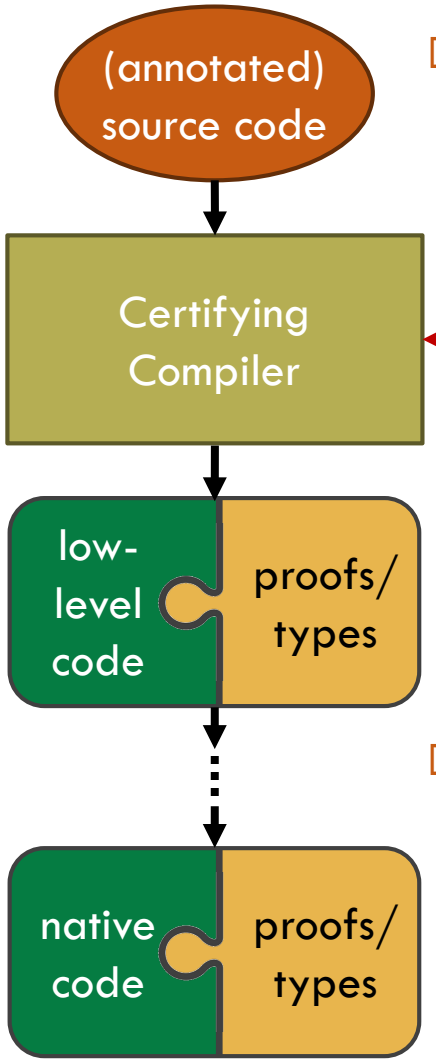
ISA-MODULAR, DEPENDENTLY TYPED VALIDATION OF NATIVE CODES

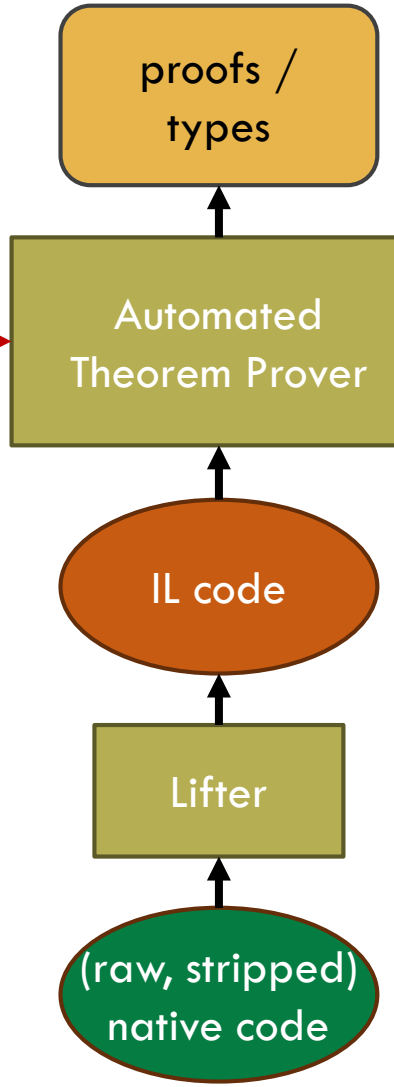
DR. KEVIN HAMLEN
EUGENE MCDERMOTT PROFESSOR
COMPUTER SCIENCE DEPARTMENT
CYBER SECURITY RESEARCH AND EDUCATION INSTITUTE
THE UNIVERSITY OF TEXAS AT DALLAS

Supported in part by:
ONR Awards N00014-14-1-0030 & N00014-17-1-2995,
NSF CAREER Award #1054629,
NSF Award 1513704
and an NSF I/UCRC Award from Lockheed Martin

Bottom-up Formal Methods



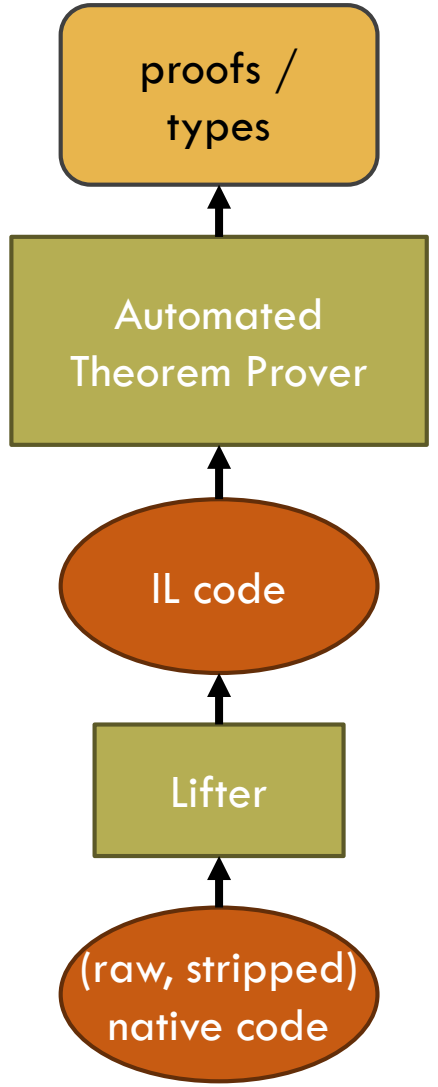
- Top-down FM
  - build high-assurance software from sources
  - proofs/types driven by source-level information
  - certifying, type-preserving compilation
- Examples: Coq/Gallina, CompCert, Isabelle/HOL, …



- Bottom-up FM
  - obtain high-assurance for source-free software
  - proofs/types driven by native-level information
  - ISA formal semantics specification & recovery
- Examples: XCAP [Yale Flint group], Bedrock [Chipala '11], RockSalt [Morrisett et al. '12]

Why Bottom-up?

- Prevalence of source-free code
  - hand-written assembly
  - native libraries derived from unverifiable source languages
  - native libraries generated by unverified/unknown tools
  - closed-source software products in mission-critical environments
- Formal verification of low-level tools
  - loaders, linkers, compilers, …
  - reverse-engineering tools (e.g., decompilers)
  - virtual machines, kernels, hypervisors, …
- Reasoning about native code **transformations**
  - binary-level control-flow integrity algorithms
  - security hotpatching
  - polymorphic malware defenses



Opaque Control-Flow Integrity

[Mohan et al., NDSS'15]

Description	Original Code	Rewritten Code (MPX-mode)	Rewritten Code (Legacy-mode)

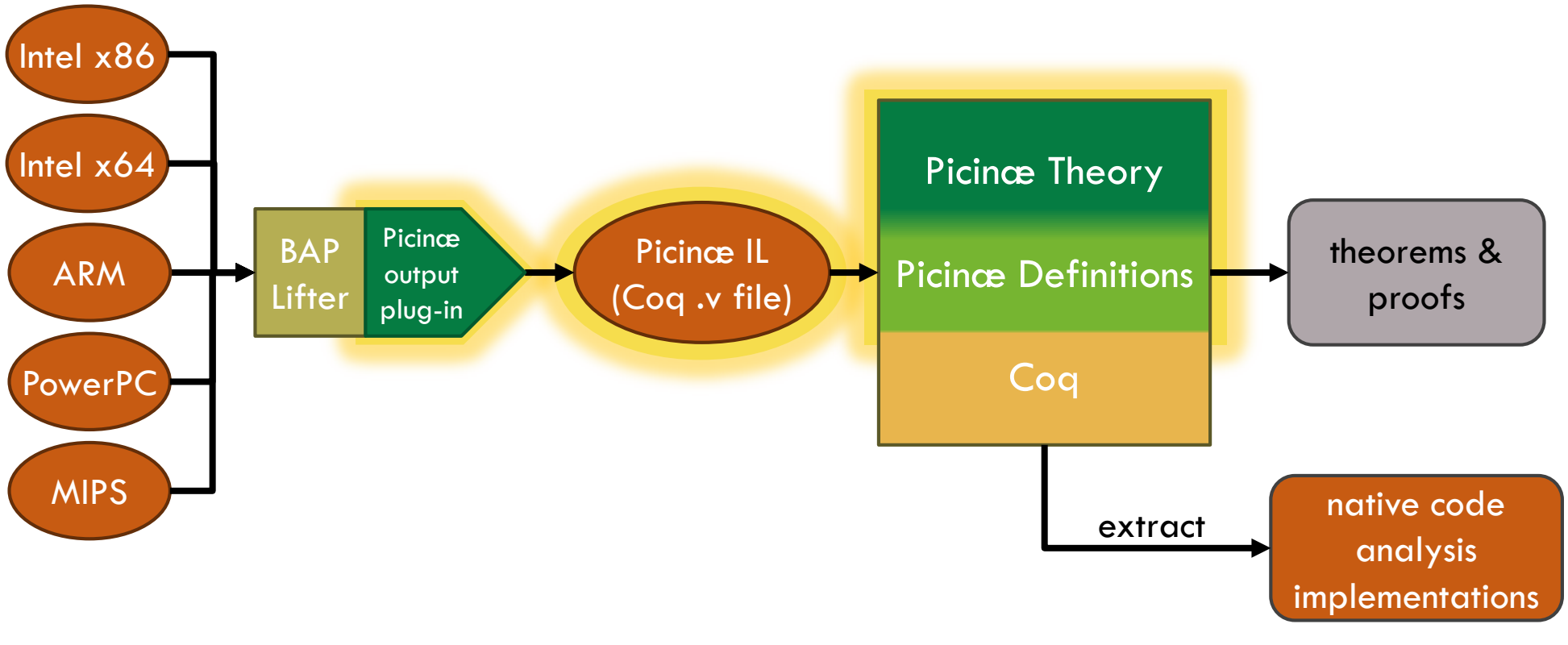
Indirect Branches	`call/jmp` $r/[m]$	`1: mov [esp-4], eax`
		`2: mov eax,` $r/[m]$
		`3: cmp byte ptr [eax], 0xF4`
		`4: cmovz eax, [eax+1]`
		— chunk boundary —
		`5: bndmov bnd1, gs:[`$branch\_id$`]`
		`6: bndcu bnd1, eax`
		`7: jmp 9`
		— chunk boundary —
		`8: xor eax, eax`
		`9: and al,` $align\_mask$
		`10: bndcl bnd1, eax`
		`11: xchg eax, [esp-4]`
		`12: call/jmp [esp-4]`
			`1: push ecx`
			`2: push eax`
			`3: mov eax,` $r/[m]$
			`4: cmp byte ptr [eax], 0xF4`
			`5: cmovz eax, [eax+1]`
			— chunk boundary —
			`6: mov ecx,` $branch\_id$
			`7: cmp eax, gs:[ecx]`
			`8: jb 10`
			`9: cmp gs:[ecx+4], eax`
			— boundary —
			`10: jbe abort`
			— chunk —
			`11: and al,` $align\_mask$
			`12: xchg eax, [esp]`
			`13: pop ecx`
			`14: pop ecx`
			`15: call/jmp [esp-8]`

Returns	`ret` $\langle n \rangle$	— chunk boundary —
		`1: xchg eax, [esp]`
		`2: and al,` $align\_mask$
		`3: bndmov bnd1, gs:[`$branch\_id$`]`
		`4: jmp 6`
		— chunk boundary —
		`5: xor eax, eax`
		`6: bndcu bnd1, eax`
		`7: bndcl bnd1, eax`
		`8: xchg eax, [esp]`
		`9: ret` $\langle n \rangle$
			— chunk boundary —
			`1: xchg eax, [esp]`
			`2: cmp eax, gs:[`$branch\_id$`]`
			`3: jb 9`
			— boundary —
			`4: and al,` $align\_mask$
			— chunk —
			`5: cmp eax, gs:[`$branch\_id + 4$`]`
			`6: jae 9`
			`7: xchg eax, [esp]`
			`8: ret` $\langle n \rangle$
			— chunk boundary —
			`9: jmp abort`

instruction misalignment!

PICNÆ: Platform In Coq for INstruction-level Analysis of Executables

- Bridging two technologies:
  - ISA Operational Semantics: Binary Analysis Platform (CMU BAP)
  - Program Proof Co-development: Coq Proof Assistant



Project Objectives

- Reason about *near arbitrary* native code without appeal to source-derived meta-data (e.g., invariants, CFGs, debug info, or disassembly maps)
- ISA-general, machine-checked theory
  - support for cross-ISA software analysis
- minimal trusted computing base
  - minimal semantic gap between native code and IL code
  - minimal base of trusted definitions
- approachable by Coq novices
  - ~250 lines of core definitions (functions + propositions)
  - ~300 theorems (searchable on as-needed basis)
- Reason about code *transformation* algorithms
  - native code is an unknown in the proof

Project Scope

- Out of Scope
  - Verifying/deriving the native-to-IL lifting
    - no comprehensive, machine-readable spec of many ISAs
    - Picinæ IL targetable by many lifting strategies
  - Modeling hardware details
    - Picinæ IL semantics resemble a Von Neumann / Harvard machine
    - no caching effects, no multicore modeling, etc. (yet?)
  - Unhandled hardware exception control-flows
    - Supportable, but significantly complicates other analyses
    - Possibly a future extension
- Supported features
  - instruction aliasing (e.g., misaligned instructions on Intel ISAs)
  - dynamic changes to page access permissions (readable/writable)
  - localized reasoning about memory (separation logic)
  - undefined processor elements (non-determinism)
  - self-modifying code (supported in theory; requires Coq port of instruction decoder)

IL Encoding

Programs

```
Definition program := addr -> option (N * stmt).
```

Statements (one per machine instruction)

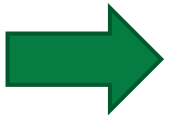
```
Inductive stmt :=
| Nop (* Do nothing. *)
| Move (v:var) (e:exp)  (* Assign to variable. *)
| Jmp (e:exp) (* Jump to a label/address. *)
| Exn (i:N) (* CPU Exception (numbered) *)
| Seq (q1 q2:stmt) (* sequence: q1 then q2 *)
| If (e:exp) (q1 q2:stmt) (* If e<>0 then q1 else q2 *)
| Rep (e:exp) (q:stmt) (* Repeat q for e iterations *).
```

Expressions (effect-free)

```
Inductive exp :=
| Var (v:var)
| Word (n:N) (w:bitwidth)
| Load (e1 e2:exp) (en:endianness) (w:bitwidth)
| Store (e1 e2 e3:exp) (en:endianness) (w:bitwidth)
| BinOp (b:binop_typ) (e1 e2:exp)
| UnOp (u:unop_typ) (e:exp)
| Cast (c:cast_typ) (w:bitwidth) (e:exp)
| Let (v:var) (e1 e2:exp)
| Unknown (w:bitwidth).
```

Lifting Example* (*fall-thru disassembly mode)


```
0x80000: xor eax, eax
0x80002: retl
```



```
Definition f : program := fun a => match a with
| 524288 => Some (2,
    Move R_EAX (Word 0 32) $;
    Move R_AF (Unknown 1) $;
    Move R_ZF (Word 1 1) $;
    Move R_PF (Word 1 1) $;
    Move R_OF (Word 0 1) $;
    Move R_CF (Word 0 1) $;
    Move R_SF (Word 0 1))
| 524290 => Some (1,
    Move (V_TEMP 2734) (Load (Var V_MEM32) (Var R_ESP) LittleE 4) $;
    Move R_ESP (BinOp OP_PLUS (Var R_ESP) (Word 4 32)) $;
    Jmp (Var (V_TEMP 2734)))
| _ => None
end.
```

Operational Semantics (18 rules)

Expressions:

$$\frac{}{\langle \textsf{Var}\ v, \sigma\rangle \Downarrow \sigma(v)} \qquad \frac{\langle e_1,\sigma\rangle \Downarrow m_w \quad \langle e_2,\sigma\rangle \Downarrow a_w \quad \mathcal{R}_\theta(\sigma, a_w, b)}{\langle \textsf{Load}\ e_1\ e_2\ en\ b, \sigma\rangle \Downarrow (\textsf{getmem}\ en\ b\ m\ a_w)_{8b}}$$

$$\frac{}{\langle \textsf{Word}\ n\ w, \sigma\rangle \Downarrow n_w} \qquad \frac{\langle e_1,\sigma\rangle \Downarrow m_w \quad \langle e_2,\sigma\rangle \Downarrow a_w \quad \langle e_3,\sigma\rangle \Downarrow n_{8b} \quad \mathcal{W}_\theta(\sigma, a_w, b)}{\langle \textsf{Store}\ e_1\ e_2\ e_3\ en\ b, \sigma\rangle \Downarrow (\textsf{setmem}\ en\ b\ m\ a_w\ n_{8b})_w}$$

$$\frac{\langle e_1,\sigma\rangle \Downarrow n \quad \langle e_2,\sigma\rangle \Downarrow n'}{\langle \textsf{BinOp}\ op\ n\ n', \sigma\rangle \Downarrow op\ n\ n'} \qquad \frac{\langle e,\sigma\rangle \Downarrow u \quad \langle e', \sigma[v := u]\rangle \Downarrow u'}{\langle \textsf{Let}\ v\ e\ e', \sigma\rangle \Downarrow u'} \qquad \frac{0 \le n < 2^w}{\langle \textsf{Unknown}\ w, \sigma\rangle \Downarrow n}$$

Programs:

$$\frac{}{\langle \theta, a, \sigma\rangle \rightsquigarrow_0 \langle \sigma, \downarrow_a\rangle}\textsc{Done} \qquad \frac{\langle p(a),\sigma\rangle \curvearrowright \langle \sigma', \uparrow_i\rangle}{\langle \theta, a, \sigma\rangle \rightsquigarrow_1 \langle \sigma', \uparrow_i\rangle}\textsc{Abort}$$

$$\frac{\langle p(a),\sigma\rangle \curvearrowright \langle \sigma', x\rangle \quad \langle \theta, \textsf{exit}_{a+|p(a)|}(x), \sigma'\rangle \rightsquigarrow_n \langle \sigma'', x'\rangle}{\langle \theta, a, \sigma\rangle \rightsquigarrow_{n+1} \langle \sigma'', x'\rangle}\textsc{Step}$$

$$\text{where } \textsf{exit}_a = \left(\downarrow \Rightarrow a \;\middle|\; \downarrow_{a'} \Rightarrow a'\right)$$

Statements:

$$\frac{}{\langle \textsf{Nop}, \sigma\rangle \curvearrowright \langle \sigma, \downarrow\rangle} \qquad \frac{\langle e,\sigma\rangle \Downarrow u}{\langle \textsf{Move}\ v\ e, \sigma\rangle \curvearrowright \langle \sigma[v := u], \downarrow\rangle}$$

$$\frac{\langle e,\sigma\rangle \Downarrow a}{\langle \textsf{Jmp}\ e, \sigma\rangle \curvearrowright \langle \sigma, \downarrow_a\rangle} \qquad \frac{}{\langle \textsf{Exn}\ i, \sigma\rangle \curvearrowright \langle \sigma, \uparrow_i\rangle}$$

$$\frac{\langle q_1,\sigma\rangle \curvearrowright \langle \sigma', x\rangle \quad x \neq \downarrow}{\langle q_1\ \textsf{\$;}\ q_2, \sigma\rangle \curvearrowright \langle \sigma', x\rangle}\textsc{Seq1} \qquad \frac{\langle q_1,\sigma\rangle \curvearrowright \langle \sigma', \downarrow\rangle \quad \langle q_2,\sigma'\rangle \curvearrowright \langle \sigma'', x\rangle}{\langle q_1\ \textsf{\$;}\ q_2, \sigma\rangle \curvearrowright \langle \sigma'', x\rangle}\textsc{Seq2}$$

$$\frac{\langle e,\sigma\rangle \Downarrow n \quad \langle n{\neq}0\ ?\ q_1 : q_2, \sigma\rangle \curvearrowright \langle \sigma', x\rangle}{\langle \textsf{If}\ e\ q_1\ q_2, \sigma\rangle \curvearrowright \langle \sigma', x\rangle} \qquad \frac{\langle e,\sigma\rangle \Downarrow n \quad \langle \overbrace{q\ \textsf{\$;}\ \cdots\ q}^{n}\ \textsf{\$; Nop}, \sigma\rangle \curvearrowright \langle \sigma', x\rangle}{\langle \textsf{Rep}\ e\ q, \sigma\rangle \curvearrowright \langle \sigma', x\rangle}$$

Picinæ Theory

- Functional Interpreter
  - Implements IL semantics as a deterministic function for faster symbolic interpretation (~300% faster than tactics, ~2s per machine instruction)
- Static Semantics
  - Proves type-soundness of lifted IL code
  - Implies basic well-formedness properties of register/memory values
- Inductive proof principles
  - Floyd-Hoare style inductive proofs of partial and total correctness
- Theory of two's complement
  - Converts IL expressions to Coq theories of N and Z
  - Facilitates validation of machine operations that conflate signed+unsigned ops
- Monotonicity & Frame Theorems
  - Proofs completed with partial info about cpu+program state hold extensionally
- Separation Logic
  - Deep embedding of Frame Axiom for localized memory reasoning about data structures

Example Theorem: Callee preserves stack ptr


```
1  Definition invariants (esp:N) (_:addr) (s:store) := Some (s R_ESP = Ⓓesp).
2  Definition postcondition (esp:N) (_:exit) (s:store) := s R_ESP = Ⓓ(esp ⊕ 4).
3  Definition invariant_set esp := invs (invariants esp) (postcondition esp).

4  Theorem f_preserves_esp s esp mem n s' x':
5      (ESP0: s R_ESP = Ⓓesp)              (* let esp be initial value of ESP *)
6      (MEM0: s V_MEM32 = Ⓜmem)            (* let mem be initial memory contents *)
7      (RET: f(mem Ⓓ[esp]) = None)         (* assume valid return address *)
8      (XP0: exec_prog h f 0 s n s' x'),   (* executing subroutine f implies … *)
9    all_invariants_satisfied (invariant_set esp f x' s').
```

Example Proof


```
Proof.
1   intros.
2   eapply prove_invs. exact XP0.  (* Use Hoare induction. *)

3   exact ESP0.  (* Prove pre-condition. *)

    (* Apply an earlier theorem to prove memory preservation. *)
4   intros. assert (MEM: s1 V_MEM32 = Ⓜmem).
5     rewrite <- MEM0. eapply f_preserves_memory. exact XP.
6   clear s MEM0 XP0 ESP0 XP.

7   shelve_cases 32 PRE. Unshelve.  (* Introduce one proof case per invariant point. *)

8   all: x86_step.  (* Invoke the symbolic interpreter. *)
9   all: solve [ reflexivity | assumption ].  (* Solve all the cases. *)
Qed.
```

ISA Modularity via Dependent Typing

- IL semantics implemented as a Coq Functor parameterized by an ISA specification

```
Inductive x86var :=
| V_MEM32 (* main memory *)
| R_AF | R_CF | R_DF | R_OF | R_PF | R_SF | R_ZF (* flags *)
| R_EAX | R_EBX | R_ECX | R_EDX | R_EDI | R_ESI (* general-purpose registers *)
...
| A_READ | A_WRITE (* page access permission bits *)
| V_TEMP (n:N) (* temporaries (introduced by lifter) *).

Module X86Arch <: Architecture.
  Module Var := Make_UDT MiniX86VarEq.
  Definition mem_bits := 8.
  Definition mem_readable s a := exists r, s A_READ = Some (VaM r 32) /\ r a <> 0.
  Definition mem_writable s a := exists w, s A_WRITE = Some (VaM w 32) /\ w a <> 0.
  Theorem mem_readable_mono: forall s1 s2 a, s1 ⊆ s2 -> mem_readable s1 a -> mem_readable s2 a.
  Proof. intros. destruct H0. eexists. split; [apply H|]; apply H0. Qed.
  Theorem mem_readable_mono: forall s1 s2 a, s1 ⊆ s2 -> mem_writable s1 a -> mem_writable s2 a.
  Proof. intros. destruct H0. eexists. split; [apply H|]; apply H0. Qed.
End X86Arch.
```

Case Study #1: Validation of x86 strlen (glibc)

- 67 shockingly difficult instructions
  - heavily optimized (e.g., loop unrolling, leverage of instruction side-effects, etc.)
  - reliance on legacy processor features (parity flag!)
  - obscure properties of bit arithmetic
    - Example: ((~x) xor (x-0x01010101)) & 0x01010100 = 0 iff x contains no zero-bytes (???)
- Proofs
  - strlen obeys architectural calling conventions (13 lines defs, 16 lines proofs)
  - strlen is memory-safe (5 lines defs, 20 lines proofs)
  - strlen is totally correct (10 lines defs, 225 lines main proof)
    - not counting 330 lines of supporting proofs about bit arithmetic (!)

Case Study #2: Code Transform Validation

- Proof that a binary-to-binary register reallocation algorithm is correct
  - takes *arbitrary* machine code as input
  - produces transformed machine code as output
- Proof statistics
  - 25-line algorithm definition in Gallina
  - ~100 lines of correctness proofs

Summary

- Picinæ bridges two important technologies:
  - ISA semantic-lifters (e.g., BAP)
  - automated theorem provers / proof assistants (Coq)
- Improvements over prior works
  - Prove properties of *near arbitrary* native code
    - not just the ISA subset produced by some particular compiler
  - No reliance on source semantics
    - Binary might have been generated by source-less tools (e.g., binary hotpatching, macro assembler)
    - Source semantics still usable to infer proof steps (e.g., invariants)
  - Suitable for verifying properties of binary code transforms
- Substantial theory
  - static semantics (progress, preservation), Floyd-Hoare induction, symbolic interpretation, separation logic, sign-unknown binary arithmetic, non-determinism, monotonic reasoning

Selected References

- E. Bauman, Z. Lin, and K.W. Hamlen. **Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics**. In *Proc. Network and Distributed Systems Security*, 2018.
- R. Wartell, Y. Zhou, K.W. Hamlen, and M. Kantarcioglu. **Shingled Graph Disassembly: Finding the Undecidable Path**. In *Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining*, 2014.
- V. Mohan, P. Larsen, S. Brunthaler, K.W. Hamlen, and M. Franz. **Opaque Control-Flow Integrity.** In *Proc. Network and Distributed Systems Security*, 2015.
- X. Leroy. "**Formal Verification of a Realistic Compiler**." Communications of the ACM 52(7):107-115, 2009.
- Z. Ni, D. Yu, and Z. Shao. "**Using XCAP to Certify Realistic System Code: Machine Context Management**." In *Proc. Int. Conf. Theorem Proving in Higher Order Logics*, 2007.
- A. Chipala. "**Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic**." In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2011.
- G. Morrisett, G. Tan, J. Tassarotti, J.B. Tristan, and E. Gan. "**RockSalt: Better, Faster, Stronger SFI for the x86**." ACM Sigplan Notices 47(6):395-404, 2012.

THANK YOU!

FINAL SECOND:

FEATURE
IDENTIFICATION,
NEUTRALIZATION, &
AUTOMATED DE-
LAYERING FOR
SECURING **C**ODE **O**N **D**EMAND

ONR Award N00014-17-1-2995

Dr. Kevin Hamlen
Eugene McDermott Professor
Computer Science Department
Cyber Security Research and Education Institute
The University of Texas at Dallas

Dr. Zhiqiang Lin
Associate Professor
Computer Science & Engineering
Ohio State University

Additional support from:
ONR N000141410030,
AFOSR FA9550-08-1-0044 (YIP) & FA9550-14-1-0173,
NSA Award H98230-15-1-0271,
NSF #1054629 (CAREER) & #1513704, and NSF I/UCRC
Awards from Raytheon Company and Lockheed-Martin