RISC-V Zkt: Portable Timing Attack Resistance (via Dynamic Taint Analysis)

Think openly, build securely

RISC-V Summit 2022

December 14, 2022 - San Jose

Dr. Markku-Juhani O. Saarinen

Staff Cryptography Architect, PQShield Ltd.

> A Quarter of a Century of Timing Attacks

Some Greatest Hits (in asymmetric crypto TA) Along the Years:

- P.C. Kocher: *"Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems."* (CRYPTO 1996. Target: RSAREF 2.0 running on MS-DOS.)
- D. Brumley and D. Boneh: *"Remote timing attacks are practical."* (USENIX Security 2003. OpenSSL RSA remote key recovery, CVE-2003-0147.)
- B. Brumley and N. Toveri: *"Remote Timing Attacks Are Still Practical."* (ESORICS 2011. OpenSSL ECDSA remote key recovery, CVE-2011-1945.)
- Q. Guo, T. Johansson. A. Nilsson, *"A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM."* (Crypto 2020, PC Oracle, demoed against a claimed const-time impl.)

Every generation gets to learn the special implementation tricks!

Basic Sources of Timing Leaks

(That are avoidable with careful programming)

1. Secret-controlled branches and loops:

   ```
   if <secret> then { delay1(); } else { delay2(); }
   ```

2. Memory accesses (cache timing attacks). Can be a load or store.

   ```
   ct = SBox[pt ^ key];      //  observe latency with different inputs.
   ```

3. Arithmetic operations whose processing time just depends on inputs

   ```
   x = y % q;      //  division and remainder ops are rarely constant-time.
   ```

When Hiring a Crypto Dev..

Constant-time coding / algorithm knowledge is fundamental

- Transform simple conditionals into straight-line code using Boolean operations 🤔.

`x = s ? a : b;` *vs.* `x = b ^ ((-(s & 1)) & (a ^ b));`

- Symmetric ciphers such as AES also affected (usually via cache attack).

- Basic techniques: Bit-slicing (entire thing as a Boolean circuit), "full scan / collect." An implementation that avoids S-Box lookups is slow unless there is ISA support.

- Eliminate division instructions from modular reduction (e.g. Montgomery, Barrett).

- Blinding, Montgomery ladders, special addition and doubling rules (ECC), etc..

Practical Testing Methods

One trick: Repurpose "Use of Uninitialized Memory" detection.

- Mark sensitive data (such as secret keys) as uninitialized memory.
- The well-known tools Valgrind and LLVM Memory Sanitizer can be made to detect if branches and load/store addresses are tainted with uninitialized data.
- Covers only branches and loads, not non-constant time instruction sequences.

Examples of real-life usage:
Adam Langley (Google): *"Checking that functions are constant time with Valgrind."*
https://www.imperialviolet.org/2010/04/01/ctgrind.html
Kris Kwiatkowski (PQShield): *"Constant-time code verification with Memory Sanitizer."*
https://www.amongbytes.com/post/20210709-testing-constant-time/

This is “Dynamic Taint Analysis”

But has limitations for Constant-Time Checking

Classical DTA systems used special intermediate languages:

E. J. Schwartz, T. Avgerinos, and D. Brumley: *"All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask.)"* IEEE S & P 2010.

CWE-733: “Compiler Optimization Removal or Modification of Security-critical Code.”

Compilers are known to modify security-critical code and you can rarely be 100% sure which instructions are generated, or removed. Hence examination of compiled binary executable rather than an abstract representation of the algorithm is important.

RISC-V Crypto Extensions

- **Scalar Crypto** (and its many sub-extensions). Zkt (data-independent time) is one. *Ratified (Nov 2021). Supported in some commercial cores, compliance suites, GCC and LLVM compilers, prominent middleware (OpenSSL).*

- **Vector Crypto.** Roughly at "freeze." Hope is to proceed with ratification quickly. All vector crypto instructions are covered by Zkt (require data-independent time.)

- **Non-Standard Extensions.** Research hacks, proprietary Stuff has its place in RISC-V.

Data Independent Execution Latency: Zkt

Ratified in November 2021 as part of Scalar Crypto Spec

- The **Zkt** extension attests that the machine has data-independent execution time for a safe subset of instructions. This property is commonly called *"constant-time"* although should not be taken with that literal meaning.
- Basically just a list of instructions that are "safe to use" to hande crypto secrets.
- First official "Side-Channel ISA contract" (I know of). Does not affect functional behavior requirements. The programs still do the same things.
- Vendors do not have to implement all of the list's instructions to be Zkt compliant; however, if they claim to have Zkt and implement any of the listed instructions, it must have data-independent latency.

Taints in our Emulation

Traditional RED and BLACK

- Taint is *shadow state* attached to all variables. “Variables” refers both to processor registers and memory (including stack), and perhaps co-processor state.

- We’ll use just two taints, "**RED**" (secret) and "**BLACK**" (non-secret) here.

- In the implementation each register **x1**-**x31** has a taint (same for all bits). Zero register **x0** is always **BLACK.**

- Each 32-bit word in the memory has a taint state. This is arbitrary, could be for individual bytes. The microcontroller has a maximum of few megabytes of RAM.

Taint rules: Load Instructions

Not on the Zkt list: Avoid due to cache-timing attacks.

```
LB      rd, imm(rs1)    // RISC-V is a pure “load-and store”
LH      rd, imm(rs1)    // .. architecture: Only these
LW      rd, imm(rs1)    // .. instructions can be used to
LBU     rd, imm(rs1)    // .. load data from memory.
LHU     rd, imm(rs1)
```

Zkt: *Not on list. Latency may depend on* ***rs1****.*

Alarm: *Violation if* ***rs1*** *is* ***RED****.*

Rule: ***rd*** *inherits the taint of memory at* ***imm(rs1).***

Taint rules: Store Instructions

Not on the Zkt list: Avoid due to cache-timing attacks.

```
SB      rs2, imm(rs1)  // All stores using these three.
SH      rs2, imm(rs1)
SW      rs2, imm(rs1)
```

Zkt: *Not on list. Latency may depend on **rs1** (or even on **rs2** !)*

Alarm: *Violation if **rs1** is **RED**.*

Rule: *Memory location **imm(rs1)** inherits the taint of **rs2.***

Taint rules: Conditional Branches

Not on the Zkt list: Avoid branches due to timing leakage.

```
BEQ     rs1, rs2, <rel addr>
BNE     rs1, rs2, <rel addr>
BLT     rs1, rs2, <rel addr>
BGE     rs1, rs2, <rel addr>
BLTU    rs1, rs2, <rel addr>
BGEU    rs1, rs2, <rel addr>
```

Zkt:	*Not on list. Latency can be dependant on **rs1, rs2**.*
Alarm:	*Violation if either **rs1** or **rs2** are **RED**.*
Rule:	*- (No inheretence.)*

Taint rules: Indirect and Unconditional Jumps

Not on the Zkt list

```
JALR      rd, <rel to rs1>    // Indirect jump
```

Zkt: *Not on the list. Can be dependant on address,* ***rs1****.*

Alarm: *Violation if* ***rs1*** *is* ***RED****.*

Rule: ***rd*** *inherits the taint of* ***rs1****.*

```
JAL       rd, <rel addr>      // Unconditional jump
```

Zkt: *Not on the list. Latency can depend on the address.*

Rule: ***rd*** *is set to* ***BLACK*** *(this can be debated).*

Taint rules: Division

Not on the Zkt list: Crypto code avoids division instructions.

```
DIV     rd, rs1, rs2          // Division
DIVU    rd, rs1, rs2
REM     rd, rs1, rs2          // Remainder
REMU    rd, rs1, rs2
```

Zkt:	*Not on the list. Latency can be dependant on **rs1, rs2**.*
Alarm:	*Violation if either **rs1** or **rs2** is **RED**.*
Rule:	***rd** inherits both taints **rs1** ∨ **rs2** (red if either is red).*

Taint rules: Multiplication

On the Zkt list: Needs to be “constant time.”

```
MUL     rd, rs1, rs2
MULH    rd, rs1, rs2
MULHSU  rd, rs1, rs2
MULHU   rd, rs1, rs2
MULW    rd, rs1, rs2
```

Zkt:	*On the Zkt list. Latency must be **rs1, rs2** - independent.*
Alarm:	*None.*
Rule:	***rd** inherits both taints **rs1** $\vee$ **rs2** (red if either is red).*

Taint rules: Immediate arithmetic

On the Zkt list: Needs to be “constant time.”

```
ADDI[W]   rd, rs1, imm        // Format of instructions:

SLTI      SLTIU               // Immediate compare
XORI      ORI       ANDI      // Immediate Boolean logic
SLLI[W]   SRLI[W]   SRAI[W]   // Immediate Shifts
```

Zkt: *On the Zkt list. Latency must be **rs1** - independent.*

Alarm: *None.*

Rule: ***rd** inherits the taint of **rs1**.*

Taint rules: Basic “R-Type” Arithmetic

On the Zkt list: Needs to be “constant time.”

```
ADD[W]   rd, rs1, rs2        // Format of instructions:

SUB[W]   SLL[W]   SLT        // “R-Type” (register-register)
XOR      OR       AND
SLTU     SRL[W]   SRA[W]
```

Zkt: *On the Zkt list. Latency must be* ***rs1, rs2*** *- independent.*

Alarm: *None.*

Rule: ***rd*** *inherits both taints* ***rs1*** $\vee$ ***rs2*** *(red if either is red).*

Taint rules: Compressed Instructions

Same selection criteria: A subset is on the Zkt list.

Compressed Loads, stores, branches are <u>not</u> on the list. Arithmetic is:

`C.NOP`	`C.ADDI`	`C.ADDIW`	`C.LUI`
`C.SRLI`	`C.SRAI`	`C.ANDI`	`C.SUB`
`C.XOR`	`C.OR`	`C.AND`	`C.SUBW`
`C.ADDW`	`C.SLLI`	`C.MV`	`C.ADD`

<u>Zkt:</u>	*On the Zkt list. Latency must be **rs1, rs2** - independent.*
<u>Alarm:</u>	*None for those on the Zkt list. Alarms as in uncompressed.*
<u>Rule:</u>	***rd** inherits both taints **rs1** $\vee$ **rs2** (red if either is red).*

Taint rules: Symmetric Cryptography

On the Zkt list: Needs to be “constant time.”

All cryptography-specific instructions need to be constant time.

```
AES32*   AES64*          // Scalar AES Instructions
SHA256*  SHA512*         // Scalar SHA-2 Instructions
SM3*     SM4*            // China Standard Cryptography
```

Zkt:	*On the Zkt list. Latency must be **rs1, rs2** - independent.*
Alarm:	*None.*
Rule:	***rd** inherits both taints **rs1** ⋁ **rs2** (red if either is red).*

Taint rules: Cryptography Subset of Bitmanip

On the Zkt list: Needs to be "constant time."

All scalar cryptography instructions need to be constant time.

`CLMUL`	`CLMULH`	`XPERM4`	`XPERM8`	`ROR`	`ROL`
`RORI`	`RORIW`	`ANDN`	`ORN`	`XNOR`	`PACK`
`PACKH`	`PACKW`	`BREV8`	`REV8`	`ZIP`	`UNZIP`

Zkt:	*On the Zkt list. Latency must be **rs1, rs2** - independent.*
Alarm:	*None.*
Rule:	***rd** inherits both taints **rs1** $\vee$ **rs2** (red if either is red).*

Our DTA RISC-V Emulator Features

Originally a model used in PQC Coprocessor Co-Design Process

- The system being emulated is a “secure microcontroller” with cryptographic peripherals. The emulation behaviorally matches certain FPGA (and ASIC!) implementations; the same binaries can be ran on both.

- The emulator is also counts the times any PC is visited; produces profiling information and an annotated listings of an execution.

- Executes pretty fast, tens of millions of instructions second (roughly on par with an FPGA target running the same code).

- Instrumentation is eased with a couple of custom instructions that the emulator understands; these allow a testbench program to set and read taints.

Instrumentation Helpers

Used in testbench to mark the secret variables

Mapped into Custom-0 opcode space. Only used known to the emulator.

```
uint32_t     xrb_paint(uint32_t x, int sc);

XRB.PAINT    rd, rs1, rs2
```

Behavior:	*Sets **rd** = **rs1**.*
Alarm:	*None.*
Rule:	***rd** taint = **rs1** taint $\vee$ **rs2** literal value.*

Instrumentation Helpers

Used in testbench to mark the secret variables

Mapped into Custom-0 opcode space. Only used known to the emulator.

```
uint32_t    xrb_cover(uint32_t x, int sc);

XRB.COVER   rd, rs1, rs2
```

Behavior:	*Sets **rd = rs1.***
Alarm:	*None.*
Rule:	***rd** taints = **rs2** literal value. (Can be used to set to **BLACK.**)*

Instrumentation Helpers

Used in testbench to mark the secret variables

Mapped into Custom-0 opcode space. Only used known to the emulator.

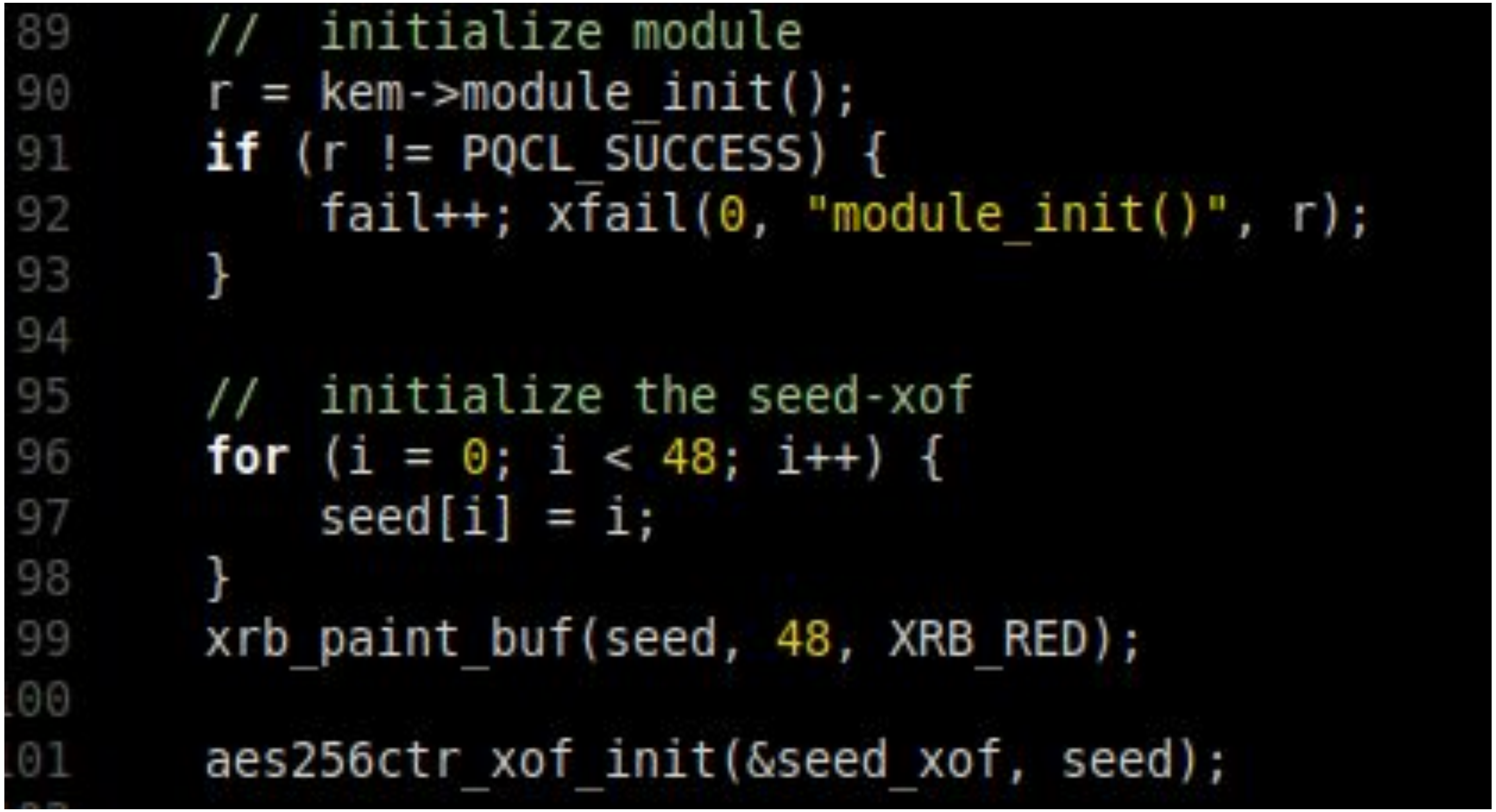
```
int xrb_test(uint32_t x);
```

```
XRB.TEST     rd, rs1
```

Behavior:	*Sets **rd** = **rs1** taint literal value.*
Alarm:	*Notify if **rs1** is **RED**.*
Rule:	***rd** taint = **BLACK.***

Example Use: Tainting Test Bench

In a test bench, taint some variables

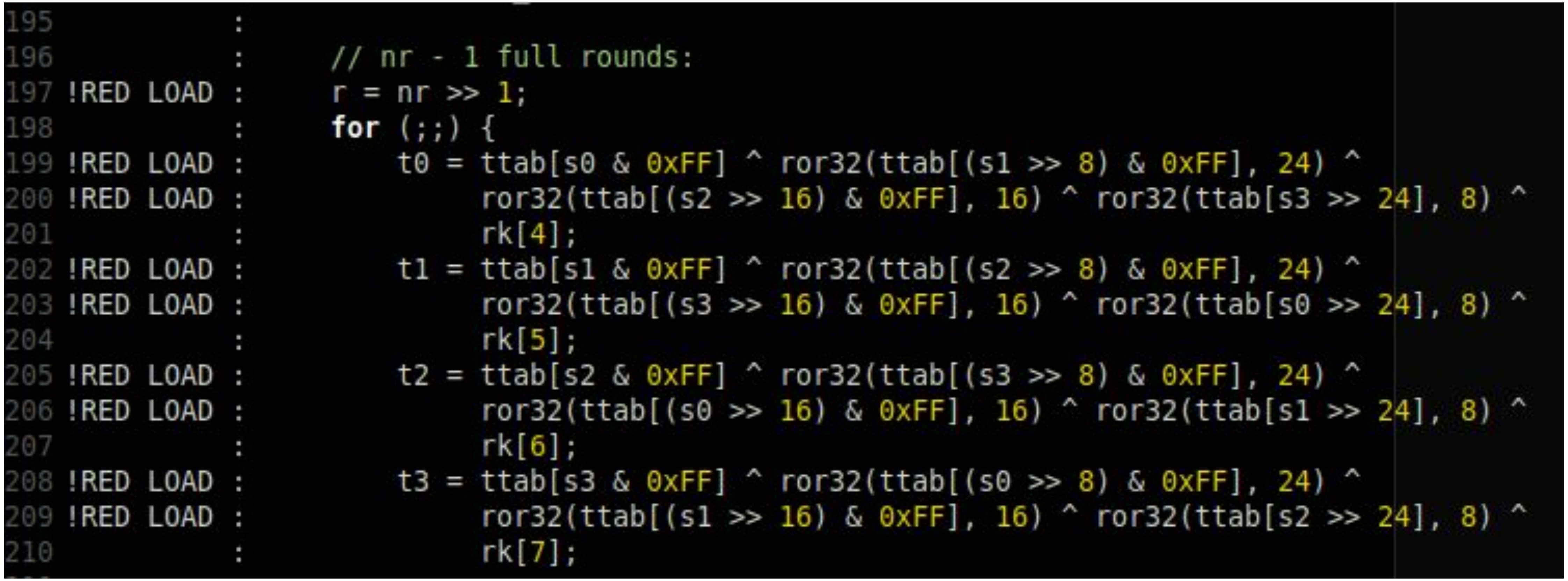


```
    //  initialize module
    r = kem->module_init();
    if (r != PQCL_SUCCESS) {
        fail++; xfail(0, "module_init()", r);
    }

    //  initialize the seed-xof
    for (i = 0; i < 48; i++) {
        seed[i] = i;
    }
    xrb_paint_buf(seed, 48, XRB_RED);

    aes256ctr_xof_init(&seed_xof, seed);
```

Example Use: Execute, Create Annotated Profile

Left margin has # times line was executed, CT Alarms

Here warnings are because the KAT bench as a non-constant time AES.



```
          :
          :     // nr - 1 full rounds:
!RED LOAD :     r = nr >> 1;
          :     for (;;) {
!RED LOAD :         t0 = ttab[s0 & 0xFF] ^ ror32(ttab[(s1 >> 8) & 0xFF], 24) ^
!RED LOAD :              ror32(ttab[(s2 >> 16) & 0xFF], 16) ^ ror32(ttab[s3 >> 24], 8) ^
          :              rk[4];
!RED LOAD :         t1 = ttab[s1 & 0xFF] ^ ror32(ttab[(s2 >> 8) & 0xFF], 24) ^
!RED LOAD :              ror32(ttab[(s3 >> 16) & 0xFF], 16) ^ ror32(ttab[s0 >> 24], 8) ^
          :              rk[5];
!RED LOAD :         t2 = ttab[s2 & 0xFF] ^ ror32(ttab[(s3 >> 8) & 0xFF], 24) ^
!RED LOAD :              ror32(ttab[(s0 >> 16) & 0xFF], 16) ^ ror32(ttab[s1 >> 24], 8) ^
          :              rk[6];
!RED LOAD :         t3 = ttab[s3 & 0xFF] ^ ror32(ttab[(s0 >> 8) & 0xFF], 24) ^
!RED LOAD :              ror32(ttab[(s1 >> 16) & 0xFF], 16) ^ ror32(ttab[s2 >> 24], 8) ^
          :              rk[7];
```

Instrumenting around False Positives

Example: Rejection Samplers Need Special Instrumentation

Rejection samplers have roughly the pattern:

```
do { x = random_try(); } while (!accept(x));
```

Check that “**x**” is independently random for each try, so that the number of iterations does not reveal information about final “**x**”.

- RSA Key Generation (finding primes among random candidates).
- Uniform random modulo q from random bits, non-uniform samplers.
- Dilithium Signing (has ~20% success per signature candidate).

Conclusions

- **The Timing Contract**: The **Zkt** data-independent latency extension allows portable crypto code in the RISC-V ecosystem.

- **Instruction-Level Variable Tainting:** We can fully trace data flows in actual executions of high-level algorithms. Produces annotated listings with profiling and constant-time violation information.

- Instrumentation for symmetric cryptography is fairly easy. Algorithms that are not “literally constant time” (but still secure) require some manual analysis for instrumentation. But this is needed only once.