# Auditable Key Directory (AKD) Implementation Review

Meta Platforms

Version 1.0 – November 14, 2023



**Prepared By**
Elena Bakos Lang
Gérald Doussot
Kevin Henry
Thomas Pornin

# 1 Executive Summary

## Synopsis

In August 2023, Meta engaged NCC Group's Cryptography Services practice to perform an implementation review of their Auditable Key Directory (AKD) library, which provides an append-only directory of public keys mapped to user accounts and a framework for efficient cryptographic validation of this directory by an auditor. The library is being leveraged to provide an AKD for WhatsApp and is meant to serve as a reference implementation for auditors of the WhatsApp AKD, as well as to allow other similar services to implement key transparency. The review was performed remotely by 3 consultants over a two-week period with a total of 20 person-days spent.

In October 2023, the project team completed a retest on a series of fixes proposed by Meta and found that they effectively addressed all findings documented in this report. These changes have been merged as of tagged release `v0.11.0` (commit `85b3b07`).

## Scope

NCC Group's evaluation targeted the open source AKD library at github.com/facebook/akd/, release `v0.9.0` (commit `be1055e`), with primary targets of *akd/* and *akd_core/*.

The review was supplemented by draft 15 of the IETF document Verifiable Random Functions (VRFs), now published as RFC 9381, as well SEEMless and Parakeet.

## Limitations

While the review covered the complete *akd* repository, it was primarily focused on cryptographic primitives, such as the use of verifiable random functions (VRFs), and the associated membership proofs for proving the existence or non-existence of an entry in the key directory at a given epoch. Less attention was given to the non-cryptographic performance optimizations within the library, such as the storage caching and parallelization strategy. Furthermore, no integration of this library with an existing application, such as WhatsApp, was reviewed as part of this report.

## Key Findings

In total, 1 medium severity, 8 low severity, and 6 informational findings were filed, including:

- Finding "Multiple Key Updates During Epoch Results in Invalid State" detailed a preventable case where a user's key updates were not correctly placed in the tree.
- Finding "VRF Hash To Curve Accepts Non-Canonical Encodings" showed how the VRF might have returned an incorrect output, impacting interoperability.
- Finding "Dangerous Public API Functions" detailed API functions which may mislead a user of the library about their behavior.

After retesting, NCC Group found that all reported findings had been addressed by Meta in a manner consistent with the recommendations put forth during the initial review.

## Strategic Recommendations

- At the time of review, all external dependencies were found to be up-to-date. Continuing to maintain such an up-to-date status at each release is recommended.
- While some negative tests are in place, more robust testing of the public API functions, focusing on negative tests or invalid input (e.g., fuzzing), is recommended, as it may reveal additional unanticipated behavior, or prevent the introduction of new bugs.
- The correct behavior of *akd* relies on proper integration with an external application that authenticates users and publishes updates to the directory. This integration must be done properly to ensure the *akd* is correctly maintained and provides the appropriate assurance to users. Further review of such an integration is recommended.

# 2 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

| Title | Status | ID | Risk |
|---|---|---|---|
| Multiple Key Updates During Epoch Results in Invalid State | Fixed | Q3U | Medium |
| VRF Hash to Curve Function May Incorrectly Return the Identity Point | Fixed | EAD | Low |
| VRF Expanded Private Key Not Fully Zeroized on Drop | Fixed | NWP | Low |
| VRF Verifier Will Not Reject Public Keys with Low Order | Fixed | DHG | Low |
| VRF Hash To Curve Accepts Non-Canonical Encodings | Fixed | KAG | Low |
| Dangerous Public API Functions | Fixed | FRK | Low |
| Malformed Input May Crash Client Applications | Fixed | Y7E | Low |
| Malformed VRF Proof May Crash Client Applications | Fixed | JJ4 | Low |
| Malformed History Proof May Crash Client Applications | Fixed | 9NP | Low |
| VRF Draft Specification Now Published as RFC 9381 | Fixed | 7Q6 | Info |
| Incorrect Function Documentation for `get_commitment_nonce()` and `compute_fresh_azks_value()` | Fixed | RKB | Info |
| The `hash_to_curve()` Function Should be Renamed `encode_to_curve()` | Fixed | CX4 | Info |
| Improved Error Messages When Auditing History Proofs | Fixed | 9JD | Info |
| Minor Optimization When Computing Longest Prefix | Fixed | CUF | Info |
| Potentially Confusing Behavior for NodeLabels | Fixed | MPR | Info |

# 3 Finding Details

Medium

## Multiple Key Updates During Epoch Results in Invalid State

| | | | |
|---|---|---|---|
| **Overall Risk** | Medium | **Finding ID** | NCC-E008327-Q3U |
| **Impact** | High | **Component** | akd |
| **Exploitability** | Undetermined | **Category** | Cryptography |
| | | **Status** | Fixed |

### Impact

Including two items for the same label when updating the tree state via the `publish()` function would have resulted in an invalid tree state with no valid key stored for the affected user and may have caused correctness or usability issues.

### Description

The function `publish()` is used to update the existing tree with a list of new or updated keys for users of the AKD system. In particular, the `publish()` function takes as input a list of updates, in the form of `(akd_label, akd_value)` pairs. For each `akd_label`, the corresponding `NodeLabel` is generated by hashing the `akd_label`, the freshness, and the version number of the key, which denotes the internal label used for the node within the AKD tree. Finally, the `NodeLabel`s are recursively inserted into the existing AKD tree.

During this process, if this is the first time a particular `akd_label` was seen, the `NodeLabel` is generated using freshness `VersionFreshness::Fresh` and version `1`, and otherwise the `NodeLabel` will be generated using `VersionFreshness::Stale` and the version incremented by `1` from the version currently stored in the tree:

```
    None => vec![(
        akd_label.clone(),
        VersionFreshness::Fresh,
        1u64,
        akd_value.clone(),
    )],
    Some((latest_version, existing_akd_value)) => {
        if existing_akd_value == akd_value {
           // Skip this because the user is trying to re-publish the same value
           return vec![];
        }
        vec![
           (
               akd_label.clone(),
               VersionFreshness::Stale,
               *latest_version,
               akd_value.clone(),
           ),
           (
               akd_label.clone(),
               VersionFreshness::Fresh,
               *latest_version + 1,
```

```
                akd_value.clone(),
            ),
        ]
    }
```

Figure 1: `publish()` in akd/src/directory.rs

However, if multiple updates for the same `akd_label` with differing values are passed to `publish()`, the version numbers included here will match, as the `latest_version` will not be updated between iterations. Hence, two items with identical `NodeLabel`s will be passed to `recursive_batch_insert_nodes()` to be inserted into the tree.

As part of this insertion process, each call to `recursive_batch_insert_nodes()` computes the longest common prefix of the current remaining nodes, and generates a new interior node if multiple elements still need to be sorted. In the above case, `recursive_batch_insert_nodes()` will eventually be called with a list containing only the two duplicated nodes, compute a longest common prefix corresponding exactly to the label of the duplicated nodes, insert an interior node with this label, and filter out both duplicates as the new interior node label is not a proper prefix of either of them:

```
        (None, _) => {
            // Case 3: The node label is None and the insertion still has
            // multiple elements, meaning that a new interior node should be
            // created with a label equal to the longest common prefix of
            // the node set.
            let lcp_label = azks_element_set.get_longest_common_prefix::<TC>();
            current_node = new_interior_node::<TC>(lcp_label, epoch);
            is_new = true;
            num_inserted = 1;
        }
```

Figure 2: `recursive_batch_insert_nodes()` in akd/src/append_only_zks.rs

This will result in an invalid tree state where an interior node has no leaves, and no valid key for the affected user is stored in the tree.

Note that the Parakeet paper documents that only one update per key should be included per epoch:

> The server stores a directory Dir of label-value pairs. Each value corresponds to a public key. The clients can request updates to their own public keys – equivalent to requesting a change to the state of Dir. For efficiency, many such requests are batched together, with updates going into effect at discrete time steps (epochs). So, Dir is stateful, has of an ordered sequence of states Dir_t, one state per epoch t.

However, this requirement is not currently enforced or documented within the `publish()` function.

## Recommendation

Ensure that invalid inputs to the `publish()` function are detected, and do not result in invalid states for the AKD tree. In particular, ensure that duplicates do not result in dangling interior nodes within the tree, by filtering out duplicates or returning an informative error.

Additionally, ensure that the requirements for a single update per epoch are well-documented for both the `publish()` function, and the system at large, and ensure that any higher-level APIs do not facilitate submitting multiple updates per epoch.

## Reproduction Steps

The test `test_simple_lookup()` can be modified to submit two values for the same label as follows:

```
    // Add two labels and corresponding values to the akd
    akd.publish(vec![
        (AkdLabel::from("hello"), AkdValue::from("world")),
        (AkdLabel::from("hello"), AkdValue::from("world2")),
    ])
```

While this will successfully complete the `publish()` and `lookup()` operations, the test will fail on the `lookup_verify()` call on line 207.

## Location

- *akd/akd/src/directory.rs*
- *akd/akd/src/append_only_zks.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 400 (commit `cd4fd18`) the `publish()` function now includes a check that all labels are distinct, and throws an error otherwise. As such, this finding is considered *fixed*.

Low

# VRF Hash to Curve Function May Incorrectly Return the Identity Point

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-EAD |
| **Impact** | High | **Component** | akd_core |
| **Exploitability** | Low | **Category** | Cryptography |
| | | **Status** | Fixed |

## Impact

The implemented approach was missing an identity check mandated in the specification, which may have introduced interoperability issues, invalidated security proofs, or weakened security guarantees of the VRF. In the worst case, the VRF private key could have been leaked.

## Description

As part of *akd_core*, an implementation of a verifiable random function (VRF) is provided. The implemented approach is `ECVRF-EDWARDS25519-SHA512-TAI` from draft 15 of the IETF document draft-irtf-cfrg-vrf-15. During both the proof generation and verification of this VRF, a hashed value must be mapped to the underlying elliptic curve using the defined method `ECVRF_encode_to_curve`. This process is non-trivial, as not all outputs will map directly to a valid point, and the process must be repeated in a deterministic manner until a valid mapping is achieved. To this end, the specification defines `ECVRF_encode_to_curve_try_and_increment`, which proceeds to loop based on the following:

> While H is "INVALID" or **H is the identity element** of the elliptic curve group

The corresponding implementation follows:

```
    pub(super) fn hash_to_curve(&self, alpha: &[u8]) -> EdwardsPoint {
        let mut result = [0u8; 32];
        let mut counter = 0;
        loop {
            let hash = Sha512::new()
                .chain([SUITE, ONE])
                .chain(self.0.as_bytes())
                .chain(alpha)
                .chain([counter, ZERO])
                .finalize();
            result.copy_from_slice(&hash[..32]);
            let wrapped_point = CompressedEdwardsY::from_slice(&result)
                .expect("Result hash should have a length of 32, but it does not")
                .decompress();
            counter += 1;
            if let Some(wp) = wrapped_point {
                return wp.mul_by_cofactor();
            }
        }
    }
```

*Figure 3: `hash_to_curve()` in akd_core/src/ecvrf/ecvrf_impl.rs*

Per the highlighted lines, the `hash_to_curve()` function will return on any valid point, **including the identity**, which contradicts the behavior in the specification. The output of this function is used to compute `gamma`, which is included in the VRF proof and used to derive the actual VRF output.

```
impl VRFExpandedPrivateKey {
    /// Produces a proof for an input (using the expanded private key)
    pub fn prove(&self, pk: &VRFPublicKey, alpha: &[u8]) -> Proof {
        let h_point = pk.hash_to_curve(alpha);
        let h_point_bytes = h_point.compress().to_bytes();
        let k_scalar = ed25519_Scalar::from_bytes_mod_order_wide(&nonce_generation_bytes(
            self.nonce,
            &h_point_bytes,
        ));
        let gamma = h_point * self.key;
        let c_scalar = hash_points(
            pk.0,
            &h_point_bytes,
            &[
                gamma,
                curve25519_dalek::constants::ED25519_BASEPOINT_TABLE * &k_scalar,
                h_point * k_scalar,
            ],
        );

        Proof {
            gamma,
            c: c_scalar,
            s: k_scalar + c_scalar * self.key,
        }
    }
```

*Figure 4:* `evaluate()` *in akd_core/src/ecvrf/ecvrf_impl.rs*

As highlighted above, if `h_point` is the identity, then `gamma` will be equal to the private key and leaked via its inclusion in the output `Proof`.

This finding is rated low, even though it is a direct deviation from the specification and may leak the private key, because it will only occur with negligible probability, when the first 32 bytes of the SHA-512 hash result in a low order element.

## Recommendation

Use `curve25519_dalek::IsIdentity` or a similar check to ensure `hash_to_curve` does not return the identity element. Note that the check should be performed after the `mul_by_cofactor()` call. See also finding "VRF Hash To Curve Accepts Non-Canonical Encodings", which is about non-canonical encodings and whose resolution also involves altering this code.

Note that this code was adapted from Diem's NextGen Crypto library, which appears to have the same issue; see *nextgen_crypto/src/vrf/ecvrf.rs*.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 401 (commit `78a5fd5`), the function `hash_to_curve()` has been renamed `encode_to_curve()` and now includes an explicit check for the identity element before returning the resulting point. As a result, this finding is considered *fixed*.

Low

# VRF Expanded Private Key Not Fully Zeroized on Drop

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-NWP |
| **Impact** | Medium | **Component** | akd_core |
| **Exploitability** | Low | **Category** | Data Exposure |
| | | **Status** | Fixed |

## Impact

Failure to clear sensitive values from memory may have allowed these values to leak to other processes running in the same memory space. In the case of cryptographic keys or similar secrets, the security of the underlying protocol may have been completely broken. In the current VRF implementation, an adversary may have gained an advantage in predicting the nonce used in a VRF proof, which in turn could have enabled an attack on the VRF private key.

## Description

The elliptic curve verifiable random function (*ecvrf*) component leverages the *curve25519_dalek* crate for elliptic curve operations. By default, this crate has the `zeroize` feature flag enabled, which ensures that all scalars and EC points are zeroized on `Drop`. The *ecvrf* component also leverages *ed25519_dalek* for several types.

The *ecvrf* implementation defines the `VRFExpandedPrivateKey` struct, which clones a similar `ExpandedSecretKey` struct from *ed25519_dalek*:

```
/// A longer private key which is slightly optimized for proof generation.
///
/// This is similar in structure to ed25519_dalek::ExpandedSecretKey. It can be produced
↳ from
/// a VRFPrivateKey.
#[derive(Clone)]
pub struct VRFExpandedPrivateKey {
    pub(super) key: ed25519_Scalar,
    pub(super) nonce: [u8; 32],
}
```

*Figure 5:* `VRFExpandedPrivateKey` *in akd_core/src/ecvrf/ecvrf_impl.rs*

One may infer that this struct was re-implemented to avoid dependence on the `hazmat` feature flag in *ed25519_dalek*. However, unlike the `ed25519_dalek::ExpandedSecretKey` struct, no zeroization of the `nonce` field is performed. Because this is a plain `u8` array, it will not be zeroized, whereas the `key` will be. Compare with the corresponding type in *ed25519_dalek*:

```
pub struct ExpandedSecretKey {
    /// The secret scalar used for signing
    pub scalar: Scalar,
    /// The domain separator used when hashing the message to generate the pseudorandom `r`
    ↳ value
    pub hash_prefix: [u8; 32],
}

#[cfg(feature = "zeroize")]
impl Drop for ExpandedSecretKey {
```

```
    fn drop(&mut self) {
        self.scalar.zeroize();
        self.hash_prefix.zeroize()
    }
}

#[cfg(feature = "zeroize")]
impl ZeroizeOnDrop for ExpandedSecretKey {}
```

*Figure 6: `ExpandedSecretKey` in ed25519_dalek/hazmat.rs*

Given that zeroization is already implemented for the `key` portion of the `VRFExpandedPrivateKey`, it would be prudent to zeroize the entire key for completeness.

As noted in Section 7.4 of the specification:

> The security of the ECVRF defined in this document relies on the fact that the nonce k used in the ECVRF_prove algorithm is chosen uniformly and pseudorandomly modulo q, and is unknown to the adversary. Otherwise, an adversary may be able to recover the VRF secret scalar x (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs

The nonce `k` referenced here *is not* the same nonce included in the expanded private key, but it is deterministically derived from expanded private key bytes and the VRF input. Therefore, knowledge of the expanded private key bytes may grant an advantage in guessing the ECVRF nonce.

## Recommendation

Consider porting the cited zeroization code from *ed25519_dalek*, or leveraging the `ed25519_dalek::ExpandedSecretKey` struct directly such that it is correctly zeroized.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 403 (commit `b0d467a`), the *zeroize* crate was added and the `drop()` function was implemented to explicitly zeroize both the key and the nonce, thereby fixing this finding.

Low

# VRF Verifier Will Not Reject Public Keys with Low Order

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-DHG |
| **Impact** | Undetermined | **Component** | akd_core |
| **Exploitability** | Low | **Category** | Cryptography |
| | | **Status** | Fixed |

## Impact

Failure to clearly document assumptions made by the library may have led to a "weak" VRF public key being accepted without detection, violating formally defined security requirements and potentially compromising security proofs and security guarantees of the VRF.

## Description

As part of *akd_core*, an implementation of a verifiable random function (VRF) is provided. The implemented approach is `ECVRF-EDWARDS25519-SHA512-TAI` from draft 15 of the IETF document draft-irtf-cfrg-vrf-15.

To verify a VRF proof, the specification defines `ECVRF_verify`, which may include the `validate_key` flag:

> validate_key - a boolean. An implementation MAY support only the option of validate_key = TRUE, or only the option of validate_key = FALSE, in which case this input is not needed. If an implementation supports only one option, it MUST specify which option is [sic] supports.

This corresponding implementation does not accept a `validate_key` parameter:

```
impl VRFPublicKey {
    /// Given a [`Proof`] and an input, returns whether or not the proof is valid for the
    ↳ input
    /// and public key
    pub fn verify(&self, proof: &Proof, alpha: &[u8]) -> Result<(), VrfError> {
```

*Figure 7:* `verify()` *in akd_core/src/ecvrf/ecvrf_impl.rs*

No code was identified in *akd_core* or *akd* that explicitly checks that the public key is not the identity element (or that `public_key * cofactor` is not the identity element). Furthermore, no comments or documentation specify this fact, contradicting the "MUST" requirement quoted earlier.

The specification defines `ECVRF_validate_key` as a helper function for validating a public key, which includes both generic approaches and an optimized approach for ed25519, which may be realized using `curve25519_dalek::IsIdentity`, for example.

Without the valid public key check, the VRF is not guaranteed to provide unpredictability under malicious key generation, as described in Section 7.1.3 of the specification, e.g.:

> Unpredictability under malicious key generation holds for the ECVRF if validate_key parameter given to the ECVRF_verify is TRUE

No rationale is given in the specification for leaving public key validation as an optional step, though for performance reasons it may make sense to only perform validation once globally, rather than once per VRF verification. For WhatsApp's use case specifically, it may be assumed that the VRF keypair will be generated under a high level of scrutiny, using a process that ensures a weak public key is not chosen. However, given that *akd* is being published as an open-source library, this property cannot be guaranteed, and the library itself does not make these requirements clear. It is recommended to ensure that public key validation occurs within the library such that a VRF proof using an invalid public key will not verify, or that the verify function will not be called with an invalid key.

Note that this finding is concerned with a deviation from the documented requirements and does not currently outline a specific attack against the *akd* implementation.

## Recommendation

Consider one or more of the following:

- Supporting the `validate_key` flag and using it appropriately.
- Clearly documenting the implementation's behavior with respect to public key validation. Annotating the `verify()` function with its validation behavior or assumptions is recommended.
- Performing public key validation at a higher level in the library, such as when the key is initially received or loaded, and such that `verify()` will never be called on a weak public key.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 410 (commit `aa0a856`), the function documentation for `verify()` was updated to state its behavior with respect to public key validation (i.e., that it is performed implicitly by the underlying *ed25519_dalek* library). This satisfies the "MUST" requirement, thereby fixing this finding.

Low

# VRF Hash To Curve Accepts Non-Canonical Encodings

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-KAG |
| **Impact** | Low | **Component** | akd_core |
| **Exploitability** | Low | **Category** | Cryptography |
| | | **Status** | Fixed |

## Impact

With a low probability, the VRF might have returned an incorrect output, which would not have been interoperable with other client implementations.

## Description

The VRF produces the pseudorandom edwards25519 curve point `H` by repeatedly hashing the input along with a counter value, until a sequence of bytes that matches the encoding of a curve point is obtained. This is expressed by RFC 9381, Section 5.4.1.1, which uses the `interpret_hash_value_as_a_point()` function, defined for the used ciphersuite (`ECVRF-EDWARDS25519-SHA512-TAI`) in Section 5.5 as follows:

```
*  The string_to_point function converts an octet string to a point
   on E according to the encoding specified in Section 5.1.3 of
   [RFC8032]. This function MUST output "INVALID" if the octet
   string does not decode to a point on the curve E.

*  The hash function Hash is SHA-512 as specified in [RFC6234], with
   hLen = 64.

*  The ECVRF_encode_to_curve function is as specified in
   Section 5.4.1.1, with interpret_hash_value_as_a_point(s) =
   string_to_point(s[0]...s[31]).
```

The interpretation of a 32-byte string into a curve point, as per RFC 8032, consists of interpreting the first 255 bits with little-endian convention into an integer lower than the base field modulus ($2^{255}$ - 19) and then checking whether that value is an appropriate *y* coordinate for a curve point; the corresponding *x* coordinate is computed, and optionally negated if its least significant bit does not match the last bit of the source 32-byte string. In particular, if the integer happens to fall in the $2^{255}$ - 19 to $2^{255}$ - 1 range, then the decoding process is supposed to fail:

```
1.  First, interpret the string as an integer in little-endian
    representation.  Bit 255 of this number is the least significant
    bit of the x-coordinate and denote this value x_0.  The
    y-coordinate is recovered simply by clearing this bit.  If the
    resulting value is >= p, decoding fails.
```

and similarly, if the recomputed *x* value happens to be zero but the last source bit is one, then the least significant bit of *x* is zero and negating *x* does *not* turn it into a one; RFC 8032 prescribes that such an input should also be rejected:

```
4.  Finally, use the x_0 bit to select the right square root.  If
    x = 0, and x_0 = 1, decoding fails.  Otherwise, if x_0 != x mod
    2, set x <-- p - x.  Return the decoded point (x,y).
```

The implementation of this process in *akd_core* uses the curve25519-dalek library:

```
235     let wrapped_point = CompressedEdwardsY::from_slice(&result)
236         .expect("Result hash should have a length of 32, but it does not")
237         .decompress();
```

*Figure 8: `hash_to_curve()` in akd_core/src/ecvrf/ecvrf_impl.rs*

However, the `CompressedEdwardsY::decompress()` function from the curve25519-dalek library slightly deviates from the strict RFC 8032 interpretation in that it accepts some non-canonical inputs:

- If the decoded integer is not lower than $2^{255}$ - 19, then it is implicitly reduced modulo $2^{255}$ - 19, instead of being rejected.
- If the recomputed *x* is zero and the last source bit is one, then the input is still accepted.

This behaviour is present for historical reasons and maintained by curve25519-dalek for backward compatibility[1]. In most uses of the primitive as part of, for instance, signature verification, this acceptance of non-canonical encodings is harmless; however, in some specific applications, especially consensus-related, it can have some impact[2]. In the case of *akd_core*, it may conceptually lead to the VRF implementation producing an output that is distinct from that mandated by RFC 9381, in which case a different implementation, implemented from the RFC, would fail to interoperate (e.g. a third-party validating client for the key transparency feature would reject some membership proofs). In practice, the issue is unlikely to ever happen, because the candidate encoding for `H` is obtained as a hash output (with SHA-512, truncated to 32 bytes) and there are only 26 sequences of 32 bytes that are accepted by `decompress()` and yet invalid as per RFC 8032. The probability of obtaining an input value that yields a non-canonical point encoding through SHA-512 is about $2^{-251.3}$, which is negligible; finding such a value on purpose would require a preimage attack on truncated SHA-512 (with 26 targets), which is considered infeasible.

## Recommendation

For strict compliance to RFC 8032, non-canonical inputs may be rejected. Some non-canonical inputs lead to a point of low order, which should also be rejected (see finding "VRF Hash to Curve Function May Incorrectly Return the Identity Point"); the list of non-canonical inputs that are accepted by curve25519-dalek and yield a high-order point corresponds to *y* coordinates equal to $2^{255}$ - *i*, for integers *i* equal to 1, 3, 4, 5, 9, 10, 13, 14, 15 or 16 (for each such *y*, there are two matching encodings, for both values of the sign-of-*x* bit). An implementation fully conforming to steps 5.c and 5.d of the `ECVRF_encode_to_curve` specification defined in Section 5.4.1.1 of RFC 9381 could thus follow the following process:

1. If the input bytes are such that bytes 1 to 30 have value 255, byte 31 has value 255 or 127, and byte 0 has value 256 - *i* for value *i* in the (1, 3, 4, 5, 9, 10, 13, 14, 15, 16) list, then the encoding is invalid.
2. Use `CompressedEdwardsY::decompress()` to tentatively decode the input into a point.
3. Multiply the obtained point by the cofactor (`mul_by_cofactor()` function call).
4. If the result is the identity point on the curve, reject the encoding.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*, lines 235-237

1. https://hdevalence.ca/blog/2020-10-04-its-25519am
2. https://eprint.iacr.org/2020/1244

## Retest Results

### 2023-09-20 – Fixed

As part of PR 401 (commit 78a5fd5), the following check was added to the interpret_hash_value_as_a_point() function:

```
    let is_invalid = hash[1..=30].iter().all(|b| *b == 255)
        && (hash[31] == 255 || hash[31] == 127)
        && [1u8, 3, 4, 5, 9, 10, 13, 14, 15, 16].contains(&((256u16 - hash[0] as u16) as u8));
    if is_invalid {
        return None;
    }
```

Additionally, a check that multiplying the decoded point by the cofactor does not result in the identity point was added to the encode_to_curve() function. Thus, this finding is considered *fixed*.

Low

# Dangerous Public API Functions

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-FRK |
| **Impact** | High | **Component** | akd, akd_core |
| **Exploitability** | Undetermined | **Category** | Cryptography |
| | | **Status** | Fixed |

## Impact

Some functions were made public for test purposes but could not be safely used by applications.

## Description

Client applications using the *akd_core* library for verifying proofs should normally use the functions `lookup_verify()` (for lookup proofs) or `key_history_verify()` (for key history proofs). However, the *akd_core/src/verify/base.rs* file also defines two public functions called `verify_membership()` and `verify_nonmembership()`:

```
/// Verify the membership proof
pub fn verify_membership<TC: Configuration>(
    root_hash: Digest,
    proof: &MembershipProof,
) -> Result<(), VerificationError> {
// <SNIP>

/// Verifies the non-membership proof with respect to the root hash
pub fn verify_nonmembership<TC: Configuration>(
    root_hash: Digest,
    proof: &NonMembershipProof,
) -> Result<(), VerificationError> {
```

*Figure 9: `verify_membership()` and `verify_nonmembership()` in akd_core/src/verify/base.rs*

Contrary to `lookup_verify()` and `key_history_verify()`, these functions do *not* include the validation of the VRF output; but that validation is necessary to achieve the expected security features of the membership and non-membership proofs. The names and documentations of these two functions do not point out the lack of the VRF output validation. The VRF validation cannot, in any case, be performed as an extra step by the calling application, because the `verify_label()` function (defined in the same file) is private, and callable only through one of `verify_existence()`, `verify_existence_with_val()` or `verify_nonexistence()`, which only have crate visibility (`pub(crate)`). Thus, the `verify_membership()` and `verify_nonmembership()` functions are a dangerous API, that must not be used by applications, but is not documented as such.

## Recommendation

The `verify_membership()` and `verify_nonmembership()` functions seem to be public so that they may be invoked from test code located in the *akd* crate (in the *append_only_zks.rs* file). The `verify_membership()` and `verify_nonmembership()` functions thus cannot be made private or crate-private without breaking this test code. Instead, they should be prominently documented as being meant for tests only, e.g. by making the functions private and adding public wrappers called `verify_membership_fortestsonly()` and `verify_nonmembership_fortestsonly()`.

There is an existing open issue (#265, from November 2022) about limiting visibility on many objects in the API.

## Location

*akd_core/src/verify/base.rs*, lines 27 and 66

## Retest Results

### 2023-09-20 – Fixed

As part of PR 409 (commit `c10a7fa`), the visibility of the `verify_membership()` and `verify_nonmembership()` functions was limited to the crate by using the `pub(crate)` designation, and new functions `verify_membership_for_tests_only()` and `verify_nonmembership_for_tests_only()` that are clearly documented as being test-only functionality were added. As such, this finding is considered *fixed*.

Low

# Malformed Input May Crash Client Applications

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-Y7E |
| **Impact** | Low | **Component** | akd_core |
| **Exploitability** | High | **Category** | Denial of Service |
| | | **Status** | Fixed |

## Impact

Maliciously crafted lookup or key history proofs may have induced the client application to panic upon decoding.

## Description

Lookup and key history proofs are encoded using protobuf, with types specified in *akd_core/src/proto/specs/types.proto*. In particular, lookup and key history proofs include members of type `MembershipProof` and `NonMembershipProof`, both of which including `NodeLabel` elements. The `NodeLabel` type is specified as follows:

```
message NodeLabel {
    optional bytes label_val = 1;
    optional uint32 label_len = 2;
}
```

*Figure 10: akd_core/src/proto/specs/types.proto*

The protobuf-generated code defines a container Rust type (`akd_core::proto::specs::type::NodeLabel`), and the crate defines another `NodeLabel` type (in `akd_core::types::node_label::NodeLabel`, ultimately reexported at the top-level of the crate as `akd_core::NodeLabel`) which is the one used for all computations. The latter type contains a 32-byte value, and a 32-bit length:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct NodeLabel {
    #[cfg_attr(
        feature = "serde_serialization",
        serde(serialize_with = "bytes_serialize_hex")
    )]
    #[cfg_attr(
        feature = "serde_serialization",
        serde(deserialize_with = "bytes_deserialize_hex")
    )]
    /// Stores a binary string as a 32-byte array of `u8`s
    pub label_val: [u8; 32],
    /// len keeps track of how long the binary string is in bits
    pub label_len: u32,
}
```

*Figure 11: `NodeLabel` in akd_core/src/types/node_label/mod.rs*

The decoding process entails invoking the protobuf decoder, then converting the protobuf-generated `NodeLabel` to the other `NodeLabel` type, through the `try_from()` function:

```
    fn try_from(input: &specs::types::NodeLabel) -> Result<Self, Self::Error> {
        require!(input, has_label_len);
        require!(input, has_label_val);
        let label_val = decode_minimized_label(input.label_val());
```

```

        Ok(Self {
            label_len: input.label_len(),
            label_val,
        })
    }
```

Figure 12: `try_from()` in akd_core/src/proto/mod.rs

The `decode_minimized_label()` function pads the input bytes to 32 bytes (with extra bytes of value zero), in case the encoded value is shorter:

```
fn decode_minimized_label(v: &[u8]) -> [u8; 32] {
    let mut out = [0u8; 32];
    out[..v.len()].copy_from_slice(v);
    out
}
```

Figure 13: `decode_minimized_label()` in akd_core/src/proto/mod.rs

If the input bytes (from the `label_val` field of the protobuf object) happens to contain a sequence of bytes strictly *longer* than 32 bytes, then the slice extraction `out[..v.len()]` will trigger a panic, since `out[]` has length 32. The consequences of a panic are usually immediate termination of the calling thread, then of the whole application process. The consequences seem limited to that kind of denial-of-service.

We may also note that `label_len` is not validated. The `NodeLabel` type is used to represent two conceptually different kinds of objects: node labels, and node label prefixes. For a full label, the length is 256; for a prefix, the length specifies how many bits are relevant. The decoding process does not check that `label_len` is in the 0 to 256 range; it does not check either whether bits beyond `label_len` are zero or not, even though the `cmp()` function on `NodeLabel` instances and the equality test on such instances (implicitly generated with the `Eq` derivation attribute) take all 256 bits into account. Out-of-range `label_len` values may induce further panics (especially in `NodeLabel::get_bit_at()`). Extra non-zero bits beyond the advertised `label_len` in a prefix may also induce unexpected comparison results, though this does not seem to be exploitable in the proof verification code. Notably, the label verification (VRF output validation) explicitly checks that the recomputed value, with a 256-bit length, exactly matches the input value, length included:

```
    if NodeLabel::new(output.to_truncated_bytes(), 256) != node_label {
        return Err(VerificationError::Vrf(VrfError::Verification(
            "Expected first 32 bytes of the proof output did NOT match the supplied label"
                .to_string(),
        )));
    }
```

Figure 14: `verify_label()` in akd_core/src/verify/base.rs

## Recommendation

`NodeLabel::try_from()` should perform explicit validation of the input value, and report an `Error` instead of panicking if the value is incorrect:

- Check that the source `input_val` length is no more than 32 bytes.
- Check that the source `label_len` is in the 0 to 256 range.
- Verify that all source bits beyond `label_len` are zero.

## Location

*akd_core/src/proto/mod.rs*, lines 128-137

## Retest Results

### 2023-09-20 – Fixed

As part of PR 407 (commit `bab0b5e`), the function `NodeLabel::try_from()` has been updated to check that `input_val` contains no more than 32 bytes, and that `label_len` is bounded by 256, and to return an error otherwise.

Regarding the source bits beyond `label_len`, the WhatsApp team clarified that in some cases such as for the `empty_label`, the extra bits may not all be 0. To account for this case, the code was updated to ensure that the source bits beyond `label_len` are handled in a consistent manner, and that comparison functions such as `get_prefix_ordering()` only consider the first `label_len` bits when comparing prefixes. Additional context on the handling of these bits within the codebase is provided in finding "Potentially Confusing Behavior for NodeLabels".

As such, this finding is considered *fixed*.

Low

# Malformed VRF Proof May Crash Client Applications

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-JJ4 |
| **Impact** | Low | **Component** | akd_core |
| **Exploitability** | High | **Category** | Data Validation |
| | | **Status** | Fixed |

## Impact

Malformed VRF proofs can lead the client (verifier) application to panic, causing a denial of service condition.

## Description

The `akd_core` crate handles a number of VRF proofs such as existence, marker, and freshness proofs. These proofs are serialized as protobuf binary types, as illustrated below in file *akd_core/src/proto/specs/types.proto* for lookup proofs:

```
message LookupProof {
    optional uint64 epoch = 1;
    optional bytes value = 2;
    optional uint64 version = 3;
    optional bytes existence_vrf_proof = 4;
    optional MembershipProof existence_proof = 5;
    optional bytes marker_vrf_proof = 6;
    optional MembershipProof marker_proof = 7;
    optional bytes freshness_vrf_proof = 8;
    optional NonMembershipProof freshness_proof = 9;
    optional bytes commitment_nonce = 10;
}
```

*Figure 15: akd_core/src/proto/specs/types.proto*

These proofs are deserialized using the `try_from()` method of the `Proof` structure, in file *ecvrf_impl.rs* of *akd_core*:

```
impl TryFrom<&[u8]> for Proof {
    type Error = VrfError;

    fn try_from(bytes: &[u8]) -> Result<Proof, VrfError> {
        let mut c_buf = [0u8; 32];
        c_buf[..16].copy_from_slice(&bytes[32..48]);
        let mut s_buf = [0u8; 32];
        s_buf.copy_from_slice(&bytes[48..]);

        let pk_point = match CompressedEdwardsY::from_slice(&bytes[..32])
            .expect("Byte string should be of length 32, but it is not")
            .decompress()
        {
            Some(pt) => pt,
            None => {
                return Err(VrfError::PublicKey(
                    "Failed to decompress public key into Edwards Point".to_string(),
                ))
```

```
            }
        };

        Ok(Proof {
            gamma: pk_point,
            c: ed25519_Scalar::from_bytes_mod_order(c_buf),
            s: ed25519_Scalar::from_bytes_mod_order(s_buf),
        })
    }
}
```

*Figure 16: akd_core/src/ecvrf/ecvrf_impl.rs*

Method `try_from()` calls highlighted method `copy_from_slice()` in the code snippet above. In the first call, the slice extraction will panic if the input `bytes` value is shorter than 48 bytes; in the second call, the `copy_from_slice()` function will panic if the source and destination slices have different lengths, i.e. if `bytes[48..]` does not have length exactly 32 bytes. The implementation does not try to validate the size of the proof before deserializing it. A malformed proof such as an empty `existence_vrf_proof` field in a `LookupProof` structure would cause such panic.

## Recommendation

The implementation should validate that the proof size size is correct (80 bytes) before accessing it. If the size is incorrect, it should return an error.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*

## Retest Results

### 2023-10-12 – Fixed

As part of PR 416 (commit `54c002b`), the function `Proof::try_from()` has been updated to check that `bytes` contains exactly 80 bytes (`PROOF_LENGTH`), and to return an error otherwise.

Low

# Malformed History Proof May Crash Client Applications

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E008327-9NP |
| **Impact** | Low | **Component** | akd_core |
| **Exploitability** | High | **Category** | Denial of Service |
| | | **Status** | Fixed |

## Impact

Malformed proofs can lead the client (verifier) application to panic, causing a denial of service condition.

## Description

Key history proofs are encoded using protobuf, with types specified in *akd_core/src/proto/specs/types.proto*. These proofs are parsed into the `HistoryProof` struct:

```
pub struct HistoryProof {
    /// The update proofs in the key history
    pub update_proofs: Vec<UpdateProof>,
    /// VRF Proofs for the labels of the values until the next marker version
    pub until_marker_vrf_proofs: Vec<Vec<u8>>,
    /// Proof that the values until the next marker version did not exist at this time
    pub non_existence_until_marker_proofs: Vec<NonMembershipProof>,
    /// VRF proofs for the labels of future marker entries
    pub future_marker_vrf_proofs: Vec<Vec<u8>>,
    /// Proof that future markers did not exist
    pub non_existence_of_future_marker_proofs: Vec<NonMembershipProof>,
}
```

*Figure 17: `HistoryProof` struct in akd_core/src/types/mod.rs*

These proofs are deserialized in using the `try_from()` function, after which the verification proceeds in the `key_history_verify()` function. As part of this process, the next marker node is computed, and the non-existence proofs of future entries up to the next marker, and for all future markers are verified:

```
    // Get the least and greatest marker entries for the current version
    let next_marker = crate::utils::get_marker_version_log2(last_version) + 1;
    let final_marker = crate::utils::get_marker_version_log2(current_epoch);

    // ***** Future checks below ***************************
    // Verify the non-existence of future entries, up to the next marker
    for (i, version) in (last_version + 1..(1 << next_marker)).enumerate() {
        verify_nonexistence::<TC>(
            vrf_public_key,
            root_hash,
            &akd_label,
            VersionFreshness::Fresh,
            version,
            &proof.until_marker_vrf_proofs[i],
            &proof.non_existence_until_marker_proofs[i],
        ).map_err(|_|
```

```
127             VerificationError::HistoryProof(format!("Non-existence of next few proof of
                ↪ user {:?}'s version {:?} at epoch {:?} does not verify",
128                     &akd_label, version, current_epoch)))?;
129     }
130 
131     // Verify the VRFs and non-membership proofs for future markers
132     for (i, pow) in (next_marker..final_marker + 1).enumerate() {
133         let version = 1 << pow;
134         verify_nonexistence::<TC>(
135             vrf_public_key,
136             root_hash,
137             &akd_label,
138             VersionFreshness::Fresh,
139             version,
140             &proof.future_marker_vrf_proofs[i],
141             &proof.non_existence_of_future_marker_proofs[i],
142         ).map_err(|_|
143             VerificationError::HistoryProof(format!("Non-existence of future marker proof
                ↪ of user {akd_label:?}'s version {version:?} at epoch {current_epoch:?} does
                ↪ not verify")))?;
144     }
```

Figure 18: `key_history_verify()` in *akd_core/src/verify/history.rs*

However, note that if insufficiently many proofs are provided in the `until_marker_vrf_proofs` or `non_existence_until_marker_proofs` fields, an out of bounds access will occur on line 124 or 125, which will cause a panic. Additionally, if more non-existence proofs than required are provided, this will not be detected and will be accepted as valid. Similar issues are present for the verification of the VRFs and non-membership proofs for future markers on lines 140 and 141.

## Recommendation

Check that `(1 << next_marker) - last_version - 1` is equal to `proof.non_existence_until_marker_proofs.len()` and `proof.until_marker_vrf_proofs.len()` before validating each proof in the `non_existence_until_marker_proofs` field. Similarly, check that `final_marker + 1 - next_marker` is equal to `proof.non_existence_of_future_marker_proofs.len()` and `proof.future_marker_vrf_proofs.len()` before validating each proof in the `non_existence_of_future_marker_proofs` field.

If any of these checks do not pass, an error should be returned instead.

## Location

*akd_core/src/verify/history.rs*

## Retest Results

### 2023-10-12 – Fixed

As part of PR 417 (commit `04116e7`), the function `key_history_verify()` has been updated to check that the `until_marker_vrf_proofs`, `non_existence_until_marker_proofs`, `future_marker_vrf_proofs` and `non_existence_of_future_marker_proofs` fields have the expected number of elements. As such, this finding is considered *fixed*.

Info

# VRF Draft Specification Now Published as RFC 9381

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-7Q6 |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Patching |
| | | **Status** | Fixed |

## Impact

Citing a draft specification instead of the release specification may have suggested that the implementation was not complete or compliant with the released specification.

## Description

As part of *akd_core*, an implementation of a verifiable random function (VRF) is provided. The implemented approach is `ECVRF-EDWARDS25519-SHA512-TAI` from draft 15 of the IETF document draft-irtf-cfrg-vrf-15.

On August 23, 2023, while this review was taking place, the draft was formally published as RFC 9381. Changes from draft 15 to RFC 9381 appear to be focused on editorial issues, and no changes in the published RFC were identified that conflict with reviewed *ecvrf* implementation. Therefore, it is likely that the sub-module could be revised to cite the published RFC in place of the currently cited draft:

```
//! This module implements an instantiation of a verifiable random function known as
//! [ECVRF-ED25519-SHA512-TAI](https://tools.ietf.org/html/draft-irtf-cfrg-vrf-15).
```

*Figure 19: akd_core/src/ecvrf/mod.rs*

Furthermore, the implemented ciphersuite is called `ECVRF-EDWARDS25519-SHA512-TAI` in the specification. This has no practical consequence since for hash computation purposes ciphersuites are identified by symbolic numeric identifiers rather than character strings.

This finding is purely informational, as the changes made in the final RFC do not affect the implementation of *ecvrf*.

## Recommendation

Review the recently published RFC 9381 and update references within the code to point to the finalized document. A diff between the two versions can be viewed at https://author-tools.ietf.org/iddiff?url1=draft-irtf-cfrg-vrf-15&url2=rfc9381&difftype=--hwdiff.

## Location

*akd_core/src/ecvrf/mod.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 401 (commit `78a5fd5`), the algorithm name and document reference were updated to cite `ECVRF-EDWARDS25519-SHA512-TAI from RFC9381` instead of the draft document, thereby fixing this finding.

Info

# Incorrect Function Documentation for `get_commitment_nonce()` and `compute_fresh_azks_value()`

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-RKB |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Other |
| | | **Status** | Fixed |

## Impact

Incorrect function documentation may have misled a user of the library, potentially leading to implementation errors in the future.

## Description

### WhatsAppV1Configuration

The documentation for `get_commitment_nonce()` does not accurately reflect what is computed in the function:

```
72      /// Used by the server to produce a commitment nonce for an AkdLabel, version, and
        ↳ AkdValue.
73      /// Computes nonce = H(commitment key || label)
74      fn get_commitment_nonce(
75          commitment_key: &[u8],
76          label: &NodeLabel,
77          version: u64,
78          value: &AkdValue,
79      ) -> Digest {
80          Self::hash(
81              &[
82                  commitment_key,
83                  &label.to_bytes(),
84                  &version.to_be_bytes(),
85                  &i2osp_array(value),
86              ]
87              .concat(),
88          )
89      }
```

*Figure 20: `get_commitment_nonce` in akd_core/src/configuration/whatsapp_v1.rs*

The hash includes the `version` and `value` parameters, suggesting that the function documentation on line 73 should be:

```
/// Computes nonce = H(commitment key || label || version || value)
```

### ExperimentalConfiguration

The documentation for function `compute_fresh_azks_value()` specifies that the nonce is computed as the hash of 4 values:

```
    /// Used by the server to produce a commitment for an AkdLabel, version, and AkdValue
    ///
    /// nonce = H(commitment_key, label, version, i2osp_array(value))
```

```
    /// commitment = H(i2osp_array(value), i2osp_array(nonce))
    ///
    /// The nonce value is used to create a hiding and binding commitment using a
    /// cryptographic hash function. Note that it is derived from the label, version, and
    /// value (even though the binding to value is somewhat optional).
    ///
    /// Note that this commitment needs to be a hash function (random oracle) output
    fn compute_fresh_azks_value(
        commitment_key: &[u8],
        label: &NodeLabel,
        version: u64,
        value: &AkdValue,
    ) -> AzksValue {
        let nonce = Self::get_commitment_nonce(commitment_key, label, version, value);
        AzksValue(Self::hash(
            &[i2osp_array(value), i2osp_array(&nonce)].concat(),
        ))
    }
```

Figure 21: `compute_fresh_azks_value()` in *akd_core/src/configuration/experimental.rs*

However, the `get_commitment_nonce()` function for an experimental configuration only computes `H(commitment key || label)`:

```
    /// Used by the server to produce a commitment nonce for an AkdLabel, version, and AkdValue.
    /// Computes nonce = H(commitment key || label)
    fn get_commitment_nonce(
        commitment_key: &[u8],
        label: &NodeLabel,
        _version: u64,
        _value: &AkdValue,
    ) -> Digest {
        Self::hash(&[commitment_key, &label.to_bytes()].concat())
    }
```

Figure 22: `compute_fresh_azks_value()` in *akd_core/src/configuration/experimental.rs*

The documentation should accurately reflect the behavior of the implementation.

## Recommendation

Revise the documentation to accurately reflect the implemented behavior.

## Location

*akd_core/src/configuration/whatsapp_v1.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 404 (commit `bf7eefd`), the identified incorrect comments were revised to match the implemented behavior, thereby fixing this finding.

Info

# The `hash_to_curve()` Function Should be Renamed `encode_to_curve()`

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-CX4 |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Other |
| | | **Status** | Fixed |

## Impact

Inaccurate function names may have misled developers about the behavior of a function.

## Description

The `hash_to_curve()` function implements the `ECVRF_encode_to_curve_try_and_increment()` function defined in draft-irtf-cfrg-vrf-15. However, this function does not define a uniform mapping to curve points, and in the past was renamed from `hash_to_curve` to `encode_to_curve` in the specification to align itself with the nomenclature used in RFC 9380.

## Recommendation

Consider renaming this function to align with the current specification, or adding a comment clarifying that this should not be used as a generic-purpose hash function.

## Location

*akd_core/src/ecvrf/ecvrf_impl.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 401 (commit `78a5fd5`), the `hash_to_curve()` function was renamed to `encode_to_curve()`, thereby fixing this finding.

Info

# Improved Error Messages When Auditing History Proofs

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-9JD |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Other |
| | | **Status** | Fixed |

## Impact

Poorly defined error messages may have hindered future debugging efforts or affected the reputation of the project.

## Description

The function `key_history_verify()` contains error messages that are not aligned with the tone and style of other messages in the library:

```
                return Err(VerificationError::HistoryProof(format!("Why did you give me
                ↳ consecutive update proofs without version numbers decrementing by 1? Version
                ↳ {} = {}; version {} = {}",
                count, proof.update_proofs[count].version,
                count-1, proof.update_proofs[count-1].version
                )));
...
                return Err(VerificationError::HistoryProof(format!(
                    "Why are your versions decreasing in updates and epochs not?!,
                    epoch = {}, previous epoch = {}",
                    update_proof.epoch, previous_update_epoch
                )));
```

*Figure 23: `key_history_verify()` in akd_core/src/verify/history.rs*

## Recommendation

For consistency, it is recommended to use more neutral, informative statements to the caller, which would better align with the rest of the library; e.g.:

- `"No update proofs included in the proof of user {akd_label:?} at epoch {current_epoch:?}!"` (line 63).
- `"Non-existence of next few proof of user {:?}'s version {:?} at epoch {:?} does not verify"` (line 127).

## Location

*akd_core/src/verify/history.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 404 (commit `bf7eefd`), the identified error messages were revised to match the tone and style of the rest of the library, thereby fixing this issue.

Info

# Minor Optimization When Computing Longest Prefix

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-CUF |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Other |
| | | **Status** | Fixed |

## Impact

The implemented approach may have been marginally slower due to a missing early abort condition.

## Description

The function `get_longest_common_prefix()` could be optimized to return early when both labels are empty, as this condition does not rely on any preceding intermediate value within the function:

```
    /// Takes as input a pointer to the caller and another [NodeLabel],
    /// returns a NodeLabel that is the longest common prefix of the two.
    pub fn get_longest_common_prefix<TC: Configuration>(&self, other: NodeLabel) -> Self {
        let shorter_len = if self.label_len < other.label_len {
            self.label_len
        } else {
            other.label_len
        };

        let mut prefix_len = 0;
        while prefix_len < shorter_len
            && self.get_bit_at(prefix_len) == other.get_bit_at(prefix_len)
        {
            prefix_len += 1;
        }

        let empty_label = TC::empty_label();
        if *self == empty_label || other == empty_label {
            return empty_label;
        }
        self.get_prefix(prefix_len)
    }
```

*Figure 24: `get_longest_common_prefix()` in akd_core/src/types/node_label/mod.rs*

The highlighted lines could be moved to the top of the function.

## Recommendation

Refactor the function to return early when both labels are empty.

## Location

*akd_core/src/types/node_label/mod.rs*

## Retest Results

### 2023-09-20 – Fixed

As part of PR 408 (commit `9280352`), the identified empty label check was moved to the start of the function, thereby fixing this finding.

Info

# Potentially Confusing Behavior for NodeLabels

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E008327-MPR |
| **Impact** | None | **Component** | akd_core |
| **Exploitability** | None | **Category** | Other |
| | | **Status** | Fixed |

## Impact

Inconsistent handling of bits beyond the set number of bits in `NodeLabel` structures may have caused future issues or confusion.

## Description

The `NodeLabel` structure contains a 32-byte array, and a length value that indicates the length of the binary string stored within the array, in bits:

```
    /// Stores a binary string as a 32-byte array of `u8`s
    pub label_val: [u8; 32],
    /// len keeps track of how long the binary string is in bits
    pub label_len: u32,
```

*Figure 25: `compute_fresh_azks_value()` in akd_core/src/types/node_label.rs*

Bits beyond the set `label_len` bits are generally ignored for the purposes of organizing the Merkle tree that contains the `NodeLabel` s. However, the additional bits in the array may be included when hashing the `NodeLabel` s, and so they are sometimes specified, such as in the case of the `empty_label` , which has `label_len = 0` and `label_val = [1u8;32]` (in the `whatsapp_v1` configuration).

In the codebase, a number of functions defined to interact with `NodeLabel` structures provide logic that operates on the bits beyond `label_len` bits in potentially unexpected ways:

- The function `get_bit_at()` is documented to `/// Returns the bit at a specified index, and a 0 on an out-of-range index` . This may cause confusion if it is called on values beyond `label_len` that are set, such as the set bits in an `empty_label` node. Note that `get_bit_at()` does not currently get called on out-of-bound bits within the codebase.
- The function `get_prefix(len)` returns a prefix of length `len` of the current `NodeLabel` and sets all other bits of the `label_val` array to `0` . This is called in the `get_prefix_ordering()` function:

```
/// Gets the prefix ordering of other with respect to self, if self is a prefix of
↳ other.
/// If self is not a prefix of other, then this returns [PrefixOrdering::Invalid].
pub fn get_prefix_ordering(&self, other: Self) -> PrefixOrdering {
    if self.get_len() >= other.get_len() {
        return PrefixOrdering::Invalid;
    }
    if other.get_prefix(self.get_len()) != *self {
        return PrefixOrdering::Invalid;
    }
    PrefixOrdering::from(other.get_bit_at(self.get_len()))
}
```

*Figure 26: akd_core/src/types/node_label/mod.rs*

Note that if `self` is a 0-length array with some “out-of-bound” bits set to 1, such as the `empty_label`, this will return `PrefixOrdering::Invalid` despite the fact a 0-length label is a prefix of every label, which may be confusing behavior. This scenario does not currently seem to occur within the codebase, as the `get_prefix_ordering` function only gets called on computed prefixes, and the `empty_label` node is a special leaf node that can only occur as a child of the root node.

However, the behavior of these functions on edge values is not very well documented and may cause confusion if they are used in different settings in the future.

## Recommendation

Ensure that the handling of the out-of-bound bits is consistent throughout the codebase, and does not cause confusion. In particular,

- Update `get_bit_at()` to return either the correct value for out-of-bound bits, or an error if this behavior is not supported
- Update `get_prefix_ordering()` to ignore out-of-bound bits when determining the prefix ordering

## Retest Results

### 2023-09-20 – Fixed

As part of PR 407 (commit `bab0b5e`), the `get_bit_at()` function was updated to return an error for any index beyond the `label_len`. Additionally, the `get_prefix_ordering()` function has been updated to ignore out-of-bound bits when comparing the prefixes:

```
        if other.get_prefix(self.get_len()) != self.get_prefix(self.get_len()) {
            // Note: we check self.get_prefix(self.get_len()) here instead of just *self
            // because equality checks for a [NodeLabel] do not ignore the bits of label_val set
            // beyond label_len.
            return PrefixOrdering::Invalid;
        }
```

As such, this finding is considered *fixed*.

# 4 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

## Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

### Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

| Rating | Description |
|---|---|
| Critical | Implies an immediate, easily accessible threat of total compromise. |
| High | Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach. |
| Medium | A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application. |
| Low | Implies a relatively minor threat to the application. |
| Informational | No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding. |

### Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

| Rating | Description |
|---|---|
| High | Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level. |
| Medium | Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information. |
| Low | Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security. |

### Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

| Rating | Description |
|---|---|
| High | Attackers can unilaterally exploit the finding without special permissions or significant roadblocks. |

| Rating | Description |
|---|---|
| **Medium** | Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding. |
| **Low** | Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely. |

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

| Category Name | Description |
|---|---|
| **Access Controls** | Related to authorization of users, and assessment of rights. |
| **Auditing and Logging** | Related to auditing of actions, or logging of problems. |
| **Authentication** | Related to the identification of users. |
| **Configuration** | Related to security configurations of servers, devices, or software. |
| **Cryptography** | Related to mathematical protections for data. |
| **Data Exposure** | Related to unintended exposure of sensitive information. |
| **Data Validation** | Related to improper reliance on the structure or values of data. |
| **Denial of Service** | Related to causing system failure. |
| **Error Reporting** | Related to the reporting of error conditions in a secure fashion. |
| **Patching** | Related to keeping software up to date. |
| **Session Management** | Related to the identification of authenticated users. |
| **Timing** | Related to race conditions, locking, or order of operations. |

# 5 Engagement Notes

This section consists of notes and observations from the review that do not represent security issues, but that may be of interest to the team at Meta.

### Suboptimal VRF Computations

The handling of VRF public keys (in the proof verifier, i.e. client-side) performs some computations redundantly. Namely, in *akd_core/src/ecvrf/ecvrf_impl.rs*, when a public key is decoded from bytes, `CompressedEdwardsY::decompress()` is called (line 169); the process involves an inverse square root computation, and is relatively expensive (though less expensive than a curve point multiplication by a scalar). After some validation, `ed25519_PublicKey::from_bytes()` is invoked on the input; this curve25519-dalek call will internally use `CompressedEdwardsY::decompress()` again, on the same input as previously. Then, whenever the public key is used in a VRF output validation, the public key bytes are decompressed a third time (*akd_core/src/ecvrf/ecvrf_impl.rs*, line 196). In total, if $n$ VRF output validations are performed against the same VRF public key, then the public key bytes are decompressed $n+2$ times, instead of just once.

Ideally, the `VRFPublicKey` type would include a cached copy of the decoded curve point, so that multiple decompressions are not needed. The `ed25519_PublicKey` type is an alias on curve25519-dalek's `ed25519_dalek::VerifyingKey`, which *already* contains such a cached copy, but that point is not made accessible to callers (it is only `pub(crate)`) and *akd_core* uses `ed25519_PublicKey` only as a generic container for 32 bytes.

### Client Response

During the retest, Meta provided the following response to the above note:

> Unfortunately, I don't think we have a good way of addressing this without expanding the internals of ed25519_dalek::VerifyingKey into ecvrf_impl.rs. Ideally, the ed25519_dalek library would expose a function to obtain underlying EdwardsPoint from VerifyingKey, but it does not at the moment. Probably what we will opt to do is wait until this is introduced, and for the moment just take the performance hit of doing decompression per call to verify().

# 6 Hashing Strategy for AKD and Merkle-Patricia Trees

As part of this review, the Merkle-Patricia tree constructions utilized in *akd* were compared against their corresponding academic references to ensure that any differences or modifications did not introduce new vulnerabilities. This section summarizes these differences but does not identify any attack or vulnerability in the implemented approaches.

### Overview

Per the *akd* library *README.md*:

> This implementation is based off of the protocols described in SEEMless, with ideas incorporated from Parakeet.

SEEMless builds a *verifiable key directory* (VKD) from *append-only zero-knowledge sets* (aZKS), which are in turn constructed from *append-only strong accumulators* (aSA). The security of the underlying aSA primitive is based on previous work in *Authentic Time-Stamps for Archival Storage*[3] by Oprea and Bowers.

At their core, these approaches rely on a combination of Merkle (hash) trees and Patricia (prefix) trees. It is well known that a naive construction of a Merkle tree may be susceptible to a second-preimage attack if the hash function used on leaves is not distinct from the hash function used on internal nodes. The Merkle-Patricia construction leveraged by the above protocols is shown to be secure if the chosen hash function provides “everywhere second-preimage resistance”, which is a theoretically weaker property than full collision resistance. The aSA construction cited above chooses to specify the stronger and more widely cited property of full collision resistance as the necessary property of the hash function.

The hashing strategy utilized is as follows, where `DS` is a domain separator string:

- Leaf nodes: `H( DS | label | value )`
    - Parakeet extends this as: `H( H(label | value) | epoch )`
- Internal nodes: `H( DS | label | left.hash | right.hash | left.label | right.label )`
    - Parakeet specifies: `H( left.hash | right.hash | left.label | right.label )`

The *akd* library provides two implementations of the above via its two supported configurations: `WhatsAppV1Configuration` and `ExperimentalConfiguration`.

### WhatsApp Configuration

The library defines `WhatsAppV1Configuration` to implement the chosen hashing strategy. For leaf nodes, the hash function `H` is implemented as vanilla BLAKE3:

```
    fn hash(item: &[u8]) -> crate::hash::Digest {
        ::blake3::hash(item).into()
    }
```

*Figure 27: `hash()` in akd_core/src/configuration/whatsapp_v1.rs*

For a leaf node, the resulting hash is computed as `H( H(label | value) | epoch )` via the functions `generate_commitment_from_nonce_client()`, `hash_leaf_with_value()`, and `hash_leaf_with_commitment()`. Notably, this **does not** contain a domain separation string.

3. Alina Oprea and Kevin D Bowers. 2009. Authentic time-stamps for archival storage. In European Symposium on Research in Computer Security. Springer, 136–151; available via *ePrint*.

For internal nodes, the resulting hash is computed as `H( H(left.hash | left.label) | H(right.hash | right.label) )` via the following:

```
    /// Computes the parent hash from the children hashes and labels
    fn compute_parent_hash_from_children(
        left_val: &AzksValue,
        left_label: &[u8],
        right_val: &AzksValue,
        right_label: &[u8],
    ) -> AzksValue {
        AzksValue(Self::hash(
            &[
                Self::hash(&[left_val.0.to_vec(), left_label.to_vec()].concat()),
                Self::hash(&[right_val.0.to_vec(), right_label.to_vec()].concat()),
            ]
            .concat(),
        ))
    }
```

*Figure 28: `compute_parent_hash_from_children()` in akd_core/src/configuration/whatsapp_v1.rs*

Notably, this differs from the generic aSA definition in its lack of domain separation and lack of inclusion of the internal node's label, as well as using an HMAC-like construction instead of concatenation.

### Experimental Configuration

The `ExperimentalConfiguration` provides an alternate hashing strategy. Notably, the leveraged hash function `H` is implemented using BLAKE3 with a domain separator:

```
    fn hash(item: &[u8]) -> crate::hash::Digest {
        // Hash(domain label || item)
        let mut hasher = blake3::Hasher::new();
        hasher.update(L::domain_label());
        hasher.update(item);
        hasher.finalize().into()
    }
```

*Figure 29: `hash()` in akd_core/src/configuration/experimental.rs*

Although the Experimental configuration leverages the same hashing approach as the WhatsApp configuration, the inclusion of the domain separator in every call to the hash function results in a leaf hash computed as `H( DS | H(DS | label | value) | epoch )`.

For internal nodes, the resulting hash is computed as `H( DS | left.hash | left.value | right.hash | right.value )` via the following:

```
    /// Computes the parent hash from the children hashes and labels
    fn compute_parent_hash_from_children(
        left_val: &AzksValue,
        left_label: &[u8],
        right_val: &AzksValue,
        right_label: &[u8],
    ) -> AzksValue {
        AzksValue(Self::hash(
            &[&left_val.0, left_label, &right_val.0, right_label].concat(),
        ))
    }
```

*Figure 30: `compute_parent_hash_from_children()` in akd_core/src/configuration/experimental.rs*

This differs from the generic aSA definition by not including the label of the internal node, as well as swapping the order of some hash inputs.

### Summary

To summarize the hashing approaches used in the reference paper versus the reviewed implementation are as follows:

```
Leaf Nodes:
           aSA: H( DS | label | value )
      Parakeet: H( H(DS | label | value) | epoch )
      WhatsApp: H( H(label | value) | epoch )
  Experimental: H( DS | H(DS | label | value) | epoch )

Interior Nodes:
           aSA: H( DS | label | left.hash | right.hash | left.label | right.label )
      Parakeet: H( left.hash | right.hash | left.label | right.label )
      WhatsApp: H( H(left.hash | left.label) | H(right.hash | right.label) )
  Experimental: H( DS | left.hash | left.label | right.hash | right.label )
```

The following remarks can be made from the above:

- The chosen hash function of BLAKE3 will provide the necessary collision resistance for the soundness of the underlying aSA for each of the above hashing strategies.
- The hashing strategy for leaf nodes vs interior nodes will prevent second-preimage attacks against the underlying Merkle tree for both configuration types.
- The lack of domain separation in the WhatsApp configuration does not appear to introduce a meaningful attack. There is no other envisioned context in which the resulting hash could be useful to an attacker, and any leaf hash will not be valid at a different location of the tree.
- Omitting the interior label from the hash does not appear to introduce any meaningful attack. Any attempt to modify this label would alter the hash computed by the parent or would alter the derived labels of the nodes below it, which would alter the root hash of the tree.
- The hashing strategy for interior nodes in WhatsApp is distinct compared to the others but incorporates the same information from each child and should provide no advantage to an attacker.

Therefore, while there is divergence in hashing strategy between the two published references and between the two implemented approaches, none of the differences appear to provide an attacker with any additional advantage in forging proofs or compromising the soundness of the approach.