Research Faculty Summit 2018

Systems | Fueling future disruptions
How Formal-Methods Adoption Should Drive Changes to System Designs

Adam Chlipala
Associate Professor, MIT CSAIL
“Mechanized, end-to-end proofs of functional correctness”
Mechanized proofs

Real System (source code)

Specification (source code)

Proof (source code)

(e.g., Coq proof assistant, which we use)

Proof Checker (algorithm)
Proofs of **Functional Correctness**

- **Real System** (source code)
- **Specification**
  - no segfaults
  - outputs right answer

---

*Research Faculty Summit 2018*

*Systems | Fueling future disruptions*
End-to-End Proofs

Layer 1

Layer 2

Layer 3

Whole-System Specification

Layers Proved Modularly
Is it *fundamental* that systems hackers need to spend their time writing intricate, bug-prone, low-level code?
d0 = r0 * 2;
d1 = r1 * 2;
d2 = r2 * 2 * 19;
d419 = r4 * 19;
d4 = d419 * 2;

t[0] = (((uint128_t) r0) * r0 + (((uint128_t) d4) * r1 + (((uint128_t) d2) * (r3 ))));
t[1] = (((uint128_t) d0) * r1 + (((uint128_t) d4) * r2 + (((uint128_t) r3) * (r3 * 19)));
t[2] = (((uint128_t) d0) * r2 + (((uint128_t) r1) * r1 + (((uint128_t) d4) * (r3 )));
t[3] = (((uint128_t) d0) * r3 + (((uint128_t) d1) * r2 + (((uint128_t) r4) * (d419 )));
t[4] = (((uint128_t) d0) * r4 + (((uint128_t) d1) * r3 + (((uint128_t) r2) * (r2 )));

r0 = (limb)t[0] & 0x7fffffffffffff; c = (limb)(t[0] >> 51);
t[1] += c; r1 = (limb)t[1] & 0x7fffffffffffff; c = (limb)(t[1] >> 51);
t[2] += c; r2 = (limb)t[2] & 0x7fffffffffffff; c = (limb)(t[2] >> 51);
t[3] += c; r3 = (limb)t[3] & 0x7fffffffffffff; c = (limb)(t[3] >> 51);
t[4] += c; r4 = (limb)t[4] & 0x7fffffffffffff; c = (limb)(t[4] >> 51);
r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7fffffffffffff;
r1 += c; c = r1 >> 51; r1 = r1 & 0x7fffffffffffff;
r2 += c;
But the experts know how to do all this, right?

Labor-intensive adaptation, with each combination taking at least several days for an expert.

And by the way, sometimes there are serious bugs.
“Knowledge of the secret key is needed to produce a signature in polynomial time.”

\[ y^2 = x^3 - x + 1 \]
Correct-by-Construction Cryptography

Mathematical algorithm

point = (x, y)

High-level modular arithmetic

x = x₀, x₁, ..., xₙ (mathematical integers)

Optimized point format

point = (x, y, z, t)

Proved abstraction relation

Proved abstraction relation

Low-level code

specialized low-level code
(assumes fixed set of integer sizes)

classic verification of functional programs

classic verification of functional programs

compile-time code specialization

compiler verification
Fiat Cryptography

Joint work with Andres Erbsen, Jade Philipoom, Jason Gross, and Robert Sloan
Implementation of Multiplication?

Just compute all the cross terms.

E.g., \([(a, x), (b, y)] \times [(c, u), (d, v)]\)

\(\rightarrow [(ac, xu), (ad, xv), (bc, yu), (bd, yv)]\)

```
Definition mul (p q:list (Z*Z)) : list (Z*Z) :=
  flat_map (fun t =>
    map (fun t' =>
      (fst t * fst t', (snd t * snd t')%RT))
    q) p.

Lemma eval_mul p q :
  eval (mul p q) = eval p * eval q.
```
Definition mulmod \{n\} (a, b : tuple Z n) : tuple Z n
:= let a_a := to_associational a in
    let b_a := to_associational b in
    let ab_a := Associational.mul a_a b_a in
    let abm_a := Associational.reduce s c ab_a in
    from_associational n abm_a.

Convert from fixed base system to simpler custom form at start of execution.

Compute in custom form.

Convert back at end.
Time for Some Partial Evaluation

- Digit Bitwidths
  - s Digits × t Digits
  - Multiply
    - Specialize
      - In Coq: just partially applying a curried function
    - s × t Digits
  - Multiply
    - Reduce
      - In Coq: just calling a standard term-reduction tactic
    - s × t Digits
## Performance on Curve25519

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>amd64-64, asm</td>
<td>151586</td>
</tr>
<tr>
<td><em>this work B</em>, 64-bit</td>
<td>152195</td>
</tr>
<tr>
<td>sandy2x, asm</td>
<td>154313</td>
</tr>
<tr>
<td>hacl-star, 64-bit</td>
<td>154982</td>
</tr>
<tr>
<td>donna64, 64-bit C</td>
<td>168502</td>
</tr>
<tr>
<td><em>this work A</em>, 64-bit</td>
<td>174637</td>
</tr>
<tr>
<td><em>this work</em>, 32-bit</td>
<td>310585</td>
</tr>
<tr>
<td>donna32, 32-bit C</td>
<td>529812</td>
</tr>
</tbody>
</table>
Performance on Many Curves

64-Bit Field Arithmetic Benchmarks

32-Bit Field Arithmetic Benchmarks
And We're in Chrome Now!

via the BoringSSL library

for Curve25519 & P256

Coming soon, pending internship success: P384
The Big Tradeoff

Is it *fundamental* that systems hackers need to spend their time writing intricate, bug-prone, low-level code?

Is it *fundamental* that abstractions bring runtime performance costs?
A General Schema for Goals of Systems SW/HW?

Real, optimized system

Specification?
Going All-In with Compile-Time Verification

- **Goal:** platform for efficient execution of functional programs, written in high-level notation so simple that auditing catches bugs well

- **Proof-Carrying Code:** no code (SW or HW) allowed on the system, in any digital component, without *proof of functional correctness*.

- **End-to-End Proofs:** all proofs connected together in a proved way, for a small TCB consisting of proof checker, plus semantics of hardware description language (~1000 lines?) and applications and system API (~1000 lines?).

- **No Runtime Enforcement of Isolation** (it's all in the proofs.)
Simplifying the Runtime Story

- **Functional code (spec)**
- **C-like code**
- **Machine code**
- **Processors**
- **Memory System**

Uses **object capabilities** and other patterns that bring security and isolation by construction.

Compiler analysis **infrers object lifetimes**

**No type system!** Expose memory directly. Fixed type systems are vestigial w/ program proof.

Compiler computes **worst-case running time,**

Thanks to proved characterizations of functions, knows which handlers need which objects. Moves objects into CPU caches preemptively, providing a clean **transactions** view to SW. No more **weak memory!**
In Summary....

• Surprisingly many hard systems challenges go away when we commit to requiring functional-correctness proofs of all installed SW.
• That kind of regime is more practical than folks would assume if they've held onto 20th-century perspectives!
• Fun question to leave you with: for various important domains, what would be the dollar cost of rewriting all platform software (& maybe digital hardware, too), with functional-correctness proofs?
  - [Conjecture: it's a small fraction of venture-capital investment in tech startups each year.]
Thank you!