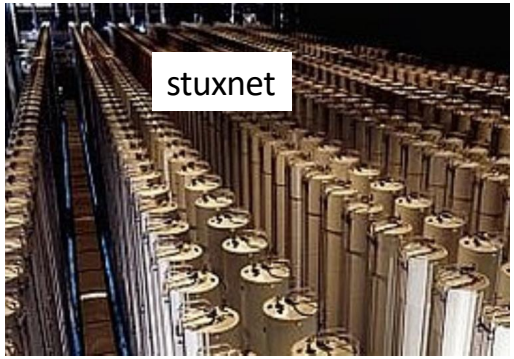
A Logical Approach to Programming Language Design and Verification

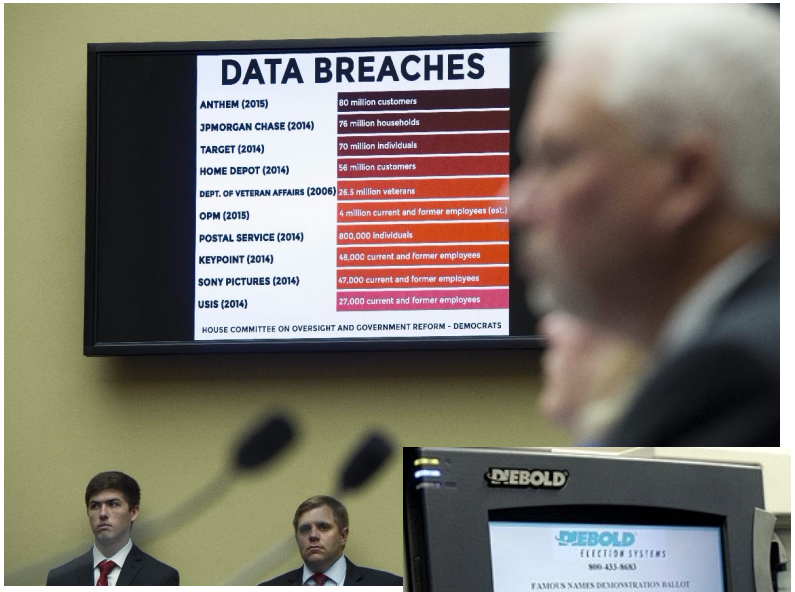
Stephanie Weirich

ENIAC President's Distinguished Professor of Computer and Information Science

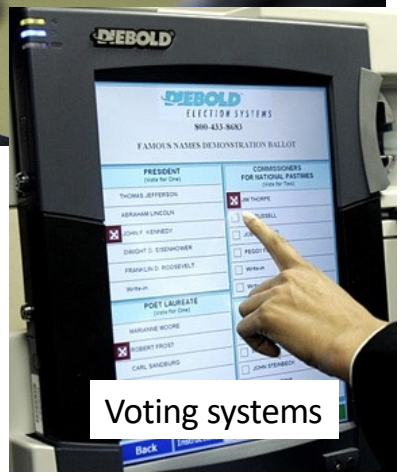

Can we use logic to make
software more reliable?

Do we really need reliable software?



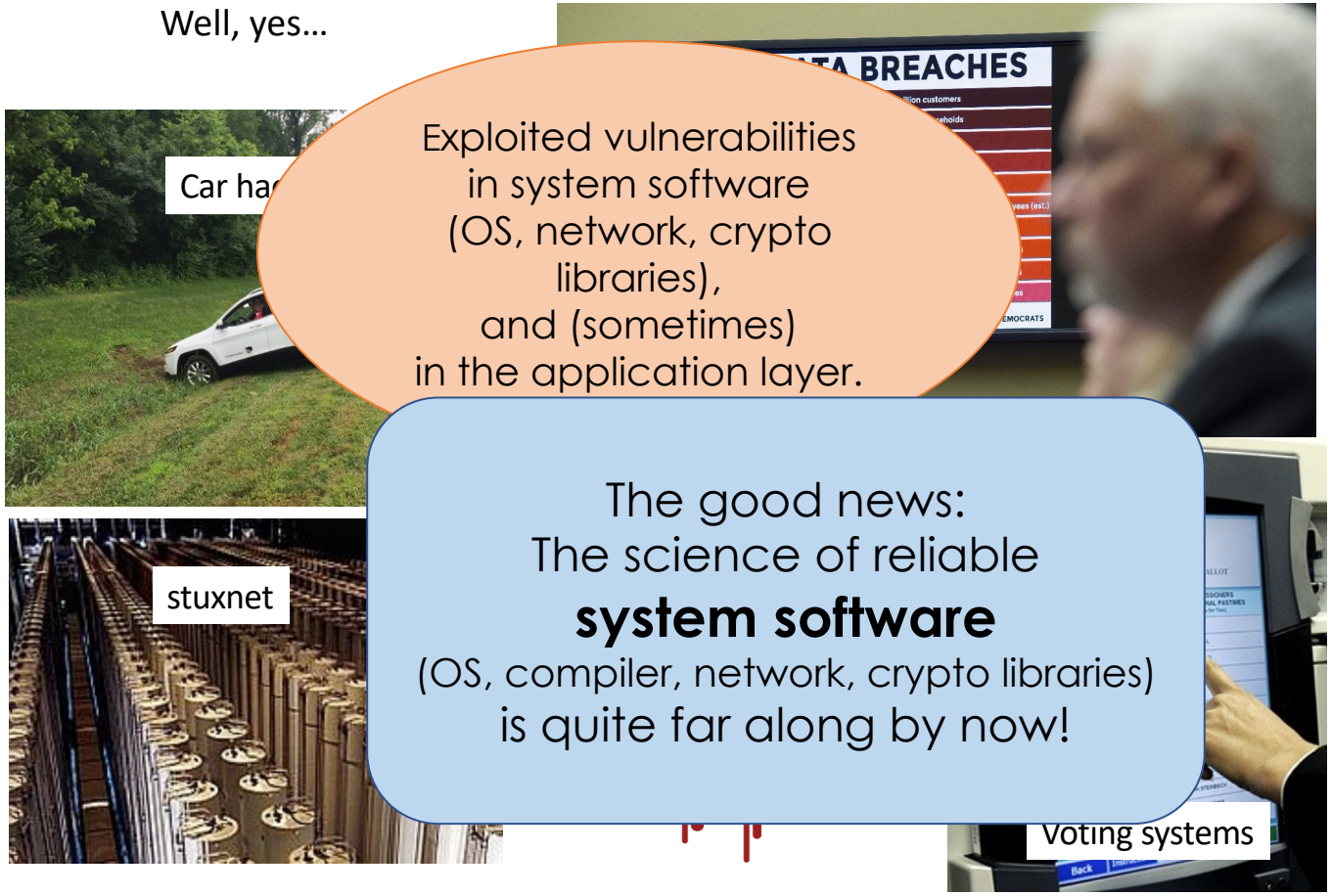


Security vulnerabilities



Do we really need reliable software?

Well, yes…

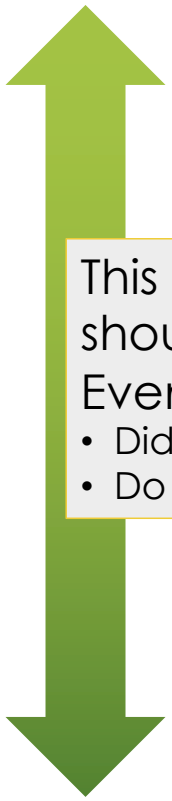


Exploited vulnerabilities in system software (OS, network, crypto libraries), and (sometimes) in the application layer.

The good news:
The science of reliable
system software
(OS, compiler, network, crypto libraries)
is quite far along by now!

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical / Logical
  - Sound programming languages tools
  - "Formal" verification



Less "formal": Techniques may miss problems in programs

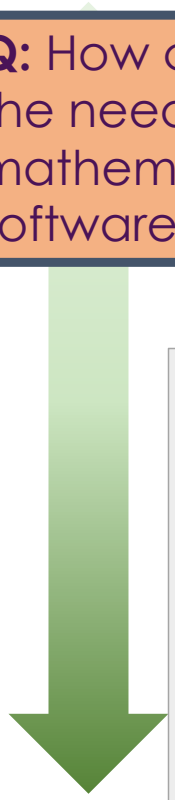
This isn't a tradeoff… all of these methods should be used.
Even "formal" methods can have holes:
- Did you prove the right thing?
- Do your assumptions match reality?

More "formal": eliminate *with certainty* as many problems as possible.

Goal: Verified Software

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical / Logical
  - Sound programming languages tools
  - “Formal” verification

Q: How can we move the needle towards more mathematically verified software?



Taking advantage of advances in computer science:

- Moore's law
- improved programming languages & theoretical understanding
- better tools: interactive theorem provers

deep spec

National Science Foundation Expedition in Computing

the science of deep specification

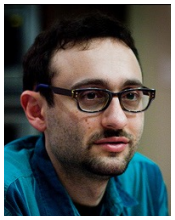

Andrew Appel


Lennart Beringer


Stephanie Weirich


Benjamin Pierce


Steve Zdancewic


Adam Chlipala


Zhong Shao

Princeton

Penn

MIT

Yale

www.deepspec.org

NSF

deepspec.org

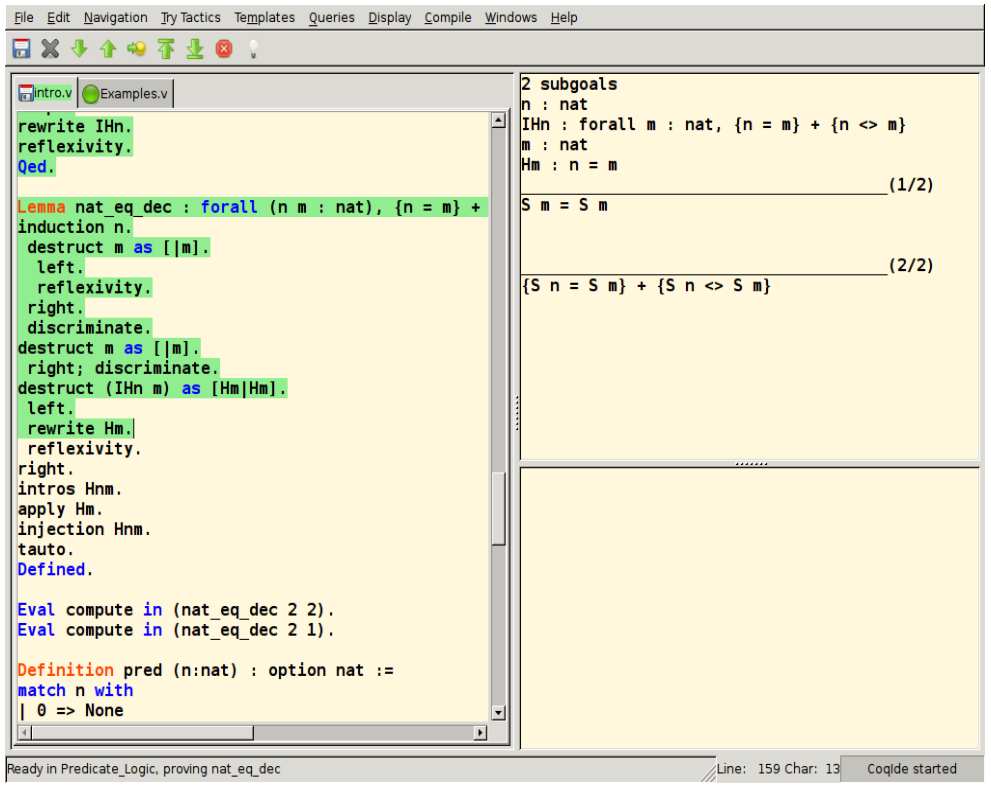
Deep Specifications

- ***Rich*** – expressive description, captures semantics
- ***Formal*** – mathematical, machine-checked
- ***2-Sided*** – tested from both sides
- ***Live*** – connected to real, executable code

Goal: Advance the reliability, safety, security, and cost-effectiveness of software (and hardware).

Unifying approach of DeepSpec project

- Encode software systems and correctness properties as **logical definitions**
- Use interactive tool (Coq) to develop a proof that the software is correct
  - No mistakes --- every step of the proof is checked!
  - Can automate proof development through domain specific scripts

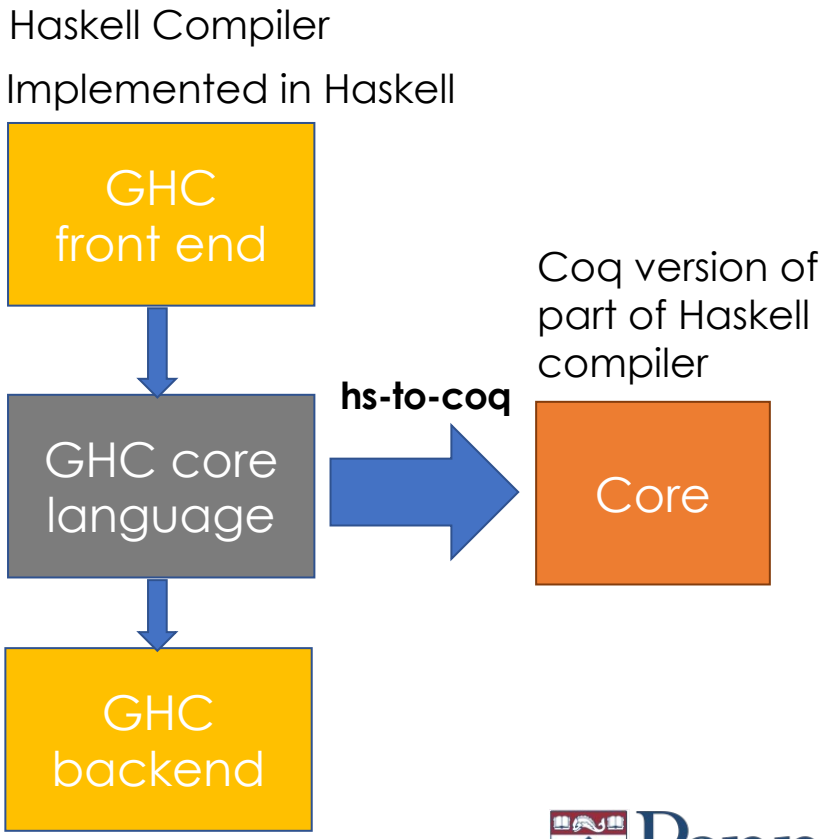


Interrelated research projects

Chlipala	Fiat/Kami	Verified Hardware (RISC-V)
Shao	CertiKOS	Verified Operating System / hypervisor
Appel	VST	Verified Software Toolchain for C
Zdancewic	Vellvm	Verified Low-level Compiler (LLVM)
Weirich	CoreSpec	Verified High-level Compiler (GHC core)
Pierce	QuickChick	Specification-based random testing
Multiple	DeepWeb	A verified web server built on DeepSpec

Verifying GHC Core

- **Goal**: develop a framework to prove the correctness of GHC core language optimizations and type system
- *Strategy*: Mechanically **translate** GHC Core implementation to Coq
- **hs-to-coq**: General purpose tool for reasoning about Haskell



Penn
University of Pennsylvania

hs-to-coq

Haskell

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (x : xs) = f x : map f xs
```

Coq
(hs-to-coq)

```
Definition map {a} {b} : (a -> b) -> list a -> list b
  := fix map arg_62__ arg_63__ :=
       match arg_62__, arg_63__ with
         | _, nil => nil
         | f, cons x xs => cons (f x) (map f xs)
       end.
```

Proof
(manual)

```
Lemma map_id : forall a (x:list a), map id x = x.
Proof.
   induction x. …
Qed.
```

hs-to-coq: highlights

- Base libraries (9k loc)
  - 40 separate modules distributed with compiler
  - Primitive data structures, interfaces and operations, like map
- Containers library (6k loc)
  - Widely used data structures (**Data.Set, Data.IntSet, Data.Map, Data.IntMap**)
- GHC Implementation (18k loc)
  - 49 modules (out of 327 modules total)
  - Translation axiomatizes/simplifies code in support of verification
  - *Complete proofs about Core language invariants for substitution and optimization passes*

https://github.com/plclub/hs-to-coq


Can we use logic to design better programming languages?

Dependent Haskell

- **Key idea**: Use features from logic to inspire the *design* of a programming language
- **Benefit**: Developers can create correctness proofs while they program
- **My role**: Collaboration with designers of Haskell programming language
- **Impact**: Community of Haskell programmers who use these features everyday

Stephanie Weirich
with Richard Eisenberg (Penn, Tweag/IO),
Simon Peyton Jones (MSR, Epic Games),
Joachim Breitner (Epic), Harley Eades III (Augusta),
Antoine Voizard, and Pritam Choudhury


PENN
PL
Penn
University of Pennsylvania
deep
spec