

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# VROOM: Accelerating (Almost All) Number-Theoretic Cryptography Using Vectorization and the Residue Number System

Simon Langowski, Kaiwen He, and Srinivas Devadas, *MIT*

https://www.usenix.org/conference/usenixsecurity26/presentation/langowski

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology




# USENIX Security '26 Artifact Appendix: VROOM: Accelerating (Almost All) Number-Theoretic Cryptography Using Vectorization and the Residue Number System

Simon Langowski
MIT

Kaiwen He
MIT

Srinivas Devadas
MIT

## A Artifact Appendix

### A.1 Abstract

Modular arithmetic with a large prime modulus is a dominant computational cost in number-theoretic cryptography. Modular operations are especially challenging to parallelize efficiently on CPUs using vector instructions; standard CPU implementations rely on costly carry operations and permutation instructions to align with the multiplication datapath, negating the benefits of vectorization.

We develop vectorized algorithms for modular addition and multiplication, and present a new, constant-time modular multiplication algorithm suitable for general moduli – prime or otherwise. Our method uses a Residue Number System (RNS) representation to align the arithmetic naturally with wide vector units, and strategically eliminate extraneous instructions. Existing works either require the use of customized hardware or fail to show latency improvements.

Reducing the latency of modular arithmetic results in speedups for cryptographic applications. We accelerate RSA-4096 signatures by 4.0× (verify) and 1.3× (sign) over OpenSSL, and speed up BLS signature verifications by 4.05× over the assembly-optimized `blst` library. Results on mapping our algorithm to Nvidia GPUs demonstrate speedups on modular multiplication over Nvidia's CGBN library.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None beyond ordinary compilation and benchmark execution. The artifact performs local cryptographic arithmetic on synthetic random inputs; it does not access network services, personal data, or privileged system configuration. The code is **research-grade and not audited** for production use—do not deploy it as a security product. No destructive steps or disabled security mechanisms are required.

#### A.2.2 How to access

**Stable reference** (Zenodo concept URL): https://doi.org/10.5281/zenodo.20187973

The archive contains two compressed git repositories, covering complementary parts of the evaluation. There is no dependency between the two repos: they can be evaluated on independent machines if needed.

**Prerequisite: Install git.** On Amazon Linux 2023 (the tested OS), run:

```
sudo dnf install git -y
```

**(A) Number-theoretic CPU artifact (Sections 6.1–6.3):** modular multiplication, modular exponentiation, and RSA signing/verification, implemented as a fork of BoringSSL with VROOM integrated.

```
cd ~
git clone \
  https://github.com/kevin-he-01/vroomssl.git
cd vroomssl
git checkout 5a1dc6c765a           # AE commit
# Optional: clone OpenSSL baseline
git submodule update --init
```

Step-by-step instructions and the full paper-claim mapping are in `README.md`; the helper scripts `install_deps.sh` (dependencies) and `reproduce.sh` (all of Sections 6.1–6.3) live at the repository root.

**(B) Elliptic-curve / pairing (CPU) and GPU artifact (Section 6.4 onward):**

```
cd ~
git clone \
  https://github.com/SimonLangowski/VROOM.git
cd VROOM
git checkout f12e58daa
```

Step-by-step instructions: `ARTIFACT.md` (CPU) and `ARTIFACT_GPU.md` (GPU). This appendix maps paper claims to commands; the per-repository guides contain full detail, pinned tool versions, and sample output.

### A.2.3 Hardware dependencies

**CPU artifacts (A and B) — same host requirements:**

- Linux x86_64 host with **AVX-512 IFMA** (`avx512ifma` in `/proc/cpuinfo`).
- **Paper reference CPU:** AWS **c7i.metal-24xl**, Intel Xeon Platinum **8488C** (Sapphire Rapids).
- ≥8 GiB RAM; We recommend at least 30 GB of empty disk space **per repo** to be evaluated (60 GB if evaluating both repo A and repo B on the same machine). [1]
- We can provide **c7i-metal-24xl** upon request for short periods of time.

**GPU artifact; separate machine from CPU:**

- NVIDIA GPU with compute capability ≥7.0.
- **Paper reference GPU:** GeForce **RTX 3090** (sm_86).
- Speedups should be visible with any recent NVIDIA GPU. We are not providing ssh GPU access at this time.

### A.2.4 Software dependencies

**CPU path (repo A — number-theoretic, Sections 6.1–6.3):**

- OS: Tested on **Amazon Linux 2023**[2]; other Linux distributions work for builds with the right dependencies.
- Compiler: **clang 21.1.8** (the official LLVM 21.1.8 prebuilt release, installed under `$HOME/LLVM-21.1.8-Linux-X64` by `./install_deps.sh`). Paper timings depend strongly on LLVM's AVX-512 IFMA codegen.
- Build tooling: **CMake**, **Ninja**, **m4** (FLINT's `configure` aborts without it), a C/C++ toolchain.
- **GMP**, **MPFR**, and **FLINT 3.2.1** — required only for the FLINT baseline columns; VROOM itself relies solely on BoringSSL `BN_*` routines. **Google Benchmark** is vendored in `third_party/` and built by CMake (no system package needed).
- Automated install: `./install_deps.sh` installs clang 21.1.8 (LLVM prebuilt) under `$HOME/LLVM-21.1.8-Linux-X64`, FLINT 3.2.1 plus Google Benchmark from source under `$HOME/.local`, and Rust/cargo 1.86.0 via `rustup` under `$HOME/.cargo` and `$HOME/.rustup` (the last needed only for the optional Arkworks baseline of Section 6.1) — all without root. The system package manager (`sudo`) installs only the base OS build tooling (including `m4`, which FLINT's `configure` requires), the GMP/MPFR development headers, and the handful of `perl-*` modules the OpenSSL baseline needs (see below). Idempotent.
- **OpenSSL baseline** (optional — only for the OpenSSL columns of Sections 6.1–6.3): a separate fork of OpenSSL fetched on demand as a git submodule by `./reproduce.sh openssl`. OpenSSL's `Configure` and build need `perl` plus the modules `FindBin`, `IPC::Cmd`, `File::Compare`, `File::Copy`, and `Time::Piece`; OpenSSL treats these as core Perl, but a minimal Amazon Linux 2023 image splits them into separate `perl-*` packages, so `./install_deps.sh` installs them via the system package manager. The baseline also needs the **Google Benchmark** that `./install_deps.sh` builds from source under `$HOME/.local` (located via `CMAKE_PREFIX_PATH`); `./reproduce.sh openssl` then compiles OpenSSL and the harness under `baselines/openssl/bench/`.

**CPU path (repo B — elliptic-curve / pairing):**

- OS: **Amazon Linux 2023** (paper host); Ubuntu also works for builds.
- Compiler: **clang 21.1.0** (LLVM 21.1.0 prebuilt). Paper timings depend strongly on LLVM's IFMA codegen.
- Packages: C++ toolchain, GMP, **cmake**, **git**, **curl**.
- **Google Benchmark** built from source with clang 21; using a mismatched distro `libbenchmark` with clang 21 may cause linking errors.
- Automated install: `./scripts/setup_toolchain.sh`

**Vendored / pinned third-party code (included or cloned by scripts):**

- **BLST** (primary CPU baseline): modified fork in `blst/`, based on `supranational/blst` @ `8065152`.
- Optional external baselines in `baselines/` (not required for the main badge): arkworks @ `598a5fba`, zkcrypto @ `6bb9695`, zksync-crypto @ `e770ffd`, gnark-crypto @ `d1dece6` (refreshed via `baselines/clone.sh`).

**GPU path (repo B):** Ubuntu 20.04+ (paper host), NVIDIA driver with CUDA 12.x toolkit and `nvcc`, Python 3.10+ with `numpy`, `matplotlib`, and **`cuda-python 12.x`** (`scripts/requirements-gpu.txt` pins `cuda-python>=12.0,<13`; version 13.x breaks `from cuda import cuda, cudart, nvrtc` used by `basic_benchmarker.py`). CGBN is cloned by `./scripts/setup_gpu_deps.sh`. Kernels are JIT-compiled

[1] Each repo requires a different version of the LLVM toolchain (required to reproduce exact numbers), and a full installation of each toolchain requires ∼11 GB of disk space.

[2] AMI name: `al2023-ami-2023.12.20260611.0-kernel-6.1-x86_64`

with `nvcc -std=c++20`; if the default host GCC is too old, set `NVCC_CCBIN` or `CUDAHOSTCXX` to a C++20-capable `g++`/`clang++` (or put one earlier on `PATH`) before running GPU scripts—see `ARTIFACT_GPU.md`.

#### A.2.5 Benchmarks

No external datasets are required. Experiments use:

- **Precomputed RNS parameters** shipped in `src/` and `scripts/*_intrns4_params.hpp` (BLS12-381 and BN254 IntRNS4 / Matrix / MatrixNoK configurations), but can be regerenated with python scripts.
- **Reference test vectors** in `test_data/` for regression tests (see `test_data/README.md`), generated using **BLST**.

### A.3 Set-up

#### A.3.1 Installation

**Compiling Repo A.** Run the following commands on an IFMA-capable Linux host:

```
cd ~/vroomssl/
./install_deps.sh
./configure.sh
```

The script `install_deps.sh` installs all the dependencies needed to build and run VROOM and all baselines for repo A specificed in section A.2.4.

The last command configures and compiles the repository, producing `./build/bssl_bench`, the binary behind all of Sections 6.1–6.3. `configure.sh` selects the `Release` build type (required for accurate timings) and the home-installed clang and FLINT, so there is nothing else to set up. It is short by design — the exact compiler and CMake flags can be found in `configure.sh` (under 20 lines).

**Repo B CPU host.** Run the following commands on an IFMA-capable Linux host:

```
cd ~/VROOM/
./scripts/setup_toolchain.sh
```

Here `setup_toolchain.sh` (repo B) installs OS packages, LLVM 21.1.0, builds Google Benchmark with clang++, and prints `export` lines (`CXX`, `BENCHMARK_LIBS`, `BENCHMARK_INC`) for repo B's builds.

**Repo B GPU host** (separate machine):

```
cd ~/VROOM/
./scripts/setup_gpu_deps.sh
# if nvcc JIT fails (old default GCC): export NVCC_CCBIN=/usr/bin/g++-12
```

This installs CGBN and Python deps from `scripts/requirements-gpu.txt` (`cuda-python>=12.0,<13`). See `ARTIFACT_GPU.md` for CUDA toolkit and host-compiler notes.

#### A.3.2 Basic Test

**Repo A smoke test.**
From the `vroomssl` root, a fast (∼1 min) subset of the modular-multiplication experiment:

```
MODMUL_QUICK=1 ./reproduce.sh modmul
```

**Expected outcome:** a Google Benchmark table at the five table modulus sizes (1024–4096 bits) with `BM_RNSRingMultiplication` (VROOM), `BM_BSSLModMulMontgomery` (BoringSSL/OpenSSL scalar), and the FLINT rows. The VROOM rows are several times faster than both baselines at ≥1536 bits, confirming the binary, FLINT linkage, and the IFMA path all work.

**Repo B CPU smoke test.** Note that we export CXX from toolchain, so you may need to re-export in subsequent shells.

```
./scripts/smoke_test.sh
```

**Expected outcome:** the script builds BLST, runs core unit tests (`make -C src test`, `test-bls12-381-field-mul`), builds and runs the five programs under `examples/`, and prints:

```
== smoke test OK ==
```

on success (exit code 0). Any compile or test failure aborts with a non-zero exit code.

**GPU smoke test.** Run the following command on a CUDA host after `setup_gpu_deps.sh`

```
./scripts/smoke_test_gpu.sh
```

**Expected outcome:** one CGBN and one RNS GPU microbenchmark each report `correct=True`; completes in ∼5 s.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1): Correctness.** The RNS Montgomery + CRNS implementation correctly implements BLS12-381 / BN254 field arithmetic, extension-field / FP12 operations, elliptic-curve group law, and pairing-related code paths. Proven by experiment (E1) (unit tests, examples, smoke test).

**(C2): Table 11 (batch size).** On the paper CPU host, amortized $F_q$ multiplication latency decreases with batch size 1…12 as in Table 11 (register spilling at batch 12). Proven by experiment (E2); artifact rows `BM_BatchModMul_MatrixNoK_*` in `results/bench_bls12_381_table.txt`.

**(C3): Table 12 (BLS field extensions). VROOM** is faster than **BLST** on $F_q$, $F_q^2$, and $F_q^{12}$ extension operations (e.g. $F_q^{12}$ multiply $\sim$6.1×, cyclotomic square $\sim$3.6×). Proven by experiment (E2); compare `BM_FP2_*`, `BM_FP12_*` rows in `bench_bls12_381_vs_blst.txt` to Table 12.

**(C4): Table 13 (BLS pairing stack). VROOM** is faster than **BLST** on G1/G2 curve ops, Miller loop ($\sim$4.4×), final exponentiation ($\sim$3.7×), full pairing ($\sim$3.9×), and BLS verification ($\sim$4.1×). Proven by experiment (E2); `BM_G1_*`, `BM_MillerLoop_*`, `BM_Pairing_RNS_BLST_Inverter_*`, and computed `BM_Pairing_ComputedVerify_*` rows.

**(C5): Table 10 (pairing libraries). VROOM** pairing latency is lower than arkworks, zkcrypto, zksync-crypto (Pairing CE), gnark, and **BLST** on BLS12-381 (paper: **VROOM** ≈128 $\mu$s vs. **BLST** ≈504 $\mu$s). Proven by experiment (E3): `reproduce_cpu_bench.sh` plus optional `baselines/reproduce_baselines.sh bls12`.

**(C6): Table 14 (BN254 G1). VROOM** G1 add / mixed-add / double on BN254 is faster than arkworks, zkcrypto, and gnark-crypto (paper: ≈85–98 ns vs. 144–315 ns). Proven by experiment (E4): `make -C src bench-ec-bn254` and/or `baselines/reproduce_baselines.sh bn254`.

**(C7): GPU modular multiplication (latency sweep and Figures 5–6).** On RTX 3090, VROOM GPU RNS Montgomery has lower per-multiplication latency than CGBN over a 64–4096 bit modulus sweep; RNS speedup vs. CGBN approaches $\sim$4× at large moduli (Figure 5); TPI= 2$t$ and warp-shuffle instructions improve small-modulus latency (Figure 6). Proven by experiment (E5): `./scripts/reproduce_gpu_latency.sh` (core sweep) and `scripts/gpugraph.py` (Figures 5–6).

**(C8): Section 6.1, modular multiplication.** On the paper CPU host, VROOM's single modular-multiplication latency is lower than `BN_mod_mul_montgomery` (the BoringSSL/OpenSSL scalar routine), FLINT, and arkworks across a 64–4096-bit modulus sweep; for moduli ≥1500 bits the speedup is ≈4× over `BN_mod_mul_montgomery` and FLINT and ≈8× over arkworks (the modular-multiplication latency/speedup figure in Section 6.1). Proven by experiment (E6); produced by repo A.

**(C9): Section 6.2, modular exponentiation (Tables 7 and 9).** VROOM is faster than BoringSSL and FLINT for both public-exponent ($e = 65537$) and secret-exponent modular exponentiation across 1024–4096-bit moduli (Table 7), and faster than BoringSSL for RSA-CRT exponentiation at 2048/3072/4096-bit moduli (Table 9; e.g. ≈2.3× at RSA-2048). Proven by experiment (E7); produced by repo A.

**(C10): Section 6.3, RSA signatures (Table 8).** VROOM lowers RSA signing and verification latency versus BoringSSL for RSA-2048/3072/4096 (sign ≈2.3–3.2×, verify ≈3.3–3.6×). Versus OpenSSL (paper abstract: RSA-4096 *verify* 4.0×, *sign* 1.3×), the OpenSSL column is an *external* baseline produced by a separate forked-OpenSSL harness in `baselines/openssl` (run via `./reproduce.sh openssl rsa`); `bssl_bench` itself directly reproduces the VROOM and BoringSSL columns. Proven by experiment (E8); produced by repo A.

### A.4.2 Experiments

**(E1): Functional verification** [15 human-minutes+ 3–5 compute-minutes + ∼500 MB disk]

**How to:** Confirm the artifact builds and passes regression tests and minimal examples.

**Preparation:** Complete Set-up (Installation) above on an IFMA host, or set `FALLBACK=1` on a non-IFMA host if you're just testing correctness.

**Execution:**
```
./scripts/smoke_test.sh
# optional extended tests:
make -C src test-pairing test-ec \
    test-bls12-381-pairing
make -C examples && \
    ./examples/01_parameter_setup
```

**Results:** All commands exit 0. Smoke test prints `== smoke test OK ==`. Extended tests print pass summaries with no assertion failures.

**(E2): Tables 11–13: BLS12-381 CPU AVX-512 IFMA vs. BLST** [60 human-minutes + ∼2 compute-minutes + ∼8 MB `results/`]

**How to:** Reproduce Google Benchmark rows that populate Tables 11–13 (MatrixNoK configuration on c7i.metal-24xl).

**Preparation:** AWS c7i with IFMA:
```
eval \
"$(./scripts/setup_toolchain.sh --print)"
```

**Execution:** `./scripts/reproduce_cpu_bench.sh`

**Results:** Use `results/bench_bls12_381_table.txt` and `bench_bls12_381_vs_blst.txt` (ratio = `blst_ns/rns_ns`; > 1 means VROOM is faster).

**Table 11:** amortized ns from `BM_BatchModMul_MatrixNoK_{1,2,4,6,8,10,12}`.

**Table 12:** `BM_ModMul_MatrixNoK`, `BM_FP2_Mul_MatrixNoK`, `BM_FP12_Mul_MatrixNoK`, `BM_FP12_CyclotomicSqr_MatrixNoK`, `BM_FP12_Inverse_RNS_BLST_MatrixNoK` vs. aligned BLST rows (speedups ≈1.3–6.1×) Python script contains mapping of VROOM to BLST benchmark names.

**Table 13:** `BM_G1_*`, `BM_G2_*`, `BM_MillerLoop_MatrixNoK` (≈50 $\mu$s), `BM_FP12_FinalExp_BLST_Inverter_MatrixNoK`, `BM_Pairing_RNS_BLST_Inverter_MatrixNoK` (≈128 $\mu$s), and computed

BM_Pairing_ComputedVerify_MatrixNoK (≈4× over BLST formula sum).

**(E3): Table 10: BLS12-381 pairing vs. external libraries** [45 human-minutes + ∼30 compute-minutes + several GiB baselines/]

**How to:** Compare full pairing latency across arkworks, zkcrypto, zksync-crypto, gnark-crypto, **BLST**, and VROOM.

**Preparation:** Rust, Go, and CPU toolchain (§ Software dependencies). Vendored trees under baselines/; pinned commits in baselines/clone.sh.

**Execution:** Run the following two commands:

```
# BLST
./scripts/reproduce_cpu_bench.sh
# arkworks, zkcrypto, zksync, gnark
./baselines/reproduce_baselines.sh bls12
```

Pairing times are in baselines_bls12_bench.log (Criterion / Go ns/op) and bench_bls12_381_table.txt (BM_Pairing_RNS_BLST_Inverter_MatrixNoK, BM_FullPairing in BLST JSON).

**Results:** Manually align pairing rows to Table 10 (no automated parser).

**(E4): Table 14: BN254 G1 elliptic-curve operations versus external libraries** [30 + ∼10 compute-minutes]

**How to:** Compare BN254 G1 add, mixed-add, and double latency.

**Preparation:** Same as E2 and E3, but with bn254 verison of script.

**Execution:** `make -C src bench-ec-bn254`
`./baselines/reproduce_baselines.sh bn254`

**Results:** ./src/bench_ec_bn254 JSON or stdout for BM_G1_Add_MatrixNoK, BM_G1_MixedAdd_MatrixNoK, BM_G1_Double_MatrixNoK. Compare to arkworks / zkcrypto / gnark rows in baselines_bn254_bench.log. Paper reference: VROOM ≈85–98 ns vs. baselines 144–315 ns.

**(E5): GPU latency sweep and Figures 5–6** [20–30 human-minutes GPU setup + ∼8 compute-minutes + ∼50 MB results/ and scripts/data/]

**How to:** Run the full GPU modular-multiplication latency sweep (CGBN vs. VROOM RNS), then regenerate the supplementary GPU plots for Figures 5–6.

**Preparation:** RTX 3090 (or similar), CUDA 12.x toolkit, and ./scripts/setup_gpu_deps.sh (installs cuda-python>=12.0,<13). If JIT compile fails because the default host GCC is too old for C++20, set NVCC_CCBIN or CUDAHOSTCXX to a newer g++ before running (see ARTIFACT_GPU.md). For Figures 5–6, scripts/gpugraph.py reads pinned CSVs under scripts/data/ (listed in the script header); the latency sweep writes a fresh CSV under scripts/data/<timestamp>.csv and copies to results/gpu_latency.csv.

**Execution:** From the repo root (both steps required):

```
./scripts/reproduce_gpu_latency.sh
cd scripts
export MPLBACKEND=Agg
python3 gpugraph.py
```

Step 1 compiles many nvcc kernels (dominates wall time). Step 2 is fast if using the shipped scripts/data/*.csv inputs; re-point gpugraph.py to the newest sweep CSV if regenerating all plots from scratch.

**Results: Latency sweep:** results/gpu_latency.csv, results/gpu_latency.png, and results/gpu_latency_resources.txt (paper host: latency_sweep_elapsed_s=491.68 on RTX 3090, 262 sweep rows; peak RSS ∼1.8 GiB). Every CSV row should have correct=True. RNS should beat CGBN in the mid-bitrange. **Figures 5–6:** scripts/CGBNBaseline.png (Figure 5: speedup vs. CGBN approaches ∼4×) and scripts/TPIWS.png (Figure 6: TPI= $2t$ and warp-shuffle gains on small moduli). Compare sweep shape to shipped scripts/data/1748900378.1196628.csv if not regenerating TPI plots from the new sweep file.

**(E6): Section 6.1: modular multiplication latency / speedup** [10 human-minutes + ∼1 compute-minute (quick subset) or ∼25 compute-minutes (full 64–4096-bit sweep), + ∼10 compute-minutes for the optional arkworks baseline]

**How to:** Reproduce the Section 6.1 figure comparing VROOM to BN_mod_mul_montgomery, FLINT, and arkworks.

**Preparation:** Build repo A (Set-up, repo A). The arkworks baseline additionally needs Rust/cargo, which ./install_deps.sh installs under $HOME via rustup; ./reproduce.sh puts it on PATH automatically, so no manual setup is required.

**Execution:** From the vroomssl root (all pinned to core 0):

```
./reproduce.sh modmul          # full sweep
# MODMUL_QUICK=1 ./reproduce.sh modmul
#   restricts to the 5 table sizes
./reproduce.sh arkworks        # arkworks
./reproduce.sh openssl modmul # OpenSSL
```

**Results:** BM_RNSRingMultiplication is VROOM; BM_BSSLModMulMontgomery is the scalar baseline; FLINT is BM_FLINTMPN<limbs> (≤1024 bits) plus BM_FLINTFMPZRing (larger); arkworks prints mul <bits> Criterion lines. For moduli ≥1500 bits, VROOM is ≈4× faster than the scalar baseline and FLINT and ≈8× faster than arkworks. (Raw bssl_bench filters are in repo A's README.md.) The OpenSSL line comes from the separate forked-

OpenSSL harness: `BM_OSSLModMulMontgomery` (the scalar `BN_mod_mul_montgomery`, essentially identical to BoringSSL) and the AVX-512 IFMA AMM points `BM_AMM52x{20,30,40}_x1_ifma256` at 1024/1536/2048 bits, over which VROOM is ≈1.7× faster for ≥1500-bit moduli.

**(E7): Section 6.2: modular exponentiation and RSA-CRT (Tables 7, 9)** [10 human-minutes + ∼10 compute-minutes]
**How to:** Reproduce the generic public-/secret-exponent exponentiation table (Table 7) and the RSA-CRT exponentiation table (Table 9).
**Preparation:** Build repo A (Set-up, repo A).
**Execution:** From the `vroomssl` root:

```
# Table 7
./reproduce.sh modexp          # VROOM
./reproduce.sh openssl modexp  # OpenSSL
# Table 9
./reproduce.sh rsacrt          # VROOM
./reproduce.sh openssl rsacrt  # OpenSSL
```

**Results:** Table 7 (microseconds, moduli 1024–4096): *public* $e = 65537$ rows `BM_RNSModExpRSAE` (VROOM), `BM_BSSLModExpRSAE` (BoringSSL), `BM_FLINTExponentiationRSAE` (FLINT); *secret*-exponent rows `BM_RNSExponentiation<bits,window>` (VROOM; the paper reports the fastest window per size), `BM_BSSLModExpConsttime`, and `BM_FLINTExponentiation` (FLINT, not constant-time). Table 9 (RSA-CRT, microseconds): `BM_SpeedRNSRSAModExpCRT` (VROOM) vs. `BM_SpeedRSAModExpCRT` (BoringSSL). VROOM is faster than every in-binary baseline. The OpenSSL columns are produced by the separate forked-OpenSSL harness: `BM_OSSLModExpRSAE` (public *e*), `BM_OSSLModExpConsttime` / `BM_OSSLModExpConsttime_x2` (secret *e*; the paper reports whichever is faster), and `BM_OSSLRSAModExpCRT` (RSA-CRT).

**(E8): Section 6.3: RSA signing and verification (Table 8)** [5 human-minutes + ∼1 compute-minute]
**How to:** Reproduce the RSA sign/verify latency table.
**Preparation:** Build repo A (Set-up, repo A).
**Execution:** From the `vroomssl` root:

```
./reproduce.sh rsa         # VROOM + BoringSSL
./reproduce.sh openssl rsa # OpenSSL column
```

**Results:** Microseconds for RSA-2048/3072/4096. VROOM rows `BM_SpeedRNSRSASign` / `BM_SpeedRNSRSAVerify`; BoringSSL rows `BM_SpeedRSASign` / `BM_SpeedRSAVerify`. VROOM signing is ≈2.3–3.2× and verification ≈3.3–3.6× faster than BoringSSL. The OpenSSL column of Table 8 is an external baseline produced by a separate forked-OpenSSL harness (`baselines/openssl`, fetched on demand as a git submodule the first time it is requested); `./reproduce.sh openssl rsa` builds it and prints the `BM_RSASign_RSA_sign` / `BM_RSAVerify_RSA_verify` rows, measuring the same `RSA_sign` / `RSA_verify` functions as BoringSSL (VROOM: RSA-4096 verify 4.0×, sign 1.3× over OpenSSL).

## A.5 Notes on Reusability

The following apply to repo B.

- **Minimal API tour:** `make -C examples` and `examples/README.md` (five programs covering parameter setup, singular modmul, sum/difference/product of products).
- **Integer fallback:** `Makefile.fallback` / `FALLBACK=1` for machines without IFMA—correctness only.
- **Parameter regeneration:** `scripts/rns_secp256k1.py` and `scripts/gen_qr.py` re-emit `.hpp` parameter files; rebuild and re-run `smoke_test.sh` afterward.
- **Module map:** `src/README.md`, `cpu/`, `gpu/README.md`.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.