Exterminator: Automatically Correcting Memory Errors with High Probability

Gene Novark **Emery Berger**

University of Massachusetts Amherst

Ben Zorn

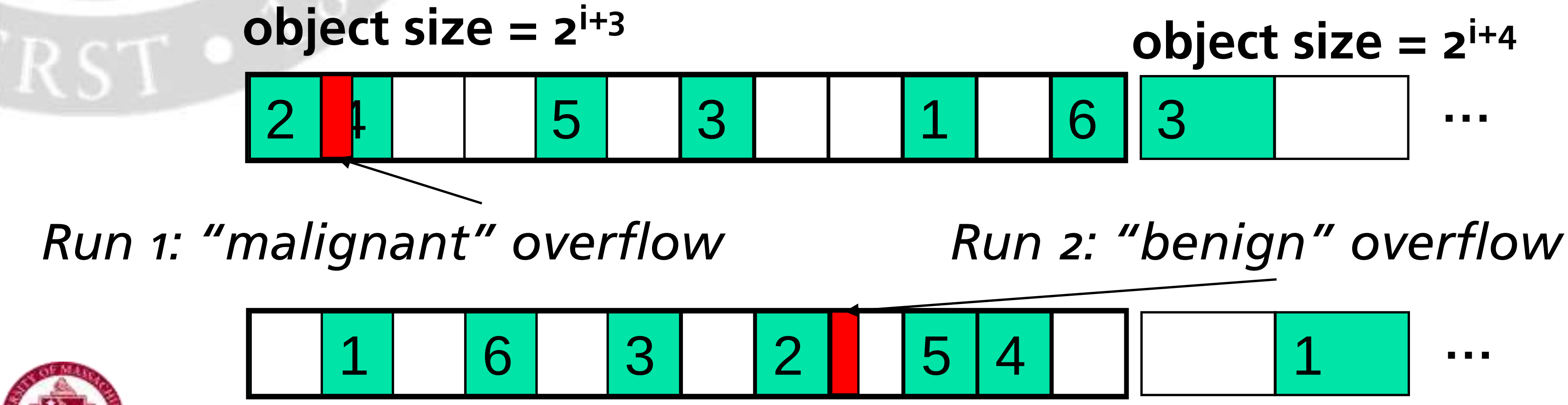
Microsoft Research

Problems with Unsafe Languages

- C, C++: pervasive apps, but **unsafe**
- Numerous opportunities for security vulnerabilities, **errors**
  - Double/Invalid **`free`**
  - Uninitialized reads
  - Dangling pointers
  - Buffer overflows (stack & **heap**)
- DieHard: eliminates some, **probabilistically** avoids others [PLDI 2006]
  - Exterminator: builds on DieHard

DieHard Overview *[PLDI 2006]*

- Use randomization & (optionally) replication to reduce risk of memory errors
  - Objects randomly spread across heap
- Different run = different heap
  - **Probabilistic memory safety**
    - Errors across heaps **independent**



object size = 2^{i+3}

object size = 2^{i+4}

Run 1: "malignant" overflow

Run 2: "benign" overflow

DieHard Limitations

- **DieHard:**
  - Fine for single error
    - But multiple errors eventually swamp probabilistic protection
    - Not great for large overflows
  - Tolerates errors
    - But doesn't find them
    - No information for programmer
- **Exterminator:**

  Automatically **isolate** and **fix** memory errors

Diagnosing Buffer Overflows

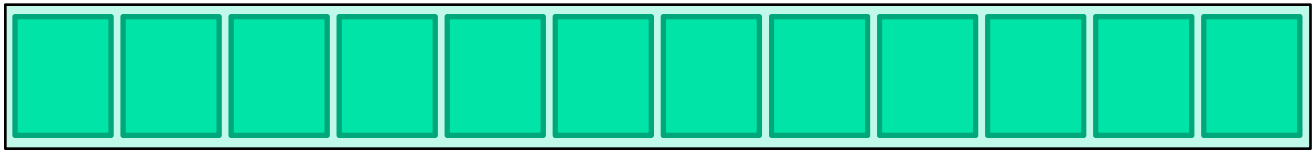
- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

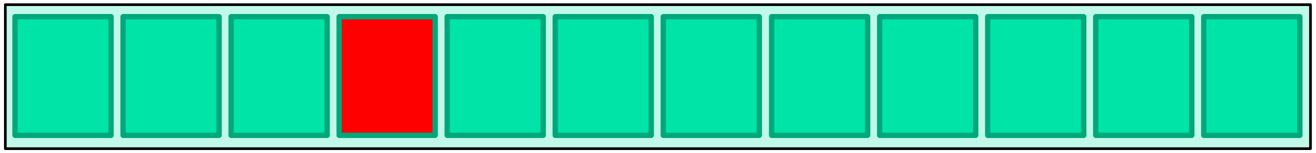
```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```



Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```



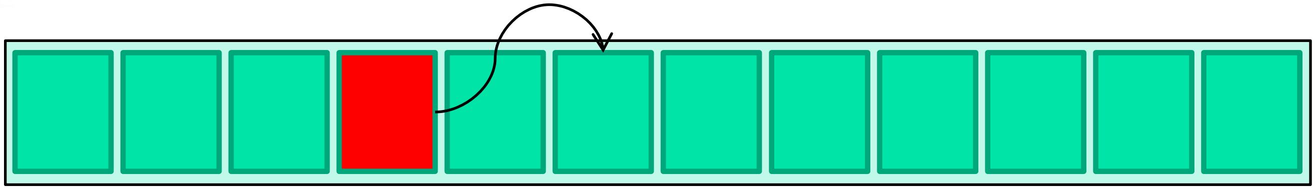
bad object
(too small)

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, "goodbye cruel world");
```

δ bytes past end



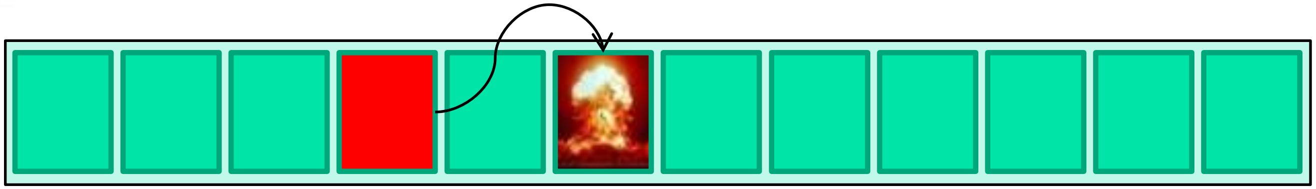
bad object
(too small)

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

δ bytes past end

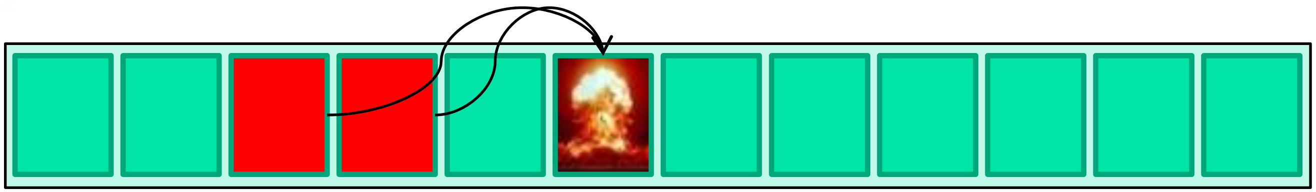


bad object (too small)

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```



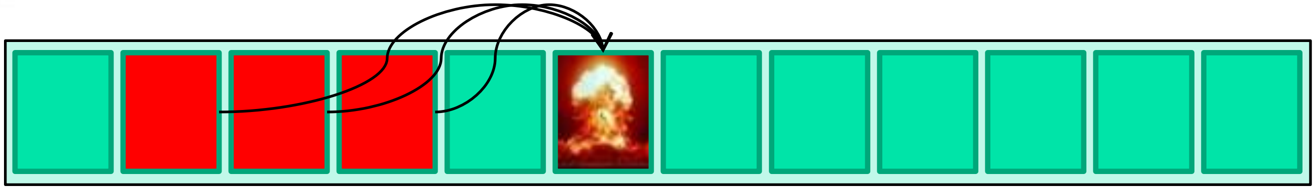
δ bytes past end

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

δ bytes past end

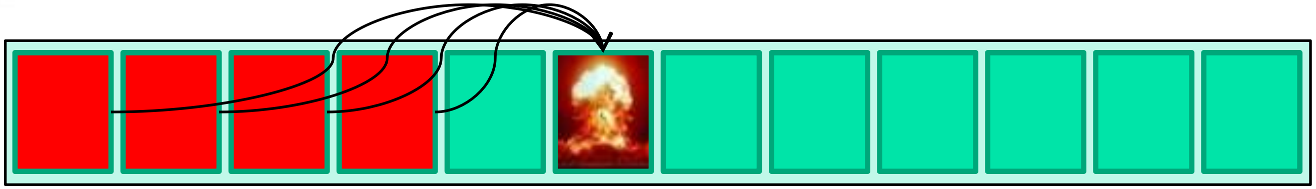


Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

δ bytes past end

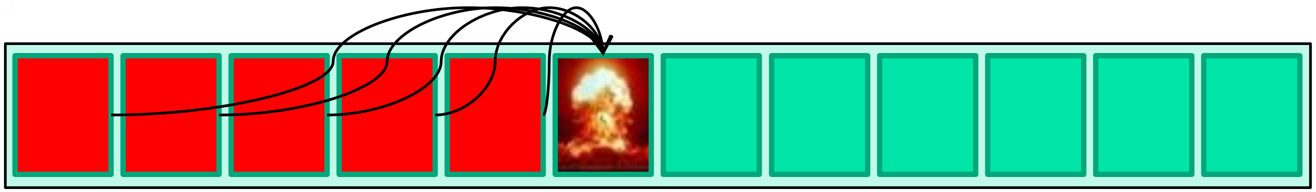


Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

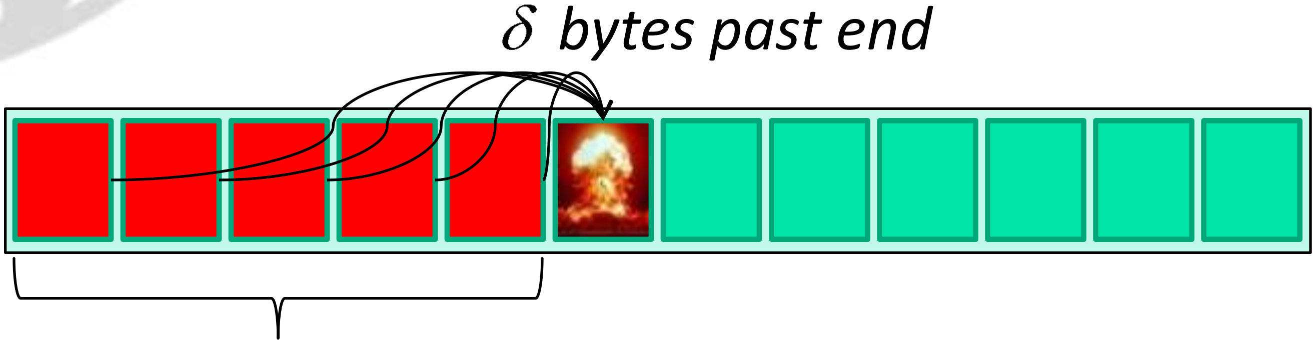
δ bytes past end



Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```



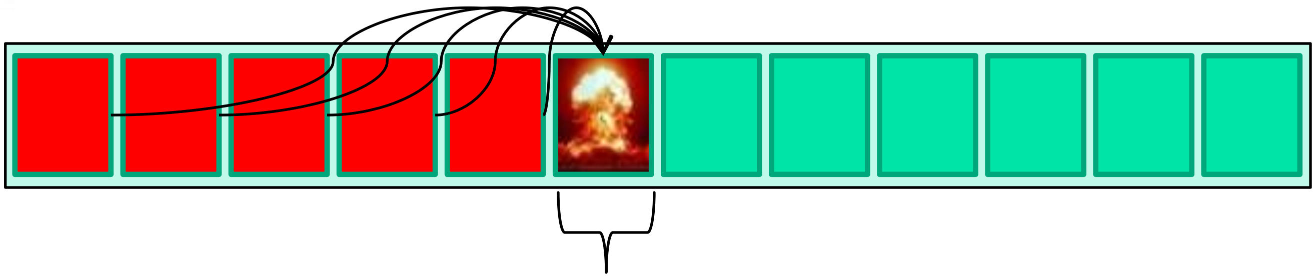
1. Heap provides no useful information

Diagnosing Buffer Overflows

- Canonical buffer overflow:
  - Allocate object – too small
  - Write past end ⇒ nukes object δ bytes forward
    - Not *necessarily* contiguous

```
char * str = new char[8];
strcpy (str, “goodbye cruel world”);
```

δ bytes past end

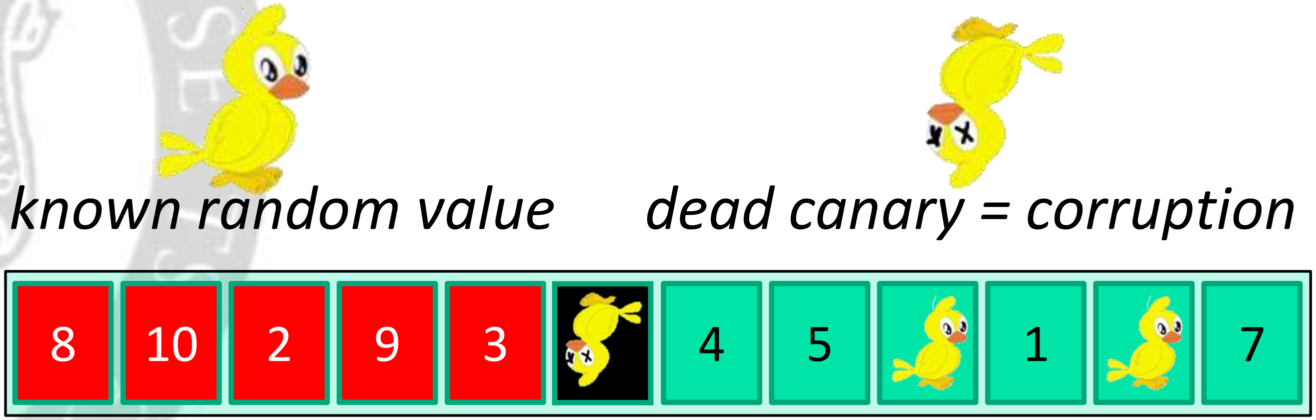


2. No way to detect corruption

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption

Red = possible bad object



known random value

dead canary = corruption

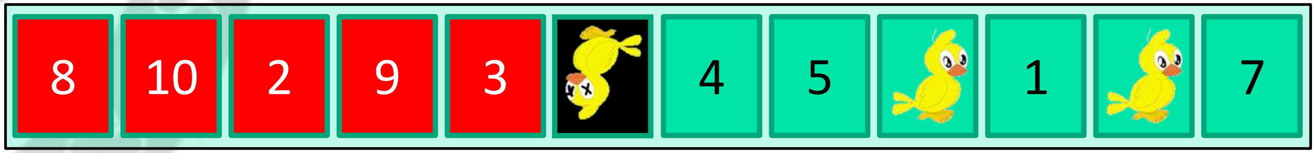
Green = *not* bad object

# = object id (allocation time)

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with "DieFast" allocator

Red = possible bad object



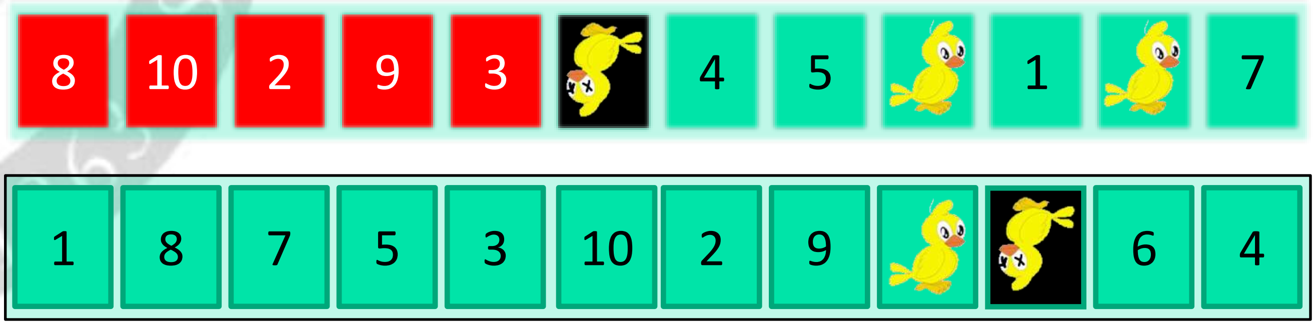
8 10 2 9 3 4 5 1 7

Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator

Red = possible bad object



8 10 2 9 3 4 5 1 7

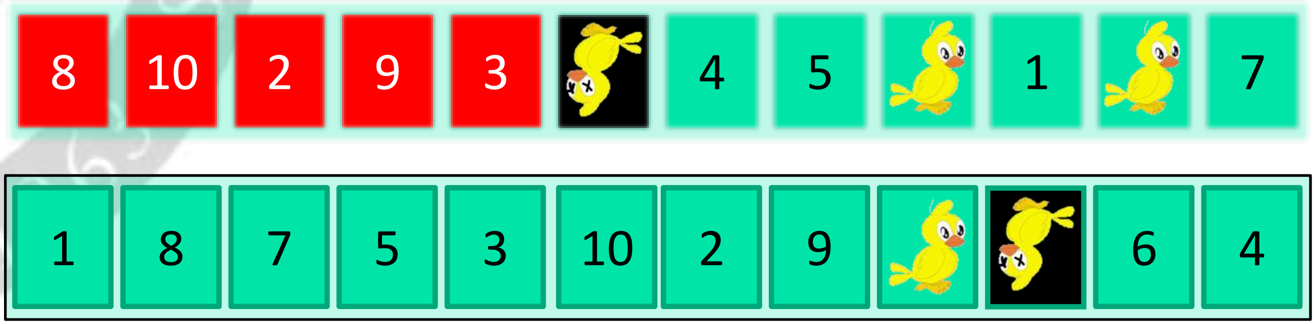
1 8 7 5 3 10 2 9 6 4

Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator
  - **Key insight:** Overflow **must be at same δ**

Red = possible bad object

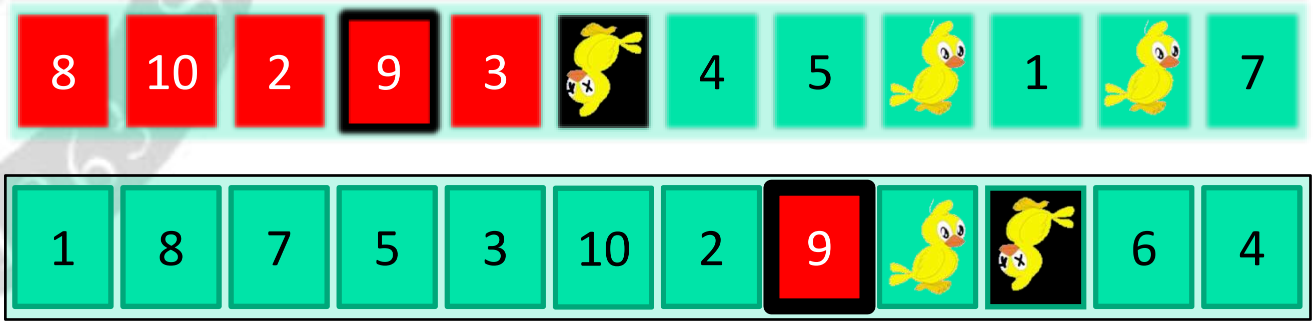


Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator
  - **Key insight:** Overflow **must be at same δ**

Red = possible bad object

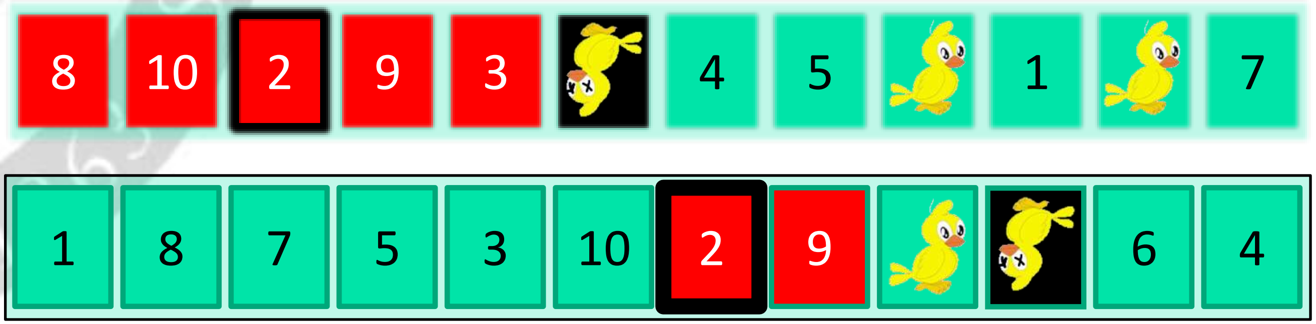


Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator
  - **Key insight:** Overflow **must be at same δ**

Red = possible bad object

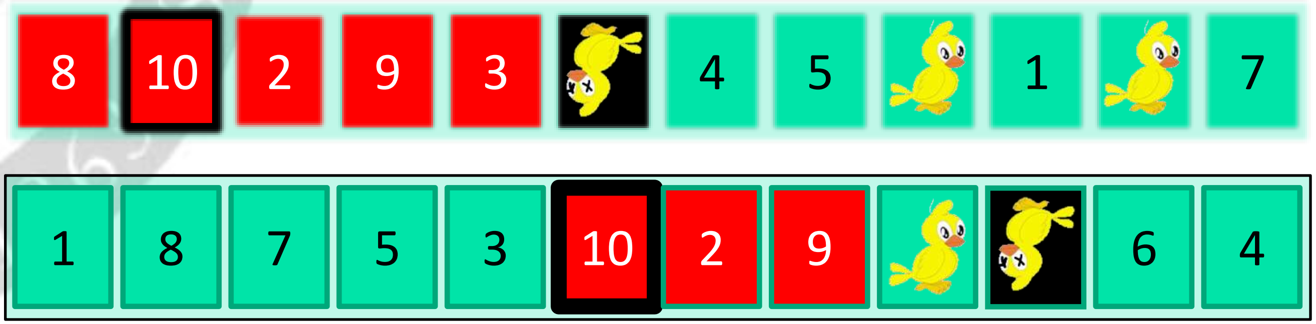


Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator
  - **Key insight:** Overflow **must be at same δ**

Red = possible bad object



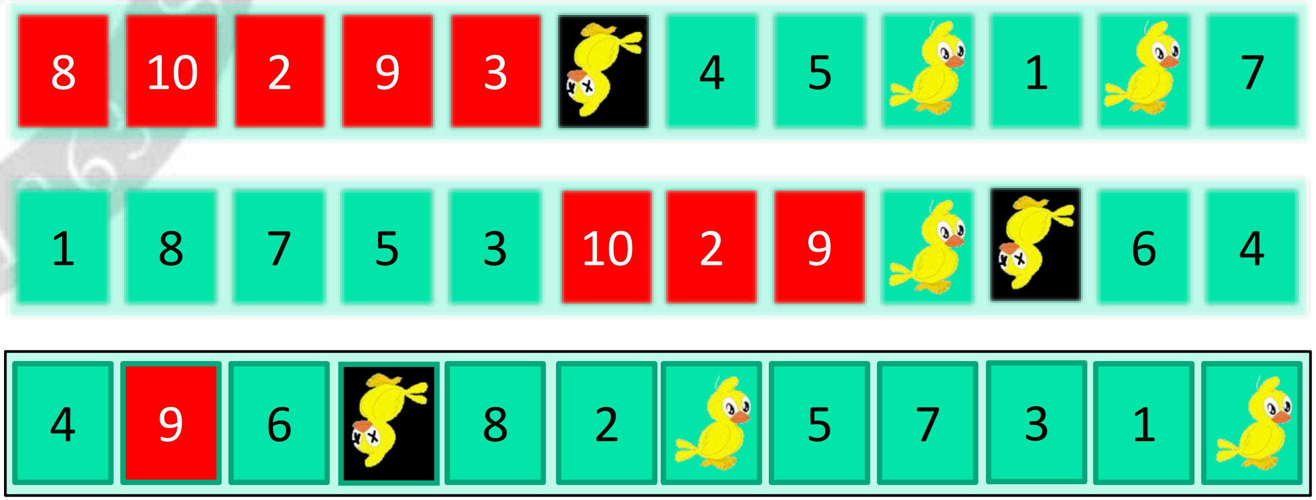
Green = *not* bad object

Isolating Buffer Overflows

- **Canaries** in freed space detect corruption
  - Run multiple times with “DieFast” allocator
  - **Key insight:** Overflow **must be at same** δ

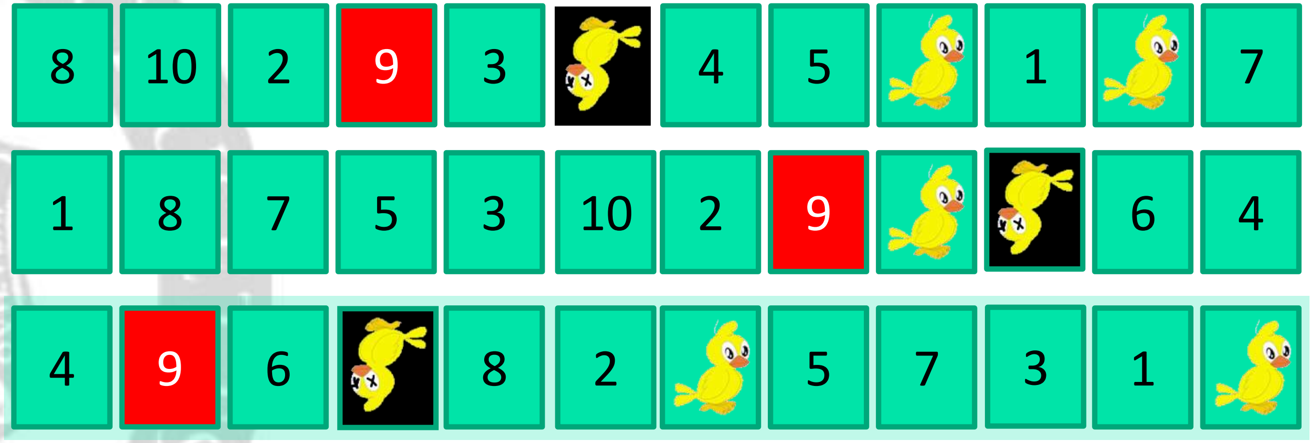
Red = possible bad object

Green = *not* bad object



⇒ object 9 overflowed, *with high probability*

Buffer Overflow Analysis



8	10	2	9	3		4	5		1		7
1	8	7	5	3	10	2	9			6	4
4	9	6		8	2		5	7	3	1	

$$\mathrm{E(false\ positives)} = \frac{1}{(H-1)^{k-2}}$$

H = # heap objects
K = # iterations

- Example: H = 1,000,000 objects
  3 iterations $\approx \frac{1}{1{,}000{,}000}$ false positives

- **Iterations exponentially increase precision**

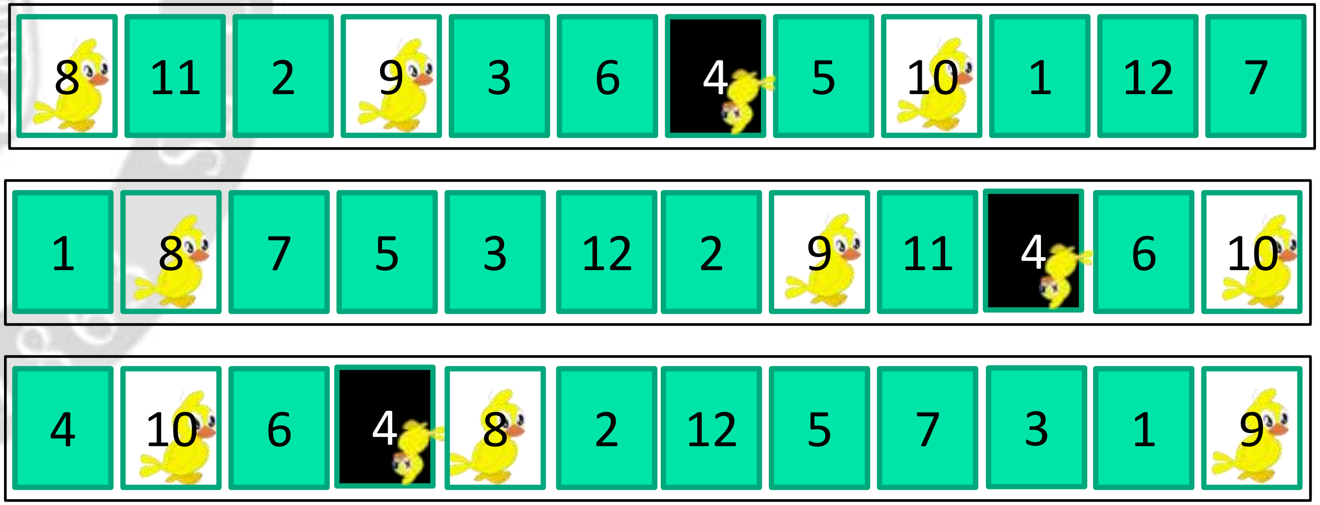
Isolating Dangling Pointers

- Dangling pointer error:
  - Live object freed too soon
  - Overwritten by some other object

```
int * v = new int[4];
…
delete [] v; // oops
…
char * str = new char[16];
strcpy (str, “die, pointer”);
v[3] = 12;
… use of v[0]
```

Isolating Dangling Pointers

- Unlike buffer overflow:
  - dangling pointer ⇒ **same** corruption in all



8	11	2	9	3	6	4	5	10	1	12	7
1	8	7	5	3	12	2	9	11	4	6	10
4	10	6	4	8	2	12	5	7	3	1	9

$$\mathrm{P}(\text{identical overflow}) \leq \left(\frac{1}{H-1}\right)^{k-1}$$

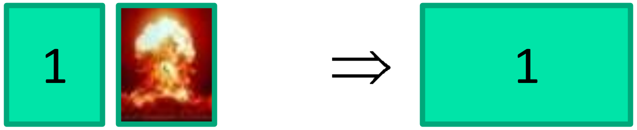
2

- k = 3 ⇒ *false negatives* ≈ $\frac{1}{1{,}000{,}000}$

Correcting Allocator

- Generate **runtime patches** to **correct** errors
    - Track object call sites in allocator
- *Prevent overflows*: **pad** overflowed objects

```
malloc(8) ⇒ malloc(8 + δ)
```



- *Prevent dangling pointers*: **defer** frees

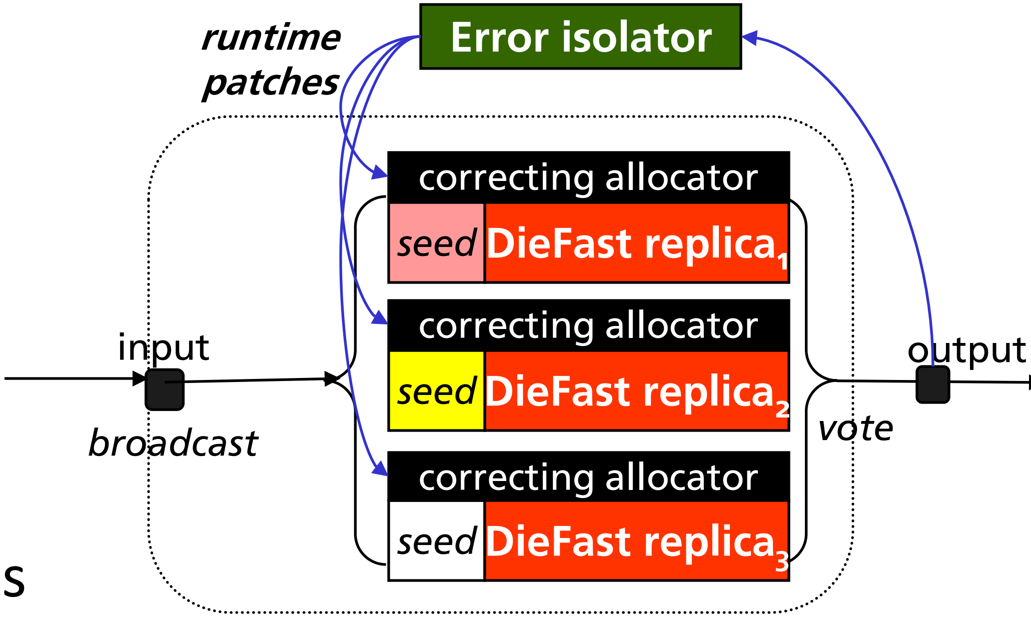
```
free(ptr) ⇒ delay δ mallocs;
            free(ptr)
```

Exterminator Architecture

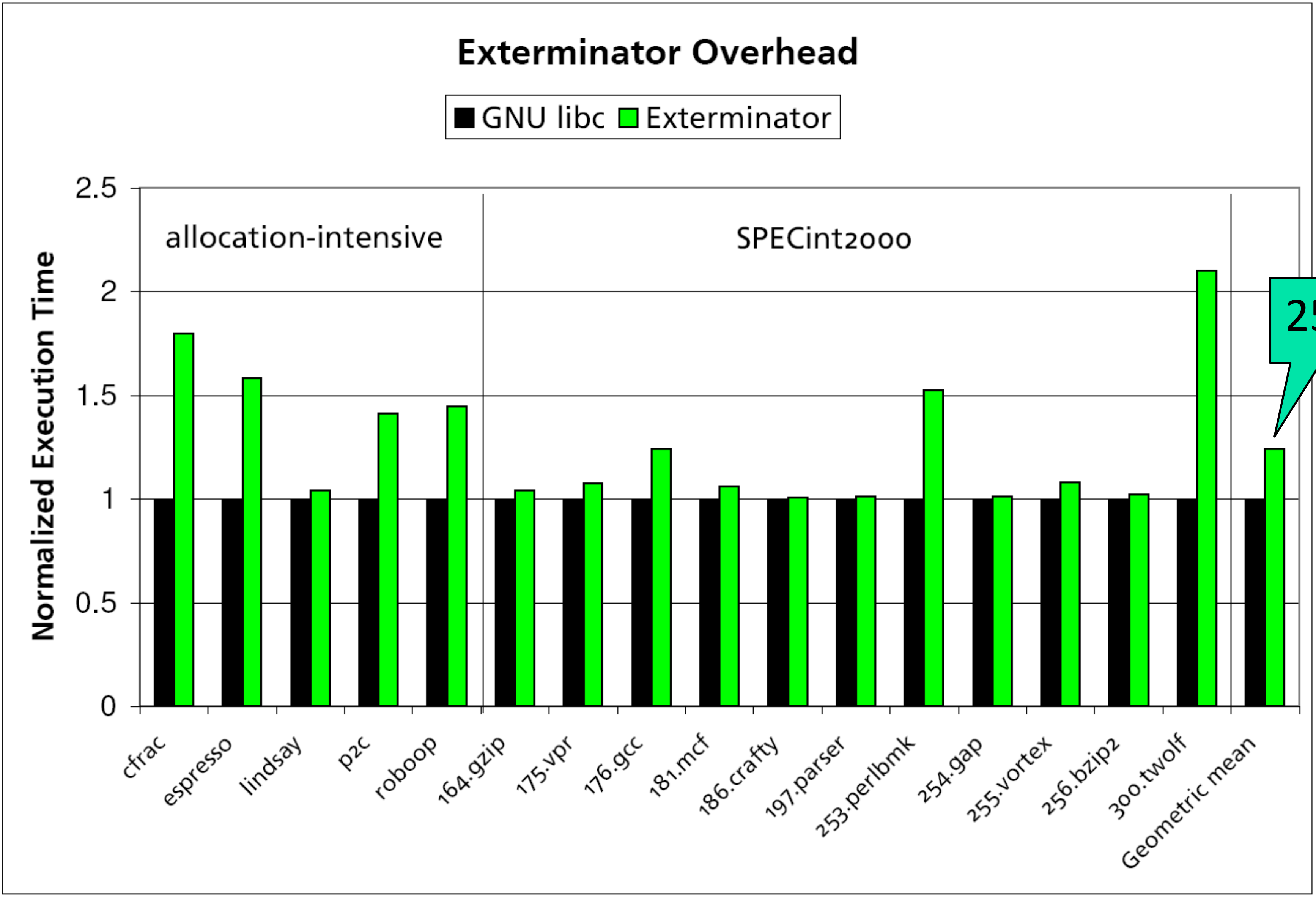
- Three main pieces:
  - DieHard-based **allocator** (DieFast)
    - Reveals bugs
  - **Error isolator**
    - Finds bugs across multiple heaps w.h.p.
  - **Correcting allocator**
    - Fixes bugs
- Multiple modes suitable for testing (debugging) or deployment

Exterminator Modes

- **Iterative**
  - Run multiple times
  - Same inputs
  - Debugging
- **Replicated**
  - Run simultaneously
  - Deployable w/limitations
  - Can fix errors on-the-fly
- **Cumulative**
  - Different inputs, nondeterminism
  - Deployable; see paper for details



Exterminator Runtime Overhead



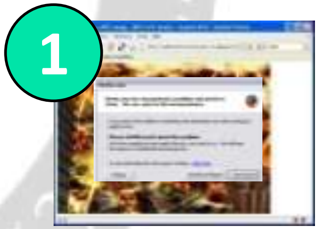
25%

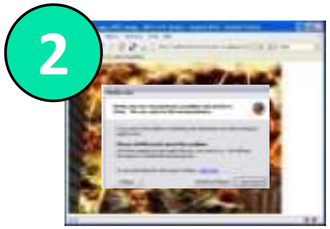
Empirical Results: Real Faults

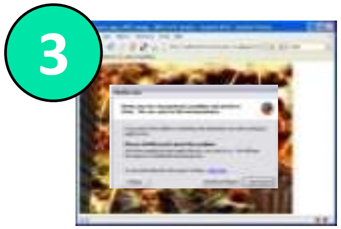
- Squid heap overflow
  - Crashes glibc 2.8.0 and BDW collector
  - 3 iterations to fix ⇒ 6 byte pad
    - Prevents overflow for all subsequent executions

Empirical Results: Real Faults

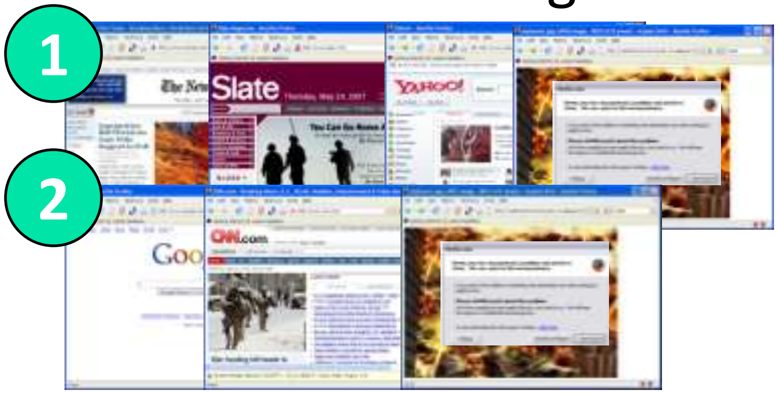
- Mozilla 1.7.3 buffer overflow
  - **Debug scenario:**
    - repeated load of PoC: 23 runs to fix overflow







  - **Deployed scenario**:
    - different browsing sessions: 34 runs to fix

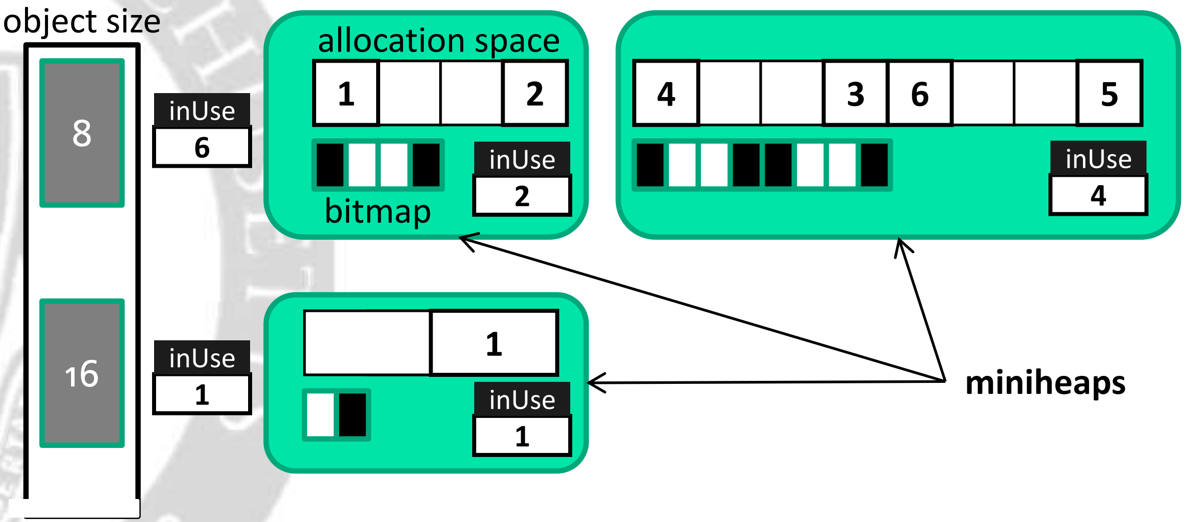


Exterminator Conclusion

- Exterminator: **automatic error correction** w.h.p.
  - **Randomization** ⇒ bugs have *different* effects
  - **Statistical analysis** combines information from multiple runs to isolate error
  - **Correcting allocator** eliminates bugs at runtime

http://www.cs.umass.edu/~gnovark/

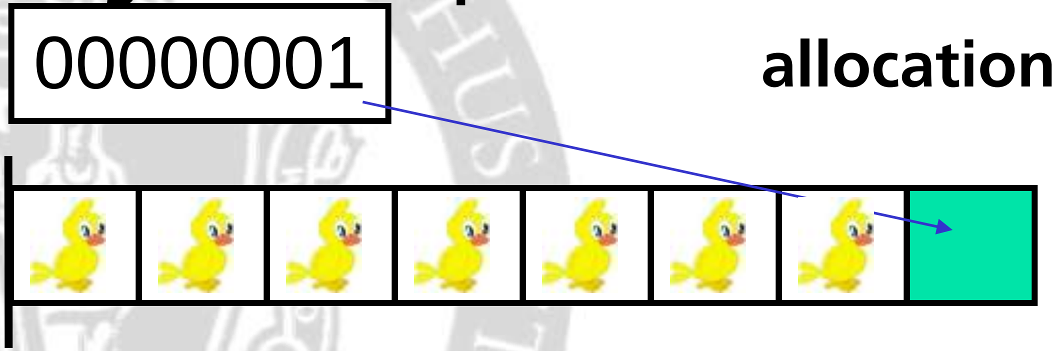
DieHard, heap layout



- Bitmap-based, **segregated** size classes
  - Bit represents one **object** of given size
    - i.e., one bit = 2^{i+3} bytes, etc.
- `malloc()`: randomly probe bitmap for free space
- `free()`: just reset bit

Exterminator Extensions

single miniheap



00000001

allocation bitmap

heap

DieHard

Exterminator

		2		1			3	object id (serial number)
		A_4		A_8			A_3	alloc site
		D_9		D_6				dealloc site
		3		2				dealloc time