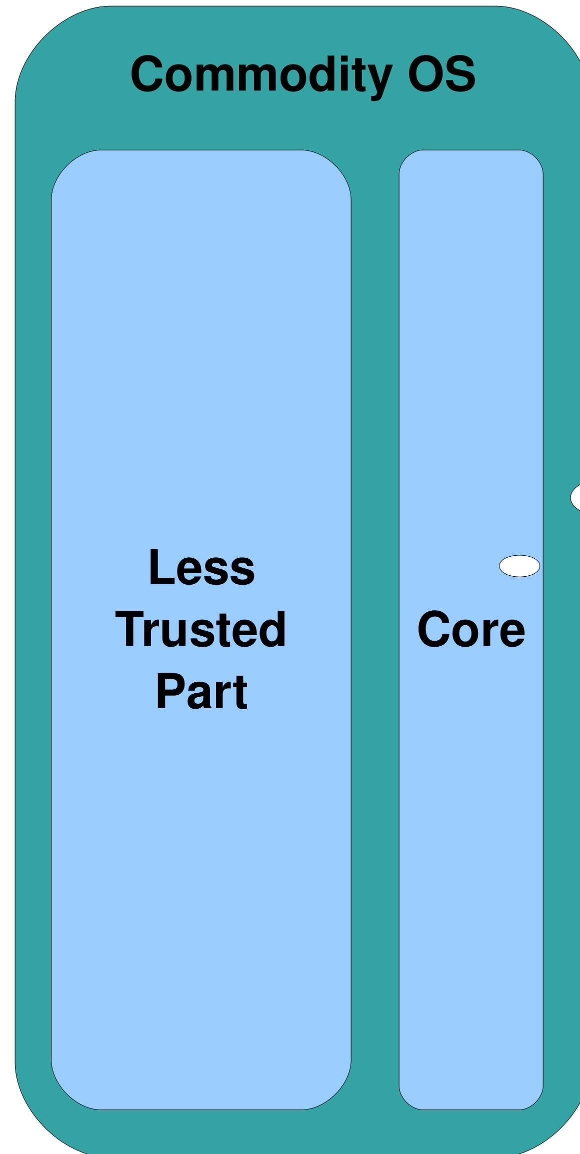
Verifying Critical Systems with Mechanized Theorem-Proving

Adam Chlipala (standing in for Greg Morrisett)
Harvard University
August 20, 2009

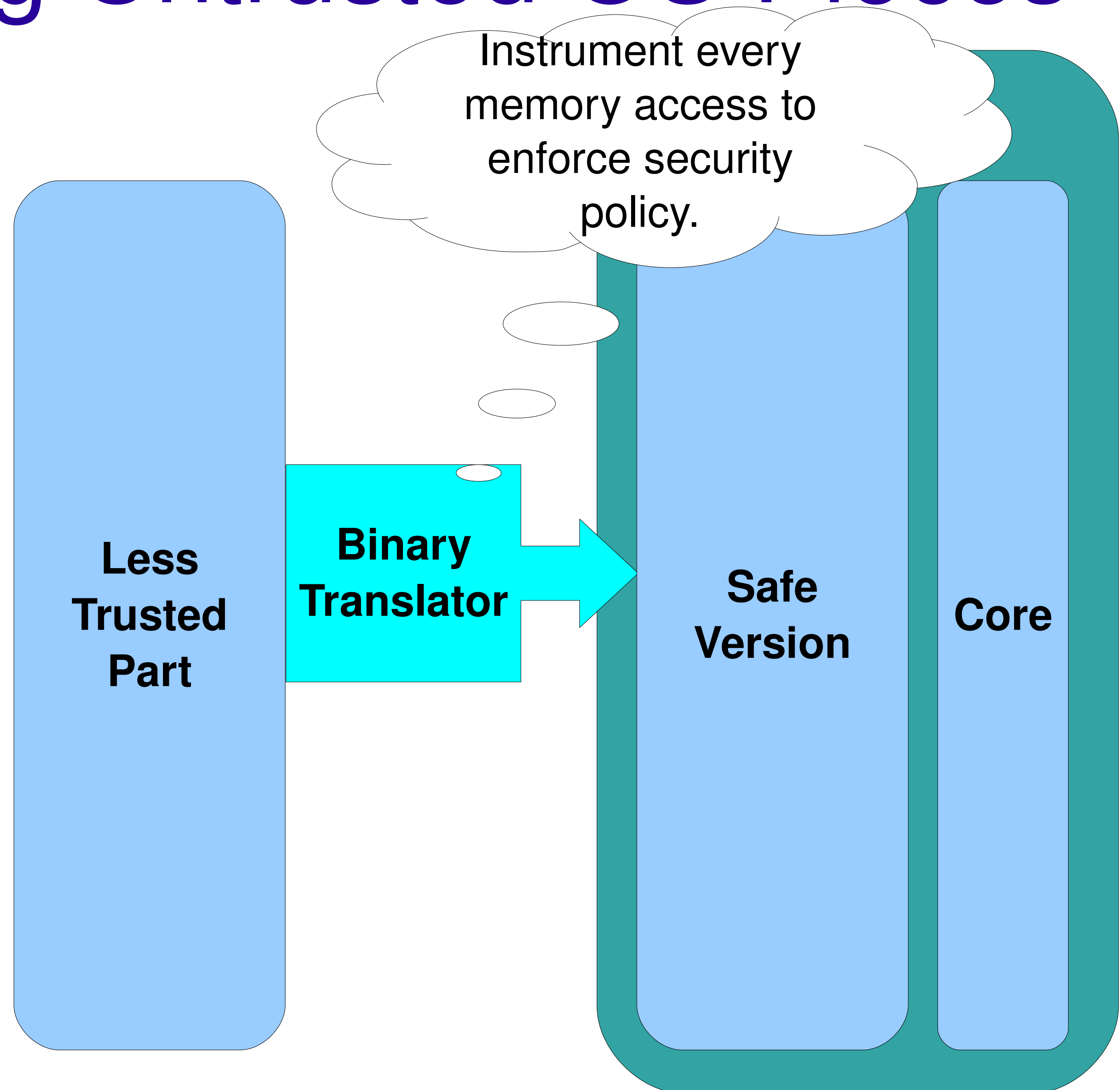
Containing Untrusted OS Pieces



Minimal and subjected to rigorous audit

Containing Untrusted OS Pieces

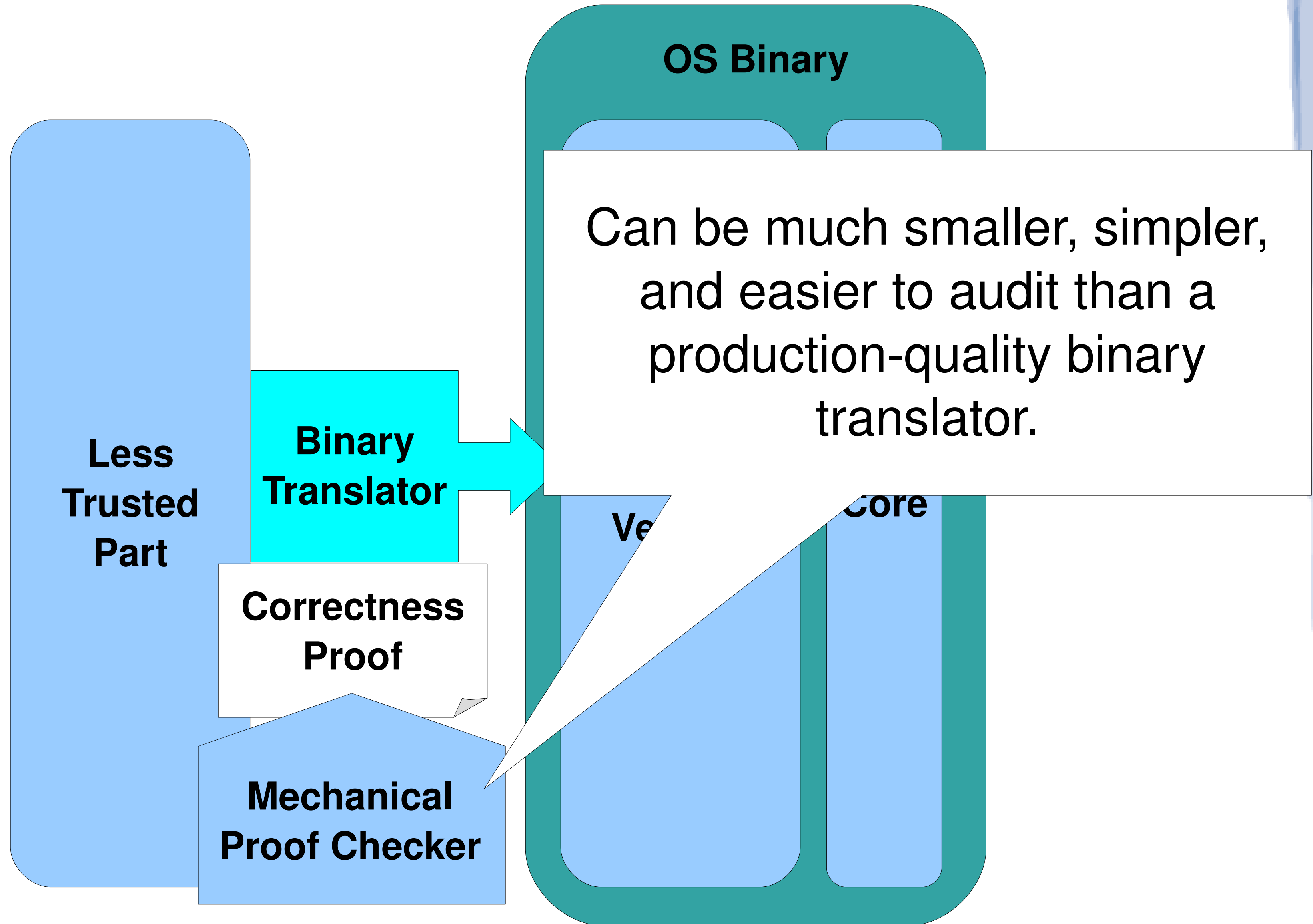
“Software-Based Fault Isolation” (SFI)



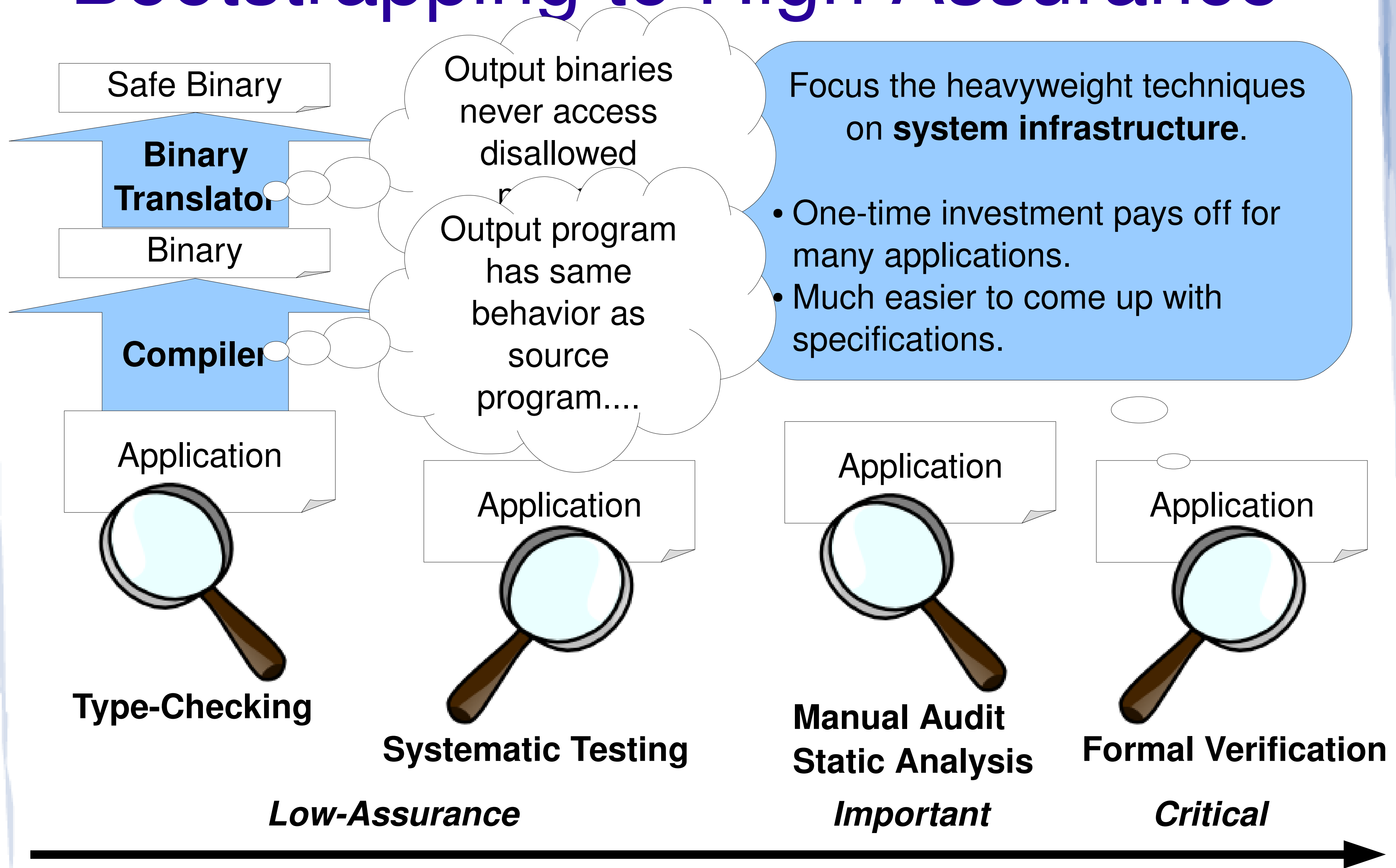
Harder Than It Looks....

- Vulnerability discovered in a well-known SFI implementation, MiSFIT by Small & Seltzer
  - Trusted the C compiler too much
- Google held a contest to find security holes in Google Native Client (which uses SFI)
  - 1st place team filed 12 distinct bug reports that Google accepted as security flaws
- Wanted: **Systematic way of avoiding these bugs!**
  - When applying this to untrusted OSes, a small bug can undermine the whole system.

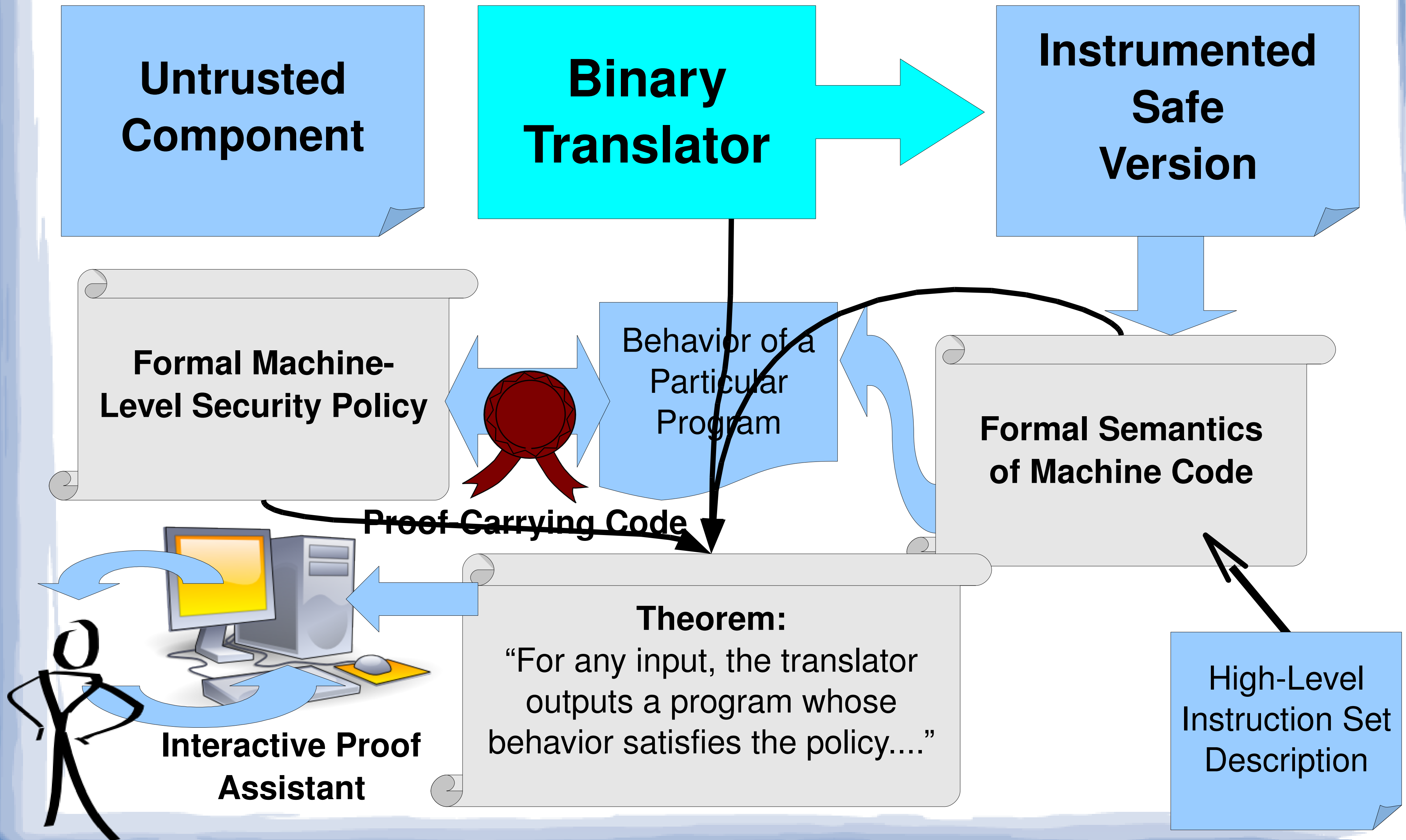
Containing Untrusted OS Pieces



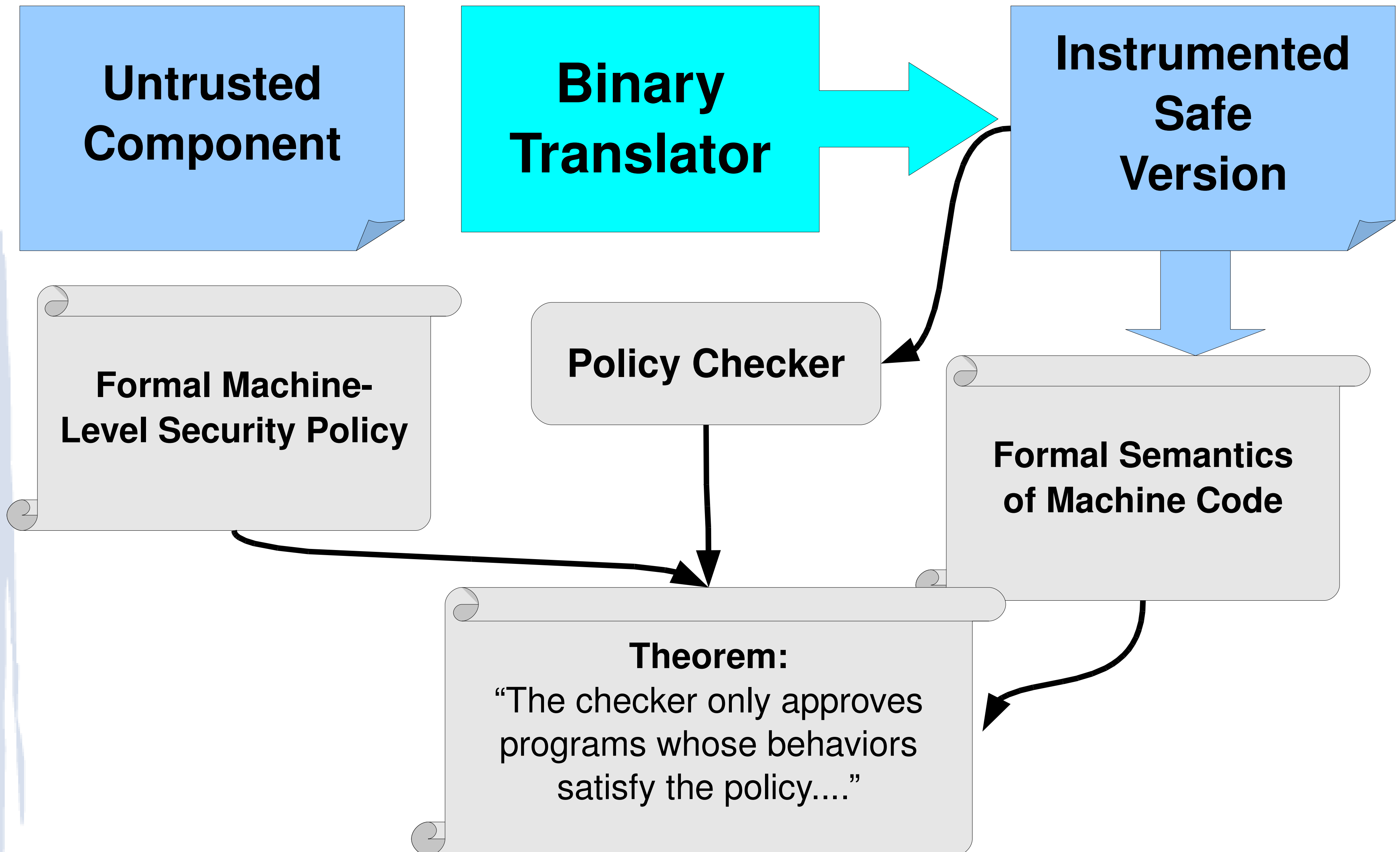
Bootstrapping to High Assurance

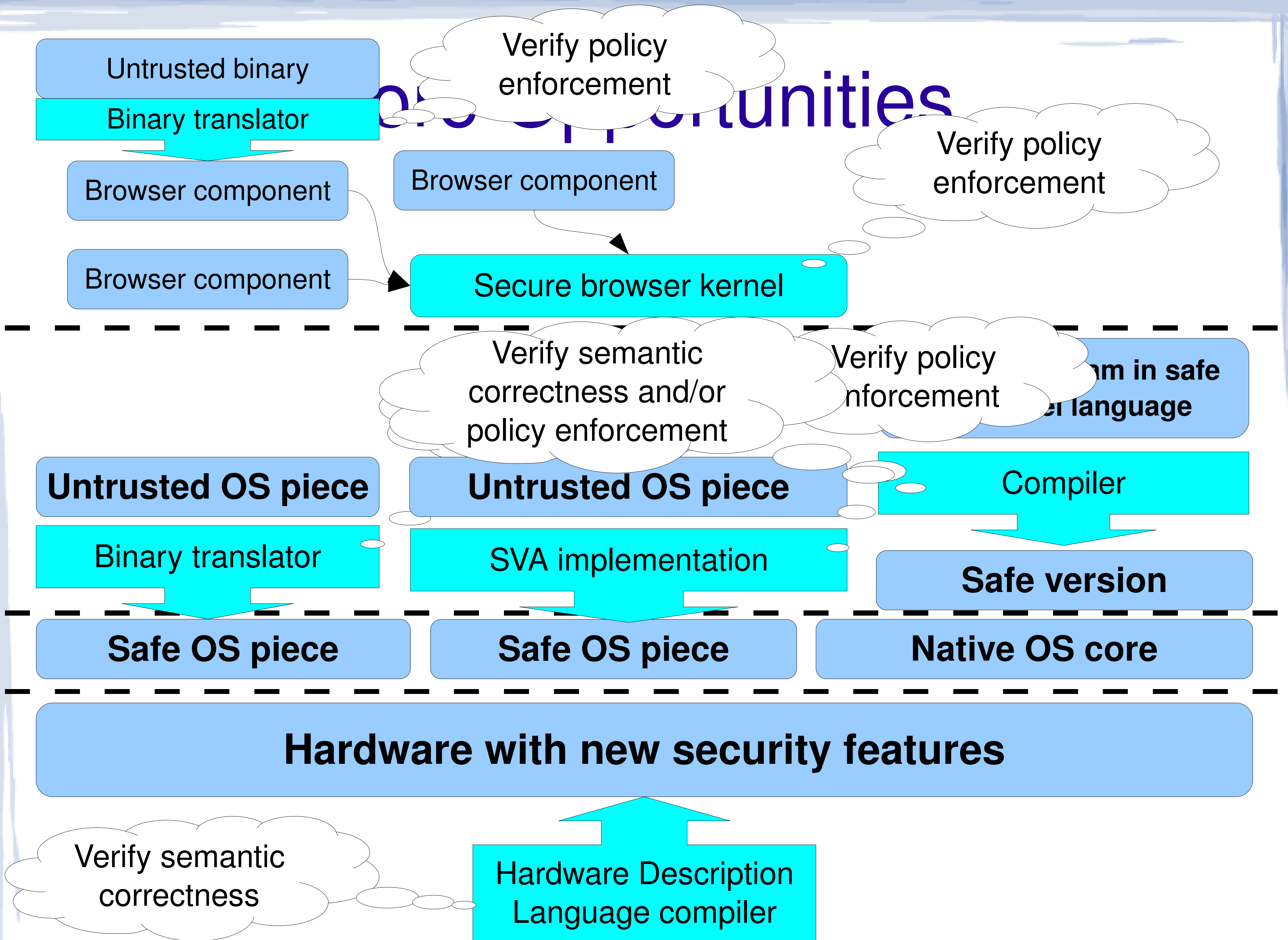


Verifying a Binary Translator



Verifying a Checker Instead





Untrusted binary
Binary translator
Browser component
Browser component
Browser component
Secure browser kernel
Verify policy enforcement
Verify policy enforcement
Verify semantic correctness and/or policy enforcement
Verify policy
nforcement
Untrusted OS piece
Untrusted OS piece
Compiler
Binary translator
SVA implementation
Safe version
Safe OS piece
Safe OS piece
Native OS core
Hardware with new security features
Verify semantic correctness
Hardware Description Language compiler

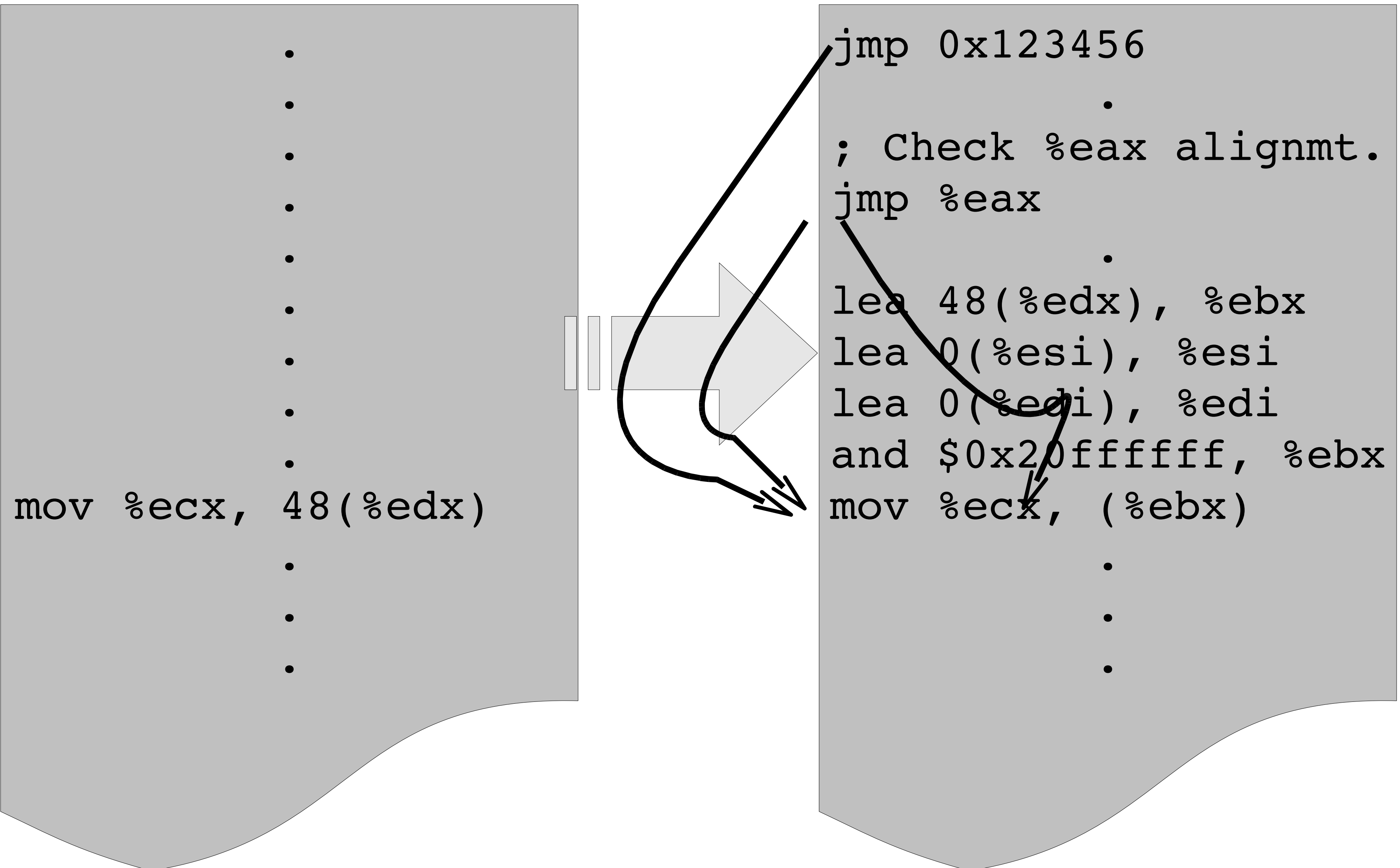
Is Full Verification Practical?

- Growing popularity of high-level languages makes it easier to verify real systems
- Advances in SAT solving, etc., help minimize time spent on inessential details
- Increased attention to “proof engineering” techniques for building maintainable proofs
- Specific recent progress on verifying tools that produce or analyze assembly/machine code

Conclusion

- Safe use of untrusted OS components requires getting *many details* right.
- Formal verification is the champion technique for this kind of detail work.
- By focusing on key components that ensure the security of many applications, we achieve a very attractive cost-benefit trade-off in the use of verification.

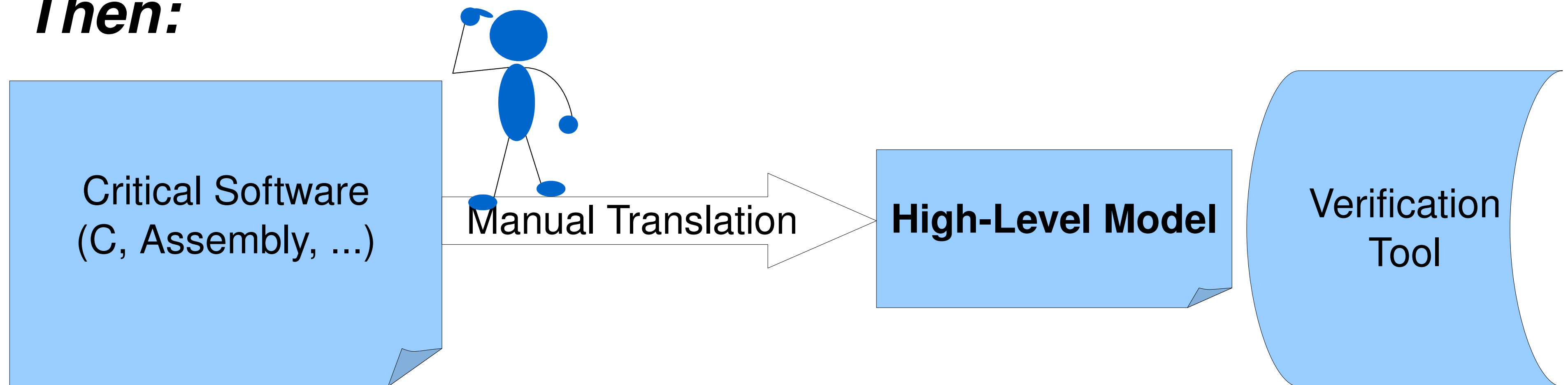
Subtleties in x86 Binary Translation

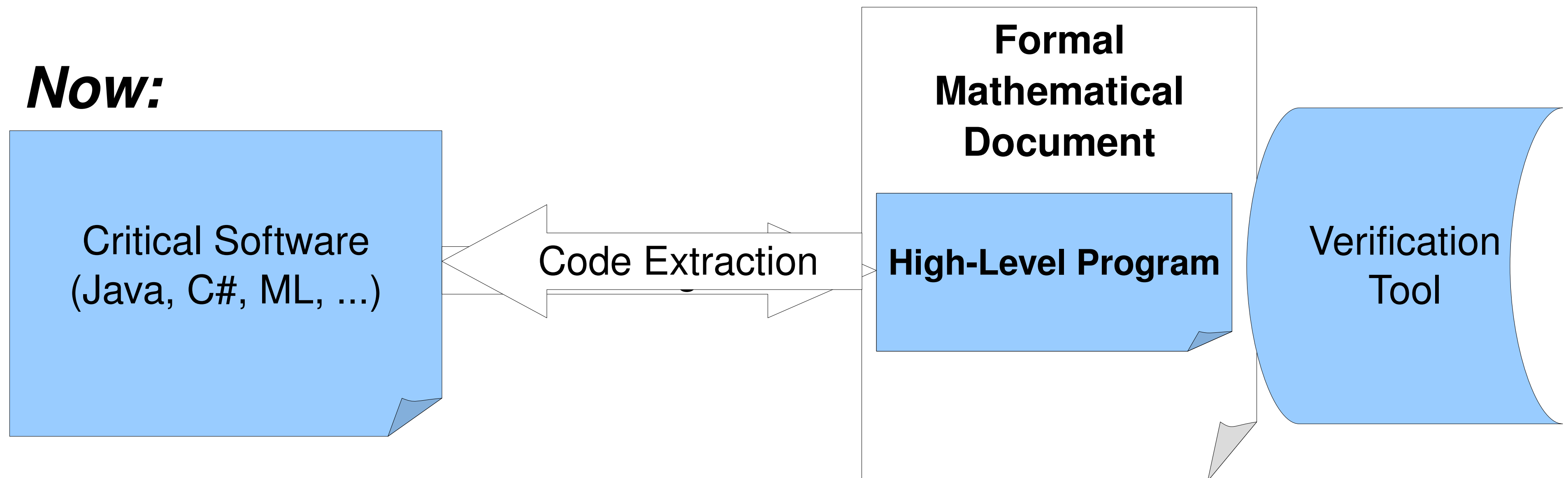


Formal Verification

Didn't we try that already?

"Why Now?" Reason #1: High-Level Languages



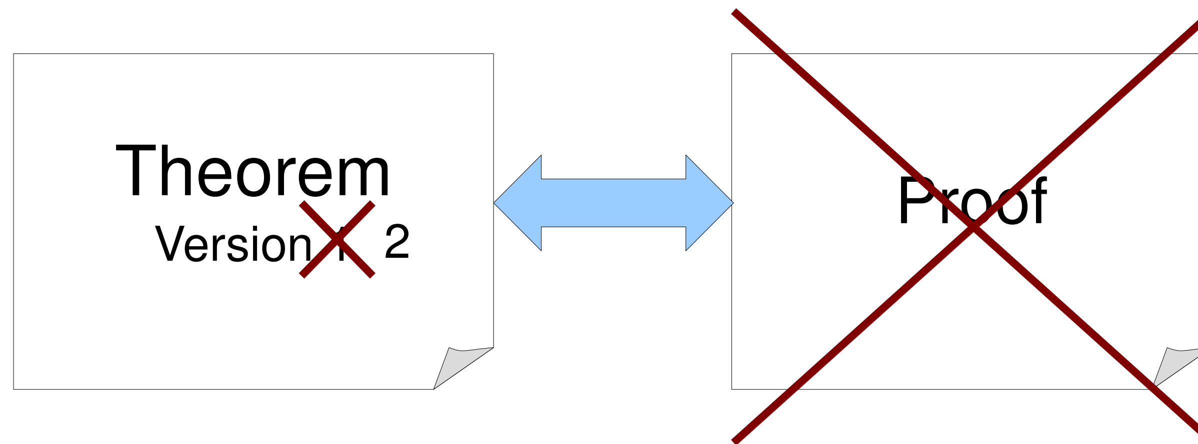


"Why Now?" Reason #2: Advances in Automation

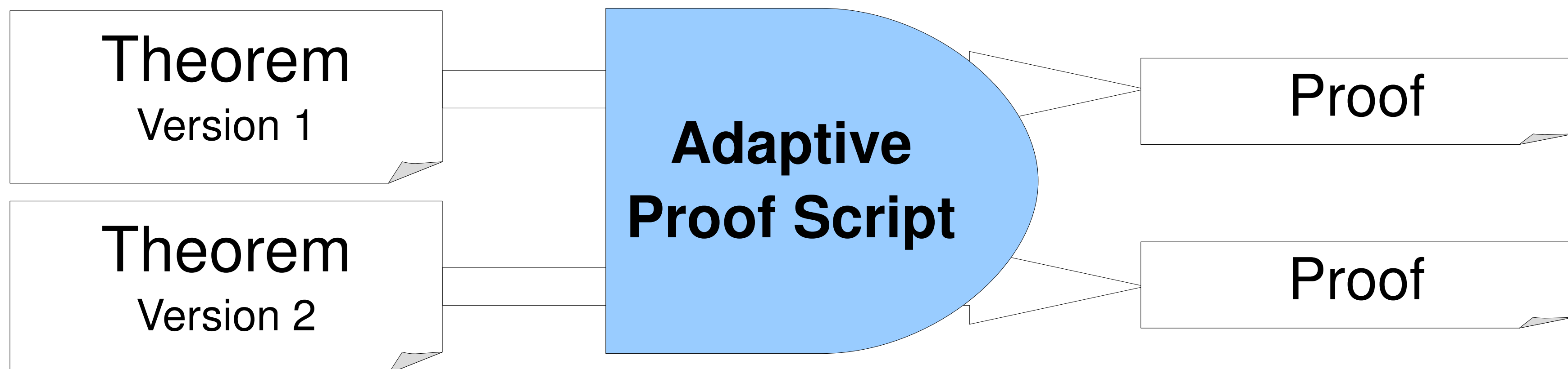
- Moore's Law makes brute-force proof search practical in many cases.
- Even SAT solvers are practical today!
  - Off-the-shelf solvers for any problem that can be reduced to Boolean logic
- Satisfiability-modulo-theories
  - Black boxes that reason about combinations of Boolean logic, linear arithmetic, arrays, ...

“Why Now?” Reason #3: Advances in Proof Engineering

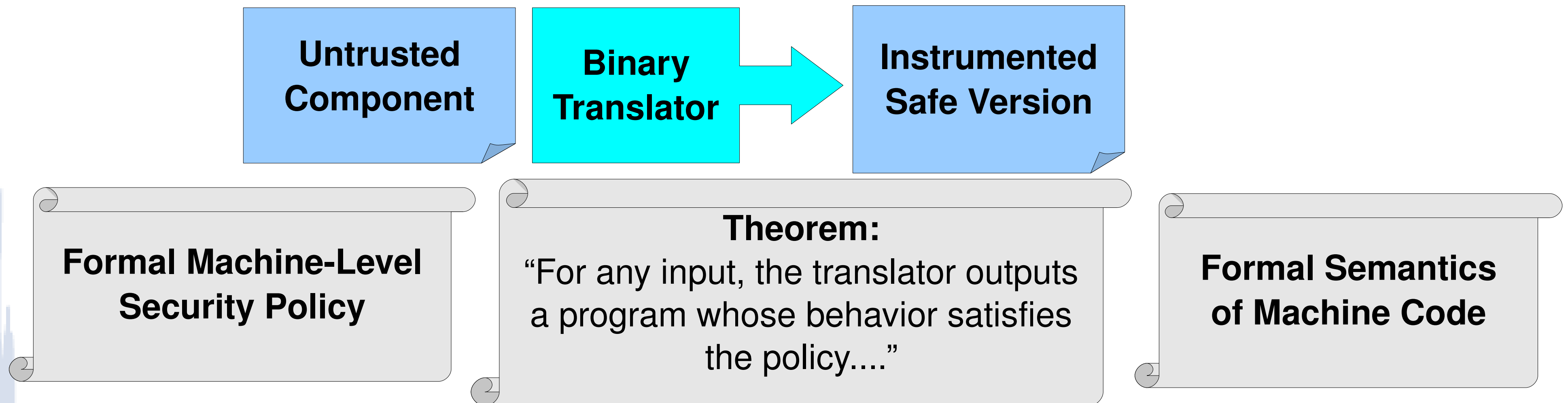
Classical Verification:



With Modern Proof Assistants:

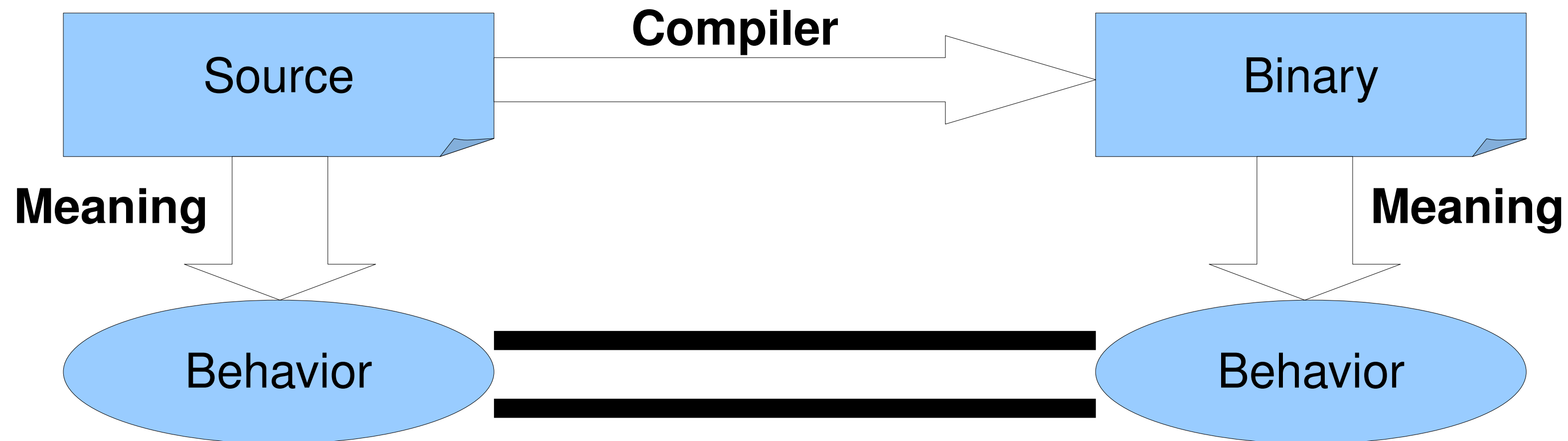


Advances in Low-Level Verification



- Foundational Proof-Carrying Code project [Appel et al.] built an ML compiler with safety of binaries proved from the ground up.
- Certified Assembly Programming project [Shao et al.] has verified garbage collectors, schedulers, etc..
- McCamant & Morrisett verified a partial model of an x86 SFI implementation.
- Certified Verifiers project [Chlipala] built verified x86 memory safety verifiers out of reusable components.

Advances in Compiler Verification



- CompCert project [Leroy]
  - Verified compiler from a large subset of C to PowerPC assembly
  - Optimizations: constant propagation, common subexpression elimination, register allocation, branch tunneling
- Verified compiler for language with key features of ML [Chlipala]
  - Every theorem proved with an adaptive script
  - Source to assembly, with optimizations, in under 6000 lines