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
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

- 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
- Bug detection applications
  - AVIO tracks last-use site of each object
  - Leak detection reports leaking objects
    [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?

- 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, J Rockit, J Probe, 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

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

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?

Solution: for each site, see how many objects it matches.
How do we find leaking sites?

Site matches all objects it allocated
How do we find leaking sites?

Site matches all objects it allocated

Site matches ~50% objects it didn’t allocate

\[f(\text{site}, \text{object})\]
How do we find leaking sites?

- Site matches **all** objects it allocated
- Site matches ~50% objects it didn’t allocate

\[ f(\text{site, object}) \]

- \( \text{matches} \sim \frac{1}{2} (\text{leakingObjs} - \text{allocObjs}) + \text{allocObjs} \)
How do we find leaking sites?

$$f(\text{site}, \text{object})$$

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

$\text{allocObjs} \sim 2 \times \text{matches} - \text{leakingObjs}$
How do we find leaking sites?

\[ f(\text{site, object}) \]

\[ \text{matches} \sim \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs}) \]

\[ \text{allocObjs} \sim 2 \times \ 6 \ - \text{leakingObjs} \]
How do we find leaking sites?

\[ f(\text{site}, \text{object}) = \begin{cases} 
\text{yes} & \text{on yes} \\
\text{no} & \text{on no} 
\end{cases} \]

\[ \text{matches} \sim \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs}) \]

\[ \text{allocObjs} \sim 2 \times 6 - 9 \]
How do we find leaking sites?

\[
f(\text{site}, \text{object})
\]

\[
\text{matches} \sim \text{allocObjs} + \frac{1}{2} (\text{leakingObjs} - \text{allocObjs})
\]

\[
\text{allocObjs} \sim 3
\]
Bell Decoding

```
foreach possible
matches ← 0

foreach potentially leaking
if \( f(\text{site}, \text{object}) = \text{object} \)’s site bit
matches ← matches + 1

allocObjs = 2 \times matches - \text{leakingObjs}

if allocObjs > \text{threshold(leakingObjs)}
print \text{site} is the site for allocObjs objects
```
Bell Decoding

```plaintext
foreach possible site
    matches ← 0
    foreach potentially leaking object
        if f(site, object) = object’s site bit
            matches ← matches + 1
    allocObjs = 2 × matches − leakingObjs
    if allocObjs > threshold(leakingObjs)
        print site is the site for allocObjs objects
```
Bell Decoding

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

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

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

**foreach possible site**

\[ \text{matches} \leftarrow 0 \]

**foreach potentially leaking object**

\[ \text{if } f(\text{site}, \text{object}) = 0 \]

\[ \text{matches} \leftarrow \text{matches} + 1 \]

\[ \text{allocObjs} = 2 \times \text{matches} - \text{leakingObjs} \]

**if** \[ \text{allocObjs} > \text{threshold(}\text{leakingObjs}) \]

**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

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

No space overhead since four free bits in object header
Maintaining Sleigh’s Bits

```java
// Object allocation:
s1: o = new MyObject();
```
// Object allocation:
$s_1$: $o = \text{new MyObject}();$

// Instrumentation:
o.allocSite = f(s_1, o);
Maintaining Sleigh’s Bits

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

// Object use:
s₂: tmp = o.f;
Maintaining Sleigh’s Bits

// Object allocation:
s_1: o = new MyObject();
// Instrumentation:
o.allocSite = f(s_1, o);

// Object use:
s_2: tmp = o.f;
// Instrumentation:
o.lastUseSite = f(s_2, o);
o.staleness = 0;
The Encoding Function

\[
\begin{align*}
\text{// Object allocation:} \\
& s_1: \text{o} = \text{new MyObject}(); \\
& \quad \text{// Instrumentation:} \\
& \quad \text{o.allocSite} = f(s_1, \text{o}); \\
\text{// Object use:} \\
& s_2: \text{tmp} = \text{o.f}; \\
& \quad \text{// Instrumentation:} \\
& \quad \text{o.lastUseSite} = f(s_2, \text{o}); \\
& \quad \text{o.staleness} = 0;
\end{align*}
\]

\[
f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} ? \text{object})
\]
// Object allocation:
s_1: o = new MyObject();
// Instrumentation:
o.allocSite = f(s_1, o);

// Object use:
s_2: tmp = o.f;
// Instrumentation:
o.lastUseSite = f(s_2, o);
o.staleness = 0;

\[ f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site}, ? \text{object}, ? \text{object}) \]
Object Movement Restrictions

// Object allocation:
s_1: o = new MyObject();
   // Instrumentation:
o.allocSite = f(s_1, o);

// Object use:
s_2: tmp = o.f;
   // Instrumentation:
o.lastUseSite = f(s_2, o);
o.staleness = 0;

\[ f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} ? \text{object} ? \text{object}) \]

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

\[ f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site} ? \text{object} ? \text{object}) \]

DaCapo [Blackburn et al. ’06]
SPEC JBB2000
SPEC JVM98
Sleigh’s Time Overhead

// Object allocation:
s_1: o = new MyObject();
// Instrumentation:
o.allocSite = f(s_1, o);

// Object use:
s_2: tmp = o.f;
// Instrumentation:
o.lastUseSite = f(s_2, o);
o.staleness = 0;

\[ f(\text{site}, \text{object}) := \text{bit}_{31}(\text{site}, ?, \text{object}, ?) \]

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

![Diagram showing few vs. many leaking structures]
Finding and Fixing Leaks

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

Need significant number of stale data structures
Finding and Fixing Leaks

Leaks in Eclipse and SPEC JBB2000
- Data structures leak
- Most interesting: stale roots
- Sleigh’s output directly useful for fixing leaks
Bell Decoding Again

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

Consider roots of stale data structures only
Related Work

- Leak detectors store per-object sites
  [J Rockit, .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
Bell encodes sites in a single bit and decodes sites using multiple objects’ bits

Leak detection with low overhead
Thank You

- Questions?