RandomX

Security Audit

Organized by the Open Source Technology Improvement Fund

Ref.	**19-07-610-REP**
Version	**1.0**
Date	**July 30th, 2019**
Made for	**OSTIF & Monero Research Lab**
Conducted by	**Quarkslab**

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris, France

Contents

1. Project Information

Document History			
Version	**Date**	**Details**	**Authors**
1.0	30/07/2019	Final version	Laurent Grémy, Christian Heitman and Philippe Teuwen

Quarkslab		
Contact	**Position**	**E-mail address**
Frédéric Raynal	Quarkslab CEO	`fraynal@quarkslab.com`
Matthieu Duez	Service Manager	`mduez@quarkslab.com`
Laurent Grémy	R&D Engineer	`lgremy@quarkslab.com`
Christian Heitman	R&D Engineer	`cheitman@quarkslab.com`
Philippe Teuwen	R&D Engineer	`pteuwen@quarkslab.com`

Open Source Technology Improvement Fund	
Contact	**E-mail address**
Derek Zimmer	`derek@ostif.org`

Monero Research Lab	
Contact	**E-mail address**
Howard Chu	`hyc@symas.com`

2. Executive summary

2.1 Synthesis

Quarkslab has studied the security of the Monero Research Lab's new Proof-of-Work algorithm called *RandomX*. The evaluation was spread over about three weeks for a total of 32 days with three engineers. It took over from three other security audits, all four made possible thanks to the Open Source Technology Improvement Fund.

Therefore, to maximize the value of a fourth review, Quarkslab focused part of its efforts on:

- the analysis of a few areas less covered by the previous reports,
- the analysis of the previous reports, the responses of Monero Research Lab, and the subsequent changes in the code and in the specifications.

Despite a highly complex and radically new subject, the documentation and code of RandomX were of very high quality. All the attack paths we could think of had already been taken into account or at least studied in the previous audits. Then we reviewed the previous reports, the Monero Research Lab replies and their subsequent code changes. We agree with them.

Moreover, we didn't find any significant optimization of the proof-of-work algorithm, even with approximations.

We only found minor inconsistencies and formulated a few recommendations. These recommendations are mainly relevant when using alternative configurations but they are not so important with the current configuration and usage of RandomX.

We chose the terms Medium, Low and None to define their priority in the following table. We didn't include some of our recommendations in this table as they had already been mentioned in the previous reports, still we mention them in this report.

The provided tests could be enhanced by adapting them to work beyond the default RandomX configuration and strengthening the tests of the more complex components such as the JIT version of the VM.

2.2 Issues and recommendations summary

ID	Issue and recommendation	Impact
RNDX-M1 (p 10)	To prevent the use of an abnormally short *Argon2d* salt, the documentation of the configuration should clarify the allowed values of *RANDOMX_ARGON_SALT* as being in the range going from 8 to $2^{32}-1$ (inclusive) and the code should check the *Argon2d* parameters, e.g., by using a patched version of *rxa2_validate_inputs* (skipping the check on the length of the output as RandomX does not use the output).	**Medium**
RNDX-M2 (p 20)	The specifications of the *datasetOffset* computation must be adapted to reflect the use of the modulus. The risk exists for other currencies choosing customized *RANDOMX_DATASET_EXTRA_SIZE* and implementing alternative clients based on the specifications to end up with two types of clients giving different PoW hash results.	**Medium**
RNDX-M3 (p 22)	We recommend checking the size of the RandomX input key although it may require some extra modification as API functions don't return error codes. It can also be explicitly stated in the specifications what happens when a longer key is used.	**Medium**
RNDX-L1 (p 10)	Permitted values for *RANDOMX_ARGON_LANES* should be upper bounded in the configuration document to $2^{24}-1$ (inclusive) and *RANDOMX_ARGON_ITERATIONS* should be upper bounded to $2^{32}-1$ (inclusive).	**Low**
RNDX-L2 (p 10)	Permitted values for *RANDOMX_ARGON_MEMORY* should be lower bounded in the configuration document to 8 (as *ARGON2_SYNC_POINTS* $= 4$) and not 1.	**Low**
RNDX-L3 (p 20)	To avoid misunderstandings, we suggest to mention explicitly the 64-byte alignment of *mx* in the specifications in 4.6.2.5 (loop). Similarly, the alignment of *ma* should be added in 4.5.3 (initialization) or 4.6.2.7 (loop).	**Low**
RNDX-N1 (p 7)	Provided that they diversify the other existing configuration parameters, RandomX alternatives can work with the various seed strings hardcoded with identical values. Nevertheless, in our opinion, these protocol personalization strings should become part of the configuration parameters. We recommend to allow the customization of these strings and then derive the constants and tables dynamically during the compilation or startup.	**None**
RNDX-N2 (p 8)	It is also possible to add warnings or disclaimers around some of the RandomX cryptographic components (*AesGenerator1R*, *AesGenerator4R* and *AesHash1R*) in the specifications and design notes to highlight the dangers of reusing them in other contexts different from this PoW, because of their lack of general cryptographic properties.	**None**

3. Context

With the support of the Open Source Technology Improvement Fund, Monero Research Lab ordered four independent security reviews of its new Proof-of-Work algorithm *RandomX*. Quarkslab security review was conducted once the reports of the three previous reviews were available ([ToB], [Kud] and [X41]). To maximize the value of a fourth review, Quarkslab focused part of its efforts into:

- the analysis of a few areas less covered by previous reports,
- the analysis of the previous reports, the responses of Monero Research Lab, and the subsequent changes in the code and in the specifications.

3.1 Description of the request

The primary goals of such audit are to verify that:

- the implementation of the protocol is well respected,
- there are no vulnerabilities,
- criterias for a proof of work are met.

Criteria for a proof of work algorithm are :

- **Optimization-free**: there is no algorithmic speed-up that allows one to calculate the hash faster than the reference algorithm.
- **Progress-free**: proof of work calculation does not depend on the history of previous calculations.
- **Approximation-free**: it is not possible to achieve speed-up larger than the inverse rate of invalid hashes

Monero Research Lab expected a full audit (implementation review and vulnerability analysis) of the basic implementation (hashing function itself which includes VM implementation). For the components used to speed up things (JIT, large pages, etc.) and for the third-party components (Blake, AES, Argon), the audit only verifies that they are correctly handled by RandomX. Their analysis is out of scope. For "optimization free" criteria, the audit verifies that there is no other way for ASIC but to fully implement this VM and that there are no ways to make it run much faster on CPU than with current JIT code.

3.2 Methodology

The evaluation that Quarkslab undertook included the four following steps:

- Global understanding of the specifications and RandomX components (5 days).
- Checking that the RandomX specifications are cryptographically secured and do not allow algorithmic optimizations (10 days).
- Validating that the code matches the specifications + vulnerability analysis (10 days).
- Verifying that implementation is "optimization free" (7 days).

3.3 Chronology

The evaluation was spread over about three weeks for a total of 32 days with three engineers.

- July 4: kick-off meeting with Monero Research Lab.
- July 11, 12, 18: intermediate results published as GitHub issues.
- July 30: final report.

4. Proof-of-work algorithm specifications

RandomX is a PoW algorithm. To justify their design choices, the authors of this project provide pieces of documentation, the most relevant ones to describe the algorithm are:

- the specifications[1],
- the rational behind the design[2] and
- the configuration of the parameters[3].

4.1 Audited versions

The reviewed versions of the specifications were:

- the tagged version 1.0.4 of RandomX[4] and
- all the subsequent commits until commit 5d815c57c0860f39[5].

4.2 Algorithm

The algorithm is described in Section 2 of the specifications[6]. The algorithm takes as inputs:

- a string `K` and
- a string `H`.

The algorithm outputs a 256-bit result `R`. In order to highlight the loop, we rewrite the algorithm as follows.

```
1. The Dataset is initialized using the key value K.
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.

// 11. Steps 7-10 are performed a total of RANDOMX_PROGRAM_COUNT times. The
//     last iteration skips steps 9 and 10.
Repeat (RANDOMX_PROGRAM_COUNT - 1) times
(
        7. The VM is programmed using 128 + 8 * RANDOMX_PROGRAM_SIZE random
           bytes using generator gen4.
        8. The VM is executed.
        9. A new 64-byte seed is calculated as S = Hash512(RegisterFile).
       10. Set gen4.state = S (modify the state of the generator).
```

(continues on next page)

[1] https://github.com/tevador/RandomX/blob/v1.0.4/doc/specs.md
[2] https://github.com/tevador/RandomX/blob/v1.0.4/doc/design.md
[3] https://github.com/tevador/RandomX/blob/v1.0.4/doc/configuration.md
[4] https://github.com/tevador/RandomX/releases/tag/v1.0.4
[5] https://github.com/tevador/RandomX/commit/5d815c57c0860f39dbe0eb8bdf1e9dd3d2dabe3d
[6] https://github.com/tevador/RandomX/blob/v1.0.4/doc/specs.md#2-algorithm-description

(continued from previous page)

```
 )

 7. The VM is programmed using 128 + 8 * RANDOMX_PROGRAM_SIZE random bytes
    using generator gen4.
 8. The VM is executed.
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).
```

This algorithm is implemented in the *src/randomx.cpp* file, more precisely in the *randomx_calculate_hash* function. We associate the following lines of code with the different steps of the algorithm description in the comments.

```
void randomx_calculate_hash(randomx_vm *machine, const void *input, size_t
                            inputSize, void *output) {
    alignas(16) uint64_t tempHash[8];
    // Steps 1 and 2
    blake2b(tempHash, sizeof(tempHash), input, inputSize, nullptr, 0);
    // Steps 3 and 4
    machine->initScratchpad(&tempHash);
    // Step 6
    machine->resetRoundingMode();
    for (int chain = 0; chain < RANDOMX_PROGRAM_COUNT - 1; ++chain) {
      // Steps 5, 7, 8 and 10
      machine->run(&tempHash);
      // Step 9
      blake2b(tempHash, sizeof(tempHash), machine->getRegisterFile(),
              sizeof(randomx::RegisterFile), nullptr, 0);
    }
    // Steps 7, 8 and 10
    machine->run(&tempHash);
    // Steps 12, 13 and 14
    machine->getFinalResult(output, RANDOMX_HASH_SIZE);
}
```

4.3 Observations

Here are a few general observations.

The observations more specifically linked to the proof-of-work (PoW) cryptographic components or to the comparison with the code will be explained respectively in Section 5 and Section 6.

4.3.1 Usage of hardcoded strings

RandomX is using defined strings at several places in its algorithm:

- *RANDOMX_ARGON_SALT* = **"RandomX\x03"**,
- *AesGenerator1R keys* are computed from *Hash512(AesGenerator1R_seed)* with *AesGenerator1R_seed* = **"RandomX AesGenerator1R keys"** ,
- *AesGenerator4R keys 0-3* are computed from *Hash512(AesGenerator4R_seed1)* with *AesGenerator4R_seed1* = **"RandomX AesGenerator4R keys 0-3"**,
- *AesGenerator4R keys 4-7* are computed from *Hash512(AesGenerator4R_seed2)* with *Aes-*

Generator4R_seed2 = **"RandomX AesGenerator4R keys 4-7"**,

- *AesHash1R state* is computed from *Hash512(AesHash1R_seed)* with *AesHash1R_seed1* = **"RandomX AesHash1R state"**,
- *AesHash1R xkeys* are computed from *Hash256(AesHash1R_seed2)* with *AesHash1R_seed2* = **"RandomX AesHash1R xkeys"**,
- *SuperScalarHash XOR* constants are computed from a subset of *Hash512(SuperScalarHash_seed)* with *SuperScalarHash_seed* = **"RandomX SuperScalarHash initialize"**.

Currently only the first one, *RANDOMX_ARGON_SALT*, is meant to be diversified for usages in other contexts. E.g., Arweave has chosen **"RandomX-Arweave\x01"**[7] and Wownero **"RandomWOW\x01"**[8]. The other strings don't have official names, we came up with our own labeling in the list above by convenience.

Observation RNDX-N1: Provided that they diversify the other existing configuration parameters, RandomX alternatives can work with the various seed strings hardcoded with identical values. Nevertheless, in our opinion, these protocol personalization strings should become part of the configuration parameters. We recommend to allow the customization of these strings and then derive the constants and tables dynamically during the compilation or startup.

For the following reasons:

- Such defined strings in cryptographic protocols are typically present, beside their *nothing-up-my-sleeve* nature, to allow easy reuse in different contexts with a content bound to their usage, here e.g., we would have *"RandomX Arweave AesGenerator1R keys"* etc.
- These are very safe parameters to modify, contrary to some other parameters of RandomX which require careful attention before being changed. So it costs no design effort to choose other values for other usages.
- If they become part of the configuration, in order to keep the implementation flexible, the hashes would need to be computed during the compilation phase or the first run phase (as well as other derived values such as the soft-AES LUTs) which is in our opinion a good thing for peer review, rather than having directly hardcoded values in the implementation for which there is today no explanation in the code itself. Indeed, today one must look in the specifications to understand and verify by computing and comparing these hardcoded values.
- In the process, RandomX could adapt *AesHash1R xkeys* creation and usage as it was done for *AesGenerator4R*, mostly for aesthetic reasons and to avoid people (wrongly) raising concerns even if it has been demonstrated in previous reports and discussions that it's not a security issue to reuse *xkey0* and *xkey1*. Each of the other key derivations is using Hash512, so RandomX could just use Hash512 instead of Hash256 for these *xkeys* and get 4 of them.

The observations on hardcoded strings were shared in Issue #103[9].

[7] https://github.com/ArweaveTeam/RandomX/blob/arweave/src/configuration.h
[8] https://github.com/wownero/RandomWOW/blob/master/src/configuration.h
[9] https://github.com/tevador/RandomX/issues/103

4.3.2 Warnings against RandomX components reuse

In the previous audits and subsequent discussions, it has been made clear that some of the RandomX cryptographic components are safe in their specific usage despite their lack of more general cryptographic properties. See also Section 5.3 for our comments on these components. Still, some developers could be tempted to reuse these components without a proper use case analysis.

Observation RNDX-N2: It is also possible to add warnings or disclaimers around some of the RandomX cryptographic components (*AesGenerator1R*, *AesGenerator4R* and *AesHash1R*) in the specifications and design notes to highlight the dangers of reusing them in other contexts different from this PoW, because of their lack of general cryptographic properties.

5. Cryptographic primitives

In this PoW algorithm, we note that some cryptographic primitives are used. However, some of them are not chosen specifically for a cryptographic usage and are only common primitives that may serve the goal of the ASIC resistance work. These primitives are:

- *BLAKE2b* [BLA] which corresponds to *Hash512* and *Hash256* in the algorithm description;
- *Argon2d* [Arg] and
- *AES* [AES] which is used in *AesGenerator1R*, *AesGenerator4R* and *AesHash1R*.

5.1 *BLAKE2b*

The *BLAKE2b* hash function can produce hash values of different bit lengths, and especially of 256 bits and 512 bits, as mandatory by the use of *Hash256* and *Hash512*. Note that in the implementation, the *BLAKE2b* calls refer to the reference implementation[1], which is not as optimized as a less portable version compatible with modern CPU providing vectorization such as the *avx2* ones (see for example the benchmarks provided by SUPERCOP[2]). Note that this remark is already discussed in Issue #60[3], which is part of [Kud]. However, since the hash function is called only $2+$`RANDOMX_PROGRAM_COUNT`, where `RANDOMX_PROGRAM_COUNT` $= 8$ in the default configuration, this will not give a decisive advantage to a miner using optimized implementation of the hash function. *BLAKE2b* is a state-of-the-art cryptographic hash function which is optimized to be efficient on 64-bit platforms, its use for producing cryptographic or non-cryptographic hash values is fair.

Note that following the RFC [RFC], the input of *BLAKE2b* must be less than 2^{128} bytes. The length of `H`, which is of arbitrary length according to the specifications, must not exceed this value. The other uses of *BLAKE2b* are with 256-byte inputs.

5.2 *Argon2d*

The *Argon2d* hash function is a parametrized hash function among which RandomX chooses to set:

- the degree of parallelism p to `RANDOMX_ARGON_LANES` $= 1$;
- the memory size to `RANDOMX_ARGON_MEMORY` $= 262144 = 2^{18}$ KiB, which is in the range $[8p, 2^{32})$ KiB according to the specifications and
- the number of iterations to `RANDOMX_ARGON_ITERATIONS` $= 3$, which is in the range $[1, 2^{32})$ according to the specifications.

The primary inputs of such a hash function are:

- a string K of 0 to 60 bytes, which is in the range $[0, 2^{32})$ bytes, and
- a nonce/salt of 8 bytes `RANDOMX_ARGON_SALT` = `RandomX\x03` which is in the range $[8, 2^{32})$ bytes of acceptable lengths.

[1] https://github.com/BLAKE2/BLAKE2
[2] https://bench.cr.yp.to/impl-hash/blake2b.html
[3] https://github.com/tevador/RandomX/issues/60

As in *BLAKE2b*, the reference implementation of *Argon2d* is used, which is not the fastest one for modern CPU. The gain by using an efficient implementation must, however, be negligible compared to using the reference implementation. *Argon2d* is a state-of-the-art cryptographic hash function, its use for producing cryptographic or non-cryptographic hash values is fair.

Nevertheless, there is a risk of misuse of *Argon2d* for cryptocurrencies using RandomX with their own parameters.

Indeed, the configuration documentation doesn't enforce the salt size to be at least 8 bytes:

"RANDOMX_ARGON_SALT: Salt value for Cache initialization. Permitted values: Any string of byte values."

The Argon code itself has means to enforce it:

Listing 5.1: argon2.h

```
#define ARGON2_MIN_SALT_LENGTH UINT32_C(8)
#define ARGON2_MAX_SALT_LENGTH UINT32_C(0xFFFFFFFF)
```

Listing 5.2: argon2_core.c

```
// in rxa2_validate_inputs function
if (ARGON2_MIN_SALT_LENGTH > context->saltlen) {
        return ARGON2_SALT_TOO_SHORT;
}
if (ARGON2_MAX_SALT_LENGTH < context->saltlen) {
        return ARGON2_SALT_TOO_LONG;
}
```

Nevertheless, nothing prevents someone to configure RandomX to use a shorter salt and the code won't raise an alarm as *rxa2_validate_inputs* is actually never used.

Observation RNDX-M1: To prevent the use of an abnormally short *Argon2d* salt, the documentation of the configuration should clarify the allowed values of *RANDOMX_ARGON_SALT* as being in the range going from 8 to $2^{32} - 1$ (inclusive) and the code should check the *Argon2d* parameters, e.g., by using a patched version of *rxa2_validate_inputs* (skipping the check on the length of the output as RandomX does not use the output).

Hopefully the two other cryptocurrencies we are aware of are using a customized configuration with a long enough *Argon2d* salt: Arweave with *RANDOMX_ARGON_SALT = "RandomX-Arweave\x01"* and Wownero with *RANDOMX_ARGON_SALT = "RandomWOW\x01"*.

Similarly, even if these are more unlikely to be misused:

RANDOMX_ARGON_LANES and *RANDOMX_ARGON_ITERATIONS* theoretical upper bounds are not specified in the configuration documentation.

Observation RNDX-L1: Permitted values for *RANDOMX_ARGON_LANES* should be upper bounded in the configuration document to $2^{24} - 1$ (inclusive) and *RANDOMX_ARGON_ITERATIONS* should be upper bounded to $2^{32} - 1$ (inclusive).

Concerning *RANDOMX_ARGON_MEMORY*, the upper bound seems correct under the as-

sumption that *CHAR_BIT=8* and *sizeof(void*) >= 4* on all target platforms, but the minimum value is wrong.

Observation RNDX-L2: Permitted values for *RANDOMX_ARGON_MEMORY* should be lower bounded in the configuration document to 8 (as *ARGON2_SYNC_POINTS* = 4) and not 1.

Let's note that, even if *Argon2d* is not in the critical path and is not running under its fastest implementation, its usage can probably be accelerated by bumping the hardcoded value *context.threads* = 1 in case *RANDOMX_ARGON_LANES* > 1.

The observations on *Argon2d* were shared in Issue #101[4].

5.3 *AES*

The *AES* encryption/decryption scheme is never used as such with an appropriate number of rounds to perform a full encryption or decryption of a message. Only a minimal number of rounds are used, where, according to the specifications of the custom functions based on AES[5]:

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

The three custom functions build upon these *AES* rounds are described in the following.

5.3.1 *AesGenerator1R*

This first function is used to fill the Scratchpad. It was noted by a previous audit [ToB] that one round of *AES* is not sufficient to make a full diffusion step. The comment[6] of the authors of RandomX is that this first and only call to *AesGenerator1R* does not decrease sufficiently the bias of the first executed program to be modified. We agree with this argument.

The keys of the *AesGenerator1R* are defined in a nothing-up-my-sleeve way, coming from the output of *BLAKE2b* on the sentence *RandomX AesGenerator1R keys*. Each key is different from the others, so we do not see any way to decrease the time spent, even if the input states for the encryption (respectively decryption) steps are the same. Indeed, if the input states are the same, the *ShiftRows*, *SubBytes* and *MixColumns* transformations will be the same since they do not depend on the key: this is nevertheless improbable, since if we consider that *BLAKE2b* acts as a random function, having two identical states only appears with a probability of $1/2^{128} \approx 2.94 \cdot 10^{-139}$.

The use of *AesGenerator1R* is fair in this context.

[4] https://github.com/tevador/RandomX/issues/101
[5] https://github.com/tevador/RandomX/blob/master/doc/specs.md#3-custom-functions
[6] https://github.com/hyc/RandomxAudits/blob/master/Comment-TrailOfBits.md#1-single-aes-rounds-used-in-aesgenerator

5.3.2 *AesGenerator4R*

Contrary to *AesGenerator1R*, the *AesGenerator4R* better mixes the bits of the input states. Each state of this function also comes from the *BLAKE2b* hash functions, which itself uses outputs of the VM as inputs. The keys are also generated in a nothing-up-my-sleeve way. We do not see any way to bypass some operations of this function. Indeed:

- if input states are the same for the decryption (respectively encryption) stages, since the chains use different keys, it is not possible to deduce from a chain the intermediate results of another chain;
- since the decryption chains use the key in the same order than the one of encryption chains, the output of an encryption (respectively decryption) chain which is the same as the input of a decryption (respectively encryption) chain may not help to deduce the output of this last chain.

The use of *AesGenerator4R* is therefore fair in this context.

5.3.3 *AesHash1R*

The last custom function based on *AES* is *AesHash1R*, which is used only one time by the algorithm. It is acknowledged by the authors that this function cannot serve as a cryptographic hash (see comments[7] about [X41]).

Unlike in the other custom functions, the inputs are the keys coming from the previous execution of the VM, and the states are defined by the implementation. The only way we see to speed up the computation is to have, in the second stage of the functions, the same input states for the decryption (respectively encryption) stages, which cannot be assumed in a highly large proportion of the executions of this function. Generating four keys instead of two with *Hash512* instead of *Hash256* will completely break this possibility.

We think that the *AesHash1R* may be used as a non-cryptographic hash function in this context[8].

5.4 Conclusion

The cryptographic primitives used in RandomX may serve their purposes, even if the cryptographic properties of these primitives are not always their main quality in the RandomX context and they should be avoided in other contexts.

[7] https://github.com/hyc/RandomxAudits/blob/master/Comment-X41.md#43-weaknesses-in-the-cryptographic-implementat
[8] https://github.com/tevador/RandomX/issues/62

6. Code review

6.1 Audited versions

The reviewed versions of the code were:

- the tagged version 1.0.4 of RandomX[1] and
- all the subsequent commits until commit 5d815c57c0860f39[2].

Most of the code of *RandomX* is written in C++ or C, with some assembly files.

6.2 Dependencies

This project does not depend on external libraries except the standard ones. However, note that the implementations of two cryptographic hash functions are the ones provided by the upstream projects https://github.com/BLAKE2/BLAKE2 and https://github.com/p-h-c/phc-winner-argon2. These two projects are placed under the Creative Commons CC0 1.0 license[3]. This license allows to include the implementations with all the needed modifications in the *RandomX* project, which is itself placed under the 3-Clause BSD license.

6.3 Overview

The codebase review was performed manually, that is, no automated vulnerability scanner tool was employed.

We carried out two main tasks during the review:

- validating that the code matches the specifications, and
- vulnerability analysis.

Most of the findings were along the lines of those reported by previous security assessments [ToB] [Kud] [X41]. Consequently, our recommendations will be aligned to the ones provided on each of the respective reports.

On the code validation part, we found that the code follows the specifications very closely, and no major issue was identified. On the vulnerability analysis, we have to consider that the attack surface is fairly small, given the nature of the project, as it interacts very little with external sources. Therefore, only few observations were made.

6.4 Code structure

The code is well structured in terms of abstraction layers and is divided in well-defined components which eases the reviewing task and provides a quick overall understanding of the project. Code-wise the most intricate parts of the project are the JIT compiler and the SuperscalarHash program generation. The first given the inherently complex nature of such component, the second given the number of rules and restrictions involved in the generation of such a program.

[1] https://github.com/tevador/RandomX/releases/tag/v1.0.4
[2] https://github.com/tevador/RandomX/commit/5d815c57c0860f39dbe0eb8bdf1e9dd3d2dabe3d
[3] Note that the second project may also be placed under the Apache Public License 2.0 for convenience reasons.

6.4.1 JITted vs Interpreted

The VM is implemented both in a native and in an interpreted way. As the time of writing this report, only support for the `x86_64` architecture is provided. However, as it can be seen in the codebase, there are plans to extend it to other architectures such as A64. The VM code is encapsulated in two main classes *CompiledVM* and *InterpretedVM*, which are further subdivided into *CompiledLightVM* and *InterpretedLightVM*, respectively.

However, the core logic of the aforementioned VMs is divided among two components: *JitCompilerX86* and *BytecodeMachine*.

6.4.2 Cryptographic primitives

Similarly to the rest of the code, the cryptographic functions are split into their own respective components. In this case, we have the two main algorithms Blake2 and Argon2, along with the implementation of AES and the hash primitives on top of them.

6.5 Observations

Here we provide a list of issues that we do not consider to be critical but that we highly recommend to take care in future releases of RandomX.

6.5.1 Instruction semantics consistency

RandomX uses a complex instruction set, which allows both register and memory addressed operands. In addition, some instructions have complex rules on how the operands are used (and, for instance, in the case of Superscalar programs, how source and destination registers are selected as well). Therefore, implementing correctly such instruction set is a difficult and delicate task.

There are tests in the current version that check for multiple issues, and which involve instruction encoding, decoding and execution. However, these are mostly performed on the *BytecodeMachine* code.

We recommend the inclusion of more tests on this matter, especially the comparison between compiled vs interpreted instruction semantics (this becomes even more relevant considering the plans of implementing native VM support for other architectures such as A64).

This was also pointed of in previous reports such as [ToB], [Kud], and [X41].

6.5.2 Program memory considerations

The JIT compiler needs the memory allocated to be both writable and executable, although, not simultaneously. In the case an attacker manages to write to these memory regions there is a threat the attacker can execute arbitrary code.

As it can be seen below, the memory allocated for the JIT compiler is marked as *READ*, *WRITE* and *EXECUTE*.

Listing 6.1: jit_compiler_x86.cpp

```
JitCompilerX86::JitCompilerX86() {
    code = (uint8_t*)allocExecutableMemory(CodeSize);
    memcpy(code, codePrologue, prologueSize);
    memcpy(code + epilogueOffset, codeEpilogue, epilogueSize);
}
```

Listing 6.2: virtual_memory.cpp

```
void* allocExecutableMemory(std::size_t bytes) {
    void* mem;
#if defined(_WIN32) || defined(__CYGWIN__)
    mem = VirtualAlloc(nullptr, bytes, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (mem == nullptr)
        throw std::runtime_error(getErrorMessage("allocExecutableMemory - VirtualAlloc
↪"));
#else
    mem = mmap(nullptr, bytes, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS |␣
↪MAP_PRIVATE, -1, 0);
    if (mem == MAP_FAILED)
        throw std::runtime_error("allocExecutableMemory - mmap failed");
#endif
    return mem;
}
```

This was also stated in the [X41] report, and at the time of writing this document remains unchanged.

It has been argued and subsequently dismissed because of performance reasons by the developers of RandomX, however, we have to stress the security risk involving such approach.

Along these lines, and in order to mitigate any issue related to code execution, it is also advisable to implement some kind of sandbox for RandomX's programs. Therefore, restricting the system calls that a running program can make to the bare minimum.

6.5.3 Hard-coded constants

We came across some hard-coded constants in the codebase whose values were not detailed in the specifications (nor the design documentation) which may lead to unexpected behavior in case of some of the default parameters change.

The first one was pointed out in the [X41] report, and fixed in a commit subsequent to the one analyzed here. The other was also fixed, but was not reported as far as we know.

1. Hard-coded *CodeSize*

The size of the buffer where the compiled code is located is restricted to 64 Kb. At first glance, this value seems appropriate, however, there is no information about how this value was calculated. Consequently, it cannot be said for sure whether it could be overflowed or not.

Listing 6.3: jit_compiler_x86.hpp

```
class Program;
class ProgramConfiguration;
class SuperscalarProgram;
```

(continues on next page)

(continued from previous page)

```
class JitCompilerX86;
class Instruction;

typedef void(JitCompilerX86::*InstructionGeneratorX86)(Instruction&, int);

constexpr uint32_t CodeSize = 64 * 1024;

class JitCompilerX86 {
public:
```

2. Hard-coded *superScalarHashOffset*

In this case, an offset within the code buffer is declared as *superScalarHashOffet* but no information about how it was calculated is provided. This constant is used in two files: *jit_compiler_x86.cpp* and *jit_compiler_x86_static.S*.

Listing 6.4: jit_compiler_x86.cpp

```
const int32_t codeSshInitSize = codeProgramEnd - codeShhInit;

const int32_t epilogueOffset = CodeSize - epilogueSize;
constexpr int32_t superScalarHashOffset = 32768;

static const uint8_t REX_ADD_RR[] = { 0x4d, 0x03 };
```

Listing 6.5: jit_compiler_x86_static.S

```
init_block_loop:
    prefetchw byte ptr [rsi]
    mov rbx, rbp
    .byte 232 ;# 0xE8 = call
    ;# .set CALL_LOC,
    .int 32768 - (call_offset - DECL(randomx_dataset_init))
```

6.5.4 CPU utilization in SuperscalarHash Programs

The main purpose of the SuperscalarHash function is "*to burn as much power as possible using only the CPU's integer ALUs*", as stated in the specifications. To achieve its goal, a simulation of a reference CPU (loosely based on the Ivy Bridge micro-architecture) is performed in order to produce a computational-intensive program. It is difficult to verify that the program will use as many features or as intensively as was designed. Although, there are tools such as Intel vTune[4] that can shed some light on this particular issue. This may help to determine if further steps need to be taken to calibrate this stage to perform as originally intended.

6.5.5 JIT compiler

The JIT compiler is one of the most delicate parts of the codebase for various reasons. Firstly, most of the code boils down to opcodes, where a single misplaced bit can lead to issue a completely different assembly instruction. Secondly, the final program generated by the compiler also has to include all the VM logic. Finally, performance is a key aspect of this component, therefore, everything has to be finely tuned to maximize it.

[4] https://software.intel.com/en-us/vtune

Including the VM code is not an easy task, take for instance, the x86 JIT compiler (the only one available at the moment). It is achieved by carefully writing the VM in assembly language (as can be seen in *jit_compiler_x86_static.S* and the entire *asm* directory) and subsequently putting it all together, along with the JITted instructions, to make sense as a whole.

To clarify this point, we provide a quick overview of the process. A program is built by first copying into the output buffer an already-compiled prologue (consisting of saving registers according to the specific calling convention used, which at the moment can be either System V AMD64 ABI or Microsoft x64, placing arguments in specific registers and initializing some other ones). It follows the VM's loop execution[5] "header" (also already compiled, and copied into the buffer). Only at this point, the JITted instructions can be placed into the buffer. Finally, the Dataset code and the VM's loop execution "footer" is added. This is a very complex process. Any modification to this code has to be done very carefully.

The following code snippets show how the compiled version of a program is constructed, as explained in the previous paragraph.

Listing 6.6: jit_compiler_x86.cpp

```
JitCompilerX86::JitCompilerX86() {
    code = (uint8_t*)allocExecutableMemory(CodeSize);
    memcpy(code, codePrologue, prologueSize);
    memcpy(code + epilogueOffset, codeEpilogue, epilogueSize);
}
```

Listing 6.7: jit_compiler_x86.cpp

```
void JitCompilerX86::generateProgram(Program& prog, ProgramConfiguration& pcfg) {
    generateProgramPrologue(prog, pcfg);
    memcpy(code + codePos, codeReadDataset, readDatasetSize);
    codePos += readDatasetSize;
    generateProgramEpilogue(prog);
}
```

Listing 6.8: jit_compiler_x86.cpp

```
void JitCompilerX86::generateProgramPrologue(Program& prog, ProgramConfiguration&␣
↪pcfg) {
    instructionOffsets.clear();
    for (unsigned i = 0; i < 8; ++i) {
        registerUsage[i] = -1;
    }
    codePos = prologueSize;
    memcpy(code + codePos - 48, &pcfg.eMask, sizeof(pcfg.eMask));
    emit(REX_XOR_RAX_R64);
    emitByte(0xc0 + pcfg.readReg0);
    emit(REX_XOR_RAX_R64);
    emitByte(0xc0 + pcfg.readReg1);
    memcpy(code + codePos, codeLoopLoad, loopLoadSize);
    codePos += loopLoadSize;
    for (unsigned i = 0; i < prog.getSize(); ++i) {
        Instruction& instr = prog(i);
        instr.src %= RegistersCount;
        instr.dst %= RegistersCount;
```

(continues on next page)

[5] https://github.com/tevador/RandomX/blob/v1.0.4/doc/specs.md#462-loop-execution

(continued from previous page)

```
        generateCode(instr, i);
    }
    emit(REX_MOV_RR);
    emitByte(0xc0 + pcfg.readReg2);
    emit(REX_XOR_EAX);
    emitByte(0xc0 + pcfg.readReg3);
}
```

Listing 6.9: jit_compiler_x86.cpp

```
void JitCompilerX86::generateProgramEpilogue(Program& prog) {
    memcpy(code + codePos, codeLoopStore, loopStoreSize);
    codePos += loopStoreSize;
    emit(SUB_EBX);
    emit(JNZ);
    emit32(prologueSize - codePos - 4);
    emitByte(JMP);
    emit32(epilogueOffset - codePos - 4);
}
```

To further show the complexity of the JIT compiler, consider the following snippet:

Listing 6.10: jit_compiler_x86_static.S

```
init_block_loop:
    prefetchw byte ptr [rsi]
    mov rbx, rbp
    .byte 232 ;# 0xE8 = call
    ;# .set CALL_LOC,
    .int 32768 - (call_offset - DECL(randomx_dataset_init))
```

This piece of code is tailor-made in such a manner that even some call targets are built in a very specific way in order to fit in the compiled program.

It is hard to assess the correctness of the aforementioned code given its complexity and the particular way in which is used. There are tests that check the overall working of the JIT compiler. However, we consider that the project can benefit from the inclusion of tests that check each part of the JIT compiler as independently as possible.

SuperscalarHash programs present similar issues as they are compiled in a similar fashion.

6.5.6 Superscalar instruction creation

While reviewing the implementation of the Superscalar instruction creation, we found a small issue for which we could not determine whether it was intended or simply a typing mistake.

As it can be seen in the code excerpt, instructions of type *ISUB_R* are created using the type *IADD_RS* for their *opGroup* field. This value is later involved in a check within the *selectDestination* function (*superscalar.cpp#L494*). However, in this case, the chosen type (either one of them) doesn't seem to have an effect on the condition.

Listing 6.11: superscalar.cpp

```
void create(const SuperscalarInstructionInfo* info, Blake2Generator& gen) {
    info_ = info;
```

(continues on next page)

(continued from previous page)

```
    reset();
    switch (info->getType())
    {
    case SuperscalarInstructionType::ISUB_R: {
        mod_ = 0;
        imm32_ = 0;
        opGroup_ = SuperscalarInstructionType::IADD_RS;
        groupParIsSource_ = true;
    } break;
```

6.5.7 *datasetOffset* computation

While reviewing the *randomx_vm::initialize* method, we noticed a mismatch between the implementation and the specifications on how the *datasetOffset* was handled.

The issue was discussed with Monero Research Lab[6].

Concretely, the specifications state: "The *datasetOffset* is calculated by bitwise AND of quadword 13 and the value *RANDOMX_DATASET_EXTRA_SIZE / 64*. The result is multiplied by *64*." This can be written in an equivalent way with a modulus (*%*), as it is done in the code:

Listing 6.12: virtual_machine.cpp

```
addressRegisters >>= 1;
config.readReg3 = 6 + (addressRegisters & 1);
datasetOffset = (program.getEntropy(13) % (randomx::DatasetExtraItems + 1)) *␣
↪randomx::CacheLineSize;
store64(&config.eMask[0], randomx::getFloatMask(program.getEntropy(14)));
store64(&config.eMask[1], randomx::getFloatMask(program.getEntropy(15)));
```

But this is true only for specific values of *RANDOMX_DATASET_EXTRA_SIZE* writable as $(2^N - 1) \times 64$, such as the value chosen in RandomX: *33554368*, which can be written as `0x7ffff` $\times 64$ or $(2^{19} - 1) \times 64$.

After discussions with Monero Research Lab, it appeared that it was an old restriction and the code now can support any non-negative integer value divisible by 64 for *RANDOMX_DATASET_EXTRA_SIZE*.

Observation RNDX-M2: The specifications of the *datasetOffset* computation must be adapted to reflect the use of the modulus. The risk exists for other currencies choosing customized *RANDOMX_DATASET_EXTRA_SIZE* and implementing alternative clients based on the specifications to end up with two types of clients giving different PoW hash results.

To avoid any out-of-band access on the Dataset, some maskings are performed on the addresses. More specifically, the Dataset is of *DatasetSize = RANDOMX_DATASET_BASE_SIZE + RANDOMX_DATASET_EXTRA_SIZE* and 64-byte accesses are done at addresses *datasetOffset + mx % RANDOMX_DATASET_BASE_SIZE*. *datasetOffset* is 64-byte aligned between 0 and RANDOMX_DATASET_EXTRA_SIZE included and *mx* between 0 and *RANDOMX_DATASET_BASE_SIZE*-64.

In the code, the alignment of *mx* is done together with then *RAN-*

[6] https://github.com/tevador/RandomX/issues/102

DOMX_DATASET_BASE_SIZE modulus, here computed in the code by bitwise AND, which is equivalent because RANDOMX_DATASET_BASE_SIZE must be a power of 2 larger or equal to 64:

Listing 6.13: vm_interpreted.cpp

```
mem.mx ^= nreg.r[config.readReg2] ^ nreg.r[config.readReg3];
mem.mx &= CacheLineAlignMask;
datasetPrefetch(datasetOffset + mem.mx);
datasetRead(datasetOffset + mem.ma, nreg.r);
std::swap(mem.mx, mem.ma);
```

Listing 6.14: common.hpp

```
constexpr size_t CacheLineSize = RANDOMX_DATASET_ITEM_SIZE;
constexpr int ScratchpadSize = RANDOMX_SCRATCHPAD_L3;
constexpr uint32_t CacheLineAlignMask = (RANDOMX_DATASET_BASE_SIZE - 1) & ~
↪(CacheLineSize - 1);
```

But in the specifications we don't see the 64-byte alignment of *mx* explicitly mentioned in the VM execution:

- "4.5.3 Registers ma and mx are initialized using the low 32 bits of quadwords 8 and 10 in little endian format."
- "4.6.2.5. The mx register is XORed with the low 32 bits of registers readReg2 and readReg3 (see Table 4.5.3)."
- "4.6.2.6. 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)."

There is only a note in 4.1 Dataset definition reminding that "All Dataset accesses read an aligned 64-byte item."

Observation RNDX-L3: To avoid misunderstandings, we suggest to mention explicitly the 64-byte alignment of *mx* in the specifications in 4.6.2.5 (loop). Similarly, the alignment of *ma* should be added in 4.5.3 (initialization) or 4.6.2.7 (loop).

In the code, *ma* is aligned during initialization.

6.5.8 RandomX *src == dst* handling in instructions

We thought initially there was a possible path to minor optimization by prefetching Scratchpad L3 values loaded from fixed addresses when source and destination registers of integer instructions reading from memory (IADD_M, ISUB_M, IMUL_M, IMULH_M, ISMULH_M, IXOR_M) were equal and we opened an issue[7] to discuss it.

But, to summarize, this would be difficult to deal with situations where Scratchpad L3 data is overwritten, costing extra checks, while, on the other hand, such fixed address values would be soon loaded in CPU L1 and could therefore be accessed pretty quickly. And approximation strategies are not viable given the high probability of prefetched data being overwritten at some point in the hash computation.

[7] https://github.com/tevador/RandomX/issues/105

Note that the special *src == dst* handling in the memory-related integer instructions was not introduced for security reason (same source and destination register could have been used without issues), but according to Monero Research Lab it was done on purpose to add L3 reads into the main loop without affecting CPU performance.

6.5.9 Check size of Key

According to the specifications, the size of the key is limited to 60 bytes. However, it doesn't seem to be enforced in the code. We compiled the examples files, *api-example1.c* and *api-example2.cpp*, with keys bigger than 60 bytes and both programs ran successfully (at least, there was no visible sign of any problem). It is yet to determine whether the key was truncated or used as such.

Observation RNDX-M3: We recommend checking the size of the RandomX input key although it may require some extra modification as API functions don't return error codes. It can also be explicitly stated in the specifications what happens when a longer key is used.

6.5.10 Overall testing considerations

Here we would like to discuss some general considerations regarding tests (we already warned about some specific issues on previous observations).

RandomX counts with several configurable parameters. Right now testing is carried out using the default values. However, it is not clear what would happen if some of them were to change or even if it was considered when they were written at all.

We think the project can benefit from making testing on a modified set of configurations easier.

6.5.11 Code quality

Here we include some overall observations regarded as good practices, some of them already mentioned in previous reports [ToB] [Kud] [X41]. Altogether they can help avoiding issues and improve the overall code quality of future releases.

- We noticed the lack of braces in *if* statements consisting only of one line of code. This practice is considered to be error-prone and led to serious bugs in the past.
- We also noticed that there are pointer dereferences without checking if indeed these are non-NULL pointers. Adding such checks, at least in API functions, is a good programming practice that will certainly pay off as code evolves.
- Finally, using stricter compilation flags can prevent programming errors that can turn into more serious issues (as well as improving the quality of the code).

7. Conclusion

Despite a highly complex and radically new subject, RandomX documentation and code were of very high quality. All the attacks paths we could think of were already taken into consideration, or at least discussed in previous audits. We reviewed the previous reports and the Monero Research Lab replies and subsequent code changes and agree with them.

We didn't find any significant optimization of the proof-of-work algorithm, even with approximations.

We only found minor inconsistencies and recommendations, mainly with a potential impact only with alternative configurations but safe in the RandomX configuration and usage. In this regard, an effort could be done in the provided tests, preparing them to work beyond the default RandomX configuration and strengthening the testing of the more complex components such as the JIT version of the VM.

8. Bibliography

[AES] NIST, Federal Information Processing Standards Publication 197, Specification for the Advanced Encryption Standard (AES), November 2001. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[Arg] A. Biryukov, D. Dinu and D. Khovratovich, Argon2: the memory-hard function for password hashing and other applications, March 2017. https://www.cryptolux.org/images/0/0d/Argon2.pdf

[BLA] J.-P. Aumasson, S. Neves, Z. Wilcox-OHearn and C. Winnerlein, BLAKE2: simpler, smaller, fast as MD5, ACNS 2013, pp. 119–135. Springer. https://blake2.net/blake2.pdf

[Kud] J.-P. Aumasson, RandomX Security Audit, July 2019. https://github.com/hyc/RandomxAudits/blob/master/Report-Kudelski-20190702.pdf

[RFC] M.-J. Saarinen and J-P. Aumasson, The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC), IETF RFC 7693, November 2015. https://tools.ietf.org/html/rfc7693

[ToB] P. Kehrer, W. Song and E. Sultanik, RandomX Security Assesment, May 2019. https://github.com/hyc/RandomxAudits/blob/master/Report-TrailOfBits.pdf

[X41] E. Sesterhenn, G. Kopf, L. Merino, S. Bazanski and M. Vervier, RandomX Audit, July 2019. https://github.com/hyc/RandomxAudits/blob/master/Report-X41-20190705.pdf