# Side Channel Resistant Sphincs+

Scott Fluhrer

Cisco Systems
March 28, 2024

**Abstract**

Here is a potential way to create a SLH-DSA-like[Nat23b] key generation/signer that aspires to be resistant to DPA side channel attacks. We say that it is "SLH-DSA-like", because it does not follow the FIPS 205 method of generating signatures (in particular, it does not have the same mapping from private key, messages, opt_rand to signatures), however it does generate public keys and signatures that are compatible with the standard signature verification method, and with the same security (with a small security loss against side channel attacks). In our tests, this idea performed 1.7 times slower compared to an unprotected version.

Keywords: Postquantum Signatures, Sphincs+, SLH-DSA, Side Channel

## 1 Introduction

Typical implementations of Sphincs+ are immune to timing and cache-based side channel attacks. The obvious implementation makes no conditional jumps or memory references based on secret data[1], as long as the hash function implementation do not (and the standard implementations of SHA-2 and SHAKE do not when hashing messages of fixed length). However, that still leaves the possibilities of more subtle attacks, such as DPA (Differential Power Analysis).

### 1.1 Differential Power Analysis

In a DPA attack[KJJ99], the attacker listens to the electrical noise from the internal gates of the device (via current draw or EMF) while processing secret information; the data collected contains information about the secrets the attacker is interested in, combined with noncorrelated noise. The attacker does this data collection a large number of times, and then uses statistical methods to reduce the noise and then is able ti extract the secrets.

In SLH-DSA, the SK.seed is used as a input to the PRF function, with all other inputs being public, and varying per use. This situation is ideal for an attacker using DPA; he can collect the requisite large number of uses of SK.seed, and use the known varying inputs to isolate the settings of specific secret bits from other secret bits. And, when the attacker learns enough bits, he can recover SK.seed and can sign any message he wants.

The standard approach to produce a DPA-resistant implementation is to use thresholding[CJRR99]; for any internal value that needs to be secret, we do not represent that value explicitly. Instead, we split it up into N 'shares', so that the logical value of that secret depends on all N shares (so that if the attacker learns N-1 of the shares, he still has no information about the secret being represented). One common way is to have the logical value be the xor of all N shares. A specific logical value can be represented by a number of bit patterns; we select the one we use uniformly. The combination method is typically linear, if the operation performed on the logical value is linear, we can perform that operation on the individual shares. However, if a nonlinear operation is required between two secret values (such as a AND operation), we end up performing $N^2$ operations between the two sets of shares, and then stir in randomness (to preserve the uniformity property).

[1] It isn't necessarily constant time; a Sphincs+ implementation may have conditionals based on, for example, the part of the tree we are exposing, or the revealed index within a Winternitz chain. However, that data can also be deduced by the attacker by examining the signature, message and public key, and hence these conditionals do not leak any secret information.

## 2 Our Approach

We will not be using the standard approach in this proposal[2] Instead, what we will do is use any secret value in only a handful of contexts. The attacker can see a secret value used many times (by monitoring multiple signature generation operations), but still he will always be limited to these limited number of contexts. Because of this, the noise generated during the operation will be consistent between multiple invocations, and so the attacker will not be able to use statistical methods to reduce it.

For a simple example: suppose that we have two secret values, for example, 0x8D6D53217824C570 and 0x460DEF2D0EB718EB, and xor them together. If the xor is performed in a single cycle by a 64-bit ALU, then the attacker should be able to obtain only partial information about those values or their xor; he may get an estimate of their hamming weight or the hamming weight of the xor (for example), but he would be unable to obtain those exact values (or even limit the possible values to a relatively small set). And, if our implementation were to repeatedly xor those exact same values, and the attacker would take measurements of them each time, he still would not be able to get more than partial information.

In a case which is more similar to what we actually propose, consider the case that we generate an internal secret $X = SHAKE(Y)$, and use it to generate two child secrets $Z0 = SHAKE(ADRS0|X)$ and $Z1 = SHAKE(ADRS1|X)$, and we never use $X$ in any other context. Then, if the attacker wants information on $X$, he can get information from the last rounds of $SHAKE(Y)$, and the initial rounds of $SHAKE(ADRS0|X)$, $SHAKE(ADRS1|X)$; however that is the only information available to him about $X$. If the information he can extract from those samples do not give him sufficient information about some bits of $X$ to overcome the noise from the other operations that occur that those times (for example, other bits of $X$), then he cannot recover $X$.

This is the property we rely on. We assume that the implementation performs its operations with a large (32 bit or better yet, 64 bit) CPU, and does not perform operations on a handful of bits, and the operations involved (which typically are limited to logical xor, and and rotate by fixed amounts for the hash function in question) do not have large operand-dependent power consumption variations.

There have been SPA attacks (which relies on a single collected sample) which have recovered secrets; these are against public key implementations that process a limited number of secret bits (possibly only one) at a time, and also do involve large power consumption variations. We assume that our hash implementations will process a number of bits at a time (e.g. 32 or 64), and that the attacker will not be able to isolate the individual bits.

That said, as observed earlier, the attacker will still be able to extract some information about $X$ (for example, perhaps the hamming weight of a subsequence of bits of $X$). If $X$ is $N$ bits, then the attacker would be able to use this partial information to recover $X$ in less than $2^N$ operations. To maintain full security, we will need to address that as well.

This alternative approach to DPA resistance would appear to have both advantages and disadvantage:

- One advantage may be that this may have the potential[3] to give better DPA protection. With the current practice, the correlations are still there; it just requires the attacker to access a large number of samples to access them (and ideally, more samples than what is available). With this system, ideally multiple samples do not give the attacker any more information (and so it suffices to design the system to be secure against SPA attacks).

- Another advantage is that this approach does not require the implementation to produce a large number of random values during operation. With the standard approach to DPA protection, this is required (and often consumes a significant amount of time to do so).

- Another potential advantage is that this usage is consistent with a single fixed private key. With the standard threshold approach, we may need to avoid the possibility of the attacker listening to early operations of the private key, and hence we may need to keep the private key in threshold format, and refreshing it each time (so there are no fixed bits for the attacker to learn) – doing

[2]We will be using something similar to this approach in our protection of the F function, however even there, the security argument still relies on everything being deterministic.

[3]We put in the qualifiers "may be that this may have the potential" because, while we believe the potential is there, this idea needs more study.

this would require storing the key in nonvolatile RAM. This NVRAM requirement is also a burden on the implementation; this approach avoids that.

- One disadvantage is that it may be difficult to assess the protection. It would appear likely that the standard tools for assessing DPA resistance may not work. They essentially check to see if the internal values have no measurable correlations – with this approach, the internal values will have correlations between runs (the values will be consistent from run to run); it's just that there won't be exploitable correlations.

- One disadvantage to this strategy for DPA resistance in general is that it may be unclear how to protect an arbitrary cryptosystem with this approach; we may need to redesign the system to use this protection. The SLH-DSA architecture happens to be well suited for it; it is unclear how we would use this to protect (for example) ML-KEM[Nat23a]. Even for SLH-DSA, we needed to redesign how the PRF works to be able to take advantage of it; it is unclear how we would use this design to protect the existing SLH-DSA PRF definition. The modified PRF would appear to work well and be secure, however the private key is not compatible with a standard SLH-DSA implementation.

# 3 The secrets within Sphincs+

In order to design DPA protection for Sphincs+, we first need to understand the secrets we need to protect.

The SLH-DSA signing process[4] consists of several steps. We also list the potential secrets that may be leaked at each step:

- The message is hashed ($PRF_{msg}$) with a secret $SK.prf$ value and an optional randomness value.

  While the result of this hash is public (the value $R$ is included in the signature), the value $SK.prf$ needs to remain secret. If that value is known and either the randomness value is predictable or omitted, then system becomes vulnerable to several chosen message attacks which are easier than the targeted security level.

- The message is again hashed ($H_{msg}$), along the above hash $R$, and other public data to select the FORS leaves, the bottom most Merkle tree, and the bottom most Merkle leaf within that tree.

  No secrets are involved here.

- The PRF function[5] is used to convert the SK.seed value into the selected FORS secret values.

  The SK.seed value and the FORS secret values are secret.

- The $F$ function is used to convert the FORS leaf secret value into FORS leaf values.

  The FORS secret values are secret; the FORS leaf values are not.

- The FORS leaf values are used to generate a Merkle tree using the $fors_node$ function, and then combined into a single value using the $T$ function.

  No secrets are involved here.

- The $PRF$ function is used again to convert the SK.seed value into WOTS initial values.

  The SK.seed and the WOTS initial values are secret.

- The $F$ function is used to advance from the WOTS initial value to the final chain top.

  These values are secret (some are exposed, but we won't know which ones will be, except that the final value is always exposed).

[4]The key generation process consists of a subset of these steps, so that is not listed separately.
[5]The FIPS 205 draft refers to this as the $fors_{SKgen}$ function when used in this context.

- These chain top values are combined with the $T$ function into a single value.

  No secrets are involved here. Note that, even though the intermediate chain values are marked as 'potentially secret', the top value never is.

- These values are used to generate a Merkle tree with the $H$ function.

  No secrets are involved here.

All values listed as 'not secret' are values that, if the adversary learns them, does not give him an advantage. These are either values that are also computed by the verifier (and so the adversary has them), or internal nodes within a Merkle tree.

Summarizing this list, the secrets that we need to protect are:

- The secret input to the $PRF_{msg}$ function.
- The inputs and outputs to the $PRF$ function.
- The inputs and (sometimes) outputs to the $F$ function.

For everything else, we can use a straight-forward (unprotected) implementation.

# 4 Our proposal

Given this list of secrets we need to protect, here are the details of our proposal.

## 4.1 Note about parameter sets

For this proposal, we will limit the parameter sets supported to the SHAKE[Nat15] parameter sets. This is for multiple reasons:

- SHAKE has an impressively large (1600 bit) internal state, and in the middle rounds, the state is effectively uniformly random. That means that we do not have to worry about leakage in the intermediate rounds of SHAKE – unless he can recover almost all that state (1344 bits for n=256), he cannot exploit that.
- SHAKE allows longer inputs and outputs (up to 167 bytes if we use SHAKE-128) without notable performance impact. We will exploit this to help protect the PRF generation.
- SHAKE is threshold friendly. That is, it is straightforward (and relatively inexpensive) to implement a thresholding implementation, which we will use to protect the F function.

In addition, while it would be straight-forward to support this idea with the robust SHAKE parameter sets, we don't bother with explaining the details. This is largely due to the fact that the SLH-DSA standard does not include them (and those details are fairly obvious).

## 4.2 Protection for SK.prf

Actually, we propose to not protect that at all. Instead, we remove the need for it to be protected.

The secret input SK.prf is there to address the case that the random input opt_rand is predictable, or omitted; if the attacker learns SK.prf, and the opt_rand is predictable (or omitted), then there are several chosen message attacks the attacker can perform, either a simple collision attack (finding two messages whose $H_{msg}$ are the same) or looking for (and requesting to be signed) a large number of messages that $PRF_{msg}$ specifies the same FORS (exposing all the secrets in that FORS)

We address this by specifying that, for this implementation, this random input opt_rand[6] must not be omitted and must not be predictable. In that case, the attacker may be able to recover the SK.prf value, but that does not give him any advantage; he still cannot conduct either attack (because the $R$ value is still unpredictable), and the SK.prf value is not used anywhere else. We would also note that injecting randomness here does not interfere with the 'make everything deterministic' strategy used for the other secrets.

Other approaches that were considered (and rejected) can be found in Appendix B.

[6] Which is now misnamed - opt_rand is no longer "optional".

### 4.3 Protection for PRF inputs and outputs

We first note that the method that we use to generate PRF outputs is completely opaque to the verifier. These outputs are generated as a function of the private key. Hence, if we don't mind changing how the private keys are interpreted, we are free to completely redesign how these values are generated, as long as we abide with the constraints that the method must be deterministic and meets the security level.

In SLH-DSA, we use a single function to generate all PRF outputs, with the only distinction between the different contexts is the value of the ADRS structure. In this proposal, we use a different approach. Instead of reusing the same SK.seed value repeatedly with different associated inputs, we will instead design a structure (somewhat analogous to the tree structure of Sphincs+). Specifically, we will create a series of 4-way (rather than binary) trees[7], and extract the PRF outputs from various leaf values of those trees. We will refer to this structure as a PRF tree.

This approach is illustrated in Figure 1. On the left side of the figure, there is the standard Sphincs+ structure, with the FORS trees at the bottom, and a series of Merkle trees (and one time (WOTS) signers) above that, forming the hypertree. In this structure, both the FORS trees and the WOTS consume PRF values. However, instead of calling a single PRF function with different ADRS values, they obtain these values from the structure on the right.

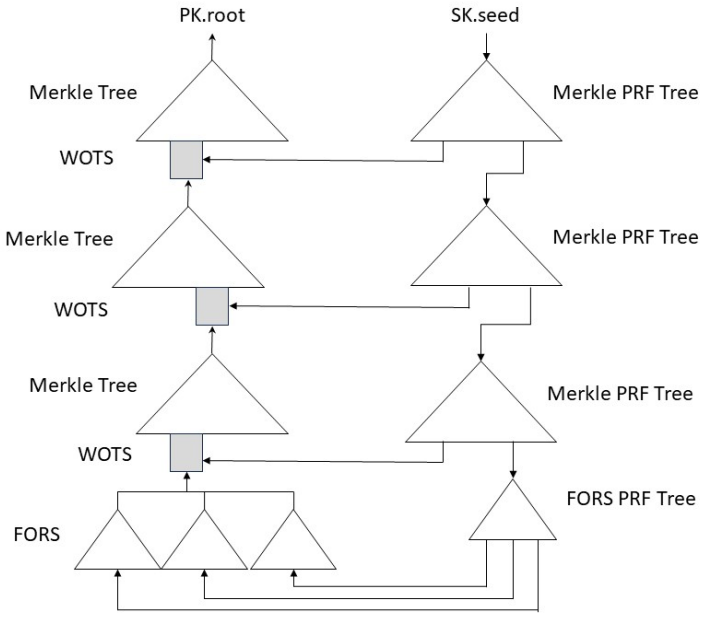


Figure 1: Overview of Proposed Sphincs+ PRF Architecture

The structures on the right have a key difference from those on the left. In the standard Sphincs+ structure, the data flows upwards - every node has a value, and that value is determined by the child values below it. In the PRF structure, the data flows downwards - every node still has a value, but that value is determined by a parent value above it.

Each PRF tree takes one value (either from the PRF tree directly above it, or for the top PRF tree, the SK.seed value), and expands that a number of derived secrets.

Each "Merkle Tree PRF" derives $(wdigit + 1)2^{h'}$ secrets, where $wdigit$ is the number of chains within a single WOTS signature[8], and $h'$ is the height of a single Merkle tree. $wdigit \cdot 2^{h'}$ of these secrets are used as PRF values for the $2^{h'}$ WOTS signers under the Merkle tree, and the remaining $2^{h'}$ are used as seeds for the $2^{h'}$ PRF trees immediately below.

[7]We use a four-way tree to increase the efficiency (while keeping the number of times any specific internal secret is used to a reasonable level).

[8]This is 35 for Level 1 parameter sets, 51 for Level 3 and 67 for Level 5 parameter sets. In section 5, the FIPS 205 draft refers to this value as *len*, however we felt that this was too generic a name for wider usage.

Each "FORS PRF" derives $k2^a$ secrets, where $k$ is the number of FORS sets (individual trees), and $a$ is the height of each tree. These values are used as PRF values for the corresponding FORS structure.

The value of each internal node will be used in only five contexts; once when the internal node is generated, and four times to generate the four child nodes. We will use an unprotected hash implementation to generate the children. Because of this, DPA attacks will be able to collect only limited information about any internal node value. If we store the values of internal nodes, the performance cost of generating all the secrets at a specific level (say, for all the FORS or WOTS+ secret values), is only a factor of $\frac{4}{3}$ more than the standard PRF method.

Now, DPA attacks may be able to collect some partial information (based on the five usages we have), and so what we do is make each internal value $3N$ bytes long (rather than $N$ bytes long). This implies that, unless the attack can obtain more than $2N$ bits of partial information on an internal value, he will not be able to use that partial information to perform an attack with any advantage. This implies that the top level value (the SK.seed value) is also $3N$ bits. Because the internal rate of SHAKE-128 is more than 3N plus the size of the ADRS structure (plus SHAKE-128 padding), this increase of the internal length does not impact performance.

This increase in the size of SK.seed means that we have private keys that are incompatible with standard SLH-DSA implementations. However, because we are changing how the SK.seed value is used, the change in the field size is relatively minor.

The main reason we specify that the internal values be $3N$ bits long is that the $F$ function specified below works with values that size (as outlined in section 4.4) and it was more straightforward to make all internal values of the PRF system that same size. And, while it is overkill in preventing any DPA style attack from learning enough about PRF internals to perform an attack with less effort than the security level, cheap overkill is not a bad thing.

Further details of the proposal can be found in Appendix A.

### 4.4 Protection for F inputs and outputs

Unlike the PRF, the F function is visible to the verifier, and hence we must implement the same functionality. Even though any input to the F function is used only once, using the straightforward implementation will still leak some information (such as the hamming weights of some sequences of bits of a secret value), and so we still need to address that.

However, assuming that the measurements of the Hamming weights are sufficiently accurate, a small amount of information is still leaked. This small amount of leakage may lead to a loss of perhaps 2 to 8 bits of security strength; that is, the attacker might be able to get enough information to perform an attack between 4 to 256 times faster than otherwise. The details of this are further explored in section 5.1.

The definition of the F function for SHAKE parameter sets is $SHAKE256(Public \| M, N)$, where $Public$ is publicly known data (PK.seed, ADRS), and $M$ is the secret input to the F function, and $N$ is the size of the internal values of Sphincs+, which is the security level.

To implement this, we start with a 3-way[9] threshold implementation of SHAKE, where the logical state is represented by the xor of the three physical shares. The shares of the Public parts are initially unblinded (the attacker knows them anyways); for M, we will take a $3N$ byte input, parse it into three inputs of $N$ bytes each, and use those as the three shares. That is, the logical input to the F function will be the xor of three bit strings of length N. When we need to output an F function input as a part of the signature, we will explicitly compute that xor.

We first perform two rounds of Keccak on that threshold implementation. After those two threshold rounds, we have the inputs states array[0], array[1], array[2]. We unblind the arrays array[0] := array[0] xor array[1] xor array[2]) (but don't throw away array[1], array[2] quite yet).

If the output of the F function is public (which it is in a FORS tree, or the top level WOTS value), we perform the final 22 SHAKE rounds using a standard Keccak round implementation, and output

[9]Why 3-way? Well, with a traditional 2-way implementation of SHAKE, the real value is the xor of the two shares. We don't place strong constraints on either the compiler or the CPU implementation; hence it is possible that the two would conspire to send those two shares consecutively over the same internal bus. The bits on that bus will either transition or not transition based on the xor of the two shares, that is, based on the real value, potentially exposing that to a side channel attack. By making it a 3-way threshold scheme, we ensure that potentially leaking information about the xor of two of the values is not catastrophic.

the unblinded result.

If the output of the F function is private, then we perform 20 intermediate SHAKE rounds using a standard Keccak round implementation. Then, we reblind it (using the saved array[1], array[2] values), that is, array[0] := array[0] xor array[1] xor array[2])[10], and then perform the final 2 rounds using a 3-way threshold Keccak implementation again. We then output the length $3N$ blinded value from the three shares; that is, we output $N$ bytes from each of the three shares.

If the 3-way threshold implementation of the SHAKE round is five times as expensive as the standard round implementation, then the total cost of 4 threshold rounds and 18 standard rounds is about 1.7 times the cost of 24 standard rounds.

We illustrate this idea in Figure 2. We start out with the blinded state (represented by the top rectangles, with the black parts of the rectangle representing the attacker does not know, and the white parts representing the parts he does know). Then, we pass the blocks through two rounds of a thresholding implementation of Keccak. Then, we convert to a standard representation, and then perform a series of standard Keccak rounds. If the output is public (left side), the standard resulting state is the output (as represented by the bottom rectangle on the left - the initial part is the part of the state we will publish - we have to keep the rest of the state secret). If the output is secret (right side), then after round 22, we switch back to a thresholding format, and finish with two thresholding rounds. The result of that are the three separate shares (all black as all parts of it are secret).

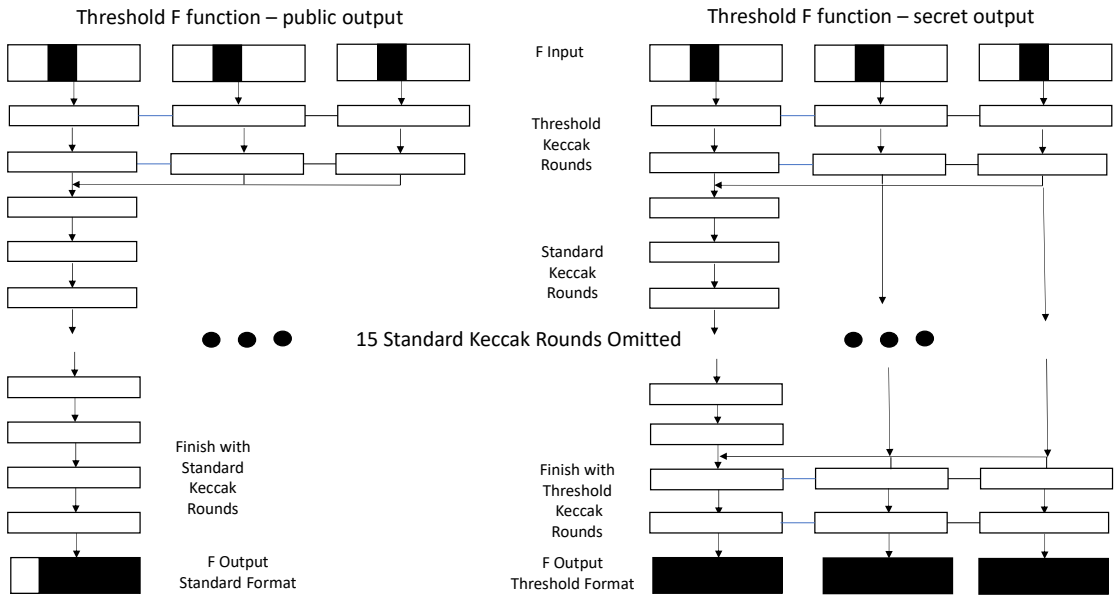


Figure 2: Overview of Proposed F implementation

The idea here is that the execution of an F function at any specific location within the Sphincs+ structure will be deterministic, that is, always run on the exact same data, and so the statistical methods used with DPA still leaves a significant amount of noise. The three-way thresholding is there to prevent us leaking information about the data in the initial and final rounds (for the middle rounds, the data is scattered over 1600 bits, and so moderate information leakage is not exploitable). Hence,

[10]Yes, the 'unblind' and 'reblind' operations are identical.

while this may resemble the standard threshold DPA protection, the reason we believe it gives us protection is different.

One objection of this idea might be that only one of the three shares of the array go through the full Keccak, while the other two participate in only 4 of the 24 rounds. We believe that is a mischaracterization of this idea. After two rounds, the two 'lesser used' shares contain essentially random data; after all, given only 2 of the 3 shares, it is impossible to recover the state, and those 2 shares may be any value independent of the logical state. Hence, when we pick them up 20 rounds latter, we are reblinding the state at that time with essentially random data (and 20 Keccak rounds effectively disguises any correlations that the third share may share with the other two).

# 5 Security

Given that this is a novel approach to side channel resistance, we would certainly need to give it some more analysis.

In the standard analysis of leakage during a DPA attack (as given in [CJRR99]), the instantaneous power power consumption when a particular value is being manipulated will be:

$$P = b \times s_0 + P \times s_0 + R \tag{1}$$

where $b$ is the contribution of power consumption by the bit the attacker is interested in, $P \times s_0$ is the contribution of events which involve $s_0$ and other state bits, and $R$ is the contribution of events which are independent of the bit $s_0$. It is noted that $R$ is typically much larger than $b$, however if we assume that it is independently distributed from $s_0$, we can statistically eliminate it by performing a number of measurements, and thus able to measure some correlation with $s_0$.

It would not appear that this analysis would apply to our situation. Whenever we perform an operation on bit $s_0$, we always have a number of other bits that are in a consistent setting. If these bits contribute to $R$, then $R$ would not at all be independently distributed from $s_0$. Hence, a different analysis is required.

One way to approach this is to denote $s$ as the vector of bits which are processed at one time. Then, we may write the instantaneous power consumption as:

$$P = b \times f(s) + P \times g(s) + R \tag{2}$$

where f(s) is the contribution of the entire bit string, $P \times g(s)$ is the contribution of events that involve $s$ and other state bits (which are also constant) and $R$ is the contribution of events that are independent of the bit string $s$. This may appear similar to Equation 1, however $f$ and $g$ are lossy functions. That is, the power consumption is based on the entire bit string, and two different bit strings may give the same power consumption profile. For this analysis, we will model $f$ and $g$ as based on the Hamming weight of $s$; that is, the power consumption of each individual bit is modeled to be the same, and that the attacker is able to recover only the sum of the power consumption from each bit. In other words, while the attacker may be able to deduce the hamming weight of a specific secret (after statistically removing $R$, which is the noise generated by other parts of the circuit, and can be assumed to be independent), however he is unable to recover information beyond that. This would appear to be independent of how many measurements that the attacker can make on the same computation.

Whenever we process a secret, we represent it in $3n$ bytes; the attacker has an advantage if he has enough information from the measurements to reduce this to $2^{8n}$ possibilities. By measuring the hamming weight, there is a possibility (if the hamming weight is either extremely low or extremely large) that the attacker can deduce a significant amount of information. In an extreme case, if the hamming weight is zero, then the attacker can deduce that all those bits are 0. However, if we assume that the secret values are uniformly distributed, we can explicitly compute the probability distribution of information available to the attacker. We have done so, and for the case of 64 bit words and N=128, the probability that the attacker learns enough to gain any advantage (which implies he reduces the number of possibilities to less than $2^{128}$ is approximately $2^{-238}$, which is less than the probability that the attacker could pick the secret value with a single guess.

We computed the probability distribution for the various possibilities of word size and security size, and summed the probabilities for all events that would give the attacker some advantage above the

designed security level. These probabilities are given in Table 5. Note that, in all cases, the probability of a leakage is less than required by the security level.

| Security Strength | Word Size | Leakage Probability |
|---|---|---|
| 128 | 32 | $2^{-223}$ |
| | 64 | $2^{-238}$ |
| 192 | 32 | $2^{-332}$ |
| | 64 | $2^{-354}$ |
| 256 | 32 | $2^{-441}$ |
| | 64 | $2^{-470}$ |

Table 1: The probability of vulnerability due to hamming weight

### 5.1 Information leakage in the processing of the F function

There is one issue that was not addressed in our analysis of the leakage due to the hamming weights: the protection of the F inputs and outputs. We protect those by having the logical value be the xor of three physical values. Now, if the attacker is able to precisely measure the hamming weight of the same segments of bits of the three physical values (and thus obtain their parity), he can then deduce the parity of the corresponding logical value. If we have a 64 bit implementation (that is, the attacker is able to measure the hamming weight of the internal 64 bit words), he is able to obtain enough information to reduce the number of possibilities for a 128 bit secret to $2^{126}$ (thus effectively reducing the security level by 2 bits; a 32 bit implementation would have a 4 bit security loss). Similarly, a 192 bit secret would lose either 3 or 6 bits of security, and a 256 bit secret would lose either 4 or 8 bits.

On the other hand, to exploit this leakage, the attacker has to precisely measure the hamming weight, that is, be able to determine that (for example) exactly 27 of the bits were set in a 64 bit word (and not 26 or 28). In this analysis, we assumed that the attacker could, in fact, measure things that precisely; it is unclear if he could in practice.

## 6 Performance

We have taken the Sphincs+ reference code[11], and modified it to incorporate this idea[Flu24]. We then compared the performance of the modified code with the original code. We did only the 'reference' implementation (and not the AVX2 version), and only the 'simple' parameter sets (not the robust).

Here are the measured performance numbers (on an Intel i7-8700 CPU) for the key generation and signature operations[12] for both the original reference code and the side channel protected code.

As you can see, the time taken by the protected code is consistently about a factor of 1.7 times the time taken by the unprotected code in this test environment.

## 7 Future Work

Future work would be needed to construct a more formal analysis of the security provided by this system. One aspect that needs more analysis is the potential information that may be leaked as a part of the SHAKE processing. Section 5 discusses the leakage due to the processing of the secrets themselves; before this can be used, we would need to analyze what potential leakages may occur in the first and last rounds of SHAKE (both when the internal secret is being generated, and while it is being hashed).

In addition, section 5.1 discusses one known leakage due to the potential leakage of the parity of various words; more analysis would be needed to show whether other subtle issues may cause further leakages.

[11] https://github.com/sphincs/sphincsplus, the consistent-basew branch as of November 30, 2023. The consistent-basew branch is the version that complies with the draft FIPS 205.

[12] We did not measure the verification operation as that code has not changed.

| Parameter Set | Operation | Reference Code | Protected Code | Performance Ratio |
|---|---|---|---|---|
| shake-128f | Gen | 7,606 | 13,308 | 1.75 |
| | Sign | 180,244 | 302,753 | 1.68 |
| shake-192f | Gen | 11,983 | 20,465 | 1.71 |
| | Sign | 284,826 | 480,743 | 1.69 |
| shake-256f | Gen | 31,959 | 57,997 | 1.81 |
| | Sign | 610,617 | 1,072,287 | 1.76 |
| shake-128s | Gen | 489,606 | 831,601 | 1.70 |
| | Sign | 3,697,068 | 6,140,307 | 1.66 |
| shake-192s | Gen | 706,266 | 1,214,987 | 1.72 |
| | Sign | 6,396,417 | 10,243,122 | 1.60 |
| shake-256s | Gen | 500,929 | 834,743 | 1.67 |
| | Sign | 5,576,062 | 9,635,712 | 1.73 |

Table 2: Cycle counts for both the original and unprotected code, in kilocycles

# 8 Conclusions

Here, we presented a possible alternative architecture for SLH-DSA that may be resistant to Side Channel attacks, while suffering from a modest slowdown of a factor of 1.7 and an incompatible private key.

We also presented a possible design paradigm for other side channel resistant primitives. In most cases, the primitive may need to be designed with this in mind, however it may give superior side channel protection.

Future work would include investigating the level of side channel resistance actually achieved with this approach.

# References


[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Flu24] Scott Fluhrer. sidechannel-resistant sphincs+ source code. github, March 2024. https://github.com/sphincs/sidechannel-resistent.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Nat15] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202, August 2015. https://doi.org/10.6028/NIST.FIPS.202.

[Nat23a] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. (U.S. Department of Commerce, Washington, DC), Draft Federal Information Processing Standards Publication (FIPS) 203, August 2023. https://doi.org/10.6028/NIST.FIPS.203.ipd.

[Nat23b] National Institute of Standards and Technology. Stateless hash-based digital signature standard. (U.S. Department of Commerce, Washington, DC), Draft Federal Information Processing Standards Publication (FIPS) 205, August 2023. https://doi.org/10.6028/NIST.FIPS.205.ipd.

# A Details about the PRF structure

To define how this is done more formally, we first define the function[13]:

$$
\begin{aligned}
Tree(root, i, m, adrs) &= H(i + \lfloor (m+1)/3 \rfloor, root, adrs) \quad \text{for } 0 \le i < m \\
H(0, root, adrs) &= root \\
H(x, root, adrs) &= SHAKE128(PK.seed || adrs_{pi=x} || H(\lfloor (x-1)/4 \rfloor, root, adrs), 3n) \quad \text{for } x > 0
\end{aligned}
$$

Where $adrs_{pi=x}$ means "set the pi (PRF index) field in the adrs structure to the value x".

With this definition, $H(x, root, adrs)$ is an "internal node" for $x < \lfloor ((m+1)/3 \rfloor$, and an "external node" for larger $x$. Every external node depends on an internal node, every internal node has at most four external nodes as children, and only the values of external nodes are used as generated derived values.

We note that this Tree function using its input in at most 4 different contexts, any intermediate values (internal nodes) appear in at most 5 different contexts, and the output appears (within the function evaluation as an external node) in 1 context. In addition, we note that even if some of the outputs are revealed[14], the remaining outputs cannot be recovered with less than $2^{8n}$ operations.

We use SHAKE128 rather than SHAKE256 (which is used everywhere else in SHAKE parameter sets of SLH-DSA). That is because, with the PK.seed and ADRS added, the length of the preimage is (for n=32) 160 bytes, which is longer than the rate of SHAKE256 (136 bytes). However, it is shorter than the rate of SHAKE128 (168 bytes), and hence it requires a single permutation operation to compute. We also note that, for a single block message (that is, it fits within the SHAKE rate), that SHAKE128 and SHAKE256 are essentially identical (with only a minor difference in the padding), hence using SHAKE128 in this case does not affect security. We decided that switching to SHAKE128 was a lesser change than removing the PK.seed (we need to include the Prf_index element of the ADRS input in the hash, and so we include the entire ADRS structure).

With that $Tree$ function defined, then for Merkle tree x (where 0 is the level of the bottom merkle tree and $d-1$ is the level of the top merkle tree, where $d$ is the number of Merkle tree levels), we define [15]:

$$
\begin{aligned}
Root_{d-1,t,adrs} &= SK.seed \\
Root_{x,t,adrs} &= Tree(Root_{x+1,\lfloor t/h' \rfloor, adrs}, (t \bmod h') + 2^{h'} \cdot wdigit, 2^{h'} \cdot (wdigit + 1)) \quad \text{for } x < d-1
\end{aligned}
$$

## A.1 The ADRS structure

In the $H$ function, each internal node has up to four children, and we need something to differentiate which child value we are deriving. In our construction, we have each child assign a different value into the PRF index field of the ADRS structure, and include that structure in the hash.

That is the only thing (apart from the parent node value) that must be in the hash. Sphincs+ uses the ADRS structure to prevent multitarget preimage attacks; with this $H$ construction (where each secret value is $3n$ bits long, that is not a concern. However, to maintain consistency with the rest of the Sphincs+ structure, we do include the ADRS structure (and the pk.seed value) in the hash.

We propose to use these for the ADRS structure used within the H function.

The Layer Address, Tree Address and Key Pair values used are the same as what is used when processing the corresponding Sphincs+ structure.

# B Alternative approaches to protect SK.prf

Here are some other approaches we could take to protect the SK.prf secret and the $PRF_{msg}$ function, and why we not recommend them.

[13] Sphincs+ has the PK.seed and the ADRS as inputs to SHAKE256 to address potential multitarget attacks. With the inputs we have here being long ($3n$ bytes), multitarget attacks are not a concern. However, as this longer input still fits within the SHAKE128 rate, there is little reason not to make this mostly consistent with the rest of Sphincs+.

[14] Some of the PRF values will appear within a valid signature.

[15] And recall that $h'$ is the height of a Merkle tree and $wdigit$ is the number of Merkle chains.

| | |
|---|---|
| Layer Address | 4 Bytes |
| Tree Address | 12 Bytes |
| type = 7 (PRF_MERKLE) | 4 Bytes |
| padding = 0 | 4 Bytes |
| Prf Index (pi) | 4 Bytes |
| padding = 0 | 4 Bytes |

Figure 3: PRF Merkle Hash Address

| | |
|---|---|
| Layer Address = 0 | 4 Bytes |
| Tree Address | 12 Bytes |
| type = 8 (PR_FORS) | 4 Bytes |
| Key Pair | 4 Bytes |
| Prf index (pi) | 4 Bytes |
| padding = 0 | 4 Bytes |

Figure 4: PRF FORS Hash Address

- One could consider an approach that gives DPA security even if opt_rand is omitted or predictable. However, that turns out to be expensive; to do that (and to follow the same base design that Sphincs+ uses), we would need to have a fully protected $PRF_{msg}$ function, including the part that hashes the message; this is because if the attacker can recover any position in the state (including while we're hashing the message), he can apply inverse Keccak permutations to recover the original state. That would be expensive (especially if we need to sign a long message); the alternative would be to redesign how $PRF_{msg}$ is designed (for example, using a threshold implementation of a Carter-Wegman style MAC), and that would be a larger change than what we wanted to consider.
- One could directly generate $R$[16] from randomness and not use $PRF_{msg}$ at all. While this is possible (and certainly cheaper), using $PRF_{msg}$ does provide some level of protection (even if not full security) if the entropy is faulty, hence we currently do not recommend it.

# C Potential optimizations that do not affect security

- For $PRF_{msg}$, one potential optimization would be to use SHAKE-128 or TurboSHAKE rather than SHAKE-256 for $PRF_{msg}$; the reduced security level of either does not practically impact the security of Sphincs+ (because the security comes from the unpredictability of opt_rand), and the better performance would help in signing long messages. Note that, even if we use this optimization, the $H_{msg}$ function would need to still use SHAKE-256, both for interoperability reasons (an unmodified verifier will need to compute that same hash), and because we do need full second preimage resistance.
- For the WOTS chain that corresponds to the most significant digit of the checksum field, that value has a reduced range. In particular, for Level 1, that digit will always be in the range 0-1; for Level 3, that digit will always be in the range 0-2; for Level 5 that digit will always be in the range 0-3. Hence, all positions in that WOTS chain past that will always be public, and so we could use our unprotected SHAKE implementation for those. That would give a small (circa 1%) performance increase for a little bit of additional complexity.

The implementation we used for our performance measurements does not use either of these optimizations.

In addition, when this system is used to sign a number of messages, the secrets corresponding to the top parts of the hypertree are used with high probability, while the secrets corresponding to the lower parts of the hypertree and the FORS trees are used relatively rarely. Hence, if an implementation were to precompute the top parts of the hypertree and cache them in the private key, the number of traces available to an attacker against any one specific secret may be drastically reduced. For example, if the implementation of the 128S parameter set were to cache all 512 leaf public keys in the top Merkle tree (and use those when computing the top Merkle tree authentication path), then any secret would be used with probability $2^{-9}$ when generating a signature. We have argued that the number of traces available to the attacker does not appear to be significant in the difficulty of the attack; if this contention is incorrect, this may also harden the system somewhat; in the example of 128S, this would mean that the number of signatures the attacker would need to collect would increase by a factor of 512.

[16] This is the output that $PRF_{msg}$ generates and is placed directly into the signature.

# D One alternative to the F threshold idea

If we are on an implementation that supports SIMD (such as AVX), the structure of Sphincs+ allows us to evaluate several F functions in parallel (and always the same grouping of F computations); the AVX version of the Sphincs+ reference code does precisely this. If the implementation does that, that in itself may give some protection for the F functions. After all, if the same four F inputs are always evaluated at the same time, the attacker would get the noise from all four; it would appear intuitive that the attacker would not be able to isolate them, and thus not get enough information to attack any one of them. However, this is speculation (as the attacker would be able to get some; for example, that the hamming weight of a sequence of bits of one of the secrets is likely to be small/large); in addition, we expect that the implementations that are most interested in side channel protection (HSMs) would not have SIMD support available to them.

This protection would not extend to the PRF portion; for that, the tree-based approach would still be needed. On the other hand, most of the slowdown seen in the current test implementation is due to the additional complexity of the F computation.

# E A more performant, less secure alternative

The above ideas assume the requirement that we need to maintain full security, even under DPA attacks. As we noted above, if we keep things deterministic, DPA attacks recover only limited amount of data. If we accepted such a security loss, it appears that we can have a Sphincs+ implementation with some DPA protection, and actually performs faster than the SLH-DSA standard. Here is how we would address the three types of secrets:

- $PRF_{msg}$ - we would the same strategy of requiring unpredictable opt_rand values
- F - use an unprotected version of SHAKE (which has some security loss under DPA attack)
- PRF - use a similar tree based approach as above, but change how we use SHAKE. We instead use mapping between parent and children as: (child0 — child1 — child2 — child3) = SHAKE( PK.seed — ADRS — parent, 4n ).

With this design, computing the PRF iteratively (that is, where we evaluate all the external nodes of the PRF tree, which is what we do most of the time) requires one Keccak permutation every three outputs (as opposed to one permutation per output in the standard SLH-DSA design). Since everything else takes the same amount of time, this design would perform somewhat faster. And, because everything is deterministic, and all secrets are used only a small number of times, we get some DPA resistance (albeit not as much as in the main idea).