

# Быстрые, детерминированные и верифицируемые вычисления на WebAssembly

Воронов Михаил

R&D engineer at Fluence Labs

fluence

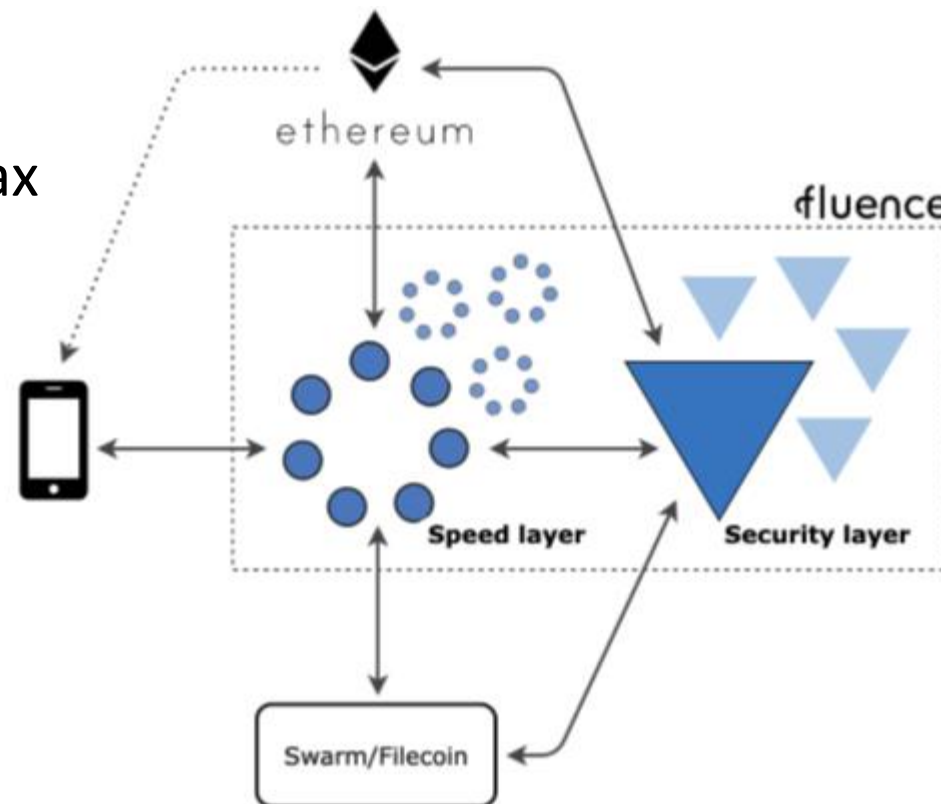
# План выступления

- Архитектура вычислительной сети Fluence
- Трансляция WebAssembly в JVM
- Детерминизм вычислений
- Наш подход к verification game

# Архитектура Fluence

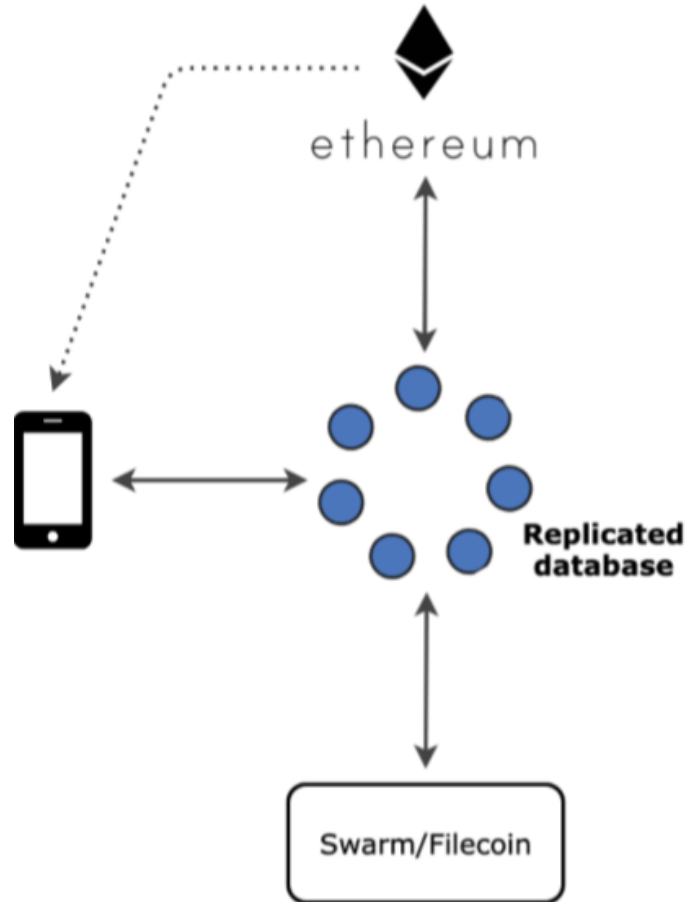
# Что такое Fluence?

- Похоже на AWS Lambda, но работает постоянно
- Wasm программа запускается в маленьких кластерах (4-21 нод)
- Полностью децентрализованно
- Два уровня: speed и security
- Три типа ролей: майнеры, разработчики, пользователи

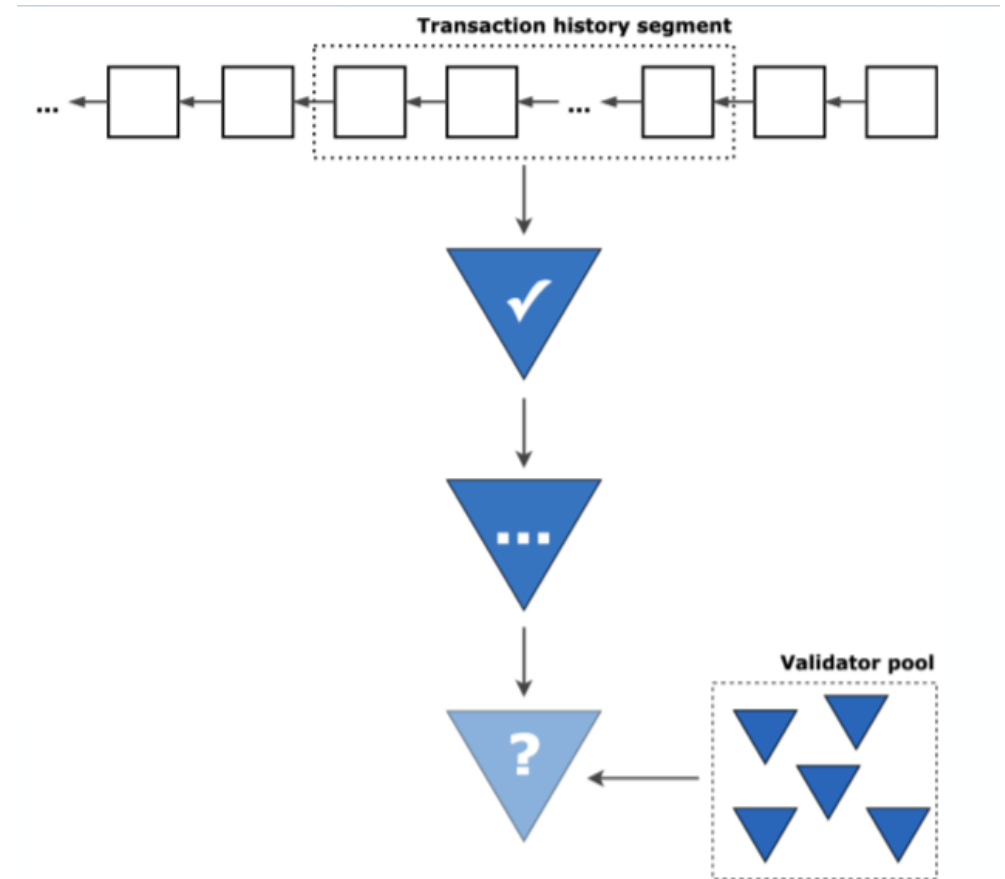


# Архитектура Fluence

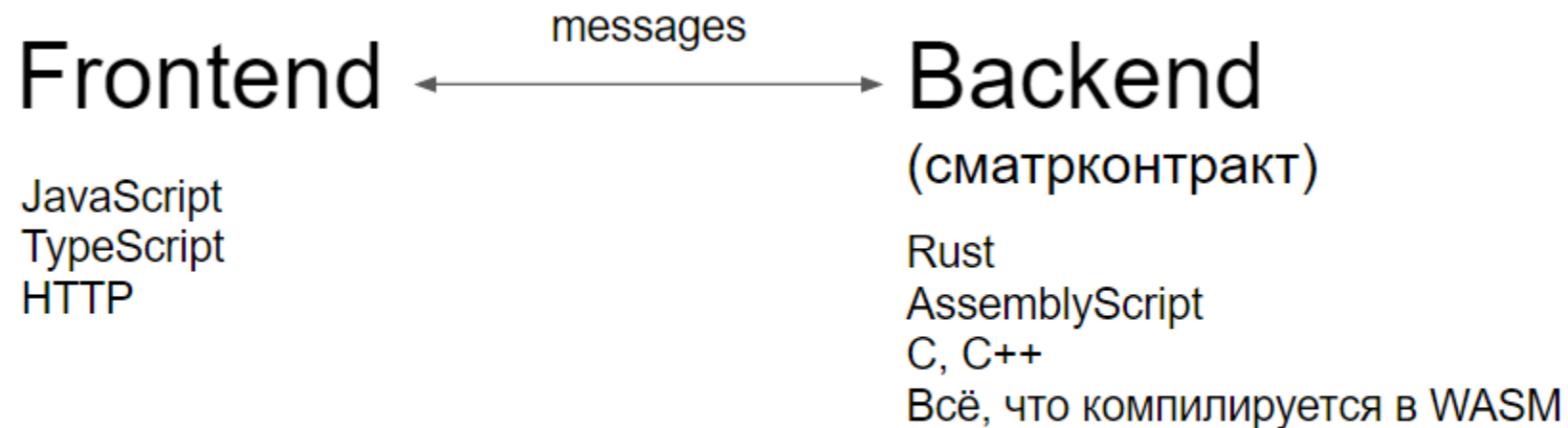
## Speed layer



## Security layer



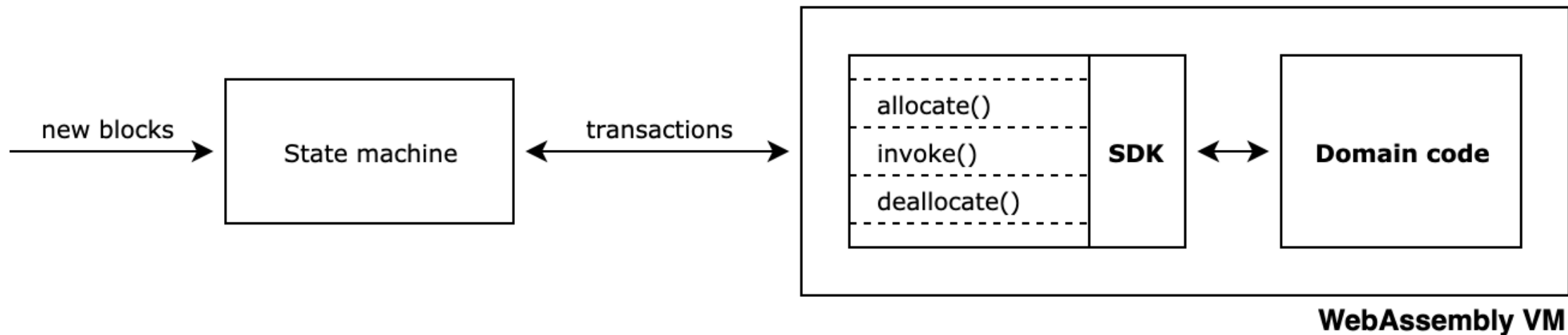
# Архитектура Fluence



# Computation machine

Разработчик загружает WebAssembly код, который инстанцируется и обрабатывается нодой.

Fluence node



# Как сделать вычисления на Wasm быстрыми

(на примере трансляции в JVM с помощью Asmble)



# Трансляция Wasm -> JVM

- Каждый загруженный пользователем Wasm код транслируется в JVM класс.
- JVM и Wasm - стековые виртуальные машины, многие команды которых могут быть легко переведены друг в друга.

```
i32.const 8    ;; wasm stack = [8]  
i32.const 42   ;; wasm stack = [42, 8]  
i32.add        ;; wasm stack = [50]
```



```
bipush 8      ;; jvm stack = [8]  
bipush 42     ;; jvm stack = [42, 8]  
iadd          ;; jvm stack = [50]
```

# Трансляция Wasm -> JVM

- {i32, i64, f32, f64} -> {int, long, float, double}
- Module -> JVM класс
- Функции Wasm -> функции JVM
- Импорты -> параметры конструктора
- Экспорты -> публичные методы класса
- Куча Wasm -> ByteBuffer
- Глобальные переменные -> поля генерируемого класса
- Table (для call\_indirect) -> массив MethodHandle

# Пример трансляции

```
uint64_t fib(uint64_t i) {  
    if (i == 1) {  
        return 1;  
    }  
    if (i == 0) {  
        return 0;  
    }  
    return fib(i-1) * fib(i-2);  
}
```



```
(module  
  (table 0 anyfunc)  
  (memory $0 1)  
  (export "memory" (memory $0))  
  (export "_Z3fiby" (func $_Z3fiby))  
  (func $_Z3fiby (; 0 ;) (param $0 i64) (result i64)  
    (local $1 i64)  
    (set_local $1  
      (i64.const 1)  
    )  
    ...  
    (i64.mul  
      (call $_Z3fiby  
        (i64.add  
          (get_local $0)  
          (i64.const -1)  
        )  
      )  
      (call $_Z3fiby  
        (i64.add  
          (get_local $0)  
          (i64.const -2)  
        )  
      )  
    )  
    ...  
  )
```

# Пример трансляции

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "_Z3fiby" (func $_Z3fiby))
  (func $_Z3fiby (; 0 ;) (param $0 i64) (result i64)
    (local $1 i64)
    (set_local $1
      (i64.const 1)
    )
    ...
    (i64.mul
      (call $_Z3fiby
        (i64.add
          (get_local $0)
          (i64.const -1)
        )
      )
      (call $_Z3fiby
        (i64.add
          (get_local $0)
          (i64.const -2)
        )
      )
    )
    ...
  )
```



```
@WasmModule
public class Fibonacci {
    private final ByteBuffer memory;
    private final MethodHandle[] table;

    public Fibonacci(int var1) {
        this(ByteBuffer.allocateDirect(var1));
    }

    public Fibonacci(ByteBuffer var1) {
        this.table = new MethodHandle[0];
        this.memory = var1;
        ((ByteBuffer)var1.limit(65536)).order(ByteOrder.LITTLE_ENDIAN);
    }

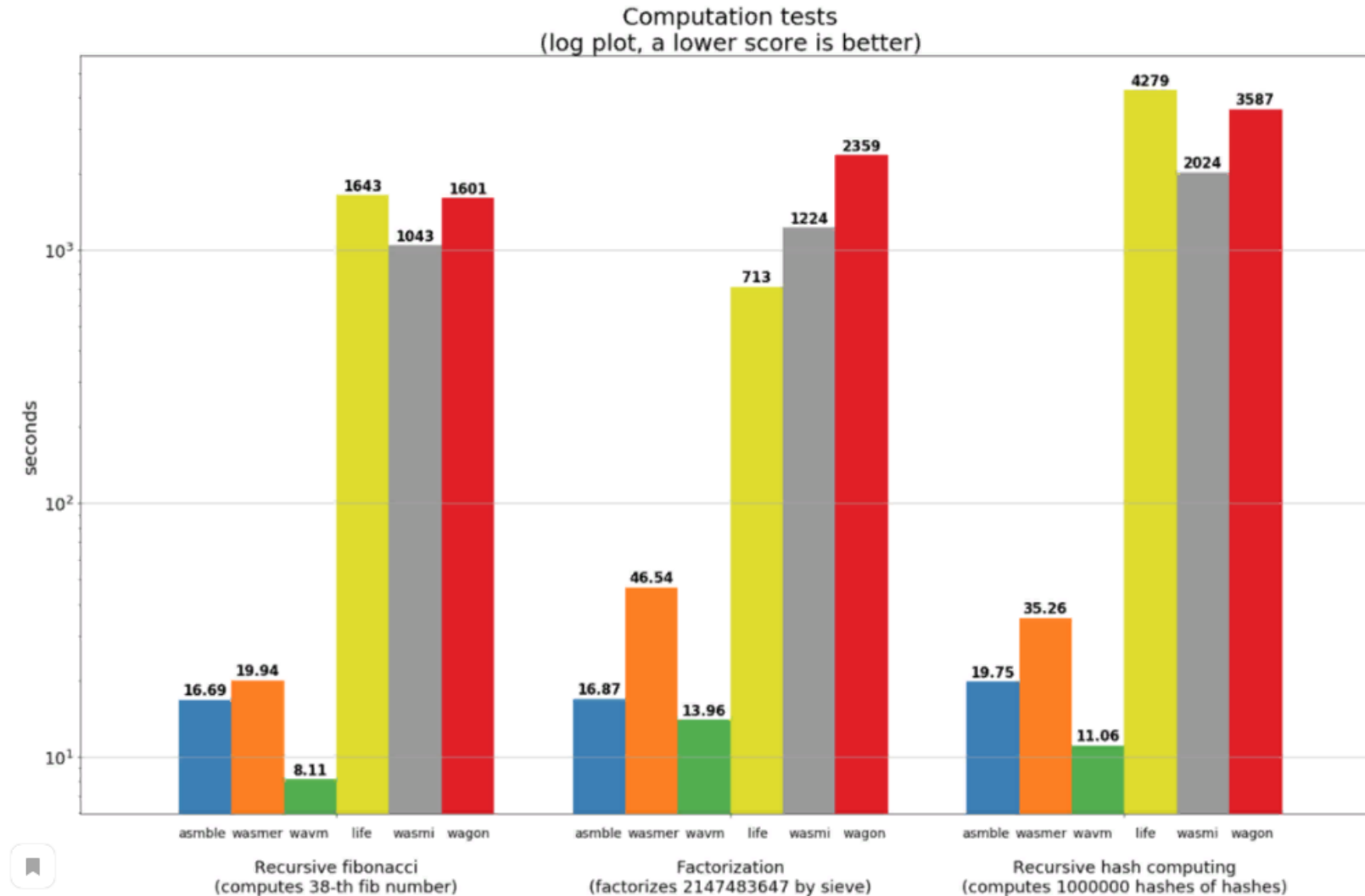
    private long _Z3fiby0(long var1) {
        long var3 = 1L;
        if (var1 != 1L) {
            var3 = 0L;
            if (var1 != 0L) {
                return this._Z3fiby0(var1 + -1L) * this._Z3fiby0(var1 + -2L);
            }
        }

        return var3;
    }
}
```

# Подводные камни

- Размер строк и методов в JVM ограничен  $2^{16}$  байт
- Семейство инструкций  $\{i32, i64\}.trunc\_s/\{f32, f64\}$  проверяет аргумент на переполнение, тогда как  $i2f$  нет
- В JVM нет возможности посчитать хэш от стека
- В JVM нет прямых аналогов для части Wasm инструкций (например, семейства  $\{*\}.reinterpret/\{*\}$ )
- В Wasm гораздо строже правила работы со стеком при завершении блока

# Насколько это быстро?



# Как сделать вычисления на Wasm детерминированными

(на примере трансляции в jvm)

# Детерминизм Wasm

Wasm имеет три источника недетерминизма:

1. Вызов функций из хоста
2. Исчерпание ресурсов VM
3. NaN payloads

```
(import "env" "abortStackOverflow" (func $env.abortStackOverflow (type $t2)))
(import "env" "nullFunc_ii" (func $env.nullFunc_ii (type $t2)))
(import "env" "nullFunc_iiii" (func $env.nullFunc_iiii (type $t2)))
(import "env" "__lock" (func $env.__lock (type $t2)))
(import "env" "__setErrNo" (func $env.__setErrNo (type $t2)))
(import "env" "__syscall140" (func $env.__syscall140 (type $t3)))
(import "env" "__syscall146" (func $env.__syscall146 (type $t3)))
(import "env" "__syscall154" (func $env.__syscall154 (type $t3)))
(import "env" "__syscall16" (func $env.__syscall16 (type $t3)))
(import "env" "__unlock" (func $env.__unlock (type $t2)))
(import "env" "_emscripten_get_heap_size" (func $env._emscripten_get_heap_size
(type $t4)))
(import "env" "_emscripten_memcpy_big" (func $env._emscripten_memcpy_big (type
$t0)))
(import "env" "_emscripten_resize_heap" (func $env._emscripten_resize_heap
(type $t1)))
(import "env" "abortOnCannotGrowMemory" (func $env.abortOnCannotGrowMemory
(type $t1)))
(import "env" "__memory_base" (global $env.__memory_base i32))
(import "env" "__table_base" (global $env.__table_base i32))
(import "env" "DYNAMICTOP_PTR" (global $env.DYNAMICTOP_PTR i32))
(import "env" "tempDoublePtr" (global $env.tempDoublePtr i32))
(import "global" "NaN" (global $global.NaN f64))
(import "global" "Infinity" (global $global.Infinity f64))
(import "env" "memory" (memory $env.memory 256 256))
```



# Вызов хостовых функций

В описываемом подходе:

- блокируются любые импорты хостовых функций
- на данный момент в качестве persistent storage используется RAM с сохранением промежуточных результатов в Swarm
- в будущем планируется добавить поддержку подмножества WASI сисколов, отвечающих за работу с диском и получение текущего времени
- детерминированный рандом (ГПСЧ) позволяет не заботиться о детерминизме внутреннего представления некоторых структур данных (например, хэш-таблиц)

# Исчерпание ресурсов VM

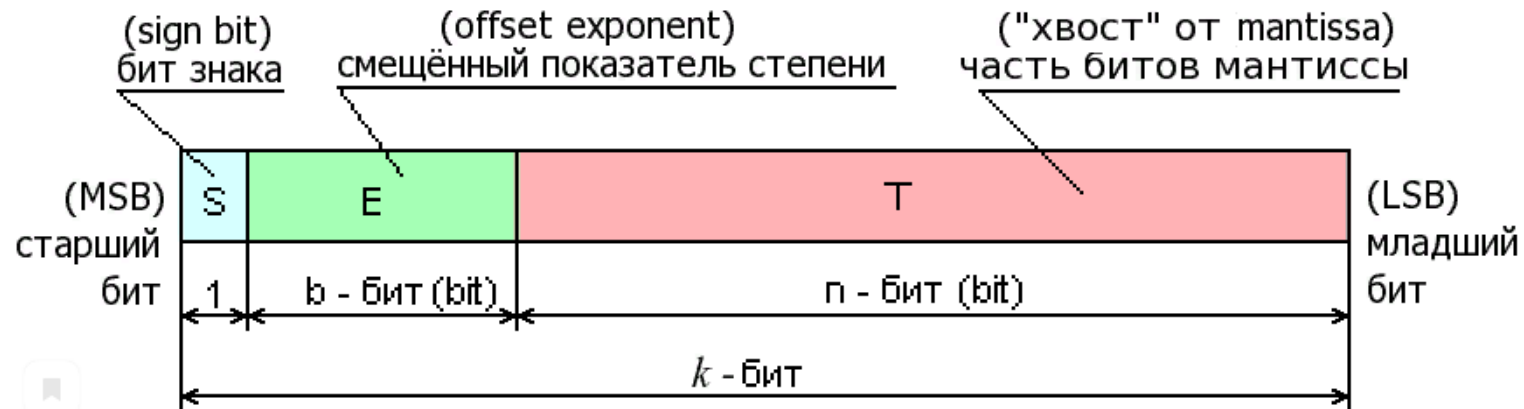
Основные ресурсы Wasm VM, которые могут повлиять на детерминизм: heap, stack, table.

В описываемом подходе:

- разработчик задаёт аллоцируемое количество памяти
- аллокация всех ресурсов происходит на старте VM
- `grow_memory` всегда возвращает -1

# NaN payloads

- NaN (not-a-number) – особое состояние floating-point числа, которое может возникнуть в случае, если предыдущая математическая операция завершилась с неопределённым результатом или если в ячейку памяти попало не удовлетворяющее условиям число.
- Floating-point число, удовлетворяющее IEEE 754-2008, имеет следующее представление в памяти:



- Согласно IEEE 754-2008, NaN число в битовом представлении должно иметь 11...11 в записи экспоненты и не 0 в мантиссе.

# NaN payloads

- Согласно спецификации Wasm, «instructions only output canonical NaNs with a non-deterministic sign bit, unless (2) if an input is a noncanonical NaN, then the output NaN is non-deterministic».
- На данный момент, мы планируем зафиксировать NaN паттерн и нормализовывать каждую инструкцию, оперирующую floating-point числами.
- Таким образом, в отличие от многих других платформ, у пользователя будет возможность floating-point с overhead'ом или же использовать только вычисления с целыми числами.

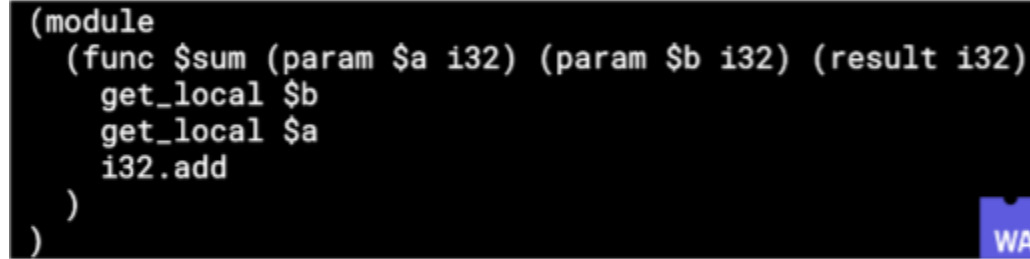
# Как сделать вычисления на Wasn детерминированными

(на примере использования verification game)

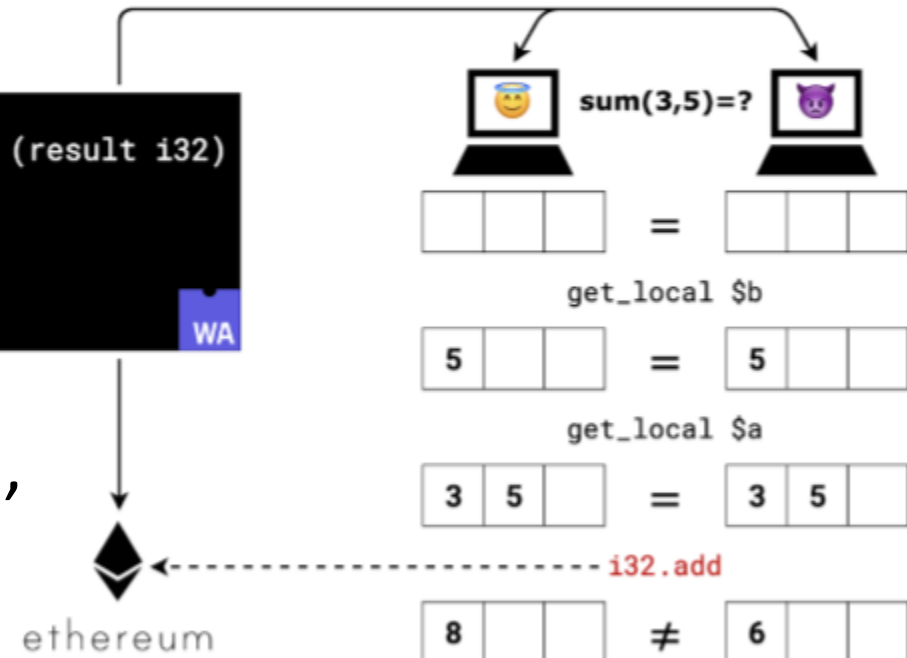
# Verification game

- Валидатор находит некорректный результат
- Открывает диспут с нодой

```
(module
  (func $sum (param $a i32) (param $b i32) (result i32)
    get_local $b
    get_local $a
    i32.add
  )
)
```



- Вычисление сужается до конкретной операции,
- на которой произошло расхождение
- Данная операция отправляется в Ethereum смарт контракт
- Он повторяет вычисление и находит виноватого



# Хэш от состояния VM

Подсчёт хэша происходит от всех мутабельных частей Wasm VM:

- heap
- dtack
- globals
- table
- gas counter
- instruction pointer

# Хэш от памяти

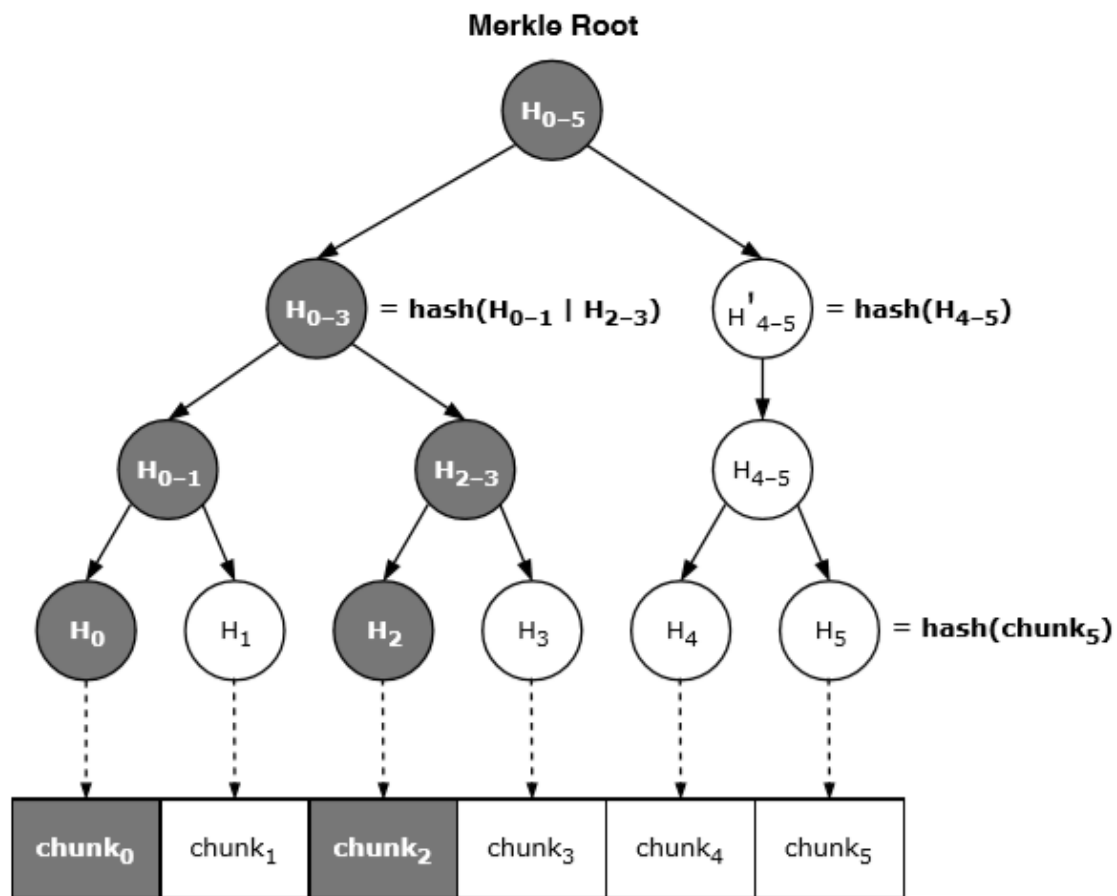
Расчёт хэша от памяти должен соответствовать следующим требованиям:

- должен быть максимально быстрым
- необходимость генерации пруфов для отправки в контракт
- должен быть сопоставим по вычислительным ресурсам с «полезными вычислениями»



# Хэш от памяти

Наиболее оптимально данным требованиям соответствует Merkle hash:



# Shadow stack

Из-за невозможности детерминированно посчитать хэш от operand stack jvm был введён отдельный Shadow stack

```
class ShadowStack {  
    private Stack<Object> backingStack;  
  
    // pushes the value to the top of the backing stack  
    public void push(int value)  
  
    // removes 2 values from the top of the backing stack  
    public void remove2()  
}
```

---

# Shadow stack

Вспомним пример трансляции простейшей арифметической операции из Wasm в JVM:

```
i32.const 8    ;; wasm stack = [8]  
i32.const 42   ;; wasm stack = [42, 8]  
i32.add        ;; wasm stack = [50]
```



---

```
bipush 8      ;; jvm stack = [8]  
bipush 42     ;; jvm stack = [42, 8]  
iadd          ;; jvm stack = [50]
```

---

# Shadow stack

С Shadow stack он будет выглядеть следующим образом:

```
i32.const 8    ;; wasm stack = [8]
i32.const 42   ;; wasm stack = [42, 8]
i32.add        ;; wasm stack = [50]
```



```
;; corresponds to 'i32.const 8'
bipush 8      ;; jvm stack = [8] | shadow = []
dup           ;; jvm stack = [8, 8] | shadow = []
aload_0       ;; jvm stack = [@shadow, 8, 8] | shadow = []
invokevirtual #1 ;; jvm stack = [8] | shadow = [8]

;; corresponds to 'i32.const 42'
bipush 42     ;; jvm stack = [42, 8] | shadow = [8]
dup           ;; jvm stack = [42, 42, 8] | shadow = [8]
aload_0       ;; jvm stack = [@shadow, 42, 42, 8] | shadow = [8]
invokevirtual #1 ;; jvm stack = [42, 8] | shadow = [42, 8]

;; corresponds to 'i32.add'
iadd          ;; jvm stack = [50] | shadow = [42, 8]
aload_0       ;; jvm stack = [@shadow, 50] | shadow = [42, 8]
invokevirtual #2 ;; jvm stack = [50] | shadow = []
dup           ;; jvm stack = [50, 50] | shadow = []
aload_0       ;; jvm stack = [@shadow, 50, 50] | shadow = []
invokevirtual #1 ;; jvm stack = [50] | shadow = [50]    28
```

# Подсчёт газа

Подсчёт газа осуществляется на уровне базовых блоков:

```
00 block $0
01   loop $1

;; <block A: begin>
02   get_local $0 ;; 2¢
03   i32.const 9   ;; 2¢
04   i32.gt_s      ;; 3¢
05   br_if $0      ;; 4¢, jumps to #12 if i > 9
;; <block A: end>

;; <block B: begin>
06   get_local $0 ;; 2¢
07   i32.const 1   ;; 2¢
08   i32.add       ;; 3¢
09   set_local $0  ;; 2¢
10   br $1         ;; 4¢, jumps to #02
;; <block B: end>

11   end label $1
12 end label $0
```

Список инструкций Wasm,  
генерирующих базовые блоки:

```
br
br_if
br_table
else
if
loop
return
end
```

# Дуализм computation machine

Из-за наличия двух уровней (speed и security) CM работает в двух режимах: режиме вычисления и режиме верификации.

## **Режим вычисления:**

- наиболее быстрое выполнение кода

## **Режим верификации:**

- поиск инструкции, на которой произошло расхождение состояния
- выключен JIT
- дополнительно подсчитывается EIC (executed instruction counter) и IP (instruction pointer).

Спасибо за внимание!