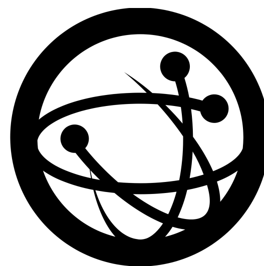
RandomX Security Audit

Final Report, 2019-07-02

FOR PUBLIC RELEASE



Contents

1 Summary **3**

1.1 Security goals 3

1.2 Methodology and reporting 4

1.3 Results summary 5

2 Introduction **6**

2.1 Principle 6

2.2 Core algorithms 7

2.3 Properties 7

2.4 Optimization approach 7

2.5 Program creation 8

2.6 Programs security goals 8

3 Observations **10**

KS-RX-O-01 AESHash1R is not a secure hash function 10

KS-RX-O-02 Unoptimized BLAKE2 10

KS-RX-O-03 Outsourceability 11

KS-RX-O-04 AES encrypt vs. decrypt 11

KS-RX-O-05 AESGenerator4R behaves like the ECB mode 11

KS-RX-O-06 "Operations that preserve entropy" 12

KS-RX-O-07 Lack of NULL pointer checks 12

KS-RX-O-08 `randomx_reciprocal()` could divide by zero 12
KS-RX-O-09 Type mismatch causing potential integer overflow 13
KS-RX-O-10 Potential int underflow in `getCodeSize()` 13
KS-RX-O-11 BLAKE2 hash errors unchecked 14
KS-RX-O-12 Potential out-of-bound write 14
KS-RX-O-13 Power-of-2 test incomplete 15

4 About 16

1 Summary

Monero is planning to use the RandomX proof-of-work (PoW) scheme, an innovative PoW attempting to maximize the advantage of miners equipped with mainstream CPUs.

Monero hired Kudelski Security to perform a security assessment of RandomX, providing access to source code and documentation, in the repository at https://github.com/tevador/RandomX/. Most of the work was performed on the version `3daceac` from the branch master (June 10, 2019).

The sections below summarize the security goals of RandomX (which define the goals of this audit, namely assessing whether these goals are satisfied), our methodology and reporting approach, and finally our results.

We would like to thank the Monero community for trusting us with this work, as well as @tevador on GitHub, the maintainer of the RandomX repository, for his rapid responses to our issues posted on GitHub. We also thank OSTIF for coordinating the work and making this audit happen.

1.1 Security goals

The general goal of this audit is to find and fix any security issues in the RandomX PoW. Of course a first step is to define what is a security issue in the context of RandomX, in other words, what are the security goals of RandomX. As a PoW, the main security goal is that there shouldn't be any algorithm to compute its results significantly more efficiently—with respect to a realistic metric–than as defined and implemented by the author. Secondary security goals include (but are not limited to):

- There shouldn't be significant speed-up of the current implementation (given the same set of target platforms).

- The attacker-controlled input (mainly, the block to validate) should not allow an attacker to DoS, crash, execute code, or make any other unexpected change to the system running the PoW.
- Miners should not be able to identify "weaker" authorized parameters (such as specific nonce values) that make the PoW computation significantly more efficient, thus giving an advantage to a miner who would know such parameters.
- There should not be any case where the PoW, under normal conditions (valid parameters and adequate platform), crashes the system, leaks memory, leaks sensitive values, etc.
- The Pow running time and resource consumption should be distributed in a way that minimizes the variance and the risk of "extreme" behaviors (in other words, the distribution should behave more like a Gaussian than a power law).
- The cryptographic components on which relies the PoW should be reliable enough and use with adequate parameters.

Note that we did not review the security of testing code, nor that of the table-based sofware AES implementation, as these won't be used for running actual PoWs.

1.2 Methodology and reporting

Our work consisted of a review of the documentation and of the source code. We also used automated static analysis tools but unsurprisingly they didn't uncover any worth-reporting issue.

Our main findings were directly reported in the issue tracker of that repository, and the maintainer of the project promptly and patiently responded to these to clarify the impact.

For completeness, all the findings reported are listed in the present report, regardless of their impact.

The audit was performed by Dr. Jean-Philippe Aumasson, VP Technology at Kudelski Security, and involved 6 person-days of work.

1.3 Results summary

We did not find any security exploitable by an attacker to significantly impact RandomX's security. We initially believed that some of our findings had a security impact, but after discussion with the RandomX designer and futher analysis, we realized that these were not realistic security concerns (for example, the cryptographic insecurity of the AES-based hash). Therefore, all our findings are listed as observations, as opposed to security issues.

Our general assessment is that, having a very narrow attack surface, and a fairly simple (thus easy to analyze) PoW mechanism, we believe that RandomX is unlikely to fail its security goals.

That said, we have less certainty about the complete match between the specification and the implementation, as well as about the optimality and correctness of the JIT logic. Given the attack model, and based on our extensive review of RandomX' code, we nonetheless believe that any such error would not be exploitable.

The least clear part, already extensively discussed online, is the relative cost-efficiency of RandomX versus that of a dedicated ASIC, when considering both the development and operation costs of the latter. This advantage is hard to quantify (and even to define rigorously) and involves a lot of technical and economical variables, We therefore don't have a specific statement to make here, except that RandomX is arguably the most hardware-unfriendly PoW considered by a major cryptocurrency.

2 Introduction

RandomX is a PoW algorithm that attempts to minimize the advantage of GPUs, FPGAs, and ASICs, by using a customer virtual machine (VM) system that executes the actual PoW and "hashing", created by picking a set of pseudo-random instructions of that VM. The VM takes as input both a description of the code to execute, and the input fed to the program running on the VM.

2.1 Principle

Such an idea was previously discussed in 2013-2014 in the context of the Password Hashing Competition (and allegedly prior to that). A conclusion was then that the potential benefits did not justify the extra complexity of such systems and the ensuing implementation/integration costs. However, that assessment was made in the context of password hashing, not of PoW for consensus protocols, for which the economic and technological context is different.

Of course a CPU is nothing but an ASIC designed for general-purpose computing operations. RandomX aims to be optimized for the class of modern CPU architectures and feature set, but one could design a CPU-like architecture specifically optimized for RandomX. This could for example include an FPGA or PLD component reconfigured on the fly to efficiently execute random programs, and a memory layout mimicking the cache architecture of common CPUs. The designers expect that FPGA/PLD reconfigure time be too slow to be CPU implementations in terms of cost-efficiency. A different, likely more efficient approach, would be to just implement the VM logic as a dedicated processor, on which would run the generated programs.

The main question is whether investments in designing and producing such a hardware-based system could be quickly amortized by the efficiency gain.

2.2 Core algorithms

RandomX relies on two software-friendly cryptographic functions: BLAKE2 (which can for example take advantage of AVX512 instructions) and mainly AES (which is arguably the most efficient building block on chips including native instructions), aimed to be efficient on CPUs thanks to NIs.

AES is clearly a good choice, thanks to the on-chip hardware acceleration. BLAKE2 is also fine, although it's not performance-critical in the RandomX design.

2.3 Properties

Like most PoWs, RandomX aims to be *asymmetric*, that is, enforce slow/costly solution generation but efficient verification.

RandomX also aims to be partially *memory-hard*, and default to 2080MiB of memory usage. This amount is understood as the memory that should be allocated when running the PoW following the standard specification. A security goal is that any method to compute the result with significantly less memory should be prohibitely less cost-effective (according to any reasonable metric) than the version using 2080MiB.

RandomX verification also requires non-negligible memory, but allows verification with lower memory while preserving the AT cost. The design document notably describes the specific case of verifying PoWs on light nodes using 256 MiB, while being 8 times slower (thought it's unclear whether this refers to the number of operations or to the actual latency when taking memory access time into account). The possibility of this trade-off is the reason why RandomX is called *partially memory-hard*.

The 256MiB limit is enforced thanks to Argon2d, whose strong memory hardness prevents any efficient computation of the result with less storage.

2.4 Optimization approach

RandomX does not target a specific microarchitecture but is designed to take advantage of CPUs low-latency caches, superscalar instructions (such as AVX512), and parallel execution units (such as Zen's double AES unit).

In particular, the "Scratchpad" is a memory buffer defined to fit into cache, and to minimize the usage of highest-latency L3 cache. The assumptions on the CPU and

cache behavior seem fair, however CPUs can sometimes behave in surprising ways.

2.5 Program creation

The random program is a sequence of 256 “random instructions” including common arithmetic operations, and in particular IEEE 754 floating-point operations (which may be a compatibility issues on certain platforms, but likely not those targetted by RandomX). Operands are determined in a way that avoids NaN results (division by zero, etc.).

By default there are 2048 iterations of the 256-instruction program, so the program latency can easily be upper bounded given the latencies of underlying instructions. The expected latency may also be easily determined given the rough distribution of instructions, under some assumptions about the branching and ratio of instructions executed.

However, programs are not straight-line, for they can include branches and in particular "random branches" (where a jump is only executed with some predefined probability). This again aims to give an edge to CPU implementations compared to GPUs or (programmable) hardware.

Programs are not sequential either, and may include flow-independent sequences of executions which permit the use of parallel ALUs or other units.

2.6 Programs security goals

Programs should be sampled in such a way that:

- There are sufficiently many distinct programs (which seems easily satisfied).
- At most a negligible fraction of programs can be significantly optimized.
- The variance of a program’s runtime should be small enough to ensure that two instances of the PoW with similar parameters will run in approximately the same time. The documentation presents empirical data on the program execution suggesting a factor between 2 and 3 between the fastest and slowest program.
- Programs cannot enter a “problematic” state, such as infinite loop, undefined operation (such as division by zero), suffer from memory corruption, etc.

- Programs are deterministic—in order to be verifiable—and in particular should avoid code patterns that are undefined or compiler-specific behavior.
- Programs should depend on the block to validate, and should not be predictable before receiving the information of the block to validate.

3 Observations

This section includes suggestions of potential improvements in terms of code quality or defensive coding, but which are, in our understanding not security issues.

KS-RX-O-01: AESHash1R is not a secure hash function

Collisions and preimages for AESHash1R can be found instantaneously, since the message is input as 16-byte blocks xored once with part of the state. (The attack is straightforward given the descriptions of AESHash1R.)

This cryptographic insecurity of AESHash1R seems well understood by the authors, and to not be an issue since this functions seems to play the role of a universal hash function rather than a cryptographic one.

Having AESHash1R to be a secure hash function would prevent the creation of final Scratchpad values that map to a given hash, but would likely make it significantly slower, which would have an impact on the relative performance of Scratchpad hashing compared to the other operations.

This issue has been reported and discussed at `https://github.com/tevador/RandomX/issues/62`, with the conclusion that the cryptographic weakness of AESHash1R is not exploitable, and apparently not required given the way it is used (as per our analysis).

KS-RX-O-02: Unoptimized BLAKE2

The implementation of BLAKE2 does not leverage vectorized instructions, and therefore may be significantly slower than an optimized implementation. For example, on platforms supporting AVX2, a reference, portable implemnentations is about 40% slower than an AVX2 implementation, as reported on a Cannonlake microarchitecture benchmark from SUPERCOP.

An AVX2 implementation of BLAKE2b can be found in the SUPERCOP archive as well as in Libsodium. An AVX512-optimized version of BLAKE2s (not BLAKE2b) is used in Wireguard Similar techniques may be used to optimize BLAKE2b for the AVX512 instruction set.

This would be an issue is BLAKE2 were performance-critical (and thus security-critical). However, unlike AESHash1R, the BLAKE2b operations are not performance-critical in the PoW construction, so an optimized BLAKE2 implementation would be little benefit to miners.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/60.

KS-RX-O-03: Outsourceability

RandomX is not designed to be nonoutsourceable, as is for example the recent Autolykos construction. RandomX therefore does not attempt to prevent mining pools by enforcing the use of a node's private key to compute the PoW.

Nonoutsourceability requires specific constructions and seems hard to add without major changes to the construction. It is anyway probably not a property required, or desired by Monero.

KS-RX-O-04: AES encrypt vs. decrypt

AES's round function is alternatively used in encryption and decryption mode over 16-byte state chunks. The reason why these two operations, as opposed to only (say) encryption is unclear; is it to force hardware miners to implement both operations' logic

KS-RX-O-05: AESGenerator4R behaves like the ECB mode

AESGenerator4R processes each 16-byte chunk of the generator's state independently, in a way that if `state0 == state2` then the transformed values will be identical as well (same for 1 and 3).

The designers may want to avoid this property by using a mode similar to CTR or CBC. If CBC is used, CBC'd decryption mode might be used instead of the encryption mode—because the former is parallelizable, whereas the former is not, which allows

to exploit Zen's two parallel AES units.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/64.

KS-RX-O-06: "Operations that preserve entropy"

The design document states that "RandomX uses all primitive integer operations that preserve entropy: addition (IADD_RS, IADD_M), subtraction (ISUB_R, ISUB_M, INEG_R), multiplication (IMUL_R, IMUL_M, IMULH_R, IMULH_M, ISMULH_R, ISMULH_M, IMUL_RCP), exclusive or (IXOR_R, IXOR_M) and rotation (IROR_R, IROL_R)."

Here "preserve entropy" is probably meant as "is invertible if one of the operands is fixed". However this is not the case for multiplication.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/60.

KS-RX-O-07: Lack of NULL pointer checks

As documented in `random.h`, most functions exposed by the API would attempted to dereference points without checking whether they're NULL. Adding checks for NULL pointers would make the code safer against developers errors.

KS-RX-O-08: `randomx_reciprocal()` could divide by zero

The `divisor` argument of `randomx_reciprocal()` is not checked in the function, and therefore a division by zero may happen:

```
uint64_t randomx_reciprocal(uint64_t divisor) {

const uint64_t p2exp63 = 1ULL << 63;

uint64_t quotient = p2exp63 / divisor, remainder = p2exp63 % divisor
...
```

We suggest to check that `divisor` isn't zero and otherwise return an appropriate value.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/63 (this function is called in such a way that division by zero cannot happen).

KS-RX-O-09: Type mismatch causing potential integer overflow

In `randomx.cpp`:

```
void randomx_init_dataset(randomx_dataset *dataset, randomx_cache *cache,
  unsigned long startItem, unsigned long itemCount) {
        cache->datasetInit(cache, dataset->memory + startItem *
  randomx::CacheLineSize, startItem, startItem + itemCount);
}
```

here `datasetInit()` will actually call `initDataset()` defined in `dataset.cpp`:

```
void initDataset(randomx_cache* cache, uint8_t* dataset, uint32_t startItem,
  uint32_t endItem) {
        for (uint32_t itemNumber = startItem; itemNumber < endItem;
  ++itemNumber, dataset += CacheLineSize)
                initDatasetItem(cache, dataset, itemNumber);
}
```

The types of `startItem` and `endItem` are 32-bit (`uint32_t`) in the former function, whereas they were initially 64-bit (`unsigned long`) on 64-bit platforms. In the unlikely case that `randomx_init_dataset()` be called with greater than 2^{32} values, the initialization would not behave as expected.

This has been reported and discussed at hhttps://github.com/tevador/RandomX/issues/70.

KS-RX-O-10: Potential int underflow in `getCodeSize()`

Although this function doesn't appear to be used, it should probably check that `codePos >= prologueSize`:

```
size_t JitCompilerX86::getCodeSize() {
        return codePos - prologueSize;
```

A related risk is that the definition of functions sizes in `jit_compiler_x86.cpp` depends on the address layout and on the relative addresses of the functions. If these assumptions were invalid, then incorrect (signed) sizes would be computed.

KS-RX-O-11: BLAKE2 hash errors unchecked

The BLAKE2b calls in the main function `randomx_calculate_hash()` are not checked for errors. In case of an error the BLAKE2 result would not be written to `tempHash` and therefore part of the computations could be bypassed:

```
1 void randomx_calculate_hash(randomx_vm *machine, const void *input, size_t inputSize,
  ↪ void *output) {
2         alignas(16) uint64_t tempHash[8];
3         blake2b(tempHash, sizeof(tempHash), input, inputSize, nullptr, 0);
4         machine->initScratchpad(&tempHash)
5         machine->resetRoundingMode();
6         for (int chain = 0; chain < RANDOMX_PROGRAM_COUNT - 1; ++chain) {
7                 machine->run(&tempHash);
8                 blake2b(tempHash, sizeof(tempHash),
  ↪ machine->getRegisterFile(), sizeof(randomx::RegisterFile), nullptr, 0);
9         }
```

We recommend to check that `blake2()` always returns zero.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/84.

KS-RX-O-12: Potential out-of-bound write

The function `fillAes4Rx4()` will write 64-byte blocks to the address space provided until the current pointer is greater or equal than the end:

```
1     while (outptr < outputEnd) {
2
3             ...
4
5             rx_store_vec_i128((rx_vec_i128*)outptr + 0, state0);
6             rx_store_vec_i128((rx_vec_i128*)outptr + 1, state1);
7             rx_store_vec_i128((rx_vec_i128*)outptr + 2, state2);
8             rx_store_vec_i128((rx_vec_i128*)outptr + 3, state3);
9
10            outptr += 64;
11    }
```

Consequently if the output length is not a multiple of 64 bytes then up to 63 bytes may be written outside the allocated space.

This has been reported and discussed at https://github.com/tevador/RandomX/issues/83.

KS-RX-O-13: Power-of-2 test incomplete

Although this function won't be called with zero as an argument (since it's called with a non-zero divisor), it's worth noting that it will return true when called with zero (which is not a power of 2):

```
inline bool isPowerOf2(uint64_t x) {
                return (x & (x - 1)) == 0;
        }
```

As discussed with the designer, this behavior is intended, and the function will be renamed `isZeroOrPowerOf2()`.

4 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com or https://kudelski-blockchain.com/.

Kudelski Security
route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland

This report and all its content is copyright (c) Nagravision SA 2019, all rights reserved.