HashEye

// SMART CONTRACT SECURITY AUDIT

Anza BLS Signatures

Security assessment by HashEye · prepared for Anza

HASHEYE AUDITED

PROJECT	Anza BLS Signatures
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	Anza
CATEGORY	Blockchain
PUBLISHED	March 1, 2026
REPORT ID	research-anza-bls-signatures-2026-03-01-17kgi1

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-anza-bls-signatures-2026-03-01-17kgi1.

Anza BLS Signatures Security Assessment February 26, 2026

Prepared for: Sam Kim Anza

Prepared by: Joop van de Pol

HashEye

PUBLIC

Table of Contents Table of Contents 1

Project Summary 2

Executive Summary 3

Project Goals 5

Project Targets 6

Project Coverage 7

Summary of Findings 8

Detailed Findings 9

About HashEye 24

Notices and Remarks 25

HashEye 1 Anza BLS Signatures

PUBLIC Security Assessment

Project Summary Contact Information The following project manager was associated with this project: Tara Goodwin-Ruffus, Project Manager tara.goodwin-ruffus@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 consultant was associated with this project: Joop van de Pol, Consultant joop.vandepol@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event February 3, 2026 Pre-project kickoff call

February 13, 2026 Delivery of report draft February 17, 2026 Report readout meeting February 26, 2026 Delivery of final comprehensive report

HashEye 2 Anza BLS Signatures

PUBLIC Security Assessment

Executive Summary Engagement Overview Anza engaged HashEye to review the security of the BLS signature module in their Solana SDK. The BLS signature module is a Rust wrapper around the blst library implementing the draft IRTF CFRG BLS RFC. It provides private key generation, public key derivation and aggregation, signature generation, verification and aggregation, and proof of possession generation. One consultant conducted the review from February 9 to February 13, 2026, for a total of one engineer-week of effort. Our testing efforts focused on adherence to the specification and on the correct use of the underlying BLS library. With full access to source code and documentation, we performed static and dynamic testing of the BLS signature module, using automated and manual processes. Observations and Impact The implementation is straightforward and uses the underlying blst library correctly. We identified a number of deviations from the specification, which, depending on the user of the library, could lead to rogue key attacks (TOB-ABLS-2), equivocation attacks (TOB-ABLS-3), or zero-valued private keys (TOB-ABLS-4). Recommendations Based on the findings identified during the security review, HashEye recommends that Anza take the following steps before deployment: • Remediate the findings disclosed in this report. These findings should be addressed through direct fixes or broader refactoring efforts. • Implement one specific scheme from the RFC. The current implementation mixes aspects from two of the three schemes in the RFC. Based on the intended use of this module within the wider system, we recommend focusing the implementation on the proof of possession scheme. • Document the responsibilities of the user. The implementation makes certain implicit assumptions on how the user uses the module, which are currently undocumented. For example, it assumes that the user has verified proofs of possession of each public key before aggregating and verifying signatures, but this is not documented in any of the verification functions.

HashEye 3 Anza BLS Signatures

PUBLIC Security Assessment

Finding Severities and Categories

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 0 Medium 1 Low 1 Informational 3 Undetermined 1

CATEGORY BREAKDOWN Category Count Access Controls 1 Cryptography 4 Data Validation 1

HashEye 4 Anza BLS Signatures

PUBLIC Security Assessment

Project Goals The engagement was scoped to provide a security assessment of the Anza BLS signatures module. Specifically, we sought to answer the following non-exhaustive list of questions: • Does the implementation deviate from the RFC? • Does the implementation contain any flaws that would enable rogue key attacks? • Does the implementation correctly use the blst library, including bindings? • Does the implementation correctly handle special points that are not on the curve, are not in the correct subgroup, or are the point at infinity? • Does the implementation correctly aggregate signatures and public keys?

HashEye 5 Anza BLS Signatures

PUBLIC Security Assessment

Project Targets The engagement involved reviewing and testing the following target. bls-signatures module

Repository https://github.com/anza-xyz/solana-sdk Version f2d15de6f7a1715ff806f0c39bba8f64bf6a587d

HashEye 6 Anza BLS Signatures

PUBLIC Security Assessment

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: • Manual source code review of the following components: ○ Private key generation and usage ○ Public key derivation and aggregation ○ Signature aggregation and verification ○ Proof of possession generation and verification ○ Utility functions (e.g., key file storage) • Static code analysis (Semgrep, CodeQL) ○ No issues found 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: • Third-party dependencies (e.g., blstrs, blst) • Usage of the module by the larger system

HashEye 7 Anza BLS Signatures

PUBLIC Security Assessment

Summary of Findings The table below summarizes the findings of the review, including details on type and severity. ID Title Type Severity 1 Mismatched and incorrect domain separation tags Cryptography Informational 2 Signature verification does not prevent rogue key attacks Cryptography Undetermined 3 Public key construction does not reject identity point Cryptography Low 4 Missing length checks on input key material Cryptography Medium 5 Parallel aggregation turns an empty iterator into the identity point Data Validation Informational 6 Writing a keypair to file does not set file permissions on Windows Access Controls Informational

HashEye 8 Anza BLS Signatures

PUBLIC Security Assessment

Detailed Findings 1. Mismatched and incorrect domain separation tags Severity: Informational Difficulty: N/A Type: Cryptography Finding ID: TOB-ABLS-1 Target: src/hash.rs, src/proof_of_possession/mod.rs

Description The domain separation tags used for hashing messages and public keys correspond to different signature schemes in the RFC, and the tag for public keys is incorrect. The tag used for hashing messages is taken from the basic scheme (NUL), as shown in the following code excerpt. /// Domain separation tag used for hashing messages to curve points to prevent /// potential conflicts between different BLS implementations. This is defined /// as the ciphersuite ID string as recommended in the /// [standard](https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05#se ction-4.2.1). pub const HASH_TO_POINT_DST: &[u8] = b"BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_"; Figure 1.1: solana-sdk/bls-signatures/src/hash.rs#L3–L7 Conversely, the domain separation tag used for hashing public keys is taken from the proof of possession scheme (POP).

/// Domain separation tag used when hashing public keys to G2 in the proof of /// possession signing and verification functions. See the /// [standard] (https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05#se ction-4.2.3). pub const POP_DST: &[u8] = b"BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_"; Figure 1.2: solana-sdk/bls-signatures/src/proof_of_possession/mod.rs#L16–L19 This is not consistent, and furthermore, this is not the correct tag for hashing public keys. This is the domain separation tag that the specification defines for the hash_to_point function instead of the tag defined for the hash_pubkey_to_point function. I.e., instead of "BLS_SIG", the tag should begin with "BLS_POP", as described in section 4.2.3 of the RFC.

HashEye 9 Anza BLS Signatures

PUBLIC Security Assessment

However, this discrepancy with the spec is not exploitable, because the specification defines this domain separation tag for the proof of possession signature scheme ("POP"), and the implementation uses the basic signature scheme instead ("NUL"). This makes it impossible to confuse signatures and proofs of possession. Recommendations Short term, update the tag to the correct one for hashing public keys. Furthermore, decide which schemes are suitable for the larger system. Rather than mix and match between the basic and proof of possession schemes, choose one to use and fully implement it. Alternatively, implement both but use Rust types to ensure that they cannot be used interchangeably. Long term, implement known-answer tests based on test vectors from the RFC once these are added to ensure that the implementation matches the specification.

HashEye 10 Anza BLS Signatures

PUBLIC Security Assessment

2. Signature verification does not prevent rogue key attacks Severity: Undetermined Difficulty: Undetermined Type: Cryptography Finding ID: TOB-ABLS-2 Target: src/signature/points.rs

Description The signature verification routines in the signature module do not verify that the messages are distinct, nor that the public keys have a valid proof of possession, which enables rogue key attacks unless the user verifies proofs of possession themselves. Consider the verify_distinct function, which is supposed to verify an aggregated signature over a set of distinct messages and public keys. /// Verifies an aggregated signature over a set of distinct messages and /// public keys. pub fn verify_distinct<'a, P, S>(public_keys: impl ExactSizeIterator<Item = &'a P>, signatures: impl ExactSizeIterator<Item = &'a S>, messages: impl ExactSizeIterator<Item = &'a [u8]>,) → Result<(), BlsError> where P: AsPubkeyAffine + 'a + ?Sized, S: AddToSignatureProjective + 'a + ?Sized, { if public_keys.len() ≠ messages.len() || public_keys.len() ≠ signatures.len() { return Err(BlsError::InputLengthMismatch); } if public_keys.len() = 0 { return Err(BlsError::EmptyAggregation); } let aggregate_signature = SignatureProjective::aggregate(signatures)?; Self::verify_distinct_aggregated(public_keys, &aggregate_signature, messages) } Figure 2.1: solana-sdk/bls-signatures/src/signature/points.rs#L101-L120 Neither it nor the called verify_distinct_aggregated function verifies that the messages are in fact distinct.

/// Verifies a pre-aggregated signature over a set of distinct messages and /// public keys. pub fn verify_distinct_aggregated<'a, P, S>(public_keys: impl ExactSizeIterator<Item = &'a P>,

HashEye 11 Anza BLS Signatures

PUBLIC Security Assessment

aggregate_signature: &S, messages: impl ExactSizeIterator<Item = &'a [u8]>,) → Result<(), BlsError> where P: AsPubkeyAffine + 'a + ?Sized, S: AsSignatureAffine + ?Sized, { if public_keys.len() ≠ messages.len() { return Err(BlsError::InputLengthMismatch); } if public_keys.len() = 0 { return Err(BlsError::EmptyAggregation); }

// TODO: remove `Vec` allocation if possible for efficiency let mut pubkeys_affine = alloc::vec::Vec::with_capacity(public_keys.len()); let public_keys_len = public_keys.len(); for pubkey in public_keys { let g1_affine = pubkey.try_as_affine()?; pubkeys_affine.push(g1_affine.0); }

let mut prepared_hashes = alloc::vec::Vec::with_capacity(messages.len()); for message in messages { let hashed_message: G2Affine = hash_signature_message_to_point(message).into(); prepared_hashes.push(G2Prepared::from(hashed_message)); }

let aggregate_signature_affine = aggregate_signature.try_as_affine()?; let signature_prepared = G2Prepared::from(aggregate_signature_affine.0);

#[cfg(feature = "std")] let neg_g1_generator = &*NEG_G1_GENERATOR_AFFINE; #[cfg(not(feature = "std"))] let neg_g1_generator_val: G1Affine = (-G1Projective::generator()).into(); #[cfg(not(feature = "std"))] let neg_g1_generator = &neg_g1_generator_val;

let mut terms = alloc::vec::Vec::with_capacity(public_keys_len.saturating_add(1)); for i in 0..public_keys_len { terms.push((&pubkeys_affine[i], &prepared_hashes[i])); } terms.push((neg_g1_generator, &signature_prepared));

let miller_loop_result = Bls12::multi_miller_loop(&terms); (miller_loop_result.final_exponentiation() = Gt::identity()) .then_some(()) .ok_or(BlsError::VerificationFailed) } Figure 2.2: solana-sdk/bls-signatures/src/signature/points.rs#L122-L174

HashEye 12 Anza BLS Signatures

PUBLIC Security Assessment

The same holds for the verify_aggregate function, which merely aggregates the public keys and signatures and uses the aggregate public key to verify the aggregate signature. Note that this issue is exploitable only if the caller does not verify proofs of possession for every public key. However, there are several issues that make this more likely: • The implementation seems to

implement the basic scheme according to the used domain separation tag for signatures. In this scheme, the signature verification is supposed to verify that all messages are distinct. • The verify_distinct function does not actually check that the messages are distinct. • However, the implementation also implements part of the proof of possession scheme. In this scheme, the signature verification is supposed to verify that each public key has a valid proof of possession. • None of the verification functions document the fact that the caller needs to verify proofs of possession for each of the public keys. Recommendations Short term, document that, for each verification function that does not guarantee distinct messages, the caller must ensure that each public key has a valid proof of possession. Add a check that messages are distinct to the verify_distinct_aggregated function. Long term, add negative tests confirming that duplicated messages are rejected by the verification functions that enforce distinct messages.

HashEye 13 Anza BLS Signatures

PUBLIC Security Assessment

3. Public key construction does not reject identity point Severity: Low Difficulty: Medium Type: Cryptography Finding ID: TOB-ABLS-3 Target: src/macros.rs, src/secret_key.rs

Description The specification requires that key validation rejects the identity point as a public key, but the implementation accepts such keys. Consider, for example, constructing a public key in affine form from the uncompressed coordinates, shown in the following code excerpt. // Uncompressed Bytes → Affine Point impl TryFrom<&$uncompressed> for $affine { type Error = crate::error::BlsError; fn try_from(bytes: &$uncompressed) → Result<Self, Self::Error> { let maybe_point: Option<$blstrs_affine> = <$blstrs_affine>::from_uncompressed(&bytes.0).into(); let point = maybe_point.ok_or(crate::error::BlsError::PointConversion)?; Ok(Self(point)) } } Figure 3.1: solana-sdk/bls-signatures/src/macros.rs#L118-L127 This is a macro that calls the blstrs from_uncompressed function for the corresponding affine type. That function checks whether the point is on the curve and whether it is torsion-free. The identity point passes both those checks and is accepted. This is one example, but all functions allow constructing public keys equal to the identity point, including by defining a secret key with value zero and deriving the corresponding public key. The implementation allows both signing and verification using those keys. As an aside, the default values for uncompressed and compressed public keys consist of all-zero byte arrays, which do not correspond to valid points on the curve. Any attempted operations with these default values will fail due to a deserialization error. Exploit Scenario A malicious signer registers the identity point as their public key. After providing a signature, they can later equivocate about which message they signed, as every message results in the same signature.

HashEye 14 Anza BLS Signatures

PUBLIC Security Assessment

Recommendations Short term, add checks ensuring that public keys are not the identity point (both when constructing using the conversion functions and when generating from a secret key). Long term, consider making it impossible to represent invalid keys using the Rust type system or whether you want to explicitly implement and invoke the KeyValidate function as required by the spec. Add negative tests using invalid keys to ensure that they are rejected by the relevant functions.

HashEye 15 Anza BLS Signatures

PUBLIC Security Assessment

4. Missing length checks on input key material Severity: Medium Difficulty: Medium Type: Cryptography Finding ID: TOB-ABLS-4 Target: src/secret_key.rs

Description The RFC states that the input key material (IKM) of key generation must be at least 32 bytes, but the implementation does not enforce this. The derive function that calls the blst_keygen function does not perform any checks, and will accept an IKM of any length, including zero. /// Derive a `BlsSecretKey` from a seed (input key material) pub fn derive(ikm: &[u8]) → Result<Self, BlsError> { let mut scalar = blst_scalar::default(); unsafe { blst_keygen(&mut scalar as *mut blst_scalar, ikm.as_ptr(), ikm.len(), ptr::null(), 0,); } scalar .try_into() .map(Self) .map_err(|_| BlsError::FieldDecode) } Figure 4.1: solana-sdk/bls-signatures/src/secret_key.rs#L33-L49 More critically, blst_keygen will silently fail and return the zero scalar when the IKM length is less than 32. Exploit Scenario A user makes a mistake and inputs an IKM shorter than 32 bytes when deriving a key. The resulting private key is the zero key. Recommendations Short term, add

checks on the input IKM length and on the blst_keygen output. Long term, add negative tests to confirm that short IKMs are rejected.

HashEye 16 Anza BLS Signatures

PUBLIC Security Assessment

5. Parallel aggregation turns an empty iterator into the identity point Severity: Informational Difficulty: N/A Type: Data Validation Finding ID: TOB-ABLS-5 Target: src/signature/points.rs, src/signature/mod.rs, src/pubkey/points.rs

Description Performing parallel aggregation over an empty list results in the identity element, whereas sequential aggregation over an empty list results in an empty aggregation error. Consider the following parallel signature aggregation code excerpt. /// Aggregate a list of signatures #[allow(clippy::arithmetic_side_effects)] #[cfg(feature = "parallel")] pub fn par_aggregate<'a, S: AddToSignatureProjective + Sync + 'a>(signatures: impl ParallelIterator<Item = &'a S>,) → Result<SignatureProjective, BlsError> { signatures .into_par_iter() .fold(|| Ok(SignatureProjective::identity()), |acc, signature| { let mut acc = acc?; signature.add_to_accumulator(&mut acc)?; Ok(acc) },) .reduce_with(|a, b| { let mut a_val = a?; let b_val = b?; a_val.0 += b_val.0; Ok(a_val) }) .ok_or(BlsError::EmptyAggregation)? } Figure 5.1: solana-sdk/bls-signatures/src/signature/points.rs#L188-L211 Because fold initializes the accumulator with the identity element even if the iterator is empty, reduce_with will always result in a value. Therefore, it will never return the BlsError::EmptyAggregation error. Compare this to the sequential aggregation shown in the following code excerpt.

HashEye 17 Anza BLS Signatures

PUBLIC Security Assessment

/// Aggregate a list of signatures #[allow(clippy::arithmetic_side_effects)] pub fn aggregate<'a, S: AddToSignatureProjective + ?Sized + 'a>(signatures: impl Iterator<Item = &'a S>,) → Result<SignatureProjective, BlsError> { let mut aggregate = SignatureProjective::identity(); let mut count = 0; for signature in signatures { signature.add_to_accumulator(&mut aggregate)?; count += 1; } if count == 0 { return Err(BlsError::EmptyAggregation); } Ok(aggregate) } Figure 5.2: solana-sdk/bls-signatures/src/signature/points.rs#L68-L83 For an empty iterator, the count will be zero and the aggregate function will return the BlsError::EmptyAggregation error, unlike the parallel case. The behavior of parallel aggregation is even captured by one of the existing test cases, as shown in the following code excerpt.

// Test empty case let empty: std::vec::Vec<SignatureProjective> = Vec::new(); let empty_agg = SignatureProjective::par_aggregate(empty.par_iter()).unwrap(); assert_eq!(empty_agg, SignatureProjective::identity()); Figure 5.3: solana-sdk/bls-signatures/src/signature/mod.rs#L375-L378 Public key aggregation has the same issues. Recommendations Short term, make the parallel and sequential aggregation functions consistent. Long term, add consistency tests between parallel and sequential aggregation for both positive and negative test cases.

HashEye 18 Anza BLS Signatures

PUBLIC Security Assessment

6. Writing a keypair to file does not set file permissions on Windows Severity: Informational Difficulty: N/A Type: Access Controls Finding ID: TOB-ABLS-6 Target: src/keypair.rs

Description Saving a keypair to a file will only set file permissions for Unix-based systems, which causes keypair files created on Windows to inherit the default ACLs from the parent directory. let mut f = { #[cfg(not(unix))] { OpenOptions::new() } #[cfg(unix)] { use std::os::unix::fs::OpenOptionsExt; OpenOptions::new().mode(0o600) } } .write(true) .truncate(true) .create(true) .open(outfile)?;

self.write_json(&mut f) Figure 6.1: solana-sdk/bls-signatures/src/keypair.rs#L137-L153 Recommendations Short term, document this behavior such that the user is aware of it and can select the folder appropriately. Long term, consider adding an additional dependency to set file permissions in Windows systems such as the windows-acl, windows-permissions, or windows crates.

HashEye 19 Anza BLS Signatures

PUBLIC Security Assessment

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 20 Anza BLS Signatures

PUBLIC Security Assessment

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 21 Anza BLS Signatures

PUBLIC Security Assessment

B. Code Quality Recommendations This appendix contains recommendations for 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 public key as payload when generating PoP for key pair. When generating a PoP for a key pair, it uses the secret key, which freshly derives the public key when no payload is provided. Instead, it would be more efficient to use the stored public key as a payload in this case. An example is shown in figure B.2. /// Generate a proof of possession for the given keypair pub fn proof_of_possession(&self, payload: Option<&[u8]>) → ProofOfPossessionProjective { self.secret.proof_of_possession(payload) } Figure B.1: Generating a proof of possession for a keypair does not use the public key (solana-sdk/src/keypair.rs#L55-L58) /// Generate a proof of possession for the given keypair pub fn proof_of_possession(&self, payload: Option<&[u8]>) → ProofOfPossessionProjective { // When payload is None, use the pre-computed pubkey bytes to avoid // recomputing the public key via scalar multiplication in SecretKey let pubkey_bytes; let effective_payload = match payload { Some(p) ⇒ Some(p), None ⇒ { pubkey_bytes = self.public.to_bytes_compressed(); Some(pubkey_bytes.as_slice()) } }; self.secret.proof_of_possession(effective_payload) } Figure B.2: Example implementation of PoP for key pairs that uses the public key • Speed up distinct aggregate verification using direct API for batch verification. The current implementation of verify_distinct_aggregated uses the blstrs wrapper, which performs the following separate steps: hash each message, convert to G2Affine, create G2Prepared, and perform the Miller loop with precomputed lines. However, the blst library exposes blst_pairing_aggregate_pk_in_g1, which internally performs hash-to-curve, accumulates directly into the pairing context, and avoids intermediate conversions and allocations. The Rust bindings expose this function through blst::Pairing::aggregate.

HashEye 22 Anza BLS Signatures

PUBLIC Security Assessment

• Remove unnecessary cloning when writing JSON. Instead of cloning the JSON, use json.as_bytes() directly. pub fn write_json<W: Write>(&self, writer: &mut W) → Result<String, Box<dyn error::Error>> { let json = serde_json::to_string(&Into::<[u8; BLS_KEYPAIR_SIZE]>::into(self).as_slice())?; writer.write_all(&json.clone().into_bytes())?; Ok(json) } Figure B.3: Unnecessary cloning when writing JSON (solana-sdk/src/keypair.rs#L121-L125)

HashEye 23 Anza BLS Signatures

PUBLIC Security Assessment

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 24 Anza BLS Signatures

PUBLIC Security Assessment

 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 25 Anza BLS Signatures

PUBLIC Security Assessment