Solving the FNV Zero-Hash Challenge on Consumer-Grade Hardware

Joseph Crail

Abstract

Even though it is a non-cryptographic hash function, FNV-1a is ubiquitous in many applications due to its speed and ease of implementation. The FNV zero-hash challenge seeks to find a minimal solution set of collision-producing strings for various alphabets and bit sizes. In this paper, we describe a novel approach that solves open zero-hash questions on consumer-grade hardware without relying on GPUs or other high-performance computing. Using deterministic acyclic finite state automata and quotient graphs, our solver answered three open questions within a constant factor of a theoretical exhaustive search on a TOP200 supercomputer.

1 Introduction

Fowler/Noll/Vo (or FNV) is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo in 1991, and was recently published as RFC 9923 [3]. Due to its speed and ease of implementation, it is ubiquitous in many applications.

The authors of FNV have an ongoing competition which asks whether one can find all of the shortest strings that produce a hash value of zero for various alphabets and sizes of both FNV-1 and FNV-1a [9].

Zero represents a fundamental vulnerability for FNV due to its reliance on multiplication and exclusive-or operations. If these operations generate a hash value of zero at any point and the remaining input bytes are also zero, the hash would remain unchanged, making collisions trivial [4].

For the scope of this article, we will focus on 64-bit FNV-1a as this algorithm variant has open questions and is computationally feasible on consumer-grade hardware. We will describe a novel approach to solving the FNV zero-hash challenges beyond strategies that rely on exhaustive search or high-performance computing.

The remainder of this paper is organized as follows. Section 2 reviews preliminary concepts and defines necessary terms. Section 3 outlines the design and strategies of the zero-hash solver. Section 4 details the specific implementation of the zero-hash solver. Section 5 presents the results of our experiments.

Section 6 suggests directions for future work. Finally, Section 7 concludes the paper and summarizes our key findings.

2 Background

2.1 FNV-1a algorithm

The general FNV-1a algorithm is listed in Algorithm 1. For each n-bit variant of the algorithm, *basis* and *prime* are respectively modified.

Algorithm 1 FNV-1a

Require: $n \in \{32, 64, 128, 256, 512, 1024\}$

1: $hash \leftarrow basis$
2: **for all** $byte \in bytes$ **do**
3: $\quad hash \leftarrow hash \oplus byte$
4: $\quad hash \leftarrow hash \cdot prime \ (mod \ 2^n)$
5: **end for**
6: **return** $hash$

Given h_i as the intermediate hash value and b_i as the respective byte to be hashed, then Algorithm 1 can be unrolled as shown in Algorithm 2.

The final hash value, h_i, must be zero as this is the objective of the zero-hash challenge.

Algorithm 2 FNV-1a (unrolled)

Require: $n \in \{32, 64, 128, 256, 512, 1024\}$

1: $h_0 \leftarrow basis$
2: $h_1 \leftarrow (h_0 \oplus b_0) \cdot prime \ (mod \ 2^n)$
...
3: $h_i \leftarrow (h_{i-1} \oplus b_{i-1}) \cdot prime \ (mod \ 2^n)$
4: **return** h_i

2.2 Zero-hash challenge

The zero-hash challenge asks whether one can find the shortest strings over a given alphabet Σ that generate a zero hash value for the 64-bit FNV1-a algorithm. If P is a predicate that is true iff the hash function maps a string s to 0, then we can formally define the problem in Equation (1).

$$\mathcal{Z} = \underset{s \in \Sigma^*}{\arg\min} |s| \text{ subject to } P(s) \tag{1}$$

The relevant set of alphabets and their respective bytes are seen in Table 1.

Alphabet	Valid bytes
Alphanumeric	[0x30–0x39, 0x41–0x7A]
Printable	[0x20–0x7E]
7-bit (non-nul)	[0x01–0x7F]
8-bit	[0x00–0xFF]

Table 1: Valid bytes for each alphabet

2.3 Hash functions and cryptographic attacks

A hash function H maps arbitrary-length strings over an alphabet Σ to a finite set of binary strings of length n, that is, $H : \Sigma^* \to \{0,1\}^n$. On its own, a hash function cannot be used as a primitive in cryptographic applications unless it satisfies at least one of three conditions: preimage resistance, second preimage resistance, and collision resistance [13], [15]. Otherwise, a hash function H is known as a non-cryptographic hash function, as is the case with the FNV family.

Definition (Preimage resistance)**.** Given a hash function H and a string y, it is computationally infeasible to find a string x such that $H(x) = y$.

Definition (Second preimage resistance)**.** Given a hash function H, a string x, and $H(x)$, it is computationally infeasible to find a string $x' \neq x$ such that $H(x') = H(x)$.

Definition (Collision resistance)**.** It is computationally infeasible to find two distinct strings that hash to the same result.

If we are able to demonstrate that any of the above definitions are weakened for a specific cryptographic hash function, then it will be vulnerable to recovering the original string. With this intention, cryptographic attacks are generally categorized as brute-force attacks, preimage attacks, and collision attacks. A brute-force attack exhaustively checks all possible strings for a particular goal. A preimage attack searches for a string that has a specific hash value, often using a precomputed table or rainbow tables [12], [18]. A collision attack tries to find two strings producing the same hash value.

Even though the FNV family are not cryptographic hash functions, we can still borrow specific cryptographic attack techniques to solve the zero-hash challenges. Since non-cryptographic hash functions are not preimage resistant, the zero-hash solver exploits this observation and is characterized as a preimage attack.

2.4 Consumer-grade hardware

The term *consumer-grade hardware* has no uniformly agreed-upon definition. Exact specifications can quickly become dated, so we will use a broad description.

Alphabet	**8**	**9**	**10**	**11**
Alphanumeric	2.48E+14	1.56E+16	9.85E+17	6.21E+19
Printable	6.63E+15	6.30E+17	5.99E+19	5.69E+21
7-bit (non-nul)	6.77E+16	8.59E+18	1.09E+21	1.39E+23
8-bit	1.84E+19	4.72E+21	1.21E+24	3.09E+26

Table 2: Search space size

Within the scope of this article, consumer-grade hardware is often a laptop or a desktop, has a relatively low to moderate price point, has a lower durability and mean-time-between-failure than enterprise hardware, may have a low number of multi-core processors, and either has an integrated GPU or no more than one discrete GPU.

3 Zero-Hash Solver

Given that a hash function operates on arbitrary-length strings, one does not know in advance how long it will take to find a solution. It is reasonable for one to seek a bound on the search or at least a probable starting point that would yield a solution before continuing further.

Assuming a perfect hash function without collisions, then by the pigeonhole principle one can expect all values of the codomain to be generated at least once when $|\Sigma|^m \geq 2^n$, where m is the minimum string length and n is the number of bits. We can then solve for m to find a theoretical upper bound on the length of the strings in Equation (1).

$$m \geq \left\lceil \frac{n \log 2}{\log |\Sigma|} \right\rceil \tag{2}$$

For those open zero-hash challenges using the 64-bit FNV-1a algorithm, see the highlighted cells in Table 2 for the respective theoretical upper bound for each given alphabet. This also matches the existing minimum solutions for those challenges. The FNV family however does not contain perfect hash functions, so the theoretical upper bound on the length of the strings in Equation (1) is instead a reasonable first approximation.

3.1 Exhaustive search

Fortunately, searching for all zero values generated by the FNV-1a algorithm is an embarrassingly parallel problem that can take advantage of the modest multi-core capabilities of modern consumer-grade hardware. Equipped with a reasonable upper bound m on the minimum string length, we can then search all strings up to length m until we find the minimum solution set.

A naive zero-hash solver could use Knuth's Algorithm M to generate all n-tuples over an alphabet Σ in lexicographic order until a solution set is found [5].

If we used a professional discrete GPU with an average estimated performance of 10 TFLOPS, a rough estimate of the time necessary to exhaustively search the entire solution space for a given alphabet and minimum string length would be measured in weeks to years [10], [11]. Even if we unlikely had access to a TOP200 supercomputer with a performance of 10 PFLOPS, the estimated search would take minutes to days [17]. See Table 3 for the estimated performance of an exhaustive search on selected architectures.

3.2 A more refined approach

Instead of generating an exhaustive enumeration of solutions, we have identified two strategies to improve on the performance of a naive zero-hash solver: 1) reformulate the problem as a graph search problem on a fixed-depth deterministic acyclic finite state automaton, and 2) reduce the search space by eliminating non-viable candidate solutions [14].

3.2.1 Deterministic acyclic finite state automaton

The FNV-1a hash function was previously unrolled in Algorithm 2. We can see there is an intermediate hash function, $f(h_{i-1}, b_{i-1}) = h_i$, that computes intermediate hash values. This suggests that the FNV-1a hash function has an implicit graph representation, since each hash value is dependent on the prior hash value and prior byte in a string; that is, intermediate hash values are nodes and bytes are edges.

This implicit graph representation can be modeled explicitly as a deterministic acyclic finite state automaton [13], which is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the input alphabet (a finite non-empty set of symbols);
- S is a finite non-empty set of states;
- s_0 is an initial state and an element of S;
- δ is the state-transition function: $\delta : S \times \Sigma \to S$;
- F is the set of final states, a (possibly empty) subset of S;
- there is no string ω over Σ and state $s \in S$ such that $\delta(s, \omega) = s$.

In the context of our search problem, Σ is one of the available alphabets, S is the set of pairs in which a pair is the current string length and intermediate hash values, s_0 is the offset basis as determined by the particular FNV algorithm, δ is the intermediate hash function, and F is $\{0\}$. Thus, our original problem can be reformulated explicitly as a single-pair shortest path problem, where the source is the offset basis and the sink is zero.

N	Consumer-Grade	Discrete GPU	TOP200 Supercomputer
8	8.27 minutes	24.82 seconds	24.82 milliseconds
9	8.69 hours	26.06 minutes	1.56 seconds
10	3.26 weeks	1.14 days	1.64 minutes
11	3.94 years	2.39 months	1.72 hours

(a) Alphanumeric

N	Consumer-Grade	Discrete GPU	TOP200 Supercomputer
8	3.69 hours	11.06 minutes	663.42 milliseconds
9	2.08 weeks	17.51 hours	1.05 minutes
10	3.80 years	2.31 months	1.66 hours
11	3.61 centuries	1.80 decades	6.58 days

(b) Printable

N	Consumer-Grade	Discrete GPU	TOP200 Supercomputer
8	1.57 days	1.88 hours	6.77 seconds
9	6.63 months	1.42 weeks	14.32 minutes
10	6.92 decades	3.46 years	1.26 days
11	8.79 millennia	4.40 centuries	5.35 months

(c) 7-bit (non-nul)

N	Consumer-Grade	Discrete GPU	TOP200 Supercomputer
8	1.17 years	3.05 weeks	30.74 minutes
9	2.99 centuries	1.50 decades	5.47 days
10	76.67 millennia	3.83 millennia	3.83 years
11	19627.41 millennia	981.37 millennia	9.81 centuries

(d) 8-bit

Table 3: Estimated time for exhaustive search

3.2.2 Search space reduction

If one constructs a deterministic acyclic finite state automaton as outlined in the previous section, then the resulting graph will be tree-like since the offset basis is the root and collisions may occur. Unfortunately, one major practical drawback of this representation is that the number of nodes required for storage is close to $O(|\Sigma|^m)$, where m is the minimum string length (the length of the shortest path from the initial state to the final state) and $|\Sigma|$ is the branching factor (the maximum number of successors for any given state).

To reduce the search space and subsequent graph representation, what if the original states (i.e. intermediate hash values) were assigned truncated values? For instance, since the 64-bit FNV-1a algorithm uses 64-bit values, the least significant 32 bits or fewer could be used as the states, thus, drastically reduce the storage space to $O(\sqrt{|\Sigma|^m})$.

Inspired by a similar technique used by rainbow tables [12], [18], this reduction function when applied to the set of intermediate hash values in S transforms the single-pair shortest path problem from one in which we want to find a 64-bit zero into one in which we look for all candidates that lead to a 32-bit zero. We could then evaluate this much smaller set of candidates to determine if they evaluate to a 64-bit zero.

However, rainbow tables are a probabilistic method and success is not guaranteed. The combination of a deterministic acyclic finite state automaton with a reduction function guarantees success in a reasonable time by eliminating non-viable candidate solutions.

See Appendix A for a formal demonstration that this reduction function can shrink the original graph representation.

3.2.3 Putting everything together

All strings over an alphabet Σ with only three symbols are highlighted in Figure 1. Each node denotes the hash digest of the string corresponding to the path; the rightmost column contains digests of all length-n strings. The dotted node marks the target zero digest.

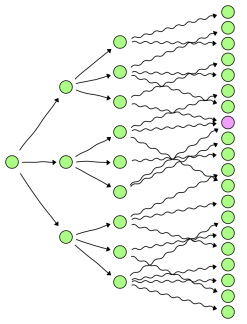


Figure 1: Graph of enumeration over alphabet Σ

Highlighting every path from the leftmost node to the dotted node yields all length-n strings whose hash equals zero in Figure 2.

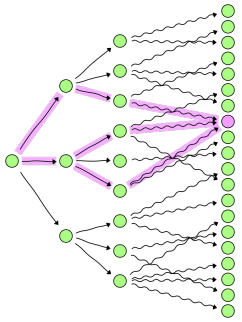


Figure 2: All paths leading to the target zero digest

If we remove any nodes and edges not on a path to the dotted node, the remaining subgraph in Figure 3 is a deterministic acyclic finite-state automaton that accepts exactly those zero-hash strings.

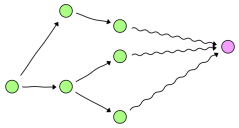


Figure 3: Pruned subgraph

While the above diagrams do not represent the actual implementation, they do help us develop an intuition for possible strategies and how they could improve the performance of an exhaustive search.

4 Implementation

Since our target environment is constrained by consumer-grade hardware, we have added flexibility into the implementation in order to adjust the time-memory trade-off as needed. Borrowing a comparable architecture from cryptographic password crackers, we use a two-phase pipeline for the solver: a precomputation phase and a subsequent search phase. The implementation is written in the Rust programming language.

4.1 Precomputation phase

As the zero-hash solver is impacted predominantly by the design choices made in the construction and serialization of the deterministic acyclic finite state automaton, we had to balance implementation simplicity, time-memory tradeoffs, and future performance enhancements.

The constructed graph of the automaton is parameterized on the alphabet, the reduction function, and the desired string length. The graph enumerates all potential solutions over an alphabet for a given length. Each node is represented by the codomain of the reduction function, $g : \{0,1\}^{64} \rightarrow \{0,1\}^{r}$, where $r = \{8, 12, 16, 20, 24, 28\}$.

We divided the precomputation phase into several steps: 1) forward pass, 2) backward pass, 3) joining, 4) pruning, and 5) serialization. First, the forward pass starts from s_0, the offset basis, and uses the hash function to generate the next level from all alphabet symbols until reaching the midpoint. Next, the backward pass starts from zero, and uses the inverse of the hash function as defined in Appendix A to generate each graph level from all values of the alphabet until reaching the midpoint. The two trees are then joined by connecting the rightmost nodes from the forward tree with the leftmost nodes in the backward tree. A minimization step is performed to prune dead and unreachable nodes. The graph is now isomorphic to the minimal acyclic finite-state automata [2]. Finally, the in-memory graph representation is serialized to disk as input for the search phase.

For each level in the graph, source nodes are represented by a 32-bit unsigned integer and edges are represented by a bit vector. The target nodes are not needed during precomputation since the full 64-bit values are calculated in the second phase. Each pass constructs each level in breadth-first order.

For example, a constructed graph before the pruning step is shown in Figure 4. The leftmost highlighted node is dead; the rightmost highlighted node is unreachable.

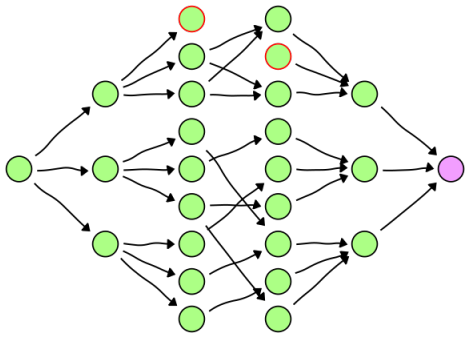


Figure 4: Unpruned graph

It should also be noted that the paths in a constructed graph are equivalent to potential solutions searched by the graph-based solver.

4.2 Search phase

In order to search the constructed graph for solutions, we generate all paths of a certain depth and add those paths to a queue from which a thread pool will select a path. Each thread will then begin a depth-first search of the graph

starting from each path endpoint. If a zero hash value is found, those solutions will be collected and displayed.

5 Experimental Results

All tests were executed on the same computer, an AMD Ryzen 7 7840U at 5.1GHz with 8 cores and 64GiB of DDR5 RAM.

For all alphabets and strings of length $n \leq 5$, we ran an exhaustive search and found no solutions matching zero.

For the graph-based solver, we tested all alphabets, all strings of length n starting from 6 up to the theoretical upper bound for the respective alphabet, and the reduction function, $g : \{0,1\}^{64} \to \{0,1\}^{28}$.

All highlighted rows in Table 4 indicate where we found solutions matching zero. Our results confirmed that the minimum solution set for each alphabet matches the existing solutions and found no new solutions at or below the respective minimums; thus, the 44th, 56th, and 68th zero-hash challenges are fully solved [9].

The charts in Figure 5 show results for each alphabet and their respective minimum string length and corroborate these discoveries. The mask size refers to the codomain size for each reduction function, as defined by $\{0,1\}^r$, where $r = \{8, 12, 16, 20, 24, 28\}$.

6 Future Work

A few limitations were identified during the course of this research. One key constraint was the limited amount of random-access memory available in our experimental environment, which restricted us to the 64-bit challenges. If we were to use more succinct or disk-based data structures for our graph, perhaps we could solve the 128-bit challenges and beyond on consumer-grade hardware. Another area for future work is to experiment with other parallelization techniques, for example, utilizing SIMD instructions or a discrete GPU to enhance the precomputation and search phases.

If we relaxed our desire for an exhaustive solution set, then certain cryptanalytic attacks and cryptanalysis methods become available. First, one could explore whether a birthday attack on the 128-bit challenges would yield collisions. Furthermore, prior work has shown the effectiveness of SAT solvers against cryptographic hash functions [6], [7], [16], so Z3 [8] or similar solvers could be an useful approach.

Even though we focused only on the 64-bit FNV-1a algorithm, the 50th and 62nd zero-hash challenges for 64-bit FNV-1 are still open, so we could update our implementation to solve them [9].

N	Nodes	Edges	Paths	RAM	Elapsed
6	7.87E+02	1.02E+03	2.36E+02	1.41GiB	0:00:04
7	3.39E+04	4.86E+04	1.47E+04	1.41GiB	0:00:04
8	1.22E+06	2.13E+06	9.24E+05	1.49GiB	0:00:05
9	2.75E+07	7.67E+07	5.82E+07	3.35GiB	0:00:28
10	2.79E+08	1.73E+09	3.67E+09	35.04GiB	0:07:27
11	5.47E+08	1.76E+10	2.31E+11	56.45GiB	2:56:45

(a) Alphanumeric

N	Nodes	Edges	Paths	RAM	Elapsed
6	6.44E+03	9.18E+03	2.74E+03	1.41GiB	0:00:04
7	3.48E+05	6.09E+05	2.60E+05	1.45GiB	0:00:05
8	1.17E+07	3.31E+07	2.47E+07	2.49GiB	0:00:14
9	1.05E+08	1.11E+09	2.35E+09	9.01GiB	0:03:49
10	3.73E+08	9.98E+09	2.23E+11	40.45GiB	2:40:12

(b) Printable

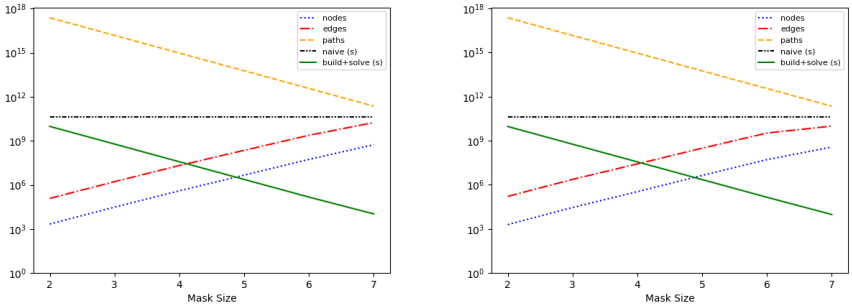
N	Nodes	Edges	Paths	RAM	Elapsed
6	2.69E+04	4.26E+04	1.57E+04	1.41GiB	0:00:04
7	1.42E+06	3.40E+06	1.98E+06	1.64GiB	0:00:06
8	3.49E+07	1.80E+08	2.52E+08	5.73GiB	0:00:46
9	1.86E+08	4.43E+09	3.20E+10	16.53GiB	0:27:35
10	4.54E+08	2.36E+10	4.07E+12	47.18GiB	45:34:28

(c) 7-bit (non-nul)

N	Nodes	Edges	Paths	RAM	Elapsed
6	1.15E+06	2.20E+06	1.05E+06	1.53GiB	0:00:05
7	2.56E+07	2.94E+08	2.68E+08	4.28GiB	0:00:46
8	1.88E+08	6.54E+09	6.87E+10	22.58GiB	0:54:36

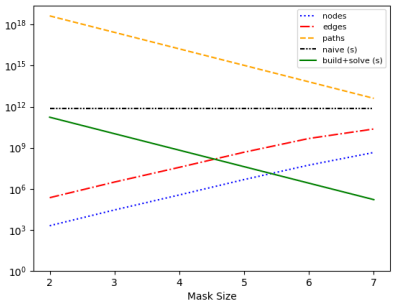
(d) 8-bit

Table 4: Results for Graph-based Solver

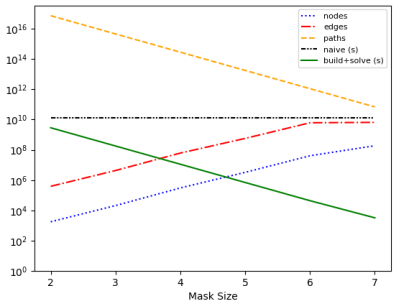


(a) n=11, alphabet=alphanumeric

(b) n=10, alphabet=printable



(c) n=10, alphabet=non-nul



(d) n=8, alphabet=binary

Figure 5: Graph and solver characteristics

7 Conclusion

We have introduced a novel approach to solve the computationally-expensive FNV zero-hash challenges on modern consumer-grade hardware. By employing judicious performance engineering and exploiting the combinatorial structure in the 64-bit FNV-1a hash algorithm, we resolved three open challenges with a significant order-of-magnitude improvement over an exhaustive search or usage of high-performance computing.

8 Acknowledgments

The author wishes to thank the numerous private reviewers for their valuable feedback.

References

[1] D. Bubboloni, "Graph homomorphisms and components of quotient graphs," *Rendiconti del Seminario Matematico della Università di Padova*, vol. 138, pp. 39–60, 2017. DOI: 10.4171/RSMUP/138-2 [Online]. Available: https://www.numdam.org/articles/10.4171/RSMUP/138-2/

[2] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson, "Incremental construction of minimal acyclic finite-state automata," *Computational Linguistics*, vol. 26, no. 1, pp. 3–16, Mar. 2000, ISSN: 0891-2017. DOI: 10.1162/089120100561601 eprint: https://direct.mit.edu/coli/article-pdf/26/1/3/1797487/089120100561601.pdf. [Online]. Available: https://doi.org/10.1162/089120100561601

[3] G. Fowler, L. C. Noll, K.-P. Vo, D. E. E. 3rd, and T. Hansen, *The FNV Non-Cryptographic Hash Algorithm*, RFC 9923, Feb. 2026. DOI: 10.17487/RFC9923 [Online]. Available: https://www.rfc-editor.org/info/rfc9923

[4] C. Hayes and D. Malone, "An Evaluation of FNV Non-Cryptographic Hash Functions," in *2024 35th Irish Signals and Systems Conference (ISSC)*, 2024, pp. 1–8. DOI: 10.1109/ISSC61953.2024.10603139

[5] D. E. Knuth, *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Upper Saddle River, NJ: Addison-Wesley, 2011.

[6] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *Journal of Automated Reasoning*, vol. 24, no. 1, pp. 165–203, 2000.

[7] I. Mironov and L. Zhang, "Applications of SAT solvers to cryptanalysis of hash functions," in *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2006, pp. 102–115.

[8] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, Mar. 2008, pp. 337–340. [Online]. Available: https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[9] L. C. Noll. "FNV hash," Accessed: May 24, 2026. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv

[10] NVIDIA. "NVIDIA A100," Accessed: May 24, 2026. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100

[11] NVIDIA. "NVIDIA H200," Accessed: May 24, 2026. [Online]. Available: https://www.nvidia.com/en-us/data-center/h200

[12] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630, ISBN: 978-3-540-45146-4.

[13] B. Preneel, "Analysis and design of cryptographic hash functions," Ph.D. dissertation, Katholieke Universiteit te Leuven Leuven, 1993.

[14] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Application to DES," in *Workshop on the Theory and Application of of Cryptographic Techniques*, Springer, 1989, pp. 429–434.

[15] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *International workshop on fast software encryption*, Springer, 2004, pp. 371–388.

[16] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2009, pp. 244–257.

[17] TOP500.org. "TOP500 Supercomputer Sites." 66th edition, November 2025. [Online]. Available: `https://www.top500.org/lists/top500/list/2025/11/`

[18] M. Vainer, A. Kačeniauskas, and N. Goranin, "Parallelization of Rainbow Tables Generation Using Message Passing Interface: A Study on NTLMv2, MD5, SHA-256 and SHA-512 Cryptographic Hash Functions," *Applied Sciences*, vol. 15, no. 15, 2025, ISSN: 2076-3417. DOI: `10.3390/app15158152` [Online]. Available: `https://www.mdpi.com/2076-3417/15/15/8152`

A Graph representation of zero-hash solver

Before we demonstrate that the zero-hash solver works as intended, we must first explain the motivation for its implementation.

Let us assume we want to find all alphanumeric strings of length 11 that have a zero hash value. If a search graph is constructed as shown in Figure 2, it will closely resemble a complete k-ary tree—a tree where each node has k children and all leaf nodes are the same depth h. The total number of nodes in a complete k-ary tree is calculated by Equation (3), so we will have approximately 6.31×10^{19} nodes in the search graph, given k equal to 63 and h equal to 11.

$$\frac{k^{h+1} - 1}{k - 1} \tag{3}$$

Since we are using the 64-bit FNV-1a hash function for the search graph, the respective nodes would be represented as 64-bit integer values. Thus, the required storage space just for the nodes would be 438EiB. We clearly need a more efficient representation in order to make the implementation tractable on consumer-grade hardware.

Forward and backward trees

Looking at the search graph in Figure 2, one can see that we are unnecessarily creating unused nodes. Since the target node of the search graph is known, we can however construct the search graph using forward and backward trees joined in the middle as shown in Figure 4. According to Equation (3), we will have approximately 2.02×10^9 nodes if we build two separate trees of depth $\lfloor \frac{11}{2} \rfloor = 5$. Thus, the required storage space just for the nodes would now be 15GiB.

Inverse of the intermediate hash function

To build the backward tree, an obvious solution would be to use a preimage table, but the additional required storage space would be prohibitive as the size of the search graph increases. Given the intermediate hash function, $\mathcal{F}$, as defined in Algorithm 2, we will show that $\mathcal{F}$ has an inverse and could be used instead.

Theorem. *For $h, b, p, n \in \mathbb{N}$, $b \leq 255$, p is prime, and $n \geq 8$, the intermediate hash function, $\mathcal{F}(h, b) \equiv (h \oplus b) \cdot p \pmod{2^n}$, is bijective and thus has an inverse.*

Proof. We know that any function is bijective iff it is also invertible.

The function $\mathcal{F}(h, b)$ can be rewritten as the composition of $g(h) \equiv h \cdot p \pmod{2^n}$ and $f(h, b) = h \oplus b$. Since the composition of two bijective functions is also bijective, then we only need to show that $f(h, b)$ and $g(h)$ are both bijective to prove the theorem.

First, $f(h, b)$ is invertible and thus is bijective.

Second, $g(h)$ is a linear congruence and thus has a modular multiplicative inverse iff p and 2^n are relatively prime. Since p is prime, this is true. Thus, $g(h)$ has an inverse and is bijective.

If p^{-1} is defined by the congruence $1 \equiv p \cdot p^{-1} \pmod{2^n}$, then the inverse of the intermediate hash function, $\mathcal{F}$, is

$$\mathcal{F}^{-1}(h, b) \equiv (h \cdot p^{-1}) \oplus b \pmod{2^n} \qquad \square$$

Quotient graphs

We now have a two-terminal graph with 64-bit nodes as shown in Figure 4. Even with the reduced set of nodes, the required storage space for the entire graph, including nodes and edges, would still exceed the available random-access memory on typical consumer-grade hardware. To proceed, a quotient graph can simplify the original graph while preserving the underlying structure.

Let $G = (V, E)$ be a graph and $\sim$ be an equivalence relation on V. For every $v \in V$, let $[v]$ represent the equivalence class of v. The quotient graph of G with respect to $\sim$, denoted by $G/\sim$, is the graph where the vertices are the set of equivalence classes of V and an edge exists between two equivalence classes $[v_1]$ and $[v_2]$ iff there is at least one edge in G joining a vertex in $[v_1]$ and a vertex in $[v_2]$ [1].

Given that a modular congruence, such as $a \equiv b \pmod{2^{32}}$, is an equivalence relation, we can create a quotient graph of the two-terminal graph with 32-bit nodes. In fact, when we use the reduction function, $g : \{0,1\}^{64} \to \{0,1\}^r$, where $r = \{8, 12, 16, 20, 24, 28\}$, in the precomputation phase, we have indirectly created the same quotient graph.