

Iron Fish FishHash

Security Assessment (Summary Report)

April 24, 2024

Prepared for:
Elena Nadolinski
Iron Fish Foundation

Prepared by: **Marc Ilunga and Joop van de Pol**

About Trail of Bits

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

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

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

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

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

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

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2024 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

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

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

Test Coverage Disclaimer

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

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

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

Table of Contents

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Project Summary **4**
Executive Summary **5**
Summary of Findings **7**
The specification contains only limited analysis 7
The access pattern does not depend on the full dataset 7
The use of a fixed dataset would allow precomputations to be useful forever 8
The initial mixing value comes from a small subset of 128-byte values 8
Result of 64-bit multiplication does not rely on all input bits 9
The specification is outdated, incomplete, and contains errors 9
Issues in the implementation 9
A. Algorithm Diagrams **11**
B. Memory Reduction Analysis **14**
C. Fix Review Results **18**
Detailed Fix Review Results 18

Project Summary

Contact Information

The following project manager was associated with this project:

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

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

Project Timeline

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

Date	Event
January 26, 2024	Pre-project kickoff call
February 5, 2024	Delivery of report draft
February 6, 2024	Report readout meeting
March 1, 2024	Delivery of summary report
April 24, 2024	Delivery of summary report with fix review appendix

Executive Summary

Engagement Overview

Iron Fish engaged Trail of Bits to review the security of FishHash, a novel proof-of-work (POW) algorithm based on Ethash and developed for the Iron Fish blockchain to replace the existing cryptographic hash-based POW using the BLAKE3 hash function. FishHash aims to be GPU-friendly and ASIC-resistant by increasing the memory requirement for miners while simultaneously making the computation more efficient for GPU implementations. We reviewed the specification and the Rust implementation of FishHash and its reference C++ implementation in commit `dfdd747` of the GitHub repository `iron-fish/fish-hash`.

A team of two consultants conducted the review from January 29 to February 1, 2024, for a total of seven engineer-days of effort, followed by a fix review on April 19, 2024. With full access to the source code and documentation, we reviewed the specification of FishHash, manually reviewed the Rust implementation and its reference C++ implementation, and performed static and dynamic testing, using automated and manual processes.

Observations and Impact

The primary focus of the engagement was to address the following questions:

- Is FishHash a memory-bound function? Is it possible to compute FishHash hashes efficiently while using considerably less memory than an honest evaluator?
- How do the changes made from Ethash affect the ASIC resistance of FishHash?
- Does the Rust implementation follow the specification and the reference implementation?

Our overall assessment is that, due to the careful use of BLAKE3, FishHash is not vulnerable to trivial attacks such as efficient pre-image computations to speed up the mining process. In other words, FishHash inherits its resistance to typical cryptographic hash attacks from BLAKE3. However, the specification currently lacks formal arguments for the viability of FishHash as a POW function. Similarly to Ethash, the memory-hardness of FishHash is not formally documented, and the specification includes no analysis of space-time trade-offs to justify the desired ASIC resistance. Furthermore, FishHash deviates from Ethash in several aspects, increasing the GPU-friendliness of FishHash; however, those changes may also provide computational advantages to custom mining hardware or other parties. Although we identified such potential risks from such hardware in the design, we could not fully determine the exploitability of these risks during the time-boxed engagement.

We confirmed that the Rust implementation follows the reference C++ implementation. However, we found that the reference C++ implementation does not match the outdated written specification, which should be updated to match the implementation, as was

already noted in a GitHub issue. The codebase contains only end-to-end tests to compare the Rust and C++ implementations; there is no unit testing or negative testing.

Recommendations

We recommend including in the specification a formal analysis of the expected space-time trade-offs of FishHash, asymptotic and concrete analyses of the expected advantages that specialized mining hardware may have over regular miners, and documentation on how deviations from Ethash strengthen FishHash.

We recommend improving the mixing function of the main hash loop. As explained in the next section, the mixing function diverges from Ethash in a manner that allows access patterns to be computed with smaller fractions of the dataset. The fact that access patterns depend only on a subset of the dataset may introduce undesirable space-time trade-offs. A mixing function that ensures that access patterns depend on as much of the dataset as possible will likely complicate space-time trade-offs and limit the potential advantages of custom mining hardware over GPU miners. Additionally, we recommend using only operations that ensure that each input bit of the operands contributes to the result of the mixing function.

Furthermore, we recommend updating the specification to match the implementation, which currently functions as the actual specification. The specification should also be completed with missing elements like test vectors, including intermediate values

Summary of Findings

The specification contains only limited analysis

Section 5 of the specification provides an analysis of the memory and computational demands of the algorithm. However, it does not provide any analysis of the trade-offs between memory usage and computation. Specifically, there is no formal analysis that specifies the relative costs of accessing uncached elements from the dataset versus recomputing these elements from the light cache. Such an analysis would increase confidence in the security of the hash algorithm. Consult the following references for models and analysis of memory-hard proposals:

- Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem: Section II presents an analysis model for memory-hard POW functions and formalizes security properties expected from POW functions.
- Towards Practical Attacks on Argon2i and Balloon Hashing: Section 2 introduces a model for general memory-hard functions. Section 3 provides a concrete analysis of memory requirements for a specific attack. Section 6 analyzes the memory-hardness of the password hashes Argon2 and Balloon Hash.

The access pattern does not depend on the full dataset

The main loop of the hashing kernel consists of two steps: reading from the dataset and mixing the read values into the current mixing value. In FishHash, the first, fifth, and ninth 32-bit words of the mixing value each determine one element to read from the dataset. Furthermore, the mixing function works on 64-bit words of the current mixing value and the corresponding 64-bit words of the read data elements (as shown in appendix A.3). Specifically, each 64-bit word of the updated mixing value depends only on the corresponding 64-bit words (i.e., with the same index) of the original mixing value and the read data elements.

This means that it is possible to determine the full access pattern for a hash computation using only 24 of the 128 bytes of all dataset elements. Storing these 24 bytes per element would reduce the total storage space from 4.8 GB to 0.9 GB (plus 72 MB for storing the light cache). Computing the access pattern does not directly give an advantage in efforts to compute the full hash function because the full 128-byte values of the accessed addresses are still required to finish the computation. However, it does enable different computation strategies that provide a novel trade-off between memory usage (to store dataset elements), BLAKE3 computations (to generate start values), and Keccak computations (to generate dataset elements). For example, consider the following strategy:

1. Store some fraction $z < 1$ of the dataset fully, and for the remaining fraction $1 - z$, store only the 24 bytes that are required to compute the access pattern.

2. Compute a start value (at the cost of applying BLAKE3 to a candidate block header), and use the stored values to compute the access pattern.
3. If not enough of the 96 accesses fall in the stored fraction, abort and compute a new start value.
4. Otherwise, compute the rest of the hash function, using the fraction of the dataset and recomputing the small number of dataset items that are required but not stored. These can be recomputed from the light cache using Keccak.

For a detailed analysis of the trade-offs of this approach, refer to appendix B.

The use of a fixed dataset would allow precomputations to be useful forever

Unlike Ethash, FishHash uses a fixed 4.8 GB dataset that is generated only once. The dependence on a fixed dataset could mean that the computation of a FishHash output for a given input would be fully deterministic over time. Consequently, there is a potential risk that a miner could precompute and optimize certain computations, like the mixing of a certain value, giving them a perpetual advantage over other miners. We did not identify to what extent such precomputations are possible and whether they could give miners a substantial advantage. Nevertheless, a design with a dynamic dataset that changes over time would reduce the utility of precomputation.

The frequency of these rotations can be tuned to ensure that only a small amount of time is spent on building the dataset relative to the time spent mining. To ensure that mining can continue with minimal interruption, it is useful to allow miners to precompute future datasets. However, miners should not be able to gain unfair advantages by analyzing precomputed datasets that will be used at some arbitrary point in the future. A possible solution is to define epochs based on block height. Each epoch can allow several dataset rotations, and each corresponding dataset can be precomputed in advance as a function of the first block header in the epoch. Consequently, datasets of future epochs can be determined only when the corresponding block has been mined.

The initial mixing value comes from a small subset of 128-byte values

The main loop of FishHash uses a 128-byte mixing value. This value is computed by hashing a block header and a nonce, resulting in a 64-byte hash. The hash is then duplicated to initialize the mixing value to a 128-byte value. This duplication constrains the initial value to only a substantially small subset of 128-byte values. BLAKE3 has a built-in extendable output function (XOF) mode, allowing arbitrarily long hash outputs to be computed. Using an XOF instead of duplication would allow the initial mixing value to come from a larger subset without significantly increasing the computational cost.

Result of 64-bit multiplication does not rely on all input bits

The `MixFunction` function uses a number of operations to mix 64-bit words from the dataset with a 64-bit word of the current mixing value. One of these operations is a 64-bit multiplication operation, which has the property that not all operand bits contribute to the output value, depending on the input operands. For example, if a number of least significant bits of one of the operands are zero, then the corresponding most significant bits of the other operand do not contribute to the output value. Contrast this with the FNV function, which ensures that all bits of the operand contribute to the result, regardless of the value of the other operand. We recommend replacing the 64-bit multiplication operation with a 64-bit FNV operation (i.e., using the hexadecimal prime `0x00000100000001B3`), at the cost of one additional 64-bit XOR operation compared to the straightforward 64-bit multiplication (which would add 1,536 additional 64-bit XOR operations for a full hash computation).

The specification is outdated and incomplete and contains errors

The written specification of FishHash does not reflect the expected behavior of the protocol. At the time this report was written, the reference C++ implementation reflects the desired functionality and diverges from the specification. A stable and complete specification is desirable since reference implementations may have subtle implementation bugs that might carry over to other implementations. Furthermore, there are some errors in the specification, hindering its readability. For example, the specification incorrectly states in the description of algorithm 1 that the `ReadDataSet` function is a function of `MixValue`, `StartValue`, and the round counter `n`, whereas in the implementation, `ReadDataSet` is a function of `MixValue` only. Furthermore, the specification currently has no information for testing such test vectors for end-to-end testing and unit testing.

Issues in the implementation

We identified the following opportunities for improvement during our manual review of the Rust implementation:

- The implementation performs arithmetic operations that can panic in debug mode due to overflow. In release mode, overflow checks are disabled by default. From a correctness standpoint, overflows reflect the intended behavior. However, Rust provides tools for explicitly describing the intended behavior. For example, because the `fnv1` function panics on certain inputs, it can instead be implemented as the following:

```
u.wrapping_mul(FNV_PRIME) ^ v.
```

- The library relies on an outdated dependency for BLAKE3. The implementation relies on version 1.3.1, whereas the latest version is 1.5.0. The release notes of the `blake3` crate indicate that releases after 1.3.1 have fixes for issues like incorrect outputs in certain execution contexts. Therefore, we recommend using the latest

version if possible to reduce the potential attack surface and ensure the reliability of the Rust implementation of FishHash.

A. Algorithm Diagrams

This appendix includes diagrams of the protocol and some of its essential functions.

A.1. Overview

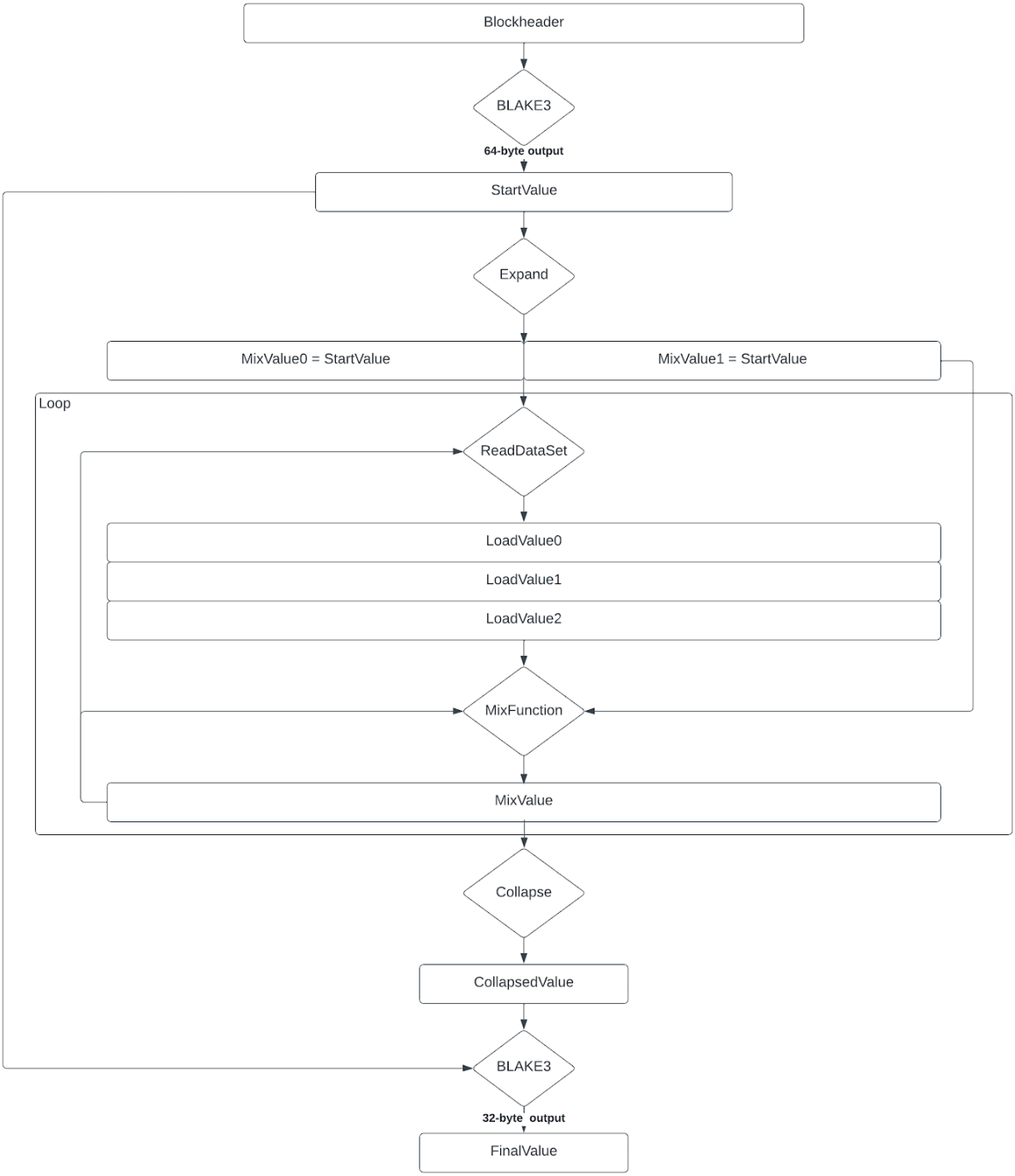


Figure A.1: The overall structure of FishHash

A.2. The ReadDataSet Function

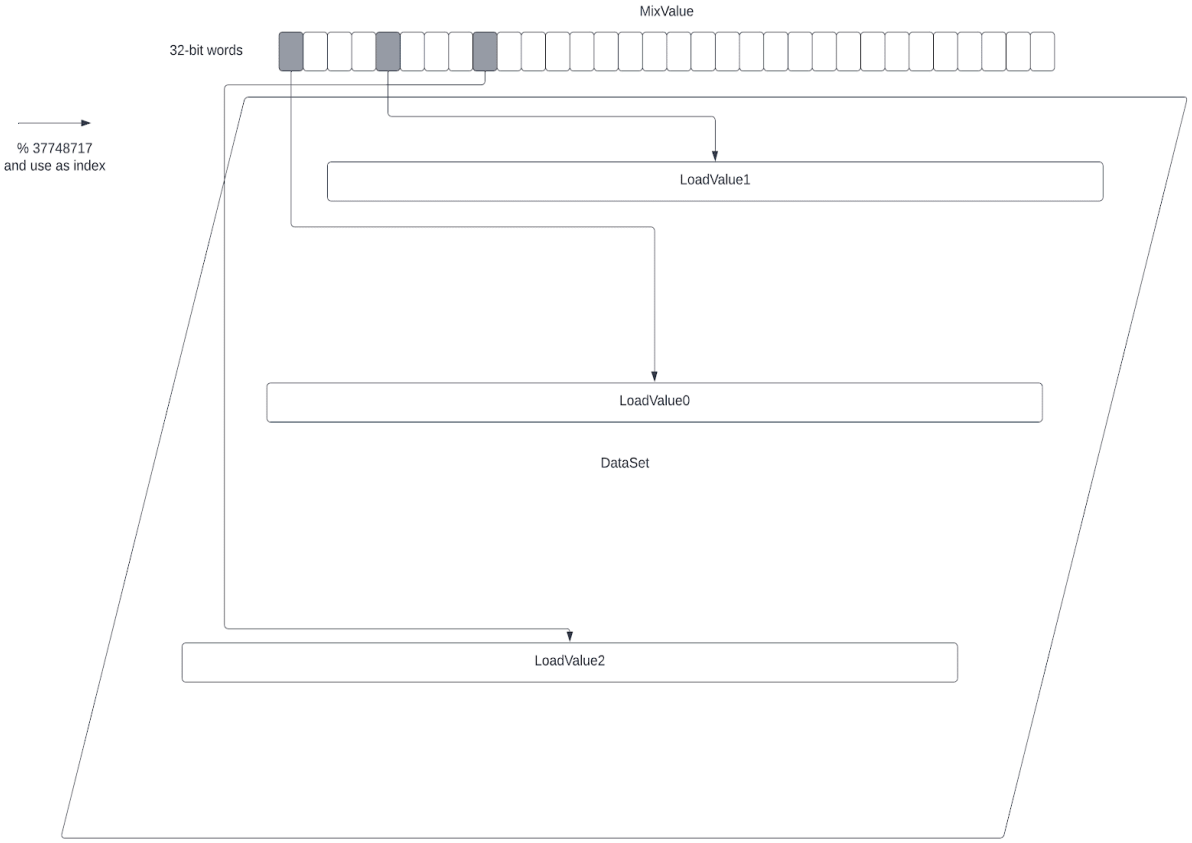


Figure A.2: A depiction of the `ReadDataSet` function

A.3. The MixFunction Function

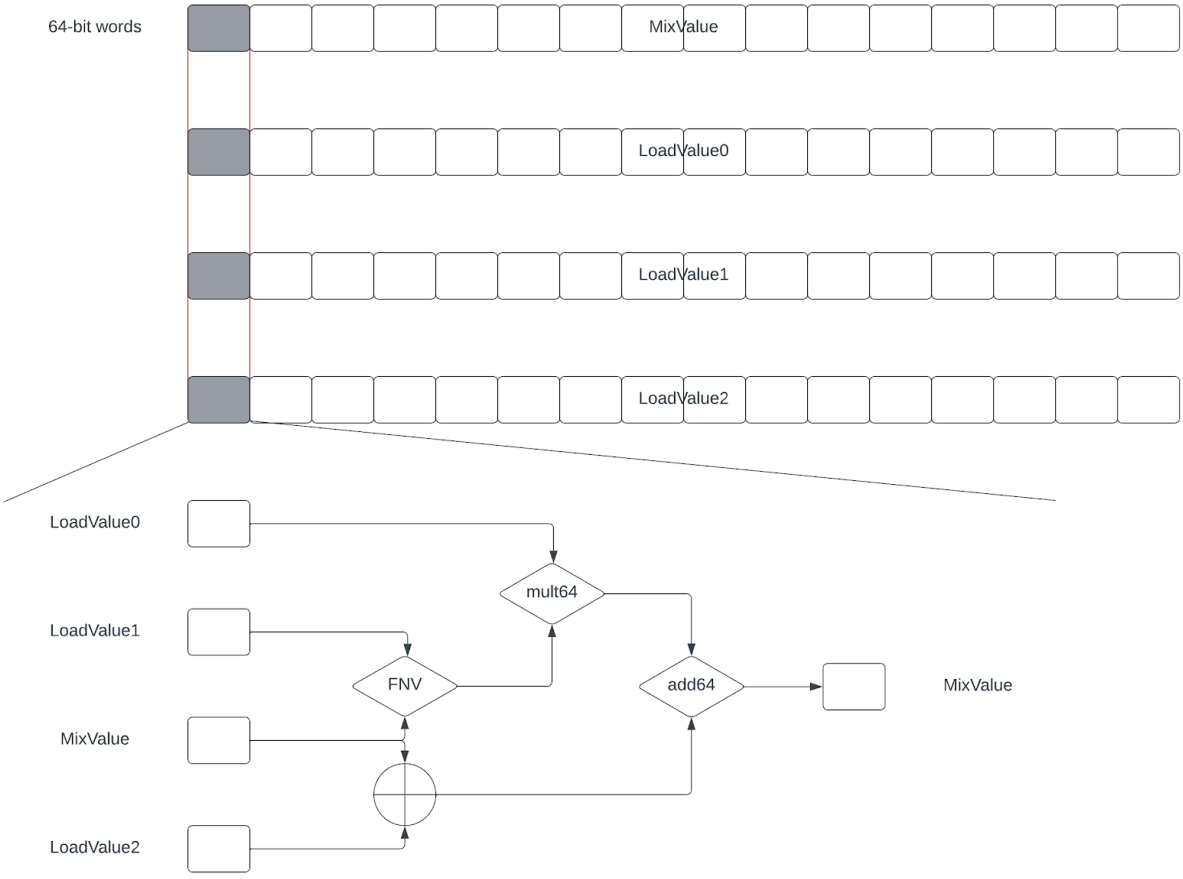


Figure A.3: A depiction of the effect of the `MixFunction` function on one 64-bit word[1] of the mixing value

A.4. The Collapse Function

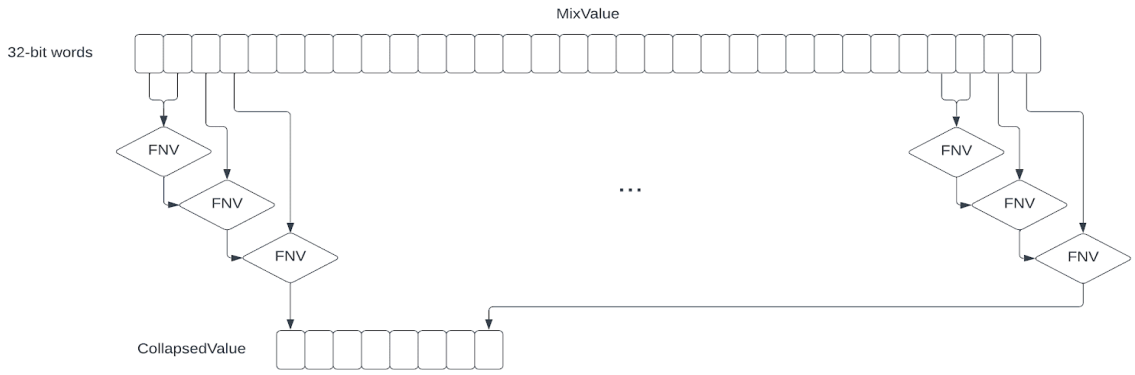


Figure A.4: A depiction of the `Collapse` function

[1] Note that the depicted FNV function acts separately on each 32-bit word of the input 64-bit words.

B. Memory Reduction Analysis

Recall the following strategy for reducing memory requirements that we described in the Summary of Findings section:

1. Store some fraction $z < 1$ of the dataset fully, and for the remaining fraction $1 - z$, store only the 24 bytes that are required to compute the access pattern.
2. Compute a start value (at the cost of applying BLAKE3 to a candidate block header), and use the stored values to compute the access pattern.
3. If not enough of the 96 accesses fall in the stored fraction, abort and compute a new start value.
4. Otherwise, compute the rest of the hash function, using the fraction of the dataset and recomputing the small number of dataset items that are required but not stored.

The storage requirements are $0.97 + 3.9z$ GB. The number of accesses that fall within the stored dataset fraction is binomially distributed for 96 trials with probability z, or, equivalently, the number of accesses that do not fall within this fraction is binomially distributed for 96 trials with probability $1 - z$. Let b be the bound on the allowed "misses." The probability that there are b or fewer misses is given by the cumulative distribution function $F(k; n, p)$ of the binomial distribution with n trials and success probability p:

$$p_b = F(b;\ 96,\ 1 - z) = \sum_{i=0}^{b} \binom{96}{i} (1 - z)^i z^{96-i}$$

The number of candidate block headers to be processed with BLAKE3 to produce a start value with b or fewer misses is geometrically distributed with probability p_b, which has a mean of $1/p_b$. The probability that there are x misses, given that $x \leq b$, is given by the following:

$$p_x = f(x;\ 96,\ 1 - z) \,/\, F(b; 96, 1 - z) = \frac{\binom{96}{x}(1-z)^x z^{96-x}}{\sum_{i=0}^{b} \binom{96}{i}(1-z)^i z^{96-i}}$$

In the above, $f(k; n, p)$ is the probability mass function of the binomial distribution with n trials and success probability p. The expected number of misses is given by the following:

$$\sum_{x=0}^{b} x p_x$$

This number of misses is proportional to the number of Keccak operations that must be performed to recompute the missing dataset elements. These dataset elements are required to compute the full hash function corresponding to the chosen start value.

Figure B.1 shows an example of the trade-off for $z = 0.5$ and a range of values for the bound b. Picking $b = 38$ means that the expected number of start values is approximately 39, whereas the expected number of misses is approximately 37.

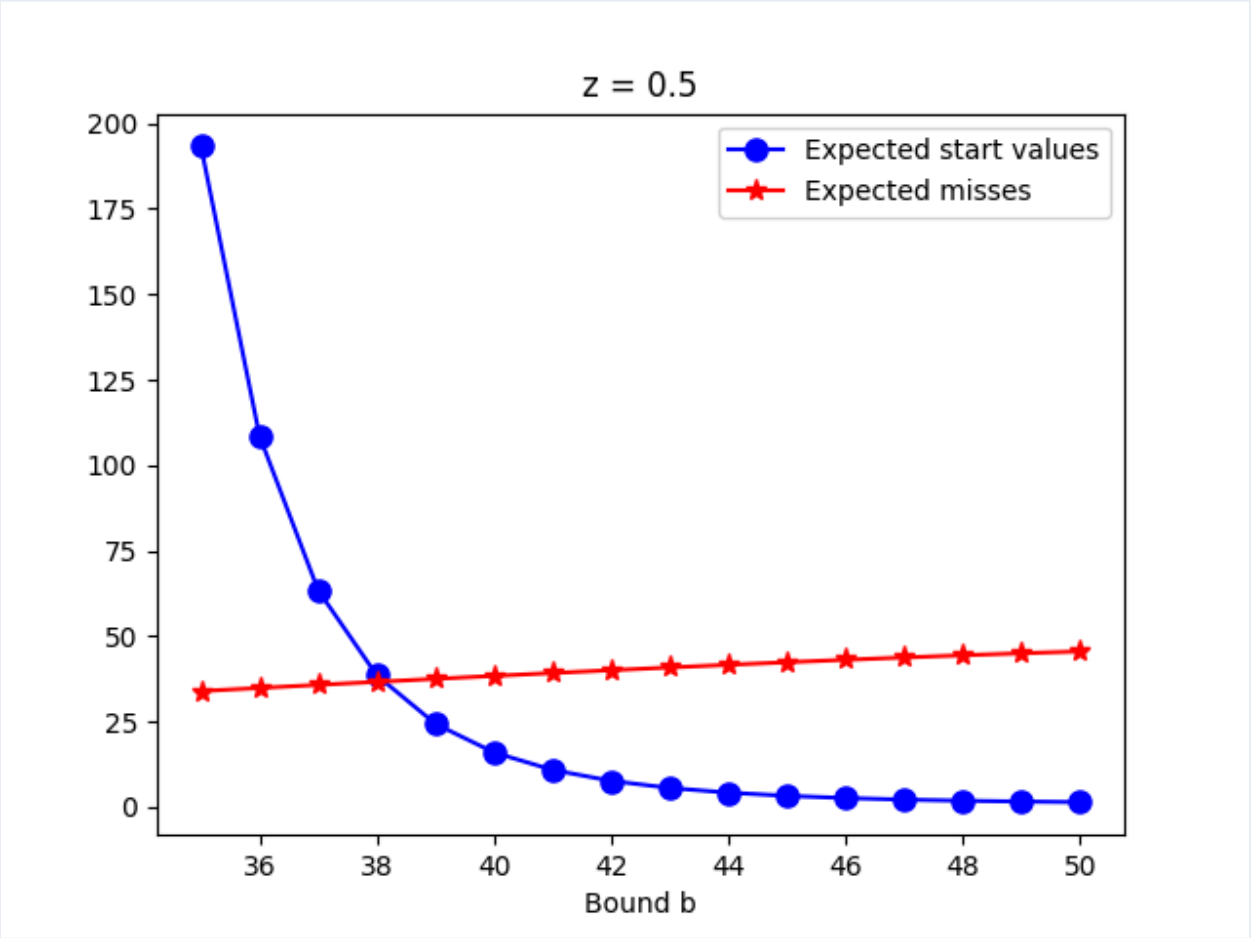


Figure B.1: The trade-off between start values and expected misses for $z = 0.5$

Figure B.2 shows an example of the trade-off for $z = 0.9$ and a range of values for the bound b. Picking $b = 6$ means that the expected number of start values is approximately 7, whereas the expected number of misses is approximately 5.

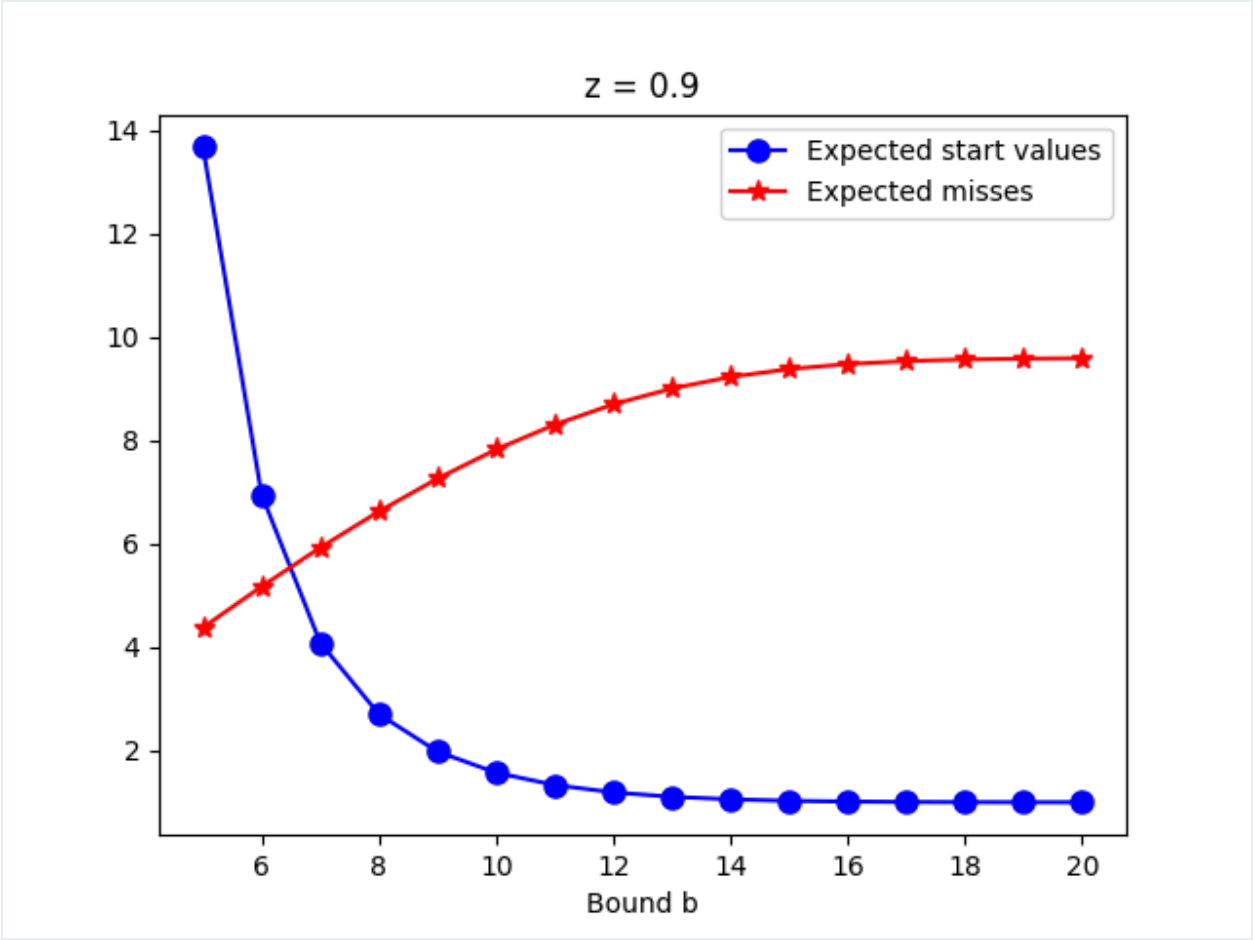


Figure B.2: The trade-off between start values and expected misses for $z = 0.9$

Table B.3 shows the trade-off between storage and the expected number of BLAKE3 rounds and Keccak invocations, given that each start value requires three BLAKE3 rounds and each dataset item computation requires four Keccak invocations. Note that this represents only part of the additional computation, as the computation of the access pattern needs to be taken into account as well. When $z = 0$, all accesses will miss, so there is no need to store the partial values used to compute the access pattern. In this case, only the light cache needs to be stored.

z	Storage (GB)	*b*	Number of BLAKE3 rounds	Number of Keccak invocations	*z*	Storage (GB)	*b*	Number of BLAKE3 rounds	Number of Keccak invocations
0.0	0.07	96	3	384	0.5	2.92	36	327	140
0.1	1.36	77	1050	308			37	192	144
		78	459	312			38	117	148
		79	213	316			39	75	152
		80	105	320	0.6	3.31	28	165	108
0.2	1.75	66	498	264			29	99	112
		67	267	264			30	63	116
		68	150	268			31	42	120
		69	87	272	0.7	3.70	20	105	76
0.3	2.14	56	303	220			21	63	80
		57	177	224			22	39	84
		58	108	228			23	27	88
		59	69	232	0.8	4.09	12	82	48
0.4	2.53	46	276	180			13	45	48
		47	165	184			14	27	52
		48	102	188			15	18	56
		49	66	192	0.9	4.48	4	99	16
							5	42	20
							6	21	24
							7	15	24

Table B.3: The trade-off between storage and expected BLAKE3 rounds and Keccak invocations

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the system design, source code, and system configuration, not comprehensive analysis of the system. In other words, we examine the commit or pull request (PR) for each fix to determine whether it properly resolves the issue; however, we do not spend additional time investigating whether the fix introduced new issues into the system.

On April 18, 2024, Trail of Bits reviewed the fixes and mitigations implemented by Iron Fish for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Of the seven issues identified, Iron Fish provided fixes for five. Two fixes have been deployed through a hard fork—namely, updates to the specification and remediation of issues in the implementation. The remaining fixes will be considered future upgrades.

Detailed Fix Review Results

The specification contains only limited analysis
Unresolved. Iron Fish did not address this issue in time for the fix review because a complete analysis of FishHash will require a nontrivial amount of time.

The access pattern does not depend on the full dataset
Resolved in PR #15. In each round of the FishHash kernel, the indices used to access the dataset now depend fully on the current mixing value. More precisely, the first and second indices are each computed as the XOR of twelve 32-bit values taken from the current mixing value. The third index is computed from the remaining 32-bit values and the round index.

The use of a fixed dataset would allow precomputations to be useful forever
Partially resolved in PR #7. The implementation has been modified to allow user-provided seeds for generating new datasets. However, Iron Fish decided not to specify any mechanisms for rotating the dataset used in the operations of the Iron Fish mainnet at this time based on its risk assessment.

The initial mixing value comes from a small subset of 128-byte values
Resolved in PR #16. The implementation now uses the extendable output function (XOF) mode of BLAKE3 to produce a 128-byte hash, subsequently used as the initial mixing value. We note that the final FishHash value is computed only from the first 64 bytes of the initial mixing value. Using only part of the initial mixing value does not hurt security and offers minor performance gains; however, the full mixing value can also be safely used to compute the final output at the cost of one additional BLAKE3 round.

Result of 64-bit multiplication does not rely on all input bits
Unresolved. Iron Fish decided not to address the issue before the fix review.

The specification is outdated and incomplete and contains errors
Partially resolved. The final specification still contains a few discrepancies with the implementation. For example, the general specification of `ReadDataSet` still does not reflect the implementation of FishHash.

Issues in the implementation
Resolved in PR #12, PR #6 and PR #8.

- **Prevention of panics in non-release builds:** The fixes prevent panics in non-release builds by using arithmetic operations with explicit wrapping behavior (i.e., `wrapping_mul` and `wrapping_add`). However, certain lines of code have been commented out in the comparison crate to avoid time-consuming tests, which reduces the code's quality.
- **Update to an outdated dependency:** The `cargo.toml` manifest has been updated to use the latest version of BLAKE3 available at the time the issue was addressed—namely, version 1.5.0. At the time of this fix review, a newer version of BLAKE3 (1.5.1) is available.