SWIFFT Hash Function

SWIFFT was proposed in a paper by Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen in *SWIFFT: A Modest Proposal for FFT Hashing*. The paper, and this report, also draw extensively from two earlier papers by the same authors:

- Vadim Lyubashevsky and Daniele Micciancio. Generalized Compact Knapsacks Are Collision Resistant
- Chris Peikert Alon Rosen. Efficient Collision-Resistant Hashing from Worst-Case Assumptions on Cyclic Lattices

Three other resources served as useful summaries of the SWIFFT function and lattices:

- Wikipedia. SWIFFT
- Daniele Micciancio, Oded Regev. Lattice-based Cryptography, Sections 3 and 4
- Chris Peikert. EECS 598: Lattices in Cryptography, Lectures 1 and 2

Hash Functions

A hash function is a method of comparing two objects without analyzing the full contents of the object. Suppose there is a website that requires users to login. One option for checking which user is which is to require that each user securely send the website their password, and have the website check to see if the password sent is the same password that was stored from when the user made the account. This is well and good, except suppose the website is breached: someone downloads all of the username and password pairs. If a user reused their username and password pair on another website, the hacker can now try to login using their credentials. One way to prevent this is to hash the passwords: instead of storing the passwords, the user sends it through a hash function, $H(x)$, that scrambles the information about the password and reduces the amount of space it takes to store the password. Now, instead of a user sending their password, they just send the output of the hash function applied to the password. On its own, this hashed password is useless: it just looks like a random collection of numbers, so if it gets leaked the user's password is still secure.

From this useage of hashes, we can deduce two important characteristics of a hash function. First, we don't want anyone to figure out the input of a hash from its output. Suppose your password is "password" and hashes to "abDkj". Good luck guessing the original password from the hash! But what if there is an inverse hash function? Then the attacker can just invert the hashed password to get back to "password." So, we want hashes to be *one-way functions*. Now, suppose that it is very hard to figure out "password" from "abDkj", but it is possible to figure out that the hash function applied to "Hunter2" also produces the

hashed password "abDkj". Well now, instead of finding your password, the attacker can just send "Hunter2" instead, and the website will see these as the same thing. This is the second condition of a hash function: it must be *collision resistant*. That means we don't want anyone to be able to easily find two different passwords that produce the same hashed output, which is called a collision.

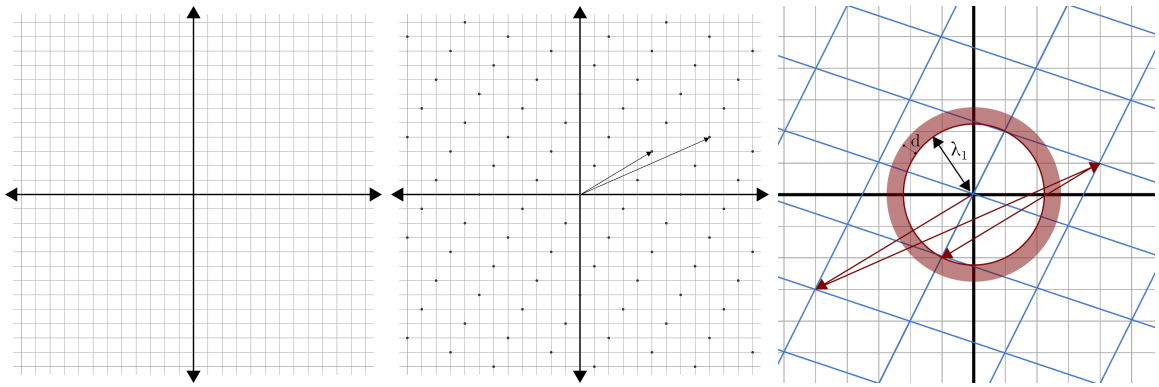
SWIFFT

The SWIFFT hash function has security based on the difficulty of a type of computational problem called a lattice problem. But why use lattice based hashes? Much of modern cryptography is based on the difficulty of two types of problems: discrete logarithm and prime factoring. The value of these problems comes from the fact that, given a certain formulation of an encryption scheme using these methods, it is very easy to take some inputs and compute an output, but it is very hard to take an output and compute what the input. Specifically, for a problem like prime factorization, it is very easy to take two large prime numbers and multiply them together to get a third number. It is very hard to, given the third number, figure out the two input numbers. Or, this is at least true for classical computers. Quantum computers, a class of computer theoretically more powerful than classical computers, are very good at certain types of tasks. Two of these tasks are computing discrete logarithm problems and prime factorizing numbers. While current quantum computers are not powerful enough to pose a risk to modern cryptography, we expect them to be in the future. Additionally, the difficulty of these two problems is based mostly on the fact that we don't know how to quickly factor prime numbers or perform a discrete logarithm on a classical computer. There's nothing that we know of though that proves we can't do it efficiently: only that we don't know how and have reason to believe we won't figure it out. Lattice based problems, which we will describe in this paper, are not known to be easily solvable for quantum computers. SWIFFT in particular is based on the security of a problem that is provably difficult to solve for a classical computer, and for which there is no known quantum algorithm to quickly reverse the hash. Given the current risk quantum computers pose to modern cryptography, it is valuable to focus on algorithms that have a stricter requirement for being secure.

The Approximate Shortest Vector Problem on a Lattice

A lattice, generally denoted $\mathcal{L}(\boldsymbol{B})$, is an object generated by the integer combination of a set of basis vectors. Consider some integer vector $\boldsymbol{x_i} \in \mathbb{Z}^n$. Given m of these vectors, each independent of the others, if we add every possible combination of these vectors with integer coefficients we generate a lattice. From the basis vectors $\boldsymbol{x_0}, \boldsymbol{x_1}, ..., \boldsymbol{x_m}$ we can generate a block matrix $\boldsymbol{B}$ that fully describes the lattice (Hence the notation $\mathcal{L}(\boldsymbol{B})$, the lattice generated by a basis).

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{x_0} & \boldsymbol{x_1} & \dots & \boldsymbol{x_m} \end{bmatrix}$$



(a) The $\mathbb{Z}^2$ Lattice (b) An Irregular Lattice (c) Shortest Vector Problem

Figure 1: **(a)** The $\mathbb{Z}^2$ lattice is the intersections of the grid, which is a subset of $\mathbb{R}^2$. **(b)** An irregular lattice spanned by $(5, 3)$ and $(9, 4)$. Each point can be reached by adding or subtracting one of the two basis vectors. **(c)** The shortest vector problem in the lattice (b). The shortest distance to the origin that is not the point (0,0) can be given as the linear sum of the basis vectors.

In Figure 1*a*, we show the lattice formed by $\mathbb{Z}^2$. We can see that given the basis vectors $[1, 0]$ and $[0, 1]$ it would be be possible to reach any other point in $\mathbb{Z}^2$ by adding together $a \cdot [1, 0] + b \cdot [1, 0]$ with $a, b \in \mathbb{Z}$. Every other lattice is essentially just a lattice in $\mathbb{Z}^n$, but the way to transform a given lattice to $\mathbb{Z}^n$ is non-trivial. Consider another lattice spanned by $[5, 3]$ and $[9, 4]$, shown in Figure 1*b*. Here, the shortest vector is not a member of the basis, and the basis vectors are not orthonormal like in $\mathbb{Z}^2$. Given such a "poor" basis it can be hard to visual and work in the lattice. In particular, there is a problem that has been proven to be very difficult for computers to solve: the approximate Shortest Vector Problem (SVP). In order to solve the problem given $\mathcal{L}(\boldsymbol{B})$, we want to produce a point that is within some multiple d of the closest point on the lattice to the origin that is not the origin. Each lattice contains the origin, so that solution would be trivial. The value d allows some flexibility in the solution, but it was proven by Miklós Ajtai in 1998 [1] that given a randomized basis, finding the approximate SVP under the Euclidean norm is NP-Hard. This is a foundational result because it means that SVP is about as hard a computational problem as we can get. There is a class of problems called NP, which is the set of all problems that we can check the answer to easily, but for which we cannot easily compute the answer; the existence of such problems is foundation of cryptography. Generally, NP-Hard problems are at least as hard to solve as the hardest of the known problems in NP. While it is still unproven whether there are indeed any hard to compute problems (P = NP), it is generally assumed that there do exist certain problems that cannot be solved on a computer in a reasonable amount of time.

Here, reasonable means a problem that, given a size of the problem n, can be solved in a time that is polynomial in n —$\mathcal{O}(n^p)$ —rather than exponential $\mathcal{O}(p^n)$.

There do exist hash functions that make direct use of approximate SVP, but there is a catch. We care immensely about the difficulty of computing a hash function in relation to the security it provides. This security can generally be thought of as, given some input size n, how many inputs would an attacker have to try before they find a collision? However, dealing with general, random lattices require dealing with an $n \times n$ matrix $\boldsymbol{B}$. Operations using this matrix through will require $\mathcal{O}(n^2)$ operations to run. These types of hashes provide poor scaling in security for the difficulty of computation, although they are about as secure as it is possible to prove.

SWIFFT makes use of a certain type of lattice known as an *ideal* lattice. A lovely property of these lattices is that they can be described using only one vector and a set of operations on that vector. A particular case of an ideal lattice is the cyclic lattice shown in Figure 2. Here, for each point on the lattice, the cyclic permutations of the point are also on the lattice. In red, we have shown a nice basis for a cyclic lattice in $\mathbb{Z}^2$, in that the basis vectors are permutations of the coordinates. For example, in $\mathbb{Z}^3$, if the lattice contains the point $(1, 2, 3)$ it also contains the points $(2, 3, 1)$ and $(3, 1, 2)$. In black, we have shown a worse basis, in which the shortest vector is non-trivial and is not a basis vector. Cyclic lattices are a special class of a general ideal lattice, where an ideal lattice could be generated by, for example, the skew-circulant basis that cycles the coordinates of each vector on the lattice but negates each point that wraps-around is

$$\boldsymbol{B} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & -1 \\ 3 & -1 & -2 \end{bmatrix} \tag{1}$$

This particular type of ideal lattice will be useful in the construction of the SWIFFT hash, because we can describe an entire lattice using only one vector.

Unfortunately, the efficiency of the SWIFFT algorithm comes at the cost of a strict security proof. Atjai's proof that approximate SVP is NP-Hard applies to randomized lattices and does not include any results about how adding structure to the lattice by requiring that it be ideal affects the hardness of the problem. However, the authors of SWIFFT assert that, although it has not been prove that approximate SVP on an ideal lattice is NP-Hard, there is no known way to exploit an ideal lattice to find the shortest vector faster than on a general lattice. Specifically, there are many ways that we know to find the shortest vector in exponential time, but none of these methods are faster in the cases of ideal lattices, as long as certain requirements about the ideal lattice are met [2, 3].

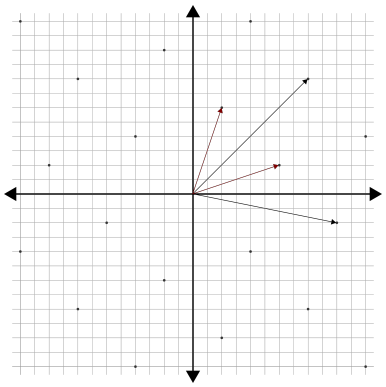


Figure 2: Two vector bases on the same cyclic lattice

SWIFFT Hardness Assumption Definitions

We define the Shortest Vector Problem on a Lattice $\mathcal{L}(\boldsymbol{B})$ and the approximate Shortest Vector Problem using definitions found in [6].

1. **Minimum Distance on $\mathcal{L}(\boldsymbol{B})$:** The minimum distance on $\mathcal{L}(\boldsymbol{B})$ is given as $\lambda_1(\mathcal{L}(\boldsymbol{B}))$ and is the distance between the origin of the lattice and the closest point on the lattice to the origin. Formally, this is $\lambda_1(\mathcal{L}(\boldsymbol{B})) := \min_{v \in \mathcal{L}(\boldsymbol{B}) \setminus \{0\}} \|v\|$.
2. **Shortest Vector Problem (SVP):** Given $\mathcal{L}(\boldsymbol{B})$, find $v \in \mathcal{L}(\boldsymbol{B})$ such that $0 < \|v\| = \lambda_1(\mathcal{L}(\boldsymbol{B}))$.
3. **Approximate Shortest Vector Problem (SVP_γ):** Given $\mathcal{L}(\boldsymbol{B})$, find $v \in \mathcal{L}(\boldsymbol{B})$ such that $0 < \|v\| = \gamma \cdot \lambda_1(\mathcal{L}(\boldsymbol{B}))$.

The definition of the security assumption for SWIFFT is based on definitions in [5].
Given parameters n, m, and p such that n is a power of 2, p is prime, and $2n|(p-1)$ we select a basis of vectors $\{\boldsymbol{a_0}, ..., \boldsymbol{a_{m-1}}\}$ with $\boldsymbol{a_i}$ selected uniformly randomly from $\mathbb{Z}_p^n$ and define the $n \times n$ skew-circulant matrix

$$\boldsymbol{F} = \left[\begin{array}{c|c} \boldsymbol{0}^T & -1 \\ \hline \boldsymbol{I} & \boldsymbol{0} \end{array}\right]$$

Given the block matrix $\boldsymbol{A}$ defined as

$$\boldsymbol{A} = \left[\begin{array}{ccccc|ccc|c|ccc} \boldsymbol{a_0} & \boldsymbol{F a_0} & \boldsymbol{F^2 a_0} & \dots & \boldsymbol{F^{n-1} a_0} & \boldsymbol{a_1} & \dots & \boldsymbol{F^{n-1} a_1} & \dots & \boldsymbol{a_{m-1}} & \dots & \boldsymbol{F^{n-1} a_{m-1}} \end{array}\right]$$

it is NP-Hard to solve SVP_γ in the lattice basis defined as

$$\Lambda_p^\perp(\boldsymbol{A}) = \{\boldsymbol{y} \in \mathbb{Z}^{mn} : \boldsymbol{A y} = \boldsymbol{0} \bmod \text{q}\}$$

Implementation of the SWIFFT compression Hash Function

Consider some input binary string with length mn, where m and n are two fixed parameters of the function. For example, to hash a 1024 bit input, we can choose values $n = 64$ and $m = 16$. We require that n be a power of 2. In the discussion of the algorithm, we will use the parameters $n = 2$, $m = 4$ which allows us to hash a byte. We take as the input of our function the binary string representing the letter $F = 0100\ 0110$. We must also choose a parameter p, which must be prime and for which $p - 1$ is a multiple of $2n$. Here, we choose $p = 5$ which satisfies $5 - 1 = 4 = 2 * 2$.
The **first** step is to assemble the input string $\boldsymbol{x}$ into an $n \times m$ binary array. For our example, this gives

$$\boldsymbol{x} = F = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

However, it is more convenient to write this input in block form with vectors $\boldsymbol{x}_i \in \mathbb{Z}_2^n$, where here use use $\mathbb{Z}_k$ to denote the group of integers 0 to $k - 1$ under addition modulus k:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x_0} & \boldsymbol{x_1} & \boldsymbol{x_2} & ... & \boldsymbol{x_m} \end{bmatrix}$$

The **second** step is to choose some random set of vectors which will form our key. We choose a total of mn parameters from the group $\mathbb{Z}_p$ and again arrange them in a block matrix format:

$$\boldsymbol{a} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & .. & a_{0m} \\ a_{10} & a_{11} & a_{12} & .. & a_{1m} \\ \vdots & & & & \vdots \\ a_{n0} & a_{n1} & a_{n2} & .. & a_{nm} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a_0} & \boldsymbol{a_1} & \boldsymbol{a_2} & ... & \boldsymbol{a_m} \end{bmatrix}$$

For our example, we choose 8 elements from $\mathbb{Z}_5$:

$$\boldsymbol{a} = \begin{bmatrix} 1 & 3 & 0 & 2 \\ 2 & 4 & 1 & 3 \end{bmatrix}$$

The **third** step is to choose a parameter we call ω. It is worth mentioning some useful properties of the group $\mathbb{Z}_p$, where p is prime. First, it is a ring. This means that $\mathbb{Z}_p$ is closed under multiplication and addition modulus p (among other properties). It is also a field because each element (except for 0) has a multiplicative inverse in $\mathbb{Z}_p$. There is a subring of elements with multiplicative inverses under $\mathbb{Z}_p$ that has exactly $p - 1$ elements. This is because we do not include 0, because it does not have a multiplicative inverse and because it is impossible to multiply two non-zero numbers to get zero. This subring commutes under multiplication and contains an identity, so we can apply the Fundamental Theorem of Abelian Groups, which states that if a number divides the number of elements in the group, in our case the number of elements is $p-1$, then there exists some set of numbers such that $\omega^{2n} = 1$ [7]. This is our requirement for ω, and motivates the choice of parameters p, n, and m. The group generated by ω, which is the set of elements which are unique and can be created by

ω^k for some $k < 2n$ has $2n$ elements.
For our example construction, we choose $\omega = 3$, which has order $4 = 2n$: $3 \times_5 3 = 9 \bmod 5 = 4 \times_5 3 = 12 \bmod 5 = 2 \times_5 3 = 6 \bmod 5 = 1$, where we have used $\times_p$ to refer to multiplication modulo p.
The **fourth** step is to construct what is known as the Vandermonde matrix of the odd powers of ω. The matrix, which we call $\boldsymbol{W}$, is constructed as:

$$\boldsymbol{W} = \begin{bmatrix} 1 & \omega & \omega^2 & \omega^3 & ... & \omega^n \\ 1 & (\omega^3) & (\omega^3)^2 & (\omega^3)^3 & ... & (\omega^3)^n \\ 1 & \vdots & & & & \vdots \\ 1 & (\omega^{2n-1}) & (\omega^{2n-1})^2 & (\omega^{2n-1})^3 & ... & (\omega^{2n-1})^n \end{bmatrix}$$

which is an $n \times n$ matrix. In our example, this gives the matrix

$$\boldsymbol{W} = \begin{bmatrix} 1 & 3 \\ 1 & 3^3 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix}$$

The **fifth** step of the algorithm, which is the most important, is to compute the output of the hash function. The output is a vector $\boldsymbol{z} \in \mathbb{Z}_p^n$, given as the output

$$\boldsymbol{z} = \Sigma_{j=0}^{m-1} \boldsymbol{a_j} \odot \boldsymbol{W x_j}$$

Here, we use the notation $\odot$ to refer to the component-wise multiplication between the two vectors $\boldsymbol{a_j}$ and $\boldsymbol{W x_j}$ which is called the Hadamard product.
Continuing our example, split the summation into four parts:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \times_5 0 \\ 2 \times_5 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} \odot \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \odot \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \times_5 4 \\ 4 \times_5 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \times_5 3 \\ 2 \times_5 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \times_5 0 \\ 3 \times_5 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The sum of all these components gives the vector

$$\boldsymbol{z} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

which can be represented as 2 bits of length 3. So, we have compressed our input of 8 bits to an output of 6 bits. For more practical parameters $m = 16$, $n = 64$, $p = 257$, the function

reduces an input of size 1024 bits to an output that can be represented by 528 bits.

There are two important features of this step that are essential to the security of the hash function. First, multiplying the input vectors by $\boldsymbol{W}$ acts to diffuse the information in $\boldsymbol{x}$ by mixing the inputs across the column space. Second, the summation over the key $\boldsymbol{a}$ serves to compress the input message in a way that is difficult to undo, and to combine the diffused inputs into one output.

The fifth step as described here omits an important computational result. The most expensive component of the hash in terms of run-time scaling with the input size is that the product $\boldsymbol{W}\boldsymbol{x_j}$ generally requires n^2 computational time for an $n \times n$ matrix $\boldsymbol{W}$. However, we can also think of this multiplication the evaluation of a polynomial $\boldsymbol{x_j}$ at the points $\omega, \omega^3, ...$ If we expand out the matrix product for n = 3, we get

$$\boldsymbol{W}\boldsymbol{x_j} = \begin{bmatrix} 1 & \omega & \omega^2 \\ 1 & \omega^3 & (\omega^3)^2 \\ 1 & \omega^5 & (\omega^5)^2 \end{bmatrix} \begin{bmatrix} (x_j)_0 \\ (x_j)_1 \\ (x_j)_2 \end{bmatrix} = \begin{bmatrix} (x_j)_0 + (x_j)_0(\omega) + (x_j)_0(\omega)^2 \\ (x_j)_1 + (x_j)_1(\omega^3) + (x_j)_1(\omega^3)^2 \\ (x_j)_2 + (x_j)_2(\omega^5) + (x_j)_2(\omega^5)^2 \end{bmatrix}$$

The value of this algorithm then is that we can apply a Fast Fourier Transform (FFT) to evaluate this matrix product in $\mathcal{O}(n \ln n)$ evaluations of a function rather than $\mathcal{O}(n^2)$ as we would initially assume. We will not elaborate on the usage of the Fast Fourier Transform to evaluate the matrix product here, but it is ultimately what allows the usage of the lattice evaluation on nearly linear with n time rather than quadratic in n time. Because FFT is implemented efficiently at the hardware level in a variety of processors architectures, SWIFFT can be run quickly on a variety of device types.

Security

The security of the Hash, and the collision resistance, is reduced to the difficulty of finding a short vector on an ideal lattice. This is provably NP-Hard on a general, randomized lattice, but has not been proven for an ideal lattice. However, the known attacks on a randomized lattice are not easier to perform on an ideal lattice, so there are no known ways to exploit the structure of the lattice in the worst case. The worst case implies that we are allowed to put constraints on the types of lattices used in the problem. In the formulation of the SWIFFT algorithm, the authors make use of a specific lattice construction which consists of lattices constructed as the skew-circulant product of other points on the lattice (as shown in Eq. 1). Additionally, the constraints on n, m, and p prevent certain types of attacks.

In the paper which introduces SWIFFT, the authors provide bounds on the difficulty of attacking SWIFFT using known attacks with the parameters defined previously on a 1024 bit input. The resulting vector $\boldsymbol{z} \in \mathbb{Z}_p^n$ has a total of p^n possible values. Using the birthday attack, it is possible to construct an algorithm running in time 2^{106} [4] that produces a

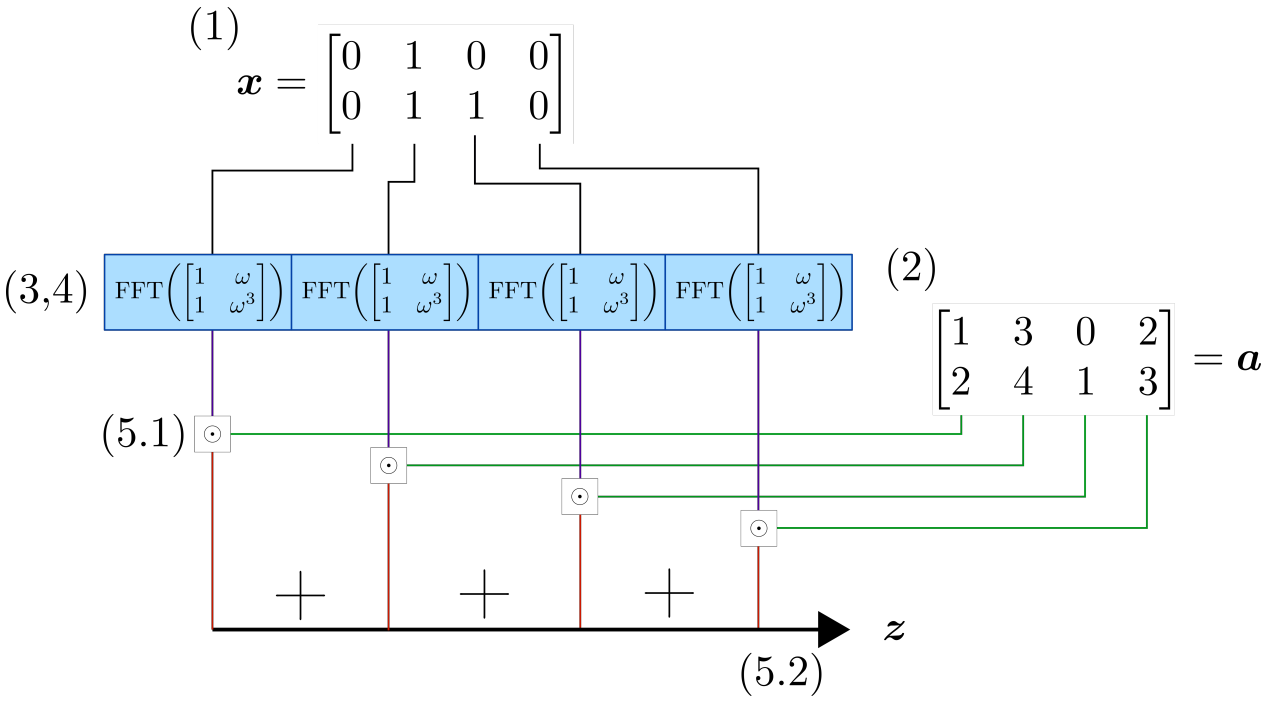


Figure 3: Example Implementation of the SWIFFT Hash Function

collision. This is an infeasible runtime. Additionally, it is possible to invert the hash using the security parameters defined above and in the paper in time 2^{448}, which is very infeasible.

Proof Strategy

The hardness of SVP_γ was proven by [1] by reducing an easier variant of SVP_γ, known as the decision problem, to a known NP-Complete function called the restricted subset sum problem. Thus, as the SVP_γ is harder than the decision problem on a lattice [6], SVP_γ for the worst case-problem in general lattices is NP-Hard. This work was the inspiration for the construction of SWIFFT.

The precursors to SWIFFT rely on on a reduction of the Generalized Knapsack Problem to the shortest vector problem on an ideal lattice [2, 3]. The generalized knapsack problem is constructed by taking some vector $\boldsymbol{a}$ in the space spanned by R^m, where R is a ring (closed under addition and multiplication, among other properties), some target value z in the ring, and finding some new vector $\boldsymbol{x}$ in the space spanned by S^m, where S is a subset of the ring R, so that $\Sigma_{i=0}^{m-1} a_i \cdot x_i = z$. This problem is not universally hard, depending on the choice of ring and of the subset [2]. However, it has been demonstrated that for the choice of ring $R = \mathbb{Z}_p[x]/\langle x^n + 1\rangle$, and by restricting the S to be polynomials with binary coefficients in R, we are able to construct a collision resistant function. Here, $\mathbb{Z}_p[x]$ refers

to the set of polynomials with degree n and with coefficients that are an element of the $\mathbb{Z}_p$. The notation $R = \mathbb{Z}_p[x]/\langle x^n + 1\rangle$ refers to a quotient ring. Due to a property called canonical embedding [4], the polynomials can be mapped to ideal lattices. So, the reduction is from the Generalized Knapsack Problem to the SVP_γ problem on the corresponding ideal lattice: solving the Generalized Knapsack problem provides a solution to SVP_γ, so while we assume SVP is NP-Hard on ideals, we must assume that the Generalized Knapsack problem as constructed must not have a polynomial time solution. The hardness corresponds to the two problems we care about for hash functions. Finding a collision pair in the generalized knapsack problem means finding two different values of $\boldsymbol{x}$ that map the same z under some fixed $\boldsymbol{a}$. Finding a collision pair allows for finding a short vector, so it cannot be efficient. Collision resistance is stronger than a function that only has the property that it is one way, so the inability to find a collision precludes inverting the knapsack problem. The SWIFFT hash function inherits the properties proved in [2, 3].

References

1. Ajtai, M. *The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract)* in *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (Association for Computing Machinery, New York, NY, USA, May 1998), 10–19. ISBN: 978-0-89791-962-3. https://doi.org/10.1145/276698.276705 (2021).
2. Lyubashevsky, V. & Micciancio, D. en. in *Automata, Languages and Programming* (eds Hutchison, D. *et al.*) Series Title: Lecture Notes in Computer Science, 144–155 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). ISBN: 978-3-540-35907-4 978-3-540-35908-1. http://link.springer.com/10.1007/11787006_13 (2021).
3. Peikert, C. & Rosen, A. en. in *Theory of Cryptography* (eds Hutchison, D. *et al.*) Series Title: Lecture Notes in Computer Science, 145–166 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). ISBN: 978-3-540-32731-8 978-3-540-32732-5. http://link.springer.com/10.1007/11681878_8 (2021).
4. Lyubashevsky, V., Micciancio, D., Peikert, C. & Rosen, A. en. in *Fast Software Encryption* (ed Nyberg, K.) ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science, 54–72 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008). ISBN: 978-3-540-71038-7 978-3-540-71039-4. http://link.springer.com/10.1007/978-3-540-71039-4_4 (2021).
5. Micciancio, D. & Regev, O. Lattice-based Cryptography. en, 33 (2008).
6. Peikert, C. & Carter, H. Lecture 2 SVP, Gram-Schmidt, LLL. en, 5. https://web.eecs.umich.edu/~cpeikert/lic15/lec02.pdf (2015).
7. Gallian, J. A. *Contemporary abstract algebra* Undefined/Unknown. ISBN: 978-1-305-65796-0. https://experts.umn.edu/en/publications/contemporary-abstract-algebra (2021) (Cengage Learning, 2017).