The Problem

- All modern computing infrastructure has to make a tradeoff between Performance and Security

Platform:	Security	Performance	Capabilities	Assurance	Notes
Linux Windows	Medium	Medium	High	Low	Large, monolithic, trusted code base. Relies on hardware assisted memory space isolation.
SEL4	High	Low	Low	High	Fully proven implementation binary. Can't do much.
HPC	Varies	High	High	Varies	Security by business relationship. Typically single user at a time.
Microservices Virtualization	High	Medium	Medium	Medium	Exchanges memory space isolation costs for network serialization costs and administration costs
C C++	Low	High	High	Low	Relies on platform level memory space isolation for cross process security. Relies on programmer perfection for intra process security.
Python Java C# Javascript	High-Medium	Medium-Low	Medium	Medium	Uses garbage collection and pervasive runtime to assure memory safety. Can't opt out of platform level memory space isolation.

The Future

- Imagine a computing world where:
  - Creating high assurance code is *easy*.
    - Cyber-Defense
    - Avionics
    - Firmware/SCADA
    - Medical
  - It is provably impossible for clicking on a link in an email to hurt you.
  - Running untrusted code from the Internet is *fast*.
  - Running untrusted code from the Internet is *safe*.

The Roadmap

- Create a proving compiler.
  - Next slides
- Using that compiler, Create an operating system that:
  - Will not load a binary unless it comes with a correct proof of good behavior.
  - Uses this 'safe only loading' to avoid the performance penalties of memory space isolation.
  - Uses this lack of memory space isolation to allow separate programs to interoperate with extremely high performance.
- Create an ecosystem of software targeting this OS.
  - Get everyone into this ecosystem.
- Push the security higher and lower in the software stack.
  - Higher: Capability based security.
  - Lower: Drivers with proofs.
  - Lower: Firmware with proofs.
  - Lower: Open hardware.

The Proving Compiler

- Input:
  - Program source code in a performant language
    - POC: A subset of C. IOC: Rust. FOC: Rust and others (Possibly C, C++, GLSL, SPIR-V, Python, Javascript, WebASM).
  - Annotations for parts of the program.
    - Suggesting invariants for the compiler to use in its proof.
    - In Rust annotations would only be needed for `unsafe` modules/blocks.
- Output:
  - A machine language binary of the program.
    - POC: unoptimized. FOC: optimized.
  - A machine verifiable proof of memory safety of *the binary* (not the program).
    - In minified, compressed Coq or Proof Carrying Code.
    - Only assumes that the executing platform behaves as specified.

The Proving Compiler
Resources needed:

- 1-2 FTE-Years to POC.
- Ability to collaborate in the open with academia and existing open projects in this space.
  - Projects: Rust Belt, Rustc, TAL, Proof Carying Code, roboglia, Redox-OS, seL4, LLVM/clang, Corrode.
  - Minimal exposure to classified information.
  - Minimal exposure to proprietary information.
  - Minimal proprietarization of work product.
- (Optional) Ability to collaborate with existing proprietary projects in this space.
  - Projects: Singularity, Midori.
  - Requires business relationships.