Programming Languages for Building Trustworthy Systems

Ben Zorn

Microsoft Research

Which Programming Language To Use?

- Safe versus unsafe, difficult choice?
  - **Safe** – Java, C#, Modula-3, …
  - **Unsafe** – C, C++, assembler, …
- But choice is really more complex
  - How much of a Java app is “safe”?
  - In a large system, there are many components
    - Should they all be safe?
    - Does it make sense to have 50% safe?
  - Platforms require extensibility
  - Economics may demand leveraging existing code
- Is the debate religious? Is one answer right?

Amdahl's Law Recast

- "Fraction of a system that is sequential determines maximum possible speedup"
  similarly…
- "Fraction of a system that is unsafe determines that maximum possible trustworthiness"

- Suggests two research agendas:
  - Build systems with 0% unsafe code (Singularity)
  - Make existing C / C++ code safer (DieHard)

- We don't know the answer yet, but we do know what questions to ask…

0% Approach - Singularity (MSR)

- Jim Larus, Galen Hunt, and others
  - “Punctuated equilibrium” approach to evolution
- Re-architect and implement OS from scratch
- Design based on latest analysis techniques
- Design principles (partial list)
  - Complete process isolation
  - Type-checked process interaction (channels)
  - As much static analysis / checking as possible
  - Controlled dynamic extensibility (no dlls)
  - Type-safe at the bottom (all code, including OS)

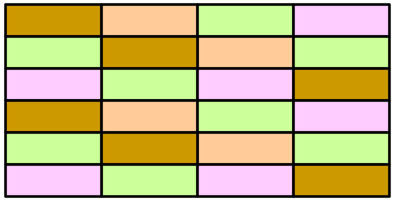
Making C and C++ Safer

- Gradualism approach
- Static analysis / safe subset of C or C++
  - Cyclone [Morrisett], SAFECode [Adve], etc.
- Runtime detection, fail fast
  - Jones & Kelly, CRED [Lam], CCured [Necula], SAFECode [Adve], SafeMem [Zhou], etc.
- Runtime toleration
  - Failure oblivious [Rinard] (unsound)
  - Rx [Zhou], Boundless Memory Blocks [Rinard], ECC, **DieHard,** Samurai, etc.

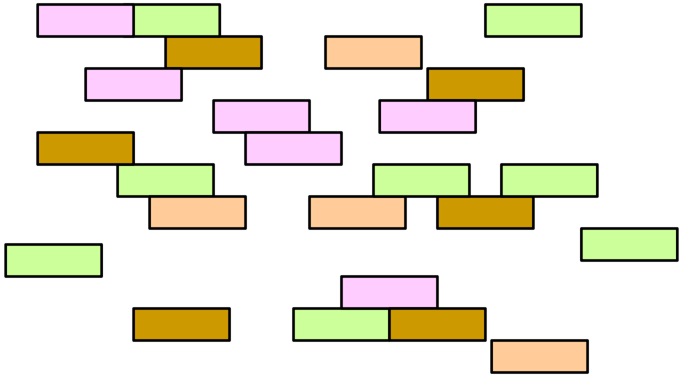
DieHard Allocator in a Nutshell

- Emery Berger and Ben Zorn
  - “Gradualism” approach
- Existing heaps are packed tightly to minimize space
  - Tight packing increases likelihood of corruption
  - Predictable layout is easier for attacker to exploit
- We randomize and overprovision the heap
  - Expansion factor determines how much empty space
  - Semantics are identical
- Easy to use – rejust relink app

Normal Heap



DieHard Heap



Summary

- Most applications and systems are…
  - Written in C and C++
  - Do not detect memory corruptions as they happen
  - Nevertheless, usually robust and reliable…
- Alternatives are available, but
  - More research is needed
  - Answering the question “rebuild from scratch” is expensive
  - Runtime technologies are promising

Additional Information

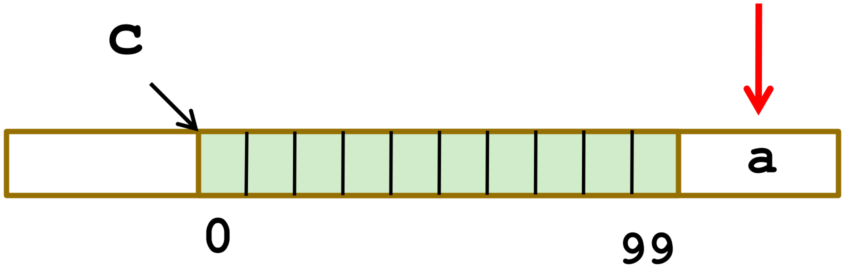
- Web sites:
  - Singularity: http://research.microsoft.com/os/singularity/
  - Spec# : http://research.microsoft.com/specsharp/
  - DieHard: http://www.diehard-software.org/
- Publications
  - Galen Hunt and James Larus, “**Singularity: Rethinking the Software Stack**”, *Operating Systems Review*, Vol. 41, Iss. 2, pp. 37-49, April 2007.
  - Emery D. Berger and Benjamin G. Zorn, "**DieHard: Probabilistic Memory Safety for Unsafe Languages**", *PLDI’06.*

Backup Slides

Avoiding Heap Memory Corruptions

- Buffer overflow

```
char *c = malloc(100);
c[101] = ‘a’;
```



- Dangling reference

```
char *p1 = malloc(100);
char *p2 = p1;

free(p1);
p2[0] = ‘x’;
```