RISC-V Summit

December 3 - 6, 2018
Santa Clara Convention Center
CA, USA

REVOLUTIONIZING THE COMPUTING LANDSCAPE AND BEYOND.

https://tmt.knect365.com/risc-v-summit
NEVER AGAIN: SPECTRE-PROOFING CHIP DESIGNS WITH END-TO-END FORMAL METHODS

Adam Chlipala
Associate Professor
MIT

https://tmt.knect365.com/risc-v-summit

@risc_v
Surprising processor-design mistakes, leading to information leaks through **timing**!
How Do We Protect Ourselves?

• Promise each other to think really hard about interactions of all IP blocks in SoCs?
• Stop using speculation and other optimizations that are apparently too hard for us to understand?
• Apply **formal verification** to guarantee that our designs avoid bad timing side channels?
Cleanup (or Lack Thereof) from Misspeculation

Cache

Processor

Branch Predictor

Faulty prediction!

Correct prediction

Doomed steps of execution
Mistake discovered

Mistake discovered
A Formal Condition to Avoid This Flaw?

1. Checkpoint state before proceeding based on branch predictor result.
2. Assert that state after rollback is really the same as before.
System Development Processes

Requirements → Design → Coding → Implementation → Release

Testing, debugging, code review, ...

Fuzzy feeling inside

Formal Methods

“Mechanized, end-to-end proofs of functional correctness”
Mechanized proofs

- **Real System** (source code)
- **Specification** (source code)
- **Proof Checker** (algorithm)
- **Proof** (source code)

(e.g., Coq proof assistant, which we use)
Hey, pal, wanna buy a chip? It's suuuuuuper secure.
Proofs of **Functional Correctness**

**Real System**
(source code)

**Specification**

- no segfaults

outputs right answer
Functional Correctness for a Processor

Equivalence notion needs to be smart about timing channels!

Out-of-order processor \approx \text{In-order reference design}
End-to-End Proofs

Layer 1

Layer 2

Layer 3

Whole-System Specification

Layers Proved Modularly
End-to-End Correctness for SoC Use?

Challenge: support **modular** verification of components
End-to-End Correctness for SoC Use?

Out-of-order processor ≈ Reference processor ≈ Cache system ≈ Naive memory

Challenge: support **modular** verification of components
Our Ongoing Work of that Kind for Timing Leaks

MIT project with Faye Duxovni, Luke Sciarappa, Murali Vijayaraghavan*, Joonwon Choi, and me

* moved to SiFive, starting related formal-methods work there

Uses Kami framework that I presented at the RISC-V Workshop a year ago
https://github.com/mit-plv/kami
Noninterference: the gold standard of information-leak avoidance

For any two runs of the same system...
...where public inputs match...
...public outputs match.
Censorship at Multiple Levels

Alternative input sequence 43, 24, 14 (same length)

must exist run...

same result of censorship!

Software

This is the “constant time” property well-known to crypto implementers.

Processor

Essentially the same property, but proven assuming it for software.

Memory

I.e., memory treats data values opaquely.
Proof Effort

**Software**

Use a verified static analysis (symbolic execution)!
Runs program with special “poison” values substituted for program inputs and flowed through system.
Analysis fails if program ever tries to inspect poison.

**Hardware**

Significant manual effort by developers.
Come up with invariants relating states of spec & implementation.
Example Software: Salsa20

// rotate x to left by n bits, the bits that go over
// the left edge reappear on the right
#define R(x,n) (((x) << (n)) | ((x) >> (32-(n))))

// addition wraps modulo 2^32
// the choice of 7,9,13,18 "doesn't seem very important" (spec)
static void quarter(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d) {
    *b ^= R(*d+*a, 7);
    *c ^= R(*a+*b, 9);
    *d ^= R(*b+*c, 13);
    *a ^= R(*c+*d, 18);
}

void salsa20_words(uint32_t *out, uint32_t in[16]) {
    int i;
    for (i=0; i<16; ++i) x[i/4][i%4] = in[i];
    for (i=0; i<10; ++i) { // 10 double rounds = 20 rounds
        // column round: quarter round on each column; start at ith element and wrap
        quarter(&x[0][0], &x[1][0],&x[2][0], &x[3][0]);
        quarter(&x[1][1], &x[2][1], &x[3][1], &x[0][1]);
        quarter(&x[2][2], &x[3][2], &x[0][2], &x[1][2]);
        quarter(&x[3][3], &x[0][3], &x[1][3], &x[2][3]);
        // row round: quarter round on each row; start at ith element and wrap around
        quarter(&x[0][0], &x[0][1], &x[0][2], &x[0][3]);
        quarter(&x[1][1], &x[1][2], &x[1][3], &x[1][0]);
        quarter(&x[2][2], &x[2][3], &x[2][0], &x[2][1]);
        quarter(&x[3][3], &x[3][0], &x[3][1], &x[3][2]);
    }
    for (i=0; i<16; ++i) out[i] = x[i/4][i%4] + in[i];
}

// inputting a key, message nonce, keystream index and constants to that transformation
void salsa20_block(uint32_t *out, uint32_t key[8], uint64_t nonce, uint64_t index) {
    static const char c[17] = "expand 32-byte k"; // arbitrary constant
    uint32_t in[16] = {LE(c), key[0], key[1], key[2],
        key[3], LE(c+4), nonce&0xffffffff, nonce>>32,
        index&0xffffffff, index>>32, LE(c+8), key[4],
        key[5], key[6], key[7], LE(c+12)};
    salsa20_words(out, in);
}

// enc/dec: xor a message with transformations of key, a per-message nonce and block index
void salsa20(uint64_t nonce) {
    int i, j;
    uint32_t msgword;
    uint32_t block[16];
    uint32_t key[8];
    for (i = 0; i < 8; i++) { key[i] = fromhost();
    }
    for (i=0; ; i++) {
        salsa20_block(block, key, nonce, i);
        for (i = 0; i < 16; i++) {
            msgword = fromhost();
            tohost(msgword ^ block[i]);
        }
    }
}
Important to Emphasize:

This approach is not a “fix” for Spectre/Meltdown-type bugs!

It's a way to validate purported fixes.

Who knows today which ones will “win in the market.”
Another Architectural Approach (work in progress)

Cache

Core

Secure Enclave Core

Sanctum hardware & software

≈

Cache

Core

Secure Enclave Core

Secure Enclave Core
Funding thanks to....
THANK YOU

https://tmt.knect365.com/risc-v-summit