



## Bell: Bit-Encoding Online Memory Leak Detection

Michael D. Bond    Kathryn S. McKinley

University of Texas at Austin



## Bugs in Deployed Software

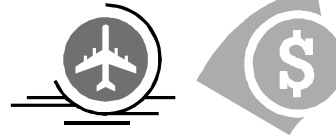
- Humans rely on software for critical tasks
  - Bugs are costly & risky
- Software more complex
  - More bugs & harder to fix





## Bugs in Deployed Software

- Humans rely on software for critical tasks
  - Bugs are costly & risky
- Software more complex
  - More bugs & harder to fix
- Bugs are a problem in deployed software
  - In-house testing incomplete
- Performance is critical
  - Focus on space overhead



## Why do bug tools want so much space?

- Store lots of info about the program
- Correlate program locations (sites) & data
  - Ex: DirectedGraph.java:309
  - Tag each object with one or more sites

Alloc site	Last-use site	Header	Field 1	Field 2	Field 3
------------	---------------	--------	---------	---------	---------



## Why do bug tools want so much space?

- Store lots of info about the program
- Correlate program locations (sites) & data
  - Ex: DirectedGraph.java:309
  - Tag each object with one or more sites

Alloc site	Last-use site	Header	Field 1	Field 2	Field 3
------------	---------------	--------	---------	---------	---------

- Bug detection applications
  - AVIO tracks last-use site of each object
  - Leak detection reports leaking objects' sites  
[JRockit, .NET Memory Profiler, Purify, SWAT, Valgrind]
- High space overhead if many small objects



## Why do bug tools want so much space?

- Store lots of info about the program
- Correlate program locations (sites) & data
  - Ex: DirectedGraph.java:309
  - Tag each object with one or more sites

Alloc site	Last-use site	Header	Field 1	Field 2	Field 3
------------	---------------	--------	---------	---------	---------

- Bug detection applications
  - AVIO tracks last-use site of each object
  - Leak detection reports leaking ob  
[JRockit, .NET Memory Profiler, Purify, SWAT, Valgrind]
- High space overhead if many small objects

SWAT: 75% space  
overhead on twolf



## How many bits do we need?

Site	Header	Field 1	Field 2	Field 3
------	--------	---------	---------	---------



## How many bits do we need?

Site	Header	Field 1	Field 2	Field 3
------	--------	---------	---------	---------

- 32 bits



## How many bits do we need?



- 32 bits
- 20 bits if # sites < 1,000,000
- 10 bits for common case (hot sites)



## How many bits do we need?



- 32 bits
- 20 bits if # sites < 1,000,000
- 10 bits for common case (hot sites)
- 1 bit?



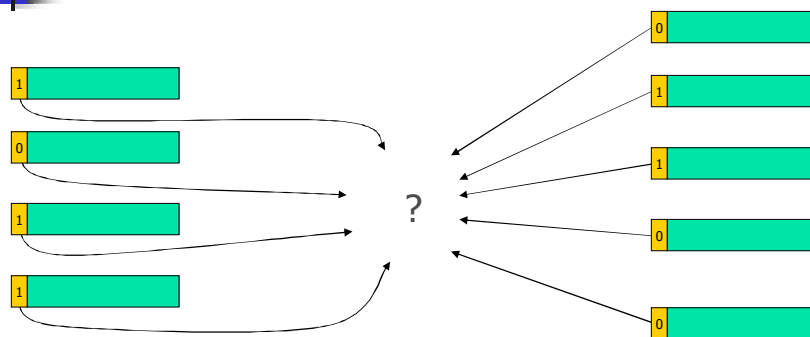
## How many bits do we need?



- 32 bits
- 20 bits if # sites < 1,000,000
- 10 bits for common case (hot sites)
- 1 bit?
  - One bit loses info about site



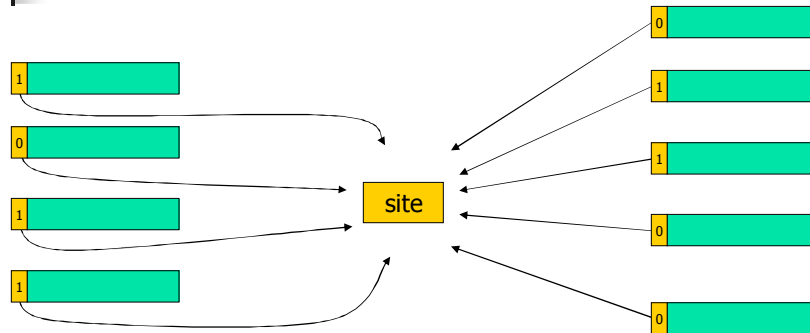
## How many bits do we need?



- 1 bit?
  - One bit loses info about site
  - But with many objects...



## Bell: Bit-Encoding Leak Location



- Stores per-object sites in single bit
- Reconstructs sites by looking at multiple objects' bits



## Outline

- Introduction
- Memory leaks
- Bell encoding and decoding
- Leak detection using Bell
- Related work



## Memory Leaks

- Memory bugs
  - Memory corruption: dangling refs, buffer overflows
  - Memory leaks
    - Lost objects: unreachable but not freed
    - Useless objects: reachable but not used again



## Memory Leaks

- Memory bugs
  - Memory corruption: dangling refs, buffer overflows
  - Memory leaks
    - Lost objects: unreachable but not freed
    - Useless objects: reachable but not used again

### Managed Languages

- 80% of new software in Java or C# by 2010  
[Gartner]
- Type safety & GC eliminate many bugs





## Memory Leaks

- Memory bugs
  - ~~Memory corruption: dangling refs, buffer overflows~~
  - Memory leaks
    - ~~Lost objects: unreachable but not freed~~
    - Useless objects: reachable but not used again

### Managed Languages

- 80% of new software in Java or C# by 2010 [Gartner]
- Type safety & GC eliminate many bugs



## Memory Leaks

Leaks occur in practice in managed languages

[Cork, JRockit, JProbe, LeakBot, .NET Memory Profiler]

- Memory leaks
  - ~~Lost objects: unreachable but not freed~~
  - Useless objects: reachable but not used again

### Managed Languages

- 80% of new software in Java or C# by 2010 [Gartner]
- Type safety & GC eliminate many bugs



## Outline

---

- Introduction
- Memory leaks
- Bell encoding and decoding
- Leak detection using Bell
- Related work



## Bell's Encoding Function

---

$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$





## Bell's Encoding Function

$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$



Color indicates site  
(ex: allocation site)



## Bell's Encoding Function

$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$

$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$





## Bell's Encoding Function

$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$

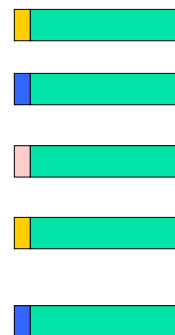
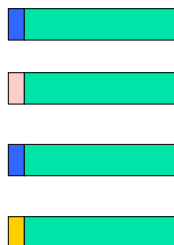
$$f(\text{site}, \text{object}) = 0 \text{ or } 1$$

may  
match

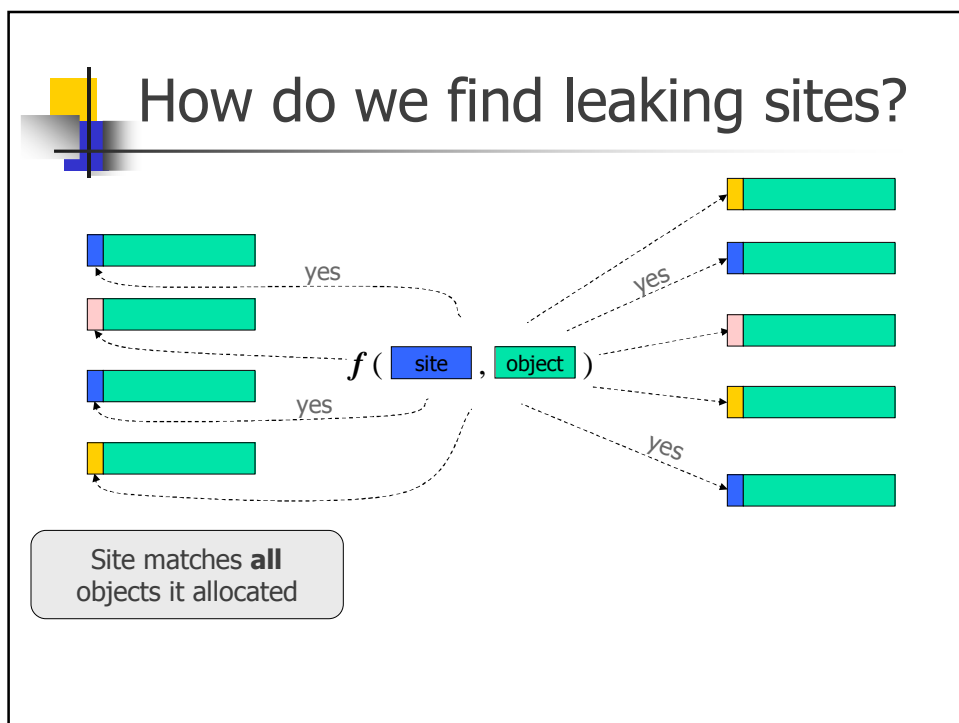
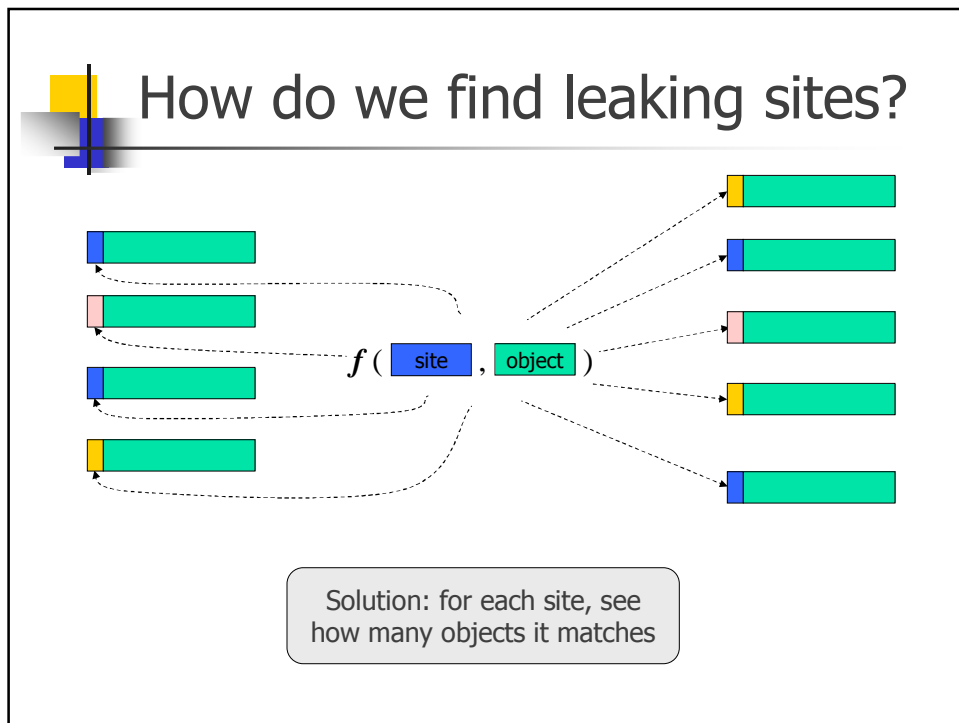
Probability of match is  $\frac{1}{2} \rightarrow$  unbiased function

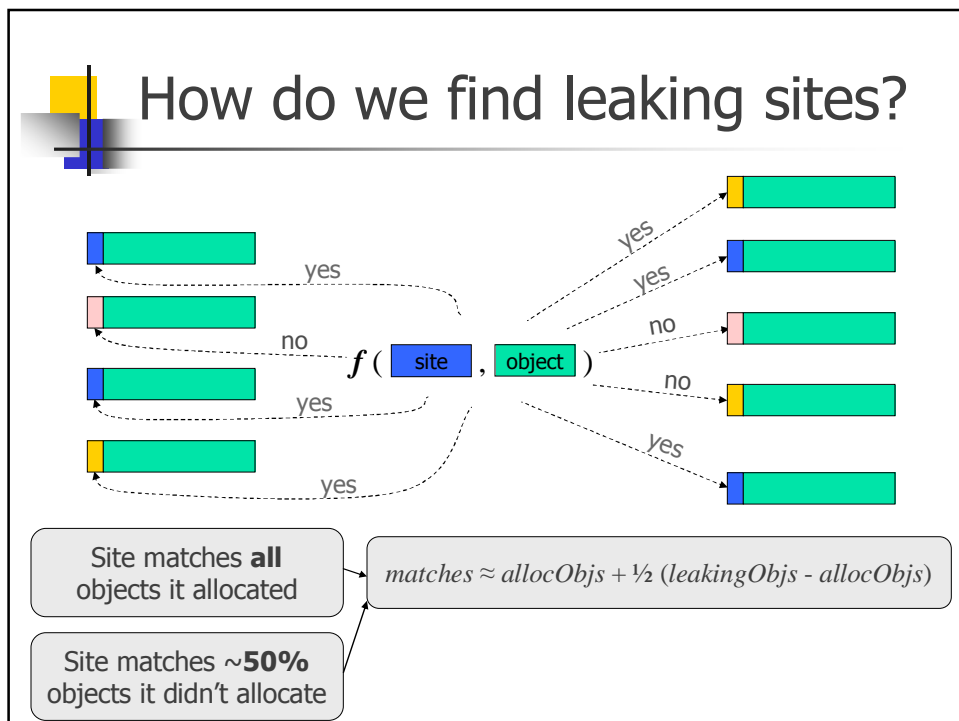
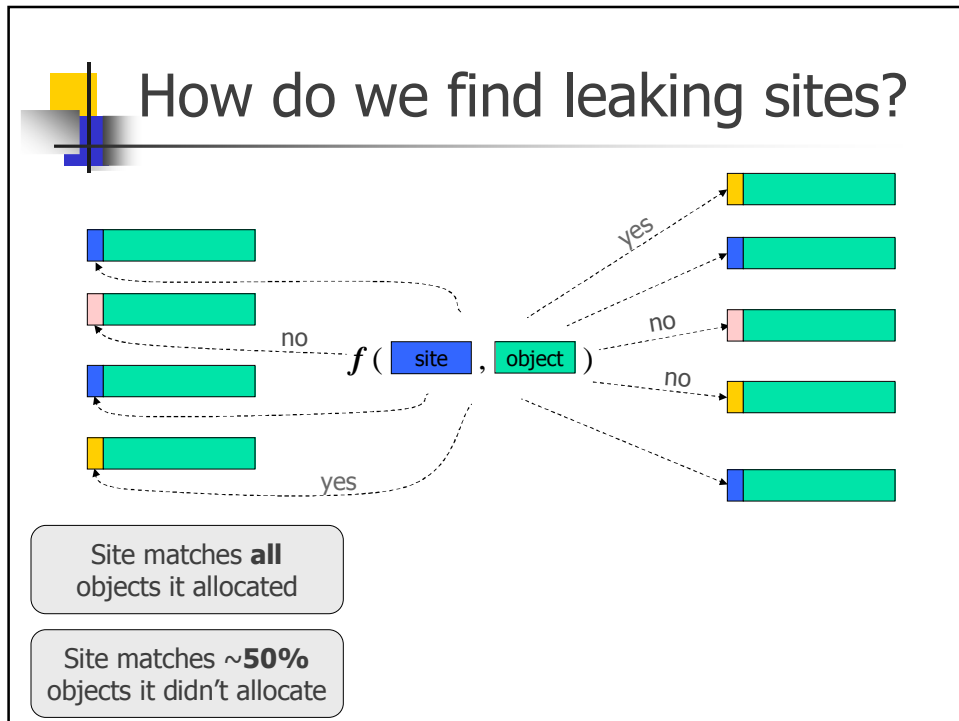


## How do we find leaking sites?



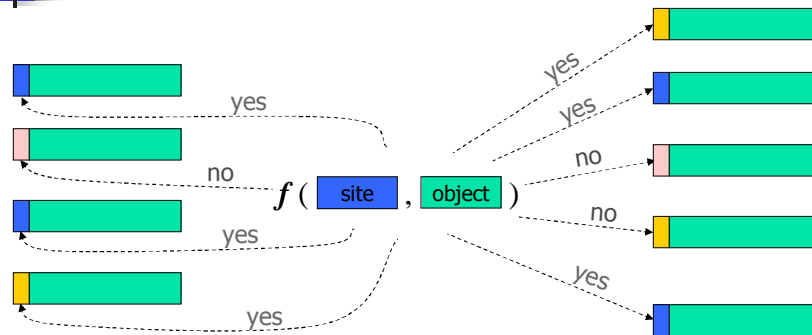
Problem: leaking objects with  
unknown allocation sites







## How do we find leaking sites?

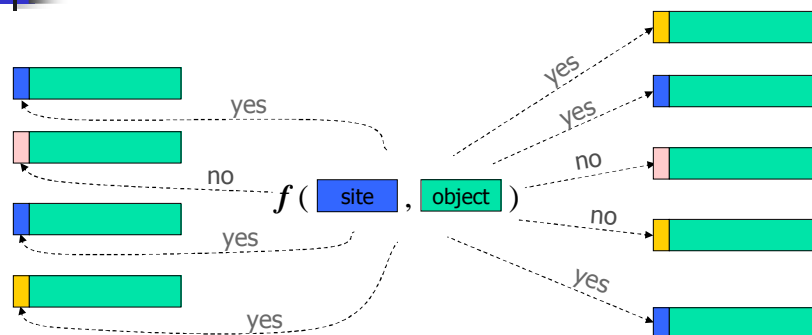


$$\text{matches} \approx \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs})$$

$$\text{allocObjs} \approx 2 \times \text{matches} - \text{leakingObjs}$$



## How do we find leaking sites?

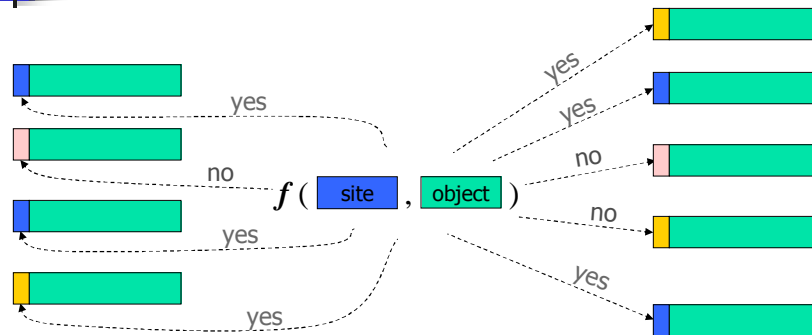


$$\text{matches} \approx \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs})$$

$$\text{allocObjs} \approx 2 \times \text{matches} - \text{leakingObjs}$$



## How do we find leaking sites?

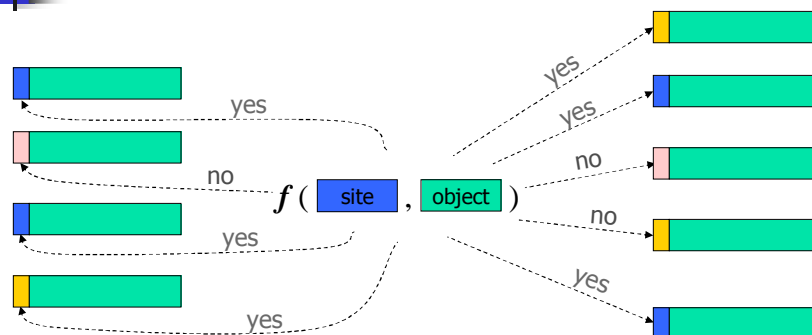


$$\text{matches} \approx \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs})$$

$$\text{allocObjs} \approx 2 \times 6 - 9$$



## How do we find leaking sites?



$$\text{matches} \approx \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs})$$

$$\text{allocObjs} \approx 3$$





## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    if  $f(\text{site}, \text{object}) = \text{object's site bit}$ 
      matches  $\leftarrow \text{matches} + 1$ 
  allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```



## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    if  $f(\text{site}, \text{object}) = \text{object's site bit}$ 
      matches  $\leftarrow \text{matches} + 1$ 
  allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```



## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    if  $f(\text{site}, \text{object}) = \text{object's site bit}$ 
      matches  $\leftarrow \text{matches} + 1$ 
    allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```



## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    if  $f(\text{site}, \text{object}) = \text{object's site bit}$ 
      matches  $\leftarrow \text{matches} + 1$ 
    allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```

Threshold avoids reporting sites that allocated no objects (false positives)



## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    if  $f(\text{site}, \text{object}) = 0$ 
      matches  $\leftarrow \text{matches} + 1$ 
  allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```

Threshold avoids reporting sites that allocated no objects (false positives)

Decoding misses sites that allocated few objects (false negatives)



## Bell Decoding

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    where site is possible
    if  $f(\text{site}, \text{object}) = 0$ 
      matches  $\leftarrow \text{matches} + 1$ 
  allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```

Dynamic type check narrows possible objects



## Outline

---

- Introduction
- Memory leaks
- Bell encoding and decoding
- Leak detection using Bell
- Related work



## Leak Detection using Bell

---

- Sleigh
  - Bell encodes allocation and last-use sites
  - Stale objects → potential leaks [SWAT]
  - Periodic decoding of highly stale objects

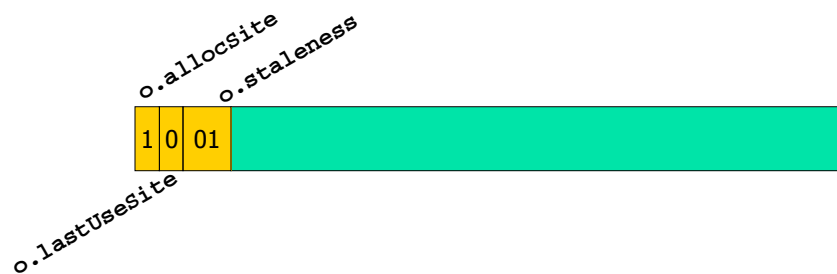


## Leak Detection using Bell

- Sleigh
  - Bell encodes allocation and last-use sites
  - Stale objects → potential leaks [SWAT]
  - Periodic decoding of highly stale objects
- Implementation in Jikes RVM
- Find leaks in Eclipse and SPEC JBB2000

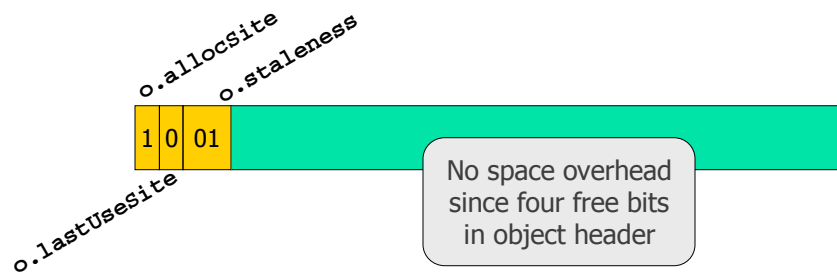


## Leak Detection using Bell



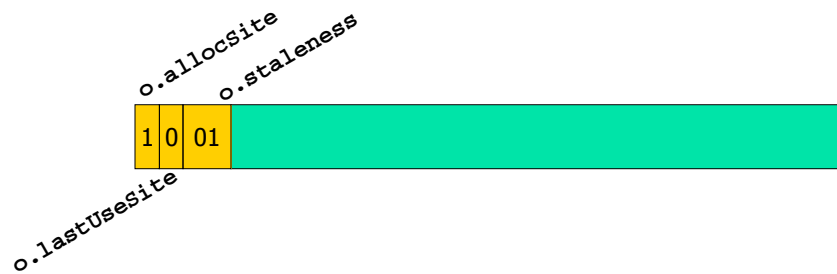


## Leak Detection using Bell



## Maintaining Sleight's Bits

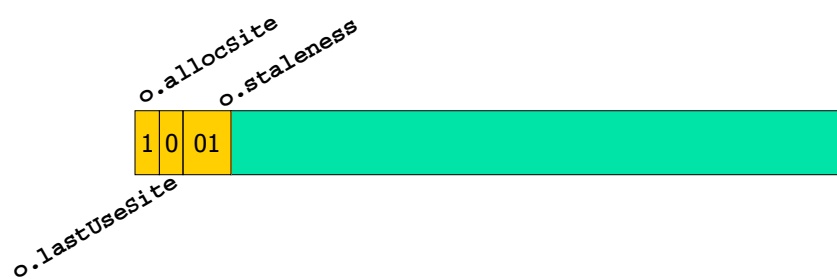
```
// Object allocation:  
s1: o = new MyObject();
```





## Maintaining Sleight's Bits

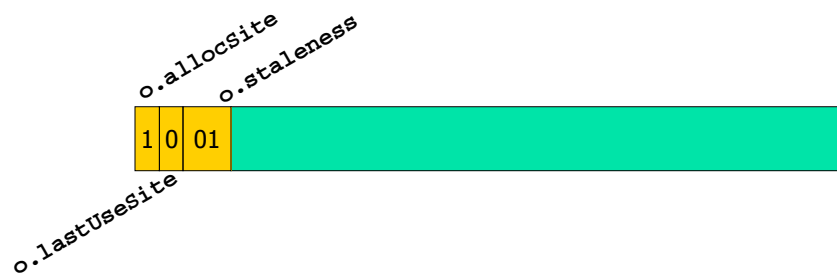
```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```



## Maintaining Sleight's Bits

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;
```

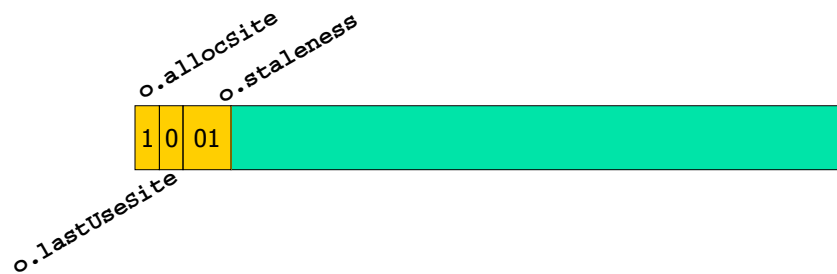




## Maintaining Sleight's Bits

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```



## The Encoding Function

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```

$$f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} \times \text{object})$$





## The Encoding Function

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```

$$f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} \times \text{object} \times \text{object})$$



## Object Movement Restrictions

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```

$$f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} \times \text{object} \times \text{object})$$

- Objects may not move
- (Mostly) non-moving collector
  - Mark-sweep
  - Generational mark-sweep
- C and C++ do not move objects



## Sleigh's Time Overhead

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```

$f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} \times \text{object} \times \text{object})$

DaCapo [Blackburn et al. '06]  
SPEC JBB2000  
SPEC JVM98



## Sleigh's Time Overhead

```
// Object allocation:  
s1: o = new MyObject();  
// Instrumentation:  
o.allocSite = f(s1, o);
```

```
// Object use:  
s2: tmp = o.f;  
// Instrumentation:  
o.lastUseSite = f(s2, o);  
o.staleness = 0;
```

$f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} \times \text{object} \times \text{object})$

DaCapo [Blackburn et al. '06]  
SPEC JBB2000  
SPEC JVM98

29% time overhead (11% with adaptive profiling)



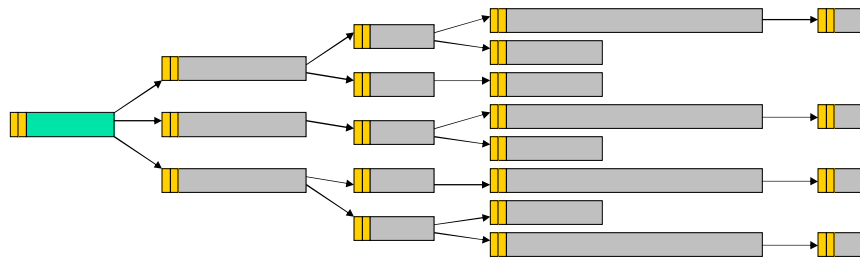
## Finding and Fixing Leaks

- Leaks in Eclipse and SPEC JBB2000



## Finding and Fixing Leaks

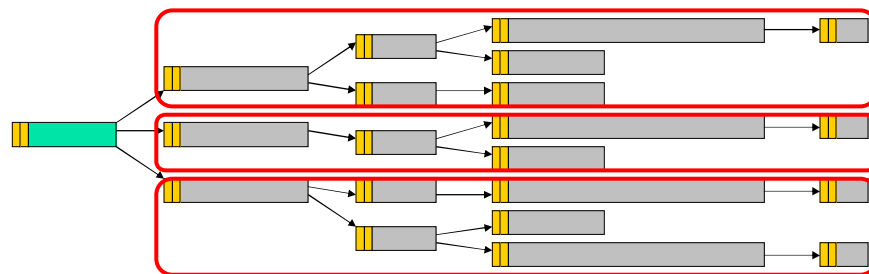
- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak





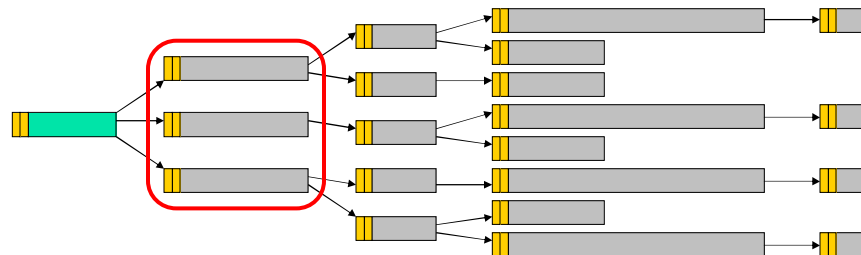
## Finding and Fixing Leaks

- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak



## Finding and Fixing Leaks

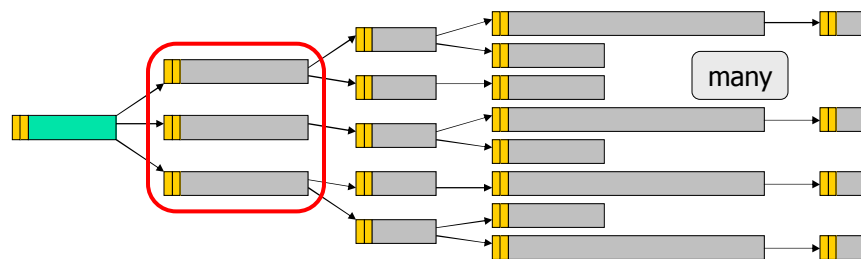
- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak
  - Most interesting: stale roots





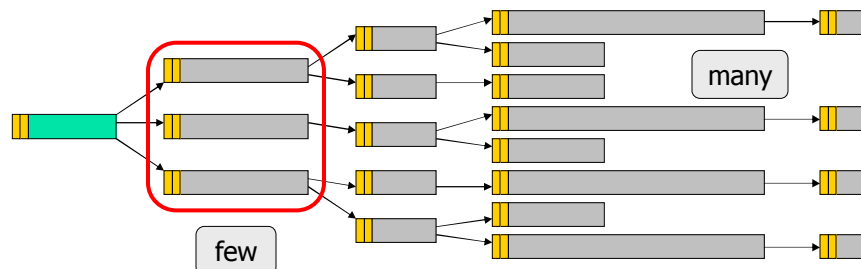
## Finding and Fixing Leaks

- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak
  - Most interesting: stale roots



## Finding and Fixing Leaks

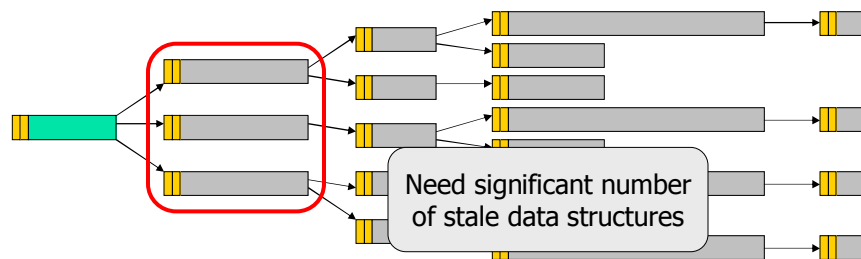
- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak
  - Most interesting: stale roots





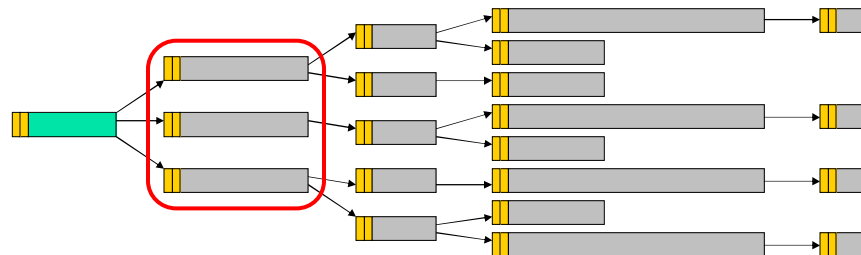
## Finding and Fixing Leaks

- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak
  - Most interesting: stale roots



## Finding and Fixing Leaks

- Leaks in Eclipse and SPEC JBB2000
  - Data structures leak
  - Most interesting: stale roots
  - Sleight's output directly useful for fixing leaks





## Bell Decoding Again

```
foreach possible site
  matches  $\leftarrow 0$ 
  foreach potentially leaking object
    where site is possible and
      object is root of stale data structure
    if  $f(\text{site}, \text{object}) = \text{object}$ 's
      matches  $\leftarrow \text{matches} + 1$ 
  allocObjs =  $2 \times \text{matches} - \text{leakingObjs}$ 
  if allocObjs > threshold(leakingObjs)
    print site is the site for allocObjs objects
```

Consider roots of stale data structures only

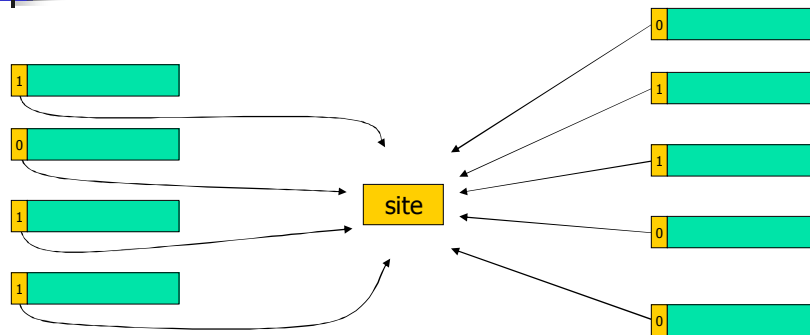


## Related Work

- Leak detectors store per-object sites  
[JRockit, .NET Memory Profiler, Purify, SWAT, Valgrind]
- Sampling [Jump et al. '04]
  - Trades accuracy for lower overhead (like Bell)
  - Adds some overhead; requires conditional instrumentation
  - No encoding or decoding
- Communication complexity & information theory



## Summary



- Bell encodes sites in a single bit and decodes sites using multiple objects' bits
- Leak detection with low overhead



## Thank You

- Questions?