Citizen Technologies: noble-curves

Security Assessment

March 7, 2023

Prepared for:
Ryan Shea
Citizen Technologies

Prepared by: **Joop van de Pol and Opal Wright**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

This report is considered by Trail of Bits to be public information; it is licensed to Citizen Technologies under the terms of the project statement of work and has been made public at Citizen Technologies’ request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Citizen Technologies engaged Trail of Bits to review the security of the `noble-curves` library. From January 27 to February 6, 2023, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and documentation. We performed a manual analysis of the source code, aided by static analysis tools. At the customer's request, we also reviewed the noble-curves git repository to evaluate its speed of development and risk profile.

Summary of Findings

The audit uncovered some flaws that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Informational	1

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 27, 2023	Project kickoff call
February 7, 2023	Delivery of report draft
February 8, 2023	Report readout meeting
March 7, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the noble-curve library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the integer and elliptic curve arithmetic implemented correctly?
    - In particular, is the secp256k1 implementation sound?
- Is there a risk that a user could create invalid/unusable elliptic curve keys?
- Are there any flaws that would compromise funds in stealth wallets?
- Are there any flaws that would compromise anonymity for stealth wallets?
- Does the API present serious misuse possibilities?
- Are there significant supply-chain risks associated with the library?
- Does the speed and recency of development present a security risk?
- Is the `POSEIDON` hash implemented securely?
- Are there “specific condition attacks” that could cause problems, such as timing attacks?

Project Targets

The engagement involved a review and testing of the following target:

noble-curves

Repository	https://github.com/paulmillr/noble-curves
Version	`7262b4219f8428dfa39ac4c81b25660ddc6a4614`
Type	Typescript library
Platform	Web browser

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- Manual review of relevant portions of the codebase
- Use of static analysis tools to identify common errors
- Review of the GitHub repository commit history
- Dependency review

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Curves other than secp256k1, including Montgomery and Edwards arithmetic
- Security of dependencies (large-integer arithmetic and random-number generation)
- Pairing functionality
- Hash-to-curve functions other than `POSEIDON`

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix E.1
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix E.2

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Finite field and elliptic curve arithmetic appears to be well implemented. We found some opportunities for speed-ups and timing mitigation discussed in Appendix C.	Strong
Auditing	Logging is not part of this library.	Not Applicable
Authentication / Access Controls	Authentication and access control are not parts of this library.	Not Applicable
Complexity Management	The code is mostly well divided into separate modules with intuitive names and reasonable functionalities. There are some exceptions, as discussed in Appendix C. There are some instances of duplicated code, as discussed in Appendix C.	Moderate
Cryptography and Key Management	Signature generation and validation appear to be correctly handled, including edge cases. Interpreted, garbage-collected languages like JavaScript and TypeScript make it difficult to clear private key data; see comments in Appendix C.	Satisfactory
Data Handling	The codebase contains frequent calls to the `assertValidity` function to validate that points are correctly formed and on-curve, including in commonly	Strong

	used construction functions like `fromHex`. Specifically, all functions that convert library input data to curve points (and vice versa for output data) call `assertValidity`.	
Documentation	The codebase contains significant inline documentation. Static analysis showed that some inline function documentation did not match the associated functions (for instance, variables were renamed). See Appendix C for details.	**Satisfactory**
Memory Safety and Error Handling	There are 187 `throw` statements, but only eight `catch` statements. Most functions allow exceptions to propagate up, but some explicitly catch and convert lower-level exceptions. Consider designating a preferred exception handling method, then documenting exceptions when needed.	**Satisfactory**
Testing and Verification	The library includes an extensive test suite covering all of its provided functionality. The test suite includes verification of standard test vectors (including but not limited to Wycheproof), both positive and negative test cases, and test cases targeting curve-specific functionality. The test coverage is high, but a few improvements can be made; see comments in Appendix C.	**Strong**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Timing issues	Cryptography	**Informational**

Detailed Findings

1. Timing issues

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-CTNC-1
Target: `src/abstract/curve.ts`	

Description

The library provides a scalar multiplication routine that aims to keep the number of `BigInteger` operations constant, in order to be (close to) constant-time. However, there are some locations in the implementation where timing differences can cause issues:

- Pre-computed point look-up during scalar multiplication (figure 1.1)
- Second part of signature generation
- Tonelli-Shanks square root computation

```
// Check if we're onto Zero point.
// Add random point inside current window to f.
const offset1 = offset;
const offset2 = offset + Math.abs(wbits) - 1; // -1 because we skip zero
const cond1 = window % 2 !== 0;
const cond2 = wbits < 0;
if (wbits === 0) {
  // The most important part for const-time getPublicKey
  f = f.add(constTimeNegate(cond1, precomputes[offset1]));
} else {
  p = p.add(constTimeNegate(cond2, precomputes[offset2]));
}
```

Figure 1.1: Pre-computed point lookup during scalar multiplication (`noble-curves/src/abstract/curve.ts:117–128`)

The scalar multiplication routine comprises a loop, part of which is shown in Figure 1.1. Each iteration adds a selected pre-computed point to the accumulator p (or to the dummy accumulator `f` if relevant scalar bits are all zero). However, the array access to select the appropriate pre-computed point is not constant-time.

Figure 1.2 shows how the implementation computes the second half of an ECDSA signature.

```
const s = modN(ik * modN(m + modN(d * r))); // s = k^-1(m + rd) mod n
```

Figure 1.2: Generation of the second part of the signature
(`noble-curves/src/abstract/weierstrass.ts:988`)

First, the private key is multiplied by the first half of the signature and reduced modulo the group order. Next, the message digest is added and the result is again reduced modulo the group order. If the modulo operation is not constant-time, and if an attacker can detect this timing difference, they can perform a lattice attack to recover the signing key. The details of this attack are described in the TCHES 2019 article by Ryan. Note that the article does not show that this timing difference attack can be practically exploited, but instead mounts a cache-timing attack to exploit it.

`FpSqrt` is a function that computes square roots of quadratic residues over F_p. Based on the value of p, this function chooses one of several sub-algorithms, including Tonelli-Shanks. Some of these algorithms are constant-time with respect to p, but some are not. In particular, the implementation of the Tonelli-Shanks algorithm has a high degree of timing variability.

The `FpSqrt` function is used to decode compressed point representations, so it can influence timing when handling potentially sensitive or adversarial data.

Most texts consider Tonelli-Shanks the “fallback” algorithm when a faster or simpler algorithm is unavailable. However, Tonelli-Shanks can be used for *any* prime modulus p. Further, Tonelli-Shanks can be made constant time for a given value of p.

Timing leakage threats can be reduced by modifying the Tonelli-Shanks code to run in constant time (see here), and making the constant-time implementation the default square root algorithm. Special-case algorithms can be broken out into separate functions (whether constant- or variable-time), for use when the modulus is known to work, or timing attacks are not a concern.

Exploit Scenario

An attacker interacts with a user of the library and measures the time it takes to execute signature generation or ECDH key exchange. In the case of static ECDH, the attacker may provide different public keys to be multiplied with the static private key of the library user. In the case of ECDSA, the attacker may get the user to repeatedly sign the same message, which results in scalar multiplications on the base point using the same deterministically generated nonce. The attacker can subsequently average the obtained execution times for operations with the same input to gain more precise timing estimates.

Then, the attacker uses the obtained execution times to mount a timing attack:

- In the case of ECDSA, the attacker may attempt to mount the attack from the TCHES 2019 article by Ryan. However, it is unknown whether this attack will work in practice when based purely on timing.
- In the case of static ECDH, the attacker may attempt to mount a recursive attack, similar to the attacks described in the Cardis 1998 article by Dhem et al. or the JoCE 2013 article by Danger et al. Note that the timing differences caused by the precomputed point look-up may not be sufficient to mount such a timing attack. The attacker would need to find other timing differences, such as differences in the point addition routines based on one of the input points. The fact that the library uses a complete addition formula increases the difficulty, but there could still be timing differences caused by the underlying big integer arithmetic.

Determining whether such timing attacks are practically applicable to the library (and how many executions they would need) requires a large number of measurements on a dedicated benchmarking system, which was not done as part of this engagement.

Recommendations

Short term, consider adding scalar randomization to primitives where the same private scalar can be used multiple times, such as ECDH and deterministic ECDSA. To mitigate the attack from the TCHES 2019 article by Ryan, consider either blinding the private scalar d in the signature computation or removing the modular reduction of $d \cdot r$, i.e., $s = modN(ik * modN(m + d * r))$.

Long term, ensure that all low-level operations are constant-time.

References

- Return of the Hidden Number Problem, Ryan, TCHES 2019
- A Practical Implementation of the Timing Attack, Dhem et al., Cardis 1998
- A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards, Danger et al., JoCE 2013

Summary of Recommendations

The `noble-curves` library is a work in progress with multiple planned iterations. Trail of Bits recommends that the developers address the findings detailed in this report and take the following additional steps prior to deployment:

- Invest time in policy development. Documentation and test policies provide a useful layer of checks to see if a new function or module is ready to be integrated: is it documented properly, and does it have good tests that cover both happy and unhappy paths? Especially as more developers contribute to the development, it will be important to ensure that their contributions meet the project's high standards.

- Invest time in “tidying” tasks such as finding duplicated functions, separating unrelated functions into different modules, and cleaning up documentation. This will reduce attack surface, make analysis easier, and help prevent library misuse down the road.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.

Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Maturity Recommendations

Trail of Bits recommends the following steps to enhance code maturity.

Arithmetic

In FpIsSquare, the result is based on calculating the Legendre symbol of the input with respect to the modulus. The Jacobi symbol is faster to compute, and equivalent to the Legendre symbol when the modulus is prime.

The advantage of the Legendre symbol is that it runs in constant time for a given modulus. The downside is that it is significantly slower than the Jacobi symbol. The standard algorithm for computing the Jacobi symbol does not run in constant time, but recent algorithmic improvements have made constant-time Jacobi symbol computations possible.

If the FpIsSquare function turns out to be a bottleneck in practice, it may be worth implementing the constant-time Jacobi symbol algorithm for a speedup.

Complexity Management

As noted in the Code Maturity Evaluation, the functions modules are, in the main, cleanly and logically laid out. There are a few exceptions, however.

First, in weierstrass.ts, there is an implementation of the HMAC-DRBG construction from NIST SP800-90. HMAC-DRBG is a high-quality random number generator, and its use for digital signatures is a good design choice. However, the implementation of the DRBG likely belongs elsewhere; it has no significant relationship to the elliptic curve arithmetic the weierstrass.ts file is meant to implement. It may be better in utils.ts or possibly even integrated into the noble-hashes library.

Second, there are several places where square root functionality is duplicated throughout the code. FpSqrt is implemented in modular.ts, and square root functions are also present in bls12-381.ts and ed25519.ts.

Cryptography and Key Management

Private keys are formatted in different ways at different points throughout the codebase. Memory behavior in interpreted languages is never guaranteed, so zeroization is a hard problem in TypeScript/JavaScript.

However, on some systems, the underlying memory of UInt8Array objects (which are one of the data types used to store keys) is effectively a thin layer on top of C-style data buffers. Consider zeroizing keys that are stored as UInt8Array objects.

Documentation

The library contains many useful comments for users and developers. However, there are several instances where comments are outdated or incorrect. This section summarizes these instances.

Most functions contain comments that describe the parameters and return values. However, the following functions (in the parts of the library covered by the audit) have comments listing parameters that have either been renamed or removed, or list only some but not all parameters:

- `precompute`, `multiply`, and `sign` in `weierstrass.ts`
- `wNAF` in `curve.ts`
- `schnorrSign` in `secp256k1.ts`

In general, not all functions use the same structured approach to describe their parameters and return values. We recommend unifying the approach and using it consistently for all functions.

Additionally, some comments appear to be incorrect. For example:

- For the function `precomputeWindow` in `curve.ts`, a comment states that for a window size of 8, the number of precomputed points is 65,536. This number depends on the curve size, and the actual formula corresponding to the implementation is $2^{W-1} \cdot ([n/W] \ + 1)$, where [.] is the ceiling function, W is the window size, and n is the bitlength of the curve order. Therefore, for a 256-bit curve and window size 8, the number of precomputed points is $128 \cdot 33 = 4224$.
- For the function `wNAF` in `curve.ts`, a comment states that the function will fail if the scalar is larger than the group order. However, the precomputed windows should cover at least all scalars of bit lengths up to $W \cdot ([n/W] \ + 1) - 2$. For a 256-bit curve with window size 8, this corresponds to $8 \cdot 33 \ - 2 = \ 262$ bits. A 264-bit scalar with the most-significant bit set would fail, because there is no pre-computed window to process the carry. A 263-bit scalar with the most-significant bit set may fail in case a carry propagates, resulting in the same issue.
- The same function contains a comment regarding the accumulators `f` and `p` and the infinity point. The accumulator `f` is initialized to the base point, and for every window where the scalar is all-zero, the first precomputed point (i.e., not a random point, as stated in the comment) of this window is added to it or subtracted from it. Similarly, the accumulator `p` is initialized to the point at infinity, and for every window where the scalar is non-zero, the corresponding precomputed point of this window is added to it or subtracted from it. Because the windows do not overlap, it is not possible for a precomputed point in a window to cancel out (a sum of) precomputed points of earlier windows. Therefore, the only way to obtain the point at infinity is when the scalar processed so far corresponds to a multiple of the group order (including zero). In any case, the accumulator `p` is initialized to the point at

infinity, so the first addition into p will always need to deal with the point at infinity. This should not be a problem, because a complete addition formula is used.

- In general, the usage of wNAF to describe the implemented scalar multiplication algorithm is slightly confusing. In a usual wNAF implementation, only a single window is precomputed and the scalar is processed in a left-to-right manner, with corresponding doubling of the accumulator(s), where additional zeroes are skipped. In this implementation, fixed windows are used, and all fixed windows are precomputed (i.e., each [multiple of W]-bit shift of the initial window).

We recommend clarifying these comments to prevent confusion.

Testing and Verification

The tests included in the library achieve a high test coverage. However, some functions are still not covered. The following list includes the untested functions in the parts of the library covered by the audit:

- `addRecoveryBit`, `normalizeS`, and `toCompactRawBytes` in `weierstrass.ts`
- `FpDiv` and `FpSqrtOdd` in `modular.ts`

We recommend adding test cases for each function (both positive and negative test cases, if applicable).

Automated testing also found that the `pow` method is defined twice for `Field<T>` in `modular.ts`, on lines 225 and 238.

In addition, the library takes user input in the form of elliptic curve points and (potentially) DER-encoded signatures. We recommend adding fuzzing test cases for all functions taking such user input, in order to determine whether the library exhibits any unexpected behavior for particular edge cases.

D. Analyzing Codebase Evolution

The noble-curves codebase has undergone consistent evolution since its initial commit on December 4, 2022. The commit history shows a consistent pattern of frequent, small changes. These changes are all made by Paul Miller, the library author and maintainer.

Our analysis focused on commit hash 7262b4219f8428dfa39ac4c81b25660ddc6a4614. Between that commit (made on January 26) and the time of this analysis (February 7), there have been an additional 19 commits to the noble-curves codebase. Most commits are small, and some deal with administrative tasks such as updating library versions for developer dependencies.

Ignoring the commits related to the README files and administrative tasks, the median commit during this interval added 32 lines of code and removed 21. There are some outliers, like commit c75129e629c46c5bdc222be93af1e5943eac4ee3, which added 179 lines and deleted 207, but the vast majority of the changes made to the codebase are small and confined to two or fewer files.

The small-but-frequent commit approach is a double-edged sword for security analysis. Tracking lots of commits can be tricky, especially in distributed development scenarios where changes can overlap, but smaller commits are easier to analyze. Larger, less frequent commits are easier to track, but require more effort to analyze.

On the whole, we believe the small-but-frequent approach is best in this case. Only one developer is active on the project at the moment (so development is mostly linear), and the commits are centralized to a single place (so they are easy to track). Assuming the changes are appropriately tested before committing, this approach can be helpful to secure development and easier ongoing review.

One example of this analysis-easing approach comes from commits dbb16b0e5ee86a660347f8527896cfd5c4f0623f and e57aec63d8fbbc32eea966c85bca9cc66df321e9. In the first commit, an assertValidity function was added for Edwards curves. In the second commit—made the same day—a typo and a minor logical bug in the same function were fixed. The first commit was logical, legible, and well documented. The second commit was limited in its purpose, and obvious in its result.

It is also worth noting that the noble-curves commit messages are concise and descriptive, giving good, high-level overviews of the associated changes. Good commit messages can help security reviewers with triage and analysis; good documentation helps with good security.

E. Automated Analysis Tool Configuration

E.1. Semgrep

We used Semgrep to identify known vulnerabilities in the codebase, but did not identify any issues. The command used to run Semgrep was `semgrep --config auto`.

E.2. CodeQL

We used CodeQL to identify known vulnerabilities in the TypeScript/JavaScript codebase, but we did not identify any security issues. The commands used to run this tool are shown in the figure below.

```
# Create the TypeScript database
codeql database create codeql --language=javascript --source-root=noble-curves-main
# Run all JavaScript and TypeScript queries
codeql database analyze codeql --format=sarif-latest
--output=codeql_tob_javascript.sarif -- tob-javascript-all.qls
```

Figure E.1: The commands used to run CodeQL on the `noble-curves` codebase

F. Supply-Chain Analysis

noble-curves lists one main dependency: the noble-hashes library, a project led by the same developer as noble-curves. The noble-hashes library lists no non-development dependencies.

noble-curves relies on some built-in functionality provided by the JavaScript environment, such as random number generation. However, we consider a compromised JavaScript interpreter to be outside the scope of a supply-chain attack.

Assuming that noble-hashes is provided by a reliable source, noble-curves is well-protected from supply-chain attacks *in production*.

Supply-chain attacks against the development environment are more difficult to analyze. There are several development dependencies for both libraries. Several of the dependencies (including micro-bmark and micro-should) are zero-dependency libraries maintained by the author of noble-curves.

Some libraries, like prettier, are developed and maintained outside of the closed ecosystem in which noble-curves and noble-hashes are developed, and those libraries can have significant exposure to supply-chain attacks.

A number of JavaScript package managers allow downloaded packages to run custom installation commands. It is possible that a malicious package could be downloaded while setting up development dependencies, allowing the attacker to modify files within the developer's local copy of the noble-curves repository. If the malicious modifications were then committed and pushed, the noble-curves library could be compromised.

Several tools are available to help prevent supply-chain attacks. Tools like it-depends can help build a picture of the overall supply-chain exposure by tracing dependencies and providing a full list of packages associated with a dependency. If using npm, the npm audit command can alert developers that selected packages rely on known-vulnerable or known-malicious library versions.

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On February 17 2023, Trail of Bits reviewed the fixes and mitigations implemented by the noble-curves team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, the `noble-curves` team partially resolved the timing issues described in this report, and fully resolved several code maturity considerations. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Timing issues	Informational	Partially resolved

Detailed Fix Review Results

TOB-CTNC-1: Timing issues

Partially resolved.

Blinding has been implemented to mitigate timing attacks during signature generation. Comments note that underlying large-arithmetic code may not be constant time, and thus may still leak some information.

`FpSqrt` remains unchanged, but may be updated in the future. Per the development team:

> *Wontfix for now. Square root works with public data: decompressing public keys, etc. It is not used with private data. Our Jacobi implementation increases speed by 8% while massively increasing complexity. We will leave this as-is.*

Access to pre-computed point lookup tables remains unchanged. Per the development team:

> *Wontfix. True, however, every new window does not intersect the old one. So second window cannot hit cached item from first window. It's only possible if some array items were cached long-term, but that seems unlikely judging from the enormous amounts of data that would overwrite the cache.*

Code maturity improvements:

Jacobi symbol computations were investigated. As noted in the comments for TOB-CTNC-1, the increased complexity of the Jacobi symbol code was not worth the increase in speed.

The HMAC-DRBG code has been moved to `utils.js`, improving code structure.

Tests have been added to cover the functions `addRecoveryBit`, `normalizeS`, `toCompactRawBytes`, `FpDiv`, and `FpSqrtOdd`. As a result, all functions within the scope of this code audit are now covered by tests.

Developers have used the `cryptofuzz` tool to fuzz the library. We have not reviewed the results.

Documentation for several of the functions we identified has been updated and clarified, addressing some outdated information and reducing the risk of user error.

Other considerations

The documentation for the library has been updated to include discussion of dependencies, including developer dependencies. The risks and mitigations are laid out clearly, giving contributors a clear understanding of the supply-chain risks.