RandomX

RandomX is a proof of work (PoW) algorithm which was designed to close the gap between general-purpose CPUs and specialized hardware. The core of the algorithm is a simulation of a virtual CPU.

Table of contents

1. Definitions

1.1 General definitions

Hash256 and **Hash512** refer to the Blake2b hashing function with a 256-bit and 512-bit output size, respectively.

Floating point format refers to the IEEE-754 double precision floating point format with a sign bit, 11-bit exponent and 52-bit fraction.

Argon2d is a tradeoff-resistant variant of Argon2, a memory-hard password derivation function.

AesGenerator1R refers to an AES-based pseudo-random number generator described in chapter 3.2. It's initialized with a 512-bit seed value and is capable of producing more than 10 bytes per clock cycle.

AesGenerator4R is a slower but more secure AES-based pseudo-random number generator described in chapter 3.3. It's initialized with a 512-bit seed value.

AesHash1R refers to an AES-based fingerprinting function described in chapter 3.4. It's capable of processing more than 10 bytes per clock cycle and produces a 512-bit output.

BlakeGenerator refers to a custom pseudo-random number generator described in chapter 3.5. It's based on the Blake2b hashing function.

SuperscalarHash refers to a custom diffusion function designed to run efficiently on superscalar CPUs (see chapter 7). It transforms a 64-byte input value into a 64-byte output value.

Virtual Machine or **VM** refers to the RandomX virtual machine as described in chapter 4.

Programming the VM refers to the act of loading a program and configuration into the VM. This is described in chapter 4.5.

Executing the VM refers to the act of running the program loop as described in chapter 4.6.

Scratchpad refers to the workspace memory of the VM. The whole scratchpad is structured into 3 levels: L3 -> L2 -> L1 with each lower level being a subset of the higher levels.

Register File refers to a 256-byte sequence formed by concatenating VM registers in little-endian format in the following order: `r0` - `r7`, `f0` - `f3`, `e0` - `e3` and `a0` - `a3`.

Program Buffer refers to the buffer from which the VM reads instructions.

Cache refers to a read-only buffer initialized by Argon2d as described in chapter 7.1.

Dataset refers to a large read-only buffer described in chapter 7. It is constructed from the Cache using the SuperscalarHash function.

1.2 Configurable parameters

RandomX has several configurable parameters that are listed in Table 1.2.1 with their default values.

Table 1.2.1 - Configurable parameters

parameter	description	default value
`RANDOMX_ARGON_MEMORY`	The number of 1 KiB Argon2 blocks in the Cache	`262144`
`RANDOMX_ARGON_ITERATIONS`	The number of Argon2d iterations for Cache initialization	`3`

parameter	description	default value
`RANDOMX_ARGON_LANES`	The number of parallel lanes for Cache initialization	`1`
`RANDOMX_ARGON_SALT`	Argon2 salt	`"RandomX\x03"`
`RANDOMX_CACHE_ACCESSES`	The number of random Cache accesses per Dataset item	`8`
`RANDOMX_SUPERSCALAR_LATENCY`	Target latency for SuperscalarHash (in cycles of the reference CPU)	`170`
`RANDOMX_DATASET_BASE_SIZE`	Dataset base size in bytes	`2147483648`
`RANDOMX_DATASET_EXTRA_SIZE`	Dataset extra size in bytes	`33554368`
`RANDOMX_PROGRAM_SIZE`	The number of instructions in a RandomX program	`256`
`RANDOMX_PROGRAM_ITERATIONS`	The number of iterations per program	`2048`
`RANDOMX_PROGRAM_COUNT`	The number of programs per hash	`8`
`RANDOMX_JUMP_BITS`	Jump condition mask size in bits	`8`
`RANDOMX_JUMP_OFFSET`	Jump condition mask offset in bits	`8`
`RANDOMX_SCRATCHPAD_L3`	Scratchpad L3 size in bytes	`2097152`
`RANDOMX_SCRATCHPAD_L2`	Scratchpad L2 size in bytes	`262144`
`RANDOMX_SCRATCHPAD_L1`	Scratchpad L1 size in bytes	`16384`

Instruction frequencies listed in Tables 5.2.1, 5.3.1, 5.4.1 and 5.5.1 are also configurable.

2. Algorithm description

The RandomX algorithm accepts two input values:

- String `K` with a size of 0-60 bytes (key)
- String `H` of arbitrary length (the value to be hashed)

and outputs a 256-bit result `R`.

The algorithm consists of the following steps:

1. The Dataset is initialized using the key value `K` (described in chapter 7).
2. 64-byte seed `S` is calculated as `S = Hash512(H)`.
3. Let `gen1 = AesGenerator1R(S)`.
4. The Scratchpad is filled with `RANDOMX_SCRATCHPAD_L3` random bytes using generator `gen1`.
5. Let `gen4 = AesGenerator4R(gen1.state)` (use the final state of `gen1`).
6. The value of the VM register `fprc` is set to 0 (default rounding mode - chapter 4.3).
7. The VM is programmed using `128 + 8 * RANDOMX_PROGRAM_SIZE` random bytes using generator `gen4` (chapter 4.5).
8. The VM is executed (chapter 4.6).
9. A new 64-byte seed is calculated as `S = Hash512(RegisterFile)`.
10. Set `gen4.state = S` (modify the state of the generator).
11. Steps 7-10 are performed a total of `RANDOMX_PROGRAM_COUNT` times. The last iteration skips steps 9 and 10.
12. Scratchpad fingerprint is calculated as `A = AesHash1R(Scratchpad)`.
13. Bytes 192-255 of the Register File are set to the value of `A`.
14. Result is calculated as `R = Hash256(RegisterFile)`.

The input of the `Hash512` function in step 9 is the following 256 bytes:

```
+---------------------------------+
|         registers r0-r7         | (64 bytes)
+---------------------------------+
|         registers f0-f3         | (64 bytes)
+---------------------------------+
|         registers e0-e3         | (64 bytes)
+---------------------------------+
|         registers a0-a3         | (64 bytes)
+---------------------------------+
```

The input of the `Hash256` function in step 14 is the following 256 bytes:

```
+---------------------------------+
|         registers r0-r7         | (64 bytes)
+---------------------------------+
|         registers f0-f3         | (64 bytes)
+---------------------------------+
```

```
|         registers e0-e3         | (64 bytes)
+---------------------------------+
|      AesHash1R(Scratchpad)      | (64 bytes)
+---------------------------------+
```

3 Custom functions

3.1 Definitions

Two of the custom functions are based on the Advanced Encryption Standard (AES).

AES encryption round refers to the application of the ShiftRows, SubBytes and MixColumns transformations followed by a XOR with the round key.

AES decryption round refers to the application of inverse ShiftRows, inverse SubBytes and inverse MixColumns transformations followed by a XOR with the round key.

3.2 AesGenerator1R

AesGenerator1R produces a sequence of pseudo-random bytes.

The internal state of the generator consists of 64 bytes arranged into four columns of 16 bytes each. During each output iteration, every column is decrypted (columns 0, 2) or encrypted (columns 1, 3) with one AES round using the following round keys (one key per column):

```
key0 = 53 a5 ac 6d 09 66 71 62 2b 55 b5 db 17 49 f4 b4
key1 = 07 af 7c 6d 0d 71 6a 84 78 d3 25 17 4e dc a1 0d
key2 = f1 62 12 3f c6 7e 94 9f 4f 79 c0 f4 45 e3 20 3e
key3 = 35 81 ef 6a 7c 31 ba b1 88 4c 31 16 54 91 16 49
```

These keys were generated as:

```
key0, key1, key2, key3 = Hash512("RandomX AesGenerator1R keys")
```

Single iteration produces 64 bytes of output which also become the new generator state.

```
state0 (16 B)    state1 (16 B)    state2 (16 B)    state3 (16 B)
     |                |                |                |
 AES decrypt      AES encrypt      AES decrypt      AES encrypt
```

```
   (key0)          (key1)          (key2)          (key3)
     |               |               |               |
     v               v               v               v
  state0'         state1'         state2'         state3'
```

3.3 AesGenerator4R

AesGenerator4R works similar way as AesGenerator1R, except it uses 4 rounds per column. Columns 0 and 1 use a different set of keys than columns 2 and 3.

```
state0 (16 B)    state1 (16 B)    state2 (16 B)    state3 (16 B)
     |               |               |               |
 AES decrypt      AES encrypt      AES decrypt      AES encrypt
   (key0)          (key0)          (key4)          (key4)
     |               |               |               |
     v               v               v               v
 AES decrypt      AES encrypt      AES decrypt      AES encrypt
   (key1)          (key1)          (key5)          (key5)
     |               |               |               |
     v               v               v               v
 AES decrypt      AES encrypt      AES decrypt      AES encrypt
   (key2)          (key2)          (key6)          (key6)
     |               |               |               |
     v               v               v               v
 AES decrypt      AES encrypt      AES decrypt      AES encrypt
   (key3)          (key3)          (key7)          (key7)
     |               |               |               |
     v               v               v               v
  state0'         state1'         state2'         state3'
```

AesGenerator4R uses the following 8 round keys:

```
key0 = dd aa 21 64 db 3d 83 d1 2b 6d 54 2f 3f d2 e5 99
key1 = 50 34 0e b2 55 3f 91 b6 53 9d f7 06 e5 cd df a5
key2 = 04 d9 3e 5c af 7b 5e 51 9f 67 a4 0a bf 02 1c 17
key3 = 63 37 62 85 08 5d 8f e7 85 37 67 cd 91 d2 de d8
key4 = 73 6f 82 b5 a6 a7 d6 e3 6d 8b 51 3d b4 ff 9e 22
key5 = f3 6b 56 c7 d9 b3 10 9c 4e 4d 02 e9 d2 b7 72 b2
key6 = e7 c9 73 f2 8b a3 65 f7 0a 66 a9 2b a7 ef 3b f6
key7 = 09 d6 7c 7a de 39 58 91 fd d1 06 0c 2d 76 b0 c0
```

These keys were generated as:

```
key0, key1, key2, key3 = Hash512("RandomX AesGenerator4R keys 0-3")
key4, key5, key6, key7 = Hash512("RandomX AesGenerator4R keys 4-7")
```

3.4 AesHash1R

AesHash1R calculates a 512-bit fingerprint of its input.

AesHash1R has a 64-byte internal state, which is arranged into four columns of 16 bytes each. The initial state is:

```
state0 = 0d 2c b5 92 de 56 a8 9f 47 db 82 cc ad 3a 98 d7
state1 = 6e 99 8d 33 98 b7 c7 15 5a 12 9e f5 57 80 e7 ac
state2 = 17 00 77 6a d0 c7 62 ae 6b 50 79 50 e4 7c a0 e8
state3 = 0c 24 0a 63 8d 82 ad 07 05 00 a1 79 48 49 99 7e
```

The initial state vectors were generated as:

```
state0, state1, state2, state3 = Hash512("RandomX AesHash1R state")
```

The input is processed in 64-byte blocks. Each input block is considered to be a set of four AES round keys `key0`, `key1`, `key2`, `key3`. Each state column is encrypted (columns 0, 2) or decrypted (columns 1, 3) with one AES round using the corresponding round key:

When all input bytes have been processed, the state is processed with two additional AES rounds with the following extra keys (one key per round, same pair of keys for all columns):

```
xkey0 = 89 83 fa f6 9f 94 24 8b bf 56 dc 90 01 02 89 06
xkey1 = d1 63 b2 61 3c e0 f4 51 c6 43 10 ee 9b f9 18 ed
```

The extra keys were generated as:

```
xkey0, xkey1 = Hash256("RandomX AesHash1R xkeys")
```

```
state0 (16 B)    state1 (16 B)    state2 (16 B)    state3 (16 B)
     |                |                |                |
 AES encrypt      AES decrypt      AES encrypt      AES decrypt
   (xkey0)          (xkey0)          (xkey0)          (xkey0)
     |                |                |                |
     v                v                v                v
 AES encrypt      AES decrypt      AES encrypt      AES decrypt
   (xkey1)          (xkey1)          (xkey1)          (xkey1)
     |                |                |                |
     v                v                v                v
finalState0      finalState1      finalState2      finalState3
```

The final state is the output of the function.

3.5 BlakeGenerator

BlakeGenerator is a simple pseudo-random number generator based on the Blake2b hashing function. It has a 64-byte internal state `S`.

3.5.1 Initialization

The internal state is initialized from a seed value `K` (0-60 bytes long). The seed value is written into the internal state and padded with zeroes. Then the internal state is initialized as `S = Hash512(S)`.

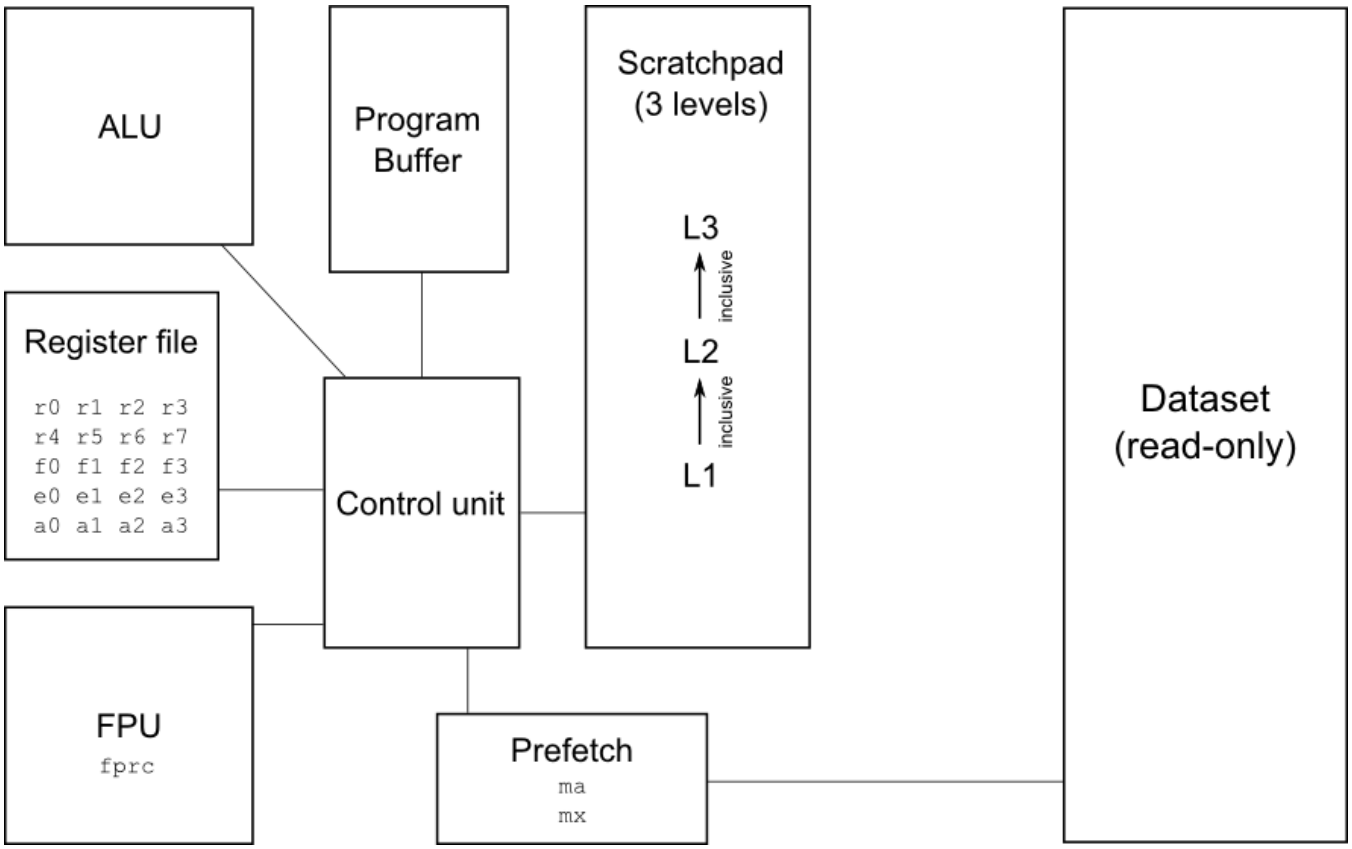
3.5.2 Random number generation

The generator can generate 1 byte or 4 bytes at a time by supplying data from its internal state `S`. If there are not enough unused bytes left, the internal state is reinitialized as `S = Hash512(S)`.

4. Virtual Machine

The components of the RandomX virtual machine are summarized in Fig. 4.1.

Figure 4.1 - Virtual Machine



The VM is a complex instruction set computer (CISC). All data are loaded and stored in little-endian byte order. Signed integer numbers are represented using two's complement.

4.1 Dataset

Dataset is described in detail in chapter 7. It's a large read-only buffer. Its size is equal to `RANDOMX_DATASET_BASE_SIZE + RANDOMX_DATASET_EXTRA_SIZE` bytes. Each program uses only a random subset of the Dataset of size `RANDOMX_DATASET_BASE_SIZE`. All Dataset accesses read an aligned 64-byte item.

4.2 Scratchpad

Scratchpad represents the workspace memory of the VM. Its size is `RANDOMX_SCRATCHPAD_L3` bytes and it's divided into 3 "levels":

- The whole scratchpad is the third level "L3".
- The first `RANDOMX_SCRATCHPAD_L2` bytes of the scratchpad is the second level "L2".
- The first `RANDOMX_SCRATCHPAD_L1` bytes of the scratchpad is the first level "L1".

The scratchpad levels are inclusive, i.e. L3 contains both L2 and L1 and L2 contains L1.

To access a particular scratchpad level, bitwise AND with a mask according to table 4.2.1 is applied to the memory address.

Table 4.2.1: Scratchpad access masks

Level	8-byte aligned mask	64-byte aligned mask
L1	`(RANDOMX_SCRATCHPAD_L1 - 1) & ~7`	-
L2	`(RANDOMX_SCRATCHPAD_L2 - 1) & ~7`	-
L3	`(RANDOMX_SCRATCHPAD_L3 - 1) & ~7`	`(RANDOMX_SCRATCHPAD_L3 - 1) & ~63`

4.3 Registers

The VM has 8 integer registers `r0` - `r7` (group R) and a total of 12 floating point registers split into 3 groups: `f0` - `f3` (group F), `e0` - `e3` (group E) and `a0` - `a3` (group A). Integer registers are 64 bits wide, while floating point registers are 128 bits wide and contain a pair of numbers in floating point format. The lower and upper half of floating point registers are not separately addressable.

Additionally, there are 3 internal registers `ma`, `mx` and `fprc`.

Integer registers `r0` - `r7` can be the source or the destination operands of integer instructions or may be used as address registers for accessing the Scratchpad.

Floating point registers `a0` - `a3` are read-only and their value is fixed for a given VM program. They can be the source operand of any floating point instruction. The value of these registers is restricted to the interval `[1, 4294967296)`.

Floating point registers `f0` - `f3` are the "additive" registers, which can be the destination of floating point addition and subtraction instructions. The absolute value of these registers will not exceed about `3.0e+14`.

Floating point registers `e0` - `e3` are the "multiplicative" registers, which can be the destination of floating point multiplication, division and square root instructions. Their value is always positive.

`ma` and `mx` are the memory registers. Both are 32 bits wide. `ma` contains the memory address of the next Dataset read and `mx` contains the address of the next Dataset prefetch. The values of `ma` and `mx` registers are always aligned to be a multiple of 64.

The 2-bit `fprc` register determines the rounding mode of all floating point operations according to Table 4.3.1. The four rounding modes are defined by the IEEE 754 standard.

Table 4.3.1: Rounding modes

`fprc`	rounding mode
0	roundTiesToEven
1	roundTowardNegative
2	roundTowardPositive
3	roundTowardZero

4.3.1 Group F register conversion

When an 8-byte value read from the memory is to be converted to an F group register value or operand, it is interpreted as a pair of 32-bit signed integers (in little endian, two's complement format) and converted to floating point format. This conversion is exact and doesn't need rounding because only 30 bits of the fraction significand are needed to represent the integer value.

4.3.2 Group E register conversion

When an 8-byte value read from the memory is to be converted to an E group register value or operand, the same conversion procedure is applied as for F group registers (see 4.3.1) with additional post-processing steps for each of the two floating point values:

1. The sign bit is set to `0`.
2. Bits 0-2 of the exponent are set to the constant value of 011_2.
3. Bits 3-6 of the exponent are set to the value of the exponent mask described in chapter 4.5.6. This value is fixed for a given VM program.
4. The bottom 22 bits of the fraction significand are set to the value of the fraction mask described in chapter 4.5.6. This value is fixed for a given VM program.

4.4 Program buffer

The Program buffer stores the program to be executed by the VM. The program consists of `RANDOMX_PROGRAM_SIZE` instructions. Each instruction is encoded by an 8-byte word. The instruction set is described in chapter 5.

4.5 VM programming

The VM requires `128 + 8 * RANDOMX_PROGRAM_SIZE` bytes to be programmed. This is split into two parts:

- `128` bytes of configuration data = 16 quadwords (16×8 bytes), used according to Table 4.5.1
- `8 * RANDOMX_PROGRAM_SIZE` bytes of program data, copied directly into the Program Buffer

Table 4.5.1 - Configuration data

quadword	description
0	initialize low half of register `a0`
1	initialize high half of register `a0`
2	initialize low half of register `a1`
3	initialize high half of register `a1`
4	initialize low half of register `a2`
5	initialize high half of register `a2`
6	initialize low half of register `a3`
7	initialize high half of register `a3`
8	initialize register `ma`
9	(reserved)
10	initialize register `mx`
11	(reserved)
12	select address registers
13	select Dataset offset
14	initialize register masks for low half of group E registers
15	initialize register masks for high half of group E registers

4.5.2 Group A register initialization

The values of the floating point registers `a0` - `a3` are initialized using configuration quadwords 0-7 to have the following value:

$+1.\text{fraction} \times 2^{\text{exponent}}$

The fraction has full 52 bits of precision and the exponent value ranges from 0 to 31. These values are obtained from the initialization quadword (in little endian format) according to Table 4.5.2.

Table 4.5.2 - Group A register initialization

bits	description
0-51	fraction
52-58	(reserved)
59-63	exponent

4.5.3 Memory registers

Registers `ma` and `mx` are initialized using the low 32 bits of quadwords 8 and 10 in little endian format.

4.5.4 Address registers

Bits 0-3 of quadword 12 are used to select 4 address registers for program execution. Each bit chooses one register from a pair of integer registers according to Table 4.5.3.

Table 4.5.3 - Address registers

address register (bit)	value = 0	value = 1
`readReg0` (0)	`r0`	`r1`
`readReg1` (1)	`r2`	`r3`
`readReg2` (2)	`r4`	`r5`
`readReg3` (3)	`r6`	`r7`

4.5.5 Dataset offset

The `datasetOffset` is calculated as the remainder of dividing quadword 13 by `RANDOMX_DATASET_EXTRA_SIZE / 64 + 1`. The result is multiplied by `64`. This offset is used when reading values from the Dataset.

4.5.6 Group E register masks

These masks are used for the conversion of group E registers (see 4.3.2). The low and high halves each have their own masks initialized from quadwords 14 and 15. The fraction mask is given by bits 0-21 and the exponent mask by bits 60-63 of the initialization quadword.

4.6 VM execution

During VM execution, 3 additional temporary registers are used: `ic`, `spAddr0` and `spAddr1`. Program execution consists of initialization and loop execution.

4.6.1 Initialization

1. `ic` register is set to `RANDOMX_PROGRAM_ITERATIONS`.
2. `spAddr0` is set to the value of `mx`.
3. `spAddr1` is set to the value of `ma`.
4. The values of all integer registers `r0`-`r7` are set to zero.

4.6.2 Loop execution

The loop described below is repeated until the value of the `ic` register reaches zero.

1. XOR of registers `readReg0` and `readReg1` (see Table 4.5.3) is calculated and `spAddr0` is XORed with the low 32 bits of the result and `spAddr1` with the high 32 bits.
2. `spAddr0` is used to perform a 64-byte aligned read from Scratchpad level 3 (using mask from Table 4.2.1). The 64 bytes are XORed with all integer registers in order `r0`-`r7`.
3. `spAddr1` is used to perform a 64-byte aligned read from Scratchpad level 3 (using mask from Table 4.2.1). Each floating point register `f0`-`f3` and `e0`-`e3` is initialized using an 8-byte value according to the conversion rules from chapters 4.3.1 and 4.3.2.
4. The 256 instructions stored in the Program Buffer are executed.
5. The `mx` register is XORed with the low 32 bits of registers `readReg2` and `readReg3` (see Table 4.5.3).
6. A 64-byte Dataset item at address `datasetOffset + mx % RANDOMX_DATASET_BASE_SIZE` is prefetched from the Dataset (it will be used during the next iteration).

7. A 64-byte Dataset item at address `datasetOffset + ma % RANDOMX_DATASET_BASE_SIZE` is loaded from the Dataset. The 64 bytes are XORed with all integer registers in order `r0` - `r7`.
8. The values of registers `mx` and `ma` are swapped.
9. The values of all integer registers `r0` - `r7` are written to the Scratchpad (L3) at address `spAddr1` (64-byte aligned).
10. Register `f0` is XORed with register `e0` and the result is stored in register `f0`. Register `f1` is XORed with register `e1` and the result is stored in register `f1`. Register `f2` is XORed with register `e2` and the result is stored in register `f2`. Register `f3` is XORed with register `e3` and the result is stored in register `f3`.
11. The values of registers `f0` - `f3` are written to the Scratchpad (L3) at address `spAddr0` (64-byte aligned).
12. `spAddr0` and `spAddr1` are both set to zero.
13. `ic` is decreased by 1.

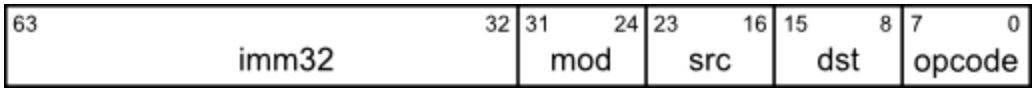
5. Instruction set

The VM executes programs in a special instruction set, which was designed in such way that any random 8-byte word is a valid instruction and any sequence of valid instructions is a valid program. Because there are no "syntax" rules, generating a random program is as easy as filling the program buffer with random data.

5.1 Instruction encoding

Each instruction word is 64 bits long. Instruction fields are encoded as shown in Fig. 5.1.

Figure 5.1 - Instruction encoding



5.1.1 opcode

There are 256 opcodes, which are distributed between 29 distinct instructions. Each instruction can be encoded using multiple opcodes (the number of opcodes specifies the frequency of the instruction in a random program).

Table 5.1.1: Instruction groups

group	# instructions	# opcodes	
integer	17	120	46.9%
floating point	9	94	36.7%
control	2	26	10.2%
store	1	16	6.2%
	29	**256**	**100%**

All instructions are described below in chapters 5.2 - 5.5.

5.1.2 dst

Destination register. Only bits 0-1 (register groups A, F, E) or 0-2 (groups R, F+E) are used to encode a register according to Table 5.1.2.

Table 5.1.2: Addressable register groups

index	R	A	F	E	F+E
0	`r0`	`a0`	`f0`	`e0`	`f0`
1	`r1`	`a1`	`f1`	`e1`	`f1`
2	`r2`	`a2`	`f2`	`e2`	`f2`
3	`r3`	`a3`	`f3`	`e3`	`f3`
4	`r4`				`e0`
5	`r5`				`e1`
6	`r6`				`e2`
7	`r7`				`e3`

5.1.3 src

The `src` flag encodes a source operand register according to Table 5.1.2 (only bits 0-1 or 0-2 are used).

Some integer instructions use a constant value as the source operand in cases when `dst` and `src` encode the same register (see Table 5.2.1).

For register-memory instructions, the source operand is used to calculate the memory address.

5.1.4 mod

The `mod` flag is encoded as:

Table 5.1.3: mod flag encoding

`mod` bits	description	range of values
0-1	`mod.mem` flag	0-3
2-3	`mod.shift` flag	0-3
4-7	`mod.cond` flag	0-15

The `mod.mem` flag selects between Scratchpad levels L1 and L2 when reading from or writing to memory except for two cases:

- it's a memory read and `dst` and `src` encode the same register
- it's a memory write `mod.cond` is 14 or 15

In these two cases, the Scratchpad level is L3 (see Table 5.1.4).

Table 5.1.4: memory access Scratchpad level

condition	Scratchpad level
`src == dst` (read)	L3
`mod.cond >= 14` (write)	L3
`mod.mem == 0`	L2
`mod.mem != 0`	L1

The address for reading/writing is calculated by applying bitwise AND operation to the address and the 8-byte aligned address mask listed in Table 4.2.1.

The `mod.cond` and `mod.shift` flags are used by some instructions (see 5.2, 5.4).

5.1.5 imm32

A 32-bit immediate value that can be used as the source operand and is used to calculate addresses for memory operations. The immediate value is sign-extended to 64 bits unless specified otherwise.

5.2 Integer instructions

For integer instructions, the destination is always an integer register (register group R). Source operand (if applicable) can be either an integer register or memory value. If `dst` and `src` refer to the same register, most instructions use `0` or `imm32` instead of the register. This is indicated in the 'src == dst' column in Table 5.2.1.

`[mem]` indicates a memory operand loaded as an 8-byte value from the address `src + imm32`.

Table 5.2.1 Integer instructions

frequency	instruction	dst	src	`src == dst ?`	operation
16/256	IADD_RS	R	R	`src = dst`	`dst = dst + (src << mod.shift) (+ imm32)`
7/256	IADD_M	R	R	`src = 0`	`dst = dst + [mem]`
16/256	ISUB_R	R	R	`src = imm32`	`dst = dst - src`
7/256	ISUB_M	R	R	`src = 0`	`dst = dst - [mem]`
16/256	IMUL_R	R	R	`src = imm32`	`dst = dst * src`
4/256	IMUL_M	R	R	`src = 0`	`dst = dst * [mem]`
4/256	IMULH_R	R	R	`src = dst`	`dst = (dst * src) >> 64`
1/256	IMULH_M	R	R	`src = 0`	`dst = (dst * [mem]) >> 64`
4/256	ISMULH_R	R	R	`src = dst`	`dst = (dst * src) >> 64` (signed)

frequency	instruction	dst	src	`src == dst ?`	operation
1/256	ISMULH_M	R	R	`src = 0`	`dst = (dst * [mem]) >> 64` (signed)
8/256	IMUL_RCP	R	-	-	`dst = 2^x / imm32 * dst`
2/256	INEG_R	R	-	-	`dst = -dst`
15/256	IXOR_R	R	R	`src = imm32`	`dst = dst ^ src`
5/256	IXOR_M	R	R	`src = 0`	`dst = dst ^ [mem]`
8/256	IROR_R	R	R	`src = imm32`	`dst = dst >>> src`
2/256	IROL_R	R	R	`src = imm32`	`dst = dst <<< src`
4/256	ISWAP_R	R	R	`src = dst`	`temp = src; src = dst; dst = temp`

5.2.1 IADD_RS

This instructions adds the values of two registers (modulo 2^{64}). The value of the second operand is shifted left by 0-3 bits (determined by the `mod.shift` flag). Additionally, if `dst` is register `r5`, the immediate value `imm32` is added to the result.

5.2.2 IADD_M

64-bit integer addition operation (performed modulo 2^{64}) with a memory source operand.

5.2.3 ISUB_R, ISUB_M

64-bit integer subtraction (performed modulo 2^{64}). ISUB_R uses register source operand, ISUB_M uses a memory source operand.

5.2.4 IMUL_R, IMUL_M

64-bit integer multiplication (performed modulo 2^{64}). IMUL_R uses a register source operand, IMUL_M uses a memory source operand.

5.2.5 IMULH_R, IMULH_M, ISMULH_R, ISMULH_M

These instructions output the high 64 bits of the whole 128-bit multiplication result. The result differs for signed and unsigned multiplication (IMULH is unsigned, ISMULH is signed). The variants with a register source operand perform a squaring operation if `dst` equals `src`.

5.2.6 IMUL_RCP

If `imm32` equals 0 or is a power of 2, IMUL_RCP is a no-op. In other cases, the instruction multiplies the destination register by a reciprocal of `imm32` (the immediate value is zero-extended and treated as unsigned). The reciprocal is calculated as $rcp = 2^x / imm32$ by choosing the largest integer x such that $rcp < 2^{64}$.

5.2.7 INEG_R

Performs two's complement negation of the destination register.

5.2.8 IXOR_R, IXOR_M

64-bit exclusive OR operation. IXOR_R uses a register source operand, IXOR_M uses a memory source operand.

5.2.9 IROR_R, IROL_R

Performs a cyclic shift (rotation) of the destination register. Source operand (shift count) is implicitly masked to 6 bits. IROR rotates bits right, IROL left.

5.2.9 ISWAP_R

This instruction swaps the values of two registers. If source and destination refer to the same register, the result is a no-op.

5.3 Floating point instructions

For floating point instructions, the destination can be a group F or group E register. Source operand is either a group A register or a memory value.

`[mem]` indicates a memory operand loaded as an 8-byte value from the address `src + imm32` and converted according to the rules in chapters 4.3.1 (group F) or 4.3.2 (group E). The lower and upper memory operands are denoted as `[mem][0]` and `[mem][1]`.

All floating point operations are rounded according to the current value of the `fprc` register (see Table 4.3.1). Due to restrictions on the values of the floating point registers, no operation results in `NaN` or a denormal number.

Table 5.3.1 Floating point instructions

frequency	instruction	dst	src	operation
4/256	FSWAP_R	F+E	-	`(dst0, dst1) = (dst1, dst0)`
16/256	FADD_R	F	A	`(dst0, dst1) = (dst0 + src0, dst1 + src1)`
5/256	FADD_M	F	R	`(dst0, dst1) = (dst0 + [mem][0], dst1 + [mem][1])`
16/256	FSUB_R	F	A	`(dst0, dst1) = (dst0 - src0, dst1 - src1)`
5/256	FSUB_M	F	R	`(dst0, dst1) = (dst0 - [mem][0], dst1 - [mem][1])`
6/256	FSCAL_R	F	-	`(dst0, dst1) = (-2^x0 * dst0, -2^x1 * dst1)`
32/256	FMUL_R	E	A	`(dst0, dst1) = (dst0 * src0, dst1 * src1)`
4/256	FDIV_M	E	R	`(dst0, dst1) = (dst0 / [mem][0], dst1 / [mem][1])`
6/256	FSQRT_R	E	-	`(dst0, dst1) = (√dst0, √dst1)`

5.3.1 FSWAP_R

Swaps the lower and upper halves of the destination register. This is the only instruction that is applicable to both F an E register groups.

5.3.2 FADD_R, FADD_M

Double precision floating point addition. FADD_R uses a group A register source operand, FADD_M uses a memory operand.

5.3.3 FSUB_R, FSUB_M

Double precision floating point subtraction. FSUB_R uses a group A register source operand, FSUB_M uses a memory operand.

5.3.4 FSCAL_R

This instruction negates the number and multiplies it by 2^x. `x` is calculated by taking the 4 least significant digits of the biased exponent and interpreting them as a binary number using the digit set `{+1, -1}` as opposed to the traditional `{0, 1}`. The possible values of `x` are all odd numbers from -15 to +15.

The mathematical operation described above is equivalent to a bitwise XOR of the binary representation with the value of `0x80F0000000000000`.

5.3.5 FMUL_R

Double precision floating point multiplication. This instruction uses only a register source operand.

5.3.6 FDIV_M

Double precision floating point division. This instruction uses only a memory source operand.

5.3.7 FSQRT_R

Double precision floating point square root of the destination register.

5.4 Control instructions

There are 2 control instructions.

Table 5.4.1 - Control instructions

frequency	instruction	dst	src	operation
1/256	CFROUND	-	R	`fprc = src >>> imm32`
25/256	CBRANCH	R	-	`dst = dst + cimm`, conditional jump

5.4.1 CFROUND

This instruction calculates a 2-bit value by rotating the source register right by `imm32` bits and taking the 2 least significant bits (the value of the source register is unaffected). The result is stored in the `fprc` register. This changes the rounding mode of all subsequent floating point instructions.

5.4.2 CBRANCH

This instruction adds an immediate value `cimm` (constructed from `imm32`, see below) to the destination register and then performs a conditional jump in the Program Buffer based on the value of the destination register. The target of the jump is the instruction following the instruction when register `dst` was last modified.

At the beginning of each program iteration, all registers are considered to be unmodified. A register is considered as modified by an instruction in the following cases:

- It is the destination register of an integer instruction except IMUL_RCP and ISWAP_R.
- It is the destination register of IMUL_RCP and `imm32` is not zero or a power of 2.
- It is the source or the destination register of ISWAP_R and the destination and source registers are distinct.
- The CBRANCH instruction is considered to modify all integer registers.

If register `dst` has not been modified yet, the jump target is the first instruction in the Program Buffer.

The CBRANCH instruction performs the following steps:

1. A constant `b` is calculated as `mod.cond + RANDOMX_JUMP_OFFSET`.
2. A constant `cimm` is constructed as sign-extended `imm32` with bit `b` set to 1 and bit `b-1` set to 0 (if `b > 0`).
3. `cimm` is added to the destination register.
4. If bits `b` to `b + RANDOMX_JUMP_BITS - 1` of the destination register are zero, the jump is executed (target is the instruction following the instruction where `dst` was last modified).

Bits in immediate and register values are numbered from 0 to 63 with 0 being the least significant bit. For example, for `b = 10` and `RANDOMX_JUMP_BITS = 8`, the bits are arranged like this:

```
cimm = SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSMMMMMMMMMMMMMMMMMMMM10MMMMMMMMM
 dst = ..............................................XXXXXXXX.........
```

`S` is a copied sign bit from `imm32`. `M` denotes bits of `imm32`. The 9th bit is set to 0 and the 10th bit is set to 1. This value will be added to `dst`.

The second line uses `X` to mark bits of `dst` that will be checked by the condition. If all these bits are 0 after adding `cimm`, the jump is executed.

The construction of the CBRANCH instruction ensures that no inifinite loops are possible in the program.

5.5 Store instruction

There is one explicit store instruction for integer values.

`[mem]` indicates the destination is an 8-byte value at the address `dst + imm32`.

Table 5.5.1 - Store instruction

frequency	instruction	dst	src	operation
16/256	ISTORE	R	R	`[mem] = src`

5.5.1 ISTORE

This instruction stores the value of the source integer register to the memory at the address calculated from the value of the destination register. The `src` and `dst` can be the same register.

6. SuperscalarHash

SuperscalarHash is a custom diffusion function that was designed to burn as much power as possible using only the CPU's integer ALUs.

The input and output of SuperscalarHash are 8 integer registers `r0` - `r7`, each 64 bits wide. The output of SuperscalarHash is used to construct the Dataset (see chapter 7.3).

6.1 Instructions

The body of SuperscalarHash is a random sequence of instructions that can run on the Virtual Machine. SuperscalarHash uses a reduced set of only integer register-register instructions listed in Table 6.1.1. `dst` refers to the destination register, `src` to the source register.

Table 6.1.1 - SuperscalarHash instructions

freq. †	instruction	Macro-ops	operation	rules
0.11	ISUB_R	`sub_rr`	`dst = dst - src`	`dst != src`
0.11	IXOR_R	`xor_rr`	`dst = dst ^ src`	`dst != src`
0.11	IADD_RS	`lea_sib`	`dst = dst + (src << mod.shift)`	`dst != src`, `dst != r5`
0.22	IMUL_R	`imul_rr`	`dst = dst * src`	`dst != src`
0.11	IROR_C	`ror_ri`	`dst = dst >>> imm32`	`imm32 % 64 != 0`
0.10	IADD_C	`add_ri`	`dst = dst + imm32`	
0.10	IXOR_C	`xor_ri`	`dst = dst ^ imm32`	
0.03	IMULH_R	`mov_rr`, `mul_r`, `mov_rr`	`dst = (dst * src) >> 64`	
0.03	ISMULH_R	`mov_rr`, `imul_r`, `mov_rr`	`dst = (dst * src) >> 64` (signed)	
0.06	IMUL_RCP	`mov_ri`, `imul_rr`	`dst = 2^x / imm32 * dst`	`imm32 != 0`, `imm32 != 2^N`

† Frequencies are approximate. Instructions are generated based on complex rules.

6.1.1 ISUB_R

See chapter 5.2.3. Source and destination are always distinct registers.

6.1.2 IXOR_R

See chapter 5.2.8. Source and destination are always distinct registers.

6.1.3 IADD_RS

See chapter 5.2.1. Source and destination are always distinct registers and register `r5` cannot be the destination.

6.1.4 IMUL_R

See chapter 5.2.4. Source and destination are always distinct registers.

6.1.5 IROR_C

The destination register is rotated right. The rotation count is given by `imm32` masked to 6 bits and cannot be 0.

6.1.6 IADD_C

A sign-extended `imm32` is added to the destination register.

6.1.7 IXOR_C

The destination register is XORed with a sign-extended `imm32`.

6.1.8 IMULH_R, ISMULH_R

See chapter 5.2.5.

6.1.9 IMUL_RCP

See chapter 5.2.6. `imm32` is never 0 or a power of 2.

6.2 The reference CPU

Unlike a standard RandomX program, a SuperscalarHash program is generated using a strict set of rules to achieve the maximum performance on a superscalar CPU. For this purpose, the generator runs a simulation of a reference CPU.

The reference CPU is loosely based on the Intel Ivy Bridge microarchitecture. It has the following properties:

- The CPU has 3 integer execution ports P0, P1 and P5 that can execute instructions in parallel. Multiplication can run only on port P1.
- Each of the Superscalar instructions listed in Table 6.1.1 consist of one or more *Macro-ops*. Each Macro-op has certain execution latency (in cycles) and size (in bytes) as shown in Table 6.2.1.
- Each of the Macro-ops listed in Table 6.2.1 consists of 0-2 *Micro-ops* that can go to a subset of the 3 execution ports. If a Macro-op consists of 2 Micro-ops, both must be

executed together.

- The CPU can decode at most 16 bytes of code per cycle and at most 4 Micro-ops per cycle.

Table 6.2.1 - Macro-ops

Macro-op	latency	size	1st Micro-op	2nd Micro-op
`sub_rr`	1	3	P015	-
`xor_rr`	1	3	P015	-
`lea_sib`	1	4	P01	-
`imul_rr`	3	4	P1	-
`ror_ri`	1	4	P05	-
`add_ri`	1	7, 8, 9	P015	-
`xor_ri`	1	7, 8, 9	P015	-
`mov_rr`	0	3	-	-
`mul_r`	4	3	P1	P5
`imul_r`	4	3	P1	P5
`mov_ri`	1	10	P015	-

- P015 - Micro-op can be executed on any port
- P01 - Micro-op can be executed on ports P0 or P1
- P05 - Micro-op can be executed on ports P0 or P5
- P1 - Micro-op can be executed only on port P1
- P5 - Micro-op can be executed only on port P5

Macro-ops `add_ri` and `xor_ri` can be optionally padded to a size of 8 or 9 bytes for code alignment purposes. `mov_rr` has 0 execution latency and doesn't use an execution port, but still occupies space during the decoding stage (see chapter 6.3.1).

6.3 CPU simulation

SuperscalarHash programs are generated to maximize the usage of all 3 execution ports of the reference CPU. The generation consists of 4 stages:

- Decoding stage
- Instruction selection
- Port assignment
- Operand assignment

Program generation is complete when one of two conditions is met:

1. An instruction is scheduled for execution on cycle that is equal to or greater than `RANDOMX_SUPERSCALAR_LATENCY`
2. The number of generated instructions reaches `3 * RANDOMX_SUPERSCALAR_LATENCY + 2`.

6.3.1 Decoding stage

The generator produces instructions in groups of 3 or 4 Macro-op slots such that the size of each group is exactly 16 bytes.

Table 6.3.1 - Decoder configurations

decoder group	configuration
0	4-8-4
1	7-3-3-3
2	3-7-3-3
3	4-9-3
4	4-4-4-4
5	3-3-10

The rules for the selection of the decoder group are following:

- If the currently processed instruction is IMULH_R or ISMULH_R, the next decode group is group 5 (the only group that starts with a 3-byte slot and has only 3 slots).
- If the total number of multiplications that have been generated is less than or equal to the current decoding cycle, the next decode group is group 4.

- If the currently processed instruction is IMUL_RCP, the next decode group is group 0 or 3 (must begin with a 4-byte slot for multiplication).
- Otherwise a random decode group is selected from groups 0-3.

6.3.2 Instruction selection

Instructions are selected based on the size of the current decode group slot - see Table 6.3.2.

Table 6.3.2 - Decoder configurations

slot size	note	instructions
3	-	ISUB_R, IXOR_R
3	last slot in the group	ISUB_R, IXOR_R, IMULH_R, ISMULH_R
4	decode group 4, not the last slot	IMUL_R
4	-	IROR_C, IADD_RS
7,8,9	-	IADD_C, IXOR_C
10	-	IMUL_RCP

6.3.3 Port assignment

Micro-ops are issued to execution ports as soon as an available port is free. The scheduling is done optimistically by checking port availability in order P5 -> P0 -> P1 to not overload port P1 (multiplication) by instructions that can go to any port. The cycle when all Micro-ops of an instruction can be executed is called the 'scheduleCycle'.

6.3.4 Operand assignment

The source operand (if needed) is selected first. is it selected from the group of registers that are available at the 'scheduleCycle' of the instruction. A register is available if the latency of its last operation has elapsed.

The destination operand is selected with more strict rules (see column 'rules' in Table 6.1.1):

- value must be ready at the required cycle
- cannot be the same as the source register unless the instruction allows it (see column 'rules' in Table 6.1.1)

    - this avoids optimizable operations such as `reg ^ reg` or `reg - reg`
    - it also increases intermixing of register values
- register cannot be multiplied twice in a row unless `allowChainedMul` is true
    - this avoids accumulation of trailing zeroes in registers due to excessive multiplication
    - `allowChainedMul` is set to true if an attempt to find source/destination registers failed (this is quite rare, but prevents a catastrophic failure of the generator)
- either the last instruction applied to the register or its source must be different than the current instruction
    - this avoids optimizable instruction sequences such as `r1 = r1 ^ r2; r1 = r1 ^ r2` (can be eliminated) or `reg = reg >>> C1; reg = reg >>> C2` (can be reduced to one rotation) or `reg = reg + C1; reg = reg + C2` (can be reduced to one addition)
- register `r5` cannot be the destination of the IADD_RS instruction (limitation of the x86 lea instruction)

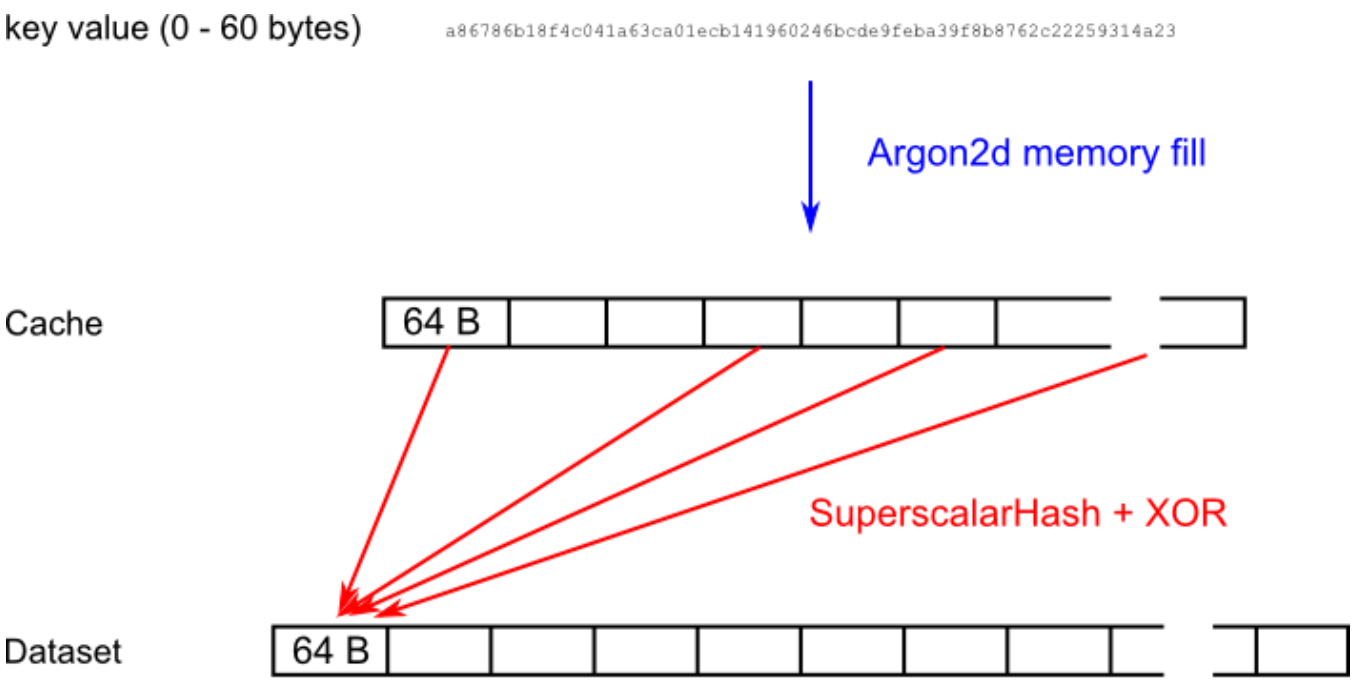
7. Dataset

The Dataset is a read-only memory structure that is used during program execution (chapter 4.6.2, steps 6 and 7). The size of the Dataset is `RANDOMX_DATASET_BASE_SIZE + RANDOMX_DATASET_EXTRA_SIZE` bytes and it's divided into 64-byte 'items'.

In order to allow PoW verification with a lower amount of memory, the Dataset is constructed in two steps using an intermediate structure called the "Cache", which can be used to calculate Dataset items on the fly.

The whole Dataset is constructed from the key value `K`, which is an input parameter of RandomX. The whole Dataset needs to be recalculated everytime the key value changes. Fig. 7.1 shows the process of Dataset construction. Note: the maximum supported length of `K` is 60 bytes. Using a longer key results in implementation-defined behavior.

Figure 7.1 - Dataset construction



7.1 Cache construction

The key `K` is expanded into the Cache using the "memory fill" function of Argon2d with parameters according to Table 7.1.1. The key is used as the "password" field.

Table 7.1.1 - Argon2 parameters

parameter	value
parallelism	`RANDOMX_ARGON_LANES`
output size	0
memory	`RANDOMX_ARGON_MEMORY`
iterations	`RANDOMX_ARGON_ITERATIONS`
version	`0x13`
hash type	0 (Argon2d)
password	key value `K`
salt	`RANDOMX_ARGON_SALT`
secret size	0

parameter	value
assoc. data size	0

The finalizer and output calculation steps of Argon2 are omitted. The output is the filled memory array.

7.2 SuperscalarHash initialization

The key value `K` is used to initialize a BlakeGenerator (see chapter 3.5), which is then used to generate 8 SuperscalarHash instances for Dataset initialization.

7.3 Dataset block generation

Dataset items are numbered sequentially with `itemNumber` starting from 0. Each 64-byte Dataset item is generated independently using 8 SuperscalarHash functions (generated according to chapter 7.2) and by XORing randomly selected data from the Cache (constructed according to chapter 7.1).

The item data is represented by 8 64-bit integer registers: `r0` - `r7`.

1. The register values are initialized as follows (`*` = multiplication, `^` = XOR):
   - `r0 = (itemNumber + 1) * 6364136223846793005`
   - `r1 = r0 ^ 9298411001130361340`
   - `r2 = r0 ^ 12065312585734608966`
   - `r3 = r0 ^ 9306329213124626780`
   - `r4 = r0 ^ 5281919268842080866`
   - `r5 = r0 ^ 10536153434571861004`
   - `r6 = r0 ^ 3398623926847679864`
   - `r7 = r0 ^ 9549104520008361294`
2. Let `cacheIndex = itemNumber`
3. Let `i = 0`
4. Load a 64-byte item from the Cache. The item index is given by `cacheIndex` modulo the total number of 64-byte items in Cache.
5. Execute `SuperscalarHash[i](r0, r1, r2, r3, r4, r5, r6, r7)`, where `SuperscalarHash[i]` refers to the i-th SuperscalarHash function. This modifies the values of the registers `r0` - `r7`.

6. XOR all registers with the 64 bytes loaded in step 4 (8 bytes per column in order `r0` - `r7`).
7. Set `cacheIndex` to the value of the register that has the longest dependency chain in the SuperscalarHash function executed in step 5.
8. Set `i = i + 1` and go back to step 4 if `i < RANDOMX_CACHE_ACCESSES`.
9. Concatenate registers `r0` - `r7` in little endian format to get the final Dataset item data.

The constants used to initialize register values in step 1 were determined as follows:

- Multiplier `6364136223846793005` was selected because it gives an excellent distribution for linear generators (D. Knuth: The Art of Computer Programming – Vol 2., also listed in Commonly used LCG parameters)
- XOR constants used to initialize registers `r1` - `r7` were determined by calculating `Hash512` of the ASCII value `"RandomX SuperScalarHash initialize"` and taking bytes 8-63 as 7 little-endian unsigned 64-bit integers. Additionally, the constant for `r1` was increased by $2^{33}+700$ and the constant for `r3` was increased by 2^{14} (these changes are necessary to ensure that all registers have unique initial values for all values of `itemNumber`).