PQSHIELD

Think openly, build securely

White paper:

Secure Update Propagation via Set-Homomorphic Signatures

Software | PQShield | November 10, 2020

Maintaining a consistent database across all nodes of distributed network is a vital but challenging task, especially in an asynchronous setting. In this document, we propose to use homomorphic signatures to perform this task securely. Unlike previous solutions, ours achieves high scalability, flexibility and statelessness. We also highlight several potential applications.

1 Introduction

The real-life motivation for this work is update propagation in distributed networks. An important constraint imposed on these networks is *asynchrony*: from their very nature, distributed networks cannot instantaneously propagate an update. They should nevertheless maintain functionality and security in the face of asynchrony.

We study the progagation of updates in distributed networks from a security viewpoint. For simplicity, we consider that one single node may send updates to the network; this node is called the distributor (denoted $\mathcal{D}$) and the other nodes are called subscribers. Our arguments seamlessly transfer to the case of several distributors.

Distributor and subscribers share a common, regularly updated database. The distributor plays a special role as he edicts the update policy: this includes deciding what the updates are and when to apply them, but can also entail finer-grained policies, such as: (a) specifying distinct updates for different groups of users, (b) specifying dependencies (or lack thereof) between distinct updates.

What is the best way to propagate an update? A naive solution with limited scalability is for the distributor to contact directly all the subscribers in a centralized way, as illustrated by the star network of Figure 1. A more scalable solution is for the distributor to send the updates to a subset of subscribers, which will themselves transfer them to other subscribers, and so on until the updates are distributed to every user in the network. The server then assumes the role of the root of a tree, with the subscribers being the other nodes. This is illustrated in Figure 2.

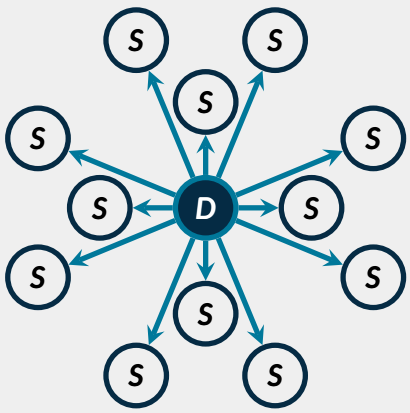

Figure 1: Centralized update propagation

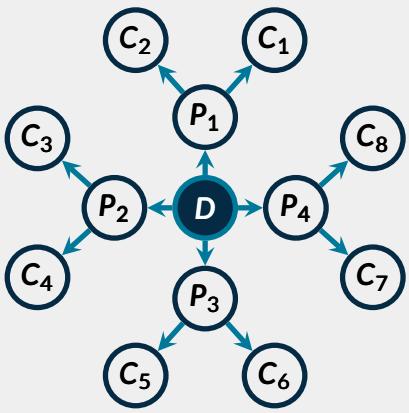

Figure 2: Distributed update propagation

1.1 Secure Update Propagation in Distributed Networks

Distributed updates better befit the nature of distributed systems and are scalable. However, they raise the issue of the *authenticity* of updates: how can a subscriber (say C_2 in Figure 2) be sure that an update was created by the distributor $\mathcal{D}$ and not by another subscriber (say P_1 in Figure 2)? This is informally the problem of *secure update propagation* (or SUP).

An obvious solution to this question is to rely on digital signatures. Just like their real world counterparts, digital signatures can be verified by anyone knowing (the public key pk of) the signer, but can be computed only by the signer (using a private key sk known only by her). In this document, we present *four* solutions to SUP, all based on signature schemes. We evaluate them with respect to three metrics: *scalability*, *flexibility* and *statelessness*.

Scalability

The first criterion for evaluating SUP solutions is their scalability. In an unprotected distributed update protocol, subscribers fetch updates from the distributor or other subscribers. Each of the secure solutions we review imposes an overhead: to authenticate the updates they receive, subscribers must receive and verify signatures (which are appended to the updates). Each solution varies by the number of signatures subscribers may receive. Depending on the solution, this number may depend on:

- The total number of updates (denoted k).
- The number of updates missed by a user (denoted m).

Flexibility

The second criterion is flexibility. As stated before, distributed networks are inherently asynchronous; this provokes several situations which have to be handled:

- *Out-of-order updates:* Updates may not be received in the intended order. For example, a subscriber may receive up_{i+1} before the update up_i.
- *Missing updates:* Updates may be lost in transit and never be received. This can be seen as an extreme case of an out-of-order update, where an update is received infinitely late.

Other situations stem from specific needs of the network's users. For example:

- *Fine-grained update policy:* The distributor may decide that a subset of users must receive some updates, and that another subset must receive other updates. The subsets of users may even intersect. This scenario is particularly relevant for the example of software updates in Linux and applications updates in iOS/Android: depending on the applications installed, users may require different updates.

We consider that a solution is flexible if is amenable to the various situations we described. Ideally, it should remain fully functional and retain the same level of scalability. On the other side of the spectrum, a solution may completely break down in one of these situations.

Statelessness

A system is said to be stateful if it requires (for functionality or security) its parties to remember previous events or interactions. In the contrary case, it is said to be stateless. In general, it is desirable for a system to be stateless rather than stateful, as the latter may complicate deployment and implementation. For all the solutions we consider, we state whether they are stateless and, if not, which parties are required to be stateful (distributor, subscribers, or both).

1.2 The Problem of Scalable, Flexible and Stateless SUP

Scalability, flexibility and statelessness are all desirable properties. A scalable system costs less resources and enables growth instead of impeding it. Likewise, flexibility increases the applicability of a solution by making it amenable to more use cases. Finally, stateful schenes are often undesirable.[1] We therefore consider the following problem:

Can we build a scalable, flexible and stateless SUP protocol?

As it turns out, it is surprisingly challenging to achieve the three properties at the same time, and doing so has remained an open problem until now. The naive solution (signing each update) is flexible and stateless but not scalable, as we will see in Section 3. [LKMW19] presented a few scalable solutions, but each of them suffered from statefulness and a lack of flexibility. In this document, we introduce a new solution which is novel is two ways:

1. It is the first solution which achieves simultaneously scalability, flexibility and statelessness. Previous solutions were either flexible and stateless, or scalable, but never realized the three properties at once.
2. It is the first solution which relies on (set-)homomorphic signatures. Our solution, like the previous ones, relies on hash functions and digital signatures. Some of the previous solutions used hash functions with special properties, but all of them used classical signatures. On the other hand, our solution relies on set-homomorphic signatures, this property is precisely what allows our solution to simultaneously be scalable, flexible and stateless.

The next table summarizes the differences between our solution (in boldface) and the previous ones, both in terms of the techniques used and the results achieved.

Solution	Hash function	Signature	Scalable?	Flexible?	Stateless?	Section
Sign update	Classical	Classical	✖	✔	✔	Sec. 3.1
Sign database	Classical	Classical	✔	✖	✖	Sec. 3.2
Merkle tree	Merkle tree	Classical	✔	✖	✖	Sec. 3.3
Homomorphic hashing	Homomorphic	Classical	✔	✖	✖	Sec. 3.4
Homomorphic signatures	**Homomorphic**	**Homomorphic**	✔	✔	✔	Sec. 4

[1] Adam Langley, cryptographer at Google: "*for most environments it's a huge foot-cannon*", (https://www.imperialviolet.org/2013/07/18/hashsig.html).

1.3 Use Cases

We present a few use cases for which our novel solution may be of interest.

Database updates in Facebook

In a white paper published by Facebook [LKMW19] and companion articles[2] , it is explained that Facebook's infrastructure relies on distributed update propagation. This raises scalability issues which Facebook resolves with a solution based on homomorphic hashing (detailed in Section 3.4), currently deployed on their servers. Our solution offers scalability, but also provides flexibility.

Applications updates in iOS/Android

The standard way to install applications in the iOS and Android operating systems is by using a centralized service: the App Store for iOS, and Google Play for Android. Here the distributor is the App Store/Google Play, the subscribers are mobiles devices equipped with iOS/Android and updates are regularly published for each application.

This example highlights why a fine-grained update policy can be desirable. Indeed, each subscriber only need updates of applications she possesses, which means that each subscriber may receive a distinct subset of updates. Likewise, applications may be interdependent (for example, a bank may propose two distinct yet interconnected applications to its clients) or completely independent (for example, Wikipedia and a newspaper application), which may call for a fine-grained update policy.

Software updates in Linux

Linux distributions allow to install and update software. Similarly to iOS and Android, this can be done in a centralized manner via a software manager. Here the distributor is the repository from which software updates are fetched. Is is common for a device which has been offline for a few days to require dozens of software updates: compared to non-scalable solutions, our solution may reduce the bandwidth and computation overhead by an order of magnitude or more. In addition, different users may install distinct software, which requires a level of flexibility which does not seem to be met by any scalable solution described in this document, except for ours. Finally, each software may have its own update policy. In Linux, this is made explicit through the notion of *dependencies*: a software may specify a list of softwares that need to be installed prior to it, and the same goes for updates. Our solution is compatible with such update policies.

1.4 Roadmap

The rest of this document is organized as follows. Section 2 presents the cryptographic tools used and formalizes SUP. Section 3 presents the previously existing solutions: one of them is stateless and flexible but not scalable, the other ones possess various levels of scalability but are neither flexible nor stateless. Section 4 makes a detailed presentation of our novel design, which achieves simultaneously scalability, flexibility and statelessness. Finally, Section 5 provides a concrete instantiation of our design.

[2] https://engineering.fb.com/security/homomorphic-hashing/
https://engineering.fb.com/data-infrastructure/location-aware-distribution-configuring-servers-at-scale/

2 Preliminaries

2.1 Hash Functions

In cryptography, a hash function that maps bit strings of arbitrary size to a bit string (called a hash or hash digest) of fixed size; it requires the important property to be a one-way function, which means that it should be infeasible to invert in practice. Cryptographic hash functions may be seen as the electronic equivalent of fingerprints, as they provide a concise way to guarantee the integrity of a document: even the slightest change to a bit string will completely change its hash.

Set-Homomorphism

Two of the solutions we present rely on classical hash functions with no particular property. The solution of Section 3.3 relies on Merkle trees, which provide a convenient manner of hashing a set; upon addition/removal/modification of an element, the hash digest can be updated in time logarithmic in the size of the set. The solution of Sections 3.4 and 4 uses set-homomorphic hash functions, which for the same operations allow to update the hash digest in *constant* time. A hash function H is said to be set-homomorphic if for any disjoint sets S, T:

$$H(S \cup T) = H(S) + H(T).$$

Additional information about Merkle trees and set-homomorphic hash functions are given in Section 3.3 and 3.4, respectively.

2.2 Signature Schemes

A signature scheme provides the digital equivalent of physical signatures. First, a party (called the signer) conjointly generates a public key pk and a private key sk, distributing the former and keeping the latter for himself. The private key allows the signer to compute a signature $\mathsf{sig} = \mathsf{Sign}(\mathsf{sk}, \mathsf{msg})$ for any message msg during what is called the signing procedure. The public key pk allows any recipient of a message msg sent by the signer to verify its authenticity by checking whether $\mathsf{Verify}(\mathsf{pk}, \mathsf{msg}, \mathsf{sig})$ accepts sig as a valid signature of msg; however, pk does not help to compute a signature. This replicates a feature of real-life signatures, which can be easily verified by anyone but (ideally) only done by the legitimate signer.

Set-Homomorphism

In this document, all the solutions except one rely on classical signatures and do not require any particular property. The last solution requires set-homomorphic signatures, which is the signature analogue of set-homomorphic hash functions. A signature scheme is said to be set-homomorphic if for any disjoint sets S, T and valid signatures $\mathsf{sig}_S \leftarrow \mathsf{Sign}(\mathsf{sk}, S)$ and $\mathsf{sig}_T \leftarrow \mathsf{Sign}(\mathsf{sk}, T)$, their sum $\mathsf{sig}_S + \mathsf{sig}_T$ is with high probability a valid signature of $S \cup T$, that is:

$$\mathsf{sig}_S + \mathsf{sig}_T \in \mathsf{Sign}(\mathsf{sk}, S \cup T).$$

3 Overview of Previous Solutions

All solutions below rely on signatures and hash functions. Solutions in Sections 3.1, 3.2 and 3.3 assume the existence of a generic hash function (say, SHA-3) but differ in how they are applied to updates/databases. The solution in Section 3.4 relies on hash functions with specific properties.

3.1 Signing Each Update

The naive solution for SUP is for the distributor $\mathcal{D}$ to sign each update with a given signature scheme: $\mathsf{sig}_i \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{up}_i)$. Each time $\mathcal{D}$ publishes an update, it sends $(\mathsf{up}_i, \mathsf{sig}_i)$ to its children, who propagate it as is. Each update is completely independent. This is the simplest solution, and is also quite flexible: missing and out-of-order updates are easily handled. However, the number of signatures and verifications is equal to the number of missed updates, so this solution has poor scalability.

✖ **Scalable?**
As each update is independent, this scales linearly in the number of (missed) updates: if a subscriber has missed m updates, he downloads m signatures and performs m verifications.

✔ **Flexible?**
Since each update is independent, this solution is very flexible: missing or out-of-order updates do no affect its functionality, and it can accommodate fine-grained update policies.

✔ **Stateless?**
As each update is self-contained, there is no need to maintain a state.

3.2 Signing the Whole Database

A downside of the method from Section 3.1 is that it does not scale well with the number of missed updates. To mitigate this efficiency loss, a solution suggested in [LKMW19] for the distributor $\mathcal{D}$ to sign the whole database: $\mathsf{sig}_i \leftarrow \mathsf{Sign}(\mathsf{sk}, D_i)$, where D_i is the state of the database after updates $\mathsf{up}_1, \ldots, \mathsf{up}_i$ have been applied. It is explained in [LKMW19] that the computational cost of this solution is linear in the *total* number of updates. A quick workaround is to instead set $\mathsf{sig}_i \leftarrow \mathsf{Sign}(\mathsf{sk}, \boldsymbol{h}_i)$, where $\boldsymbol{h}_1 = H(\mathsf{up}_1)$ and for any $i > 1$, $\boldsymbol{h}_i$ is recursively defined as $\boldsymbol{h}_i = H(\mathsf{up}_i \| \boldsymbol{h}_{i-1})$.

✔ **Scalable?**
This solution offers excellent scalability in the number of missed updates: a subscriber who missed m updates only needs to download and verify 1 signature (along with the m updates).

✖ **Flexible?**
As each update depends on all the previous ones, this solution offers no flexibility.

✖ **Stateless?**
The distributor needs to keep track of all the updates he has published. This can be done efficiently (the state can consist of a single hash digest), but still the system requires the distributor to be stateful. Likewise, each subscriber needs to be stateful as well.

3.3 Merkle Trees

It has been suggested in [LKMW19] to tweak the solution of Section 3.2 by relying on trees. We show here that by carefully employing it, it enjoys a good level of scalability.

Merkle trees and their properties.

A Merkle tree (or hash tree) is a binary tree in which the value of each internal node is the hash of the values of its children. An illustration can be found in Figure 3.

As often when tree structures are used, Merkle trees provide dramatic gains over naive hashing for ensuring the integrity of a database. To hash a whole database, one makes each element of the database a leaf of the Merkle tree (● nodes in Figure 3) and takes as the hash of the database the root of the Merkle tree (■ in Figure 3). Simple alterations of the database (adding, deleting or modifying) can be taken into account in the hash digest in time $O(\log t)$, where t denotes the total number of elements in the database. In comparison, a naive hash would need time $O(t)$.

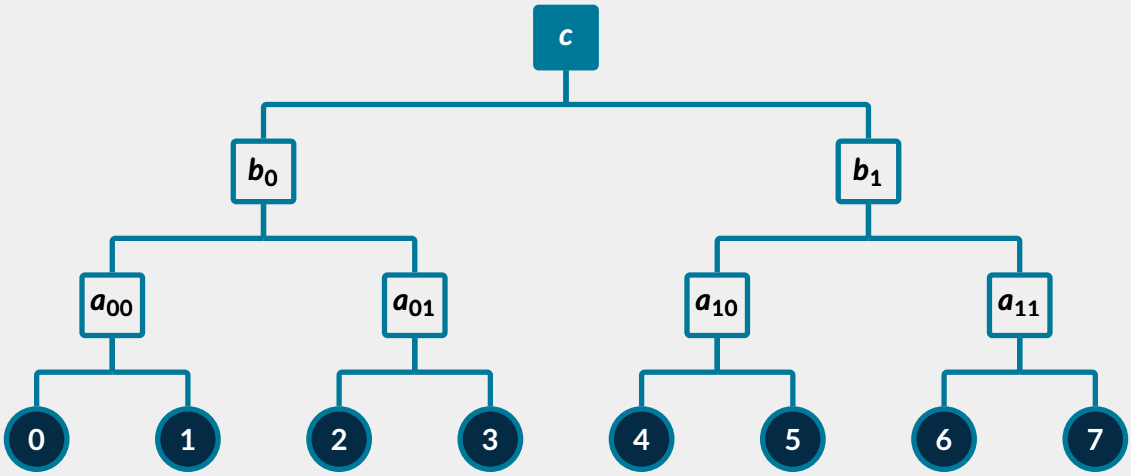

Figure 3: A Merkle tree. Edges connecting nodes mean "the parent is the hash of the two children". The value of a_{00} is $H(0\|1)$, the value of b_0 is $H(a_{00}\|a_{01}) = H(H(0\|1)\|H(2\|3))$, and so on.

It has been noted in [LKMW19] that in order to reach that level of efficiency, the whole Merkle tree needs to be stored, which means storing $O(t)$ values. Consequently, [LKMW19] has dismissed Merkle trees as being ultimately impractical for secure update propagation. However, we note that this $O(t)$ storage requirement is true only if we need to perform additions AND deletions AND modifications. If we *only require additions*, then one only needs to store what is called the copath (or authentication path) of the last leaf added to the Merkle tree.[3] This allows to can relax the storage requirement to $O(\log t)$.

[3] We say that the path of a node is the set of its ancestors (including itself), and its co-path is the set of siblings of each element in the path. For example, the path of the leaf 1 is $\{0, a_{00}, b_0\}$, and its copath is $\{1, a_{01}, b_1\}$.

A solution based on Merkle trees

With that in mind, we can now provide an efficient Merkle tree-based solution to secure update propagation. Unlike [LKMW19], we see updates as black-box objects and do not consider how they impact a database. This makes our solution more generic, but also more scalable than what we presume [LKMW19] had in mind. We order the updates $(\mathrm{up}_i)_i$ in a chronological order. A dynamic Merkle tree is constructed from the updates. By convention, if an existing node has no sibling then its parent will have the same value; this allows to handle the overwhelming number of cases when the number of updates is not a power-of-two.

Each update up_j is published along with $\mathrm{Sign}(\mathsf{sk}, H_j)$, where H_j is the Merkle hash of all the updates up to up_j; by the process we just described, a subscriber can itself recompute H_j upon reception of up_j, and only requires time and storage $O(\log j)$ in order to do so. Hence this solution is very competitive in terms of efficiency. However, we note that this efficiency requires a strict ordering of the updates and remembering the last homomorphic hash, so this solution is neither stateless nor flexible.

✔ **Scalable?**

This solution offers good scalability. When a subscriber fetches m missing updates out of a total of t updates, he only needs to download and verify one single signature. Verification requires to compute $O(m + \log t)$ hashes.

✖ **Flexible?**

For the same reason as the previous solution, this solution is not flexible.

✖ **Stateless?**

Each subscriber needs to keep a state of nodes which help him recompute the root of the Merkle tree. This state consists of $O(\log t)$ nodes.

3.4 Homomorphic Hashing

A final solution has recently been proposed by Lewi, Kim, Maykov and Weis [LKMW19]. Unlike the previous ones, this solution does not rely on generic hash functions but instead on hash functions with the peculiar property of being set-homomorphic. Indeed, they require the existence of a hash function H on sets such that for any disjoint sets S, T:

$$H(S \cup T) = H(S) + H(T).$$

This property is known as set-homomorphism. The notion of set-homomorphic hash functions, as well as concrete instantiations, have first been proposed by Bellare and Micciancio [BM97]. In particular, [BM97] proposed LtHash, a set-homomorphic hash function based on the SIS assumption, which is a standard hardness assumption in lattice-based cryptography. LtHash has been used by [LKMW19] as the underlying set homomorphic hash function.

In the solution of [LKMW19], each update up_j is published along with $\mathsf{sig}_i = \mathsf{Sign}(\mathsf{sk}, H_j)$, where H_j is the homomorphic hash of the set of all updates up to up_j.[4] The efficiency gain of this solution lies in the fact that if a subscriber has missed m updates $\mathsf{up}_{i+1}, \ldots, \mathsf{up}_{i+m}$, he only needs to be sent $\mathsf{sig}_{i+m} = \mathsf{Sign}(\mathsf{sk}, H_{i+m})$, as he can use H_i (which he stored) as well as $\mathsf{up}_{i+1}, \ldots, \mathsf{up}_{i+m}$ (which he needs anyway) to recompute himself H_{i+m} by leveraging the set-homomorphic properties of H and subsequently check the validity of sig_{i+m}. Thus this solution achieves excellent scalability. However, we note that it is not flexible as the updates should follow a strict ordering.

- ✔ **Scalable?**
  This solution offers excellent scalability: a subscriber fetching m missing updates needs to download and verify one signature.
- ✖ **Flexible?**
  For the same reason as the previous solutions, this solution is not flexible.
- ✖ **Stateless?**
  Each subscriber needs to keep a state consisting of one single hash (the one corresponding to its current update state).

[4] The original construction of [LKMW19] is more complex as it takes into account the nature of the updates (adding, removing or modifying an element in the database). We present here a simplified variant of their scheme which is agnostic to the nature of updates, making it more generic yet as efficient.

4 A Novel Design Approach: SUP via Set-Homomorphic Signatures

Looking back at the solutions for SUP described in Section 3, they all rely on the same high-level idea (hash-then-sign the update or database), and only differ in their execution of how the hash operation is realized. The first solution hashes each update naively, the second one hashes the database naively, the third one hashes the database with a Merkle tree and the last one hashes each update using homomorphic hashing. A common point is that they all rely on a classical signature scheme with no specific properties.

Our solution stands apart as it requires a set-homomorphic signature scheme, which is the signature counterpart to a set-homomorphic hash function. This novel approach effectively solves the problem described in the introduction: we obtain scalable, flexible and stateless SUP. This section first presents set-homomorphic signatures, and then shows how to use them for SUP.

4.1 Set-Homomorphic Signatures

A set-homomorphic signature scheme is a signature scheme such as if $\mathsf{sig}_S \leftarrow \mathsf{Sign}(\mathsf{sk}, S)$ and $\mathsf{sig}_T \leftarrow \mathsf{Sign}(\mathsf{sk}, S)$ are valid signatures of disjoint sets S and T, then with high probability:

$$\mathsf{sig}_{S\cup T} = \mathsf{sig}_S + \mathsf{sig}_T \text{ is a valid signature of } S \cup T.$$

This notion is related but distinct from incremental signatures [BGG94, BGG95], which allow to quickly compute the signature of an updated document. Incremental signatures schemes can be constructed in a generic way from a signature scheme [BGG95], and in existing constructions the updated signature is computed by the signer itself. In contrast, is is not currently known how to construct set-homomorphic signatures in a generic and efficient way, but on the other hand computing an updated set-homomorphic signature can be done by anyone (not only the signer) as it does not require any secret information.

Examples of set-homomorphic signatures

In 2008, Gentry, Peikert and Vaikuntanathan [GPV08] showed how to obtain secure hash-then-sign schemes using lattices problems. In their framework (which we will call the GPV framework), the public key is a random-looking matrix $\boldsymbol{A}$ with coefficients in $\mathbb{Z}_q$, and the signature of a message msg is a short vector sig (in the sense that its norm is small) such that $\mathsf{sig} \cdot \boldsymbol{A} = H(\mathsf{msg})$.[5] Several variations and refinements to this framework have been proposed [MP12, SS13, GM18], and a few practical instantiations have been proposed [DLP14, MSO17, BFRLS18, PFH+19]. If the underlying hash function H is homomorphic, then GPV-based signature schemes are set-homomorphic.

[5] Many concrete instantiations of the GPV framework randomize the hash with a salt which is sent along with the signature, but our argument is indifferent to the presence of a salt.

Indeed, if $\mathsf{sig}_1, \ldots, \mathsf{sig}_m$ are valid signatures for $\mathsf{msg}_1, \ldots, \mathsf{msg}_k$, then:

$$\begin{aligned}(\mathsf{sig}_1 + \cdots + \mathsf{sig}_m) \cdot \boldsymbol{A} &= H(\mathsf{msg}_1) + \cdots + H(\mathsf{msg}_m) \\ &= H(\{\mathsf{msg}_1, \ldots, \mathsf{msg}_m\})\end{aligned}$$

So from m valid signatures for distinct messages $\mathsf{msg}_1, \ldots, \mathsf{msg}_m$, we can construct one single signature that is valid for the set $\{\mathsf{msg}_1, \ldots, \mathsf{msg}_m\}$. An important caveat is that for the signature $(\mathsf{sig}_1 + \cdots + \mathsf{sig}_m)$ to be valid, it should remain small. On the other hand summing m signatures will increase the norm of the result by $\sqrt{m}$ on average, so it is clear that one can only allow a finite number of summations while maintaining security. Therefore any application (including ours) exploiting the set homomorphism of GPV-style signatures needs to strike a balance between security and the number of summations allowed.

4.2 Our Novel Solution

Our solution requires a set-homomorphic signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. As for the scheme of Section 3, the distributor $\mathcal{D}$ initially generates a keypair $(\mathsf{pk}, \mathsf{sk})$ and distributes the public key pk to the subscribers. As in the solution of Section 3.1, each update up is published along with its signature $\mathsf{sig}_{\{\mathsf{up}\}} = \mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}\})$. When a subscriber receives a single update up, he verifies its validity by checking that $\mathsf{Verify}(\mathsf{pk}, \mathsf{sig}_{\{\mathsf{up}\}}, \{\mathsf{up}\}) = \textbf{\textit{True}}$. Thus for individual updates this solution works just like the one of Section 3.1.

The set-homormorphism property kicks in when considering batch updates. If a child subscriber has missed m updates $\mathsf{up}_{i+1}, \ldots, \mathsf{up}_{i+m}$ and fetches them from a parent subscriber, the signatures $\mathsf{sig}_{\mathsf{up}_{i+1}}$ can be aggregated into one single signature. By the set-homomorphic properties of the signature scheme:

$$\mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+1}\}) + \cdots + \mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+k}\}) = \mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+1}, \ldots, \mathsf{up}_{i+m}\})$$

Thus the parent subscriber which possesses $\mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+1}\}), \ldots, \mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+m}\})$ can sum them into a single signature $\mathsf{Sign}(\mathsf{sk}, \{\mathsf{up}_{i+1}, \ldots, \mathsf{up}_{i+m}\})$ and send it to the child subscriber, saving a factor m in bandwidth and verification time.

- ✔ **Scalable?**
  This solution offers excellent scalability: a subscriber fetching m missing updates needs to download and verify one signature.
- ✔ **Flexible?**
  As each update is independent, this solution achieves very good flexibility: the protocol remains fully functional and secure in the presence of out-of-order or missing updates. It can also accommodate fine-grained policies such as forwarding distinct batch of updates to different sets of subscribers.
- ✔ **Stateless?**
  Neither the distributor nor subscribers need to maintain a state.

5 Practical Instantiation with Falcon

We focus our attention on a family of signatures which are set-homomorphic (when the underlying hash function is also set-homomorphic): signatures based on the GPV framework [GPV08]. Instantiations of the GPV framework [GPV08] can essentially be separated in two lines of research. The first one follows the Micciancio-Peikert [MP12] variant of GPV, and was instantiated in [BFRLS18, GM18, GPR+18, CGM19]. The second one instantiates GPV over NTRU lattices [SS13, DLP14, MSO17, PFH+19]. The signature scheme Falcon [PFH+19] is based on the second line of research and, as a finalist to NIST's standardization process [NIS19], has arguably achieved a certain level of maturity and received extensive scrutiny. In addition, it is one of the most compact post-quantum signatures schemes and we therefore choose it as a starting point to instantiate our SUP solution.

Description of Falcon

We present Falcon at a high level, a more complete description can be found in [PFH+19]. We work over the rings $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. Thus elements of $\mathcal{R}$ polynomials with integer coefficients of degree at most $n - 1$, and elements of $\mathcal{R}$ have the additional constraint that their coefficients are in $\{0, 1, \ldots, q - 1\}$. Key generation of Falcon is described in Algorithm 1.

Algorithm 1 KeyGen(n, q)

Require: Ring dimension n, integer modulus q

Ensure: A public key $\mathsf{pk} = h \in \mathcal{R}_q$, a private key $\mathsf{sk} = \boldsymbol{B} \in \mathcal{R}^{2\times 2}$

1: Generate short polynomials f, g, F, G in $\mathcal{R}$ such that $fG - gF = q$

2: $h = g \cdot f^{-1} \bmod q \in \mathcal{R}_q$

3: $\boldsymbol{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$

4: **return** $(\mathsf{pk} = h, \mathsf{sk} = \boldsymbol{B})$

Algorithm 2 Sign(msg, sk)

Require: A message msg and a private key sk

Ensure: A signature sig of msg

1: Use sk to compute $(s_1, s_2) \in \mathcal{R}$ such that:

- $s_1 + s_2 h = H(\mathsf{msg})$
- $\|(s_1, s_2)\|_2 \leq \beta$
- $\|(s_1, s_2)\|_\infty \leq \beta_\infty$

2: **return** $\mathsf{sig} = s_2$

Algorithm 3 Verify(msg, sig, pk)

Require: A message msg, a public key pk and a signature sig

Ensure: ***True*** or ***False***

1: $s_2 = H(\mathsf{msg}) - s_1 h$

2: **if** $\|(s_1, s_2)\|_2 \leq \beta$ **and** $\|(s_1, s_2)\|_\infty \leq \beta_\infty$ **then**

3: **return** ***True***

4: **else**

5: **return** ***False***

Table 1: Falcon parameters for SUP

Description	Notation	Value
Ring dimension	n	1024
Integer modulus	q	6873089
Standard deviation	σ	$1.54\sqrt{q}$
Number of supported aggregations	k	1000
Constraint on $\|\|(s_1, s_2)\|\|_2$	β	$\left\lceil \tau_{\text{sig}} \cdot \sigma\sqrt{2nk} \right\rceil$
Tailcut rate for the signature	τ_{sig}	1.2
Constraint on $\|\|(s_1, s_2)\|\|_\infty$	β_∞	$\left\lceil \sigma\sqrt{2k \cdot \log\left(\frac{4n}{p}\right)} \right\rceil$
Signature size in bytes	$\|\text{sk}\|$	2816
Quantum security level	λ	100

Tweaking Falcon for SUP

Accounting for the aggregation of signatures require to tweak the parameters of Falcon. We use the parameters provided in table 1.

Aggregating k signatures still gives 2816 bytes. We briefly explain the rationale for our parameters:

- β is chosen so that the aggregation of k signatures will be shorter than β with overwhelming probability. This is described later in the paragraph *Security Analysis*.
- The security level slowly degrades as k increases. We fix an upper bound of 1000 on k, as it allows for a large number of aggregations while providing a reasonable bit-security of 100. This is discussed in the paragraph *Security Analysis* and illustrated by Figure 4.
- We require $q \geq \beta$ to definitely rule out a situation where security is unclear. This is described in the paragraph *The danger zone* $\beta \geq q$.
- We impose an additional bound β_∞ on the infinity norm as an additional safety measure. This is described later in the paragraph *The bound* β_∞.

The rest of this section details the choice of parameters, their interplay and how we obtain the claimed security level.

Security Analysis

The state of the art of the cryptanalysis on Falcon is given in [PFH+19, Section 2.5.2]. This include key recovery via lattice reduction, forgery via lattice reduction, hybrid attacks, "overstretched NTRU" attacks and algebraic attacks. Among these attacks, we found that the best cryptanalysis against Falcon is also the best one against our modified scheme: *forgery via lattice reduction*.

Each non-aggregated signature (s_1, s_2) follows a $2n$-dimensional Gaussian of standard deviation

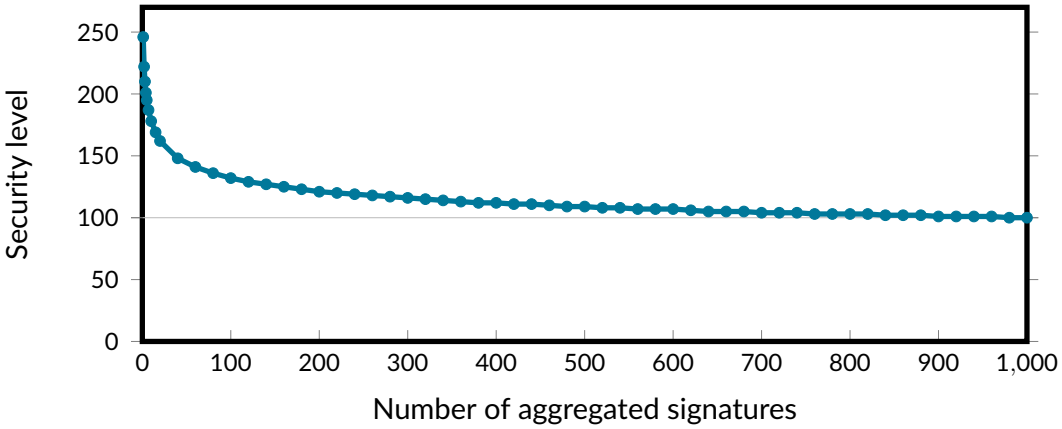

Figure 4: Security level as a function of the number of aggregated signatures

σ. Summing k of them yields a $2n$-dimensional Gaussian of standard deviation $\sigma\sqrt{k}$. According to [Lyu12, Lemma 4.4, Item 3], the norm of this sum is less than $\beta = \tau_{\text{sig}} \cdot \sigma\sqrt{2nk}$ with probability $\leq 2^{-111}$ for $\tau_{\text{sig}} = 1.2$.

In lattice-based signatures, the larger a signature is the easier it is to forge it. Therefore the security level will degrade with the number of aggregated signatures. We re-evaluate the security using Martin Albrecht's LWE estimator (https://bitbucket.org/malb/lwe-estimator). We found out that the security level enjoys a *graceful degradation* with the number of aggregations; it quickly dwindles when k is small, but for larger values of k the decrease is very small. Figure 4 illustrates the security level as a function of the number of aggregated signatures.

References

[BFRLS18] Pauline Bert, Pierre-Alain Fouque, Adeline Roux-Langlois, and Mohamed Sabt. Practical implementation of ring-SIS/LWE based signature and IBE. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 271–291. Springer, Heidelberg, 2018.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 216–233. Springer, Heidelberg, August 1994.

[BGG95] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *27th ACM STOC*, pages 45–56. ACM Press, May / June 1995.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Heidelberg, May 1997.

[CGM19] Yilei Chen, Nicholas Genise, and Pratyay Mukherjee. Approximate trapdoors for lattices and smaller hash-and-sign signatures. Cryptology ePrint Archive, Report 2019/1029, 2019. https://eprint.iacr.org/2019/1029.

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 22–41. Springer, Heidelberg, December 2014.

[GM18] Nicholas Genise and Daniele Micciancio. Faster gaussian sampling for trapdoor lattices with arbitrary modulus. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 174–203. Springer, Heidelberg, April / May 2018.

[GPR+18] Kamil Doruk Gür, Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, and Erkay Savas. Implementation and evaluation of improved gaussian sampling for lattice trapdoors. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '18, pages 61–71, New York, NY, USA, 2018. ACM.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

[LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. Securing update propagation with homomorphic hashing. Cryptology ePrint Archive, Report 2019/227, 2019. https://eprint.iacr.org/2019/227.

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In Pointcheval and Johansson [PJ12], pages 738–755.

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In Pointcheval and Johansson [PJ12], pages 700–718.

[MSO17] Sarah McCarthy, Neil Smyth, and Elizabeth O'Sullivan. A practical implementation of identity-based encryption over NTRU lattices. In Máire O'Neill, editor, *16th IMA International Conference on Cryptography and Coding*, volume 10655 of *LNCS*, pages 227–246. Springer, Heidelberg, December 2017.

[NIS19] NIST. Nistir 8240 - status report on the first round of the nist post-quantum cryptography standardization process, 2019. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf.

[PFH+19] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[PJ12] David Pointcheval and Thomas Johansson, editors. *EUROCRYPT 2012*, volume 7237 of *LNCS*. Springer, Heidelberg, April 2012.

[SS13] Damien Stehlé and Ron Steinfeld. Making NTRUEncrypt and NTRUSign as secure as standard worst-case problems over ideal lattices. Cryptology ePrint Archive, Report 2013/004, 2013. http://eprint.iacr.org/2013/004.