CHESS
CENTER FOR HYBRID & EMBEDDED SOFTWARE SYSTEMS

COLLEGE OF ENGINEERING, UC BERKELEY

Verifying Safety Properties in Assembly Code without Compiler Support

Matt Harren
George Necula

http://chess.eecs.berkeley.edu

Problem:

We need to ensure binary code is safe to execute

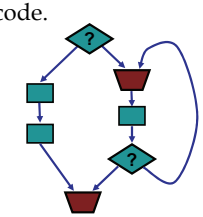
- Is it memory-safe/type-safe?
- Does it obey resource constraints?
- Does it use APIs properly?
- What about other security properties?

- Many software vendors won't release source code, but binary code is hard to analyze.
- We have security tools that analyze source code. Can we use these to help us prove that binary code is safe?

Idea: Proof-Carrying Code that extends easily to different source-code analyses.

How can we do this?

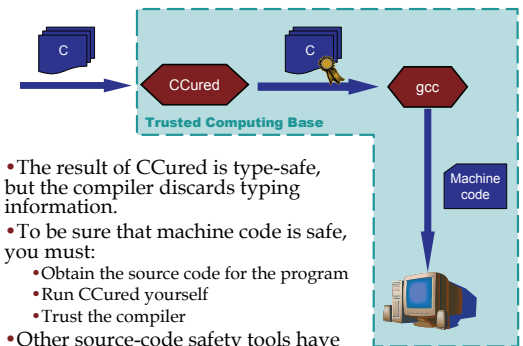
1. We need a dependently-typed assembly language to express the complex invariants of the source-level analysis.
2. We need type inference so that we don't need a type-preserving compiler.
   - Take advantage of off-the-shelf compilers.
   - Strategy: Abstract interpretation over the assembly code.
     - Track the type of each register.
     - Challenge: Join points.
       - How do we deal with dependent values?
       - What information can we throw away to improve performance?



Prototype Implementation

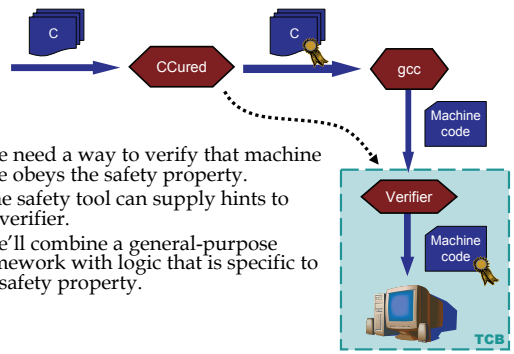
- We can verify most of CCured's output.
- In the spec95 "go" benchmark, it takes about 0.40s to verify each function.
- CCured supplies us with a few hints (e.g. function signatures.)

Old:



- The result of CCured is type-safe, but the compiler discards typing information.
- To be sure that machine code is safe, you must:
  - Obtain the source code for the program
  - Run CCured yourself
  - Trust the compiler
- Other source-code safety tools have the same problem.

New:

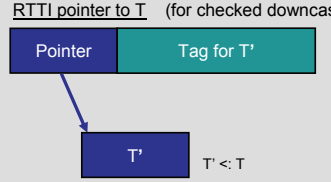


- We need a way to verify that machine code obeys the safety property.
- The safety tool can supply hints to the verifier.
- We'll combine a general-purpose framework with logic that is specific to the safety property.

CCured

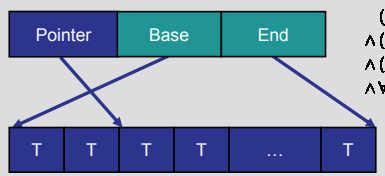
- A source-to-source translator that makes C programs type-safe.
- Inserts runtime checks before unsafe operations.
- Adds *metadata* to some pointers to support these checks.

RTTI pointer to T (for checked downcasts)



- These "fat pointers" are dependent types!
- We must track dependencies between registers.
- Challenge: Changes to memory structures.

Sequence pointer to T (for array bounds checks)



$$(base \le end) \wedge (end - base) \bmod \texttt{sizeof}(T) = 0 \wedge (pointer - base) \bmod \texttt{sizeof}(T) = 0 \wedge \forall i.(base \le (pointer + i \cdot \texttt{sizeof}(T)) < end) \Rightarrow ((pointer + i \cdot \texttt{sizeof}(T)) \texttt{ isPtr } T)$$

Future Work

1. Extend the implementation to more safety tools.
   - Type safety for OO languages.
   - Security properties using Cqual.
2. Generate low-level proofs of safety properties.
   - Foundational PCC
3. Stronger dependent types
   - Allow dependencies among different memory objects.
   - Immutable dependencies.