

Ripple Labs XRP Ledger Confidential Transfer

Security assessment by HashEye · prepared for Ripple Labs

HASHEYE AUDITED

PROJECT	Ripple Labs XRP Ledger Confidential Transfer
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	Ripple Labs
CATEGORY	Blockchain
PUBLISHED	April 1, 2026
REPORT ID	research-ripple-labs-xrp-ledger-confidential-transfer-2026-04-01-1bkbv4

This report was produced by HashEye's security review team under the HashEye review process, combining source review, deployment-state checks, and senior manual verification. Full report detail is available at hasheye.io/audits/research-ripple-labs-xrp-ledger-confidential-transfer-2026-04-01-1bkbv4.

Ripple Labs XRP Ledger Confidential Transfer Security Assessment April 9, 2026

Prepared for: Kenny Lei Ripple Labs, Inc.

Prepared by: Paul Bottinelli and Opal Wright

HashEye

PUBLIC

Table of Contents Table of Contents 1 Project Summary 3 Executive Summary 4 Project Goals 7 Project Targets 8 Project Coverage 9 Automated Testing 10 Summary of Findings 12 Detailed Findings 14 1. Undocumented ElGamal decryption limit 14 2. Deprecated OpenSSL SHA-256 API and unchecked return values 16 3. Fiat-Shamir domain tag considerations 18 4. Timing side channels in ElGamal decryption 20 5. Timing side channels in Bulletproof generation 22 6. Aggregated Bulletproofs cannot handle certain in-range values 24 7. Missing participant count validation 25 8. Missing return value checks 29 9. Insufficient compiler warning flags and security hardening options 31 10. Missing range proof enables confidential balance overdraft 34 11. ConfidentialClawback resets the balance version counter 36 12. ConfidentialSend increments the receiver's version counter 38 13. MergeInbox allows version counter increment without balance change 40 14. Discrepancies between specification fields and implementation 42 15. Incorrect TransactionContextID for Convert and ConvertBack 44 16. Multi-ciphertext equality proof uses per-recipient nonces instead of shared r 46 17. Missing input length validation in low-level cryptographic helper functions 48 A. Vulnerability Categories 51 B. Code Quality Issues 53 C. Performance Improvement Opportunities 57 D. API Improvement Opportunities 58 E. Automated Analysis Tool Configuration 63 F. Fix Review Results 64 Detailed Fix Review Results 65

HashEye 1 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

6. Fix Review Status Categories 68 About HashEye 69 Notices and Remarks 70

HashEye 2 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Project Summary Contact Information The following project manager was associated with this project: Emily Doucette, Project Manager emily.doucette@hasheye.io The following engineering director was associated with this project: Jim Miller, Engineering Director, Application Security and Cryptography james.miller@hasheye.io The following consultants were associated with this project: Paul Bottinelli, Consultant Opal Wright, Consultant paul.bottinelli@hasheye.io opal.wright@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event January 29, 2026 Pre-project kickoff call February 9, 2026 Status update meeting #1 February 17, 2026 Status update meeting #2 March 2, 2026 Delivery of report draft and report readout meeting April 2, 2026 Completion of fix review April 9, 2026 Delivery of final comprehensive report

HashEye 3 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Executive Summary Engagement Overview Ripple Labs engaged HashEye to review the security of its XRP Ledger confidential transfer implementation. The XRP Ledger confidential transfer system provides confidential multipurpose tokens (CMPTs), allowing users to maintain accounts with balances hidden from other users and to make transactions that keep transfer amounts confidential. The implementation introduces new transaction types; a dedicated cryptographic library (mpt-crypto) providing ElGamal encryption, Pedersen commitments, and Bulletproof range proofs; and several new

zero-knowledge proof systems. A team of two consultants conducted the review from February 2, 2026 to February 27, 2026, for a total of six engineer-weeks of effort. Our testing efforts focused on protocol correctness, cryptographic primitive correctness, memory handling, error handling, and overall design. With full access to source code and documentation, we performed static and dynamic testing of the underlying cryptographic primitives and the protocol implementation, using automated and manual processes. **Observations and Impact** We identified 17 findings across the two codebases, spanning cryptographic correctness, protocol conformance, data validation, and configuration. Three overarching issues account for the majority of these findings: a Bulletproof range proof component not yet integrated into the protocol layer, repeated deviations from the CMPT protocol specification, and insufficient input validation and edge-case handling. The most pressing finding is that ConfidentialConvertBack and ConfidentialSend do not include a range proof to verify that a sender's spending balance exceeds the claimed transaction amount (TOB-RIPCTX-10). Without this proof, any account holder can claim more tokens than their confidential balance holds, effectively minting unbacked plaintext tokens and draining the issuance pool. The mpt-crypto library already implements the required Bulletproof range proof, but at the time of the review, that library had not been integrated as a dependency; its functions were instead copied directly into the rippled C++ source files, and the range proof call sites in the transaction handlers were left as acknowledged stubs. The two affected transaction types must not be enabled until this gap is closed. Two additional high-severity findings concern protocol-level correctness. The aggregated Bulletproof implementation fails to process inputs whose values are all zero or all UINT64_MAX (TOB-RIPCTX-6), covering valid in-range amounts that holders may legitimately hold. If a holder attempts to spend an entire balance but submits inputs of either all zero or the maximum representable amount, they would receive a proof failure with no indication of the cause. Separately, ConfidentialSend incorrectly increments the

HashEye 4 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

replay-protection version counter on the receiver's account in addition to the sender's (TOB-RIPCTX-12). Because any pending proof encodes the version at generation time, an incoming transfer from any counterparty silently invalidates precomputed proofs in the recipient's wallet, leaving the recipient unable to submit previously prepared transactions. Three medium-severity findings reflect narrower deviations from the specification. ConfidentialClawback resets the version counter to zero rather than incrementing it (TOB-RIPCTX-11), which can lock a holder out of the confidential system after an issuer-initiated clawback. The TransactionContextID transcript for Convert and ConvertBack does not match the notation in the specification (TOB-RIPCTX-15), meaning any third-party client that implements proof generation according to the specification will produce proofs that fail on-chain verification. Finally, the multi-ciphertext equality proof generates per-recipient nonces rather than a shared randomness scalar, as the specification defines (TOB-RIPCTX-16); this produces proofs roughly twice the expected size and breaks interoperability with specification-compliant implementations. Overall, we found the code to be well documented, particularly in the cryptographic library, though the coding style across the codebase is inconsistent. We identified several non-security-related code quality issues, which we document in appendix B. We also identified several design decisions that unnecessarily complicate the API and contributed to several of the findings in this report; some opportunities to improve the library design are further documented in appendix D. **Recommendations** Based on the findings identified during the security review, HashEye recommends that Ripple Labs take the following steps before releasing the confidential transfer protocol of the XRP Ledger software:

- Remediate the findings disclosed in this report. These findings should be addressed through direct fixes or broader refactoring efforts.
- Incorporate Bulletproofs. The Bulletproof range proofs are a critical component of the confidential transaction protocol, but have not yet been incorporated into the relevant portions of the confidential transaction handling code (see TOB-RIPCTX-10). This must be addressed before the confidential transfer system is brought online.
- Match the implementation more closely with the specification. As mentioned in the "Observations and Impact" section, several of the findings in this report stem from a failure to closely adhere to the CMPT specification provided by Ripple Labs. Ongoing code reviews to ensure full compliance with the specification should be an integral part of the development process. Incorporating portions of the protocol specification into the relevant code (as is done in the cryptographic primitive code) may help with future review and development.

HashEye 5 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

- Significantly expand testing. Several of the issues identified in this report (e.g., TOB-RIPCTX-6) could be identified through a more rigorous testing setup that checks “unhappy path” execution and edge cases. Additionally, the higher-level protocol relies on maintaining several key invariants; an extensive and clearly documented invariant testing setup is a critical component of a proper testing framework for this system.
- Refactor APIs and implementation to improve clarity and avoid unnecessary serialization. Appendix D provides a list of proposed improvements to the confidential transfer codebase that will enhance its performance, maintainability, and security by creating cleaner, easier-to-understand code that avoids several pitfalls (such as the secp256k1 library’s inability to serialize the point at infinity and undetected parameter inconsistencies). Finding Severities and Categories

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS
Severity Count High 3 Medium 3 Low 3 Informational 7 Undetermined 1

CATEGORY BREAKDOWN Category Count Configuration 1 Cryptography 7 Data Validation 7 Error Reporting 1 Patching 1

HashEye 6 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Project Goals The engagement was scoped to provide a security assessment of the XRP Ledger confidential transfer feature. Specifically, we sought to answer the following non-exhaustive list of questions: • Are cryptographic primitives correctly implemented? • Does the implementation follow the protocol specification? • Does the implementation follow best practices for C and C++? • Are there “sharp edges” in the API that could lead to security issues? • Are edge cases handled correctly and properly exercised by tests? • Are protocol-level invariants correctly maintained?

HashEye 7 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Project Targets The engagement involved reviewing and testing the following targets. mpt-crypto Repository <https://github.com/XRPLF/mpt-crypto> Version f196bdb25b39330f905b57e7b69449b2f38b7932 Type C rippled Repository <https://github.com/XRPLF/rippled/tree/ripple/confidential-transfer> Version fc8b7898c5548a3857de26ce190ea9d52804a2d5 Type C++

HashEye 8 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following: • Review of the specification described in the reference papers, ConfidentialMPT_20260201.pdf and ConfidentialMPT_20260106.pdf • Manual review of the files in the mpt-crypto codebase • Manual review of the relevant files in the rippled codebase, specifically the following: ◦ [src/libxrpl/protocol/ConfidentialTransfer.cpp/h](#) ◦ [src/xrpld/app/tx/detail/ConfidentialClawback.cpp/h](#) ◦ [src/xrpld/app/tx/detail/ConfidentialConvert.cpp/h](#) ◦ [src/xrpld/app/tx/detail/ConfidentialConvertBack.cpp/h](#) ◦ [src/xrpld/app/tx/detail/ConfidentialMergeInbox.cpp/h](#) ◦ [src/xrpld/app/tx/detail/ConfidentialSend.cpp/h](#) ◦ Files relevant to the protocol-related changes in pull request [XRPLF/rippled#5860](#)

- Static code analysis using automated tools as described in Automated Testing Coverage Limitations
- Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review: • The security of rippled, the server software that powers the XRP Ledger, outside of the relevant files described above • The security of the private ledger protocol itself • The security of the secp256k1 library used by the mpt-crypto module • The security of other dependencies, such as OpenSSL

HashEye 9 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use open-source static analysis tools, 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 Cppcheck An open-source tool for static analysis of C and C++ code Appendix E.3 Areas of Focus Our automated testing and verification work focused on the following:

- General code quality issues and unidiomatic code patterns
- Memory safety issues
- Dependencies that are outdated or vulnerable to known attacks
- General undefined behavior Test Results

The results of this focused testing are detailed below. Semgrep Semgrep identified a number of improvement opportunities, most of which are presented in appendix D. We did not identify any security issues using the C/C++ Semgrep rules. CodeQL We did not identify any security issues using the C++ CodeQL rules.

HashEye 10 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Cppcheck We did not identify any security issues using Cppcheck in the in-scope code. Cppcheck did highlight a few minor issues in test code.

HashEye 11 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

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

ID	Title	Type	Severity
1	Undocumented ElGamal decryption limit	Data Validation	Low
2	Deprecated OpenSSL SHA-256 API and unchecked return values	Patching	Informational
3	Fiat-Shamir domain tag considerations	Cryptography	Informational
4	Timing side channels in ElGamal decryption	Cryptography	Informational
5	Timing side channels in Bulletproof generation	Cryptography	Undetermined
6	Aggregated Bulletproofs cannot handle certain in-range values	Cryptography	High
7	Missing participant count validation	Data Validation	Low
8	Missing return value checks	Error Reporting	Informational
9	Insufficient compiler warning flags and security hardening options	Configuration	Informational
10	Missing range proof enables confidential balance overdraft	Cryptography	High
11	ConfidentialClawback resets the balance version counter	Data Validation	Medium
12	ConfidentialSend increments the receiver's version counter	Cryptography	High

HashEye 12 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

13	MergeInbox allows version counter increment without balance change	Data Validation	Informational
14	Discrepancies between specification fields and implementation	Data Validation	Informational
15	Incorrect TransactionContextID for Convert and ConvertBack	Data Validation	Medium
16	Multi-ciphertext equality proof uses per-recipient nonces instead of shared r	Cryptography	Medium
17	Missing input length validation in low-level cryptographic helper functions	Data Validation	Low

HashEye 13 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Detailed Findings 1. Undocumented ElGamal decryption limit Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-RIPCTX-1 Target: mpt-crypto/src/elgamal.c, mpt-crypto/include/secp256k1_mpt.h

Description The ElGamal encryption implementation accepts any 64-bit unsigned integer (0 to $2^{64}-1$) as input, but, as shown in figure 1.1, the decryption function can only recover amounts up to 1,000,000 (using a brute-force discrete logarithm approach). This asymmetry creates a silent failure mode where valid ciphertexts become unrecoverable if they encrypt values above the search range (which could also be obtained as a result of homomorphic operations). for ($i = 1$; $i \leq$

```

1000000; ++i) { // Fast comparison if (pubkey_equal(ctx, &current_M, &M_target)) { *amount = i;
return 1; }

// Increment: current_M = current_M + G pts[0] = &current_M; pts[1] = &G_point; if
(!secp256k1_ec_pubkey_combine(ctx, &next_M, pts, 2)) return 0; current_M = next_M; }

return 0; // Amount not found in range Figure 1.1: Decryption loop in function
secp256k1_elgamal_decrypt ( mpt-crypto/src/elgamal.c#L146-L159 ) Neither the API documentation nor
function signatures indicate this limitation. The header file (include/secp256k1_mpt.h) describes
the encryption function as “Encrypts a 64-bit amount” without mentioning the practical decryption
constraint. Exploit Scenario An account starts with 500,000 tokens. After receiving multiple
confidential transfers totalling 600,000 additional tokens, the resulting encrypted balance
represents 1,100,000 tokens. This ciphertext is mathematically valid but becomes permanently
undecryptable

```

HashEye 14 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

using the provided API; there was no indication at encryption time that this threshold would be exceeded. Recommendations Short term, update the API documentation in include/secp256k1_mpt.h to explicitly state the supported range. Long term, consider whether the 1,000,000 limit aligns with the expected use cases for confidential assets. If higher values are required, either increase the search range (with corresponding performance analysis) or implement more efficient discrete logarithm algorithms such as baby-step giant-step (complexity) or Pollard’s kangaroo method. ☐ () Document the performance characteristics and expected value ranges in the specification.

HashEye 15 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

2. Deprecated OpenSSL SHA-256 API and unchecked return values Severity: Informational Difficulty: Low Type: Patching Finding ID: TOB-RIPCTX-2 Target: mpt-crypto/src/commitments.c, mpt-crypto/src/bulletproof_aggregated.c, mpt-crypto/src/equality_proof.c, mpt-crypto/src/proof_same_plaintext.c, mpt-crypto/src/proof_same_plaintext_multi.c, mpt-crypto/src/proof_same_plaintext_multi_shared_r.c, mpt-crypto/src/proof_link.c, mpt-crypto/src/proof_pok_sk.c

Description The mpt-crypto codebase uses OpenSSL’s deprecated SHA-256 API functions and fails to validate their return values. The functions SHA256_Init, SHA256_Update, and SHA256_Final were deprecated in OpenSSL 3.0 and are marked with the OSSL_DEPRECATEDIN_3_0 attribute. Each function returns an integer status code (1 for success, 0 for failure), but the codebase ignores these return values throughout all cryptographic operations. The deprecated SHA-256 functions are used extensively for cryptographic operations including NUMS generator derivation, Fiat-Shamir challenge computation, and transcript binding. Figure 2.1 shows a representative example from the NUMS generator derivation code. SHA256_CTX sha; SHA256_Init(&sha); SHA256_Update(&sha, "MPT_BULLETPROOF_V1_NUMS", 23); // Domain Sep SHA256_Update(&sha, "secp256k1", 9); // Curve Label

```

if (label && label_len > 0) { SHA256_Update(&sha, label, label_len); }

```

SHA256_Update(&sha, idx_be, 4); SHA256_Update(&sha, ctr_be, 4); SHA256_Final(hash, &sha); Figure 2.1: Deprecated SHA-256 API usage without error checking (mpt-crypto/src/commitments.c#L71-L82)

HashEye 16 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

First, while these functions continue to work in OpenSSL 3.x for backward compatibility, they may not be supported with future versions of OpenSSL and they generate compiler deprecation warnings that clutter build output and may obscure other compiler diagnostics. Second, the lack of return value validation violates defensive programming principles, as potential failures could silently corrupt cryptographic operations without detection. The issue affects eight source files responsible for core cryptographic operations: NUMS generator derivation, Bulletproof range proof generation and verification, equality proofs, same-plaintext proofs, linkage proofs, and proof-of-knowledge protocols. Additionally, test code in tests/test_ipa.c uses the same deprecated API pattern. Recommendations Short term, add return value validation to all SHA-256 API calls. Wrap

each function invocation with an error check that returns failure status to the caller. This pattern should be applied consistently across all eight affected source files. For example adapt the code pattern shown in figure 2.1 to validate each operation: if (SHA256_Init(&sha) $\neq$ 1) return 0; if (SHA256_Update(&sha, "MPT_BULLETPROOF_V1_NUMS", 23) $\neq$ 1) return 0; if (SHA256_Update(&sha, "secp256k1", 9) $\neq$ 1) return 0; Figure 2.2: Proper error checking of SHA-256 API calls It may be possible to roll this pattern into a macro to maintain readability. Long term, consider migrating to OpenSSL's EVP digest API. Plan the migration carefully with comprehensive performance benchmarking to ensure cryptographic operations maintain acceptable performance characteristics. Alternatively, consider dropping the OpenSSL dependency altogether, since only the SHA-256 implementation and the OPENSSL_cleanse function are currently used by the mpt-crypto library.

HashEye 17 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

3. Fiat-Shamir domain tag considerations Severity: Informational Difficulty: High Type: Cryptography Finding ID: TOB-RIPCTX-3 Target: mpt-crypto/src/proof_same_plaintext.c, mpt-crypto/src/proof_same_plaintext_multi.c, mpt-crypto/src/proof_link.c

Description Two distinct (minor) cryptographic issues affect Fiat-Shamir domain separation in the zero-knowledge proof implementations. First, as shown in figures 3.1 and 3.2, two different proof systems share the same domain tag, "MPT_POK_SAME_PLAINTEXT_PROOF". The build_same_plaintext_hash_input function in proof_same_plaintext.c (proving two ciphertexts encrypt the same plaintext) and the compute_challenge_multi function in proof_same_plaintext_multi.c (proving N ciphertexts encrypt the same plaintext) use identical domain tags despite implementing structurally different protocols from different sections of the specification (ConfidentialMPT_20260106.pdf sections 3.3.3 and 3.3.4, respectively). While transcript structures differ in input ordering and length, cryptographic best practice requires unique domain tags for each proof system. const char* domain = "MPT_POK_SAME_PLAINTEXT_PROOF"; Figure 3.1: Domain tag in 1-to-1 proof (mpt-crypto/src/proof_same_plaintext.c#L84) const char* domain = "MPT_POK_SAME_PLAINTEXT_PROOF"; Figure 3.2: Identical domain tag in 1-to-N proof (mpt-crypto/src/proof_same_plaintext_multi.c#L84) Second, as shown in figure 3.3, the ElGamal-Pedersen link proof in proof_link.c uses the generic domain tag "MPT_ELGAMAL_PEDERSEN_LINK", but the specification (ConfidentialMPT_20260201.pdf section 3.3.5) defines two distinct protocol variants:

- Variant A (Encryption-Randomness-Based Linkage), which should use "MPT_ELGAMAL_PEDERSEN_LINK_R"
- Variant B (Secret-Key-Based Linkage), which should use "MPT_ELGAMAL_PEDERSEN_LINK_SK"

HashEye 18 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

The implementation does not differentiate between these variants at the domain separation level, potentially allowing proof substitution between variants if both are later implemented. const char* domain = "MPT_ELGAMAL_PEDERSEN_LINK"; Figure 3.3: Generic domain tag without variant differentiation (mpt-crypto/src/proof_link.c#L76) Recommendations Short term, assign unique domain tags to differentiate proof systems and variants, such as "MPT_POK_SAME_PLAINTEXT_PROOF_PAIR" for 1-to-1 proofs and "MPT_POK_SAME_PLAINTEXT_PROOF_MULTI" for 1-to-N proofs, and update the link proof to use "MPT_ELGAMAL_PEDERSEN_LINK_R" to match the specification designation. Long term, establish a systematic naming convention for all Fiat-Shamir domain tags that encodes the proof type, statement count, and protocol variant, and enforce this convention through automated checks to prevent future collisions as new proof variants are added.

HashEye 19 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

4. Timing side channels in ElGamal decryption Severity: Informational Difficulty: Medium Type: Cryptography Finding ID: TOB-RIPCTX-4 Target: mpt-crypto/src/elgamal.c

Description The ElGamal decryption implementation contains two timing side channels that leak information about plaintext amounts through execution time variations. Both vulnerabilities arise from early-exit conditional branches in secp256k1_elgamal_decrypt that depend on secret plaintext values. Zero-Amount Early Exit The decryption function contains a fast-path check for zero amounts that returns immediately: /* 2. Check for Amount = 0 (C2 == S) */ /* This is much faster than doing point subtraction first */ if (pubkey_equal(ctx, c2, &S)) { *amount = 0; return 1; } Figure 4.1:

Early exit on zero amount (mpt-crypto/src/elgamal.c#L127-L132) When the plaintext is zero, decryption completes after a single point comparison, skipping the expensive brute-force search. This creates a measurable timing difference between zero and nonzero plaintexts. Variable-Time Search Loop The decryption function performs a brute-force search that terminates upon finding the plaintext: for (i = 1; i ≤ 1000000; ++i) { // Fast comparison if (pubkey_equal(ctx, ¤t_M, &M_target)) { *amount = i; return 1; } Figure 4.2: Early exit in discrete logarithm computation (mpt-crypto/src/elgamal.c#L146-L151)

HashEye 20 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Decryption time is directly proportional to the plaintext value. An amount of 1 executes in around 1 iteration, while 1,000,000 requires the full loop. Combined with the zero-amount fast path, an attacker can measure decryption timing to determine the approximate plaintext value: near-instant for zero, very fast for small amounts, increasingly slower for larger amounts. Recommendations Short term, eliminate timing-dependent branches in the decryption function by executing constant-time code paths. Long term, consider whether decryption timing is operationally sensitive in the deployment model. If decryption occurs only in trusted environments where timing attacks are impractical, document this assumption and the associated security boundaries. If decryption occurs in shared or multitenant environments, constant-time implementation becomes critical. Additionally, consider alternative discrete-logarithm-problem algorithms (baby-step giant-step, Pollard's kangaroo) that might offer better performance characteristics while maintaining constant-time properties.

HashEye 21 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

5. Timing side channels in Bulletproof generation Severity: Undetermined Difficulty: Undetermined Type: Cryptography Finding ID: TOB-RIPCTX-5 Target: mpt-crypto/src/bulletproof_aggregated.c

Description Several portions of the proving infrastructure introduce timing side channels that could reveal sensitive information. First, the secp256k1_bulletproof_ipa_msm function, which is incorporated into the msm_try_add function, performs a multiscalar multiplication but uses the timing-unsafe memcmp function and skips a point multiplication and addition whenever the comparison returns zero. The function includes a disclaimer in the documentation: NOTE: This MSM is only used for Bulletproofs where all scalars are public. It is NOT constant-time with respect to the scalars and MUST NOT be used for secret-key operations. for (size_t i = 0; i < n; ++i) { if (memcmp(scalars + i * 32, zero, 32) == 0) continue;

```
secp256k1_pubkey term = points[i]; if (!secp256k1_ec_pubkey_tweak_mul(ctx, &term, scalars + i * 32)) return 0;
```

```
if (!add_term(ctx, &acc, &initialized, &term)) return 0; }
```

```
/* All scalars zero → result is infinity (not representable here) */ if (!initialized) return 0;
```

Figure 5.1: Early exit for zero-value vectors with variable-time memcmp call (mpt-crypto/src/bulletproof_aggregated.c#L202) However, this code is used in the Bulletproof prover, via the msm_try_add function, under the secp256k1_bulletproof_run_ipa_prover function that calls secp256k1_bulletproof_ipa_compute_LR, which handles private information (specifically, the values being proven):

HashEye 22 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

```
/* Try adding terms. correct logic updates acc_initd to 1 */ if (!msm_try_add(ctx, &acc, &acc_initd, G_R, a_L, half_n)) goto cleanup; if (!msm_try_add(ctx, &acc, &acc_initd, H_L, b_R, half_n)) goto cleanup; up; Figure 5.2: Calls to variable-time msm_try_add in secp256k1_bulletproof_ipa_compute_LR ( mpt-crypto/src/bulletproof_aggregated.c#L541 ) The conditional additions and multiplications leak information about the Hamming weight of the number whose range is being proven, narrowing the space of possible values. Second, the secp256k1_bulletproof_ipa_compute_LR function also conditionally skips elliptic curve operations based on the results of comparison via memcmp. This is in addition to the calls to msm_try_add. The scalar_is_zero function also relies on memcmp. Because memcmp will return when the first byte
```


PUBLIC Cryptographic Review

Figure 7.1: Unbounded memory allocation based on unvalidated participant count (mpt-crypto/src/proof_same_plaintext_multi_shared_r.c#L255-L257) An attacker can provide an extremely large value for n, causing memory exhaustion. While the code checks for allocation failure, the attempt itself consumes system resources and can cause memory exhaustion, potentially leading to process termination on resource-constrained systems. Issue 2: Lack of Integer Overflow Protection in Size Calculation The function secp256k1_mpt_proof_equality_shared_r_size computes proof size without overflow protection: `size_t secp256k1_mpt_proof_equality_shared_r_size(size_t n_recipients) { // Tr (33) + N * Tm_i (33*N) + sm (32) + sr (32) return (33 * (n_recipients + 1)) + 64; }` Figure 7.2: Size calculation vulnerable to integer overflow (mpt-crypto/src/proof_same_plaintext_multi_shared_r.c#L61-L64) For large values of n_recipients, the multiplication $33 * (n_recipients + 1)$ can overflow, wrapping the result around to a small value. Issue 3: Delayed Length Validation The verification function secp256k1_mpt_verify_equality_shared_r (the example function with the unbounded memory allocation for issue 1) also includes a “strict” length check to ensure the entire proof buffer was consumed. However, this check occurs after all parsing and cryptographic operations complete: `// Strict Length Check: Ensure we read exactly what was expected if ((size_t)(ptr - proof) != expected_len) goto cleanup;` Figure 7.3: Length validation performed after all cryptographic operations (mpt-crypto/src/proof_same_plaintext_multi_shared_r.c#L324-L325) If the proof buffer is malformed or truncated, the function performs extensive point operations and challenge computations before detecting the issue. This wastes computational resources and delays error detection. The function should perform the length check immediately after parsing completes in order to fail fast on invalid inputs. Issue 4: Missing proof_len Validation Functions for plaintext equality with shared randomness (in proof_same_plaintext_multi_shared_r.c) are missing validation of the proof_len parameter, which is properly implemented in the equivalent functions for plaintext equality without shared randomness (in proof_same_plaintext_multi.c).

HashEye 26 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Specifically, the non-shared-randomness secp256k1_mpt_prove_same_plaintext_multi function properly validates the proof buffer size before any allocation or computation: `size_t required_len = secp256k1_mpt_prove_same_plaintext_multi_size(n); if (!proof_len || *proof_len < required_len) { if (proof_len) *proof_len = required_len; return 0; } *proof_len = required_len;` Figure 7.4: Proper prover validation in non-shared-randomness variant (mpt-crypto/src/proof_same_plaintext_multi.c#L147-L152) And the secp256k1_mpt_verify_same_plaintext_multi function validates the proof length at function entry: `if (proof_len != secp256k1_mpt_prove_same_plaintext_multi_size(n)) return 0;` Figure 7.5: Proper verifier validation in non-shared-randomness variant (mpt-crypto/src/proof_same_plaintext_multi.c#L260-L260) However, the equivalent shared-randomness variants of these functions are missing this validation. Note that the non-shared-randomness function secp256k1_mpt_prove_same_plaintext_multi_size in src/proof_same_plaintext_multi.c also fails to check for overflow. `size_t secp256k1_mpt_prove_same_plaintext_multi_size(size_t n) { // (1 Tm + 2N Tr) * 33 + (1 sm + N sr) * 32 return ((1 + 2 * n) * 33) + ((1 + n) * 32); }` Figure 7.6: mpt-crypto/src/proof_same_plaintext_multi.c#L130-L133 Exploit Scenario An attacker provides $n_recipients = SIZE_MAX / 33 - 1$. The size calculation overflows, returning a small value (e.g., 31 bytes instead of the correct multi-gigabyte size). A caller allocating a 31-byte buffer then invokes the proof generation function with $N = (SIZE_MAX / 33 - 1)$ recipients. The function writes 33 bytes for Tr, followed by thousands of 33-byte Tm_i commitments, causing a buffer overflow and potential code execution. Recommendations Short term, to address issue 1, add explicit validation of the participant count at function entry, and define a maximum reasonable value (e.g., 1,000 recipients) based on expected use cases. To address issue 2, add integer overflow protection to

HashEye 27 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

secp256k1_mpt_proof_equality_shared_r_size. To address issue 3, move the strict length check immediately after parsing completes so that the function fails fast. To address issue 4, update the affected functions to accept a proof_len parameter (like secp256k1_mpt_prove_same_plaintext_multi

and `secp256k1_mpt_verify_same_plaintext_multi`) and ensure the proof argument is consistent with that length. Long term, ensure validation patterns are consistent across all functions and audit all other proof functions for similar issues. Write tests covering these edge cases. Consider using overflow-safe arithmetic functions or `calloc` with explicit overflow checking. Document the maximum supported participant count in the API specification so callers can validate inputs before invoking proof functions.

HashEye 28 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

8. Missing return value checks Severity: Informational Difficulty: Low Type: Error Reporting Finding ID: TOB-RIPCTX-8 Target: `mpt-crypto/src/bulletproof_aggregated.c`, `mpt-crypto/src/equality_proof.c`, `mpt-crypto/src/proof_link.c`, `mpt-crypto/src/proof_pok_sk.c`, `src/proof_same_plaintext.c`

Description Multiple areas of the codebase are missing validation of the output length parameter after calls to `secp256k1_ec_pubkey_serialize`. While `secp256k1_ec_pubkey_serialize` always returns 1 and cannot fail, it modifies the `outputlen` parameter to indicate the actual number of bytes written. The code assumes this parameter will always be 33 bytes for compressed serialization but never verifies this assumption. If the serialization produces a different size than expected (though this should not occur with valid compressed points), the code would proceed to serialize the wrong number of bytes. The affected code appears in multiple functions where public keys are serialized and hashed to derive cryptographic challenges. For example, in the Bulletproof aggregated proof generation, the code does not check the `create_commitment` return value or verify the serialization length: `for (size_t i = 0; i < m; i++) { /* ... (re-add commitments) ... */ secp256k1_pubkey V_temp; unsigned char V_ser[33]; size_t v_len = 33; secp256k1_bulletproof_create_commitment(ctx, &V_temp, values[i], blindings_flat + 32*i, pk_base); secp256k1_ec_pubkey_serialize(ctx, V_ser, &v_len, &V_temp, SECP256K1_EC_COMPRESSED); SHA256_Update(&sha, V_ser, 33); }` Figure 8.1: Unchecked commitment creation and serialization length ( `mpt-crypto/src/bulletproof_aggregated.c#L1281-L1289`) While compressed public keys should always serialize to exactly 33 bytes with the supported curve, this invariant should be verified, according to defensive programming practices.

HashEye 29 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Additionally, in a few (presumably inconsequential) instances, including one in the code snippet above, the return value of `secp256k1_bulletproof_create_commitment` is not checked before the resulting commitment point is used. Recommendations Short term, add checks after each `secp256k1_ec_pubkey_serialize` call to ensure that the `outputlen` parameter equals 33 after serialization. Additionally, add checks to verify that the return value of any preceding `secp256k1_bulletproof_create_commitment` call is 1, indicating successful commitment creation. Long term, audit the codebase to identify other instances where input/output parameters are assumed to have specific values without verification. Consider creating wrapper functions or macros that combine serialization with automatic length validation to prevent future occurrences of this pattern. Implement comprehensive unit tests that deliberately violate expected invariants (such as using mock implementations that return unexpected lengths) to verify proper error detection and propagation.

HashEye 30 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

9. Insufficient compiler warning flags and security hardening options Severity: Informational Difficulty: Low Type: Configuration Finding ID: TOB-RIPCTX-9 Target: `mpt-crypto/CMakeLists.txt`

Description The `mpt-crypto` library is compiled with minimal compiler warning flags and lacks common security hardening options. The default build configuration uses only architecture-specific flags (`-arch arm64`) without enabling any warning diagnostics or runtime security protections. This reduces the ability to detect potential bugs at compile time and leaves the code vulnerable to certain classes of runtime attacks. Current Build Configuration The project's `CMakeLists.txt` does not specify any compiler warning flags or hardening options: `cmake_minimum_required(VERSION 3.10) project(mpt-crypto C CXX)`

--- Find Dependencies --- find_package(OpenSSL REQUIRED) Figure 9.1: Original CMakeLists.txt without warning or hardening flags (mpt-crypto/CMakeLists.txt) This results in compilation with only default flags; no build type is specified, defaulting to an unoptimized configuration without debug symbols or optimization. In comparison, the libsecp256k1 dependency uses -Wall -Wextra -Wpedantic -Wcast-align -Wconditional-uninitialized -Wshadow -Wstrict-prototypes, among others. The lack of warning flags means that code quality issues, potential bugs, and security vulnerabilities may go undetected during development. Impact of Adding Basic Warning Flags When the project is recompiled with -Wall -Wextra -Wpedantic, several code quality issues are revealed: • Library source code issues:

HashEye 31 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

◦ Unused parameter ctx (four instances) in src/bulletproof_aggregated.c: Line 145: unused parameter 'ctx' [-Wunused-parameter] Line 245: unused parameter 'ctx' [-Wunused-parameter] Line 255: unused parameter 'ctx' [-Wunused-parameter] Line 267: unused parameter 'ctx' [-Wunused-parameter] Figure 9.2: Unused ctx parameter These functions accept a secp256k1_context* ctx parameter but never use it. ◦ Unused variable sha (one instance) in src/elgamal.c: SHA256_CTX sha; Figure 9.3: Unused sha variable (mpt-crypto/src/elgamal.c) This is dead code that may have been left behind during refactoring and should be removed. ◦ Set but unused variable ok (one instance) in src/bulletproof_aggregated.c: int ok = 0 ; Figure 9.4: Set but unused variable ok (mpt-crypto/src/bulletproof_aggregated.c) The variable ok is declared, but the function secp256k1_bulletproof_compute_vectors_block returns explicit values 0 or 1. • Test code issues (more than 30 warnings): The test suite also contains a number of constructs highlighted by the above flags, such as missing function prototypes (30 instances), ignored return values (five instances), and variable length array warnings (three instances). Recommendations Short term, enable basic warning flags to improve code quality and catch common bugs, and address the issues documented in this report. # --- Compiler Warning Flags --- add_compile_options(-Wall -Wextra -Wpedantic) Figure 9.5: Suggested flags to add

HashEye 32 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Long term, consider enabling comprehensive security hardening for production builds: • Enable comprehensive warning flags (-Wall -Wextra -Wpedantic -Wformat=2 -Wformat-security -Wnull-dereference) • Add stack protection (-fstack-protector-strong) to detect buffer overflows at runtime • Enable fortified source (_FORTIFY_SOURCE=2) for bounds checking on memory operations • Add linker hardening flags on Linux • Migrate from the deprecated OpenSSL SHA-256 API to the EVP API

HashEye 33 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

10. Missing range proof enables confidential balance overdraft Severity: High Difficulty: Low Type: Cryptography Finding ID: TOB-RIPCTX-10 Target:
rippled/src/xrpld/app/tx/detail/ConfidentialConvertBack.cpp,
rippled/src/xrpld/app/tx/detail/ConfidentialSend.cpp

Description ConfidentialConvertBack and ConfidentialSend do not prove that a holder's spending balance is greater than or equal to the claimed transaction amount. A holder can claim to convert back or send more tokens than their confidential balance holds, receiving plaintext tokens that were not backed by a corresponding confidential balance or inflating a recipient's inbox, effectively creating tokens from nothing. The proof system for ConfidentialConvertBack verifies two properties. First, verifyRevealedAmount confirms that the submitted ciphertexts correctly encrypt the claimed plaintext amount under the respective public keys using the disclosed blinding factor. Second, verifyBalancePcmLinkage confirms that the on-chain spending balance sfConfidentialBalanceSpending and the submitted sfBalanceCommitment encode the same value. Neither proof establishes any relationship between the balance value and the claimed amount. The sole guard against overdraft in preclaim checks the issuance-wide total outstanding confidential supply, not the individual holder's balance: // if the total circulating confidential balance is smaller than what the // holder is trying to convert back, we know for sure this txn should // fail if


```
((*sfIssuance)[~sfConfidentialOutstandingAmount].value_or(0) < amount) { return
tecINSUFFICIENT_FUNDS; } Figure 10.1: Global outstanding supply check in
ConfidentialConvertBack::preclaim (
rippled/src/xrpld/app/tx/detail/ConfidentialConvertBack.cpp#L156-L162 ) This check limits any
single convert-back transaction to the total circulating confidential supply, but places no
restriction on how much of that supply a single holder can claim relative to their actual balance.
A holder with a spending balance of one token can claim the entire issuance-wide confidential
supply, draining tokens that belong to every other holder.
```

HashEye 34 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

The same gap exists in ConfidentialSend, where the codebase explicitly acknowledges the missing control with a developer note: `// todo: Extract range proof once the lib is ready if (remainingLength  $\neq$  0) return tecINTERNAL; // LCOV_EXCL_LINE` Figure 10.2: Acknowledged missing range proof in ConfidentialSend::verifySendProofs (rippled/src/xrpld/app/tx/detail/ConfidentialSend.cpp#L118-L120) In this case, the impact is that a recipient can inflate their confidential inbox, growing the effective token supply beyond what was originally converted in. Exploit Scenario A holder with a confidential spending balance of 10 tokens constructs a ConfidentialConvertBack transaction claiming an sfMPTAmount of 5,000. They produce a valid sfHolderEncryptedAmount that correctly encrypts 5,000 under their public key with the disclosed blinding factor, satisfying verifyRevealedAmount. They then provide an sfBalanceCommitment committing to their real balance of 10, and generate a valid verifyBalancePcmLinkage proof binding that commitment to their on-chain spending ciphertext. Both proofs pass. The holder then receives 5,000 plaintext MPT tokens, and their sfConfidentialBalanceSpending is set to an elliptic curve point representing -4,990, corrupting their account state for all future transactions. The other holders of that issuance have collectively lost 5,000 tokens' worth of backing. Recommendations Short term, ensure confidential transactions are disabled until range proofs are integrated. Both transaction types (i.e., ConfidentialConvertBack and ConfidentialSend) require a nonnegativity proof of the form $\text{balance} - \text{amount} \geq 0$ to be sound, and no interim substitute exists because the individual holder balance is encrypted and cannot be compared in plaintext at the protocol level. Long term, integrate the range proof component implemented in mpt-crypto, and add comprehensive tests that attempt to submit convert-back or send transactions where the claimed amount exceeds the holder's actual spending balance, verifying the transaction is rejected.

HashEye 35 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

11. ConfidentialClawback resets the balance version counter Severity: Medium Difficulty: Low Type: Data Validation Finding ID: TOB-RIPCTX-11 Target:

```
rippled/src/xrpld/app/tx/detail/ConfidentialClawback.cpp,
rippled/include/xrpl/protocol/ConfidentialTransfer.h
```

Description ConfidentialClawback::doApply hard-codes sfConfidentialBalanceVersion to zero instead of incrementing it, deviating from the protocol specification and from the behavior of every other transaction that modifies the version counter. Section 6.6 of the specification defines the state transition for the ConfidentialClawback transaction as $\text{CB_S_Version} \leftarrow \text{CB_S_Version} + 1$. However, ConfidentialClawback does not follow the specification and assigns zero directly:

```
(*sfHolderMPToken)[sfConfidentialBalanceVersion] = 0; Figure 11.1: ConfidentialClawback hard-codes
zero instead of calling incrementConfidentialVersion. (
```

```
rippled/src/xrpld/app/tx/detail/ConfidentialClawback.cpp#L132 ) sfConfidentialBalanceVersion is
embedded in the context that zero-knowledge proof transcripts for confidential transactions are
bound to. A proof generated against version V is cryptographically invalid at any other version.
This lack of counter advancement is a deviation from the specification's replay-protection model
and may also lead to interoperability issues. Exploit Scenario A holder's CB_S_Version reaches V
through a series of confidential transactions. The issuer submits a ConfidentialClawback
transaction, which resets the on-ledger version to zero. A specification-compliant wallet observes
the clawback event and updates its local version record to V+1. When the holder subsequently
converts new tokens into confidential form and attempts to generate a ConfidentialConvertBack
proof, the wallet embeds version V+1 in the TransactionContextID. The ledger, however, holds
version zero, so the verifier computes a different hash and returns tecBAD_PROOF. The holder's
```


attempts to exit the confidential system fail silently, with no error message indicating that the version counter is the source of the mismatch. The holder is effectively locked.

HashEye 36 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Recommendations Short term, replace the direct assignment in ConfidentialClawback::doApply with a call to incrementConfidentialVersion, consistent with every other transaction that advances the counter. Long term, add a test asserting that sfConfidentialBalanceVersion equals V+1 after a clawback against an account at version V, for multiple values of V including zero. This will detect regressions if introduced and document the intended behavior in an executable form. Implement additional comprehensive tests exercising other edge cases.

HashEye 37 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

12. ConfidentialSend increments the receiver's version counter Severity: High Difficulty: Low Type: Cryptography Finding ID: TOB-RIPCTX-12 Target: rippled/src/xrpld/app/tx/detail/ConfidentialSend.cpp

Description ConfidentialSend::doApply increments sfConfidentialBalanceVersion on both the sender's and the receiver's MPToken objects. Section 6.3 of the specification defines the state transition for a send transaction as incrementing only CB_S_Version(A) (the sender's version); the receiver's version counter should not be modified. This deviation means any account can unilaterally invalidate precomputed proofs belonging to any other account by sending them a dust amount. The version counter is a replay-protection mechanism embedded in the TransactionContextID of every zero-knowledge proof. Before a holder submits a ConfidentialSend or ConfidentialConvertBack transaction, they read their on-ledger sfConfidentialBalanceVersion, include it in the Fiat-Shamir transcript, and sign the resulting proof. If the on-ledger counter advances between proof generation and submission, the verifier rejects the proof with tecBAD_PROOF. Under the specification, only the holder's own spending actions advance their counter, so a pregenerated proof remains valid until the holder acts. Under the implementation, every incoming send also advances the counter, so an unsolicited transfer can silently invalidate any outstanding proof. Figure 12.1 shows the relevant lines of ConfidentialSend::doApply. // increment version incrementConfidentialVersion(*sleSenderMPToken); incrementConfidentialVersion(*sleDestinationMPToken); Figure 12.1: Both sender and receiver version counters incremented in doApply (rippled/src/xrpld/app/tx/detail/ConfidentialSend.cpp#L406-L408) Figure 12.2 shows an excerpt of section 6.3 of the specification. Sender A: ... CB_S_Version(A) ← CB_S_Version(A) + 1 ...

Receiver B:

HashEye 38 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

CBIN(B) ← CBIN(B) ⊕ EncB(m) EncI(B) ← EncI(B) ⊕ EncI(m) Figure 12.2: Excerpt of section 6.3 of the specification No increment to CB_S_Version(B) appears in the receiver's transition. Exploit Scenario Alice pregenerates a ConfidentialConvertBack proof against her current sfConfidentialBalanceVersion, which is set to 5. Before she submits the transaction, Bob sends her one "dust amount" of the confidential token. doApply increments Alice's counter to 6. When Alice broadcasts her presigned proof, the verifier reconstructs the TransactionContextID with version 6 from the ledger, while the proof encodes version 5; the Fiat-Shamir challenge does not match, and the node rejects the transaction with tecBAD_PROOF. Recommendations Short term, remove the incrementConfidentialVersion(*sleDestinationMPToken) call from ConfidentialSend::doApply. Long term, add an automated conformance test that replays each transaction type and asserts the exact set of sfConfidentialBalanceVersion fields that change, comparing the result against the specification's state-transition table. Additionally, implement processes such that any implementation change is checked against the specification and vice versa.

HashEye 39 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

13. MergeInbox allows version counter increment without balance change Severity: Informational
Difficulty: High Type: Data Validation Finding ID: TOB-RIPCTX-13 Target:
rippled/src/xrpld/app/tx/detail/ConfidentialMergeInbox.cpp

Description ConfidentialMergeInbox does not require that the inbox balance (sfConfidentialBalanceInbox) encodes a nonzero amount before executing. The preclaim stage verifies only that the field is present, without checking whether the encrypted value represents a non-trivial balance: if (!sleMptoken->isFieldPresent(sfConfidentialBalanceInbox) || !sleMptoken->isFieldPresent(sfConfidentialBalanceSpending) || !sleMptoken->isFieldPresent(sfHolderElGamalPublicKey)) return tecNO_PERMISSION; Figure 13.1: Preclaim stage checks only field presence, not inbox balance value. (rippled/src/xrpld/app/tx/detail/ConfidentialMergeInbox.cpp#L46-L49) As a result, a holder can submit a MergeInbox transaction when sfConfidentialBalanceInbox already holds the canonical encryption of zero, causing doApply to perform a no-op merge and still increment sfConfidentialBalanceVersion: // it's fine if it reaches max uint32, it just resets to 0 (*sleMptoken)[sfConfidentialBalanceVersion] = (*sleMptoken)[~sfConfidentialBalanceVersion].value_or(0u) + 1u; Figure 13.2: Version counter is incremented unconditionally at the end of doApply. (rippled/src/xrpld/app/tx/detail/ConfidentialMergeInbox.cpp#L89-L91) sfConfidentialBalanceVersion is embedded in the TransactionContextID that all zero-knowledge proof transcripts for ConfidentialSend and ConfidentialConvertBack are bound to. Any proof generated against a given version is cryptographically invalid for any subsequent version. A holder who generates a proof for version V and then submits a no-op MergeInbox will find that proof rejected on-chain, because the ledger now records version V+1.

HashEye 40 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Recommendations Short term, document the behavior of MergeInbox and of sfConfidentialBalanceVersion as a proof-invalidation mechanism. Long term, consider adding a validation check in ConfidentialMergeInbox::preclaim that verifies that sfConfidentialBalanceInbox is not equal to the canonical encryption of zero for the account and issuance before allowing the transaction to proceed. Add unit tests covering this and similar edge cases.

HashEye 41 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

14. Discrepancies between specification fields and implementation Severity: Informational
Difficulty: Low Type: Data Validation Finding ID: TOB-RIPCTX-14 Target:
rippled/include/xrpl/protocol/detail/transactions.h, macro,
rippled/src/xrpld/app/tx/detail/ConfidentialSend.cpp,
rippled/src/xrpld/app/tx/detail/ConfidentialConvertBack.cpp

Description The specification defines CB_S_Version (sfConfidentialBalanceVersion) as a mandatory transaction field in both ConfidentialSend and ConfidentialConvertBack, but neither transaction includes it in its schema. Instead, the implementation reads the version directly from the sender's MPToken ledger object at proof-verification time. The field is absent from both format definitions: TRANSACTION(ttCONFIDENTIAL_SEND, 88, ConfidentialSend, Delegation::delegatable, featureConfidentialTransfer, noPriv, ({ {sfMPTokenIssuanceID, soeREQUIRED}, {sfDestination, soeREQUIRED}, {sfSenderEncryptedAmount, soeREQUIRED}, {sfDestinationEncryptedAmount, soeREQUIRED}, {sfIssuerEncryptedAmount, soeREQUIRED}, {sfAuditorEncryptedAmount, soeOPTIONAL}, {sfZKProof, soeREQUIRED}, {sfAmountCommitment, soeREQUIRED}, {sfBalanceCommitment, soeREQUIRED}, {sfCredentialIDs, soeOPTIONAL}, }) Figure 14.1: ttCONFIDENTIAL_SEND schema with no sfConfidentialBalanceVersion field (rippled/include/xrpl/protocol/detail/transactions.h#L1116-L1131) The same schema also includes sfCredentialIDs as an optional field, which has no counterpart in the specification. The specification does not define a credentials field for ConfidentialSend, and the handling of sfCredentialIDs in this transaction deviates from the pattern established by Payment: there is no checkExtraFeatures method to gate the field on featureCredentials, and ConfidentialSend::preclaim does not call credentials::valid() to verify credential ownership and acceptance before doApply runs.

HashEye 42 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Recommendations Short term, align the transaction schemas with the specification by adding `sfConfidentialBalanceVersion` as a required field to `ttCONFIDENTIAL_SEND` and `ttCONFIDENTIAL_CONVERT_BACK` (or removing it from the specification), and by removing `sfCredentialIDs` from `ttCONFIDENTIAL_SEND` (or completing its handling by adding `checkExtraFeatures` and a `credentials::valid()` preclaim call). Long term, maintain a field-level conformance test that asserts that the transaction schema matches the specification table for each confidential transaction type, so future changes to either do not silently diverge.

HashEye 43 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

15. Incorrect TransactionContextID for Convert and ConvertBack Severity: Medium Difficulty: Medium Type: Data Validation Finding ID: TOB-RIPCTX-15 Target: `rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp`

Description The specification's notation for the `TxSpecific` component of `TransactionContextID` does not match the implementation for two transaction types. The `TransactionContextID` is included in every zero-knowledge proof Fiat-Shamir transcript and must be reproduced exactly by any external client generating proofs; incorrect documentation will cause client-generated proofs to fail verification. Section 3.3.1 of the specification, "TransactionContextID (proof context binding)," on pages 7 and 8 states the following:

- Send/ConvertBack: `TxSpecific = Receiver ⊕ CB_S_Version`
- Convert: `TxSpecific = Account ⊕ 0`

The implementation diverges on two counts. For `Convert`, `getConvertContextHash` appends only `Amount` (the 64-bit plaintext value) as the transaction-specific field:

```
uint256 getConvertContextHash( AccountID const& account, std::uint32_t sequence, uint192 const& issuanceID, std::uint64_t amount) {
    Serializer s;
    addCommonZKPFields( s,
        ttCONFIDENTIAL_CONVERT, account, sequence, issuanceID);
```

```
    s.add64(amount);
```

```
    return s.getSHA512Half(); }
}
// Figure 15.1: getConvertContextHash uses Amount as TxSpecific, not Account ⊕ 0.
// ( rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L57-L71 )
// For ConvertBack, getConvertBackContextHash appends Amount ⊕ CB_S_Version:
```

HashEye 44 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

```
uint256 getConvertBackContextHash( AccountID const& account, std::uint32_t sequence, uint192 const& issuanceID, std::uint64_t amount, std::uint32_t version) {
    Serializer s;
    addCommonZKPFields( s,
        ttCONFIDENTIAL_CONVERT_BACK, account, sequence, issuanceID);
```

```
    s.add64(amount);
    s.addInteger(version);
```

```
    return s.getSHA512Half(); }
}
// Figure 15.2: getConvertBackContextHash uses Amount ⊕ CB_S_Version as TxSpecific, not Receiver ⊕ CB_S_Version.
// ( rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L73-L89 )
// In summary, the TxSpecific values, as observed in the implementation and in the specification, are as follows:
// Transaction TxSpecific (Implementation) TxSpecific (Specification)
// Convert Amount Account ⊕ 0 Send Destination ⊕ CB_S_Version Receiver ⊕ CB_S_Version
// ConvertBack Amount ⊕ CB_S_Version Receiver ⊕ CB_S_Version
// The Send entry is functionally correct (field naming aside); Convert and ConvertBack carry meaningful discrepancies.
// Recommendations Short term, make the necessary adjustments so that the specification and implementation match. Long term, add test vectors for each getXxxContextHash function that encode the exact byte layout of the serialized transcript, and include those vectors in the specification as normative examples.
```

HashEye 45 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

16. Multi-ciphertext equality proof uses per-recipient nonces instead of shared `r` Severity: Medium Difficulty: Low Type: Cryptography Finding ID: TOB-RIPCTX-16 Target:

rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp

Description The ConfidentialSend multi-ciphertext equality proof deviates from the protocol specification by generating an independent random nonce for each recipient instead of reusing a single shared randomness scalar across all ciphertexts. This structural departure produces proofs that are incompatible with any external implementation following the specification and increases on-chain proof size by a factor approaching two. The specification defines a shared-randomness Sigma protocol over N ciphertexts that all use the same ElGamal encryption scalar r . Because all ciphertexts share $C1 = r \cdot G$, a single response scalar $sr = k_r + e \cdot r$ is sufficient to bind the randomness proof across all recipients. The resulting proof has size $33(1+N) + 64$ bytes (196 bytes for $N=3$, 229 bytes for $N=4$, adding an auditor). The implementation instead generates N independent random scalars $k_{r[i]}$, one per recipient, each responding to its own $C1$ component $R[i]$.

```
secp256k1_mpt_prove_same_plaintext_multi_size(size_t n) { // (1 point T_m + 2*N points T_r) * 33 + (1 scalar s_m + N scalars s_r) * 32 return ((1 + 2 * n) * 33) + ((1 + n) * 32); }
```

Figure 16.1: Implementation of the proof size formula (rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L1723-L1728)

For $N=3$ recipients (sender, receiver, issuer), the implementation proof is 359 bytes, versus the specification's 196 bytes. For $N=4$ (adding an auditor), the sizes are 457 bytes versus the specification's 229 bytes. The per-recipient nonce generation is visible in the prove function, where independent $k_{r[i]}$ scalars and their corresponding commitments $T_{rG}[i] = k_{r[i]} \cdot G$ and $T_{rP}[i] = k_{r[i]} \cdot PK[i]$ are produced for each recipient. The verifier reflects this structure by checking a per-recipient $C1$ binding equation ($s_{r[i]} \cdot G = T_{rG}[i] + e \cdot R[i]$) for each i rather than a single shared- r equation ($sr \cdot G = Tr + e \cdot C1$).

HashEye 46 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

The proof is cryptographically sound for proving that all ciphertexts encrypt the same message m . However, the shared-randomness constraint (which is a deliberate design choice in the specification) is not enforced. The sender can supply ciphertexts with distinct $C1$ components under this construction, whereas the specification requires all $C1$ values to be identical. Because all provers and verifiers in the current codebase share this implementation, proofs validate internally; the impact is on cross-implementation compatibility and protocol conformance.

Exploit Scenario A developer building a specification-compliant wallet library implements the ConfidentialSend multi-ciphertext equality proof according to the specification's shared-randomness construction, producing a proof of 196 bytes for a three-recipient send operation. When the wallet submits a ConfidentialSend transaction, the on-chain preflight size check, which enforces exactly `getMultiCiphertextEqualityProofSize(nRecipients)` bytes, rejects the transaction because the expected proof size is 359 bytes, not 196. The transaction fails with no path to recovery.

Recommendations Short term, replace the per-recipient nonce construction with the shared-randomness scheme that is described in the specification and implemented in the mpt-crypto library. Update the proof serialization, the size formula, and the `getMultiCiphertextEqualityProofSize` helper accordingly. Long term, add test vectors that cross-validate against independently written proof implementations.

HashEye 47 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

17. Missing input length validation in low-level cryptographic helper functions Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-RIPCTX-17 Target: `src/libxrpl/protocol/ConfidentialTransfer.cpp`

Description Four helper functions in ConfidentialTransfer.cpp accept Slice parameters representing public keys or ciphertexts but do not validate their lengths before accessing the underlying data. Each function either constructs slices past the end of the buffer, performs an unchecked `std::memcpy` into a fixed-size struct, or passes an unvalidated slice to a secp256k1 API. The functions are currently called only from paths where preflight has already enforced field lengths, but the absence of local guards creates a latent out-of-bounds read vulnerability for any future caller that omits the upstream check.

`makeEcPair` does not check buffer length before constructing `s2`. `makeEcPair` parses a concatenated ciphertext ($C1 \parallel C2$) by splitting a flat buffer into two Slice objects at the fixed offset `ecGamalEncryptedLength`. No check is made that `buffer.length()` is greater than or equal to $2 * \text{ecGamalEncryptedLength}$ before the split: `Slice s1{buffer.data(), ecGamalEncryptedLength}; Slice s2{buffer.data() + ecGamalEncryptedLength, ecGamalEncryptedLength};`

Figure 17.1: `s2` constructed without verification that the buffer is large enough (

rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L102-L103) If buffer.length() is smaller than 2 * ecGamalEncryptedLength, s2.data() points past the end of the allocated region. encryptAmount does not check pubKeySlice length before the call to memcpy. encryptAmount copies the caller-supplied public key directly into a secp256k1_pubkey struct without first checking that pubKeySlice.size() is equal to ecPubKeyLength: std::memcpy(pubKey.data, pubKeySlice.data(), ecPubKeyLength); Figure 17.2: Unchecked pubKeySlice in encryptAmount (rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L226)

HashEye 48 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

If pubKeySlice.size() is smaller than ecPubKeyLength, memcpy reads beyond the slice's data pointer. By contrast, the sibling function verifyElGamalEncryption performs this check explicitly before its own call to memcpy. verifyElGamalEncryption does not check ciphertext length before calling makeEcPair. verifyElGamalEncryption validates blindingFactor and pubKeySlice lengths but not ciphertext: secp256k1_pubkey c1, c2; if (!makeEcPair(ciphertext, c1, c2)) return tecINTERNAL; // LCOV_EXCL_LINE Figure 17.3: ciphertext passed to makeEcPair without length check in verifyElGamalEncryption (rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L317-L319) An undersized ciphertext would propagate into makeEcPair and trigger the out-of-bounds slice construction described above. verifyClawbackEqualityProof checks neither ciphertext nor pubKeySlice length. verifyClawbackEqualityProof calls makeEcPair on ciphertext and then copies pubKeySlice into a secp256k1_pubkey struct, without checking either argument's length: secp256k1_pubkey c1, c2; if (!makeEcPair(ciphertext, c1, c2)) return tecINTERNAL; // LCOV_EXCL_LINE

secp256k1_pubkey pubKey; std::memcpy(pubKey.data, pubKeySlice.data(), ecPubKeyLength); Figure 17.4: Both ciphertext and pubKeySlice are unchecked in verifyClawbackEqualityProof. (rippled/src/libxrpl/protocol/ConfidentialTransfer.cpp#L435-L440) Exploit Scenario A future transaction type calls encryptAmount with a caller-provided public key slice without first enforcing the ecPubKeyLength precondition. Due to the unchecked public key length before the call to memcpy, ecPubKeyLength bytes are read starting from an attacker-controlled offset, forwarding stack or heap data beyond the slice boundary into the secp256k1 parser.

HashEye 49 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Recommendations Short term, add length assertions at the entry point of each affected function, matching the pattern already established in verifyElGamalEncryption: • makeEcPair: Return false if buffer.length() < 2 * ecGamalEncryptedLength • encryptAmount: Return std::nullopt if pubKeySlice.size() ≠ ecPubKeyLength • verifyElGamalEncryption: Return tecINTERNAL if ciphertext.size() ≠ ecGamalEncryptedTotalLength • verifyClawbackEqualityProof: Return tecINTERNAL if ciphertext.size() ≠ ecGamalEncryptedTotalLength, pubKeySlice.size() ≠ ecPubKeyLength, or proof.size() ≠ ecEqualityProofLength Long term, write comprehensive unit tests exercising these edge cases. Consider consolidating all low-level cryptographic helpers behind a single validated entry point (analogous to the existing checkEncryptedAmountFormat) that enforces all preconditions before dispatching to the secp256k1 API.

HashEye 50 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

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

HashEye 51 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

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 Not Applicable This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply. 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.

HashEye 52 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

B. Code Quality Issues This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- Use of magic numbers. In several areas throughout the codebase, unnamed and undocumented constant values are used. This is particularly prevalent in the Fiat-Shamir challenge generation code, with calls such as `SHA256_Update(&sha, last_challenge, 32)`. Frequently used constants like 32 or 33 should be given names (for instance, `PUBKEY_SERIALIZED_LEN`), with succinct documentation of their purpose and derivation. This will help reduce risks from typos and find-and-replace errors in the future.
- Unnecessary checks for calls to free. The `secp256k1_bulletproof_prove_agg` function includes several lines that conditionally call free if the relevant pointer is non-NULL. This is not necessary; calling free on a NULL pointer has well-defined behavior (it has no effect).
- Conditionals without brackets. Throughout the codebase, there are many conditional jumps to clean up code. The associated goto statements are not protected within blocks. This can lead to potential confusion, especially if the goto statements are on separate lines from the conditional statements that control their execution. This can lead to security failures, such as the Apple "goto fail" bug of 2014. We recommend placing all conditionals inside brackets to prevent potential problems.
- Empty cleanup labels in verification functions. Several verification functions use cleanup: labels that do not perform any cleanup operations (such as those in `equality_proof.c`, `proof_pok_sk.c`, and `proof_same_plaintext_multi.c`). While this is not a security issue since verification functions process public inputs that do not require secure erasure, the empty cleanup labels may be misleading. `ok = 1;`

```
cleanup: return ok;
```

Figure B.1: Empty cleanup label (`mpt-crypto/src/equality_proof.c#L251-L254`) In contrast, prover functions that handle secret values properly use cleanup labels to securely erase sensitive data with `OPENSSL_cleanse`, as can be seen in `proof_same_plaintext.c`. Either the verification functions could be updated to remove the cleanup labels entirely and use direct returns, or the labels could be renamed to something like `done:` to clarify that no cleanup is necessary.

HashEye 53 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

- Modular reduction obscured by NULL overflow parameter in scalar conversions. All calls to `secp256k1_scalar_set_b32` in `mpt_scalar.c` pass NULL as the overflow parameter, which prevents the caller from determining whether the input 32-byte array was reduced modulo the group order. This information could be useful for debugging or validation purposes.

```
void  
secp256k1_mpt_scalar_add(unsigned char *res, const unsigned char *a, const unsigned char *b) {  
    secp256k1_scalar s_res, s_a, s_b; secp256k1_scalar_set_b32(&s_a, a, NULL);  
    secp256k1_scalar_set_b32(&s_b, b, NULL); secp256k1_scalar_add(&s_res, &s_a, &s_b);  
    secp256k1_scalar_get_b32(res, &s_res);
```

Figure B.2: NULL passed as overflow parameter (`mpt-crypto/src/mpt_scalar.c#L60-L65`)

- Unchecked return value from scalar addition. The function `secp256k1_mpt_scalar_add` in `mpt_scalar.c` (excerpted in figure B.2) calls `secp256k1_scalar_add` but does not check its return value. The `secp256k1_scalar_add` function returns an integer indicating whether the addition overflowed the modular field (returning 1 if an overflow occurred and 0 otherwise).
- Unchecked size parameter with unsafe cast to smaller integer type. The function `secp256k1_mpt_get_generator_vector` in `commitments.c` accepts a `size_t n` parameter but immediately casts it to `uint32_t` without validation. On 64-bit platforms, this cast can silently truncate large values, causing the function to generate fewer generators than requested.

```
int secp256k1_mpt_get_generator_vector( const secp256k1_context* ctx, secp256k1_pubkey* vec, size_t n, const unsigned char* label, size_t label_len ) { for (uint32_t i = 0; i < (uint32_t)n; i++) { if (!secp256k1_mpt_hash_to_point_nums(ctx, &vec[i], label, label_len, i)) { return 0; } }
```

Figure B.3: Cast of `size_t` to `uint32_t` (`mpt-crypto/src/commitments.c#L124-L135`)

While it is unlikely that legitimate use cases require more than four billion generators, the silent truncation could lead to subtle bugs.

HashEye 54 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

- Missing zero check before scalar inversion. The function `secp256k1_mpt_scalar_inverse` in `mpt_scalar.c` does not check whether the input scalar is zero before invoking `secp256k1_scalar_inverse`. Zero has no multiplicative inverse in a finite field, and passing zero to the underlying function produces zero as the result rather than signaling an error. A caller relying on the output to perform a division-equivalent operation would silently receive an incorrect result.

```
void secp256k1_mpt_scalar_inverse(unsigned char *res, const unsigned char *in) { secp256k1_scalar s; secp256k1_scalar_set_b32(&s, in, NULL); secp256k1_scalar_inverse(&s, &s); secp256k1_scalar_get_b32(res, &s);
```

```
/* SECURE CLEANUP */ OPENSSL_cleanse(&s, sizeof(s)); }
```

Figure B.4: Inversion function allows inversion of zero (`mpt-crypto/src/mpt_scalar.c#L86-L94`)

The function should check whether the input is zero and return an error code before proceeding.

- Missing zeroization. In the function `secp256k1_mpt_prove_same_plaintext` in `src/proof_same_plaintext.c`, several intermediate sensitive variables—`s_m`, `s_r1`, `s_r2`, and `term`—are not zeroized, as shown in figure B.5. `cleanup: OPENSSL_cleanse(k_m, 32); OPENSSL_cleanse(k_r1, 32); OPENSSL_cleanse(k_r2, 32); OPENSSL_cleanse(m_scalar, 32); return ok;`

Figure B.5: Only a few variables are zeroized. (`mpt-crypto/src/proof_same_plaintext.c#L193-L198`)

- Undocumented optional context ID in challenge computation. The internal challenge hash computation functions (such as `build_link_challenge_hash` in `proof_link.c` and `build_same_plaintext_hash_input` in `proof_same_plaintext.c`) silently accept a `NULL` `context_id`, omitting it from the hash when absent. `if (context_id) { SHA256_Update(&sha, context_id, 32); }`

HashEye 55 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Figure B.6: Optional `context_id` (`mpt-crypto/src/proof_link.c#L98-L100`)

This optionality is not reflected in the public API signatures, which declare `context_id` as a plain `const unsigned char*` with no indication that `NULL` is a valid input or that it changes the challenge computation. This field is also crucial to the security of the higher-level confidential transaction protocol and should probably not be considered optional.

- Discrepancy between implementation and specification for zero-handling. The underlying function `secp256k1_ec_seckey_verify` rejects zero, but the specification permits it as randomness. For example, under the prover steps of section 3.3.4, “Proof of Equality of Plaintexts with Shared Randomness,” the random blinding scalar are sampled as the following:

```
0 0 , 0 0 ←$ 0 0
```

The above technically allows zero to be sampled, while the implementation rejects it. This small discrepancy does not have serious practical implications.

- Misleading constant name `ecGamalEncryptedLength`. The function `ecGamalEncryptedLength` defined in `Protocol.h` is defined as 33 bytes and documented as “EC ElGamal ciphertext length.” In reality, this is the length of a compressed secp256k1 point. An EC-ElGamal ciphertext is composed of a pair of such elliptic curve

points. The constant `ecGama1EncryptedLength`, therefore, names a compressed curve point, not a ciphertext, and is used throughout `ConfidentialTransfer.cpp` precisely in that role. A name such as `ecCompressedPointLength` would more accurately reflect what the constant measures and prevent future readers from conflating a single group element with a full ElGamal ciphertext.

- Missing zeroization of blinding factor in `generateBlindingFactor`. The function `generateBlindingFactor` allocates a stack buffer, fills it with a cryptographically random scalar via `RAND_bytes`, copies it into a heap-allocated `Buffer`, and returns. The stack buffer is never explicitly zeroed out before the function returns. The blinding factor is the ElGamal encryption randomness r ; disclosure of r allows anyone who knows it to decrypt any ciphertext produced with that randomness by computing $C2 - r \cdot pk = mG$ and solving the discrete logarithm for small m .

HashEye 56 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

C. Performance Improvement Opportunities This appendix discusses areas where we noted opportunities to improve efficiency and performance. These observations do not relate to security or maintainability, only to opportunities to make proving and verification run more quickly.

scalar_pow_u32 Issues The `scalar_pow_u32` function computes mod , where is the `secp256k1` curve order. It uses iterated multiplication, multiplying by itself times in a loop. Assuming a variable-time implementation is acceptable, a simple square-and-multiply algorithm uses about 48 operations per exponentiation on average. The `secp256k1_verify_bulletproof_agg` function calls `scalar_pow_u32` inside several loops. These loops compute successive powers of the same value; that is, they compute $\dots, 0 \cdot 0+1, 0 \cdot 0+2, \dots, 0 \cdot 0+0$ multiplications. If the power were computed iteratively in the outer loops, this would be reduced to somewhere between 64 and 70 multiplications (depending on the starting power). We recommend updating the loops to use successive multiplications inside each loop. If `scalar_pow_u32` is expected to be used elsewhere with values larger than 48, consider switching over to a square-and-multiply algorithm.

Recomputation of H Generator The function `secp256k1_mpt_get_h_generator` recomputes a constant on every call. The function derives the H generator by running a try-and-increment hash loop (in `commitments.c`), hashing the fixed inputs "MPT_BULLETPROOF_V1_NUMS", "secp256k1", "H", index 0, and a counter until a valid curve point is found. Because all inputs are constants, the resulting point is always the same. Therefore, the SHA-256 computation and curve point parsing are performed redundantly on every call, including every time a commitment or proof function is invoked. The resulting point could instead be computed once offline, hard-coded as a 33-byte compressed constant, and parsed at startup or lazily cached, eliminating the repeated work entirely.

HashEye 57 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

D. API Improvement Opportunities The `mpt-crypto` library implements cryptographically sound primitives for confidential transactions on the XRP Ledger. While the core cryptographic operations are correct, the API design has several areas that could be improved to enhance safety, usability, and maintainability. This document provides actionable recommendations for hardening the library interface before deployment.

Missing or Inconsistent Parameter Validation Most API functions do not validate input pointers before dereferencing them. If a caller passes `NULL`, the library will segfault rather than returning an error. Consider the `secp256k1_elgamal_decrypt` function, shown in figure D.1, which does not check the validity of any of its pointer parameters before using them.

```
int secp256k1_elgamal_decrypt( const secp256k1_context* ctx, uint64_t* amount, const secp256k1_pubkey* c1, const secp256k1_pubkey* c2, const unsigned char* privkey ) { secp256k1_pubkey S, M_target, current_M, G_point, next_M; const secp256k1_pubkey* pts[2]; uint64_t i; unsigned char one[32] = {0}; one[31] = 1;
```

```
/* 1. Recover Shared Secret: S = privkey * C1 */ S = *c1; if (!secp256k1_ec_pubkey_tweak_mul(ctx, &S, privkey)) return 0; Figure D.1: Missing NULL pointer checks in ElGamal encryption ( mpt-crypto/src/elgamal.c#L111-L125 )
```

In addition, semantic validation of parameters is performed inconsistently across the codebase. For example, the prover function `secp256k1_mpt_prove_same_plaintext_multi` does not validate the input randomness array `r_array` before using it, while the function `secp256k1_elgamal_pedersen_link_prove` validates its witness parameter:

```
/* 0. Validate Witnesses */ if (!secp256k1_ec_seckey_verify(ctx, r)) return 0; Figure D.2: Witness validation in link proof that is missing in other functions ( mpt-crypto/src/proof_link.c#L131-L132 )
```

PUBLIC Cryptographic Review

Recommendation: Add comprehensive NULL pointer checks and parameter validation at the start of all public API functions to prevent undefined behavior and enable graceful error handling. Implicit Length Assumptions Many functions accept byte array pointers with implicit length assumptions documented only in comments; an example is shown in figure D.3. This creates a risk of buffer over-reads if callers provide incorrectly sized buffers. `int generate_canonical_encrypted_zero( const secp256k1_context* ctx, secp256k1_pubkey* enc_zero_c1, secp256k1_pubkey* enc_zero_c2, const secp256k1_pubkey* pubkey, const unsigned char* account_id, // 20 bytes const unsigned char* mpt_issuance_id // 24 bytes` Figure D.3: Fixed-size byte arrays without explicit length parameters (`mpt-crypto/src/elgamal.c#L211-L217`)

Proving and verifying functions similarly accept proof buffers with lengths that are only documented in the prover implementation rather than validated at the API boundary: `/** * Verifies the proof of equality with shared randomness. */ int secp256k1_mpt_verify_equality_shared_r( const secp256k1_context* ctx, const unsigned char* proof, size_t n, const secp256k1_pubkey* C1, const secp256k1_pubkey* C2_vec, const secp256k1_pubkey* Pk_vec, const unsigned char* context_id` Figure D.4: Variable-length arrays without bounds validation (`mpt-crypto/include/secp256k1_mpt.h#L436-L446`)

Recommendation: Add explicit length parameters for all buffer arguments and validate them at function entry to prevent buffer overflows and over-reads. Generic Error Codes Currently, functions return only 0 for error and 1 for success, with no distinction between different failure modes. This makes debugging difficult, as callers cannot distinguish between NULL pointer errors, invalid inputs, cryptographic failures, or out-of-range values. Recommendation: Consider introducing structured error codes (e.g., -1 for NULL pointer, -2 for invalid input, -3 for crypto failure), as shown in figure D.5.

HashEye 59 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

`#define SECP256K1_MPT_OK 1 #define SECP256K1_MPT_ERROR 0 #define SECP256K1_MPT_ERROR_NULL_POINTER -1 #define SECP256K1_MPT_ERROR_INVALID_INPUT -2 #define SECP256K1_MPT_ERROR_CRYPTO_FAILED -3 #define SECP256K1_MPT_ERROR_OUT_OF_RANGE -4 #define SECP256K1_MPT_ERROR_BUFFER_TOO_SMALL -5` Figure D.5: Suggested return code convention Incomplete Parameter Documentation The API header file lacks comprehensive documentation for function parameters. Many functions do not specify whether parameters are input or output, what the required buffer sizes are, whether pointers can be NULL, or what specific values cause failures. `int secp256k1_bulletproof_verify_agg(const secp256k1_context* ctx, const secp256k1_pubkey* G_vec, /* length n = 64*m */ const secp256k1_pubkey* H_vec, /* length n = 64*m */ const unsigned char* proof, size_t proof_len, const secp256k1_pubkey* commitment_C_vec, /* length m */ size_t m, const secp256k1_pubkey* pk_base, const unsigned char* context_id` Figure D.6: Missing function documentation ( `mpt-crypto/include/secp256k1_mpt.h#L459-L468`)

Recommendation: Adopt comprehensive Doxygen-style documentation specifying input/output semantics, required buffer sizes, NULL-ability, thread safety, side-channel properties, and specific failure conditions for all public API functions. Adoption of a Unified Coding Standard The codebase lacks consistent coding conventions for control flow and error handling. Some functions use multiple return statements throughout the code, while others use single-exit patterns with goto cleanup. Some functions validate inputs incrementally, while others batch validation at the start. An example is shown in figure D.7. `free(L_vec); free(R_vec);`
`return ok ? 1 : 0;`
`fail: free(L_vec); free(R_vec); return 0;`

HashEye 60 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Figure D.7: Inconsistent error handling and control flow patterns (`mpt-crypto/src/bulletproof_aggregated.c#L2178-L2186`) Recommendation: Establish and document unified coding standards covering parameter validation order, error handling patterns (single-exit versus

multiple returns), memory allocation strategy, and naming conventions to improve code consistency and maintainability. Lack of Unified Data Structures Proofs are stored as separate arrays of values that are passed individually to proof generation and verification functions (see figure D.8, for example). This fragmented approach reduces API ergonomics and creates opportunities for integration errors when callers must manually coordinate multiple related parameters. SECP256K1_API int secp256k1_mpt_prove_same_plaintext_multi(const secp256k1_context* ctx, unsigned char* proof_out, size_t* proof_len, uint64_t amount_m, size_t n_ciphertexts, const secp256k1_pubkey* R_array, const secp256k1_pubkey* S_array, const secp256k1_pubkey* Pk_array, const unsigned char* r_array, // Flat array: r1 || r2 || ... (N * 32 bytes) const unsigned char* tx_context_id); Figure D.8: Related proof values passed as separate arrays (mpt-crypto/include/secp256k1_mpt.h#L239-L250)

In the example, the caller must manually ensure that R_array, S_array, Pk_array, and r_array all have exactly n_ciphertexts elements. If these arrays have mismatched lengths, the function will read out of bounds or use incorrect data, but the API provides no mechanism to detect this error at compile time or runtime. Similarly, proof outputs are returned as raw byte buffers rather than structured types, requiring callers to understand the internal binary layout. Recommendation: Introduce structured types that collect related values (e.g., secp256k1_mpt_ciphertext grouping R and S components, secp256k1_mpt_proof wrapping proof buffers) to improve readability, enable compile-time validation, and provide a more ergonomic API to library users.

Non-Distinguished Data Types Throughout the mpt-crypto codebase, the secp256k1_pubkey type is consistently used to represent elliptic curve points used more generally in ElGamal encryption, Pedersen

HashEye 61 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

commitments, Bulletproofs, and other contexts. However, it is also used in other portions of the protocol codebase to represent actual public keys. This can lead to confusion and variable misuse, potentially putting sensitive values at risk. Recommendation: Create different types to separate distinct uses of the same data structures. From a performance and compile-time perspective, typedef is free. However, when coupled with suitable compiler flags (such as -Wpedantic), it can identify when cross-uses of the same data structure occur, catching potential errors. Since valid cross-uses can be resolved by casting, false positives can be easily resolved.

HashEye 62 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

E. Automated Analysis Tool Configuration We used the following tools to perform automated testing of the codebase. E.1. Semgrep We used the static analyzer Semgrep to search for weaknesses in the source code repository. semgrep --metrics=off --sarif --config "p/hasheye" Figure E.1: The invocation command used to run Semgrep with HashEye' public rules The Testing Handbook provides detailed information about how to set up Semgrep and how to add it to continuous integration. E.2. CodeQL We used CodeQL to detect known vulnerabilities present in the codebase. This analysis did not identify any issues in the implementation in scope. After having built the CodeQL database, one can run the query suite cpp-security-extended.qls on an existing database, using the following command. codeql database analyze --format=sarif-latest --output=cpp-security-extended.sarif --codeql.db cpp-security-extended.qls Figure E.2: The command used to run the query suite cpp-security-extended.qls on the codebase

E.3. Cppcheck To install Cppcheck, we followed the instructions on the official website. We used the following command to run Cppcheck on the codebase. cppcheck --output-file=cppcheck.sarif --output-format=sarif . Figure E.3: The command used to run Cppcheck on the codebase

HashEye 63 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

F. Fix Review Results When undertaking a fix review, HashEye 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. From March 31 to April 2, 2026, HashEye reviewed the fixes and mitigations implemented by the Ripple team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving

the associated issue. We reviewed a number of commits across a resolution project. In summary, of the 17 issues described in this report, Ripple has fully resolved 14, has partially resolved two, and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Undocumented ElGamal decryption limit	Low	Resolved
2	Deprecated OpenSSL SHA-256 API and unchecked return values	Informational	Resolved
3	Fiat-Shamir domain tag considerations	Informational	Resolved
4	Timing side channels in ElGamal decryption	Informational	Resolved
5	Timing side channels in Bulletproof generation	Undetermined	Partially Resolved
6	Aggregated Bulletproofs cannot handle certain in-range values	High	Resolved
7	Missing participant count validation	Low	Resolved
8	Missing return value checks	Informational	Resolved
9	Insufficient compiler warning flags and security hardening options	Informational	Resolved

HashEye 64 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

10 Missing range proof enables confidential balance overdraft High Resolved 11 ConfidentialClawback resets the balance version counter Medium Resolved 12 ConfidentialSend increments the receiver's version counter High Resolved 13 MergeInbox allows version counter increment without balance change Informational Unresolved 14 Discrepancies between specification fields and implementation Informational Partially Resolved 15 Incorrect TransactionContextID for Convert and ConvertBack Medium Resolved 16 Multi-ciphertext equality proof uses per-recipient nonces instead of shared r Medium Resolved 17 Missing input length validation in low-level cryptographic helper functions Low Resolved

Detailed Fix Review Results

TOB-RIPCTX-1: Undocumented ElGamal decryption limit Resolved. The discrete logarithm recovery routine has been renamed to `secp256k1_solve_dlp_small_range_ct`, and the hard-coded limit has been replaced with an argument, `max_range`. However, the code that calls `secp256k1_solve_dlp_small_range_ct` calls it with a "magic number" of 1000000. As mentioned in appendix B, we recommend replacing such constants with descriptive names.

TOB-RIPCTX-2: Deprecated OpenSSL SHA-256 API and unchecked return values Resolved. Calls to the deprecated `SHA256_*` API have been migrated to the EVP API, and return values are now checked appropriately.

TOB-RIPCTX-3: Fiat-Shamir domain tag considerations Resolved. The same-plaintext proofs have been consolidated to the "shared R" variant, and all proofs now use distinct domain tags.

HashEye 65 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

TOB-RIPCTX-4: Timing side channels in ElGamal decryption Resolved. The `secp256k1_solve_dlp_small_range_ct` function now runs in constant time, relying on masking operations to track failures and successes and using constant-time comparisons.

TOB-RIPCTX-5: Timing side channels in Bulletproof generation Partially resolved. The `fix-audit-recommendations` branch does not remove the branching behavior, but it does replace the `memcmp` calls with a non-branching alternative. Additionally, the function documentation has been updated to reflect side channel risks: This function is called in two contexts: 1. `secp256k1_bulletproof_ipa_compute_LR` (prover only): - Round 0: scalars are a_L/b_R in $\{0,1\}$, derived from the prover's secret. - Rounds 1+: scalars are general 256-bit folded values. Timing varies with the scalar's Hamming weight. Because the prover operates on their own secret, exploitation requires an external attacker to have precise timing observation over the prover's local execution environment.

TOB-RIPCTX-6: Aggregated Bulletproofs cannot handle certain in-range values Resolved. When a multiscalar multiplication results in the point at infinity, this is flagged in the return value, which is checked by the calling code. We note that the tests for these cases do not validate the expected return value, only that the underlying function call does not crash.

TOB-RIPCTX-7: Missing participant count validation Resolved. A strict limit of four inputs/participants has been instituted for the Bulletproofs and same-plaintext proofs. It is worth noting that, in the Bulletproofs implementation, a named constant (`BP_MAX_VALUES`) is used for the check, while the same-plaintext proof uses a hard-coded "magic number" of 4.

TOB-RIPCTX-8: Missing return value checks Resolved. The cited calls to `secp256k1_ec_pubkey_serialize` have been updated to check the output length.

TOB-RIPCTX-9: Insufficient compiler warning flags and security hardening options Resolved. The build files have been updated to enable warnings, and portions of the codebase have been updated to address unsafe behavior.

HashEye 66 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

TOB-RIPCTX-10: Missing range proof enables confidential balance overdraft Resolved. The confidential send and confidential convert back modules have both been updated to include the new range proofs. TOB-RIPCTX-11: ConfidentialClawback resets the balance version counter Resolved. The balance version counter is now incremented and checked before return. TOB-RIPCTX-12: ConfidentialSend increments the receiver's version counter Resolved. The receiver's version counter is no longer incremented, and is checked to match the pre-transaction value before return. TOB-RIPCTX-13: MergeInbox allows version counter increment without balance change Unresolved. The Ripple team noted that no change was made to the implementation. TOB-RIPCTX-14: Discrepancies between specification fields and implementation Partially resolved. Credential checks have been integrated into the confidential send operation. However, the sfConfidentialBalanceVersion field has not been added to the protocol schema. TOB-RIPCTX-15: Incorrect TransactionContextID for Convert and ConvertBack Resolved. The context hash computations have been updated to match the specification. TOB-RIPCTX-16: Multi-ciphertext equality proof uses per-recipient nonces instead of shared r Resolved. The confidential transfer code now uses the "shared R" variant of the proof provided in the MPT library. TOB-RIPCTX-17: Missing input length validation in low-level cryptographic helper functions Resolved. Length checks for all inputs have been added to the identified functions.

HashEye 67 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

6. Fix Review Status Categories The following table describes the statuses used to indicate whether an issue has been sufficiently addressed. Fix Status Status Description Undetermined The status of the issue was not determined during this engagement. Unresolved The issue persists and has not been resolved. Partially Resolved The issue persists but has been partially resolved. Resolved The issue has been sufficiently resolved.

HashEye 68 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

About HashEye Founded in 2012 and headquartered in New York, HashEye 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/hasheye-io/publications>, with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye 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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom. To keep up with our latest news and announcements, please follow hasheye on X or LinkedIn and explore our public repositories at <https://github.com/hasheye-io>. To engage us directly, visit our "Contact" page at <https://www.hasheye.io/contact> or email us at info@hasheye.io. HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 <https://www.hasheye.io> info@hasheye.io

HashEye 69 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review

Notices and Remarks Copyright and Distribution © 2026 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. HashEye considers this report public information; it is licensed to Ripple Labs, Inc. under the terms of the project statement of work and has been made public at Ripple Labs, Inc.'s request. Material within this report may not be reproduced or distributed in part or in whole without HashEye' express written permission. The sole canonical source for HashEye publications is the HashEye Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic. Test Coverage Disclaimer

HashEye performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan. Security assessment projects are time-boxed and often rely on information 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. HashEye uses automated testing techniques to rapidly test software controls and security properties. 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. A project's time and resource constraints also limit their use.

HashEye 70 Ripple Labs XRP Ledger Confidential Transfer

PUBLIC Cryptographic Review