>> Zero to RandomX.js:
>> Bringing Webmining Back From The Grave

Thursday, November 6th | K15 Old main G32 | 2 PM - 4 PM

2. About me

- Liam, l-m, github.com/l1mey112 (GH)
- https://l-m.dev/, l-m at l-m.dev
- These slides: https://l-m.dev/talk

- ~~Highschool student~~ University Student
- Adv Mathematics/Computer Science @ UNSW

- Compilers, **Systems Programming**, WebAssembly, Linux
- (new!) Interactive Theorem Proving w/ Lean

3. This Talk?

```
// TODO: (1.RX) Introduce RandomX.
// TODO: (2.JS) Introduce RandomX.js
// TODO: (3.FP) Cover the semifloat impl. in RandomX.js
```

>> (1.RX) RandomX

5. (1.RX) ASICs

6. (1.RX) RandomX

RandomX is a proof-of-work algorithm that is optimized for general-purpose CPUs. It has stood the test of time as an **ASIC resistant** hash function.

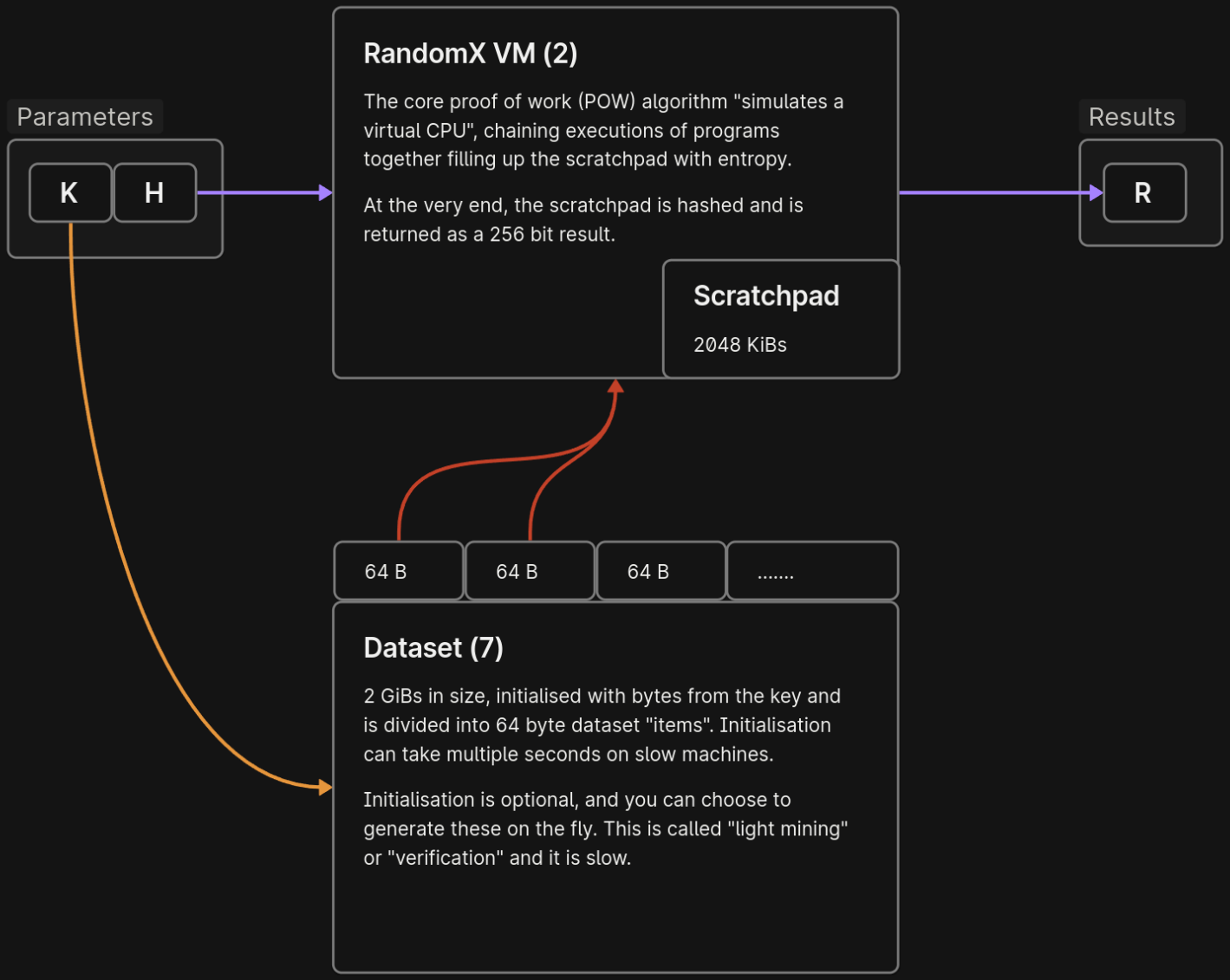
On November 30th, 2019, Monero switched from its old POW algorithm, *cryptonight*, to RandomX. Which had already been broken.

> RandomX uses random code execution (hence the name) together with several memory-hard techniques to minimize the efficiency advantage of specialized hardware.
>
> — tevador/RandomX

(Goal: Promote egalitarian mining!)

https://github.com/tevador/RandomX

Defn. ***A***pplication-***S***pecific-***I***ntegrated-***C***ircuit — ASIC are specialised circuits.

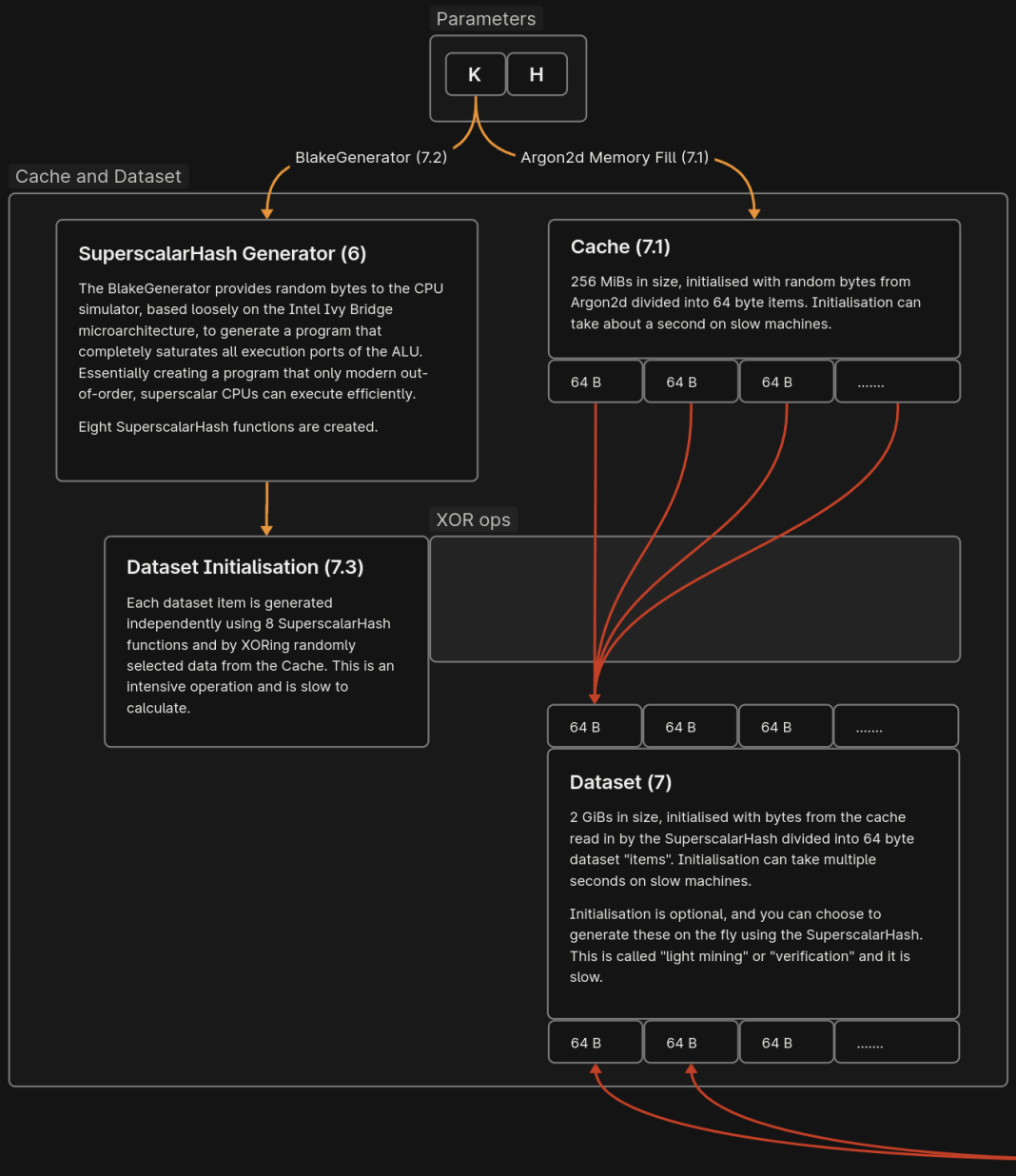

Parameters
K
H
RandomX VM (2)
The core proof of work (POW) algorithm "simulates a virtual CPU", chaining executions of programs together filling up the scratchpad with entropy.
At the very end, the scratchpad is hashed and is returned as a 256 bit result.
Scratchpad
2048 KiBs
Results
R
64 B
64 B
64 B
.....
Dataset (7)
2 GiBs in size, initialised with bytes from the key and is divided into 64 byte dataset "items". Initialisation can take multiple seconds on slow machines.
Initialisation is optional, and you can choose to generate these on the fly. This is called "light mining" or "verification" and it is slow.

8. (1.RX) Defense in depth (of the ASIC)

Goal. Economic infeasability of ASICs.

- 2 GiBs of (external) DRAM + prefetch - **Memory-Hardness**.
- **Random** code e**X**ecution.
- **Superscalar design**, Speculative execution, branch prediction.
- 2 MiBs of scratchpad (L1 ⊆ L2 ⊆ L3).
- Full compliant IEEE754. (floating point)
- Hardware SIMD, integer dividers, AES instructions.

Each increases die area for an ASIC!



Pre-initialisation.

- The key K is passed to the SSH generator, which generates 8 “superscalar” hash functions.
- K is also used to fill 256 MiBs of static memory.

Dataset-initialisation.

- Each 64-byte Dataset item is generated by executing each of those 8 functions, and XORing data from the cache.
- Cache $\Longrightarrow$ Dataset.

Algorithm.

1. Initial seed S is calculated. (Blake2b)
2. Scratchpad is filled. (AesGenerator1R=S)

---

3. Program Buffer filled. (AesGenerator4R*)

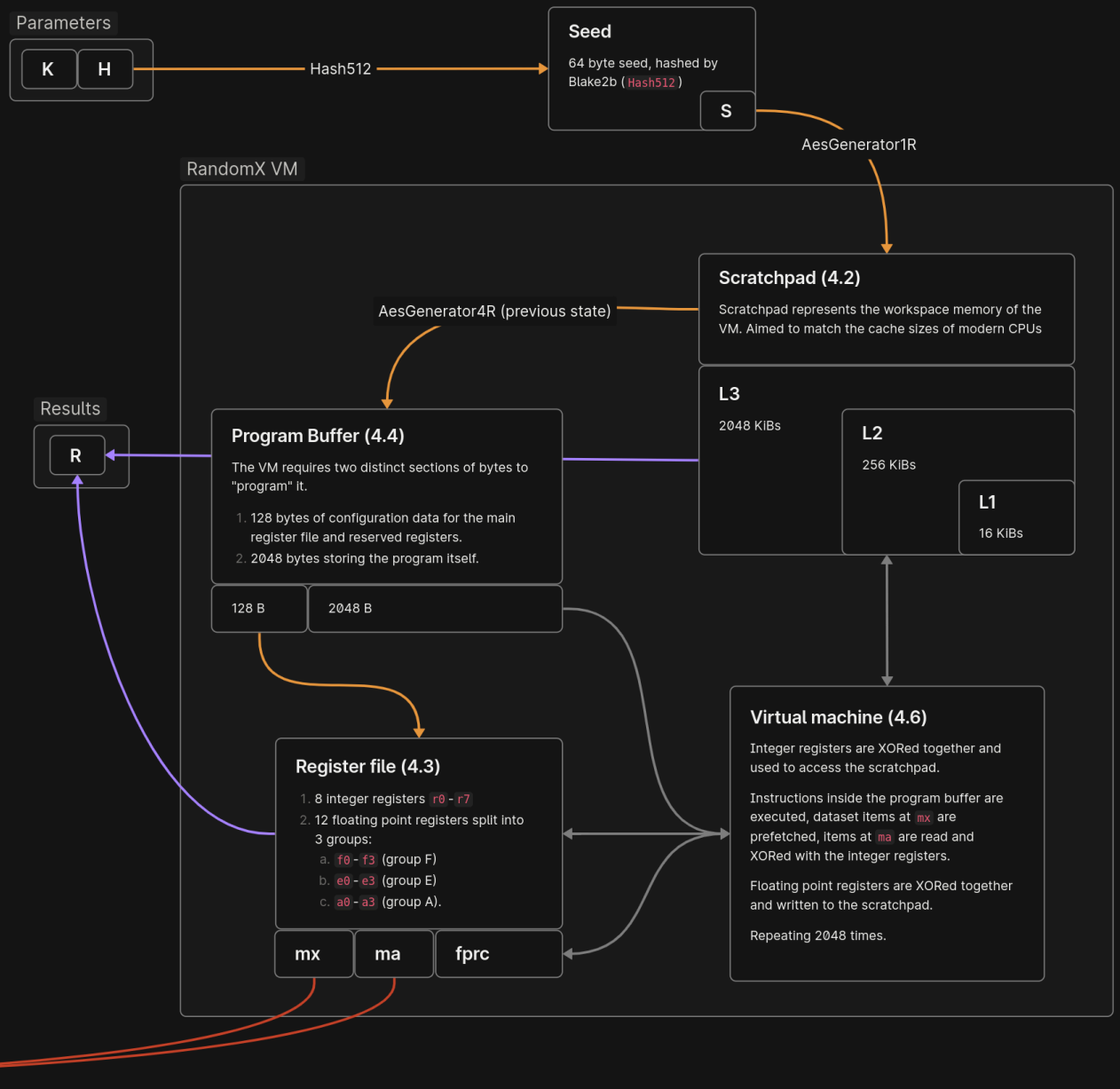
   (VM is programmed from contents)

4. **The VM is executed.**
5. Register file is hashed and copied to AesGenerator4R* state.
6. Steps 3-5 are performed **8** times.

   (chained VM execution)

---

7. Scratchpad fingerprint is calculated. (AesHash1R)
8. Final result R is the hash of scratchpad fingerprint + register file. (Blake2b)



11. (1.RX) Instructions + Registers

Instructions (integer).

- IADD, ISUB, IMUL, IMULH, ISMULH, INEG
- IXOR, IROL, ISWAP_R
- IMUL_RCP (division by imm32)

Instructions (FP).

(IEEE754) FADD, FSUB, FMUL, FDIV, FSQRT

- FSWAP_R: (dst0, dst1) = (dst1, dst0)
- FSCAL_R: (XOR by 0x80F0000000000000)

Instructions (misc).

- ISTORE: *arbitrary* scratchpad addr.
- CBRANCH: backwards branch, flat loops
- CFROUND: change FP rounding (IEEE754)

  (adjusts **fprc** = 0, 1, 2, 3)

0	roundTiesToEven
1	roundTowardNegative
2	roundTowardPositive
3	roundTowardZero

(most instructions have *_R, *_M variants)

12. (1.RX) IEEE754

- `0.1 + 0.2 = 0.30000000000000004 != 0.3` (?? wtf)
- **IEEE754** makes a *tradeoff*. Define R as set of *doubles* without NaN. Then $|\mathrm{R}| < 2^{64}$.

Result. R is a finite amount of **useful** numbers, with $\pm\infty$. (IEEE754)

```
bool chk(double x, double y, double z) {
    // 99% this is false
    return (x + y) + z == x + (y + z);
}
```

$$\circ(\circ(x+y)+z) \neq \circ(x+\circ(y+z))$$

FP *feels random*, is portable/well defined?

13. (1.RX) IEEE754

FP is perfectly specified, with $+$, $-$, $\times$, $\div$, $\sqrt{x}$ **correctly rounded**. (IEEE754)

- P/L defn. as $\mathtt{x} + \mathtt{y} \coloneqq \mathbf{RN}(\mathtt{x} + \mathtt{y})$, $\mathtt{sqrt}(\mathtt{x}) \coloneqq \mathbf{RN}(\sqrt{\mathtt{x}})$, etc.
- Anything else? Sorry. $\sin(x)$, $\exp(x)$ behave differently on other platforms.

Define $\circ : \mathbb{R}^\infty \to \mathrm{R}$ to be a function that rounds according to a (global) fprc.

- $\mathrm{fadd}(x, y) = \circ(x + y)$, $\mathrm{fsqrt}(x) = \circ(\sqrt{x})$, etc.

Caveat. All P/L give you $\mathbf{RN}$ only.
(foreshadowing)

```
○(x) {
  case fprc = 0: RN(x)
  case fprc = 1: RD(x)
  case fprc = 2: RU(x)
  case fprc = 3: RZ(x)
}
```

fprc		
0	roundTiesToEven	$\mathbf{RN} : \mathbb{R}^\infty \to \mathrm{R}$
1	roundTowardNegative	$\mathbf{RD} : \mathbb{R}^\infty \to \mathrm{R}$
2	roundTowardPositive	$\mathbf{RU} : \mathbb{R}^\infty \to \mathrm{R}$
3	roundTowardZero	$\mathbf{RZ} : \mathbb{R}^\infty \to \mathrm{R}$

14. (1.RX) IEEE754 in P/L

Assumed $\mathrm{add}(x, y) = \mathbf{RN}(x + y)$ in all P/L.

```c
// in C
double add(double x, double y) {
    return x + y                        // RN(x + y)
}
```

```javascript
// in JavaScript
function add(x, y) {
    return x + y                        // RN(x + y)
}
```

```rust
// in Rust
pub fn add(x: f64, y: f64) -> f64 {
    x + y                               // RN(x + y)
}
```

Opt-in to avoid UB.

```c
#include <fenv.h>

double add(double x, double y) {
    #pragma STDC FENV_ACCESS ON

    return x + y; // ○(x + y)
}
```

```
add(double, double):
    ; xmm0 = ○(xmm0 + xmm1)
    addsd   xmm0, xmm1
    ret
```

15. (1.RX) Conclusion

> RandomX uses random code execution (hence the name) together with several memory-hard techniques to minimize the efficiency advantage of specialized hardware.
>
> — tevador/RandomX

(Reminder: the goal is to make ASICs resemble CPUs.)

An ASIC for RandomX needs:

- 2 GiBs of (external) DRAM - **Memory-hard**
- **Random** code e**X**ecution.
- Superscalar design, instruction/µop cache.
- Speculative execution, branches.
- Full compliant IEEE754.
- Hardware AES, dividers, SIMD floating point.
- 2 MiBs of scratchpad (L1, L2, L3).

Verdict. No advantage over CPU.

(economically infeasible)

Success!

>> (2.JS) RandomX.js

17. (2.JS) RandomX.js?

Monero has a certain relationship with Web Mining.

- *Could* replace ads, just used for cryptojacking websites.

  The Pirate Bay used **Coinhive**. CH discontinued in 2019.

No **serious/working** implementations of RandomX since its introduction.

(suggesting that one could be implemented was laughable)

Why?

18. (2.JS) RandomX.js?

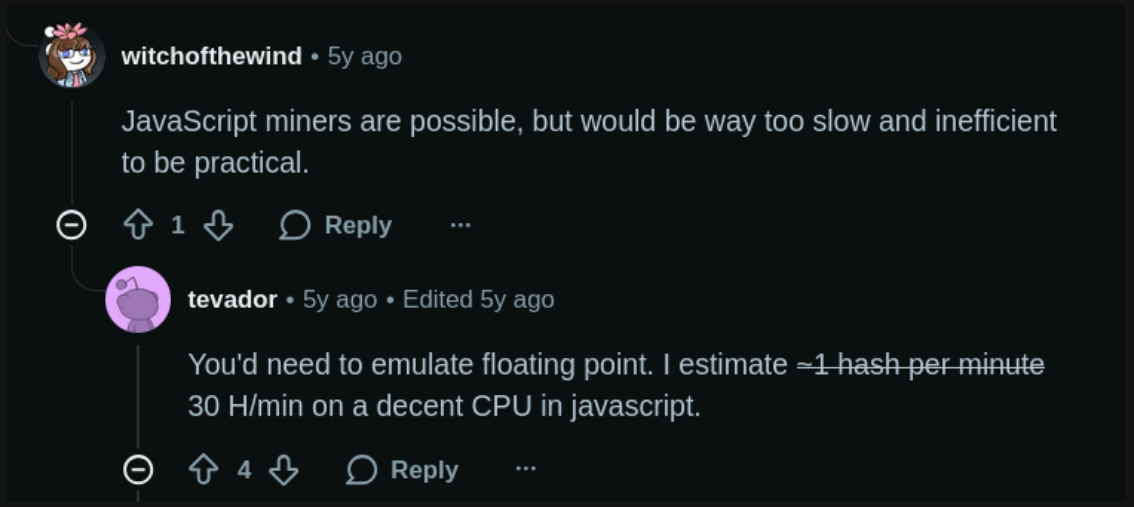
Does RandomX facilitate botnets/malware mining or web mining?

[…]

Web mining is infeasible due to the large memory requirement and the lack of directed rounding support for floating point operations in both Javascript and WebAssembly.

— tevador/RandomX README.md

(Stuck with RN !!)



19. (2.JS) RandomX.js

> RandomX.js is an implementation of the ubiquitous Monero POW algorithm RandomX in JavaScript.
>
> — l1mey112/randomx.js

Why? It’s a fun engineering problem!

~5 years later (2024):

- JavaScript has WebAssembly now, with SIMD and shared memory.
- JavaScript has threads (web workers).
- WASM is pretty expressive, with branch tables, memcpy(), etc.

No native AES instruction, no RD, RU, RZ, and

- **shouldn’t** have 2 GiB dataset, so only light mining.

  (compute dataset items on fly, expensive…)

---

https://github.com/l1mey112/randomx.js

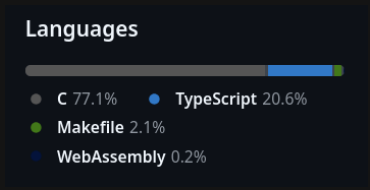
20. (2.JS) Overview

- Complete rewrite of core RandomX (C++) in C, 5000 lines.
- 500 lines of host *glue* TypeScript. (to execute JIT VM)
- Same tests as RX, and 200k LOC test suite for FP impl.

  (Most readable RandomX impl. in existence)

- No interpreter, RX code generated in WASM and executed by JS.
- Rewritten cryptography to use WASM SIMD.
- Performant directed rounding, **semifloat**. (!!)
- Support for shared memory/threads.

  (i.e. share the 256 MiB cache)



RandomX WASM **JIT** - VM implementation	Dataset/Crypto	FP+mul	VM
2600 lines	1200 lines	400	300

21. (2.JS) Performance?

```
node examples/randomx.js
# machine id: AMD Ryzen 7 3800X 8-Core Processor [rx/0+relaxed-simd+!fma] Node.js/v22.9.0 (linux x64)
# cache construction time 535.8 ms
# average hashrate: 22.0 H/s

node examples/randomx_threaded.js
# machine id: AMD Ryzen 7 3800X 8-Core Processor [rx/0+relaxed-simd+!fma] Node.js/v22.9.0 (linux x64)
# initialising thread 0..15
# average hashrate: 208.0 H/s
```

On the same machine (with light mining), I got 100 H/s per thread.

5x slower is pretty good!

22. (2.JS) Motivation

- During VM, a branch table (call_indirect) selects correct FP op w/ rounding, so FADD_R has *fadd_0* **RN**, *fadd_1* **RD**, *fadd_2* **RU**, *fadd_3* **RZ**.

```
case FADD_R:
    // dst = dst + src
    WASM_U8_THUNK({
        0x20, F(inst->dst), // local.get $dest
        0x20, A(inst->src), // local.get $src
        0x23, $fprc,        // global.get $fprc
        0x11, 3, 0,         // call_indirect table 0
        0x21, F(inst->dst), // local.set $dest
    });
    break;

v128_t fadd_0(v128_t a, v128_t b) {
    return wasm_f64x2_add(a, b); // RN(a + b)
}
```

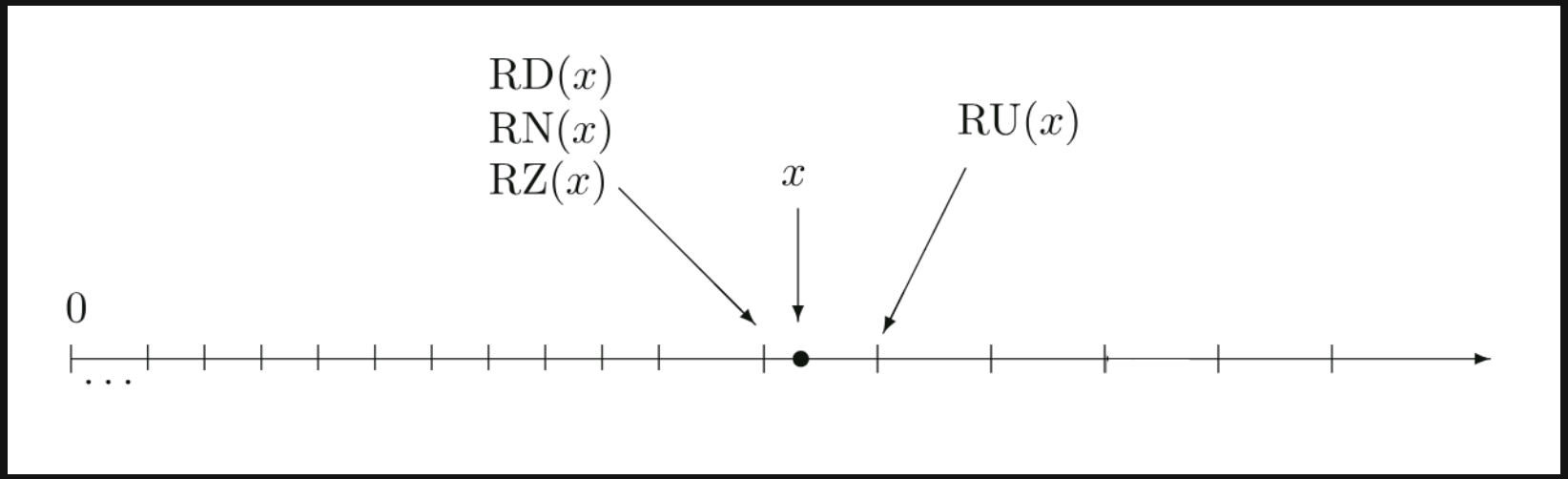
```
WASM_SECTION(WASM_SECTION_ELEMENT, {
    WASM_U8_THUNK({
        5, // elems = vec(5)

        // elem 0: fadd
        0x02,
        0,                                  // table index = 0
        0x41, 0, 0x0b,                      // offset = i32.const 0
        0x00,                               // funcref
        4,                                  // elements = vec(4)
        FIDX(3), FIDX(4), FIDX(5), FIDX(6), // fprc_0..fprc_3
        // fadd_0,  fadd_1,  fadd_2,  fadd_3
        //
        // setup fsub_0-3, fmul_0-3, fdiv_0-3, fsqrt_0-3, ...
```

Goal. How do we actually implement fadd_1, fadd_2, etc?

>> (3.FP) Semifloat

24. (3.FP) How does rounding actually work?



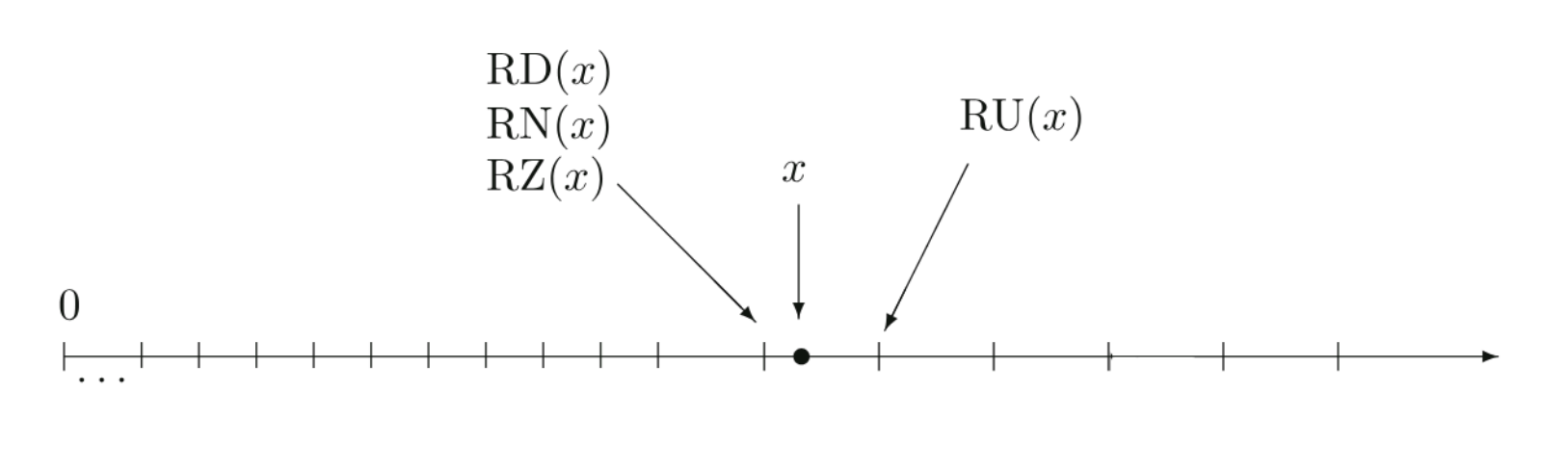
- **RN** returns the closest FP to x (with conditions to break ties)
- **RD** returns largest FP $\leq x$ (round toward $-\infty$)
- **RU** returns smallest FP $\geq x$ (round toward $+\infty$)

Then, we define

$$\mathbf{RZ}(x) = \begin{cases} \mathbf{RD}(x) & x > 0, \\ \mathbf{RU}(x) & x < 0, \\ 0 & \text{otherwise.} \end{cases}$$

25. (3.FP) How does rounding actually work?

Defn. *Unit in the last-place* (**ULP**) is the gap between two FP numbers nearest to x. (Kahan)



Each FP number (tick) is at most one ULP away! (can we use this?)

Note. All ways to round either go up a tick, or go down a tick, nothing else.

26. (3.FP) Measuring error

Take some $x \in \mathbb{R}$, then $x = \circ(x) + \varepsilon$.

Solving for $\varepsilon = x - \circ(x)$, then if

- $\varepsilon = 0$, the operation was **exact**,
- $\varepsilon < 0$, we rounded up, as $\circ(x) > x$,
- $\varepsilon > 0$, we rounded down, as $\circ(x) < x$.

Upshot. If we find ε, even just the $\pm$, we can detect when rounding happens and correct it!

Imagine $\mathbf{RN}^* : \mathbb{R} \to (\mathrm{R} \times \mathrm{R})$,

$$(X, \varepsilon) = \mathbf{RN}^*(x) \implies X + \varepsilon = x$$

($\mathbf{RN}^*$ doesn't exist in general)

```
// impl. fprc = 1, so RD (round down)
fadd_1(x, y) {
  sum, ε = RN*(x + y)

  if ε < 0 {
    // sum rounded up
    return sum - ulp
  } else {
    // sum rounded down or exact
    return sum
  }
}
```

27. (3.FP) TwoSum (error free transform)

```
// RN*(x + y)
TwoSum(x, y) {
  s  = RN(a + b)
  a' = RN(s - b)
  b' = RN(s - a')
  δa = RN(a - a')
  δb = RN(b - b')
  ε  = RN(δa + δb)
  return (s, ε)
}

fadd_1(x, y) {
  sum, ε = TwoSum(x, y)

  if ε < 0 {
    return sum - ulp
  } else {
    return sum
  }
}
```

```
#include <math.h> // for nextafter()

double sum_residue(double a, double b, double c) {
    double delta_a = a - (c - b);
    double delta_b = b - (c - a);
    double res = delta_a + delta_b;
    return res;
}

// return RD(a + b)
double fadd_1(double a, double b) {
    double c = a + b;
    double res = sum_residue(a, b, c);
    if (res < 0) {
        // c - ulp
        return nextafter(c, -INFINITY);
    } else {
        return c;
    }
}
```

28. (3.FP) FP implementations

- FP impl. by hardware is often called **hardfloat**.
- FP impl. by software (integer operations) is called **softfloat**.

  (Motv. CPU only supports integer operations, emulate FP in software.)

What is this thing called?

- In vein of these similar defn. FP emulations in terms of $\mathbb{RN}$ will be called **semifloat**. (Is semifloat any faster than softfloat?)

```c
#include <fenv.h> // fesetround()

double hard_fadd_1(double a, double b) {
    #pragma STDC FENV_ACCESS ON

    fesetround(FE_DOWNWARD);
    double res = a + b;
    fesetround(FE_TONEAREST);
    return res;
}
```

```c
v128_t sum_residue(v128_t a, v128_t b, v128_t c) {
    v128_t delta_a = wasm_f64x2_sub(a, wasm_f64x2_sub(c, b));
    v128_t delta_b = wasm_f64x2_sub(b, wasm_f64x2_sub(c, a));
    v128_t res = wasm_f64x2_add(delta_a, delta_b);
    return res;
}

v128_t semi_fadd_1(v128_t dest, v128_t src) {
    v128_t c = wasm_f64x2_add(dest, src);
    v128_t res = sum_residue(dest, src, c);
    // performs the if (res < 0) nextafter(c, -INFINITY); as SIMD
    return nextafter_1_finite_nozero(res, c);
}
```

29. (3.FP) Softfloat (additive)

```c
// github.com/ucb-bar/berkeley-softfloat-3
static float64_t f64_add(float64_t a, float64_t b) {
    // ...

    uA.f = a;
    uiA = uA.ui;
    signA = signF64UI( uiA );
    uB.f = b;
    uiB = uB.ui;
    signB = signF64UI( uiB );
    if ( signA == signB ) {
        return softfloat_addMagsF64( uiA, uiB, signA );
    } else {
        return softfloat_subMagsF64( uiA, uiB, signA );
    }
}
```

---

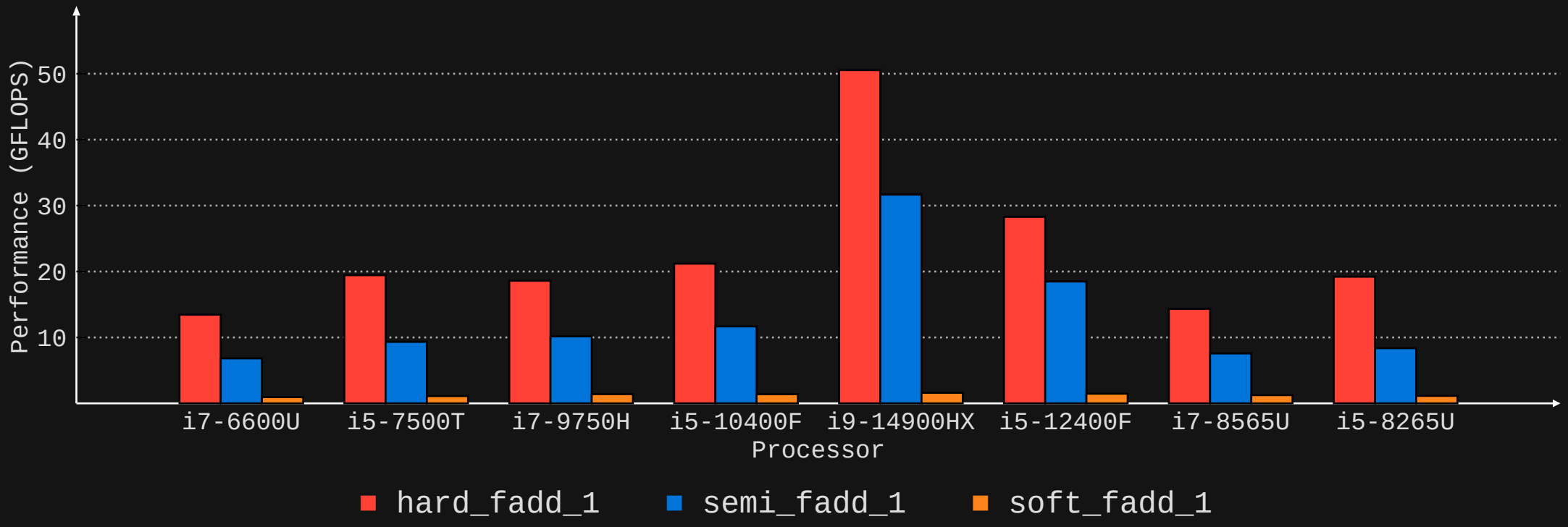
github.com/l1mey112/fp-rounding-emulation

(check bench_RD folder)

```c
        sigA <<= 9;
        sigB <<= 9;
// performs |A| + |B|
static float64_t
softfloat_addMagsF64( uint_fast64_t uiA, uint_fast64_t uiB,
bool signZ )
{
    // .... }
    expA = expF64UI( uiA );
    sigA = fracF64UI( uiA );
    expB = expF64UI( uiB );
    sigB = fracF64UI( uiB );
    expDiff = expA - expB;
    if ( ! expDiff ) {
        if ( ! expA ) {
        }
        sigZ = UINT64_C( 0x2000000000000000 ) + sigA + sigB;
        if ( sigZ < UINT64_C( 0x4000000000000000 ) ) {
            expZ = expA;
            sigZ <<= 1;
        }
    } else {
        sigA <<= 9;
        sigB <<= 9;
        if ( expDiff < 0 ) {
            expZ = expB;
            if ( expA ) {
                sigA += UINT64_C( 0x2000000000000000 );
            }
            sigA = softfloat_shiftRightJam64( sigA, -expDiff );
        } else {
            expZ = expA;
            if ( expB ) {
                sigB += UINT64_C( 0x2000000000000000 );
            }
            sigB = softfloat_shiftRightJam64( sigB, expDiff );
        }
        uiZ = uiA + sigB;
        goto uiZ;
    }
    // perform rounding here
    return softfloat_roundPackToF64( signZ, expZ, sigZ );
uiZ:
    uZ.ui = uiZ;
    return uZ.f;
}
```

30. (3.FP) Benchmark (additive)

```
$ git clone https://github.com/l1mey112/fp-rounding-emulation
$ cd fp-rounding-emulation/bench_RD
$ clang ... && ./bench
# fprc(1):    XX.XX GFLOPS "hard_fadd_1"
# fprc(1):    XX.XX GFLOPS "semi_fadd_1" (x1.88 overhead)
# fprc(1):     X.XX GFLOPS "soft_fadd_1" (x17.56 overhead)
#                  (average overhead)    ^^^^^^^^^^^^^^^^^
```



31. (3.FP) Semifloat (multiplicative)

```
v128_t semi_fmul_1(v128_t dest, v128_t src) {
    v128_t c = wasm_f64x2_mul(dest, src);

    // scale a_scaled, b_scaled, c_scaled to be within [0.5,1) - frexp(3), ldexp(3)
    v128_t expa, expb;
    v128_t a_scaled = frexp_reg_e_nozero_noinf(dest, &expa);
    v128_t b_scaled = frexp_reg_e_nozero_noinf(src, &expb);
    v128_t c_scaled = ldexp_reg_e_nozero_noinf(c, wasm_i64x2_sub(wasm_i64x2_neg(expa), expb)); // -expa - expb
    v128_t res = mul_residue(a_scaled, b_scaled, c_scaled);

    v128_t isinf = wasm_f64x2_eq(c, wasm_f64x2_const(INFINITY, INFINITY));

    // branch hit 0.024882% to 0.003414% of the time, so never
    if (unlikely(wasm_v128_any_true(isinf))) {
        v128_t isfinite_a = wasm_f64x2_ne(dest, wasm_f64x2_const(INFINITY, INFINITY));
        // fin * fin = _inf_; round down to the nearest representable number
        // inf * fin = inf
        //
        // res = isinf ? (isfinite_a ? -1.0 : 1.0) : res
        //                             ^^^^   ^^^
        //                    rounding down   no rounding
        v128_t res_isinf = wasm_v128_bitselect(wasm_f64x2_const(-1.0, -1.0), wasm_f64x2_const(1.0, 1.0), isfinite_a);
        res = wasm_v128_bitselect(res_isinf, res, isinf);
    }

    return nextafter_1_finite_nozero(res, c);
}
```

32. (3.FP) Semifloat (multiplicative)

```c
static inline v128_t upper_half(v128_t x) {
    v128_t secator = wasm_f64x2_const(134217729.0, 134217729.0);
    v128_t p = wasm_f64x2_mul(x, secator);
    return wasm_f64x2_add(p, wasm_f64x2_sub(x, p));
}

static inline v128_t mul_residue(v128_t a, v128_t b, v128_t c) {
    v128_t aup = upper_half(a);
    v128_t alo = wasm_f64x2_sub(a, aup);
    v128_t bup = upper_half(b);
    v128_t blo = wasm_f64x2_sub(b, bup);

    v128_t high = wasm_f64x2_mul(aup, bup);
    v128_t mid = wasm_f64x2_add(wasm_f64x2_mul(aup, blo), wasm_f64x2_mul(alo, bup));
    v128_t low = wasm_f64x2_mul(alo, blo);
    v128_t ab = wasm_f64x2_add(high, mid);
    v128_t resab = wasm_f64x2_add(wasm_f64x2_sub(high, ab), mid);
    resab = wasm_f64x2_add(resab, low);

    v128_t fma = wasm_f64x2_sub(ab, c); // a*b - c ----- we want a * b - RN(a * b) = res
    return wasm_f64x2_add(resab, fma);
}
```

33. (3.FP) Semifloat FMA? (multiplicative)

Let $\mathtt{c} = \mathbf{RN}(a \times b)$, assume $\varepsilon \in \mathrm{R}$. Going back to what we know,

$$a \times b = \mathbf{RN}(a \times b) + \varepsilon$$
$$a \times b = \mathtt{c} + \varepsilon.$$

Solving for ε,

$$a \times b - \mathtt{c} = \varepsilon,$$

apply $\mathbf{RN}$ on both sides,

$$\begin{aligned}\mathbf{RN}(a \times b - \mathtt{c}) &= \mathbf{RN}(\varepsilon) \\ &= \varepsilon,\end{aligned}$$

as ε is already a floating point number.

Goal. Is there a way to do $a \times b - \mathtt{c}$ in one rounding?

34. (3.FP) Semifloat FMA (multiplicative)

IEEE754-2008 specifies $+$, $-$, $\times$, $\div$, $\sqrt{x}$, $\mathrm{fma}(a,b,c)$ correctly rounded. (?? fma)

Defn. Fused multiply-add,

$$\mathrm{fma}(a,b,c) = \circ(a \times b + c).$$

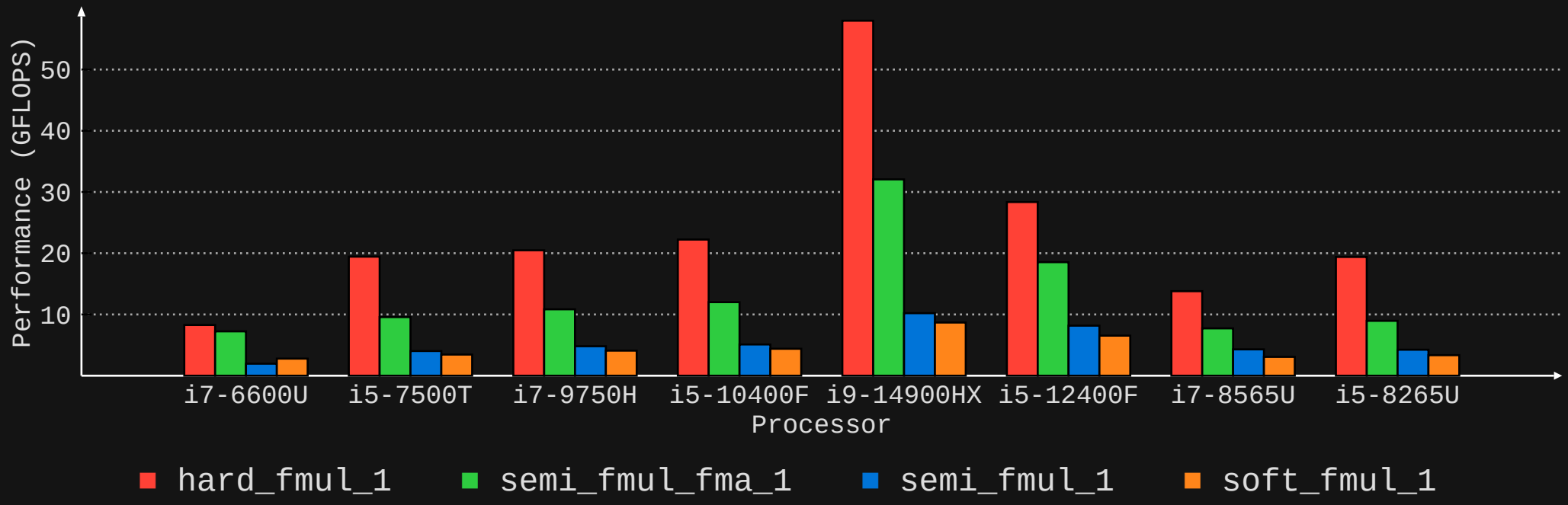
(Not a double round! So $\mathrm{fma}(a,b,c) \neq \circ(\circ(a \times b) + c)$.)

```
v128_t mul_residue_fma(v128_t a, v128_t b, v128_t c) {
    // return fma(a, b, -c)
    return wasm_f64x2_relaxed_madd(wasm_f64x2_neg(c), b, a); // RN(a*b - c)
}

v128_t semi_fmul_fma_1(v128_t dest, v128_t src) {
    v128_t c = wasm_f64x2_mul(dest, src);
    v128_t res = mul_residue_fma(dest, src, c);
    return nextafter_1_finite_nozero(res, c);
}
```

35. (3.FP) Benchmark (multiplicative)

```
$ git clone https://github.com/l1mey112/fp-rounding-emulation
$ cd fp-rounding-emulation/bench_RD
$ clang ... && ./bench
# fprc(1):    XX.XX GFLOPS "hard_fmul_1"
# fprc(1):    XX.XX GFLOPS "semi_fmul_fma_1" (x1.78 overhead)
# fprc(1):     X.XX GFLOPS "semi_fmul_1"     (x4.33 overhead)
# fprc(1):     X.XX GFLOPS "soft_fmul_1"     (x5.00 overhead)
#                  (average overhead)        ^^^^^^^^^^^^^^^^
```



36. (3.FP) Softfloat (multiplicative)

```
float64_t f64_mul( float64_t a, float64_t b )
{
    union ui64_f64 uA, uB, uZ;
    uA.f = a; uB.f = b;
    uint64_t uiA = uA.ui;            uint64_t uiB = uB.ui;
    int16_t expA  = expF64UI( uiA ); uint64_t sigA  = fracF64UI( uiA );
    int16_t expB  = expF64UI( uiB ); uint64_t sigB  = fracF64UI( uiB );

    if ( expA == 0x7FF ) {
        goto infArg;
    }
    int16_t expZ = expA + expB - 0x3FF;
    sigA = (sigA | UINT64_C( 0x0010000000000000 ))<<10;
    sigB = (sigB | UINT64_C( 0x0010000000000000 ))<<11;

    // perform 64 bit multiply -> returning 128 bit result
    struct uint128 sig128Z = softfloat_mul64To128( sigA, sigB );
    uint64_t sigZ = sig128Z.v64 | (sig128Z.v0 != 0);
    if ( sigZ < UINT64_C( 0x4000000000000000 ) ) {
        --expZ;
        sigZ <<= 1;
    }
    // perform rounding, you've seen this before with f64_add
    return softfloat_roundPackToF64( 0 /* signZ */, expZ, sigZ );
infArg:
    uZ.ui = packToF64UI( 0 /* signZ */, 0x7FF, 0 );
    return uZ.f;
}
```

Take two FP numbers, a and b. Then,

$$a = m \times 2^e,$$

$$b = n \times 2^u.$$

Multiplying,

$$a \times b = (m \times n) \times 2^{e+u},$$

which is much, much easier to do in software.

(compared to FP addition.)

37. (3.FP) In short

- RandomX “forces” nontrivial FP rounding on you to maximise die space on an ASIC.
- WASM, JS, Rust, Python, etc only have access to $\mathbf{RN}$, what now?

- Error free transforms allow you to impl. $\mathbf{RD}$, $\mathbf{RU}$, $\mathbf{RZ}$ in terms of $\mathbf{RN}$, without using slow software impl.
- This speeds up emulation of addition considerably, multiplication, a little.

  (EFTs still win!)

- The bane of slow software implementations is unpredictable branches.

  (the cost of multiple FP operations $<$ cost of a single unpredictable branch)

This allows RandomX.js to be fast!

>> Finishing Up

39. Q&A - The End + Reference material

Thank you everyone! It’s Q&A time.

Slides cut to make time. Easy program problem, memory-hardness, scratchpad, longer exposition on the ASIC problem, relaxed SIMD caveats, RandomX FP assumptions.

Error Free Transforms (EFTs)

- Muller, J.-M. et al. (2018) Handbook of Floating-Point Arithmetic.
- Arnold, J. (2012) “Floating-Point Arithmetic.”

  https://indico.cern.ch/event/166141/sessions/125684/attachments/201414/282782/Arnold-FPWorkshop-Print.pdf
- (My question on SO) https://stackoverflow.com/questions/78776730

RandomX

- (docs) https://github.com/tevador/RandomX/blob/master/doc/docs.md
- (specs) https://github.com/tevador/RandomX/blob/master/doc/specs.md

---

RandomX.js

- (perf tracking) https://github.com/l1mey112/randomx.js/issues/1