RandomX Audit
for Monero Labs

Final Report and Management Summary

2019-07-10

PUBLIC

X41 D-SEC GmbH
Dennewartstr. 25-27
D-52068 Aachen
Amtsgericht Aachen: HRB19989

`https://x41-dsec.de/`
`info@x41-dsec.de`

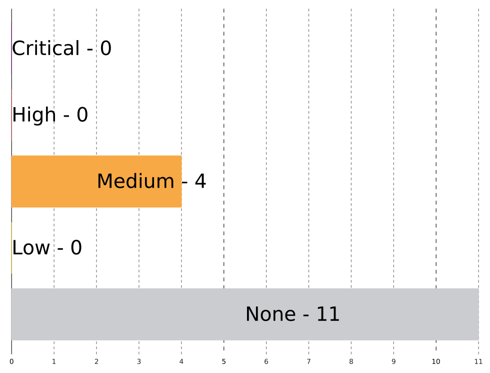
Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Editor*
1	2019-06-12	Initial Report	E. Sesterhenn
2	2019-06-27	Findings	G. Kopf, L. Merino, S. Bazanski
3	2019-06-27	Summaries	E. Sesterhenn, G. Kopf
4	2019-06-30	Finalization	M. Vervier, E. Sesterhenn, S. Bazanski
5	2019-07-05	Corrections	M. Vervier, S. Bazanski
6	2019-07-10	Add OSTIF Logo	M. Vervier

Contents

Dashboard

Target	
Customer	Monero Labs
Name	RandomX
Type	Sourcecode
Version	Commit e4b227010428571b0c4e3209d714bbcfeb943a61
Engagement	
Type	Source Code Audit and Design Review
Consultants	3: Gregor Kopf (SFS), Luis Merino (X41), Serge Bazanski
Engagement Effort	30 days, 2019-06-10 to 2019-06-28
Total issues found	4



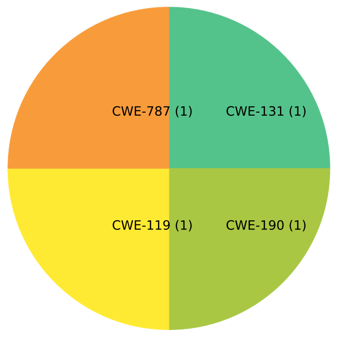


Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

X41 was tasked by the Monero project with a review of the newly-developed RandomX PoW scheme. The project was coordinated by the *Open Source Technology Improvement Fund*.

This document describes the results of the security review of RandomX.

A total of four vulnerabilities were discovered during the test by X41. None were rated as critical, none were classified as high severity, four as medium, and none as low. Additionally, eleven issues without a direct security impact were identified.

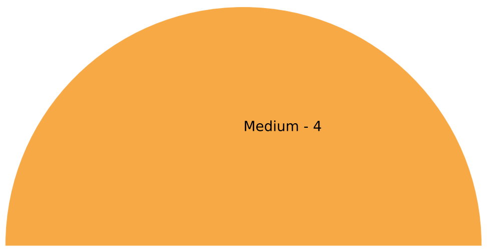


Figure 1.1: Issues and Severity

The design review was performed based on the documentation available in the RandomX GitHub repository. The code review has been performed based on the publicly available source code. All tasks have been performed in a manual fashion, without leveraging tools such as automated vulnerability scanners. In addition to the code review, limited dynamic tests have been performed - mainly in order to verify assumptions and to better understand the target implementation. The analyses were performed on commit ID e4b227010428571b0c4e3209d714bbcfeb943a61 in the RandomX GitHub repository.

The overall aim of the review was to identify potential security vulnerabilities in both, design and implementation of the new scheme. Furthermore, an analysis of the feasibility of a hardware implementation of RandomX was performed.

The review was conducted in the time frame from 2019-06-03 to 2019-06-28 by three experienced security experts.

The most severe findings include a number of out-of-bounds memory accesses in non-standard configurations. The current Monero parameters are not affected. When changing compile time configuration such as the program size, the described vulnerabilities come into effect. X41 found that it is likely possible to implement RandomX as a relatively simple, albeit large, VLIW CPU without any branch prediction or instruction scheduling logic, which yet allows for IPC rates similar to speculative, out of order and super scalar execution of a Conventional CPU.

Furthermore, a large number of side-findings have been identified, which are described in section 4.4 of this document. These do not represent actual vulnerabilities in the analyzed version of RandomX, but might become problematic in case of code re-use or future extensions of the implementation.

At the time of writing, the developers of RandomX have already developed fixes for the vulnerabilities described in this report.

In the time given, X41 was able to identify several flaws, but nothing that is considered critical.

2 Introduction

X41 reviewed the design and implementation of RandomX which is a PoW[1] algorithm, that is optimized for general-purpose CPUs. To achieve this, random code execution with memory-hard techniques are used.

Since the RandomX algorithm will be used in the cryptocoin context of Monero, any flaws and insecurities can have a financial impact. The goal of this test was to uncover such flaws before they are exploited by real adversaries.

2.1 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Hard-Coded CodeSize	MEDIUM	RNDX-PT-19-01	4.1.1
Integer Handling in Jump Target Calculation	MEDIUM	RNDX-PT-19-02	4.1.2
Integer Truncation on Dataset Allocation Size	MEDIUM	RNDX-PT-19-03	4.1.3
Incorrect Code Generated in Emulation Mode	MEDIUM	RNDX-PT-19-04	4.1.4
Low reliance on Branch Prediction Unit	NONE	RNDX-PT-19-100	4.2.1
Low reliance on Front-End Scheduling Logic (dynamic superscalar and out-of-order execution)	NONE	RNDX-PT-19-101	4.2.2
Lack of reliance on other modern CPU elements	NONE	RNDX-PT-19-102	4.2.3
Insecure AesHash1R	NONE	RNDX-PT-19-103	4.3.1
Reversible AesGenerator	NONE	RNDX-PT-19-104	4.3.2
Insufficient Diffusion in AesGenerator4R	NONE	RNDX-PT-19-105	4.3.3
Poor Code Coverage	NONE	RNDX-PT-19-106	4.4.1
JIT Memory Pages for Generated Code are Writable and Executable	NONE	RNDX-PT-19-107	4.4.2
Sandboxing RandomX Execution	NONE	RNDX-PT-19-108	4.4.3
Incorrect SuperScalarHash Latency when Program Size is too Small	NONE	RNDX-PT-19-109	4.4.4
Lack of Machine Readable Specification	NONE	RNDX-PT-19-110	4.4.5

Table 2.1: Security Relevant Findings

[1] Proof-of-Work

2.2 Scope

The audit was based on commit ID e4b227010428571b0c4e3209d714bbcfeb943a61 in the RandomX GitHub repository[2].

The design review was performed based on the documentation available in the RandomX GitHub repository. The code review has been performed based on the publicly available source code. All tasks have been performed in a manual fashion, without leveraging tools such as automated vulnerability scanners. In addition to the code review, limited dynamic tests have been performed - mainly in order to verify assumptions and to better understand the target implementation.

2.3 Recommended Further Tests

While economically viable FPGA[3] implementations of the algorithm have been ruled out, further research into practical ASIC[4] implementations of a RandomX processor is needed. To that end, one could begin the microarchitectural design of the a VLIW[5] architecture as described earlier, or look into further available optimizations and approaches. Once a microarchitecture design is achieved, more precise gate/cell counts can be established and thus a price tag can be set on the cost of implementing RandomX in VLSI[6] CMOS[7]. With this number, and the attained Fmax/MIPS of the design at a given process/node size, one can establish the minimum scale to achieve economic viability of running RandomX on ASICs.

[2] `https://github.com/tevador/RandomX`
[3] Field Programmable Gate Array
[4] Application-Specific Integrated Circuit
[5] Very Long Instruction Word
[6] Very Large Scale Integration
[7] Complementary metal-oxide-semiconductor

3 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Monero Labs are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total five different ratings exist, which are the following:

Severity Rating
None
Low
Medium
High
Critical

Findings with security impact are classified using CWE[1]. The CWE is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable X41 gives a CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information is found on the CWE-Site at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration
[2] `https://www.mitre.org`

4 Results

This chapter describes the security relevant results of the RandomX security review. Following security relevant findings in Section 4.1, the feasibility of an implementation of RandomX in dedicated hardware is discussed in section 4.2. Furthermore, weaknesses in the implementation and usage of cryptographic algorithms such as AES[1] are discussed in section 4.3. Additionally, findings without a direct security impact are documented in Section 4.4.

4.1 Findings

The following subsections describe findings with a direct impact on the security of RandomX, depending on the configuration and context it is used in.

4.1.1 RNDX-PT-19-01: Hard-Coded CodeSize

Severity:	MEDIUM
CWE:	*787 – Out-of-bounds Write*

4.1.1.1 Description

While reviewing the implementation of the RandomX JIT[2] compiler it was found that the `CodeSize` parameter is hard-coded to 64k. This is illustrated by the code excerpt below:

```
namespace randomx {

 class Program;
 class ProgramConfiguration;
```

[1] Advanced Encryption Standard
[2] Just In Time

```
class SuperscalarProgram;
class JitCompilerX86;
class Instruction;

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

constexpr uint32_t CodeSize = 64 * 1024;
}
```

Listing 4.1: Definition of CodeSize

When generating the program code, the JIT compiler however loops over all input data, which is of length ***prog.getSize()***. This is illustrated by the below code excerpt.

```
    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;
            generateCode(instr, i);
}
```

Listing 4.2: Generation of the Program

In the default configuration, this appears to have no impact, as CodeSize is large enough to hold all instructions generated by the JIT. However, if the configuration parameter *RANDOMX_PROGRAM_SIZE* is set to larger values (e.g., to 131070), the generated instructions will not fit in the hard-coded code buffer anymore, which results in an out-of-bounds access.

The below crash dump further illustrates the problem.

```
$ bin/benchmark --mine --jit --init 4
RandomX benchmark
 - full memory mode (2080 MiB)
 - JIT compiled mode
 - hardware AES mode
 - small pages mode
Initializing (4 threads) ...
Memory initialized in 43.465 s
Initializing 1 virtual machine(s) ...
Running benchmark (1000 nonces) ...
=================================================================
==2646==ERROR: AddressSanitizer: unknown-crash on address 0x7fc9acd14000 at pc 0x7fc9b00fec41 bp
 ↪  0x7fff55d8e060
WRITE of size 3 at 0x7fc9acd14000 thread T0
    #0 0x7fc9b00fec40 in __interceptor_memcpy
     ↪  /build/gcc/src/gcc/libsanitizer/sanitizer_common/sanitizer_common_
    #1 0x5625e28871cb in randomx::JitCompilerX86::emit(unsigned char const*, unsigned long)
     ↪  (/home/lm/pkg/Random
    #2 0x5625e28880be in void randomx::JitCompilerX86::emit<3ul>(unsigned char const (&) [3ul])
     ↪  (/home/lm/pkg/Ra
    #3 0x5625e287e1ce in randomx::JitCompilerX86::genAddressReg(randomx::Instruction&, bool)
     ↪  src/jit_compiler_x8
    #4 0x5625e2884a83 in randomx::JitCompilerX86::h_FSUB_M(randomx::Instruction&, int)
     ↪  src/jit_compiler_x86.cpp:
    #5 0x5625e287c7a9 in randomx::JitCompilerX86::generateCode(randomx::Instruction&, int)
     ↪  src/jit_compiler_x86.
    #6 0x5625e287baee in randomx::JitCompilerX86::generateProgramPrologue(randomx::Program&,
     ↪  randomx::ProgramCon
    #7 0x5625e287a874 in randomx::JitCompilerX86::generateProgram(randomx::Program&,
     ↪  randomx::ProgramConfigurati
    #8 0x5625e2865531 in randomx::CompiledVm<randomx::AlignedAllocator<64ul>, false>::run(void*)
     ↪  src/vm_compiled
    #9 0x5625e285f455 in randomx_calculate_hash src/randomx.cpp:242
    #10 0x5625e28447d9 in mine(randomx_vm*, std::atomic<unsigned int>&, AtomicHash&, unsigned
     ↪  int, int) src/test
    #11 0x5625e28479ad in main src/tests/benchmark.cpp:231
    #12 0x7fc9af0e6ce2 in __libc_start_main (/usr/lib/libc.so.6+0x23ce2)
    #13 0x5625e284352d in _start (/home/lm/pkg/RandomX/bin/benchmark+0x17652d)

Address 0x7fc9acd14000 is a wild pointer.
SUMMARY: AddressSanitizer: unknown-crash
 ↪  /build/gcc/src/gcc/libsanitizer/sanitizer_common/sanitizer_common_inter
```

Listing 4.3: Crash Dump

Furthermore, it should be noted that other parameters can also trigger the vulnerability. For instance, setting *RANDOMX_SUPERSCALAR_LATENCY* to $170 \cdot 64$ and *RANDOMX_SUPERSCALAR_MAX_SIZE* to $512 \cdot 64$ will also cause a crash.

The exploitability of this situation depends on the particular setting. It cannot be ruled out that in certain configurations this issue could become exploitable. One possible scenario could be that *RANDOMX_PROGRAM_SIZE* is set to a value that in most cases does not cause the JITed code to exceed the hard-coded size. Such a configuration could be undetected until an attacker specifically mines for a crashing program.

The resulting impact would at least be a DoS issue allowing an attacker to disrupt the network's mining/verification process. Code execution attacks might furthermore be possible.

4.1.1.2 Solution Advice

It is recommended to calculate the value of CodeSize depending on the size of the input to be translated or to place an upper limit on the `RANDOMX_PROGRAM_SIZE` parameter in order to prevent the above situation.

4.1.2 RNDX-PT-19-02: Integer Handling in Jump Target Calculation

Severity: MEDIUM
CWE: 190 – Integer Overflow or Wraparound

4.1.2.1 Description

While reviewing the implementation of the RandomX JIT compiler, it was found that during the calculation of jump targets, the code does not appear to consider potential corner cases, which could lead to integer-related issues. Please consider the following excerpt from the source code:

```
void JitCompilerX86::h_CBRANCH(Instruction& instr, int i) {
 int reg = instr.dst;
 int target = registerUsage[reg] + 1;
 emit(REX_ADD_I);
 emitByte(0xc0 + reg);
 int shift = instr.getModCond() + ConditionOffset;
 uint32_t imm = instr.getImm32() | (1UL << shift);
 if (ConditionOffset > 0 || shift > 0)
  imm &= ~(1UL << (shift - 1));
 emit32(imm);
 emit(REX_TEST);
 emitByte(0xc0 + reg);
 emit32(ConditionMask << shift);
 emit(JZ);
 emit32(instructionOffsets[target] - (codePos + 4));
 //mark all registers as used
 for (unsigned j = 0; j < RegistersCount; ++j) {
  registerUsage[j] = i;
 }
}
```

Listing 4.4: CBRANCH Handling

It can be observed that when emitting the jump offset, the code assumes that `instructionOffsets[target]` is smaller than `(codePos + 4)`. It should however be noted that both variables are defined as signed integers. Furthermore, the JIT compiler code does now appear to impose any limits on these values. For programs larger than 2GB, the values could therefore wrap and become negative. In particular, it might be possible that `instructionOffsets[target]` holds a positive value, while `(codePos + 4)` is negative. In such a situation, the computed jump offset could point to a location outside of the generated code, which might result in a code execution issue.

It should be noted that currently, such long programs appear to be not supported by RandomX.

However, this is not an explicit limitation introduced by proper checks - it rather stems from the fact that certain values (such as ProgramSize) are currently hard-coded.

4.1.2.2 Solution Advice

The recommended approach for preventing such integer-related issues from happening is to introduce explicit limits for the program size, which should be governed by respective checks in the code.

4.1.3 RNDX-PT-19-03: Integer Truncation on Dataset Allocation Size

Severity: MEDIUM
CWE: *131 – Incorrect Calculation of Buffer Size*

4.1.3.1 Description

DatasetSize is computed by adding *RANDOMX_DATASET_BASE_SIZE* and *RANDOMX_DATASET_EXTRA_SIZE* and storing it in a 64 bit unsigned integer variable. This value is then passed as an argument to ***allocMemory()*** to indicate the buffer size to allocate.

```
constexpr uint64_t DatasetSize = RANDOMX_DATASET_BASE_SIZE + RANDOMX_DATASET_EXTRA_SIZE;
```

Listing 4.5: Definition of DatasetSize

```
// allocator prototype
	struct LargePageAllocator {
		static void* allocMemory(size_t);
		static void freeMemory(void*, size_t);
	};

// dataset allocation
randomx_dataset *randomx_alloc_dataset(randomx_flags flags) {
	randomx_dataset *dataset = new randomx_dataset();

	try {
		if (flags & RANDOMX_FLAG_LARGE_PAGES) {
			dataset->dealloc = &randomx::deallocDataset<randomx::LargePageAllocator>;
			dataset->memory = (uint8_t*)randomx::LargePageAllocator::allocMemory(
			 ↪  randomx::DatasetSize);
		}
		else {
			dataset->dealloc = &randomx::deallocDataset<randomx::DefaultAllocator>;
			dataset->memory = (uint8_t*)randomx::DefaultAllocator::allocMemory(
			 ↪  randomx::DatasetSize);
		}
	}
```

Listing 4.6: Dataset Allocation

When compiling RandomX for 32-bits architectures, the value of *DatasetSize* is truncated from 64 bits (*uint64_t*) to 32 bits (*size_t*) while invoking ***allocMemory()***. When the value of *DatasetSize* is bigger than 4294967295, it will not fit in 32 bits and overflow to a very small value, hence leading to the allocation of a small buffer. Subsequent write and read operations during data set initialization and usage will effectively lead to invalid memory accesses, which could lead to memory corruption.

```
src/randomx.cpp:123:21: runtime error: pointer index expression with base 0xd99ff800 overflowed
 ↪  to 0x1a1ff7c0
AddressSanitizer:DEADLYSIGNAL
=================================================================
==1991==ERROR: AddressSanitizer: SEGV on unknown address 0x1a1ff7c0 (pc 0x5665b000 bp 0x00000038
 ↪  sp 0xd8fff0a0 T2)
==1991==The signal is caused by a WRITE memory access.
AddressSanitizer:DEADLYSIGNAL
    #0 0x5665afff in randomx::initDatasetItem(randomx_cache*, unsigned char*,
    unsigned long long) src/dataset.cpp:182
    #1 0x5665b19d in randomx::initDataset(randomx_cache*, unsigned char*,
    unsigned int, unsigned int) src/dataset.cpp:187
    #2 0x5663465c in randomx_init_dataset src/randomx.cpp:123
```

Listing 4.7: Crash Dump

Although the default values provided for *RANDOMX_DATASET_BASE_SIZE* and *RANDOMX_DATASET-_EXTRA_SIZE* in the RandomX implementation we reviewed are safe and dont́ lead to an overflow of *DatasetSize*, adjusting these values in the future could lead to an exploitable security flaw.

4.1.3.2 Solution Advice

X41 recommends performing sanity checks on the constants used for the data set size allocation, aborting the execution when these values overflow. Furthermore, it is recommended to use the same precision types when passing around sizes and offsets to avoid integer truncation and overflow.

4.1.4 RNDX-PT-19-04: Incorrect Code Generated in Emulation Mode

Severity: MEDIUM
CWE: 119 – Improper Restriction of Operations within the Bounds of a Memory Buffer

4.1.4.1 Description

When mining in emulation mode after increasing *RANDOMX_PROGRAM_SIZE* to 256k, the virtual machine reaches an unreachable state inside ***executeBytecode()***, which is triggered with a build instrumented with UndefinedSanitizer.

Furthermore, when using instrumentation of AddressSanitizer instead, an invalid memory access is also triggered inside ***executeBytecode()***.

```
==3075==ERROR: AddressSanitizer: SEGV on unknown address 0x558ee23ad588 (pc 0x558ee2358197 bp
↪   0x7ffe2be506b0 sp 0x7ffe2be505e0 T0)
==3075==The signal is caused by a READ memory access.
    #0 0x558ee2358196 in randomx::InterpretedVm<randomx::AlignedAllocator<64ul>, false>
    ::executeBytecode(int&, unsigned long (&) [8], double __vector(2) (&) [4],
     double __vector(2) (&) [4], double __vector(2) (&) [4]) src/vm_interpreted.cpp:82
    #1 0x558ee235a183 in randomx::InterpretedVm<randomx::AlignedAllocator<64ul>, false>
    ::executeBytecode(unsigned long (&) [8], double __vector(2) (&) [4],
    double __vector(2) (&) [4], double __vector(2) (&) [4]) src/vm_interpreted.cpp:60
    #2 0x558ee235a88c in randomx::InterpretedVm<randomx::AlignedAllocator<64ul>, false>
    ::execute() src/vm_interpreted.cpp:245
    #3 0x558ee235b184 in randomx::InterpretedVm<randomx::AlignedAllocator<64ul>, false>
    ::run(void*) src/vm_interpreted.cpp:54
    #4 0x558ee233f506 in randomx_calculate_hash src/randomx.cpp:242
    #5 0x558ee2336867 in mine(randomx_vm*, std::atomic<unsigned int>&,
     AtomicHash&, unsigned int, int) src/tests/benchmark.cpp:96
    #6 0x558ee233a68c in main src/tests/benchmark.cpp:231
    #7 0x7fdc40542ee2 in __libc_start_main (/usr/lib/libc.so.6+0x26ee2)
    #8 0x558ee23364ed in _start (/home/lm/pkg/RandomX/bin/benchmark+0xc4ed)
```

Listing 4.8: Invalid memory access on RandomX emulation

X41 has not identified the root cause of this issue.

4.1.4.2 Solution Advice

X41 advises reviewing and testing the implementation of the RandomX emulator with nondefault parameters.

4.2 Feasibility of Implementing RandomX in Hardware

The goal of RandomX is not to make ASIC implementations of the algorithm impossible - merely, to make them economically unviable. An important factor in achieving this goal is reliance on economies of scale: theoretically, if RandomX were to be only implementable on conventional, modern, high performance CPUs, then any ASIC implementation would have to not only implement such a CPU, but also do it at a scale comparable to industry leaders (like Intel or AMD). To try to break this and achieve an ASIC implementation of RandomX, we have to look closer into this assumption: does RandomX actually exercise all elements of a modern CPU[3]?

At a first glance, we see that is the case: RandomX makes heavy use of integer and floating point arithmetic to exercise FPUs and ALUs, scratchpad accesses to exercise L1 to L3 cache, and the dataset to exercise DRAM[4]. However, we have found a few weaknesses. These mostly stem from one important different between a typical instruction stream executed by RandomX vs. one executed by conventional CPUs: we know, ahead of time, that they will be executed a number of times in sequence, and that they will have a given statistical likelihood of some instruction composition.

4.2.1 RNDX-PT-19-100: Low reliance on Branch Prediction Unit

Severity:	NONE
CWE:	*None*

4.2.1.1 Description

In Conventional CPUs, a dynamic branch predictor has to keep track of branches taken so far and make predictions in advance which branch will be taken in order to prevent unnecessary pipeline flushes, before those instructions on which the branch relies get executed. In other words, the Branch Prediction Unit is responsible for the statistical and heuristic analysis of an instruction stream as it is executed, and for telling the CPU speculative execution logic which side of the branch is likely to be taken. High accuracy branch prediction is crucial not only to achieve meaningful speculative execution, but even to fill a core's pipeline before another branch instruction is to be execute. The branch predictor makes up a sizable part of the end die area of a conventional CPU.

In RandomX however, it is trivial to predict whether a branch will be taken: 99.61% [5] of RandomX ***CFBRANCH*** instructions are not taken. As such, a RandomX CPU implementation can have a static

[3] Central Processing Unit
[4] Dynamic Random-Access Memory
[5] tested by instrumenting *vm_interpreted.cpp* to count branch statistics over 10000 nonces

branch prediction of “not taken”, and have as good of an accuracy, if not better, as a conventional CPU branch predictor for RandomX instruction streams. Therefore, we have eliminated the need for a RandomX implementation to have any branch prediction logic in order to get meaningful speculative execution. Checkpoint/rollback logic still needs to be implemented, but that is a relatively small die area and complexity compared to the size of a branch predictor itself.

4.2.2 RNDX-PT-19-101: Low reliance on Front-End Scheduling Logic (dynamic superscalar and out-of-order execution)

Severity:	NONE
CWE:	*None*

4.2.2.1 Description

In a conventional CPU, out-of-order and superscalar execution provides a IPC[6] boost by finding optimized ways to map an incoming (μ)instruction stream into execution units in the core. For instance, three incoming integer operations that have no data dependencies between them can be immediately scheduled to execute in parallel on three ALUs. If the next instruction stream is a memory load, that can then be mapped onto a free memory execution unit. Then, retiring logic would coalesce the result of those parallel executions into a data state that then other incoming instructions can depend on, and write them back into a register file (which may or may not reflect ISA[7] registers). The logic required to schedule these instructions is fairly complex and also one of the main drivers in end-user visible execution speed of a CPU - especially in x86 CPUs. The x86's very strict memory model (that forces strict ordering of data effects visibility, both within a core and across cores) pushes a lot of logic into the scheduler on-die in order to keep this model in check while also reordering and doing multiple-issue as much as possible.

If we want to implement a RandomX CPU that keeps a IPC ratio high, we cannot remove the extra execution units available to the core. However, with preprocessing, we can eliminate the need for nearly all scheduling logic required in the core front-end. This is where hybrid execution comes to play: if we can preprocess the RandomX instruction stream to transform into a stream that has execution unit scheduling information encoded, we can have extreme complexity savings on the hardware side, and thus make it more economically feasible to implement such hardware. As the *perf-simulation.cpp* example shows, a simple 4/2 execution/memory unit implementation has an over four-fold increase in IPC compared to a simple in-order, non-superscalar implementation. Implementing this statically in a preprocessor would allow us to 'move' the expensive front-end scheduling logic into a program running once for every RANDOMX_PROGRAM_ITERATIONS (default: 2048) in advance.

One example of achieving this implementation would be a VLIW CPU, where every instruction in its native stream defines multiple operations to happen simultaneously on different native execution units that output on different buses/registers. These operations could be either directly RandomX ISA instructions, chunks of functionality thereof, or a fully separate instruction set (equivalent

[6] Instruction per Clock
[7] Instruction Set Architecture

to CPU μops). A preprocessor, taking on the role of a CPU frontend, would convert a RandomX instruction stream into such VLIW instructions ahead of time (see: figure 4.1).

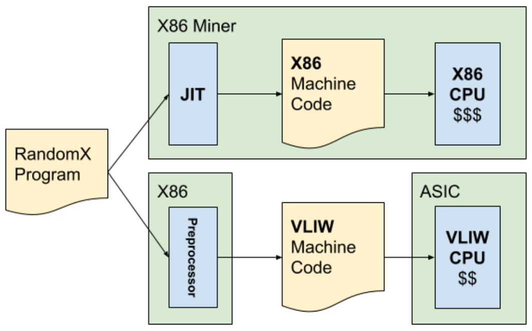


Figure 4.1: JIT vs Preprocessor

In order to demonstrate the potential of this approach, X41 wrote independent analysis software (see appendix A) to prove that inter-instruction data dependency is low enough to make this approach attractive.

Our model preprocessor ('parallelizer') has a few implementation details that make it pessimistic when it comes to possible parallelization:

- all branches are barriers that parallelization cannot happen across
- all memory writes->reads are fully dependent (e.g., any write is a barrier to all reads)

There are also some implementation details that make it optimistic when it comes to the resulting hardware:

- there is no limit on the parallel execution of integer and floating point operations
- all instructions are single-cycle - this does not have to be the case in an actual hardware implementation, but simplifies the proof of concept significantly

This allows for some very wide instructions to be generated - ones that employ the use of nearly 20 execution units in parallel without any data dependency between them!

```
$ python3 parallelizer.py

BasicBlock 00-17
 00 | c76e5e4b7b469931 |   | ISUB_M r1, r6
 01 | b3d23f340a68e963 |   | IXOR_R r1, r0
 02 | 42d469735d83124b |   | IMULH_R r2, r3
 03 | 7e021ca32db34a5c |   | IMUL_RCP r2, r3
 04 | cfe32e37bd7845b7 |   | FSUB_M f1, f1, mem[r0+3487772215]
 05 | f28aacfe9de27c7b |   | IROR_R r4, r2
 06 | 6ccf393b77c4f597 |   | FADD_R f1, f1, a0
 07 | 418a0cc5bf67b524 |   | ISUB_R r5, r7
 08 | b2ffe80058cf804a |   | IMUL_M r0, r7
 09 | 31ea35b93686c807 |   | IADD_RS r0, r6
 0a | 4646607e9f451514 |   | IADD_RS r5, r5
 0b | 88b7993a5386a397 |   | FADD_R f3, f3, a2
 0c | 448b80f33aff208a |   | FADD_R f0, f0, a3
 0d | 1e537185e3564719 |   | IADD_M r7, r6
 0e | 3c6eb30968fbbe69 |   | IXOR_R r6, r3
 0f | b4ec6f9f83ed7829 |   | ISUB_R r0, r5
 10 | 79f9ddc78e3f6500 |   | IADD_RS r5, r7
 11 | 43faad1273a67fa9 |   | FSUB_R f3, f3, a2
 12 | af31b8290df6e8a7 |   | FSUB_R f0, f0, a2
 13 | 1e195f4824154cb9 |   | FSUB_M f0, f0, mem[r5+504979272]
 14 | 3757a1986808c0db |   | FSQRT_R
 15 | d02202fb2ed43c64 |   | IXOR_R r4, r4
 16 | 7a92b0dde6af13ea |   | CBRANCH r3, 00 (7a92b0dd)

Parallelized BasicBlock 00-17
 | 00 01 02 04 07 08 09 0b 0c 0d 0e 14 15                                         |  4 FP,  9
 ↪  ALU,  4 MEM | srcs e0, f0, f3, m, m, m, m, r0, r0, r3, r3, r6, r6, r6, r7, r7, dsts a0, a2,
 ↪  a3, e0, f0, f1, f1, f3, r0, r0, r1, r1, r2, r4, r5, r6, r7
 | 03 06 0a 10 11 12                                                              |  3 FP,  3
 ↪  ALU,  0 MEM | srcs f0, f1, f3, r2, r5, r7, dsts a0, a2, a2, f0, f1, f3, r2, r5, r5
 | 05 0f 13                                                                       |  1 FP,  2
 ↪  ALU,  1 MEM | srcs m, r2, r5, r5, dsts f0, f0, r0, r4
 | 16                                                                             |  0 FP,  0
 ↪  ALU,  0 MEM |

[...]

     Program instructions: 256
Parallelized instructions: 75
                  Speedup: 3.41x
```

Listing 4.9: Parallelizer Sample

With such a low data dependency we get a preprocessable parallelization factor of on average 3.4x

(VLIW instructions vs RandomX instructions).

4.2.3 RNDX-PT-19-102: Lack of reliance on other modern CPU elements

Severity:	NONE
CWE:	*None*

4.2.3.1 Description

RandomX does not rely on any of the following features of modern CPUs, thereby additionally lowering the complexity of a possible ASIC implementation:

- privileged instructions
- interrupts and interrupt routing (x86 APIC)
- TLB[8]s,the MMU[9] and virtual memory in general
- inter-core communication and cache coherence

While these do not make up a sizable portion of a modern CPU, they are responsible for a lot of the complexity of a modern system. Eliminating them from a potential ASIC implementation vastly decreases R&D costs.

[8] Translation Lookaside Buffer
[9] Memory Management Unit

4.2.4 Conclusion

As such, it is likely possible to implement RandomX as a relatively simple, albeit large in terms of surface area, VLIW CPU without any branch prediction or instruction scheduling logic, which yet allows for IPC execution rates of RandomX rivaling that of Conventional CPU.

At 400 hashes per second per core on a mid-range conventional CPU, the execution speed to beat is 1677MIPS [10]. If we optimistically assume a 4x parallelization factor for our VLIW architecture, we get a clock speed of 420MHz for meeting this hash speed, per core. This, in addition to multiple integer and floating point execution units, a 2MiB on-die cache, and a DRAM controller for a 2GiB DIMM[11] is certainly within the available resources on a high-end FPGA (e.g., Xilinx Virtex 7), provided the design is optimized for FPGAs. However, prices of such FPGAs are $5,000+, which makes them economically unattractive compared to $100+ CPUs that can run multiple threads of RandomX. In addition, such an FPGA-based design would likely have a much lower IPS-per-watt rating than a conventional CPU.

Considering the optimizations we performed here, ASIC feasibility even in a non-high-performance processes (28nm) is not out of the question. Determining the scale required for this to be economically viable would need more careful consideration - either a gate estimation or a full implementation of the VLIW unit, DRAM controller, cache and execution units. However, implementing these is within the capability of a small team of experienced engineers, and does not require the the long experience, industry know-how and contacts of a mainstream conventional CPU manufacturer like Intel, ARM or AMD. Even small companies can successfully do commercial tape-outs of advanced CPUs[12].

4.2.4.1 Solution Advice

Solving the lack of reliance on the aforementioned CPU elements is something that has to be considered carefully. A few solutions come to mind at first:

- increase the utility of a conventional branch predictor by increasing the likelihood of a branch being taken
- increase data dependency between instructions - this lowers the parallelization factor of instructions, but also starves execution units of conventional CPUs
- generate more programs and lower their execution count to make preprocessing less attractive

[10] 400*RANDOMX_PROGRAM_SIZE*RANDOMX_PROGRAM_ITERATIONS*RANDOMX_PROGRAM_SIZE
[11] Dual Inline Memory Module
[12] `https://www.sifive.com/chip-designer#fu540`

- consider self-modifying programs, or even dynamic register selection based on execution - this limits preprocessing, but also JIT
- generate multi-core programs that share a dataset
- rely on hardware virtualization to exercise features like virtual memory

4.3 Weaknesses in the Cryptographic Implementations and Algorithms

The following observations are weaknesses in the RandomX implementation and usage of cryptographic algorithms. Due to the design of RandomX, none of them provide a direct threat to the overall PoW scheme. Nevertheless, they should be checked and potentially fixed to prevent future vulnerabilities due to yet unknown changes or use cases.

4.3.1 RNDX-PT-19-103: Insecure AesHash1R

Severity:	NONE
CWE:	*None*

4.3.1.1 Description

While reviewing the RandomX design it was found that the ***AesHash1R*** function is not a cryptographically secure hash function. It is trivial to identify collisions and (second) pre-images in this function. This is due to the fact that ***AesHash1R*** consists of only one AES round (SubBytes, ShiftRows, MixColumns and roundkey addition). ***AesHash1R*** operates block-wise.
For a given block x of input, the function can be described as

$$state_i = MixColumns(ShiftRows(SubBytes(state_{i-1}))) \oplus x.$$

Assuming $i = 1$ (i.e., only one block of input) one can directly compute the pre-image

$$x = state_1 \oplus MixColumns(ShiftRows(SubBytes(state_0))).$$

This can be trivially extended to $i > 1$ by selecting arbitrary $state_{j>0,j<i}$ and solving the resulting equations. The final steps of ***AesHash1R*** consist in encrypting $state_i$ with hard-coded round keys. This operation is directly invertible by applying the reverse transformations.

It should be noted that this property of ***AesHash1R*** does not directly impact the security of the overall RandomX scheme, as an attacker cannot derive a NONCE[13] value or other input parameters of the full algorithm using the above approach. This is due to the fact that RandomX applies a cryptographically secure hash function to the input parameters prior to applying ***AesHash1R***.

[13] Number only used once

4.3.1.2 Solution Advice

The ***AesHash1R*** function should be used with care, in particular when modifying or extending the RandomX algorithm. Directly reusing ***AesHash1R*** as a replacement for a cryptographically secure hash function is not advisable.

4.3.2 RNDX-PT-19-104: Reversible AesGenerator

Severity:	NONE
CWE:	*None*

4.3.2.1 Description

While reviewing the design of the AesGenerator functions, it was found that both functions ***AesGenerator1R*** and ***AesGenerator4R*** are trivially reversible. That is, for any desired output of these functions, the required input can be directly computed. This is due to the fact that both functions compute one or rounds of AES on their input, using hard-coded round keys. It is not advisable to use these functions as general-purpose PRNGs. In the context of RandomX, this does not directly result in an exploitable issues, as it does not allow an attacker to compute a NONCE values or another input parameter of the overall algorithm due to the fact that such parameters are subject to a cryptographic hash before they are applied to the AesGenerator functions.

4.3.2.2 Solution Advice

The AesGenerator functions should be used with care, in particular when modifying or extending the RandomX algorithm. Directly reusing such functions as a replacement for a cryptographically secure PRNG[14] is not advisable.

[14] Pseudo Random Number Generator

4.3.3 RNDX-PT-19-105: Insufficient Diffusion in AesGenerator4R

Severity:	NONE
CWE:	*None*

4.3.3.1 Description

While reviewing the design of the ***AesGenerator4R*** function, it was found that it uses the same set of AES keys for the state_0 and state_2 (and state_1 and state_3) slots. This means that in case an input of the form $XYXY$ (i.e., an input containing repeated blocks at the respective locations) is provided, the generator will produce identical output values at the respective locations. It should be noted that in the current design of RandomX this is not a likely event, as the inputs to the ***AesGenerator4R*** function are the result of a cryptographic hash function. However, it should be noted that AesGenerator4R should not be used as a general-purpose PRNG.

4.3.3.2 Solution Advice

It is recommended to make use of an individual set of keys for each of the generator state slots.

4.4 Side Findings and General Observations

The following observations do not have a direct security impact, but are related to security hardening or affect functionality and other topics that are not directly related to security.

4.4.1 RNDX-PT-19-106: Poor Code Coverage

Severity:	NONE
CWE:	*None*

4.4.1.1 Description

RandomX has very poor code coverage, apart from end-to-end tests via the benchmark example.

4.4.1.2 Solution Advice

One potentially attractive solution to test RandomX coverage is Mutation Testing[15]. This would show if there is any mutation in the code base does not yield a change in the output hash - this in turn would show that some parts of the reference implementation are not exercised during testing.

[15] `https://en.wikipedia.org/wiki/Mutation_testing`

4.4.2 RNDX-PT-19-107: JIT Memory Pages for Generated Code are Writable and Executable

Severity: NONE
CWE: *None*

4.4.2.1 Description

While reviewing the memory allocator used in the JIT compiler (see *virtual_memory.cpp* and *jit_compiler_x86.cpp*), X41 observed that the memory is allocated with the attributes ***READ***, ***WRITE*** and ***EXECUTE*** enabled.

```
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;
}
```

Listing 4.10: Allocation of Executable Memory

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

Listing 4.11: JIT Memory Allocation

Although the JIT compiler needs write permission to store the generated code and the execute permission is later required to run it, both permissions are not required to be enabled at the same time.

When considering the scenario in which an attacker can write to memory locations, write-and-execute memory pages are usually abused to reliably inject shellcode and execute arbitrary code. Having read-only access to executable pages has been an important mitigation technique to effectively reduce the impact of security flaws.

4.4.2.2 Solution Advice

X41 strongly recommends the allocation of writable and non-executable pages during the JIT code generation phase, and subsequently move page protection to execute-and-read access at the code execution phase (see ***VirtualProtect()***[16] and ***mprotect()***[17]).

[16] `https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualprotect`
[17] `http://man7.org/linux/man-pages/man2/mprotect.2.html`

4.4.3 RNDX-PT-19-108: Sandboxing RandomX Execution

Severity:	NONE
CWE:	*None*

4.4.3.1 Description

The computation of RandomX effectively requires the execution of randomized programs controlled by an input block and a NONCE. When the input blocks are obtained remotely, as it usually happens when validating blocks in a blockchain deployment, the RandomX programs executed will partially be under control of third parties.

We caní rule out the existence of certain RandomX programs that are able to bypass the security boundaries of the RandomX VM[18] when certain conditions are met. Even though we have not identified such programs, exploitable security flaws could be identified in the future. In such situation, an attacker could be able to supply certain blocks that, when hashed, exploit the security flaw.

4.4.3.2 Solution Advice

Considering the expressiveness of the RandomX virtual machine and taking into account that the computation is controlled by third parties, X41 recommends implementing sandboxing techniques to isolate the execution of the RandomX programs. Technologies like AppContainer and seccomp can effectively mitigate the impact of an attacker when trying to exploit security flaws in the sandboxed components.

[18] Virtual Machine

4.4.4 RNDX-PT-19-109: Incorrect SuperScalarHash Latency when Program Size is too Small

Severity:	NONE
CWE:	*None*

4.4.4.1 Description

While testing RandomX with non-default parameters, X41 identified a misbehavior when *RANDOMX_SUPERSCALAR_MAX_SIZE* value is too small to fit enough instructions to satisfy *RANDOMX_SUPERSCALAR_LATENCY*.

When we set *RANDOMX_SUPERSCALAR_LATENCY* to 170 and *RANDOMX_SUPERSCALAR_MAX_SIZE* to 1, it is expected that the proposed latency wont́ be achieved because of a SuperScalar program size too small. Instead of failing with an error indicating that the requested latency cant́ be achieved, SuperScalarHash and RandomX were executed without any error message.

Even though the value we used for *RANDOMX_SUPERSCALAR_MAX_SIZE* is artificially small, this situation would still apply with any other combination of parameters, and the user wouldt́ know if SuperScalarHash satisfies the requested latency during the data set generation.

4.4.4.2 Solution Advice

X41 advises to implement additional sanity checks, making sure the combination of parameters supplied to RandomX are sane and failing with a meaningful error when any of the requested parameters cant́ be satisfied.

4.4.5 RNDX-PT-19-110: Lack of Machine Readable Specification

Severity:	NONE
CWE:	*None*

4.4.5.1 Description

Currently, RandomX is defined by a human-readable markdown document of the architecture that accompanies the code base. The specification is not explicitly versioned, machine parsable, guaranteed to be up to date in comparison to the implementation or even valid at all. The reference implementation of RandomX consists of at least two separate VM execution/compilation engines, and those are only tested against a reference result in a 'benchmark' test binary.

As RandomX is a candidate for a Proof-of-Work algorithm in Monero, such a specification is indispensable. RandomX is otherwise likely run into the 'Bitcoin Core problem', where any change to the code (from bug fix to refactor) is pushed back because of the possibility of unknowingly breaking backwards compatibility - and thus causing a fork in a blockchain. In addition, such a fork might happen when an alternative, surface-level-only tested reimplementation of RandomX becomes popular, and after some point manages to be exercised to the point of showing a difference in them. The current test-against-golden-reference does not guarantee any exhaustiveness and is only a simple end-to-end integration test of the entire code base.

In addition to verification of implementations, a formal model of the VM instruction set could be used to prove certain difficult to reason about properties of the VM, such as that that there are no infinite loops or that there exist some particular access patterns to different micro-architectural components and execution units, without having to write statistical tests on the generated instructions.

4.4.5.2 Solution Advice

While formally proving the conformity of a C++ JIT engine and its output against a specification of a CPU is not necessarily trivial, there are several easy steps to take in order to have at least a machine-readable model and a CPU implementation stemming from such a model.

One entry-level approach would consist of the following:

- a declarative, high-level, machine readable specification of the RandomX ISA
- an interpreter generated from the specification, used as the source of truth for generating 'golden' test results for a given program (not seed)

- multiple golden test results (order of a few thousand) stored in machine-parsable format
- a continuously running test of all ’production’ implementations against the golden test results

5 About

The following companies and individuals participated in this audit:

5.1 X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Review of the Mozilla Firefox updater[1]
- X41 Browser Security White Paper[2]
- Review of Cryptographic Protocols (Wire)[3]
- Identification of flaws in Fax Machines[4][5]
- SmartCard Stack Fuzzing[6]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical

[1] https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/
[2] https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf
[3] https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf
[4] https://www.x41-dsec.de/lab/blog/fax/
[5] https://2018.zeronights.ru/en/reports/zero-fax-given/
[6] https://www.x41-dsec.de/lab/blog/smartcards/

background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

5.2 Serge Bazanski

Serge Bazanski is a hardware and embedded security consultant with over five years of experience.

Notable works include:

- Speaker at REcon Brussels 2018 (Hacking Toshiba Laptops)
- BootROM extraction from Tegra X1
- work on open source FPGA toolchains (yosys, nextpnr)
- reverse engineering of Siglent oscilloscopes and continued development of open bitstream[7]

5.3 Secfault Security GmbH

Secfault Security GmbH is an independent IT security consulting company, founded in 2016. Our aim is to support our customers in securing their implementations, strengthening their designs and in evaluating the security aspects of IT solutions. The company was founded by Dirk Breiden and Gregor Kopf, who worked at Recurity Labs GmbH prior to founding Secfault Security.

Secfault Security has a strong connection the IT security scene. We are in active exchange with the community and have a network of experts in different areas in IT security (from hardware analyses to compliance).

5.3.1 Focus Areas

Secfault Security offers a broad spectrum of experience and expertise. Several areas in IT security are covered, including but not limited to:

- Source Code Reviews:
  - Java, JavaScript, C, C++, Python, Perl, Ruby, Haskell, etc.

[7] `https://github.com/360nosc0pe`

  - Experience with common frameworks and technologies (such as Spring MVC, Ruby on Rails etc.)
- Analysis of embedded systems from both, a software and a hardware point of view
- Reverse Engineering:
  - All major CPU architectures (ARM, X86/64, MIPS, PPC, etc.)
- Cryptographic Tasks:
  - From protocol design to the implementation of cryptographic attacks
- Web Application Penetration Testing
- Network Penetration Testing

Secfault Security has a strong technical focus. Our goal is not only to identify vulnerabilities, but also to propose practical solutions and improvements. One of our core strengths is our ability to easily familiarize ourselves with complex systems, to dig deep into their implementation and to identify non-standard vulnerabilities and potential solution approaches.

Acronyms

AES	Advanced Encryption Standard	9
ASIC	Application-Specific Integrated Circuit	7
CMOS	Complementary metal-oxide-semiconductor	7
CPU	Central Processing Unit	18
CWE	Common Weakness Enumeration	8
DIMM	Dual Inline Memory Module	25
DRAM	Dynamic Random-Access Memory	18
FPGA	Field Programmable Gate Array	7
IPC	Instruction per Clock	20
ISA	Instruction Set Architecture	20
JIT	Just In Time	9
NONCE	Number only used once	27
PoW	Proof-of-Work	6
PRNG	Pseudo Random Number Generator	29
TLB	Translation Lookaside Buffer	24
VLIW	Very Long Instruction Word	7
VLSI	Very Large Scale Integration	7
VM	Virtual Machine	34
MMU	Memory Management Unit	24

A Parallelizer

```python
import math
import random
import sys

# from src/configuration.h
weights = {
    'RANDOMX_FREQ_IADD_RS': 25,
    'RANDOMX_FREQ_IADD_M': 7,
    'RANDOMX_FREQ_ISUB_R': 16,
    'RANDOMX_FREQ_ISUB_M': 7,
    'RANDOMX_FREQ_IMUL_R': 16,
    'RANDOMX_FREQ_IMUL_M': 4,
    'RANDOMX_FREQ_IMULH_R': 4,
    'RANDOMX_FREQ_IMULH_M': 1,
    'RANDOMX_FREQ_ISMULH_R': 4,
    'RANDOMX_FREQ_ISMULH_M': 1,
    'RANDOMX_FREQ_IMUL_RCP': 8,
    'RANDOMX_FREQ_INEG_R': 2,
    'RANDOMX_FREQ_IXOR_R': 15,
    'RANDOMX_FREQ_IXOR_M': 5,
    'RANDOMX_FREQ_IROR_R': 10,
    'RANDOMX_FREQ_IROL_R': 0,
    'RANDOMX_FREQ_ISWAP_R': 4,
    'RANDOMX_FREQ_FSWAP_R': 8,
    'RANDOMX_FREQ_FADD_R': 20,
    'RANDOMX_FREQ_FADD_M': 5,
    'RANDOMX_FREQ_FSUB_R': 20,
    'RANDOMX_FREQ_FSUB_M': 5,
    'RANDOMX_FREQ_FSCAL_R': 6,
    'RANDOMX_FREQ_FMUL_R': 20,
    'RANDOMX_FREQ_FDIV_M': 4,
    'RANDOMX_FREQ_FSQRT_R': 6,
    'RANDOMX_FREQ_CBRANCH': 16,
    'RANDOMX_FREQ_CFROUND': 1,
    'RANDOMX_FREQ_ISTORE': 16,
    'RANDOMX_FREQ_NOP': 0,
}

# from src/vm_interpreted.cpp, InterpretedVm::precompileProgram
```

```python
instructions = [
    'IADD_RS',
    'IADD_M',
    'ISUB_R',
    'ISUB_M',
    'IMUL_R',
    'IMUL_M',
    'IMULH_R',
    'IMULH_M',
    'ISMULH_R',
    'ISMULH_M',
    'IMUL_RCP',
    'INEG_R',
    'IXOR_R',
    'IXOR_M',
    'IROR_R',
    'IROL_R',
    'ISWAP_R',
    'FSWAP_R',
    'FADD_R',
    'FADD_M',
    'FSUB_R',
    'FSUB_M',
    'FSCAL_R',
    'FMUL_R',
    'FDIV_M',
    'FSQRT_R',
    'CBRANCH',
    'CFROUND',
    'ISTORE',
    'NOP',
]

assert len(weights) == len(instructions) == 30

# construct decoder matrix
decoder = {}
counter = 0
for i in instructions:
    w = weights['RANDOMX_FREQ_' + i]
    for j in range(w):
        decoder[counter] = i
        counter += 1

assert list(decoder.keys()) == list(range(256))

program_size = 256 # in instructions
instruction_size = 8 # in bytes


class Instruction:
    def __init__(self, w):
        # specs.md, 5.1 - Instruction Encoding
        self.word = w
        self.mnem = decoder[w & 0xff]
        self.dst = (w >> 8) & 0xff
        self.src = (w >> 16) & 0xff
        self.mod = (w >> 24) & 0xff
        self.imm32 = (w >> 32) & 0xffffffff

        self.target = None
        self.istarget = False

        if self.mnem.startswith('I') or self.mnem == 'CBRANCH':
            self.dst &= 0b111
            self.src &= 0b111
        elif self.mnem == 'FSWAP_R':
            self.dst &= 0b111
        elif self.mnem.startswith('F'):
            self.dst &= 0b11
            if self.mnem.endswith('_M'):
                self.src &= 0b111
            else:
                self.src &= 0b11

    def modifies_register(self):
        if not self.mnem.startswith('I'):
            return None

        intdst = 'r{}'.format(self.dst)

        if self.mnem == 'IMUL_RCP':
            # if zero
            if self.imm32 == 0:
                return None
            # if power of 2
            if (self.imm32 & (self.imm32 - 1)) == 0:
                return None
            return intdst
        if self.mnem == 'ISWAP_R':
            if self.dst == self.src:
                return None
            return intdst

        return intdst

    def deps(self):
        sources = []
        dests = []

        def s(v):
            sources.append(v)
        def d(v):
            dests.append(v)
        def rr(r):
            return 'r{}'.format(r)
        def rf(r):
            return 'f{}'.format(r)
        def re(r):
            return 'e{}'.format(r)
        def ra(r):
            return 'a{}'.format(r)
        def rfe(r):
            if r < 4:
                return 'f{}'.format(r)
            else:
                return 'e{}'.format(r-4)

        m = self.mnem

        if m == 'IADD_RS':
            d(rr(self.dst))
            if self.src == self.dst:
                s(rr(self.dst))
            else:
                s(rr(self.src))
        elif m in ('IADD_M', 'ISUB_M', 'IMUL_M', 'IMULH_M', 'ISMULH_M', 'IXOR_M'):
            d(rr(self.dst))
            s('m')
            if self.src != self.dst:
                s(rr(self.src))
        elif m in ('ISUB_R', 'IMUL_R', 'IXOR_R', 'IROL_R', 'IROR_R'):
            d(rr(self.dst))
            if self.src != self.dst:
                s(rr(self.src))
        elif m in ('IMULH_R', 'ISMULH_R', 'ISWAP_R'):
            d(rr(self.dst))
            if self.src == self.dst:
                s(rr(self.dst))
            else:
                s(rr(self.src))
        elif m in ('IMUL_RCP', 'INEG_R'):
            d(rr(self.dst))
            s(rr(self.dst))
        elif m == 'FSWAP_R':
            d(rfe(self.dst))
            s(rfe(self.dst))
        elif m.startswith('F') and m.endswith('_R'):
            d(ra(self.src))
            if m in ('FMUL_R', 'FSQRT_R'):
                s(re(self.dst))
                d(re(self.dst))
            else:
                s(rf(self.dst))
                d(rf(self.dst))
        elif m.startswith('F') and m.endswith('_M'):
```

```python
            s('m')
            s(rr(self.src))
            if m == 'FDIV_M':
                s(re(self.dst))
                d(re(self.dst))
            else:
                d(rf(self.dst))
                d(rf(self.dst))
        elif m == 'ISTORE':
            s(rr(self.src))
            d('m')
        else:
            raise Exception("Unhandled instruction {}".format(m))

        return sources, dests


    def __repr__(self):
        if self.mnem == 'CBRANCH' and self.target is not None:
            return 'CBRANCH r{}, {:02x} ({:x})'.format(self.dst, self.target, self.imm32)

        if self.mnem.startswith('I'):
            return '{} r{}, r{}'.format(self.mnem, self.dst, self.src)
        if self.mnem == 'FSWAP_R':
            if self.dst < 4:
                r = 'f{}'.format(self.dst)
            else:
                r = 'e{}'.format(self.dst-4)
            return '{} {}, {}'.format(self.mnem, r, r)
        if self.mnem in ('FADD_R', 'FSUB_R'):
            return '{} f{}, f{}, a{}'.format(self.mnem, self.dst, self.dst, self.src)
        if self.mnem in ('FADD_M', 'FSUB_M'):
            return '{} f{}, f{}, mem[r{}+{}]'.format(self.mnem, self.dst, self.dst, self.src,
             ↪  self.imm32)
        if self.mnem == 'FSCAL_R':
            return '{} f{}, f{}'.format(self.mnem, self.dst, self.dst)
        if self.mnem == 'FMUL_R':
            return '{} e{}, e{}, a{}'.format(self.mnem, self.dst, self.dst, self.src)
        if self.mnem == 'FDIV_M':
            return '{} e{}, e{}, mem[r{}+{}]'.format(self.mnem, self.dst, self.dst, self.src,
             ↪  self.imm32)
        return self.mnem


class Macro:
    def __init__(self, program):
        self.insns = []
        self.program = program

    def render(self):
        sys.stdout.write(' | ')
        flops = 0
        iops = 0
        mops = 0
        srcs = []
        dsts = []
        for addr in self.insns:
            sys.stdout.write('{:02x} '.format(addr))
            insn = self.program[addr]

            if not insn.mnem.startswith('C'):
                s, d = insn.deps()
                srcs += s
                dsts += d

            if insn.mnem.startswith('I'):
                iops += 1
            if insn.mnem.startswith('F'):
                flops += 1
            if insn.mnem.endswith('_M'):
                mops += 1

        sys.stdout.write(' ' * (80 - len(self.insns)*3))
        sys.stdout.write('| {:2d} FP, {:2d} ALU, {:2d} MEM '.format(flops, iops, mops))

        if srcs != [] and dsts != []:
            srcs = sorted(srcs)
            dsts = sorted(dsts)
            sys.stdout.write('| srcs {}, dsts {}'.format(', '.join(srcs), ', '.join(dsts)))
        else:
            sys.stdout.write('|')
        sys.stdout.write('\n')
        sys.stdout.flush()

class BasicBlock:
    def __init__(self, program, addr):
        self.program = program
        self.start = addr
        self.end = addr
        self.insns = []

    def consume(self):
        addr = self.start
        while True:
            if addr >= len(self.program):
                self.end = addr
                return addr

            ins = self.program[addr]
            if ins.mnem in ('CBRANCH', 'CFROUND') or ins.istarget:
                # cbranch/cfround is part of end basic block - ie, block points at itself
                self.end = addr + 1
                return self.end

```

```python
            addr += 1

    def render(self):
        print('BasicBlock {:02x}-{:02x}'.format(self.start, self.end))
        for i in range(self.start, self.end):
            target = 'T' if self.program[i].istarget else ' '
            print(' {:02x} | {:016x} | {} | {}'.format(i, self.program[i].word, target,
             ↪   self.program[i]))
        print()

    def threads(self):
        """Returns threads of execution that can be ran in parallel."""
        print("Parallelized BasicBlock {:02x}-{:02x}".format(self.start, self.end))

        valid = ['m', ] + \
            ['r{}'.format(r) for r in range(8)] + \
            ['f{}'.format(r) for r in range(4)] + \
            ['e{}'.format(r) for r in range(4)] + \
            ['a{}'.format(r) for r in range(4)]

        macros = [Macro(self.program)]
        used = {}

        for addr in range(self.start, self.end-1):
            sources, dests = self.program[addr].deps()
            for s in sources + dests:
                if s not in valid:
                    raise Exception("Invalid dependency {}".format(s))

            s_newest_macro = -1
            for s in sources:
                if s not in used:
                    continue
                if used[s] > s_newest_macro:
                    s_newest_macro = used[s]

            if s_newest_macro+1 == len(macros):
                macros.append(Macro(self.program))

            macros[s_newest_macro+1].insns.append(addr)

            for d in dests:
                used[d] = s_newest_macro+1

        if self.end-self.start > 1:
            macros.append(Macro(self.program))
        macros[-1].insns.append(self.end-1)

        for m in macros:
            m.render()

        print()

        return macros

class Tracer:
    def __init__(self, program):
        self.program = program
        self.reg = {}
        # mark lack of last use by None
        for r in range(8):
            self.reg['r{}'.format(r)] = None
        for a in range(4):
            self.reg['a{}'.format(a)] = None
        for f in range(4):
            self.reg['f{}'.format(a)] = None
        for e in range(4):
            self.reg['e{}'.format(a)] = None

        self._calculate_branch_targets()

    def _calculate_branch_targets(self):
        regUsed = {}
        for addr, p in enumerate(self.program):
            if p.mnem == 'CBRANCH':
                used = regUsed.get('r{}'.format(p.dst), -1)
                p.target = used + 1
                if used >= 0:
                    self.program[used].istarget = True

                # mark all registers as used
                for r in range(8):
                    regUsed['r{}'.format(r)] = addr
                continue
            else:
                modifies = p.modifies_register()
                if modifies is None:
                    continue
                regUsed[modifies] = addr

    def basic_blocks(self):
        bbs = [BasicBlock(self.program, 0)]
        while True:
            end = bbs[-1].consume()
            if end >= len(self.program):
                return bbs
            bbs.append(BasicBlock(self.program, end))
        return bbs

def run():
    # build random program (not crypto random, but this should be good enough for analysis)
    program_bytes = [random.choice(range(256)) for _ in range(program_size * instruction_size)]

    # convert to program words
    program_words = []
    for i in range(program_size):
        b = program_bytes[i*instruction_size:(i+1)*instruction_size]
        w = 0
        for bb in b:
            w = (w << 8) | bb
        program_words.append(w)

    # convert to program instructions
    program = []
    for addr, w in enumerate(program_words):
        program.append(Instruction(w))

    t = Tracer(program)
    bb = t.basic_blocks()

    macros = []
    for bbb in bb:
        bbb.render()
        macros += bbb.threads()

    print('Parallelized program:')
    for m in macros:
        m.render()

    print('     Program instructions: {}'.format(len(program)))
    print('Parallelized instructions: {}'.format(len(macros)))
    print('                  Speedup: {:.2f}x'.format(len(program)/float(len(macros))))

run()
```

Listing A.1: Parallelizer