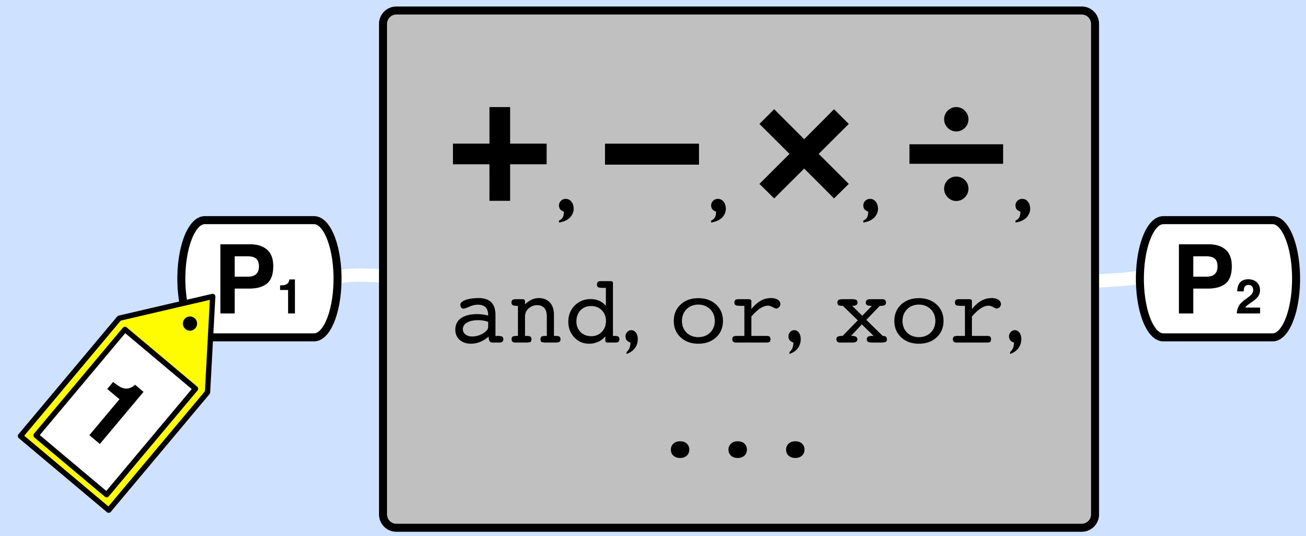
# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division,
OR, XOR



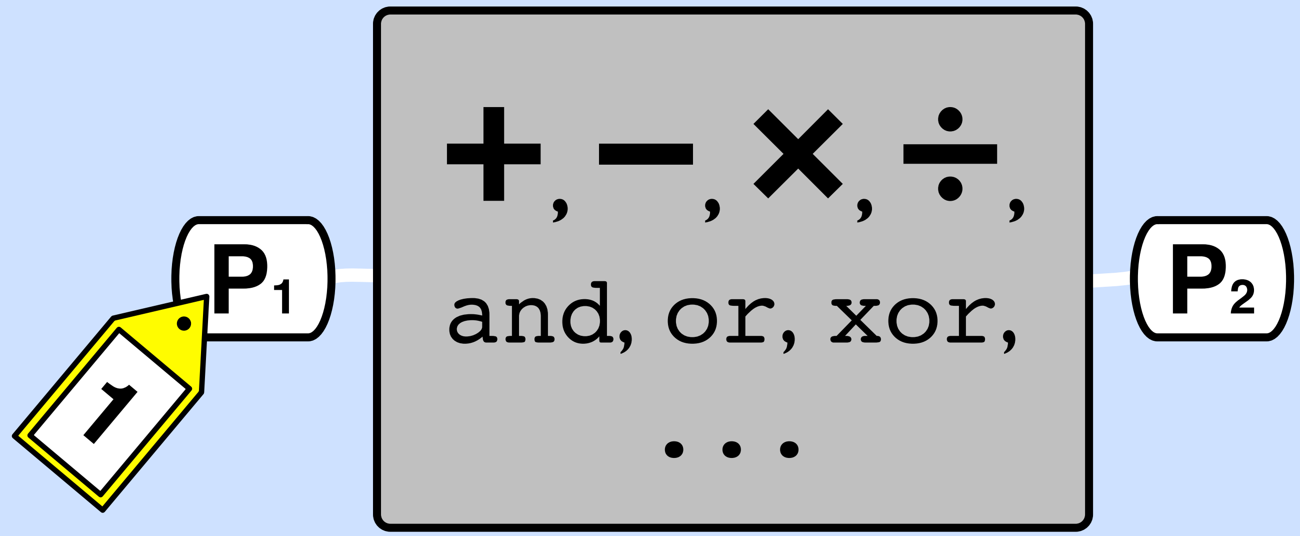
# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division,
OR, XOR




Should the result be tainted?
If so, how?

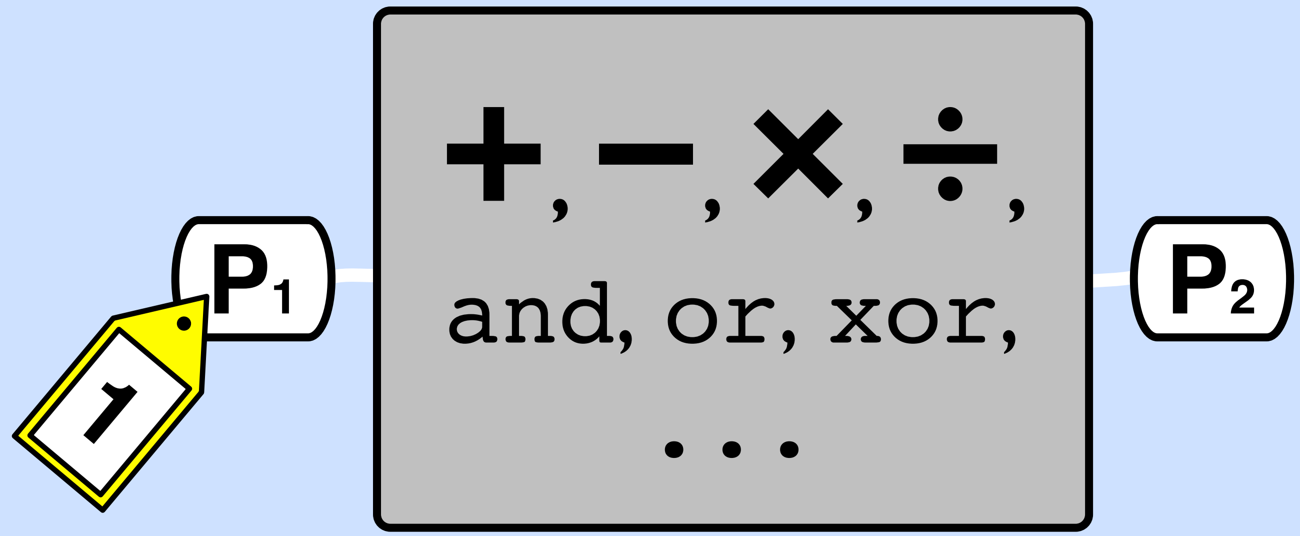
# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR




Should the result be tainted?
If so, how?

- Propagation must take into account both operation semantics and programmer intent

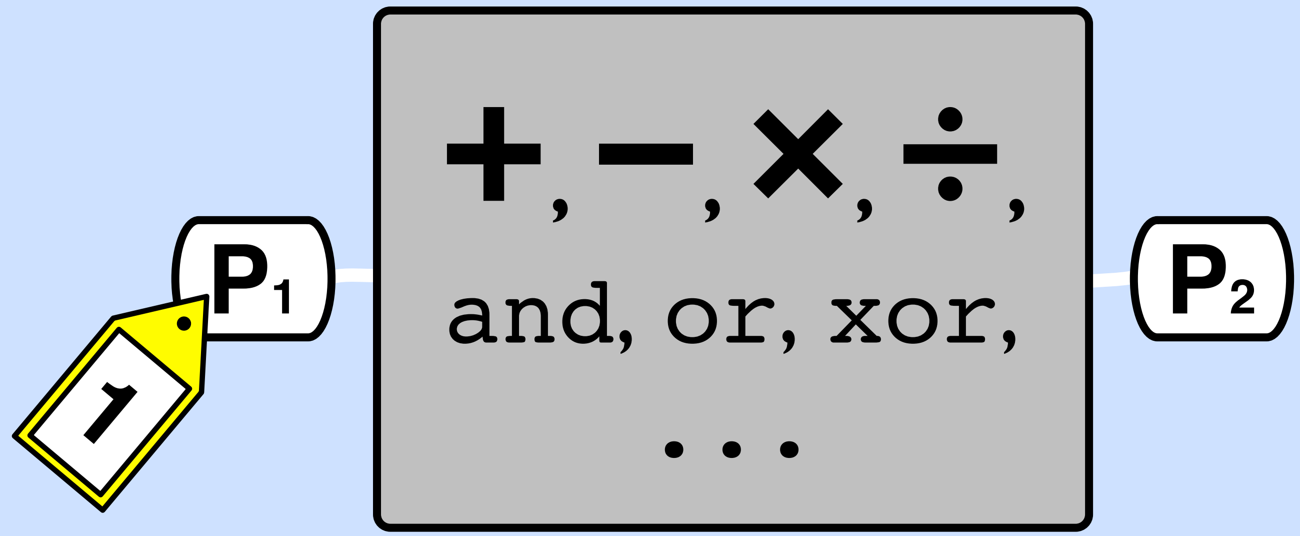
# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division,
OR, XOR




Should the result be tainted?
If so, how?

- Propagation must take into account both operation semantics and programmer intent
- Our policy is based on knowledge of C/C++/assembly and patterns observed in real software

# 2 Propagating taint marks

Overview

**Addition, Subtraction**

AND

Multiplication, Division, OR, XOR

## A +/– B = C

| A | B | C |
| --- | --- | --- |
| 1 | | 1 |
| | 1 | 1 / no taint |
| | ••• | |

Most common use of addition and subtraction is to add or subtract a pointer and an offset

# 2 Propagating taint marks

- Overview
- Addition, Subtraction
- **AND**
- Multiplication, Division, OR, XOR

## A & B = C

| A | B | C |
|---|---|---|
| 1 | | 1 or no taint |
| | ••• | |

The result of anding a pointer and a mask should be treated differently depending on the value of the mask

`c = a & 0xffffff00` - base address

`c = a & 0x000000ff` - offset

# 2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR

We found zero cases where the result of any of these operations was a pointer

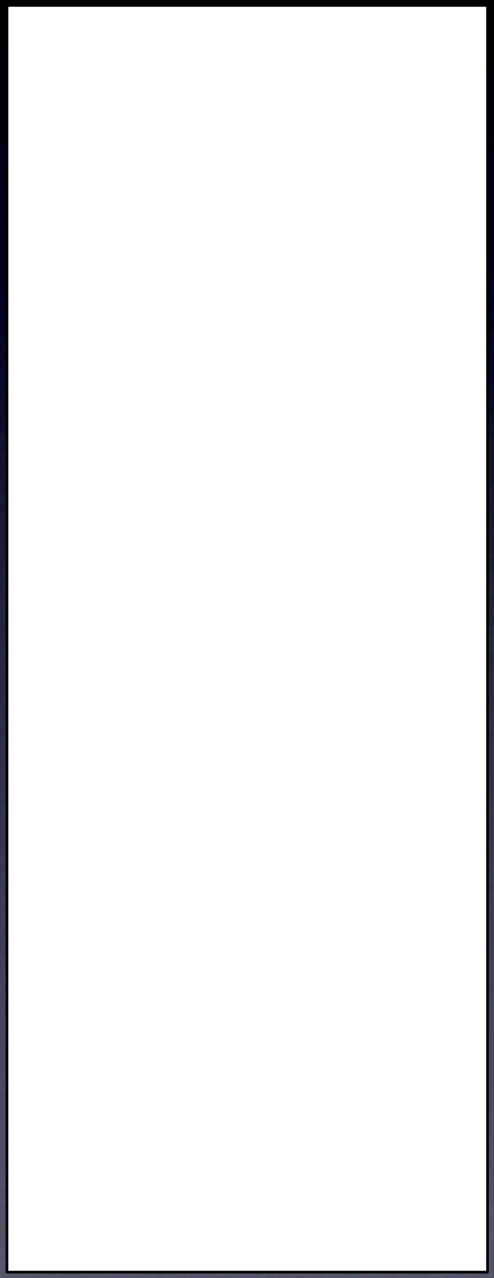
# 3 Checking taint marks

When memory is accessed through a pointer:
compare the memory taint mark and the pointer taint mark

| Pointer | Memory | IMA? |
|---|---|---|
| 2 | 2 | no |
| 1 | 2 | yes |
| 3 | | yes |
| | 3 | yes |
| | | yes |

# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



# Preventing IMAs

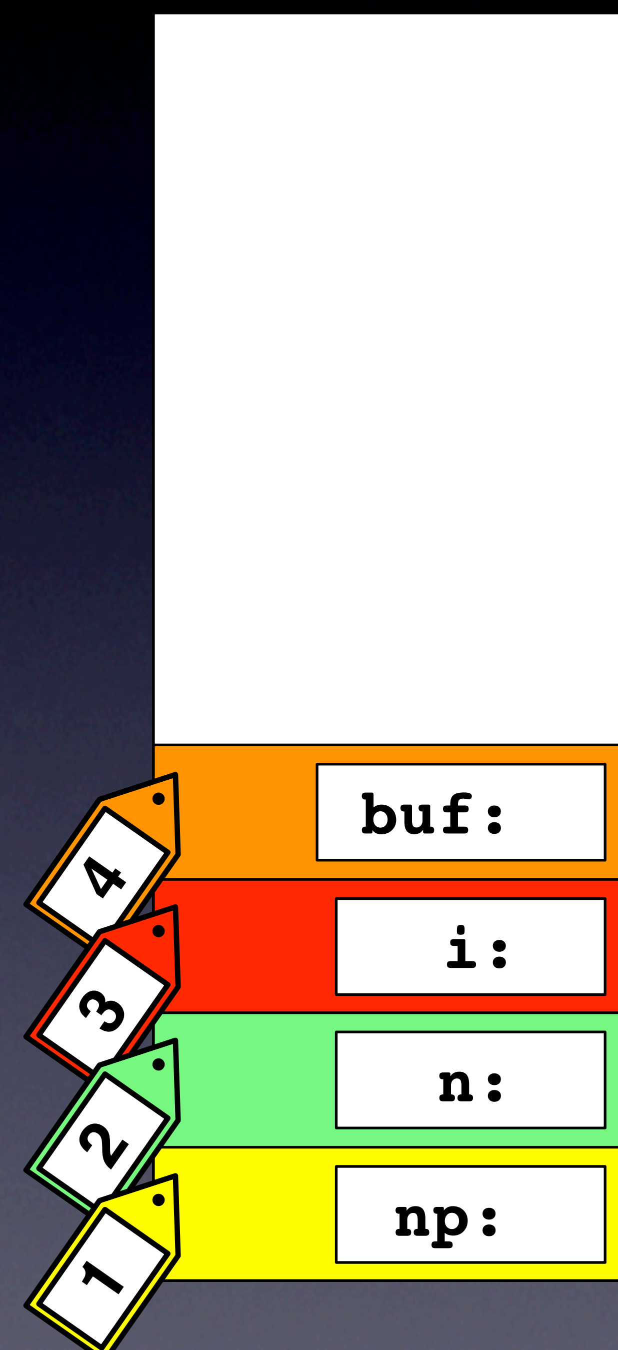
```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

# Preventing IMAs

```
void main() {
int *np, n, i, *buf;

np = &n;

printf(“Enter size: “);
scanf(“%d”, np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```

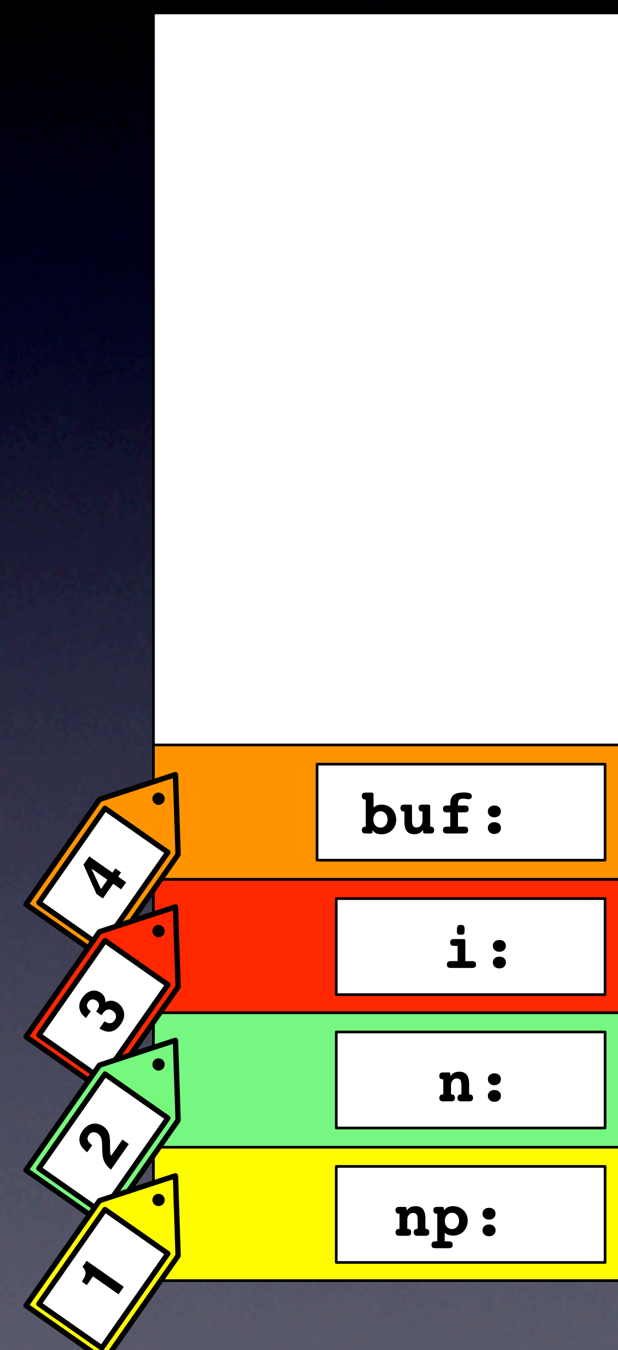
# Preventing IMAs

```c
void main() {
int *np, n, i, *buf;

np = &n;

printf(“Enter size: “);
scanf(“%d”, np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```



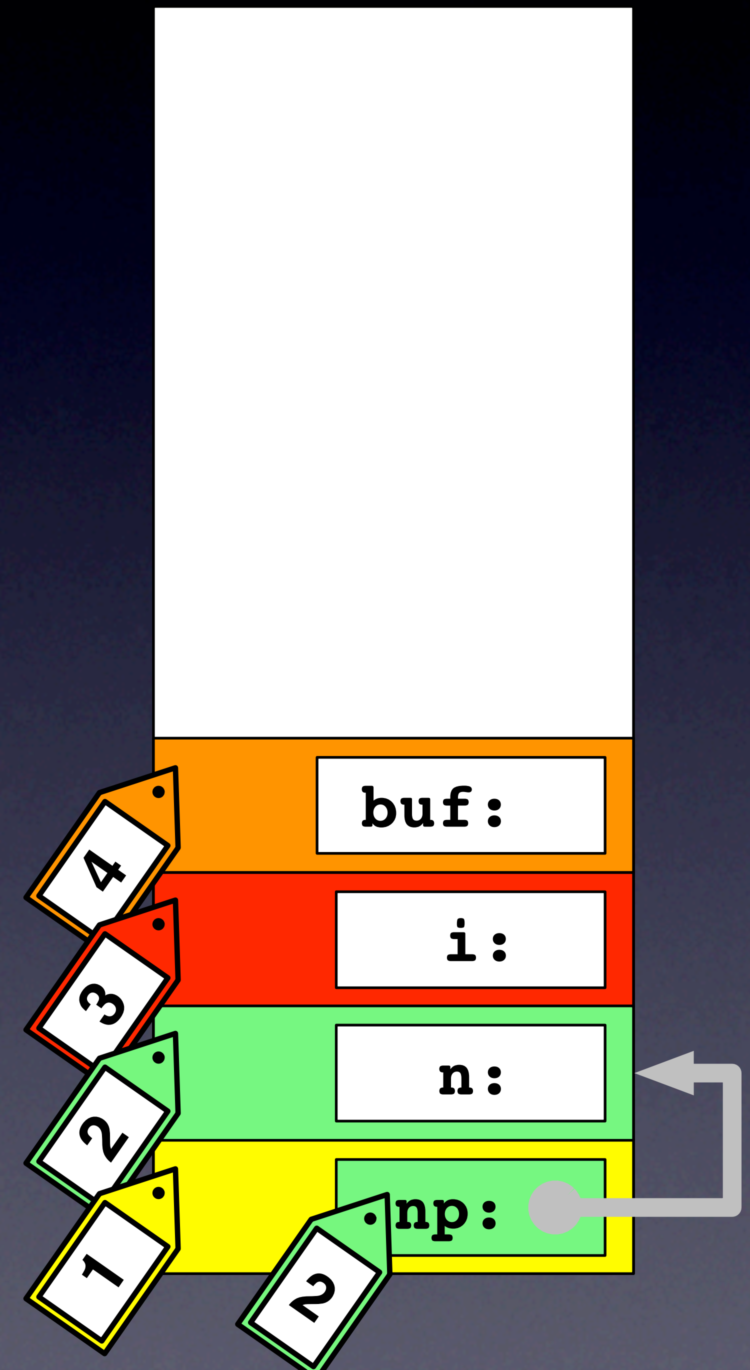
# Preventing IMAs

```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
...
}
```



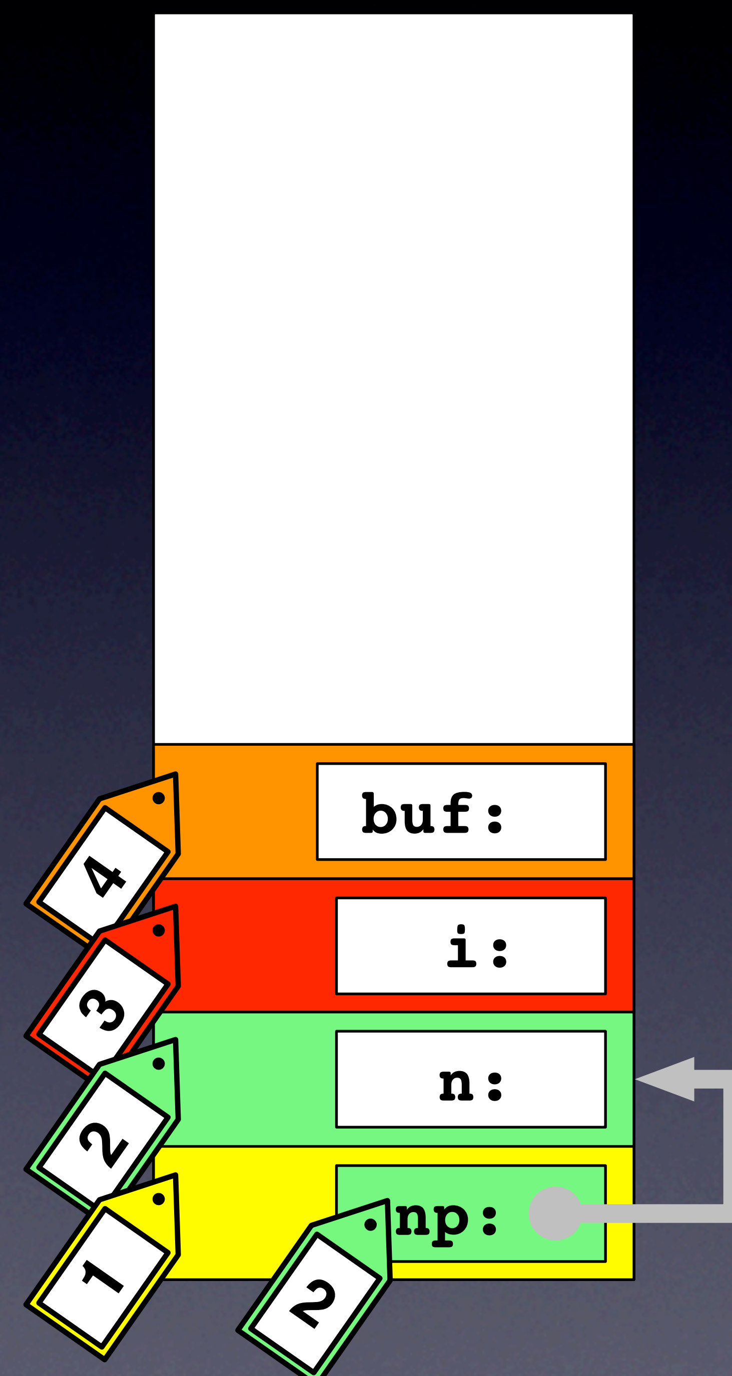
# Preventing IMAs

```
   void main() {
1.  int *np, n, i, *buf;

2.  np = &n;

3.  printf("Enter size: ");
4.  scanf("%d", np);

5.  buf = malloc(n * sizeof(int));

6.  for(i = 0; i <= n; i++)
7.    *(buf + i) = rand()%10;
    ...
}
```



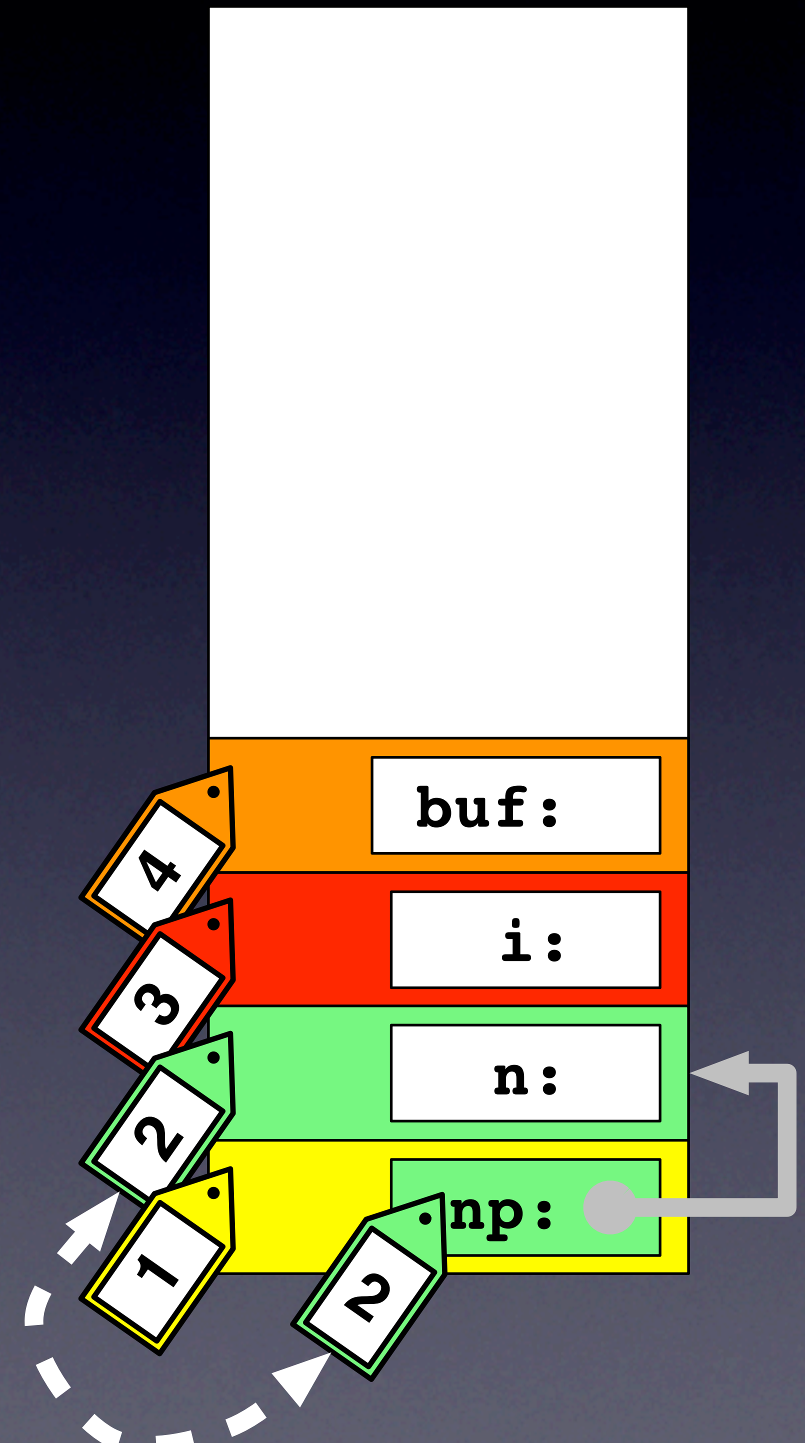
# Preventing IMAs

```c
   void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



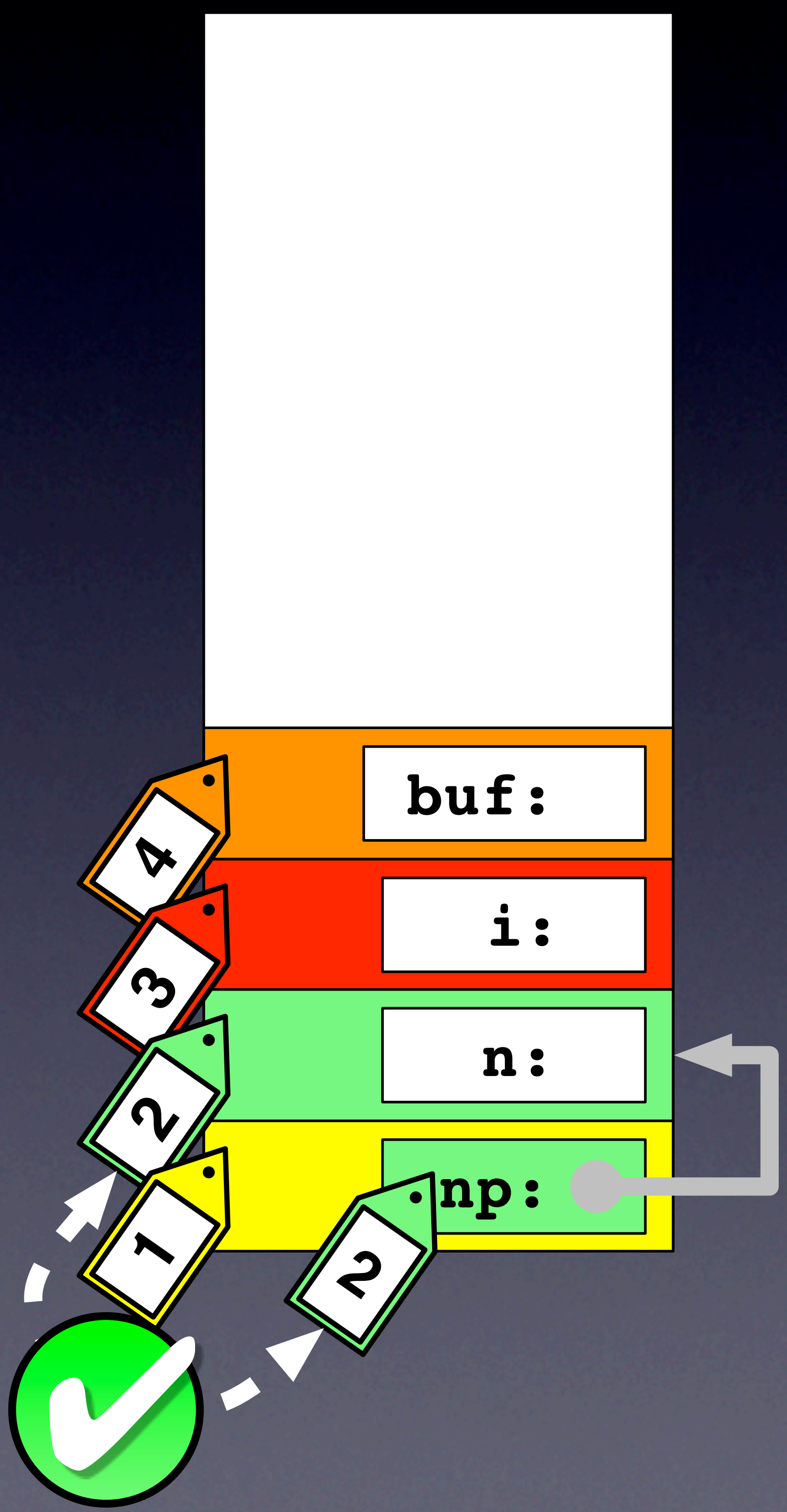
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



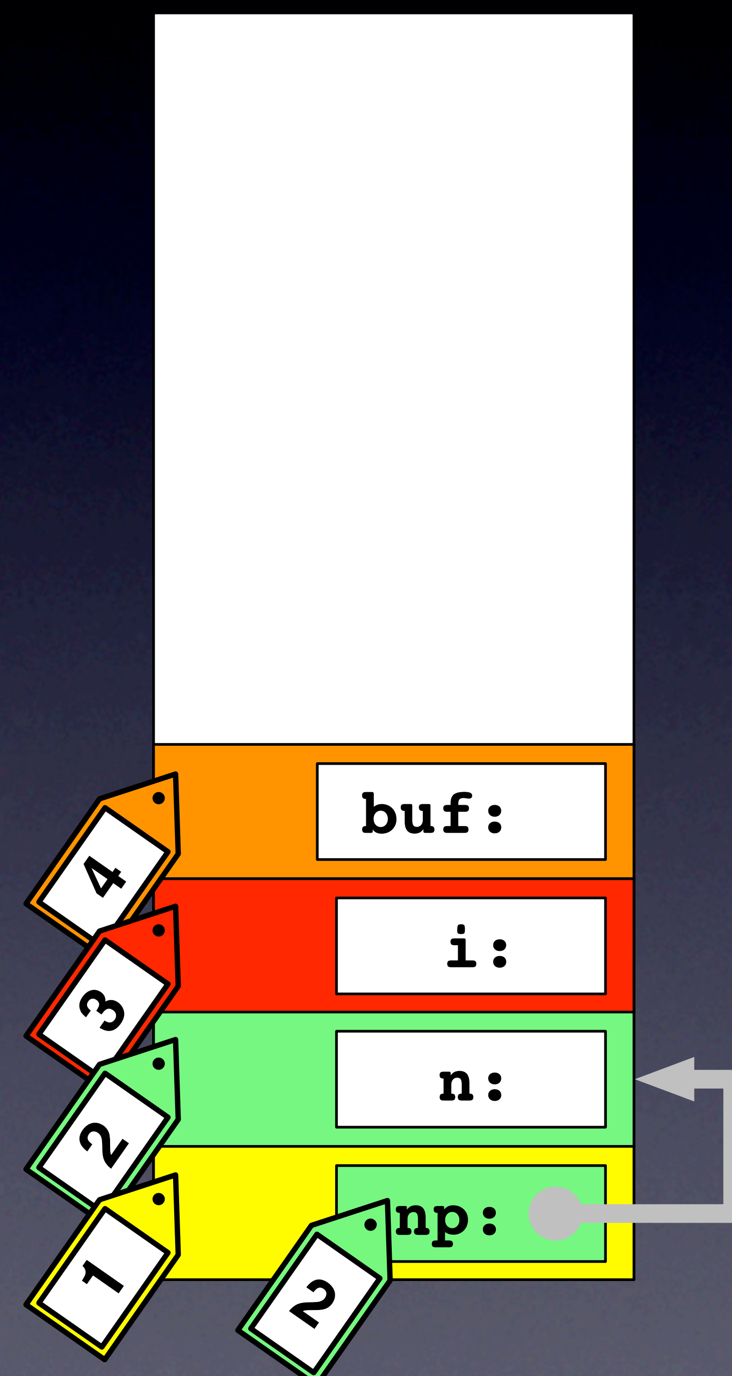
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



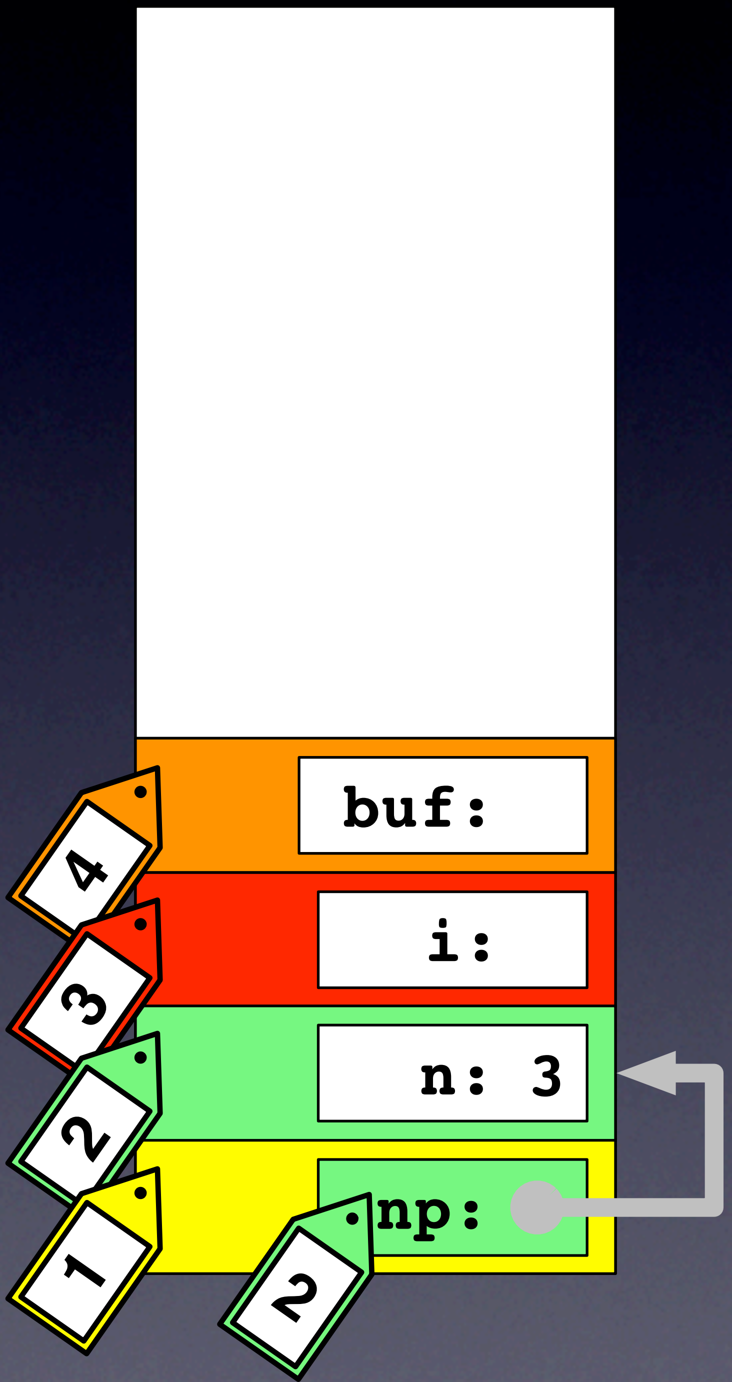
# Preventing IMAs

```
   void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



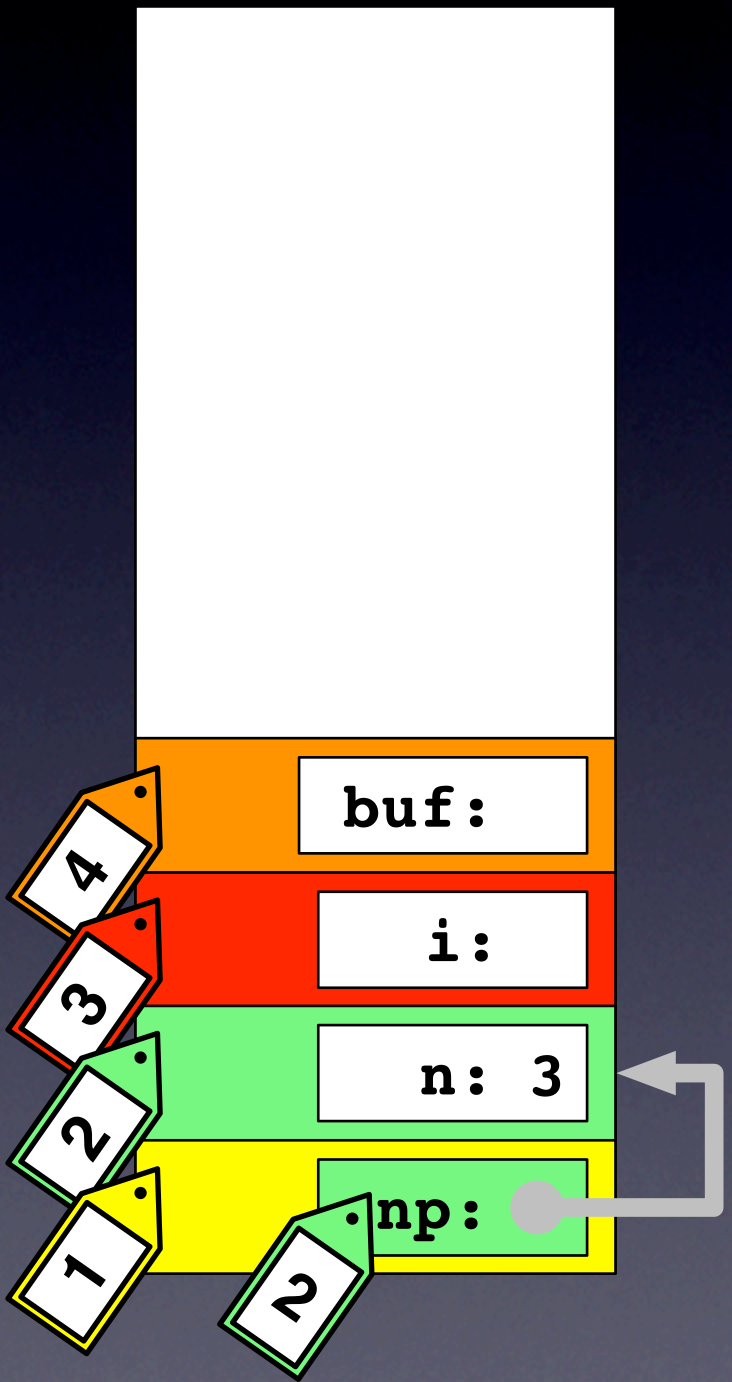
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



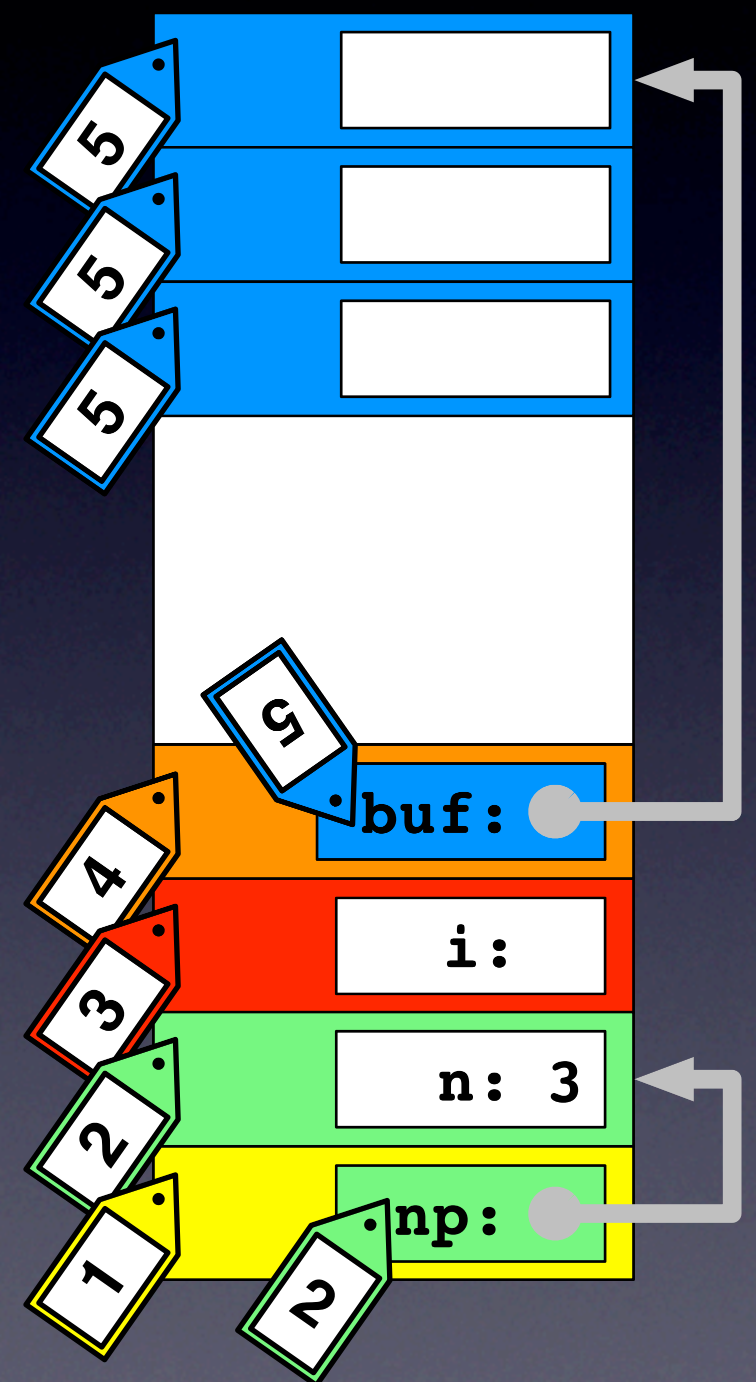
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
   ...
}
```



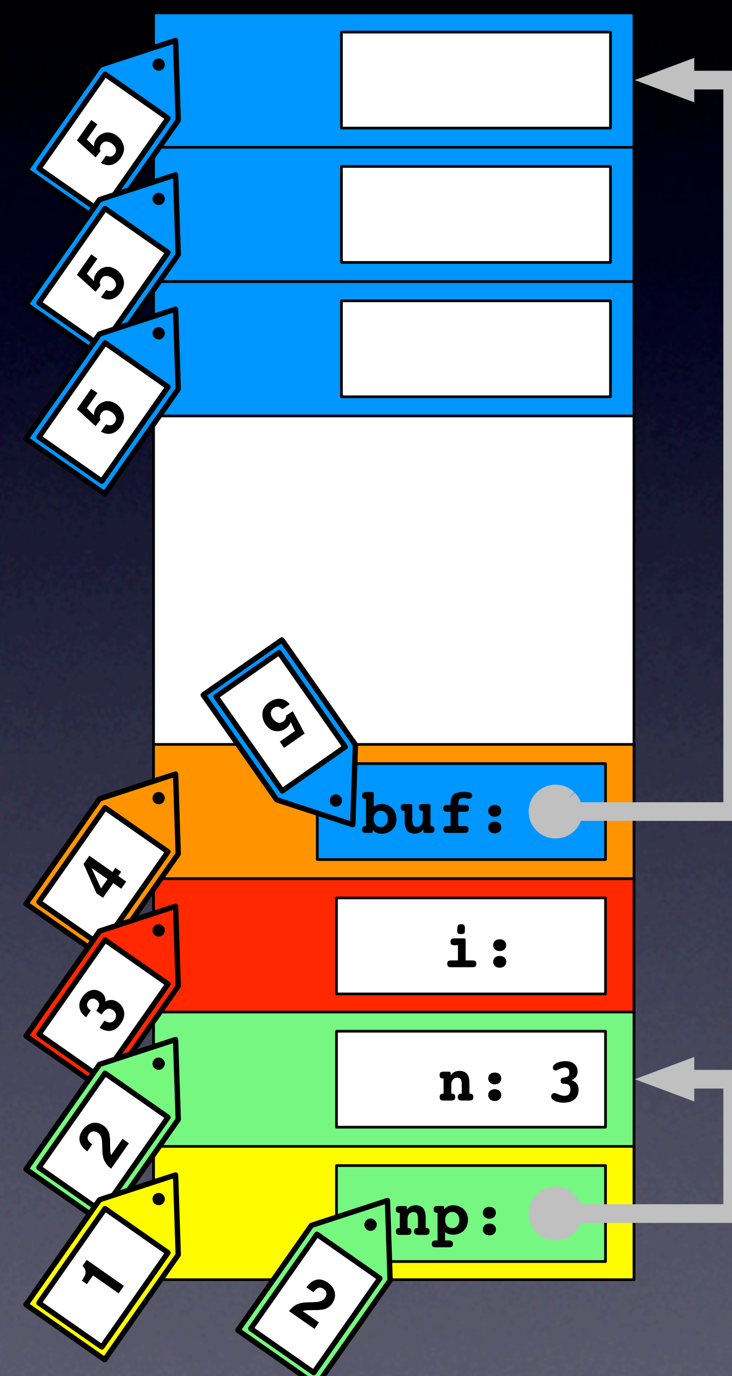
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



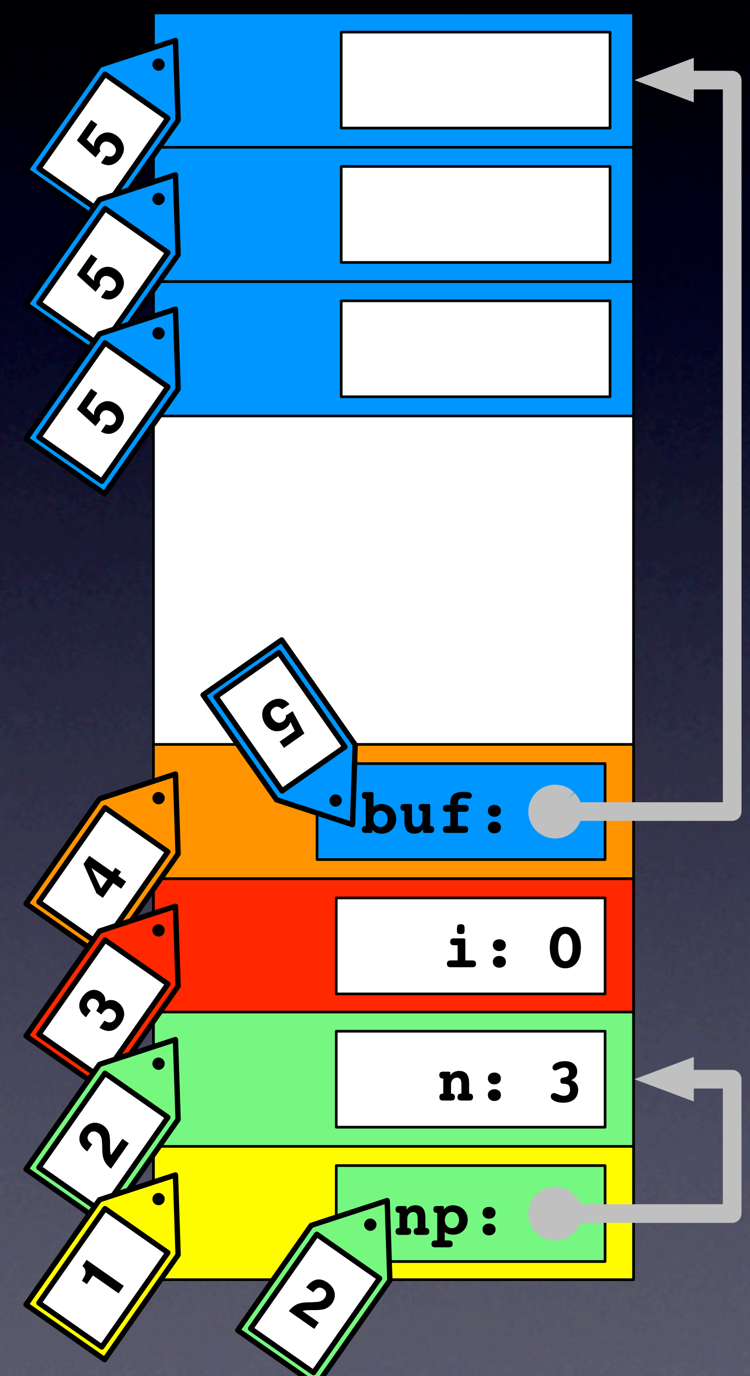
# Preventing IMAs

```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```



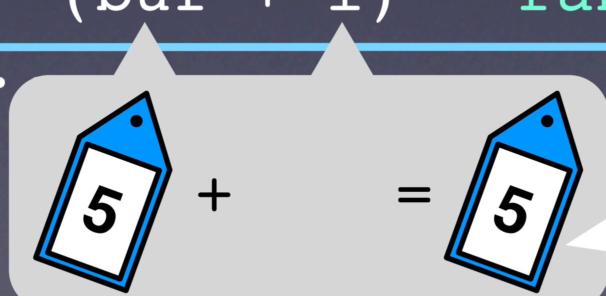
# Preventing IMAs

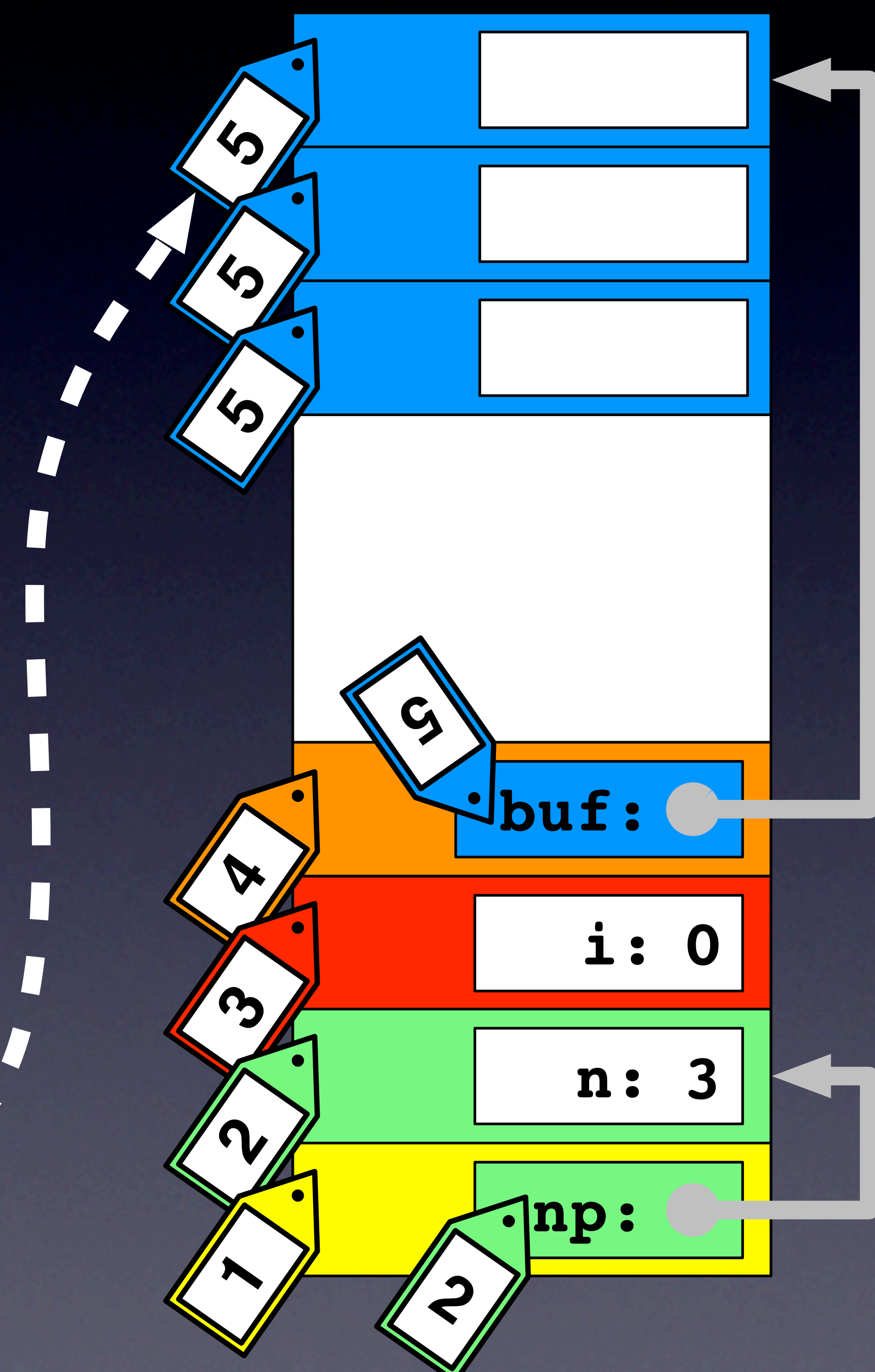
```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



# Preventing IMAs

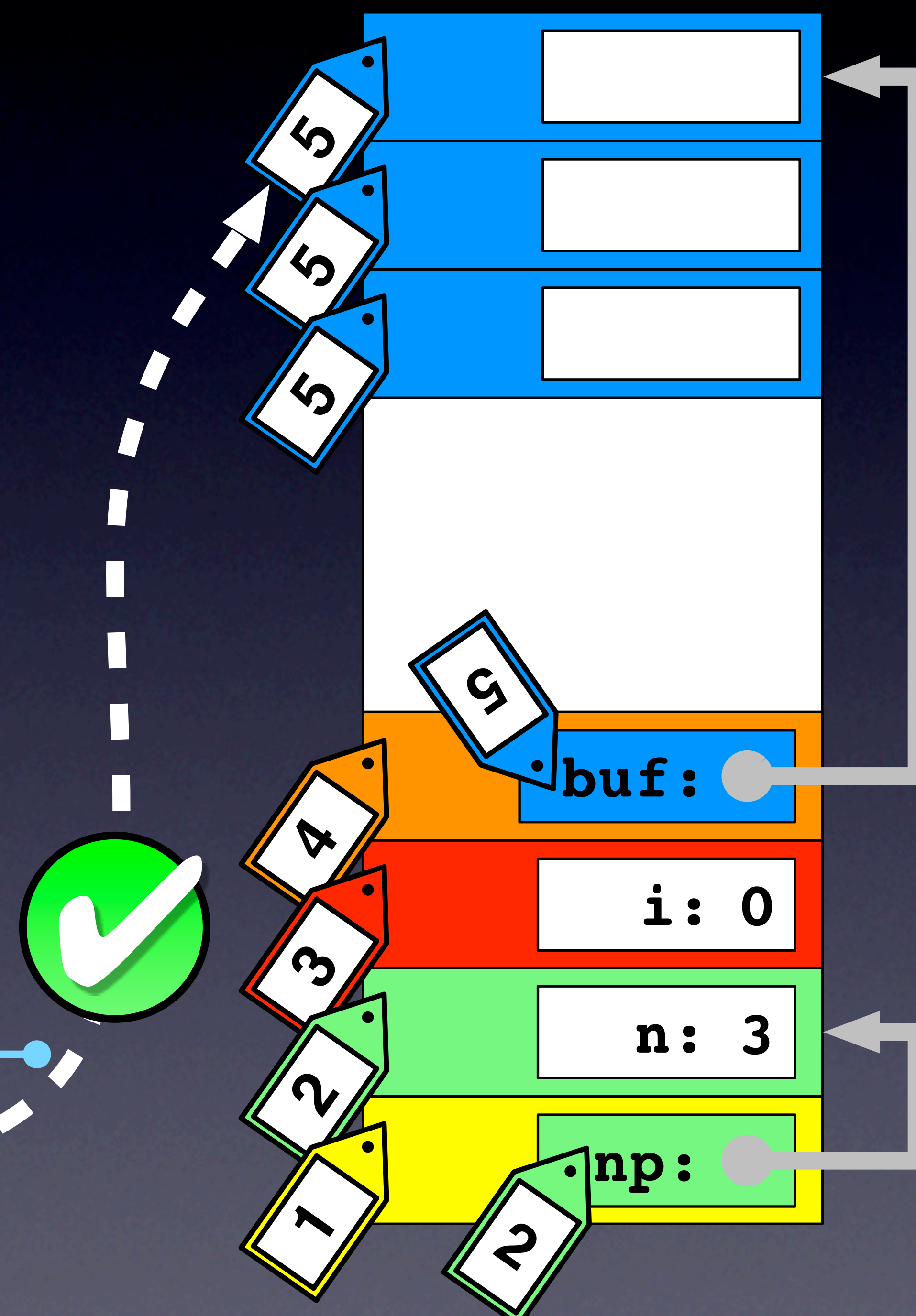
```
  void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
 *(buf + i) = rand()%10;
 ...
}
```



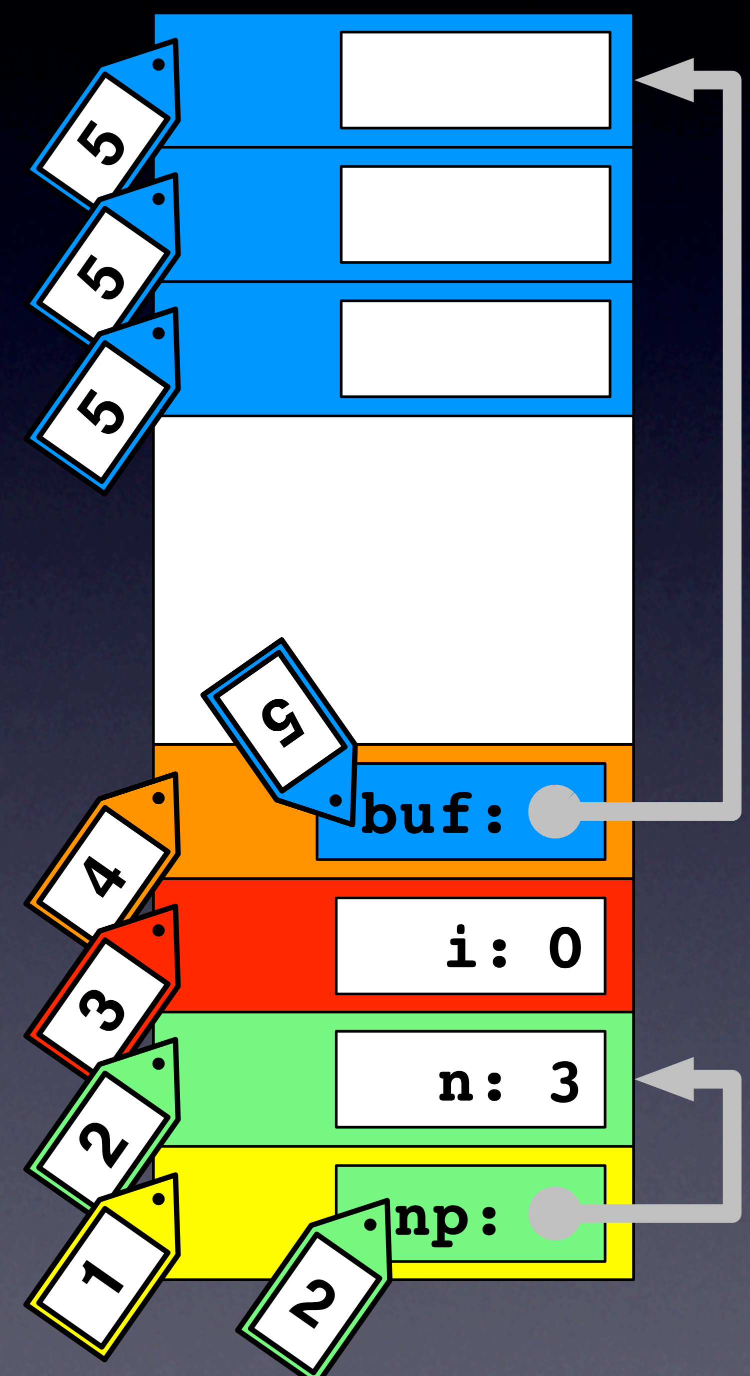
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```

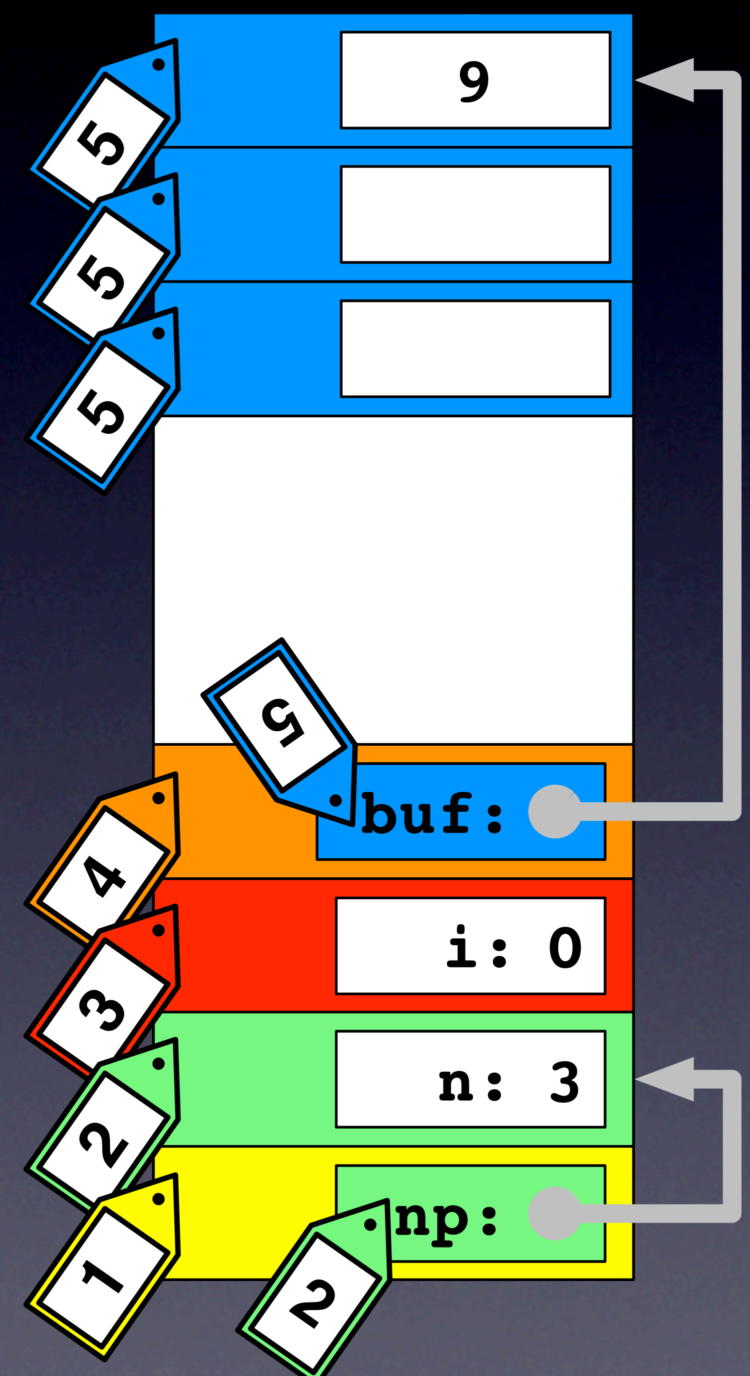
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



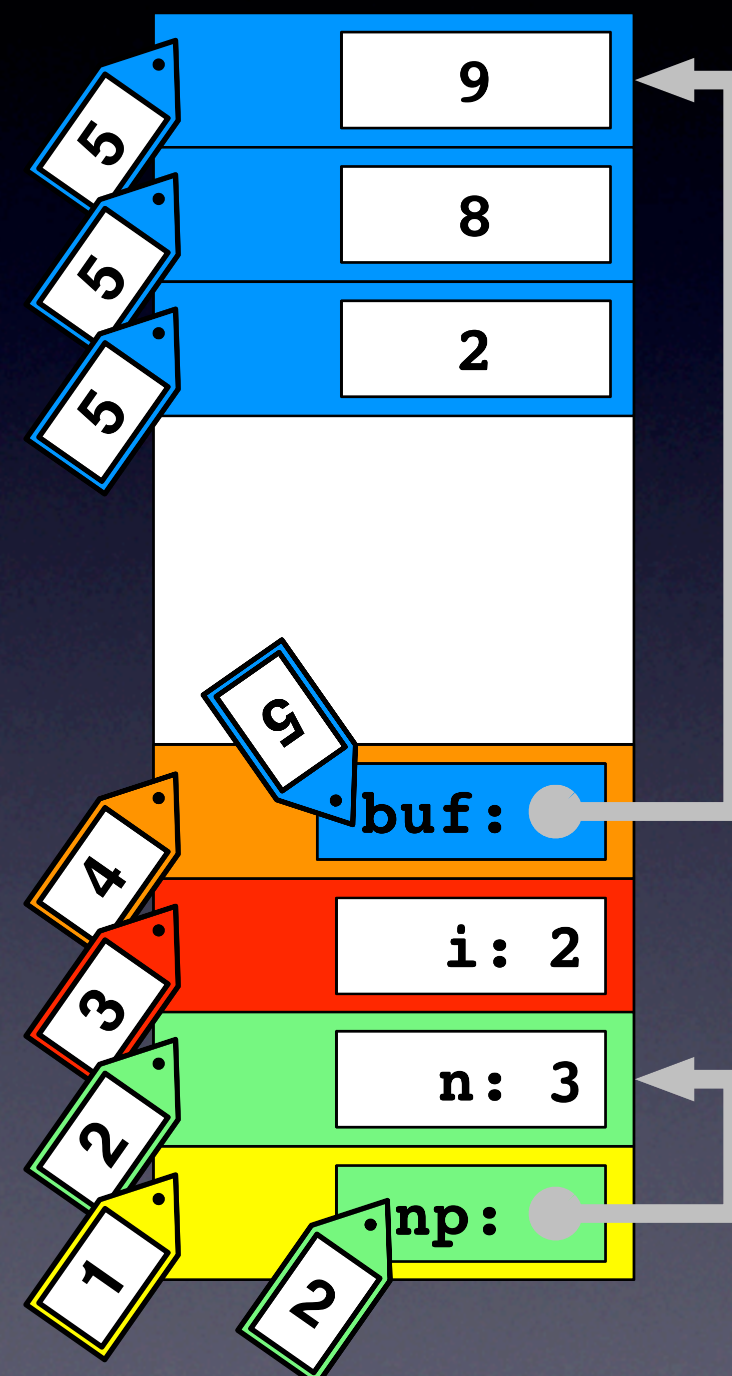
# Preventing IMAs

```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



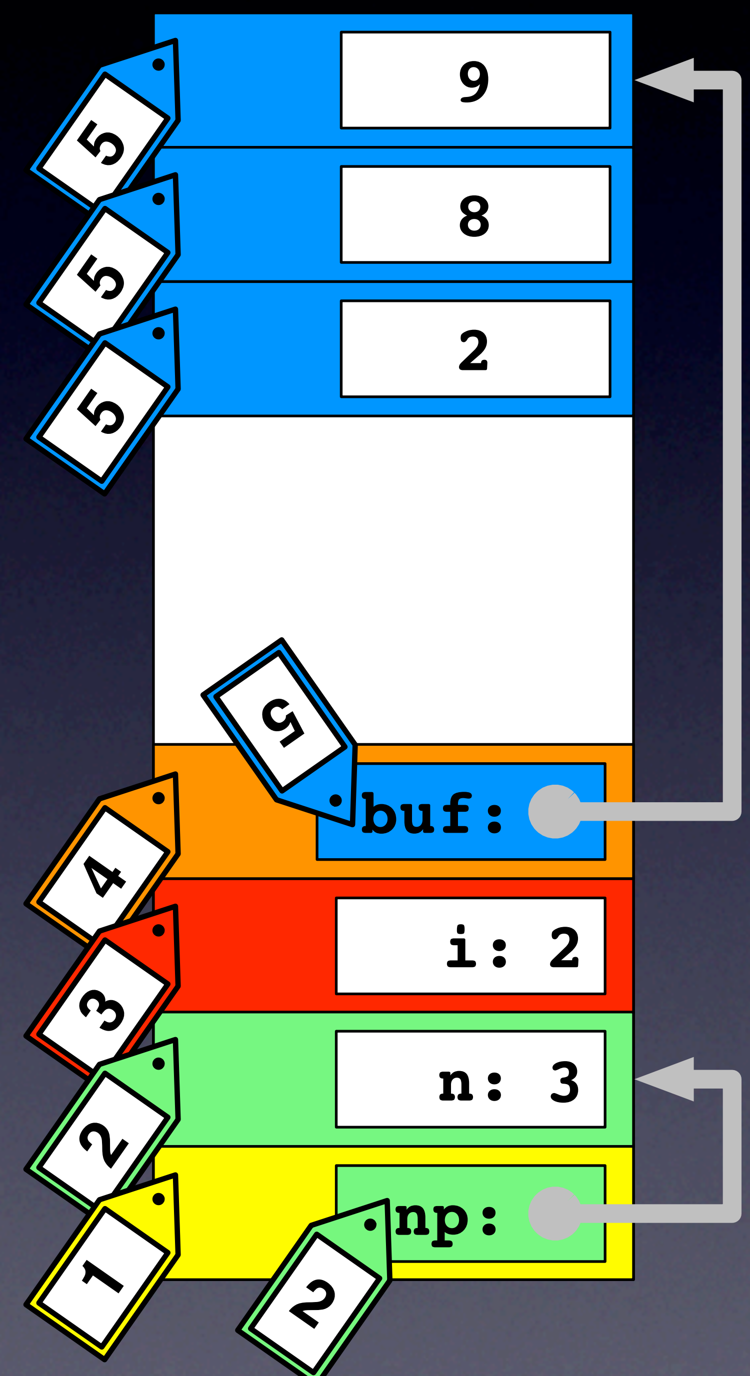
# Preventing IMAs

```
  void main() {
  int *np, n, i, *buf;

  np = &n;

  printf("Enter size: ");
  scanf("%d", np);

  buf = malloc(n * sizeof(int));

  for(i = 0; i <= n; i++)
   *(buf + i) = rand()%10;
  ...
}
```



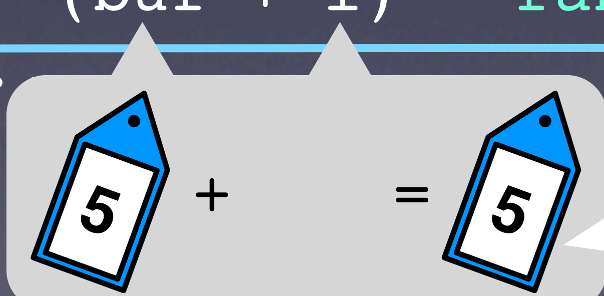
# Preventing IMAs

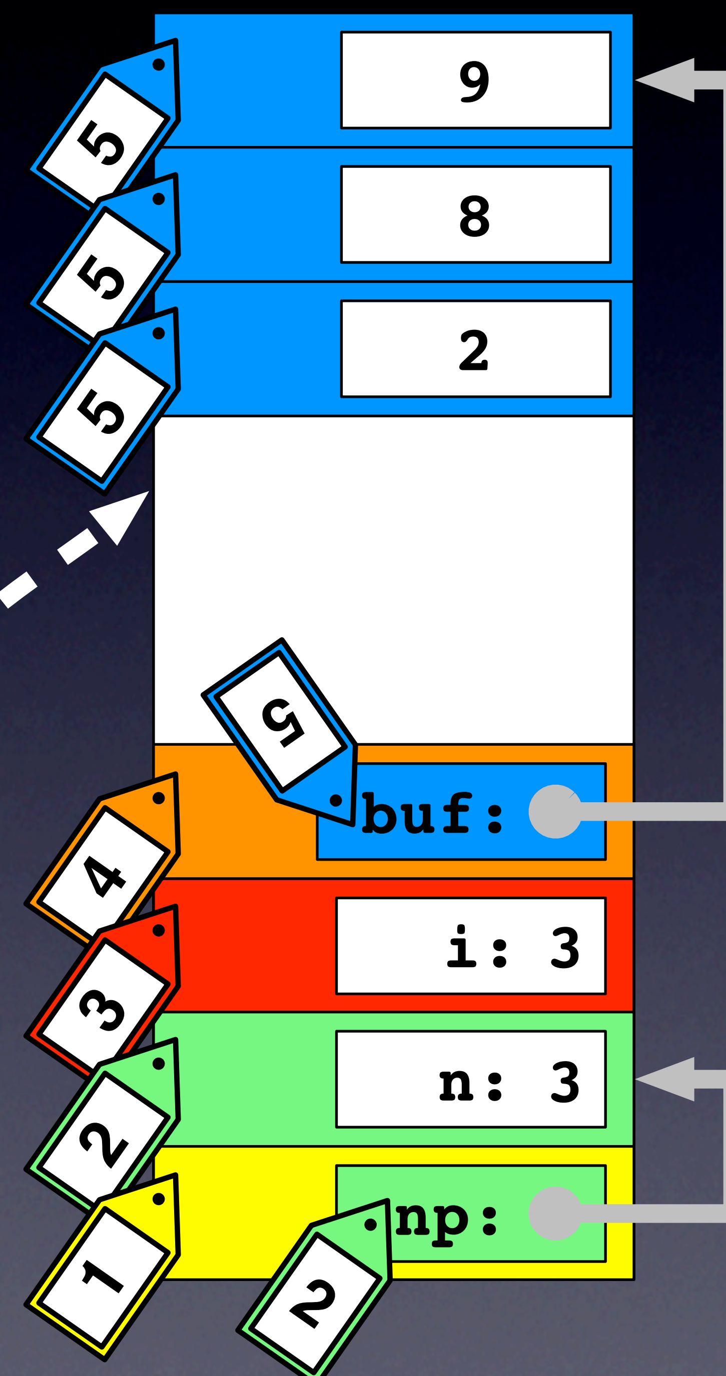
```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```



# Preventing IMAs

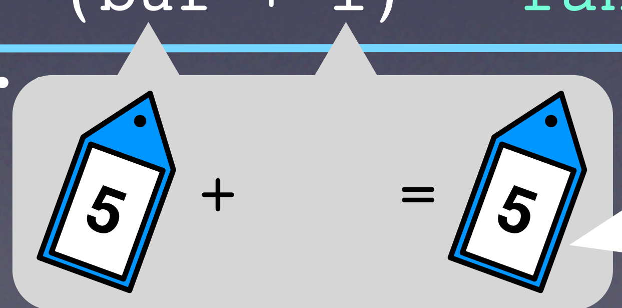
```
  void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}
```

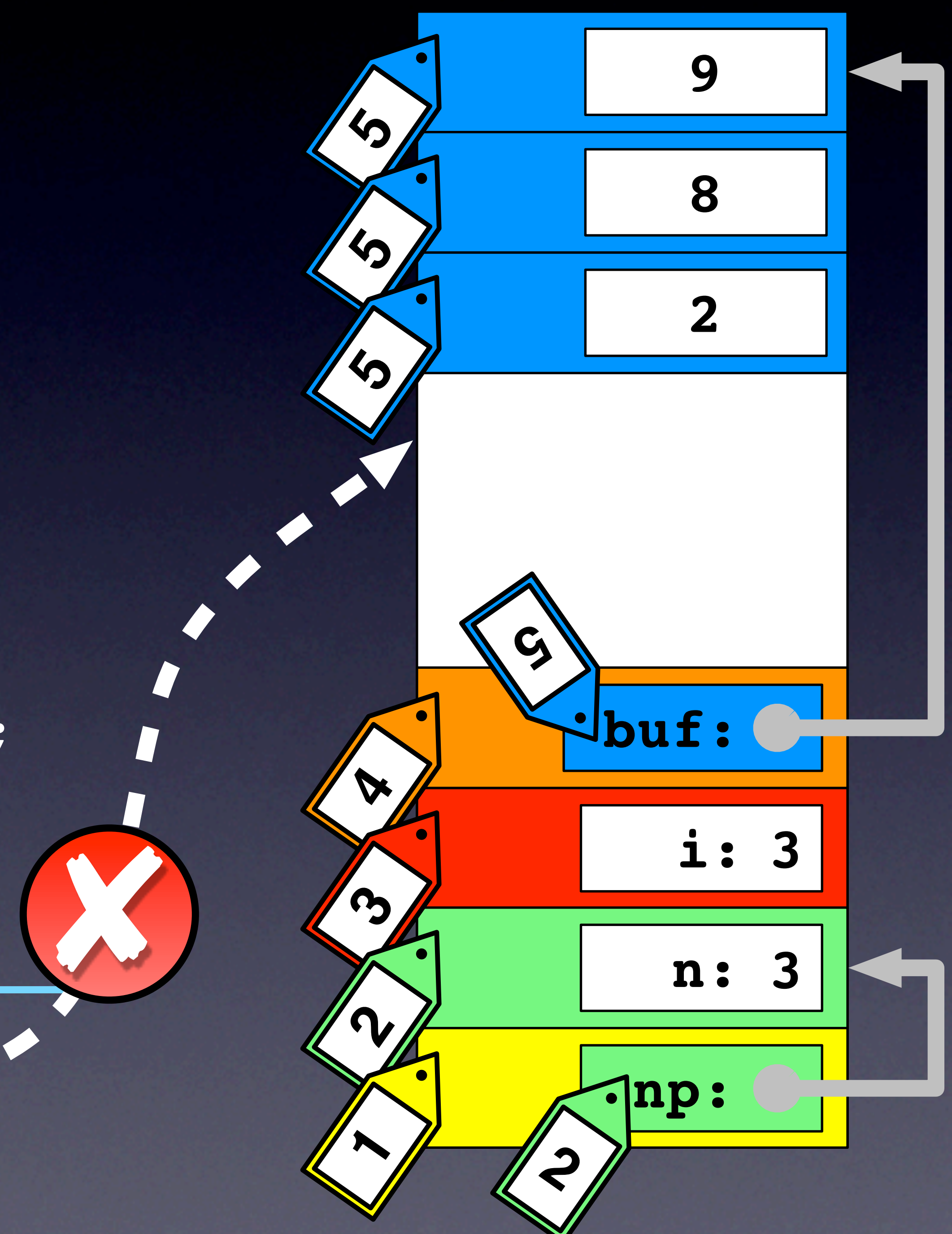


# Preventing IMAs

```
void main() {
int *np, n, i, *buf;

np = &n;

printf("Enter size: ");
scanf("%d", np);

buf = malloc(n * sizeof(int));

for(i = 0; i <= n; i++)
  *(buf + i) = rand()%10;
...
}
```



# Limiting the number of taint marks

An unlimited number of taint marks makes a hardware implementation infeasible

- increases the overhead (time and space)
- complicates the design

# Limiting the number of taint marks

An unlimited number of taint marks makes a hardware implementation infeasible

- increases the overhead (time and space)
- complicates the design

➡ Assign taint marks from a limited, reusable pool

# Effects on the approach

## ⚠ IMAs are detected probabilistically

With an random assignment of *n* taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

# Effects on the approach

## ⚠ IMAs are detected probabilistically

With an random assignment of *n* taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%

# Effects on the approach

## IMAs are detected probabilistically

With an random assignment of *n* taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%

1. The technique can be tuned by increasing or decreasing the number of taint marks

# Effects on the approach

## ⚠️ IMAs are detected probabilistically

With an random assignment of *n* taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%

1. The technique can be tuned by increasing or decreasing the number of taint marks

2. In practice the approach is successful with only a small number (2) of taint marks

# Empirical evaluation

RQ1: Is the efficiency of our approach sufficient for it to be applied to deployed software?

RQ2: What is the effectiveness of our technique when using limited number of taint marks?

# RQ1: experimental method

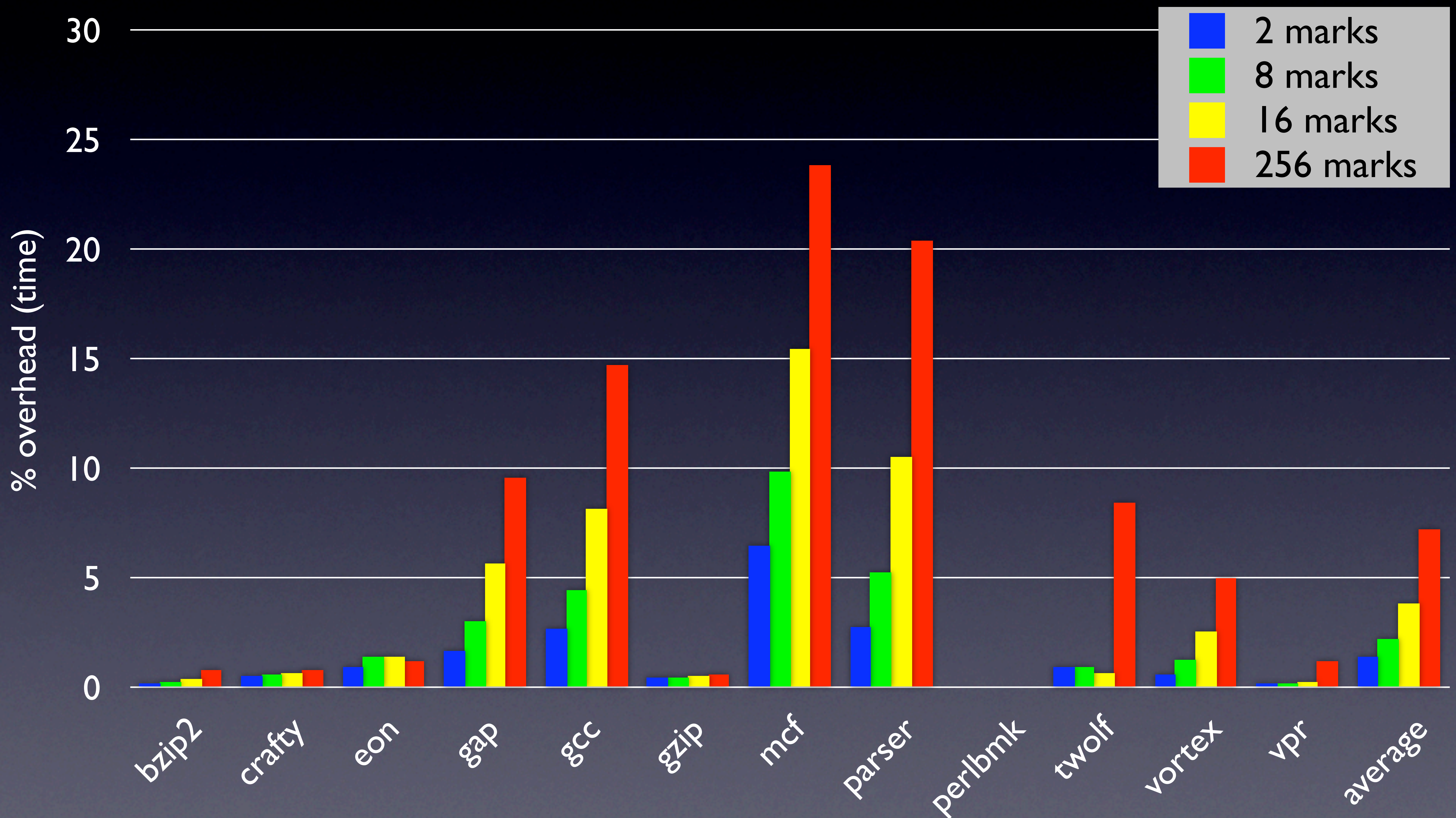
- Hardware implementation
  - Cycle accurate simuator (SESC)
  - Treat taint marks as first class citizens
- Subjects
  - SPEC CPU2000 benchmark (12 applications)
- Calculate the overhead imposed by our approach for each subject application

# RQ1: experimental method

- Hardware implementation
  - Cycle accurate simulator (SESC)
  - Treat taint marks as first class citizens
- Subjects
  - SPEC CPU2000 benchmark (12 applications)
- Calculate the overhead imposed by our approach for each subject application

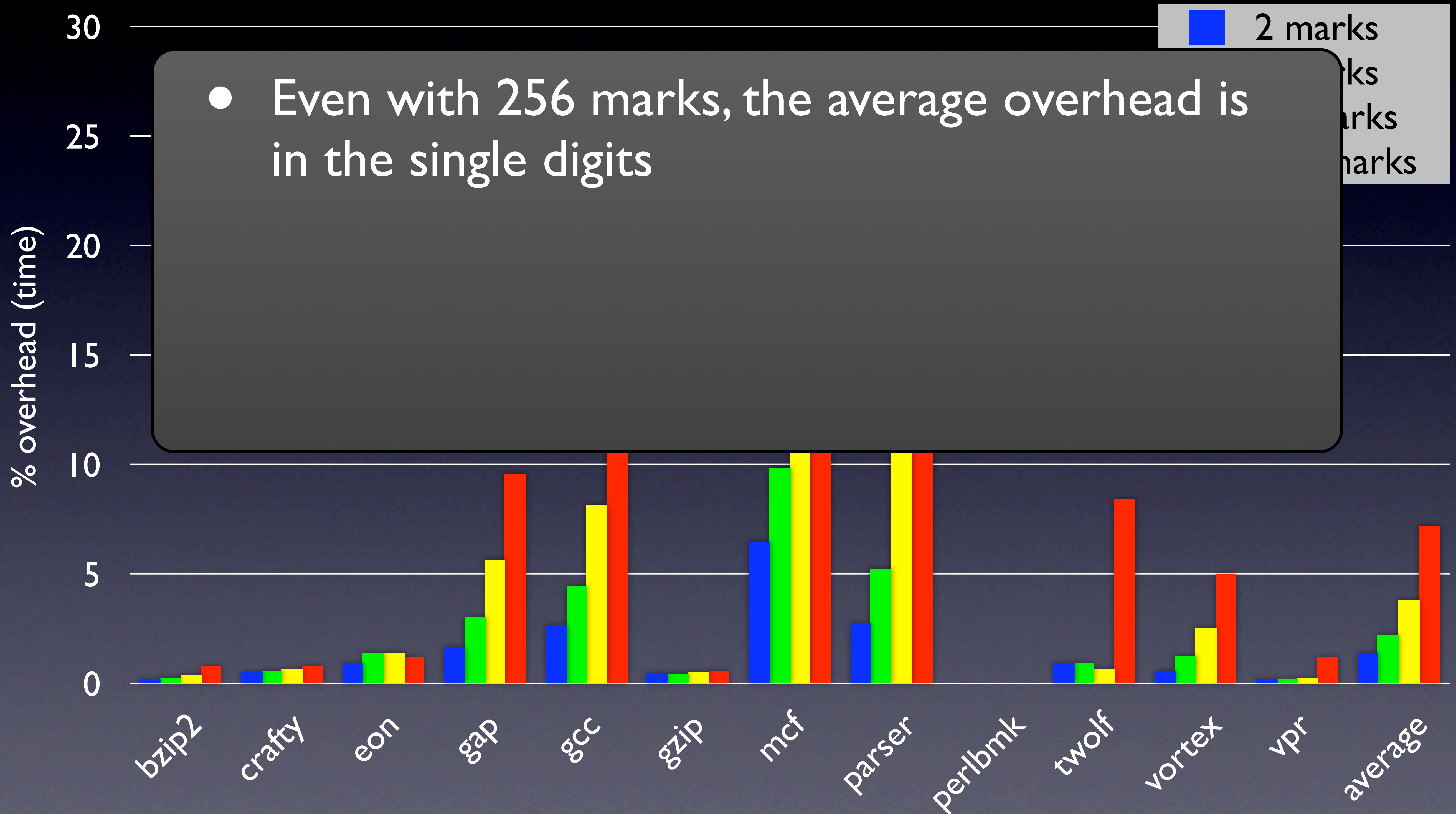
Current implementation assigns taint marks only to dynamically allocated memory, but propagation and checking are fully implemented

# RQ1: results



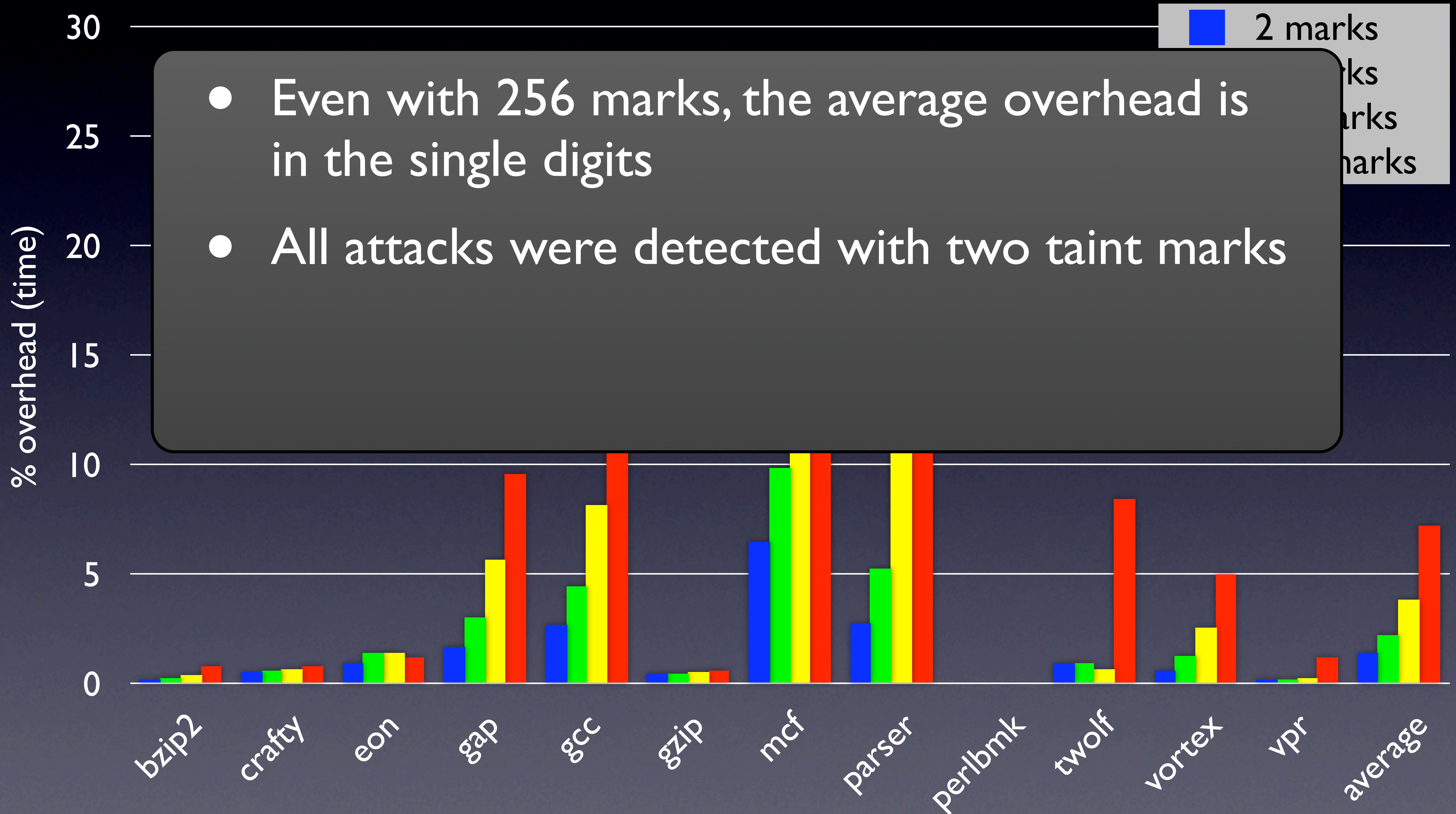
2 marks
8 marks
16 marks
256 marks
% overhead (time)
30
25
20
15
10
5
0
bzip2
crafty
eon
gap
gcc
gzip
mcf
parser
perlbmk
twolf
vortex
vpr
average

# RQ1: results




- Even with 256 marks, the average overhead is in the single digits

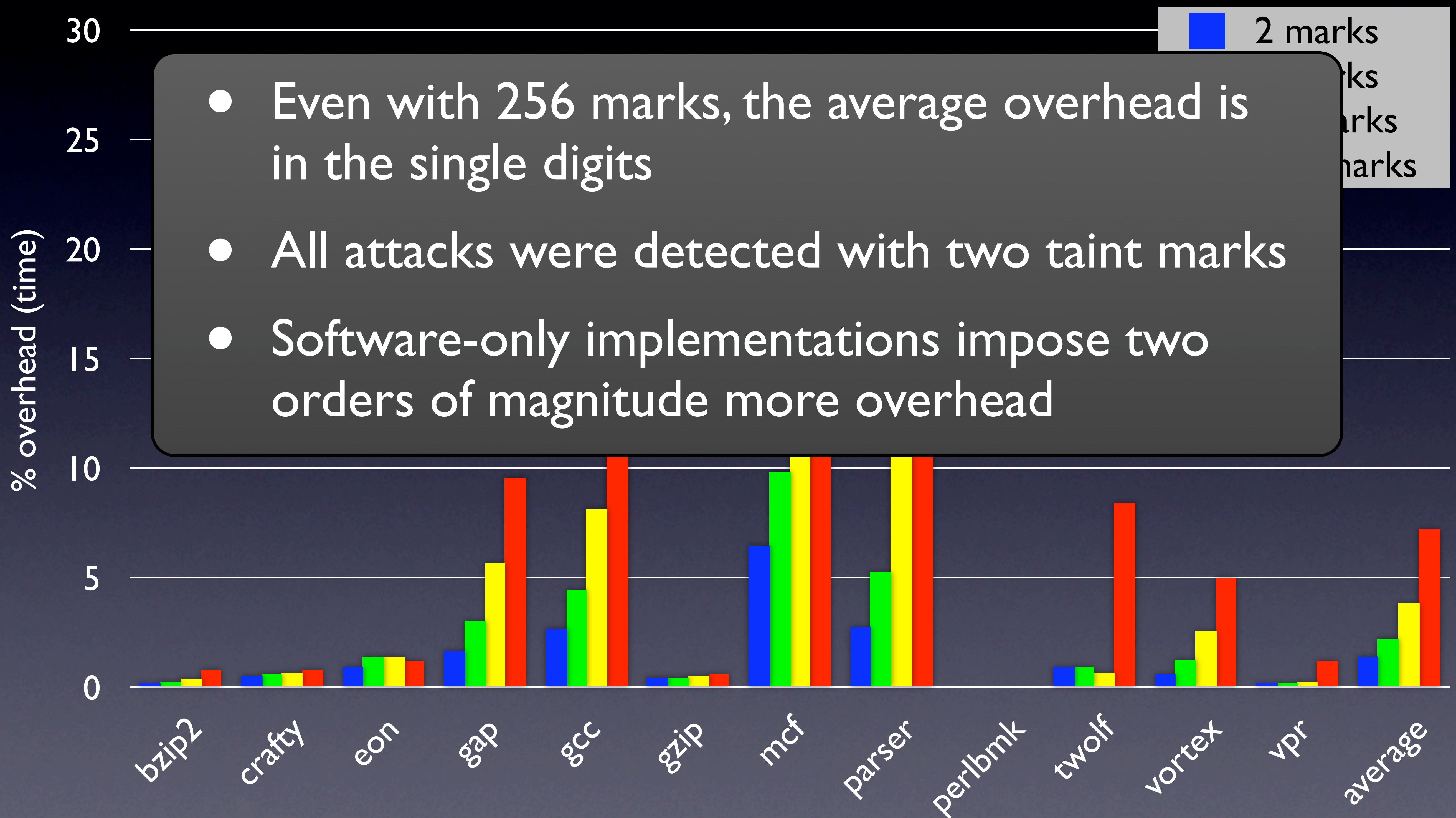
# RQ1: results




- Even with 256 marks, the average overhead is in the single digits
- All attacks were detected with two taint marks

# RQ1: results

- Even with 256 marks, the average overhead is in the single digits
- All attacks were detected with two taint marks
- Software-only implementations impose two orders of magnitude more overhead



# RQ2: experimental method

- Software implementation
  - Binary instrumenter (Pin)
  - Use instrumentation to assign, propagate, and check taint marks
- Subjects
  - SPEC CPU2000 benchmark (12 applications)
  - 5 applications with 7 known IMAs
- Run both each applications protected by our software implementation and check that only the known illegal memory accesses are detected (5 times)

# RQ2: results

## Applications with known IMAs

| Application | IMA location | Type | Detected |
|---|---|---|---|
| bc-1.06 | more_arrays: 177 | buffer overflow | ✔ (5/5) |
| bc-1.06 | lookup: 577 | buffer overflow | ✔ (5/5) |
| gnupg-1.4.4 | parse_comment: 2095 | integer overflow | ✔ (5/5) |
| mutt-1.4.2.li | utf8_to_utf7: 199 | buffer overflow | ✔ (5/5) |
| php-5.2.0 | php_char_to_str_ex: 3152 | integer overflow | ✔ (5/5) |
| pine-4.44 | rfc882_cat: 260 | buffer overflow | ✔ (5/5) |
| squid-2.3 | ftpBuildTitleUrl: 1024 | buffer overflow | ✔ (5/5) |

# RQ2: results

## Applications with known IMAs

| Application | IMA location | Type | Detected |
|---|---|---|---|
| bc-1.06 | more_arrays: 177 | buffer overflow | ✔ (5/5) |
| bc-1.06 | lookup: 577 | buffer overflow | ✔ (5/5) |
| gnupg-1.4.4 | parse_comment: 2095 | integer overflow | ✔ (5/5) |
| mutt-1.4.2.li | utf8_to_utf7: 199 | buffer overflow | ✔ (5/5) |
| php-5.2.0 | php_char_to_str_ex: 3152 | integer overflow | ✔ (5/5) |
| pine-4.44 | rfc882_cat: 260 | buffer overflow | ✔ (5/5) |
| squid-2.3 | ftpBuildTitleUrl: 1024 | buffer overflow | ✔ (5/5) |

All attacks were detected with two taint marks

# RQ2: results

## Applications with known IMAs

| Application | IMA location | Type | Detected |
|---|---|---|---|
| bc-1.06 | more_arrays: 177 | buffer overflow | ✔ (5/5) |
| bc-1.06 | lookup: 577 | buffer overflow | ✔ (5/5) |
| gnupg-1.4.4 | parse_comment: 2095 | integer overflow | ✔ (5/5) |
| mutt-1.4.2.li | utf8_to_utf7: 199 | buffer overflow | ✔ (5/5) |
| php-5.2.0 | php_char_to_str_ex: 3152 | integer overflow | ✔ (5/5) |
| pine-4.44 | rfc882_cat: 260 | buffer overflow | ✔ (5/5) |
| squid-2.3 | ftpBuildTitleUrl: 1024 | buffer overflow | ✔ (5/5) |

All attacks were detected with two taint marks

## SPEC Benchmarks ("IMA free")

| Application | IMA location | Type | Detected |
|---|---|---|---|
| vortex | SendMsg: 279 | null-pointer dereference | ✔ (5/5) |

# Future work

- Complete implementation that handles static memory
- Additional experiments with a wider range of IMAs
- Further optimization of the hardware implementation

# Conclusions

- Definition of an approach for preventing illegal memory accesses in deployed software
  - uses dynamic taint analysis to protect memory
  - uses probabilistic detection to achieve acceptable overhead
- Empirical evaluation showing that the approach
  - is effective at detecting IMA in real applications
  - can be implemented efficiently in hardware