Conclusions

CS 6371: Advanced Programming Languages

Kevin W. Hamlen

April 30, 2024

Final Exam Content

- Pre-midterm material
  - functional programming (OCaml)
    - parametric polymorphism
    - partial evaluation, currying
    - tail-recursion
  - operational semantics (large-step and small-step)
  - denotational semantics
  - structural induction
  - fixpoint induction
  - semantic equivalence
- Post-midterm material (emphasized)
  - logic programming (Prolog)
    - backtracking search
    - reversible programming
    - order-sensitivity, confluence, Church-Rosser
  - untyped, simply-typed, and polymorphic λ-calculus (System F)
    - type-checking vs. type-inference
    - shallow vs. non-shallow types
    - Curry-Howard isomorphism and type-inhabitation
    - λ-cube
  - axiomatic semantics
    - partial vs. total correctness
    - Hoare Logic
    - loop invariants
    - weakest precondition, strongest postcondition

Summary

APL: The Big Picture

Language Families

- Programming Language Semantics
  - **Operational:** defines program behavior
  - **Denotational:** defines program mathematical equivalent
  - **Static:** defines program safety
  - **Axiomatic:** verifies program correctness
- Three styles of programming:
  - **Imperative:** programs are sequences of instructions
  - **Functional:** programs are functions from inputs to outputs
  - **Logic:** programs are declarative input-output relations

Language Popularity

- Which languages are most popularly used in the "real world" (e.g., industry)?
  - unquestionably the imperative ones (C/C++/Java)
- Why?
  - easy to compile (no longer a compelling reason)
  - momentum (large labor pool, well-developed tools)
  - easy to write code that almost works
- The "software crisis"
  - Microsoft spends over half of its budget on testing.
  - Programs still regularly buggy, exploited, etc.
  - "Find better programmers" is not the answer.

Better Programming Languages

What makes a language "advanced"?

- Correctness/reliability over efficiency (within reason)
  - "If I want it to run faster, I'll buy more processors."
  - Compilers as proof-assistants
- Small-gap translation from mathematical spec to code
- Separation of concerns (the "what" vs. the "how")
- Succinctness
  - less code = fewer bugs
  - code-reuse (parametric polymorphism)
- Modularity
  - object-oriented programming
  - See also: OCaml module system, aspect-oriented programming
- *Programmer efficiency* vs. program efficiency

Language Choice

Should we bury C/C++/Java?

- No! C/C++ is the right solution for many things:
  - writing the inner loop of a matrix multiplier
  - writing device drivers (but use formal verification)
  - implementing some runtime libraries (e.g., fast string libs)
- But unsafe languages are overused/misused due to lack of programmer awareness of alternatives and trade-offs.
- Java/Rust was/is a great step forward...
  - brought type-safe programming to the masses
  - popularized automated garbage-collection
  - extended (limited) forms of safety to low-level code
- But such languages still have major weaknesses:
  - uncaught exceptions are only slightly better than crashes
  - Java language definition defies optimization
  - foundation on untyped, unsafe runtime leads to many compromises
  - lack of dependent typing means no correctness-checking

Grand Challenges

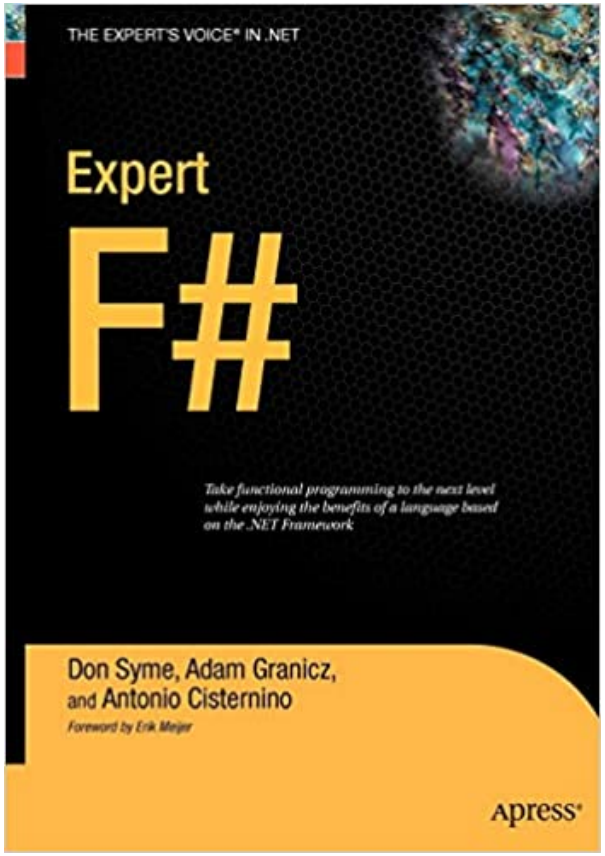
- How can we make it "easy" (or easier) to construct iron-clad, fully machine-validated software?
  - Example: *Compositional CompCert* [Stewart et al., POPL 2015]
- What kind of languages might segue the imperative world toward strongly typed, functional/declarative programming?
  - Example: F# [Syme et al., 2001]
- Can we use modern PL theory to debug/correct/analyze legacy codes?
  - Example: Compiler-instrumented Dynamic Secret Redaction for C [Araujo & Hamlen, USENIX Security 2015]
- Can we use PL theory to solve security problems like data confidentiality enforcement?
  - Example: Java Information Flow [Myers, POPL 1999]
- How can we create verified, highly parallelized software?
  - Example: n-Variant Framework of Verification for Java Source Code on CPU$\times$GPU Hybrid Platform [Duan et al., HPDC 2019]

Relevance/Usefulness

- Practical right now
  - functional and logic programming
  - operational/denotational semantics for compiler design and analysis
  - static semantics for program safety verification
- Fundamental for understanding real-world software verification
  - λ-calculus
  - Hoare Logic
  - structural/fixpoint induction
- Learning to write formal, rigorous proofs
  - essential if you want to do science, not just programming
  - infrequently taught at the undergraduate level
  - If you can't prove easy things, you can't program hard things.
    - every program is a constructive proof (Curry-Howard)
    - Example: If you can't reason inductively, you can't program recursively.

Functional Programming at Microsoft

Microsoft's Functional Language:
F# (OCaml for Visual Studio .NET)

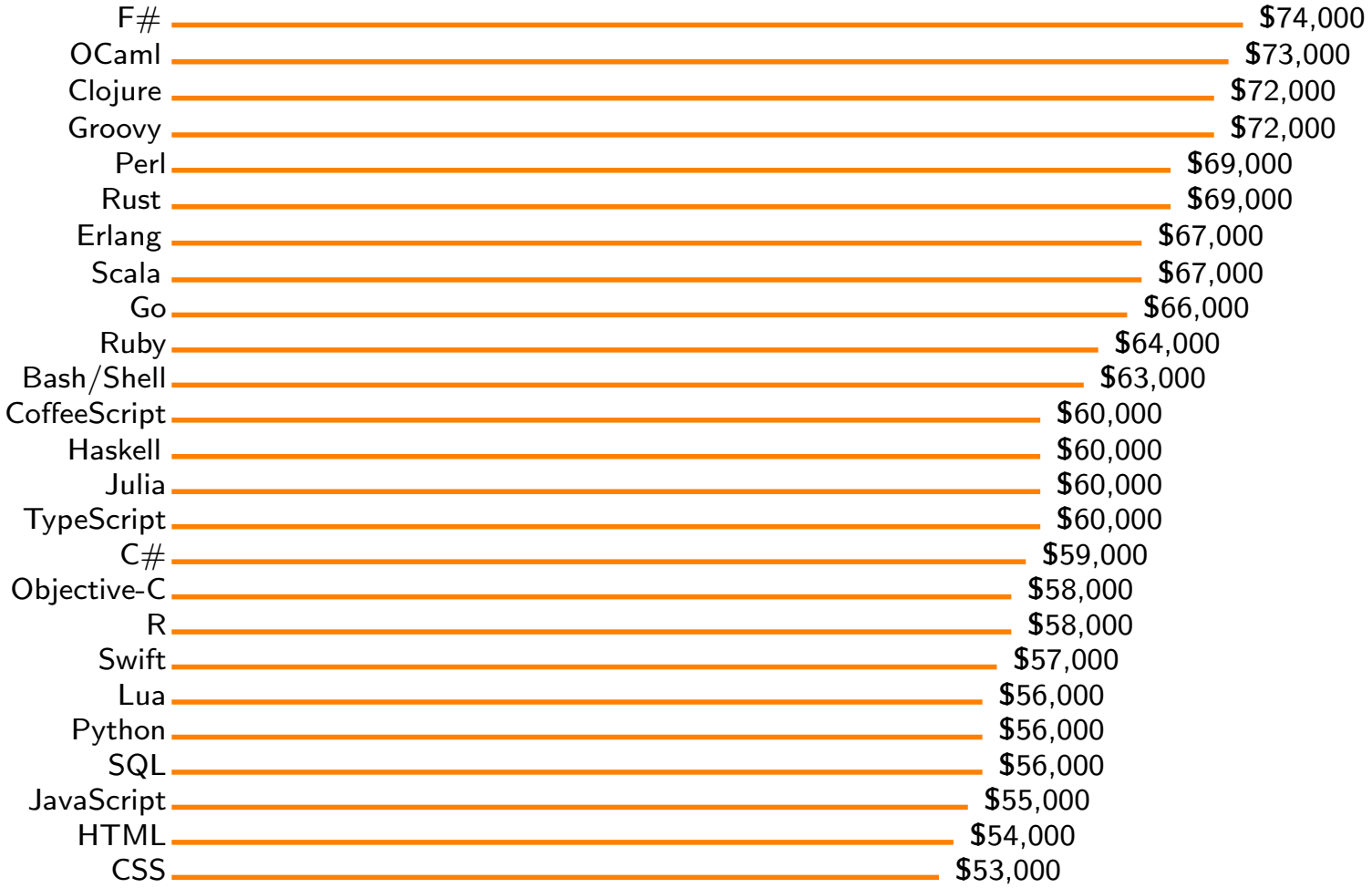


Functional Programming at Facebook

Facebook's Polymorphic, Statically-typed Web Dev Language

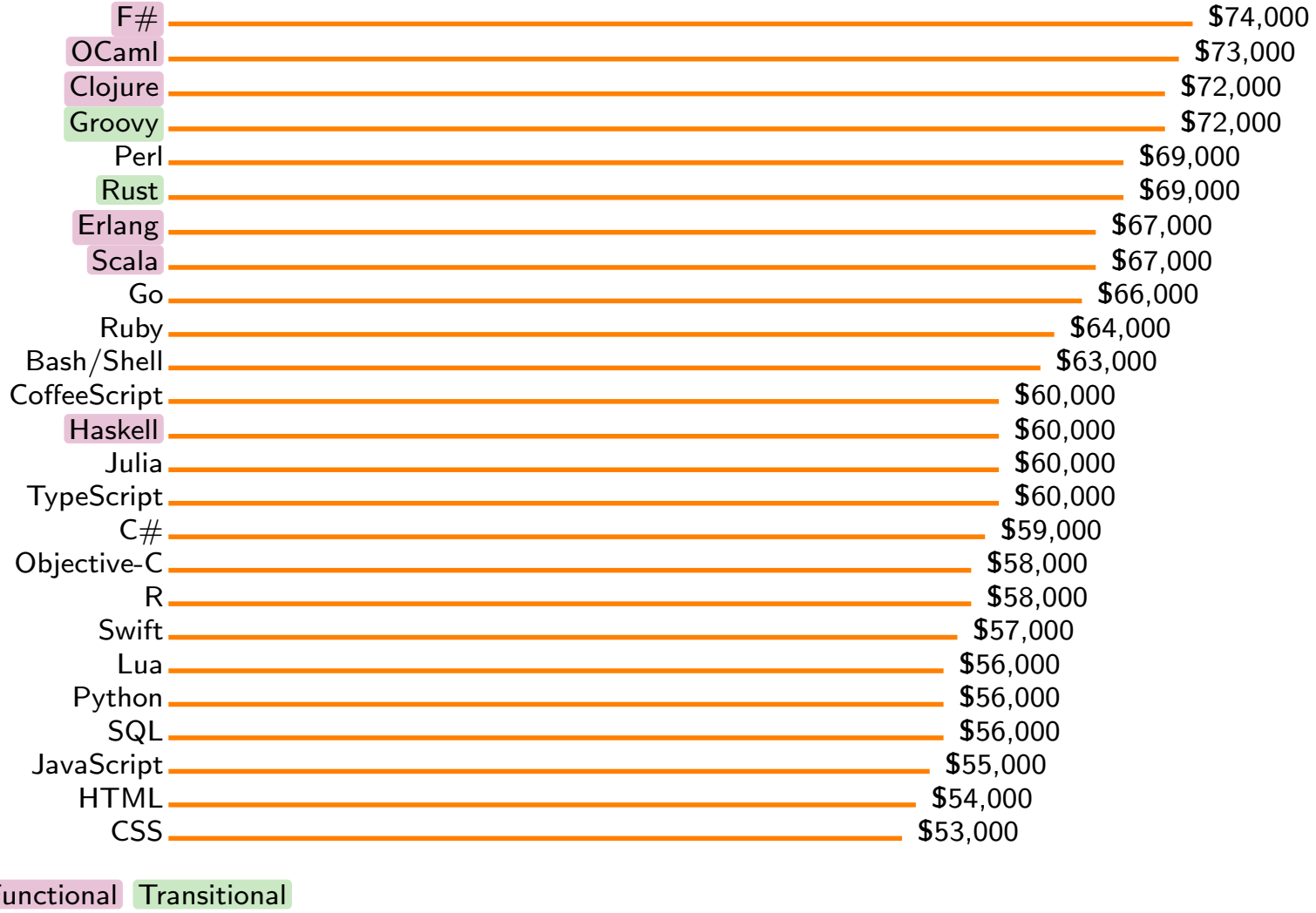
Stack Overflow Developer Survey

What Languages Are Associated with the Highest Salaries Worldwide?



Stack Overflow Developer Survey

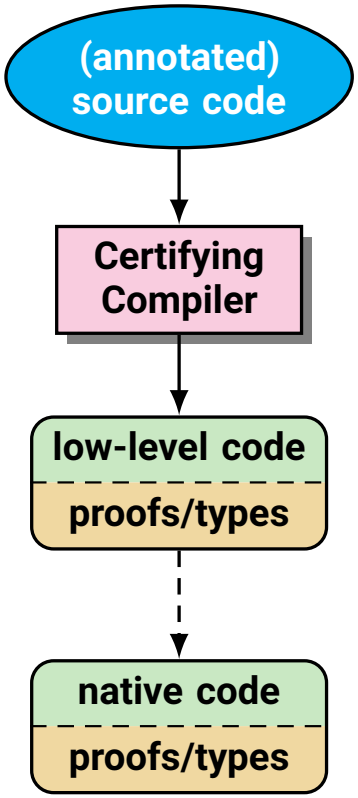
What Languages Are Associated with the Highest Salaries Worldwide?



Next Steps

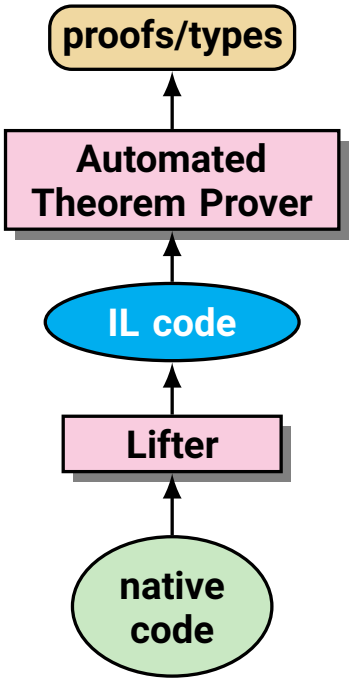
- CS 6353: Compiler Construction
  - We learned how to design and analyze a language.
  - CC teaches how to build a compiler for a language.
- CS 6374: Computational Logic
  - learn about automated theorem proving
  - tools for doing formal software verification
- CS 6301.nnn: Language-based Security (shameless plug)
  - type theory for security analysis & enforcement
  - information flow, access control, etc.
- Independent study research
  - Dr. Gupta: logic programming
  - Me: language-based security

Bottom-up Formal Methods



Top-down FM:

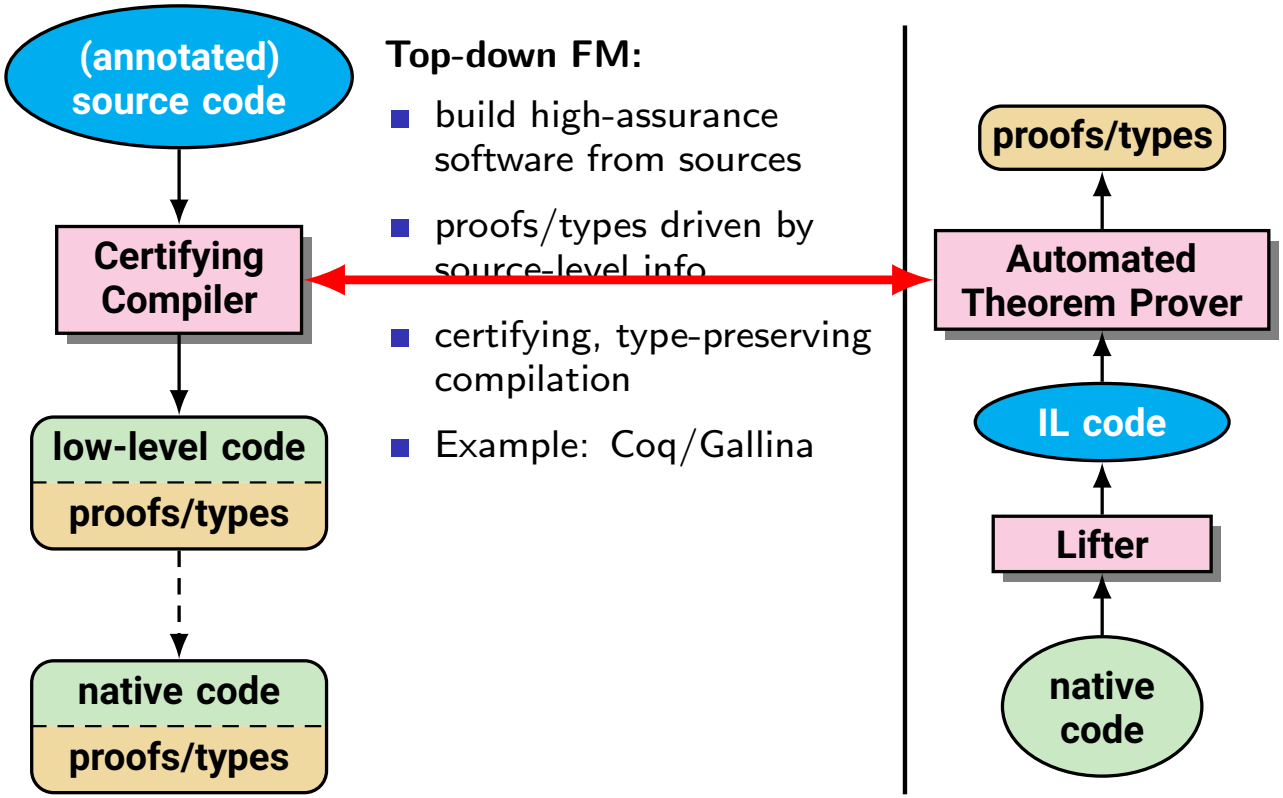
- build high-assurance software from sources
- proofs/types driven by source-level info
- certifying, type-preserving compilation
- Example: Coq/Gallina



Bottom-up FM:

- obtain high assurance for source-free software
- proofs/types driven by native-level information
- ISA formal semantics specification & recovery

Bottom-up Formal Methods



Top-down FM:

- build high-assurance software from sources
- proofs/types driven by source-level info
- certifying, type-preserving compilation
- Example: Coq/Gallina

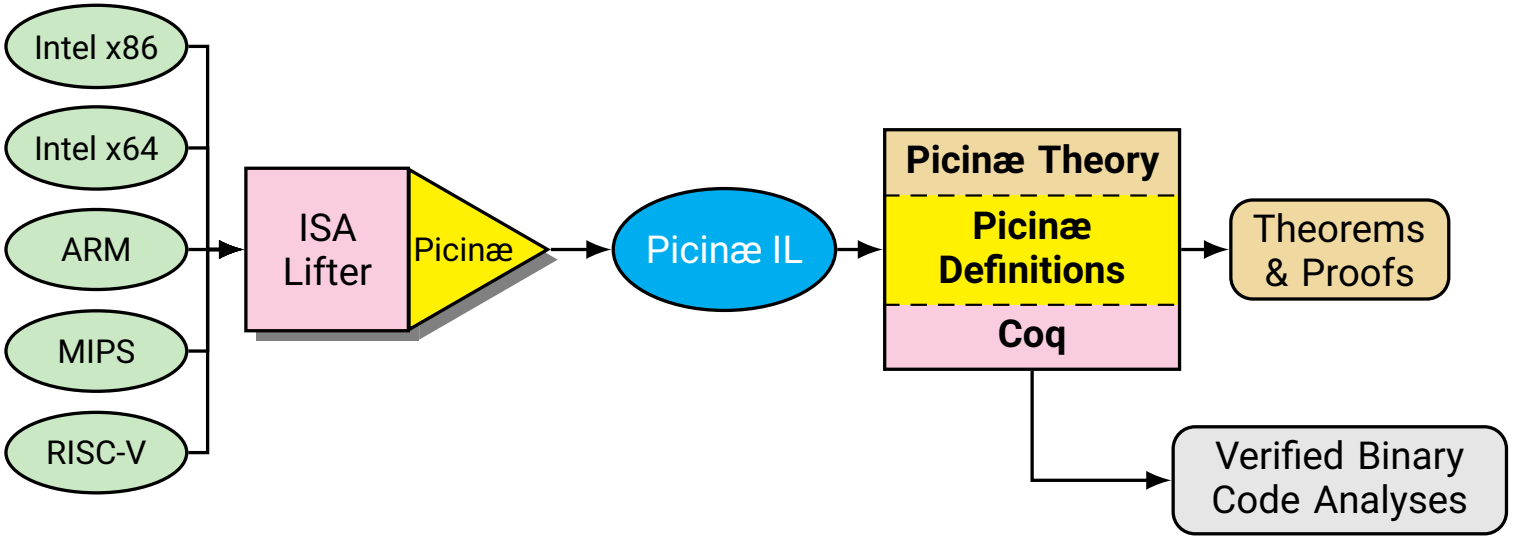
Bottom-up FM:

- obtain high assurance for source-free software
- proofs/types driven by native-level information
- ISA formal semantics specification & recovery

Picinæ: Platform in Coq for INstruction-level Analysis of Executables

Bridging two technologies:

- ISA Operational Semantics: Ghidra P-Code (NSA), Binary Analysis Platform (CMU)
- Program-proof Co-development: INRIA Coq



Soft(ware) Targets

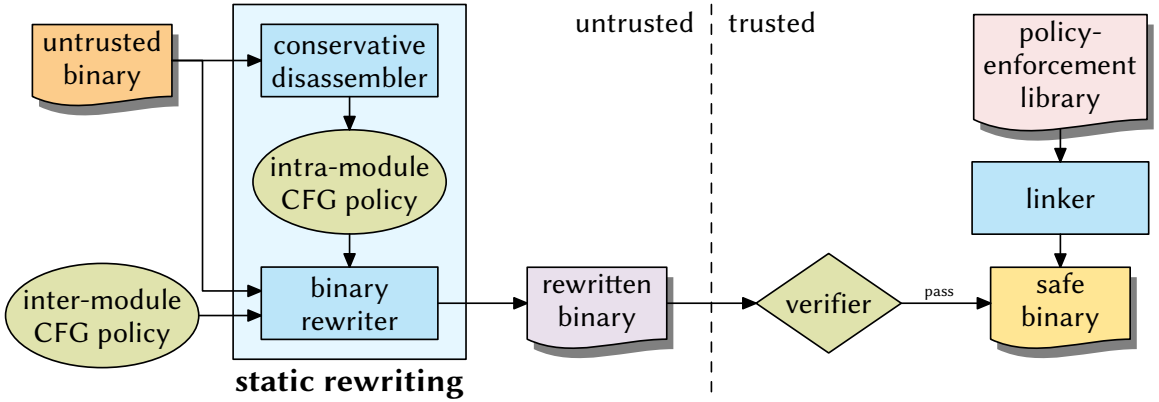
Windows / MacOS in mission-critical environments:

- "$\approx 75\%$ of control systems are on **Windows XP** or other **nonsupported** OSes" [Daryl Haegley, Office of Assistant Secretary of Defense for Energy, Installations and Environment]
- $> 25\%$ of all government computers currently run an **outdated Windows** or **MacOS** operating system [BitSight, 2017]
- DHS, Coast Guard and Secret Service currently store top secret information on **outdated Windows 2003** servers [OIG-18-56, 2018]
- Hundreds of satellites run **Windows 95** and/or are contreolled by **Windows Mobile** devices.

Object Flow Integrity

Securing Large, Component-driven Software Products:

- denotational semantics of C++ object hierarchies
- reflective programming for C++
- automated code synthesis from security specifications
- median 0.34% overhead on COTS Windows applications



W. Wang, X. Xu, K.W. Hamlen, "Object Flow Integrity." In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pp. 1909–1924, November 2017. [tier-1]

Course Feedback

Feedback is welcome!

- Topics you wish had been included but weren't?
- Homework/exam difficulty level
- Style of instruction, lecture format
- Course resources (no textbook...)

Reminder: Please take time to submit course evals!