# The Good, The Bad, and The Binary

## Formal Timing and Correctness of Binary Code

Charles Averill

The University of Texas at Dallas & Dartmouth College

August 2024

# The Problem

- Code has vulnerabilities

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited
- We can't just run the code to find all vulnerabilities, code is too big

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited
- We can't just run the code to find all vulnerabilities, code is too big
- We have a lot of ways to heuristically catch and patch large classes of bugs (e.g. null pointer dereferences) **statically**

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited
- We can't just run the code to find all vulnerabilities, code is too big
- We have a lot of ways to heuristically catch and patch large classes of bugs (e.g. null pointer dereferences) **statically**
- What if the bug patcher misses cases, adds bugs, fails to patch correctly, etc.?

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited
- We can't just run the code to find all vulnerabilities, code is too big
- We have a lot of ways to heuristically catch and patch large classes of bugs (e.g. null pointer dereferences) **statically**
- What if the bug patcher misses cases, adds bugs, fails to patch correctly, etc.?
- What if the code being patched is so mission-critical that there can be no doubt that the code is bug-free?

# The Problem

- Code has vulnerabilities
- We want to be able to catch vulnerabilities before they can be exploited
- We can't just run the code to find all vulnerabilities, code is too big
- We have a lot of ways to heuristically catch and patch large classes of bugs (e.g. null pointer dereferences) **statically**
- What if the bug patcher misses cases, adds bugs, fails to patch correctly, etc.?
- What if the code being patched is so mission-critical that there can be no doubt that the code is bug-free?
- Dealing with (possibly-handwritten) binary code exacerbates each of these consideration

# The Solution

- Build formal models of software
- Formally state your correctness specification
- Write a proof that your software meets your specification
- Employ a machine to check that your proof is correct

# The Challenges

- Build formal models of software
  - Which models do you formalize? Languages? Interpreters/Compilers? CPU semantics? Hardware?
  - Some languages/compilers/CPUs have no formal specification, many more have no machine-readable formal specification
  - The proposition that the model matches the implementation is usually an assumption and unproven
- Formally state your correctness specification
- Write a proof that your software meets your specification
- Employ a machine to check that your proof is correct

# The Challenges

- Build formal models of software
- Formally state your correctness specification
  - Correctness specifications are difficult to write and may be miscommunicated
  - May contain many layers of abstraction to represent high-level concepts in a formal environment
- Write a proof that your software meets your specification
- Employ a machine to check that your proof is correct

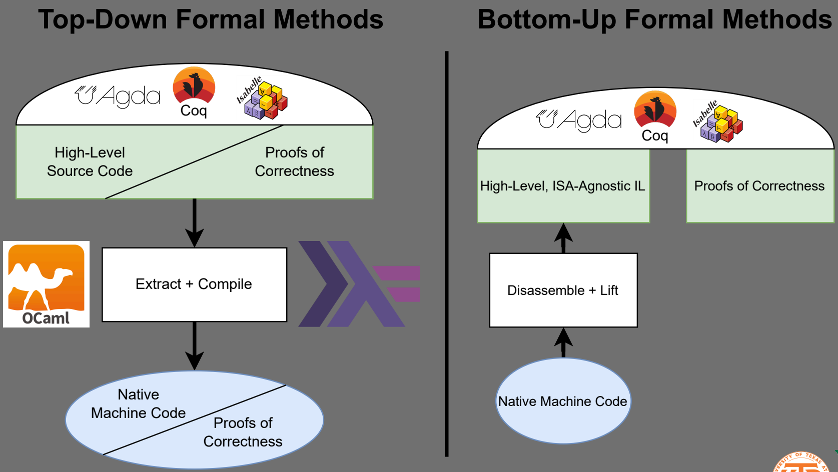
# The Challenges

- Build formal models of software
- Formally state your correctness specification
- Write a proof that your software meets your specification
    - Writing formal proofs requires learning a proof language
    - Proofs about code must handle deep concepts such as decidability and termination
    - Reasoning about loops requires non-trivial insight learned by experience
- Employ a machine to check that your proof is correct

## The Challenges

- Build formal models of software
- Formally state your correctness specification
- Write a proof that your software meets your specification
- Employ a machine to check that your proof is correct
    - Can we trust the machine to properly check?

# Bottom-Up vs. Top-Down



# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting

# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting
- Incompatible with low-level tasks

# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting
- Incompatible with low-level tasks
- Best case: a high-level software project with formal methods planned from the start

# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting
- Incompatible with low-level tasks
- Best case: a high-level software project with formal methods planned from the start

Bottom-up:

- Can be applied to all binary code

# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting
- Incompatible with low-level tasks
- Best case: a high-level software project with formal methods planned from the start

Bottom-up:

- Can be applied to all binary code
- Loses nice behavior of high-level languages

# Bottom-Up vs. Top-Down

Top-down:

- Unfit for security retrofitting
- Incompatible with low-level tasks
- Best case: a high-level software project with formal methods planned from the start

Bottom-up:

- Can be applied to all binary code
- Loses nice behavior of high-level languages
- Only suitable pathway towards verifying source-free mission-critical code

# Trusted Computing Base

All formal approaches have a set of assumptions upon which all further reasoning is built.

Picinæ's assumptions:

- Accuracy of user-provided correctness specifications - these are always assumptions

# Trusted Computing Base

All formal approaches have a set of assumptions upon which all further reasoning is built.

Picinæ's assumptions:

- Accuracy of user-provided correctness specifications - these are always assumptions
- Correctness of binary lifter - systematically tested against real hardware [BAP - CAV 2011]

# Trusted Computing Base

All formal approaches have a set of assumptions upon which all further reasoning is built.

Picinæ's assumptions:

- Accuracy of user-provided correctness specifications - these are always assumptions
- Correctness of binary lifter - systematically tested against real hardware [BAP - CAV 2011]
- Correctness of proof checker and underlying logic - systematically verified by hand and by formal approaches [MetaCoq - POPL 2018]

# Example

### RISC-V Assembly

```
        ori     t2,zero,1
        andi    t3,t3,0
add:
        beq     t0,t3,20 <end>
        addi    t1,t1,1
        sub     t0,t0,t2
        beq     t3,t3,10 <add>
end:
```

### RISC-V Binary

```
0x00106393000e7e13
  01c2886300130313
  407282b3ffce0ae3
```

### Picinæ IL

```
Move R_T2 (BinOp OP_OR (Word 0 32) (Word 1 32))
Move R_T3 (BinOp OP_AND (Var R_T3) (Word 0 32))
If (BinOp OP_EQ (Var R_T0) (Var R_T3)) (Jmp (Word 32 32)) Nop
Move R_T1 (BinOp OP_PLUS (Var R_T1) (Word 1 32))
Move R_T0 (BinOp OP_MINUS (Var R_T0) (Var R_T2))
If (BinOp OP_EQ (Var R_T3) (Var R_T3)) (Jmp (Word 16 32)) Nop
```

# Example

```
Definition postcondition (s : store) (x y : N) :=
    (* Tie register value to output of Coq binnat add function *)
    s R_T1 = (D)(x (+) y).

Definition addloop_correctness_invs (_ : store) (p : addr)
        (x y : N) (t:trace) := (* Proof of these invariants is ~40 LOC *)
    match t with (Addr a, s) :: _ => match a with
            (* Tie the contents of a CPU state to Coq variables x and y *)
        | 0x8  => Some (s R_T0 = (D)x /\ s R_T1 = (D)y)
        | 0x10 => Some (exists t0 t1,
            (* Same as above *)
            s R_T0 = (D)t0 /\ s R_T1 = (D)t1 /\
            s R_T2 = (D)1 /\ s R_T3 = (D)0 /\
            (* Invariant: the sum of R0 and R1 is always equal to
               the sum of x and y *)
            t0 (+) t1 = x (+) y)
            (* When the program exits, this postcondition is satisfied *)
        | 0x20 => Some (postcondition s x y)
        | _ => None end
    | _ => None
    end.
```

# The Point

- We can write arbitrary proofs of correctness for binary code

# The Point

- We can write arbitrary proofs of correctness for binary code
- This is now possible because of the advent of machine-readable specification

# The Point

- We can write arbitrary proofs of correctness for binary code
- This is now possible because of the advent of machine-readable specification
- Difficulties are stating specifications and translating arguments

# There's More

Forget correctness, Coq is expressive enough to represent arbitrary properties about these lifted structures:

```
(* Assume we have some function that maps instructions to execution times *)
Axiom time_of_addr (s : state) (a : addr) : N.

(* We can define a function that encodes the execution time of a trace,
   a.k.a. a list of program states created as a program executes *)
Definition cycle_count_of_trace (t : trace) : N :=
    List.fold_left N.add (List.map
        (fun '(e, s) => match e with
            | Addr a => time_of_addr s a
            | Raise n => max32
            end) t) 0.

(* Worst-case number of cycles addloop can take to execute on the
   NEORV32 RISC-V processor. `x` is our first operand! *)
Definition addloop_timing_postcondition (t : trace) (x : N) :=
    cycle_count_of_trace t = 9 + (ML - 1) + x * (12 + (ML - 1)).
```

# Timing Approaches

Sometimes we want to know how long code takes to run.

Big-O approach satisfactory for:

- Algorithm design

# Timing Approaches

Sometimes we want to know how long code takes to run.

Big-O approach satisfactory for:

- Algorithm design
- High-level optimization

# Timing Approaches

Sometimes we want to know how long code takes to run.

Big-O approach satisfactory for:

- Algorithm design
- High-level optimization

Some tasks require a more concrete approach:

- Constant-time cryptography

# Timing Approaches

Sometimes we want to know how long code takes to run.

Big-O approach satisfactory for:

- Algorithm design
- High-level optimization

Some tasks require a more concrete approach:

- Constant-time cryptography
- Real-time-constrained code

# Summer Work

As of a few weeks ago, Picinæ can encode concrete timing properties!

Goals:

- Show that RTOS code does not violate timing constraints after a program transformation (e.g. CFI injection)

# Summer Work

As of a few weeks ago, Picinæ can encode concrete timing properties!

Goals:

- Show that RTOS code does not violate timing constraints after a program transformation (e.g. CFI injection)
- Formally prove various crypto ciphers don't contain "timing leaks"

# Summer Work

As of a few weeks ago, Picinæ can encode concrete timing properties!

Goals:

- Show that RTOS code does not violate timing constraints after a program transformation (e.g. CFI injection)
- Formally prove various crypto ciphers don't contain "timing leaks"
- Automate writing invariants (trivial for `downto`-style loops) and timing proofs (tougher but promising)

# Summer Work

As of a few weeks ago, Picinæ can encode concrete timing properties!

Goals:

- Show that RTOS code does not violate timing constraints after a program transformation (e.g. CFI injection)
- Formally prove various crypto ciphers don't contain "timing leaks"
- Automate writing invariants (trivial for `downto`-style loops) and timing proofs (tougher but promising)
- Introduce system to people in different security fields - let's work together and formalize some cool binary analysis techniques!

# Summer Work

As of a few weeks ago, Picinæ can encode concrete timing properties!

Goals:

- Show that RTOS code does not violate timing constraints after a program transformation (e.g. CFI injection)
- Formally prove various crypto ciphers don't contain "timing leaks"
- Automate writing invariants (trivial for `downto`-style loops) and timing proofs (tougher but promising)
- Introduce system to people in different security fields - let's work together and formalize some cool binary analysis techniques!

A very cool, unforseen result of this research: timing proofs utilize all of the same machinery as correctness proofs, but are vastly simpler to write. Our research team will likely onboard new Picinæ users starting with timing proofs first.