# Type-Safe Programming in C

George Necula

EECS Department
University of California, Berkeley

## Why Memory Safety ?

- The most basic safety property
  - Ensures isolation of component failures
- All programs must have it
  - Does not even need an explicit specification
- Most program analyses are unsound without it
- 50% of security errors are due to buffer overruns

## C and Memory Safety

A large part of embedded software is written in C

C was designed with flexibility and efficiency in mind
- Has many operators that can be used in an unsafe way
- Memory safety is sacrificed

But…
- Many C programs use unsafe operators safely
- In the remaining C programs there are only small portions which are responsible for unsafe behavior



## CCured Idea

1. Devise a program analysis that discovers the safe uses of potentially unsafe operators
2. Insert run-time checks (e.g. array-bounds checks) in those places where safety cannot be statically verified

This way we sacrifice performance instead of safety
- Goal: 0-30% performance penalty
- Performance improves with hardware progress. Unlike safety!

## Checkable Errors

- Array bounds checks
  - Pointer arithmetic outside of object bounds
  - Not always caught by Purify
- Dereferencing a non-pointer (or NULL)
  - Complicated by casts and union types
- Freeing non-pointers, using freed memory



## Kinds of Pointers

- Many pointers are completely "safe"
  - No bad casts, no arithmetic, etc.
  - e.g., `FILE * fin = fopen("input", "r");`
  - These can be represented without any extra information (just a NULL check when used)
- Other pointers are involved in pointer arithmetic but not in bad casts
  - Must carry bounds with them for bounds checking
- Other pointers cannot be typed statically
  - Must carry type information with them

## Static Analysis and Inference

- For every pointer in the program
  - Try to infer the "fastest" sound representation
  - This is like eliminating classes of run-time checks we know will never fail
- Can be formulated as constraint-solving
  - Linear-time whole-program algorithm
  - Can be modularized if the interfaces are annotated
- Proved sound
- Extremely simple, fast and predictable



## Experimental Results

- We have a working prototype
  - Handles the complete ANSI C and gcc extensions
  - Used on programs up to 1M lines of code
  - Used on low level code
- Experimented with
  - SPEC95
  - Linux device drivers
  - Apache modules
  - Network applications: sendmail, openssl, bind, ftpd
- Found new bugs
  - Bugs that Purify missed
- Slowdown: 10-80% (with an average at 50%)
- Typically 70% of the pointers are found safe

## Conclusion

- C programs are "mostly" type safe
  - A static analysis can figure this out
- C programs can be made provable type safe
  - Use run-time checking where static analysis fails
- Safe native methods for Java and C#
- The performance cost is acceptable in some cases
- More work is required to reduce the cost
  - Code size, data size, running time
- Try it out: http://www.cs.berkeley.edu/~necula/ccured