Reusable Tools for Formal Modeling of Machine Code

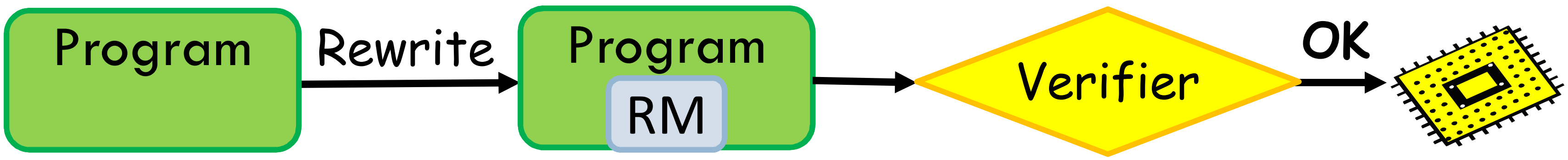
Gang Tan (Lehigh University)
Greg Morrisett (Harvard University)

Other contributors:
Joe Tassarotti   Edward Gan
Jean-Baptiste Tristan (Oracle Labs)

@ PiP; Jan 25th, 2014

Our Need for an x86 Machine Model

- **Certified Inlined-Reference Monitors (IRM)**
- IRM: Integrate a reference monitor into the code

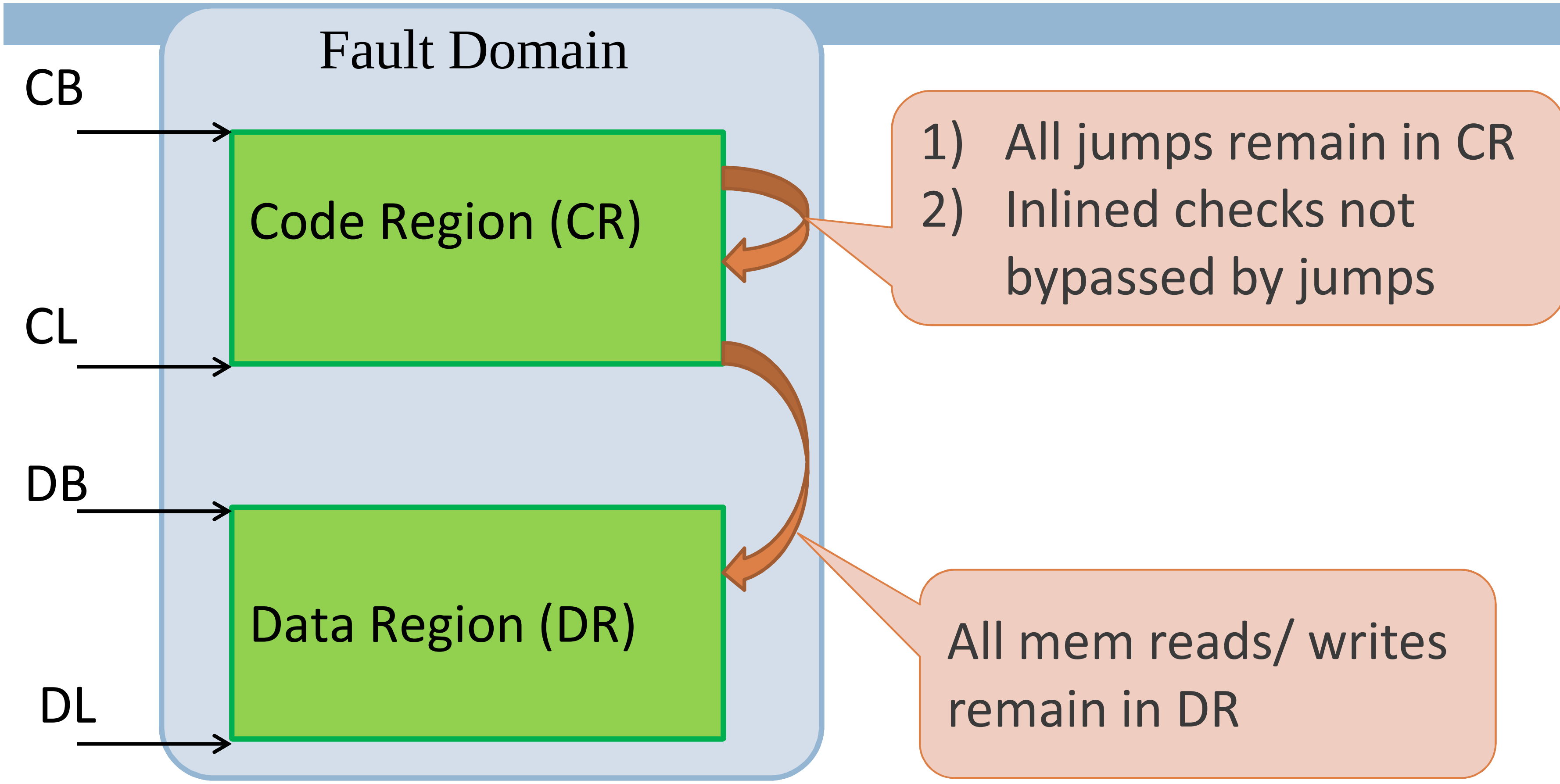


- Verifier: checking the monitor code is inlined correctly (so that the proper policy is enforced)
  - No need to trust the IRM-insertion phase

Software-Based Fault Isolation (SFI)

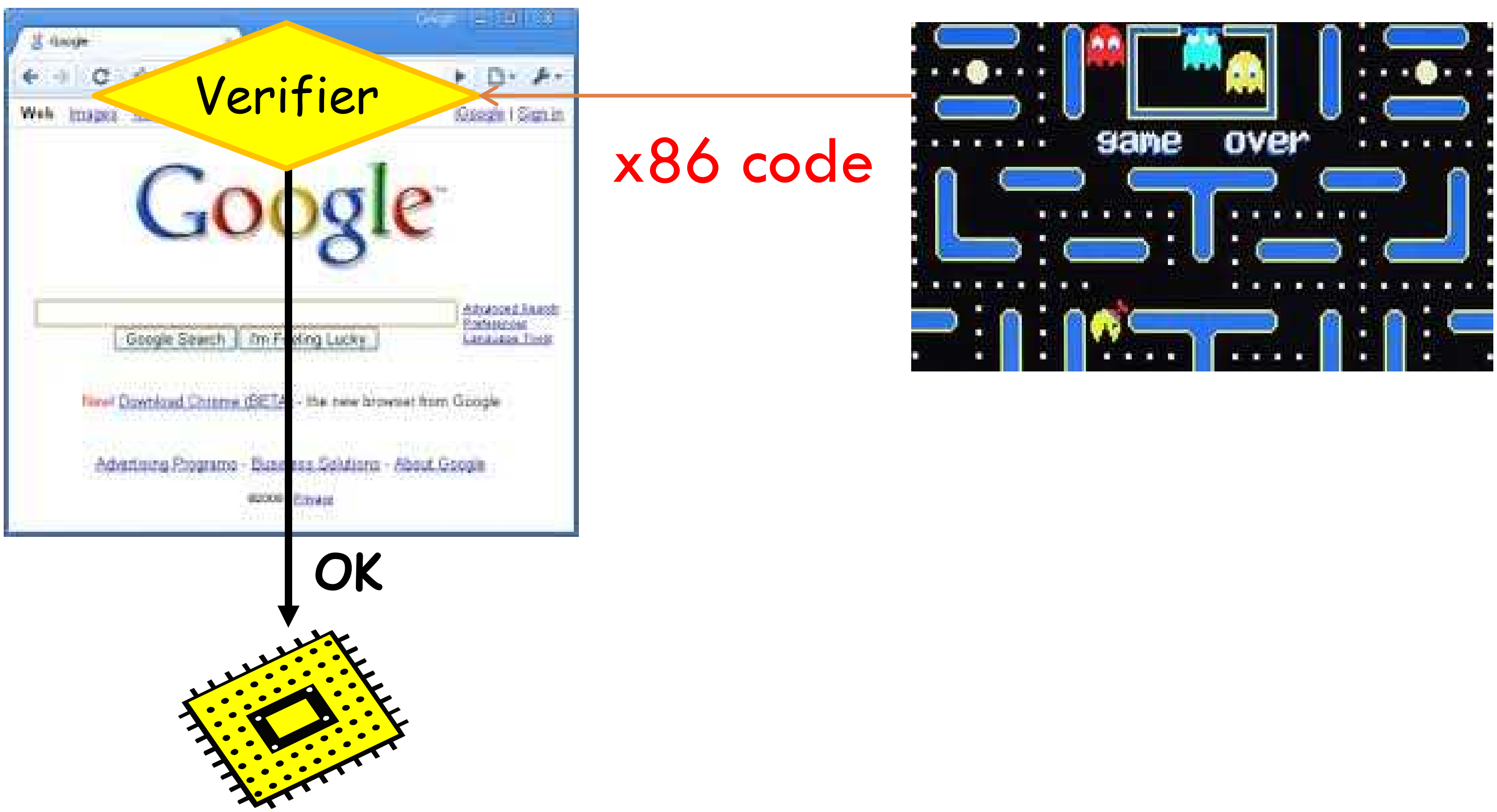
- A special kind of IRM
  - Isolate untrusted code into a “logical fault domain” within a process’s address space
- Wahbe, Luco et al (1991) for MIPS
  - McCamant & Morrisett (2006) extended it to CISC machines (x86)

The SFI Sandboxing Policy



Enforcing the policy: insert checks before unsafe instructions (memory operations, jumps, …)

The Native Client (NaCl) Verifier



One Critical Issue

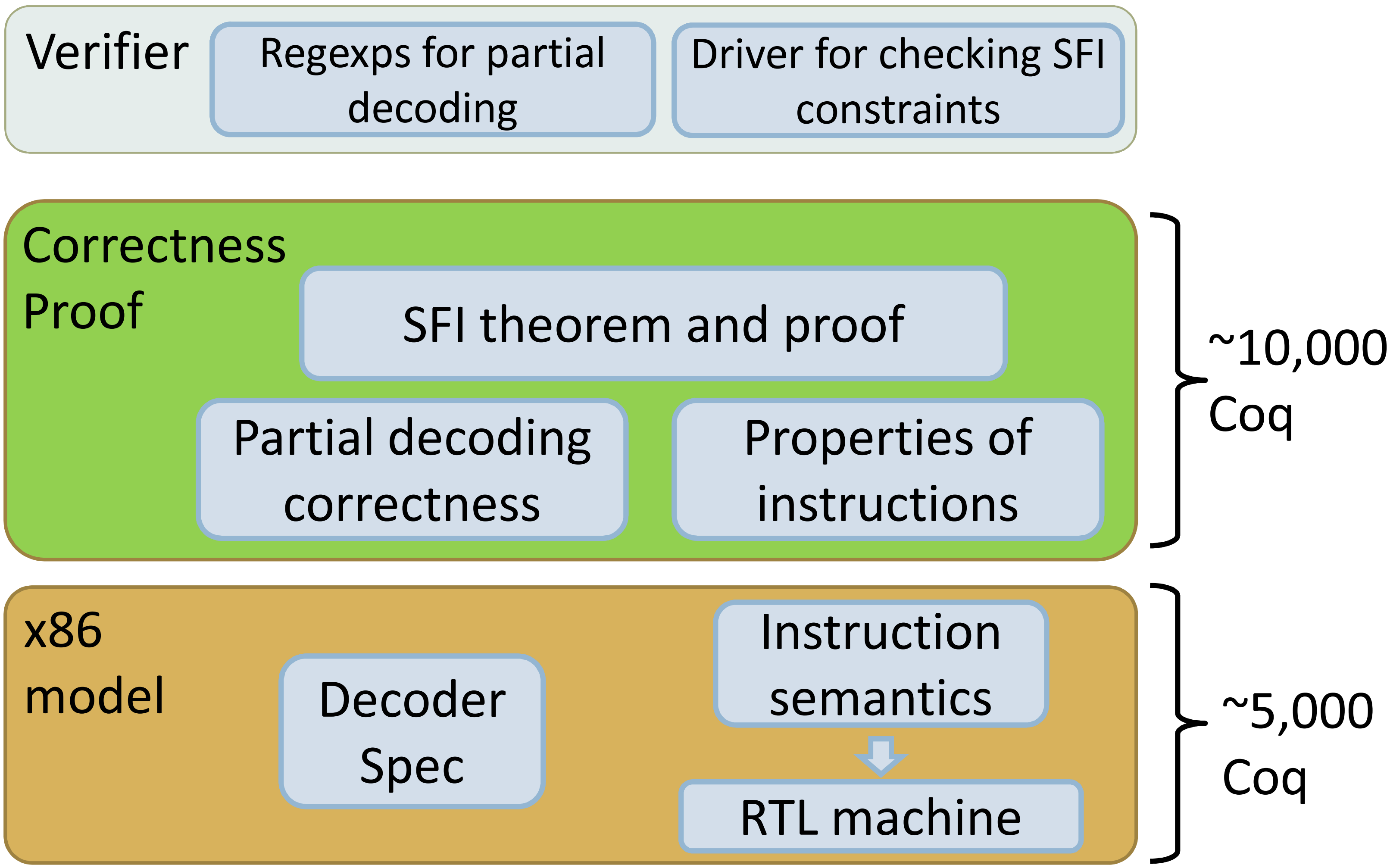
- A bug in the verifier could result in a security breach
  - NaCl's verifier: pile of C code with manually written partial decoders for x86 binaries
  - Google ran a security contest early on its NaCl verifier: bugs found!

- Goal: **a provably correct SFI verifier**
- Correctness theorem: if some binary passes the verifier, then the execution of the binary should obey the SFI policy

RockSalt Punchline

- **RockSalt**: a new verifier for x86-32 NaCl
  - [Morrisett, Tan, Tassarotti, Gan, Tristan PLDI 2012]
- **Smaller**
  - Google: 600 lines of C with manually written code for partial decoding
  - RockSalt: 80 lines of C + regexps for partial decoding
- **Faster**: on 200Kloc of C
  - Google's: 0.9s
  - RockSalt: 0.2s
- **Stronger**: (mostly) proven correct
  - The proof is machine checked in Coq

RockSalt Architecture



The Real Challenge

- Building a model of the x86
  - And to gain some confidence that it is correct!

Some Related Models

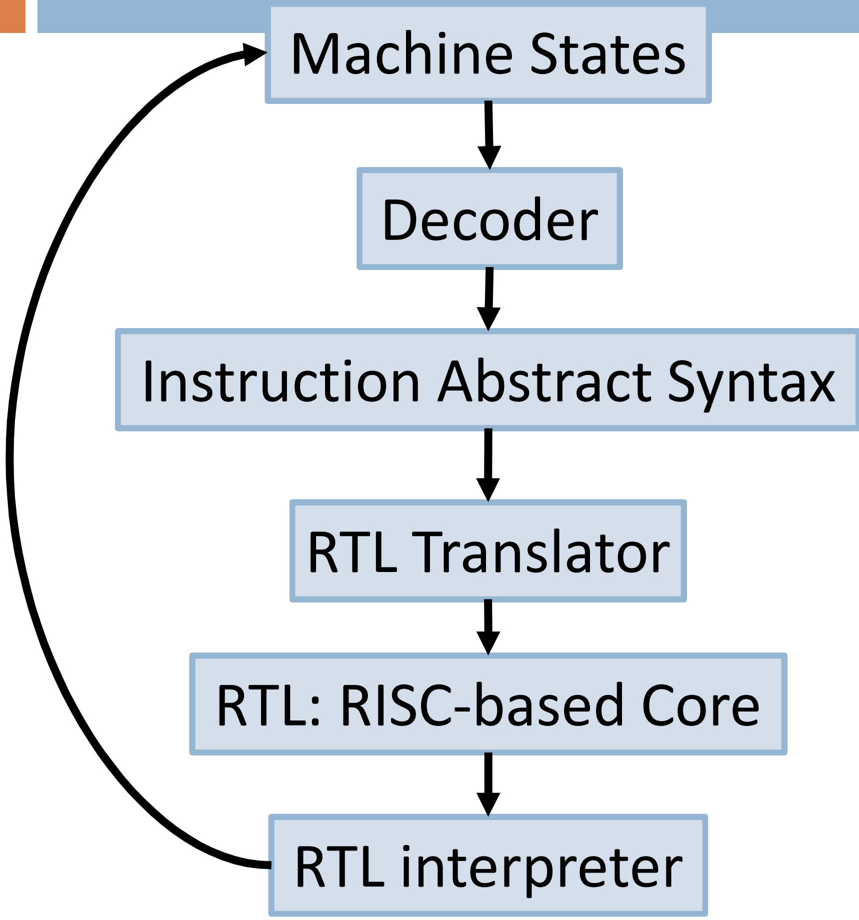
- CompCert's x86 model (Coq)
  - Actually an abstract machine with a notion of stack
  - Code is not explicitly represented as bits
- Y86 model (ACL2)
  - Tens of instructions, monolothic interpreter
  - But you can extract relatively efficient code for testing!
- Cambridge x86 work (HOL)
  - Inspired much of our design
  - Their focus was on modeling concurrency (TSO)
  - Semantics encoded with predicates (need symbolic computation)
- MSR [Benton and Kennedy]
- …

Our x86 Model

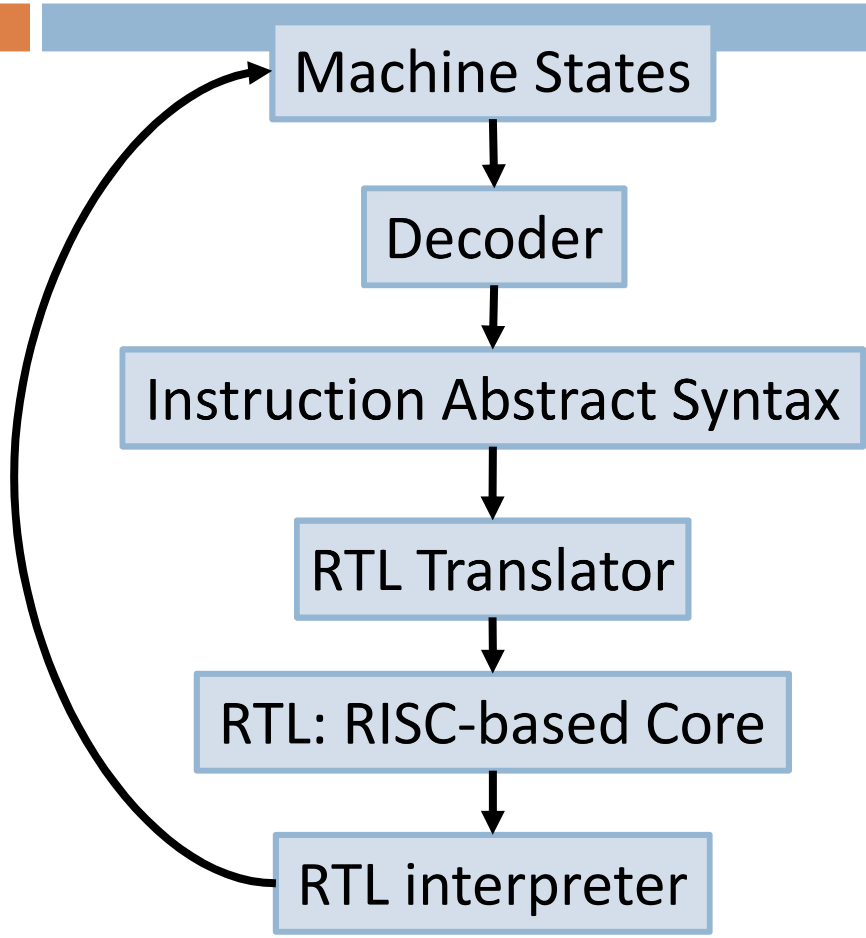
- Re-usable domain-specific languages to specify the semantics of machine models
  - We have modeled about 300 different x86 instructions (including all addressing modes and most of the prefixes)

1. Decoder specification language
   - Regular grammars for declarative specification of the decoder
2. Register Transfer Language (RTL)
   - Core RISC machine with simple operational semantics
   - Translate x86 instructions into RTLs

Our x86 Model in Coq

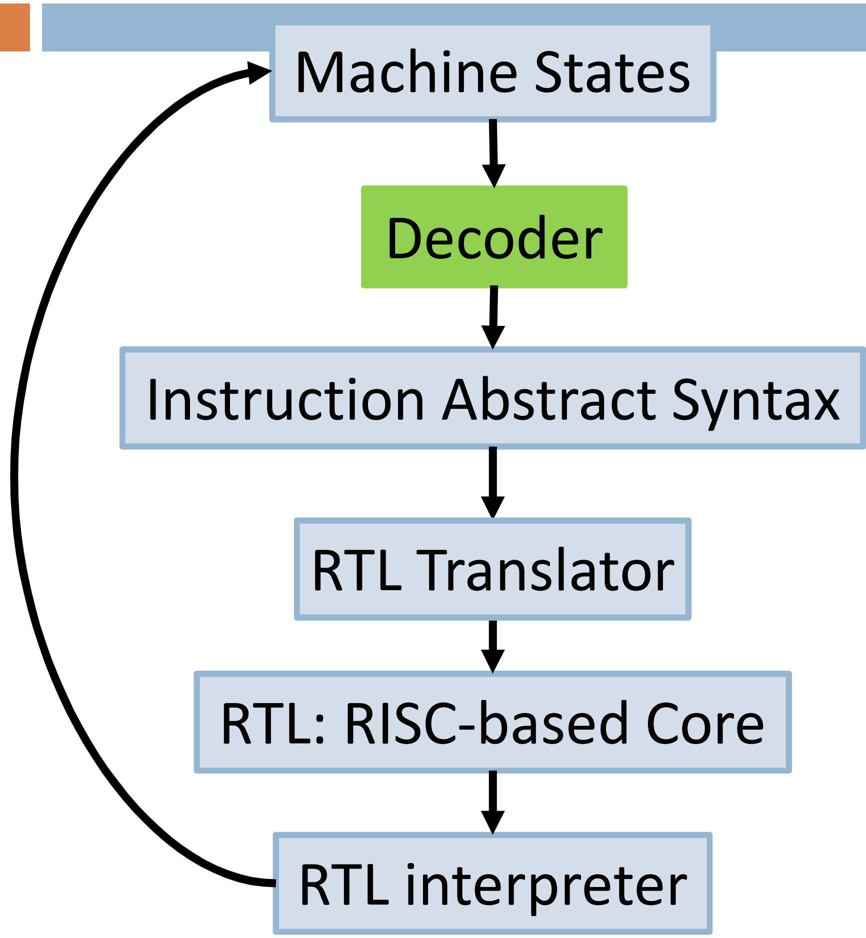


Our x86 Model in Coq



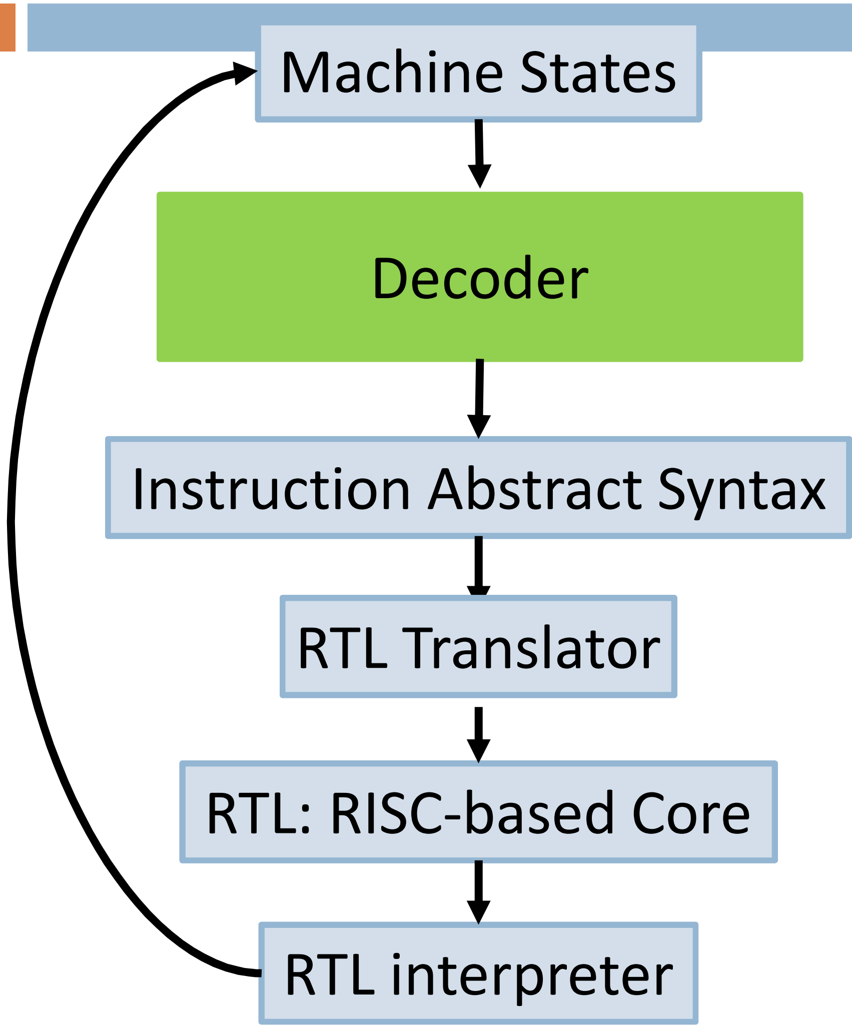
Importantly, we extract an x86 emulator in OCaml that we use for validation.

Our x86 Model in Coq



In this talk, we focus on the discussion of the decoder.

Our x86 Model in Coq



Turns out much harder than we thought!

Decoding for x86

- Incredibly difficult
  - Thousands of opcodes; many addressing modes
  - Prefix bytes override things like size of constants
  - The number of bytes for an instruction depends upon earlier bytes seen and can range from 1 to 15
- Plus, we need to reason about decoding
  - The SFI verifier uses partial decoders to recognize classes of instructions (e.g., indirect jumps)
  - Need to relate those partial decoders to the model's full decoder

Our Decoder Specification Language

- Type-indexed parsing combinators for regular grammars
  - Regular grammars: regular expressions + semantic actions
- Denotational semantics: so that we can reason about grammars
- An operational semantics (interpreter) via derivatives
  - Proven correct w.r.t the denotational semantics
- A parser generator (compiler) via efficient, table-based parsers
  - Also proven correct

Example Grammar for INC

INC – Increment by 1	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m

```
Definition INC_g : grammar instr :=
   "1111" $$ "111" $$ bit $ "11000" $$ reg
   @ (fun (w,r) => INC w (Reg_op r))
|| "0100" $$ "0" $$ reg
   @ (fun r => INC true (Reg_op r))
|| "1111" $$ "111" $$ bit $ (emodrm "000")
   @ (fun (w,op1) => INC w op1).
```

Decode pattern

Semantic action

Alternatives

Regular Grammar DSL


```
Inductive grammar : Type -> Type
| Char : char -> grammar char
| Eps : grammar unit
| Cat : ∀T U, grammar T -> grammar U -> grammar (T*U)
| Zero : ∀T, grammar T
| Alt : ∀T U, grammar T -> grammar U -> grammar (T+U)
| Star : ∀T, grammar T -> grammar (list T)
| Map : ∀T U, grammar T -> (T -> U) -> grammar U

Infix "+" := Alt.
Infix "$" := Cat.
Infix "@" := Map.
...
```

Indexed by types of semantic values returned by the grammar

Concatenation: returns a pair

Kleene star: returns a list

Apply a semantic action

Denotational Semantics


```
[[ ]] : grammar T -> (string * T) -> Prop.

[[Eps]] = {(nil, tt)}
[[Zero]] = {}
[[Char c]] = {(c::nil, c)}
[[Alt g₁ g₂]]={(s,inl v) | (s,v) in [[g₁]]} U
              {(s,inr v) | (s,v) in [[g₂]]}
[[Cat g₁ g₂]] =
         {(s₁++s₂,(v₁,v₂)) | (sᵢ,vᵢ) in [[gᵢ]]}
[[Star g]] = {(nil, nil)} U
         {(s,v) | s≠nil /\
                  s in [[Cat g (Star g)]]}
[[Map g f]] = {(s, f v) | (s,v) in [[g]]}
```

Typed Grammars as Specs

- The grammar language is very attractive for specification:
  - Typed “semantic actions”
  - Easy to build new combinators
  - Easy transliteration from the Intel manual
- Unlike Yacc/Flex/etc., has a good semantics:
  - Easy inversion principles
  - Good algebraic properties
    - e.g., easy to refactor or optimize grammar

Operational Semantics: Derivative-Based Parsing

- Old idea due to Brzozowski (1964), revitalized by Reppy et al., and extended by Might
- For a regexp `r` and char `c`, "`deriv c r`" returns a **residual regexp** that matches strings after matching c through r
  - E.g., `deriv c (cb*) = b*;`
    `deriv c (c*) = c*`
- For regular grammars, the semantics of derivatives is:

```
[[deriv c g]] =
  {(s,v) | (c::s,v) in [[g]]}
```

Derivatives for Grammars


```
deriv c (Char c) = Eps @ (fun _ => c)
deriv c (g1 + g2) =  deriv c g1 + deriv c g2
deriv c (g*) = (deriv c g $ g*) @ (::)
deriv c (g1 $ g2) =
  (deriv c g1 $ g2) || (null g1 $ deriv c g2)
deriv c (g @ f) = (deriv c g) @ f
deriv c _ = Zero
```

- Similar to Brzozowski's derivatives for regexps, but also taking semantic actions into account
- For efficiency, we must optimize the grammars as they are constructed. E.g.,

```
Eps $ g → g @ (fun x => (tt,x))
Zero $ g → Zero
```

Derivative-Based Parsing

Given a grammar g and an input string, a parser can be constructed by keep calculating derivatives:

```
parse g (c::s) := parse (deriv c g) s
parse g nil := extract g
```

```
[[extract g]] = {v | (nil,v) in [[g]]}
```

```
Correctness Theorem:
v ∈ (parse g cs) <-> (cs,v) in [[g]].
```

X86 Decoder by Computing Derivatives Online

- The parser just showed calculates derivatives online
  - Can be thought of as an interpreter
  - Was used in the first version of our x86 model described in PLDI 2012
- This worked okay, but the extracted OCaml x86 emulator was slow because of the decoding
  - Slowed down our model testing effort
  - Still tested over 10 million instruction instances but took over 60 hours

Speeding up the Decoder

- One idea: calculate a DFA table offline and use the table for parsing
- Brzozowski showed how to construct a DFA from a regular expression using derivatives
  - Calculate (deriv c r) for each c in the alphabet
  - Each unique (up to the optimizations) derivative corresponds to a state
  - Continue by calculating all reachable states' derivatives
  - Guaranteed this process will terminate!

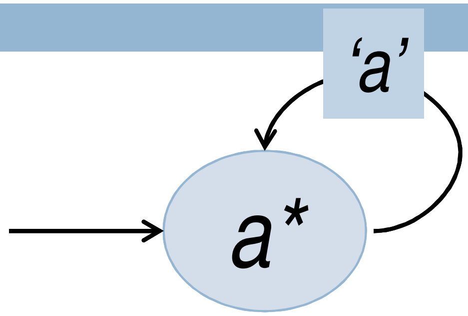
Bad News

- The derivatives for regular expressions are finite
- But as defined, we can have an unbounded number of derivatives for our typed, regular grammars

Breaking Finite Derivatives

For regular expressions:

```
deriv a (a*) = a*
```

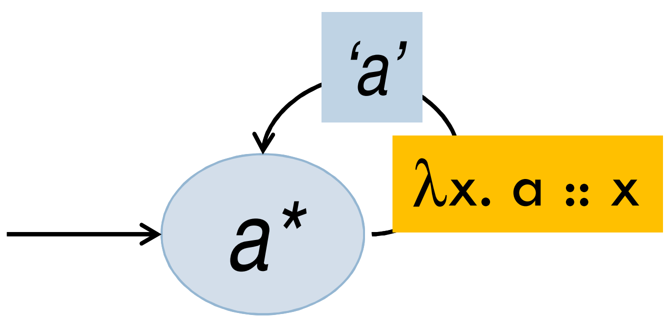


For regular grammars:

```
deriv a (a*) = a* @ (λx => a::x)
deriv a (a* @ (λx => a::x)) =
        a* @ (λx => a::a::x)
...
```

Our Solution: Use a Finite-State Transducer

- An edge is associated with
  - An input character
  - And an **output semantic action**: the action to apply after parsing the rest of the input



Input string: “aaa”

Output: three $\lambda x.\ a :: x$

Parsing result:

apply the three functions to nil to get [‘a’, ‘a’, ‘a’]

More Details

- Split the original grammar into a map-free grammar and a single semantic action that applies at the end

```
split: grammar T ->
        {a : ast_gram & (ast_tipe a) -> T}
```

- As we calculate derivatives, we continue to split
  - The states correspond to AST grammars
  - The edges are labeled input characters and output semantic actions

The Table-Driven Parser

- Lead to an easy, algebraic proof of correctness.
- We can also use the table to determine if the grammar is ambiguous.
  - Any terminal state (i.e., that accepts the empty string) shouldn't have alternatives.
- With more work on optimizations, we scaled up this technique to produce a table-driven x86 decoder
  - ~100-times faster than the previous decoder!

Lessons Learned when Building Models at Scale

- Certified parsing is critical and difficult
  - Windows: hundreds of parsers for different file formats; many security-critical bugs were found [GoDefRoiD et al. CACM 2012]
  - Future work: beyond regular grammars (e.g., CFGs)
    - [Barthwal and Norrish 09]: verified SLR parsing
    - [Jourdan, Pottier, and Leroy 12]: translation validation for LR(1) parsing
- Validation is absolutely essential
  - The parsing technique is aimed at building a faster model so we can do more testing/validation
- Re-use is crucial
  - Forced us to re-think how we do parsing and semantics

Future Directions for x86 Model

- Better validation
  - Cross-validation with other x86 models
- Extending the execution model
  - concurrency, system state, …
- Applications
  - CFI, XFI, TAL, …; CompCert
- Break the DSLs out of coq as first-class citizens
  - Connect with the Lem tool at Cambridge

Acknowledgements

- Support from NSF and Google Research