Université de Limoges

Thales Communications & Security

Master 2 Cryptis 2016-2017

Sécurité physique de schémas cryptographiques post-quantiques

Laurent Castelnovi

Rapport de stage

soutenu le 12 septembre 2017

Tuteurs TCS	Thomas Prest Ange Martinelli
Encadrant universitaire	Olivier Ruatta
Responsable pédagogique	Philippe Gaborit

Table des matières

Table des figures

Liste des algorithmes

Introduction générale

C'est au sein de l'équipe Cryptologie de Thales Communications & Security (TCS), située à Gennevilliers (Hauts-de-Seine) que j'ai effectué mon stage de fin d'études. L'entreprise fournit des solutions de communication sécurisées à un large éventail de clients allant des banques aux armées. L'équipe Cryptologie est chargée de proposer les schémas cryptographiques les mieux adaptés aux différents produits conçus par l'entreprise. À ce titre, elle se doit, d'une part, d'être toujours au niveau de l'état de l'art et, d'autre part, de développer de nouveaux schémas et de s'assurer de la sécurité de ceux existants.

Or, surtout dans le contexte des moyens de communications mobiles, la sécurité des schémas cryptographiques doit s'apprécier autant du point de vue mathématique que du point de vue physique. Qu'entendons-nous par : « physique » ? En fait ces schémas, s'ils sont mis en œuvre sans précaution, peuvent se révéler peu fiables à cause de défauts de conception qui ne sont pas affaire de mathématiques mais d'implémentation : le temps d'exécution de l'algorithme, la consommation électrique du composant qui l'exécute, la manipulation en mémoire des données qu'il utilise, sont autant de sources de renseignements au service de l'adversaire.

En effet, le temps d'exécution d'un algorithme, notamment ceux qui consistent essentiellement en une boucle conditionnelle, peut varier en fonction des données qu'il manipule. En particulier, si un attaquant sait que ce temps d'exécution varie en fonction de la valeur d'une donnée secrète, il peut la retrouver en mesurant cette durée. Ce type d'attaque est dit par analyse du temps d'exécution, ou *timing-attack* [Koc96].

La consommation électrique du composant qui effectue l'algorithme peut elle aussi varier en fonction des données qu'il manipule. Relever cette consommation pendant l'exécution de l'algorithme peut permettre de deviner les valeurs des données utilisées, ce qui pose un problème si elles sont secrètes ; ce type d'attaque est dit par analyse du courant, et a été décliné en plusieurs variantes. Les deux principales sont l'analyse dite *simple* [KJJ99], qui consiste à n'exécuter qu'une seule fois l'algorithme et à relever la consommation de courant induite, et celle dite *différentielle* [KJJ99, BCO04], qui consiste à exécuter plusieurs fois l'algorithme, chaque fois sur une entrée différente, à relever à chaque fois la consommation de courant induite puis à effectuer une analyse différentielle reposant sur ces courbes.

La manipulation des données en mémoire constitue également un danger : certains algorithmes utilisent des tables de valeurs précalculées pour gagner en vitesse. Lorsqu'un tel algorithme est lancé, les valeurs auxquelles il accède sont placées par l'ordinateur dans la mémoire cache (ou simplement : « cache »), une petite zone mémoire très rapide d'accès, afin que le processeur puisse disposer rapidement des valeurs qu'elle contient. Mais il peut être possible pour un attaquant de retirer certaines valeurs du cache. Si le processeur a besoin de ces valeurs, il doit alors aller les chercher en mémoire vive, d'accès beaucoup plus

lent. Cette différence de temps d'accès peut être perçue par l'attaquant qui en déduit que l'algorithme a manipulé la valeur retirée du cache. Cette attaque est dite *cache-timing*.

Ces vecteurs d'attaque sont exploités — du moins publiquement — depuis 1996 et l'article de P. Kocher [Koc96]. Ils ont depuis fait la preuve de leur efficacité : des *timing-attack* ont été menées avec succès contre RSA et de nombreux autres schémas [Koc96] ; l'analyse de la consommation de courant s'est avérée également redoutable [KJJ99, PPM17] ; l'observation des mouvements de données dans la mémoire cache est tout aussi prometteuse [BHLY16]. D'autres éléments physiques, comme les émissions électromagnétiques du composant de calcul [QS01], peuvent être exploités ; le lecteur intéressé peut se reporter à ce document technique [HRG+15] qui les mentionne de façon exhaustive.

Les attaques par injection de faute, qui consistent à perturber le fonctionnement de l'algorithme visé, se sont elles aussi montrées redoutablement efficaces contre un grand nombre de schémas [BDL01, BMM00, GKT10].

Ces attaques, dites *attaques physiques* ou *attaques par canaux auxiliaires* lorsque l'injection de faute n'est pas incluse, sont donc un objet de préoccupation aussi important que la sécurité théorique pour les concepteurs et intégrateurs des nouveaux schémas cryptographiques.

Parallèlement, l'ordre cryptographique établi est menacé à plus ou moins long terme par l'arrivée de l'ordinateur quantique. Un algorithme de P. Shor [Sho94] conçu à l'intention de cette machine casse sans trop d'espoir de rémission (cf par exemple [BHLV17]) tous les schémas actuels reposant sur la factorisation ou le logarithme discret et assurant la sécurité d'une part conséquente de nos communications. Depuis plusieurs années fleurissent donc de nouveaux schémas s'appuyant sur des problèmes mathématiques que l'ordinateur quantique peine encore à résoudre : parmi ces irréductibles se trouvent des problèmes liés aux codes correcteurs, aux polynômes multivariés, aux réseaux euclidiens ou aux fonctions de hachage. Certains de ces schémas dits *post-quantiques*, comme le protocole d'échange de clefs NewHope [ADPS15], qui repose sur les réseaux, sont d'ailleurs déjà utilisés.

Malheureusement, leur résistance à l'ordinateur quantique ne les prémunit en rien des attaques physiques. Eux aussi doivent être protégés contre elles, et ce fut là l'objet de mon stage.

Contributions de ce rapport. Ce rapport présente les deux principaux travaux que j'ai effectués durant le stage avec mes deux tuteurs, Thomas Prest et Ange Martinelli, tous deux ingénieurs cryptologues dans l'équipe Cryptologie de TCS. Chacun de ces deux travaux a permis d'étudier un versant différent de la sécurité physique : la cryptanalyse, d'une part, la conception de contremesures, d'autre part.

La première partie, qui s'est étendue de fin mars à début juin, a été consacrée à une attaque par injection de faute contre le schéma de signature SPHINCS [BHH+15], à base de fonctions de hachage. Nous proposons une attaque permettant de recueillir suffisamment d'éléments privés pour falsifier la signature d'un message arbitraire. Notre attaque ne requiert qu'une moyenne de 74,5 signatures fautées, voire une seule signature fautée en contrepartie d'une plus grande quantité de calcul. Elle fera l'objet d'une soumission à PQCrypto-2018.

La seconde partie, entre mi-juin et début août, a été consacrée à la conception sécurisée d'un générateur pseudo-aléatoire échantillonnant une loi normale sur $\mathbb{Z}$ de petit écart-type. Ce type de générateur est utilisé par des schémas cryptographiques basés sur les réseaux [GPV08, ABB10, Boy10]. Il s'est agi de trouver des vulnérabilités dans une première

implémentation et de proposer des contremesures adaptées, tout en améliorant la vitesse d'échantillonnage du générateur. Ce travail a abouti à un générateur qui s'exécute en temps indépendant de ses entrées et de ses sorties et qui fournit jusqu'à 14,5 millions d'échantillons par seconde, ce qui fait de lui l'un des échantillonneurs de loi normale discrète les plus rapides actuellement. Il devrait faire l'objet d'un article que nous co-écrivons avec Thomas Ricosset, doctorant dans l'équipe, et que nous comptons soumettre à PQCrypto-2018 également.

Pistes avortées. Entre-temps, ou parallèlement à ces recherches, quelques pistes ont été suivies ou quelques tentatives menées mais n'ont pas abouti. Au rang de celles-ci se trouvent la tentative d'apprivoiser un outil de simulation de fautes [1], celle tout aussi peu fructueuse de faire fonctionner celui de Y. Yarom, nommé Mastik [2], pour mener des attaques *cache-timing*, ou l'extension d'une analyse de courant à laquelle [EFGT17] a procédé contre la signature BLISS [DDLL13] en une *timing-attack*, comme le suggère T. Prest [Pre17, section 4.3], ou bien un embryon d'attaque contre RankSign [GRSZ14]. Il ne sera rien dit dans le rapport de ces travaux qui sont restés à l'état de brouillon, mais j'ai estimé que le temps qui leur a été consacré valait bien qu'on ne les oublie pas.

Remerciements. Avant d'entrer dans le vif du sujet je tiens à remercier Ange et Thomas pour tout, en particulier pour la patience notable dont ils ont fait preuve tout au long de ces six mois, surtout face à mes exposés et explications chaotiques ! Pour la relecture de ce rapport également, qu'ils ont grandement amélioré — voire réformé ; enfin pour m'avoir offert la chance d'effectuer ce stage avec eux et d'être co-auteur de deux articles que nous espérons bientôt voir sur l'ePrint !

Plus largement, je remercie toute l'équipe crypto : Olivier O., Olivier B., Alexandre, Sonia, David, Émeline, Mickaël, Renaud, Philippe, Sylvain, Éric, et les doctorants Thomas, Mélissa et Aurélien dont l'accueil a largement contribué à faire de ce stage une très bonne expérience. Mention spéciale pour les petites phrases bien senties de David qui vont me manquer ! Clin d'œil aux autres stagiaires de tous horizons cryptographiques : Matthieu, Nathy, Chloé, Jean-Paul, Thibaut, Simon avec lesquels j'ai partagé de (parfois abusivement) longues pauses-café !

Remerciements également à Andreas Hülsing et Tanja Lange dont les remarques ont parmi d'améliorer notablement notre attaque contre SPHINCS.

Enfin je n'oublie pas tous celles et ceux qui ont été à mes côtés durant ces six mois : mes frère, sœur et parents et leurs visites ressourçantes, mes camarades de Limoges devenus de bons amis Charline et Mathieu, enfin les Toulousaings Maxime et Maxime.

Organisation du rapport. Le rapport est divisé en trois parties. La première partie expose l'attaque par injection de faute que nous avons menée contre le schéma de signature SPHINCS ; elle comprend une description simplifiée du schéma, une explication théorique de l'attaque ainsi que les résultats de nos simulations et quelques pistes de contremesures. La deuxième partie détaille les contributions que j'ai apportées au générateur pseudo-aléatoire, en décrivant d'abord son fonctionnement, en étudiant ensuite la présence de fuites dues au temps d'exécution — seul paramètre qui nous ait intéressé pour les raisons avancées dans l'introduction à cette partie II — puis en exposant les contremesures

1. Qui se trouve à cette adresse : `https://github.com/Secure-Embedded-Systems/tsim-fault`.
2. Qui se trouve à cette adresse : `http://cs.adelaide.edu.au/~yval/Mastik/`.

développées pour remédier à ces fuites. Cette partie s'achève une estimation du débit du générateur et par une brève analyse des vulnérabilités restantes. On conclut le rapport dans une dernière partie.

Partie I

Attaque par injection de faute contre SPHINCS

Parmi les schémas de signature post-quantiques, ceux basés sur les fonctions de hachage semblent les plus robustes du point de vue de la sécurité. Celle-ci reposant sur la difficulté de trouver des collisions ou des antécédents pour une fonction de hachage, ces schémas sont à l'abri d'avancées mathématiques dans les domaines de théorie des nombres et d'algèbre soutenant d'autres schémas de signatures, comme [CFS01] ou [GRSZ14] reposant sur les codes correcteurs ou [DDLL13] sur les réseaux. Plus généralement, J. Rompel [Rom90] a montré qu'un schéma de signature sûr existait si, et seulement si, une fonction à sens unique existait, résultat qui, d'une certaine façon, permet même d'avancer que la sécurité des schémas à base de fonction de hachage est théoriquement optimale.

Le principal problème des signatures basées sur les fonctions de hachage [BDK+07, BCGD+06, BDH11, HRB13, DOTV08] réside dans le fait qu'elles nécessitent, pour la plupart, de maintenir un état, c'est-à-dire une liste de modifications apportées à la clef privée signature après signature. En plus d'être incompatible avec la définition stricte d'une clef privée, cela complique l'utilisation de la même clef privée par plusieurs signataires dépendant de la même personne — par exemple plusieurs serveurs. Le maintien d'un état peut certes être évité [Gol86], mais au prix d'une forte augmentation du temps et de la taille de la signature. Le schéma de signature SPHINCS [BHH+15], proposé par D. Bernstein et coll. en 2015, à base de fonctions de hachage, parvient cependant à supprimer la nécessité d'un état tout en offrant un temps et une taille de signature raisonnables.

La publication par A. Hülsing et coll. [HRS16a] d'une implémentation de SPHINCS sur microcontrôleur a ouvert la question de sa sécurité vis-à-vis des attaques physiques. Toutefois, à notre connaissance, aucune investigation sur le sujet n'a encore été menée.

La première partie de ce rapport y apporte donc une première contribution. On y expose une attaque par injection de faute à laquelle SPHINCS s'est trouvé sensible lors des simulations que nous avons menées sur l'implémentation *proof-of-concept* [BHH+14] de SPHINCS réalisée par les auteurs de [BHH+15]. Elle ne nécessite pas plus de 75 signatures fautées en moyenne pour donner à l'adversaire le pouvoir de falsifier la signature d'un message arbitraire ; ce pouvoir peut même être donné par une *unique* faute. De plus, il est possible, avec une très faible contrainte, de produire des signatures fautées valides.

Organisation de la partie. La première section I.1 a pour objet de décrire la fonctionnement de SPHINCS. Nous ne reprenons pas les détails théoriques de [BHH+15] ni, plus généralement, ceux qui ne sont pas nécessaires pour la compréhension de l'attaque. Celle-ci est exposée dans la section I.2, suivant deux stratégies aux avantages différents : l'une produit des signatures fautées valides mais le moment durant lequel infliger la faute est plus contraint que dans la seconde qui, en contrepartie, requiert plus de signatures fautées — potentiellement invalides. Cette section I.2 détaille également une estimation du nombre de signatures fautées requis par chaque stratégie et les modalités de nos simulations ; on y aborde finalement la question des contremesures. Enfin, on conclut ce travail dans la section I.3.

Notations. On utilisera dans toute cette partie les notations suivantes :
- log pour $\log_2$;
- λ pour le paramètre de sécurité de SPHINCS ;
- H pour une fonction de hachage cryptographique de $\{0,1\}^*$ dans $\{0,1\}^\lambda$;
- $\mathrm{A}(f)$ pour le chemin d'authentification de la feuille f (voir la section I.1.1) ;
- $\&f$ pour l'adresse de la feuille f (cf section I.1.2).

De plus, on conviendra que :
- les étages des arbres sont comptés depuis les feuilles (étage 0) jusqu'à la racine ;
- un étage d'un super-arbre SPHINCS correspond à une couche d'arbres de Merkle (voir aussi la figure I.3) ;
- la racine d'un arbre n'est jamais comptée dans le calcul de sa hauteur mais est toujours considérée comme un étage de l'arbre, de sorte qu'un arbre (binaire équilibré) de hauteur h possède 2^h feuilles et $h+1$ étages.

I.1 Fonctionnement général de SPHINCS

I.1.1 Structure du super-arbre SPHINCS

Le schéma de signature SPHINCS repose sur la composition d'arbres de Merkle et d'un arbre de Goldreich pour former ce qu'on appelle un *super-arbre* SPHINCS. Ces deux types d'arbre — Merkle et Goldreich — sont le fruit des propositions respectives de R. Merkle [Mer90] et de O. Goldreich [Gol86] pour répondre au problème suivant : comment, d'une signature à usage unique, c'est-à-dire un schéma dans lequel une même clef privée ne peut être utilisée qu'une seule fois, obtenir une *many-time signature*, c'est-à-dire un schéma où la même clef privée peut être utilisée un nombre fixé de fois ? Cette sous-section rappelle brièvement en quoi consistent ces deux propositions.

Arbre de Merkle

R. Merkle propose dans [Mer90] de s'appuyer sur les arbres de hachage pour dériver un schéma de signature *many-time* d'une une signature à usage unique — OTS, pour *one-time signature*, dans la suite. Un arbre de hachage est un arbre binaire construit depuis les feuilles jusqu'à la racine, le nœud père de deux nœuds fils n_1 et n_2 étant le haché de $n_1||n_2$. Dans le schéma de R. Merkle, chaque feuille de l'arbre de hachage est une clef publique pour une OTS (on appellera désormais *arbre de Merkle* cet arbre de hachage particulier). La clef publique de ce schéma est la racine de l'arbre de hachage et la clef

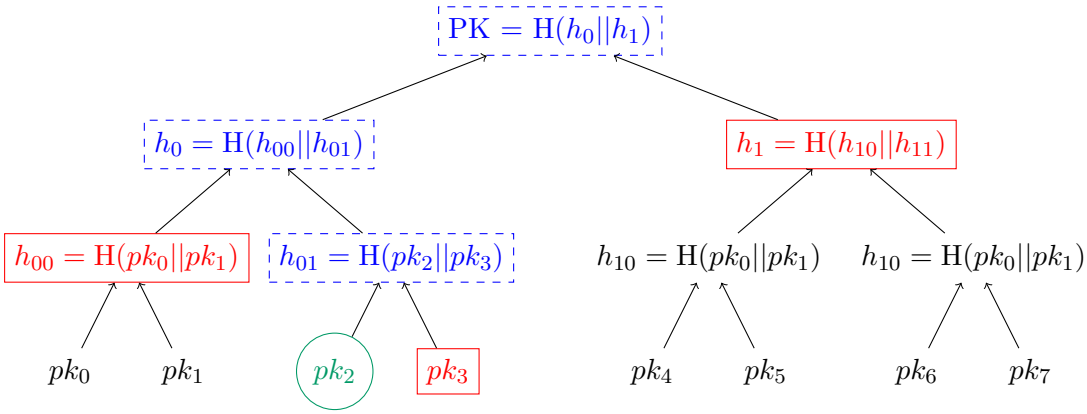

FIGURE I.1 – Un arbre de Merkle. Encerclée, une feuille f d'arbre de Merkle, encadrés en trait plein les éléments de son chemin d'authentification $\mathrm{A}(f)$, encadrés en trait discontinus les nœuds qui peuvent être calculés grâce à f et $\mathrm{A}(f)$.

privée est l'ensemble des clefs privées d'OTS dont les clefs publiques sont les feuilles de l'arbre.

Pour signer un message, une des clefs privées est utilisée. La clef publique qui lui correspond et son chemin d'authentification, c'est-à-dire l'ensemble des nœuds — la racine exceptée — de l'arbre de Merkle qui doivent être révélés pour calculer sa racine, sont ensuite transmis avec la signature. Le receveur vérifie la signature du message grâce à la clef publique donnée et s'assure que cette clef est légitime en recalculant la racine de l'arbre de Merkle (voir la figure I.1).

Cette signature possède toutefois deux inconvénients importants. En premier lieu, le temps de signature est exponentiel en la hauteur de l'arbre, car il doit être entièrement recalculé à chaque signature, à moins qu'on ne le conserve en mémoire, auquel cas signer devient exponentiel en espace. En second lieu, puisque les clefs privées sont celles d'une OTS, il faut conserver en mémoire, pour ne pas les réutiliser, une liste des clefs usagées, qu'on appelle état. Cela pose un problème, par exemple, lorsqu'on veut que plusieurs signataires puissent utiliser la même clef privée (par exemple, plusieurs serveurs). Trouver une solution sans état est alors nécessaire.

Signature de Goldreich

O. Goldreich [Gol86] propose, tout comme Merkle, une construction à base d'arbre, mais sans état. L'arbre de Goldreich est en fait un arbre binaire équilibré où chaque nœud est une paire de clefs pour une OTS. La paire de clefs d'un nœud père sert à signer le haché des clefs publiques des nœuds fils concaténées. Les clefs contenues dans les feuilles serviront à signer les messages. Les nœuds sont indexés de façon unique par une chaîne de bits et engendrés à partir d'une graine (partie de la clef privée de l'arbre) et de cet index. La clef publique de l'arbre entier est la clef publique de la racine de l'arbre. La clef privée est composée de la clef privée de la racine et de la graine à partir de laquelle tout l'arbre est engendré. La figure I.2 décrit tout cela.

Concrètement, supposons qu'on veuille signer un message m en utilisant la signature de Goldreich — on se réfère dans toute cette description à la figure I.2) :

1. sélectionner au hasard[1] une feuille de l'arbre et utiliser la paire de clefs qu'elle contient (ici (pk_{101}, sk_{101})) pour signer m ;
2. prendre la clef publique de cette feuille (pk_{101}), la concaténer avec la clef publique de sa sœur (pk_{100}) et en signer le haché en utilisant la clef privée contenue dans leur nœud père (sk_{10}) ;
3. prendre la clef publique de ce nœud (pk_{10}), la concaténer avec la clef publique de son frère (pk_{11}) et en signer le haché en utilisant la clef privée contenue dans leur nœud père (sk_1) ;
4. ainsi de suite jusqu'à la racine ;
5. *signature :* l'ensemble des signatures produites aux quatre étapes précédentes, et l'ensemble des clefs publiques pour les vérifier.

La signature est valide si, et seulement si, toutes les signatures qu'elle contient sont valides.

Pour un arbre de hauteur 256 — soit 128 bits de sécurité —, la signature finale du message contient 256 signatures et 512 clefs publiques. Pour donner une idée de la taille de signature qu'on obtient, supposons que l'OTS utilisée soit WOTS+ (cf I.1.2 pour une description et I.1.2 pour les paramètres) ; alors la signature de Goldreich d'un message de 256 bits pèse environ 1,65 Mo.

1. Comme la signature qu'on obtient est requise sans état, il y a un très grand nombre de feuilles. Il en faut 2^{256} pour 128 bits de sécurité à cause du paradoxe des anniversaires.

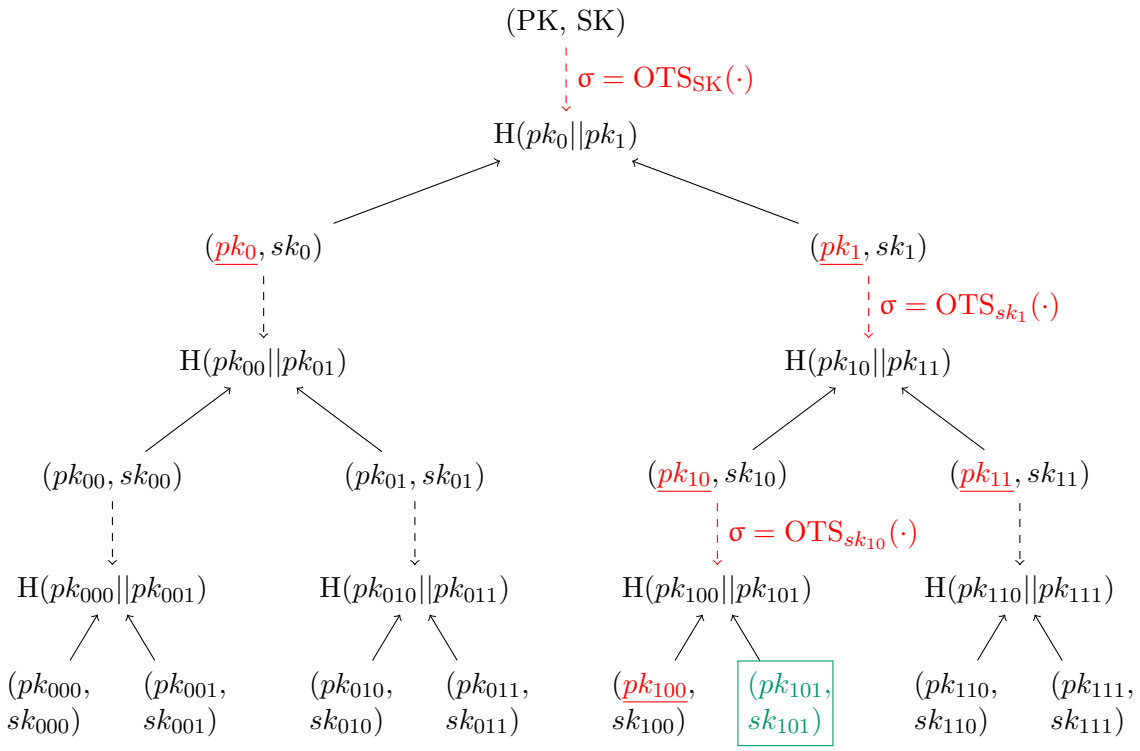

FIGURE I.2 – Un arbre de Goldreich. Soulignés, tous les éléments transmis pour la signature d'un message par la feuille encadrée. Sont également transmises les signatures correspondant aux flèches notées par un σ. Les flèches discontinues signifient : « signe ».

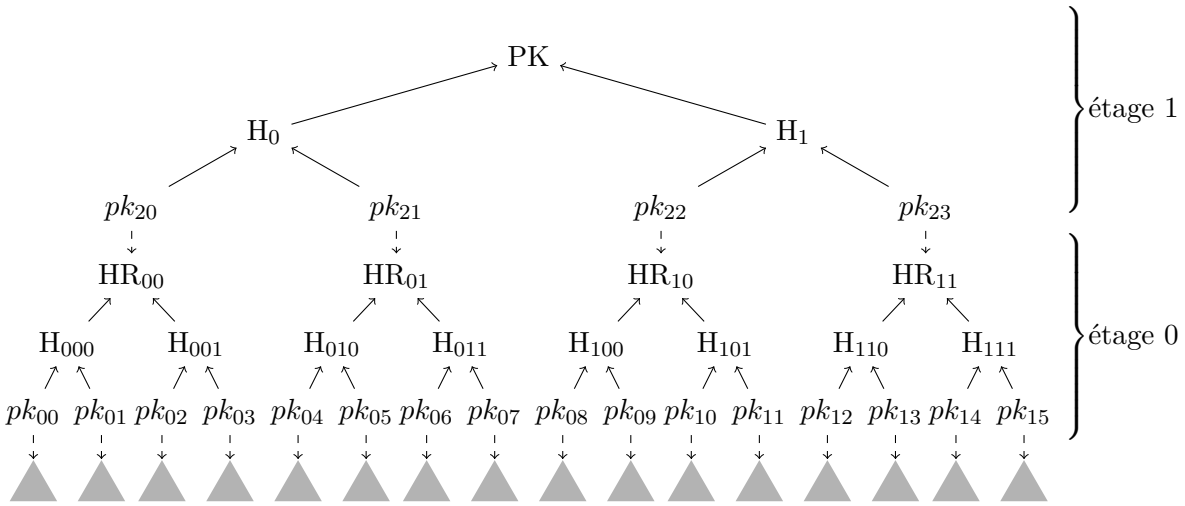

FIGURE I.3 – Un super-arbre SPHINCS à deux étages (hauteur 4), ou arbre de Goldreich à deux étages d'arbres de Merkle de hauteur 2. La lettre H rappelle que le nœud est un haché, la lettre R une racine. Les flèches discontinues signifient : « signe », les continues : « [pk] est la moitié de la chaîne dont le haché est [H] ». Les triangles gris sont les clefs publiques de la signature *few-time*.

Arbre SPHINCS

L'objectif de SPHINCS est d'allier le caractère sans état de la signature de Goldreich à la taille raisonnable de la signature de Merkle. Pour atteindre ce but, on fait une observation concernant l'arbre de Goldreich : on peut voir les structures en trait plein de la figure I.2 comme des arbres de Merkle à deux étages, donc l'arbre de Goldreich total comme un arbre dont les nœuds sont des arbres de Merkle.

SPHINCS ne fait que généraliser cette observation : un arbre SPHINCS de hauteur h est un arbre de Goldreich à d étages d'arbres de Merkle de hauteur h/d. La même idée se retrouve d'ailleurs dans GMSS [BDK+07] ou XMSS [BDH11, HRB13]. Une nouveauté a toutefois été introduite dans SPHINCS pour pouvoir diminuer h sans sacrifier la sécurité et la réutilisabilité du schéma : les feuilles de l'étage 0 de l'arbre SPHINCS ne servent pas à signer directement un message mais à signer une clef publique d'un schéma de signature *few-time*, dont une même clef privée ne peut signer que quelques messages différents (voir la figure I.3).

Enfin, il faut noter que SPHINCS utilise une forme légèrement modifiée d'arbre de Merkle : avant d'être haché, le concaténé de deux nœuds frères subit d'abord un « ou exclusif » avec un masque. Cette variante, proposée par [DOTV08], permet de n'exiger de H que la résistance à la seconde pré-image et pas celle aux collisions, ce qui offre un meilleur niveau de sécurité [2].

2. Il existe en effet un algorithme quantique de G. Brassard et coll. [BHT98] qui renvoie une collision d'une fonction $h : \mathrm{E_D} \to \mathrm{E_A}$, où $\mathrm{E_D}$ et $\mathrm{E_A}$ sont deux ensembles finis non vides tels que $\operatorname{card} \mathrm{E_A} = n$ et $\operatorname{card} \mathrm{E_D} = rn$, en $\mathrm{O}(\sqrt[3]{n/r})$ évaluations de h, alors que la résistance à la seconde préimage n'a toujours pas été éprouvée, à notre connaissance, par un algorithme (quantique ou pas) meilleur que celui de L. Grover [Gro96] qui donne un résultat en $\mathrm{O}(\sqrt{n})$.

I.1.2 Signature SPHINCS

On explique maintenant comment procède SPHINCS pour produire une signature. On commence par décrire l'OTS choisie dans SPHINCS.

WOTS+

Il s'agit du schéma de signature à usage unique utilisé dans SPHINCS et décrit dans [Hül13]. Il ne peut signer que des messages de taille fixée à l'avance ; dans SPHINCS cette taille s'élève à λ bits. Le schéma possède également un paramètre entier $w > 0$. On définit maintenant les trois valeurs suivantes :

$$\ell_1 = \left\lceil \frac{\lambda}{\log w} \right\rceil, \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log w} + 1 \right\rfloor, \ell = \ell_1 + \ell_2.$$

Pour $x \in \{0,1\}^\lambda$ et $\boldsymbol{r} = (r_1, \ldots, r_{w-1}) \in (\{0,1\}^\lambda)^{w-1}$, on définit pour $0 \leqslant i \leqslant w-2$:

$$c_{i+1}(x, \boldsymbol{r}) = \mathrm{H}(c_i(x, \boldsymbol{r}) \oplus r_{i+1})$$

avec $c_0(x, \boldsymbol{r}) = x$.

On peut à présent décrire l'algorithme de génération de clefs :

1. prendre au hasard une graine $g \in \{0,1\}^\lambda$ et un vecteur $\boldsymbol{r} \in (\{0,1\}^\lambda)^{w-1}$;
2. soit $(\breve{s}_1, \ldots, \breve{s}_\ell)$ ℓ chaînes de λ bits engendrées pseudo-aléatoirement à partir de la graine g ;
3. calculer $(\breve{p}_i = c_{w-1}(\breve{s}_i, \boldsymbol{r}))_{1 \leqslant i \leqslant \ell}$;
4. *clef publique :* $(\breve{p}_1, \ldots, \breve{p}_\ell)$ et $\boldsymbol{r}$, *clef privée :* $(\breve{s}_1, \ldots, \breve{s}_\ell)$.

Pour la signature, on procède ainsi, avec m le message à signer :

1. exprimer m en base w : $m = (m_1 \ldots m_{\ell_1})_w$;
2. calculer[3] $\kappa = \sum_{i=1}^{\ell_1}(w - 1 - m_i)$;
3. exprimer κ en base w : $\kappa = (\kappa_1 \ldots \kappa_{\ell_2})_w$;
4. on note : $\boldsymbol{b} = (b_1, \ldots, b_\ell) = (m_1, \ldots, m_{\ell_1}, \kappa_1, \ldots, \kappa_{\ell_2})$;
5. *signature :* $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_\ell) = (c_{b_i}(\breve{s}_i, \boldsymbol{r}))_{1 \leqslant i \leqslant \ell}$.

La vérification est naturelle :

1. calculer $\boldsymbol{b} = (b_1, \ldots, b_\ell)$ comme dans l'algorithme de signature ;
2. accepter la signature si, et seulement si :

$$(\breve{p}_1, \ldots, \breve{p}_\ell) = (c_{w-1-b_i}(\sigma_i, (r_{b_i+1}, \ldots, r_{w-1})))_{1 \leqslant i \leqslant \ell}.$$

On souligne, car ce sera d'intérêt dans la suite, qu'une clef publique WOTS+ se déduit d'une signature WOTS+ légitime.

3. Il s'agit d'une somme de contrôle nécessaire pour la sécurité : si l'on reçoit la signature d'un message $m = (m_1 \ldots m_{\ell_1})_w$ sans somme de contrôle, il est trivial de falsifier une signature d'un message m' tel que $m'_i \geqslant m_i$ pour tout $1 \leqslant i \leqslant \ell_1$. La somme de contrôle garantit que, dans ce cas, au moins l'un des κ'_i vérifiera $\kappa'_i < \kappa_i$, ce qui rend la falsification impossible sans inverser la fonction de hachage.

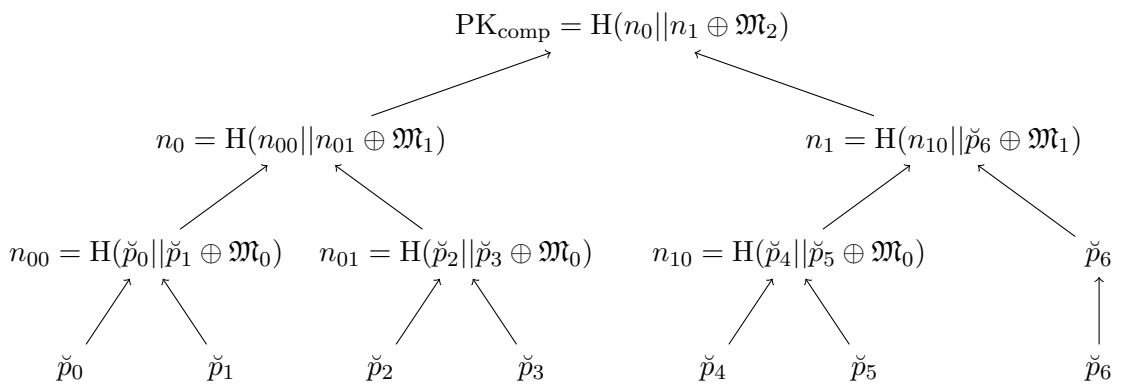

FIGURE I.4 – Processus de compression des clefs publiques WOTS+. On l'illustre pour une clef publique réduite à sept éléments. C'est donc $\mathrm{PK}_{\mathrm{comp}}$ qui servira de feuille à un arbre de Merkle de SPHINCS. Les $\mathfrak{M}_i$ sont des masques.

HORST

HORST est la signature *few-time* avec laquelle SPHINCS signe les messages. C'est une variante de HORS [RR02] adaptée sous forme d'arbre pour réduire la taille de la signature. Comme sa connaissance n'est pas d'intérêt pour l'attaque qu'on présentera, on se bornera à souligner, que comme WOTS+, HORST est entièrement déterministe et qu'une signature HORST légitime permet de déduire une clef publique HORST.

Signature SPHINCS

SPHINCS est une association des signatures de Goldreich et de Merkle. Avant de décrire en détail son fonctionnement, il importe d'attirer l'attention du lecteur sur deux points.

D'abord, il faut signaler que les arbres de Merkle présents dans SPHINCS sont construits de façon un peu différente de celle exposée en I.1.1. En plus du fait que les opérations de hachage fassent intervenir des masques (cf I.1.1), les feuilles sont des *formes compressées* de clef publique. Les clefs publiques WOTS+ sont en effet assez grosses (2,1 ko pour SPHINCS-256, voir I.1.2 pour les paramètres) ; elles subissent donc le traitement suivant : les ℓ parties de la clef publique sont placées chacune dans une feuille d'un arbre de hachage à ℓ feuilles. On calcule ensuite la racine de l'arbre de hachage, chaque opération de hachage faisant intervenir un masque ; si un nœud se trouve sans frère, il passe sans transformation dans l'étage supérieur de l'arbre jusqu'à en avoir un (figure I.4). La racine de cet arbre de hachage — qu'on appellera dans la suite *arbre de compression* — est la forme compressée de la clef publique WOTS+. Cette procédure permet d'éviter de supposer de H qu'elle soit résistante aux collisions [HRB13].

Ensuite, à chaque feuille de chaque arbre de Merkle présent dans le super-arbre, on va associer une adresse unique. Cette adresse est construite ainsi :

— $\lceil \log(d+1) \rceil$ bits pour le numéro de l'étage du super-arbre dans laquelle se trouve la feuille ;
— $(d-1)\frac{h}{d}$ bits pour le numéro de l'arbre dans l'étage (en comptant de gauche à droite) ;
— $\frac{h}{d}$ bits pour le numéro de la feuille dans l'arbre (en comptant de gauche à droite).

Cette adresse servira, comme dans Goldreich (cf I.1.1), de graine pour engendrer les clefs WOTS+ qui s'y trouvent.

Génération des clefs. Le super-arbre qu'on construit ici possède d étages d'arbres de Merkle de hauteur h/d, où h est donc la hauteur totale du super-arbre.

1. Prendre $(\breve{\mathrm{S}}_1, \breve{\mathrm{S}}_2) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ au hasard ;
2. prendre un certain nombre p de masques[4] $\mathbf{Q} = (\mathrm{Q}_1, \ldots, \mathrm{Q}_p) \in \{0,1\}^{\lambda \times p}$;
3. engendrer l'arbre de Merkle de l'étage $d-1$;
4. *clef publique :* $\mathbf{Q}$ et la racine de l'arbre de l'étage $d-1$, *clef privée :* $(\breve{\mathrm{S}}_1, \breve{\mathrm{S}}_2)$.

Signature. On signe un message m de longueur quelconque. La figure I.5 explique également cette signature en reprenant les notations ci-après et en gérant les masques comme ils le sont dans le schéma.

1. engendrer pseudo-aléatoirement $(\mathrm{R}_1, \mathrm{R}_2) \in (\{0,1\}^\lambda)^2$ à partir de m et de $\breve{\mathrm{S}}_2$;
2. calculer $\mathrm{D} = \mathrm{H}(\mathrm{R}_1 || m)$;
3. extraire $i =$ les $(d-1)\frac{h}{d}$ bits de poids fort de R_2, et $j =$ les $\frac{h}{d}$ bits suivants ;
4. engendrer une paire de clefs HORST ;
5. $\boldsymbol{\sigma}_\mathrm{H} =$ signature de D en utilisant cette paire de clefs ;
6. soit $\mathfrak{f}_0$ la feuille d'adresse $0||i||j$:
   (a) engendrer pseudo-aléatoirement une graine g à partir de $\&\mathfrak{f}_0$ et de $\breve{\mathrm{S}}_1$;
   (b) engendrer pseudo-aléatoirement la clef privée WOTS+ associée à $\mathfrak{f}_0$ en utilisant g comme graine ;
   (c) $\boldsymbol{\sigma}_0 =$ la signature de la racine de l'arbre HORST en utilisant cette clef WOTS+ ;
7. engendrer pseudo-aléatoirement à partir de $\breve{\mathrm{S}}_1$ et de leur adresse toutes les feuilles de l'arbre de Merkle courant et en déduire $\mathrm{A}(\mathfrak{f}_0)$;
8. pour $1 \leqslant z < d$:
   (a) $\boldsymbol{\sigma}_z =$ la signature de la racine de l'arbre de Merkle qui contient $\mathfrak{f}_{z-1}$, en utilisant la clef privée WOTS+ associée à la feuille $\mathfrak{f}_z$ qui convient[5] ;
   (b) engendrer pseudo-aléatoirement à partir de $\breve{\mathrm{S}}_1$ et de leur adresse toutes les feuilles de l'arbre de Merkle courant et en déduire $\mathrm{A}(\mathfrak{f}_z)$;
9. *signature :* $\boldsymbol{\sigma} = (i||j, \mathrm{R}_1, \boldsymbol{\sigma}_\mathrm{H}, \boldsymbol{\sigma}_0, \mathrm{A}(\mathfrak{f}_0), \ldots, \boldsymbol{\sigma}_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1}))$.

Vérification.

1. Calculer $\mathrm{D} = \mathrm{H}(\mathrm{R}_1 || m)$;
2. calculer la clef publique HORST (qui est une racine d'arbre de hachage) en supposant que $\boldsymbol{\sigma}_\mathrm{H}$ est valide ;
3. pour $0 \leqslant i < d$:

4. Dans SPHINCS-256, ce nombre est imposé par HORST à 32.
5. Si $\&\mathfrak{f}_{z-1} = x||y||v$, où x, y et v sont vus comme des entiers, alors $\&\mathfrak{f}_z = (x+1)||q||r$ avec $y = q2^{h/d} + r$ et $r < 2^{h/d}$.

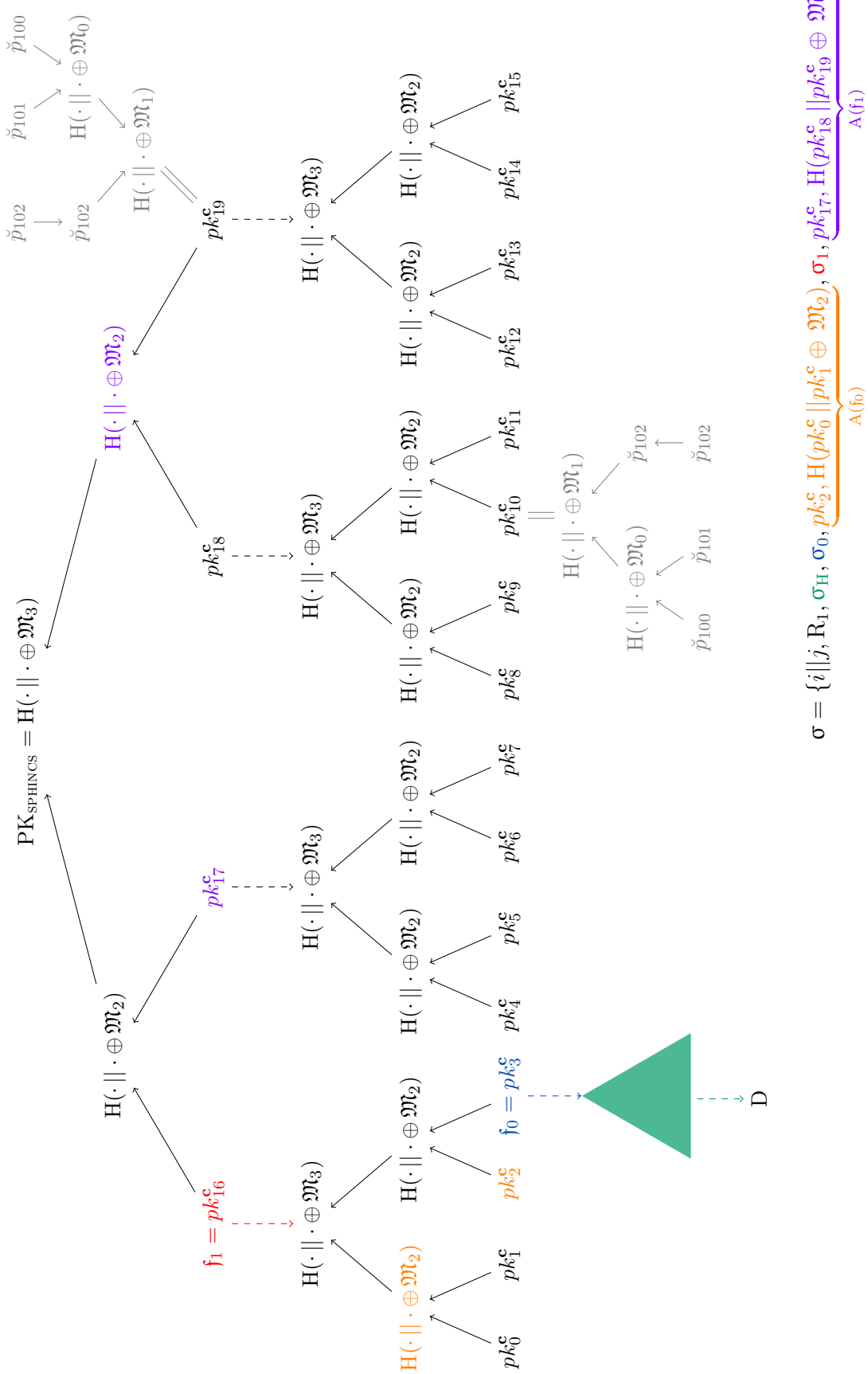

FIGURE I.5 – Schéma général de SPHINCS. Sur le schéma, $\mathfrak{M}_i = \mathrm{Q}_{2i}\|\mathrm{Q}_{2i+1}$, les flèches discontinues signifient : « signe », les flèches pleines signifient : « est un argument de ». Le triangle est un arbre HORST, les deux arbres gris sont des arbres de compression de clef WOTS+ ; on ne les a bien entendu pas tous représentés. Le **c** en exposant rappelle que les feuilles sont des clefs publiques compressées. σ est la signature de D.

(a) supposer que la signature $\boldsymbol{\sigma}_i$ est valide et en déduire une clef publique (le message dont $\boldsymbol{\sigma}_i$ est la signature est en effet connu : c'est la racine calculée au tour précédent) ;

(b) supposer que $\mathrm{A}(\mathfrak{f}_i)$ est correct et calculer la racine de l'arbre de Merkle ;

4. accepter si, et seulement si, la dernière racine obtenue est égale à la clef publique de SPHINCS.

Paramètres pour SPHINCS-256. Les paramètres pour SPHINCS-256 sont les suivants : $\lambda = 256$, $w = 16$, $h = 60$, $d = 12$. Il en découle $\ell = 67$ et $p = 32$ (pour cette dernière valeur, les paramètres pour HORST sont dominants dans le calcul). Une clef publique pèse 1 056 octets, une clef privée 1 088 octets et une signature 41 000 octets. Ceci dit, [HRS16b] propose un moyen de diminuer de presque moitié la taille des clefs et, surtout, de la signature.

I.2 Attaque par injection de faute contre SPHINCS

L'objet de nos travaux a été d'étudier la résistance de SPHINCS aux attaques physiques en supposant sûres vis-à-vis de ces attaques les fonctions de hachages et les générateurs pseudo-aléatoires sous-jacents, puisqu'ils peuvent être changés s'il apparaît qu'ils ne sont pas sûrs.

Pour atteindre son objectif qui consiste à créer des signatures valides pour une clef publique qui ne lui appartient pas, un attaquant peut soit trouver un moyen de falsifier des signatures à partir de celles, légitimes, qu'il a collectées, soit tenter de retrouver des éléments privés du schéma.

Dressons une liste des éléments privés disséminés dans SPHINCS : $\breve{\mathrm{S}}_1$ et $\breve{\mathrm{S}}_2$ bien sûr, mais aussi toutes les clefs privées HORST et WOTS+ qui bourgeonnent partout dans le super-arbre. Les tentatives pour retrouver $\breve{\mathrm{S}}_1$ et $\breve{\mathrm{S}}_2$ ont tourné court :

— en ce qui concerne l'analyse du temps de signature : les hypothèses qu'on a faites sur les fonctions qui composent SPHINCS impliquent que ce temps ne dépend ni de $\breve{\mathrm{S}}_1$, ni de $\breve{\mathrm{S}}_2$;

— en ce qui concerne l'analyse du courant : $\breve{\mathrm{S}}_1$ et $\breve{\mathrm{S}}_2$ ne sont, en-dehors de ces fonctions, manipulées dans [BHH+14] (qui est notre implémentation de travail) que par des boucles qui servent à les copier où à les annuler, opérations peu propices à ce type d'attaque.

Restent les clefs privées HORST et WOTS+. En ce qui les concerne également, le temps de calcul d'une signature HORST ou WOTS+ est indépendant d'elles. Pour ce qui est de la consommation de courant, son exploitation ne semble pas évidente. La réflexion sur cette question a été, il est vrai, abrégée par l'approfondissement de la piste explorée dans cette section. Elle tire parti d'une caractéristique structurelle que le super-arbre SPHINCS hérite de sa parenté avec l'arbre de Goldreich : ses étages sont indépendants les uns des autres et seulement liés par une OTS, ici WOTS+.

Notre attaque profite justement du fait que WOTS+ soit à usage unique. L'idée est de forcer l'algorithme de signature à signer avec une même clef privée WOTS+ plusieurs messages différents afin de la reconstituer en entier. Ainsi, l'attaquant serait capable de substituer à la branche légitime signée par cette clef sa propre branche qu'il pourrait utiliser

pour forger[6] des signatures valides pour la clef publique SPHINCS légitime. En particulier, si l'attaquant peut retrouver toutes les clefs privées WOTS+ de l'arbre du dernier étage, il sait signer n'importe quel message en lieu et place du signataire légitime.

C'est la reconstruction partielle, totalement déterministe, de l'arbre SPHINCS à chaque signature qui rend ceci envisageable dans le cadre d'une attaque par injection de faute.

Formellement, soit une paire de clefs WOTS+ $(\boldsymbol{sk}, \boldsymbol{pk})$ signant une racine δ de l'étage[7] $d-2$, avec l'objectif de recouvrer $\boldsymbol{sk} = (\check{s}_1, \check{s}_2, \ldots, \check{s}_\ell)$ (cf I.1.2). Demandons d'un message m la signature SPHINCS, dont on suppose qu'elle nécessite la signature de δ (notons cette dernière signature $\boldsymbol{\sigma}_{d-1} = (\sigma_{d-1,1}, \ldots, \sigma_{d-1,\ell})$). On reçoit $\boldsymbol{\sigma} = (x, \mathrm{R}_1, \boldsymbol{\sigma}_\mathrm{H}, \boldsymbol{\sigma}_0, \mathrm{A}(\mathfrak{f}_0), \ldots, \boldsymbol{\sigma}_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1}))$. Regardons la succession des calculs qui ont été faits pour l'obtenir (on suit [BHH+14] qui est l'implémentation des auteurs de [BHH+15]) :

1. de m et de $\check{\mathrm{S}}_2$ ont été dérivés x, $\mathrm{R}_1, \mathrm{R}_2$ et D ;
2. l'arbre HORST signataire (appelons-le $\mathcal{A}$) a été choisi à partir de R_2 puis engendré ;
3. D a été signé par $\mathcal{A}$;
4. la clef WOTS+ qui doit signer $\mathcal{A}$ a été engendrée puis $\mathcal{A}$ a été signé ;
5. l'arbre de Merkle contenant cette clef (compressée) a été construit ;
6. la clef WOTS+ qui doit signer la racine de cet arbre a été engendrée puis la racine a été signée ;
7. les points 5 et 6 ont été répétés jusqu'à atteindre la racine du super-arbre.

Cet algorithme est entièrement déterministe. En particulier, pour un même message m, l'ensemble des racines signées sera toujours le même, tout comme l'ensemble des clefs WOTS+ qui servent à les signer. Ainsi, si on perturbe aléatoirement le calcul de δ lorsqu'on demande une signature de m, on récupère une signature d'une racine distincte de δ, effectuée avec la même clef privée WOTS+ (figure I.6).

Maintenant, intéressons-nous à la façon dont WOTS+ livre des morceaux de sa clef privée. Du message signé (ici δ) est directement dérivé un vecteur $\boldsymbol{b} = (b_1, b_2, \ldots, b_\ell)$ avec $0 \leqslant b_i < w$ pour tout i, puis la i^e partie de la signature est définie comme étant $c_{b_i}(\check{s}_i, \boldsymbol{r})$ avec $c_0(\check{s}_i, r) = \check{s}_i$ (cf I.1.2 pour un exposé détaillé). Par suite, la i^e partie d'une signature WOTS+ est un morceau de clef privée si, et seulement si, $b_i = 0$. Pour reconstituer $\boldsymbol{sk}$, l'attaquant doit donc être capable d'identifier si le coefficient b_i qui a servi à calculer $\sigma_{d-1,i}$ est nul ou pas. Il peut le faire de deux façons ($\star$ signifiera : « fauté ») :

— **stratégie 1 :** en retrouvant la valeur de $\delta^\star$ à partir de $\mathrm{A}(\mathfrak{f}_{d-2})^\star$ et en calculant le vecteur $\boldsymbol{b}$ induit ;

— **stratégie 2 :** en demandant suffisamment de signatures fautées pour disposer de toutes les valeurs pouvant être prises par chacun des $\sigma_{d-1,i}$.

Dans la suite, on développe ces deux stratégies dans le cas particulier de SPHINCS-256.

I.2.1 Première stratégie

Remarquons que, pour que la première stratégie réussisse, $\mathrm{A}(\mathfrak{f}_{d-2})$ doit être fauté : dans le cas contraire, c'est δ qu'on en déduirait et non $\delta^\star$. Cela implique qu'on ne peut

6. C'est un anglicisme — issu du verbe anglais *to forge* — mais il a le mérite d'être éloquent ; nous l'emploierons donc assez souvent.

7. On se place dans ce cas particulier car c'est le plus intéressant du point de vue de l'attaquant, mais l'attaque peut viser n'importe quelle feuille présente dans le super-arbre SPHINCS.

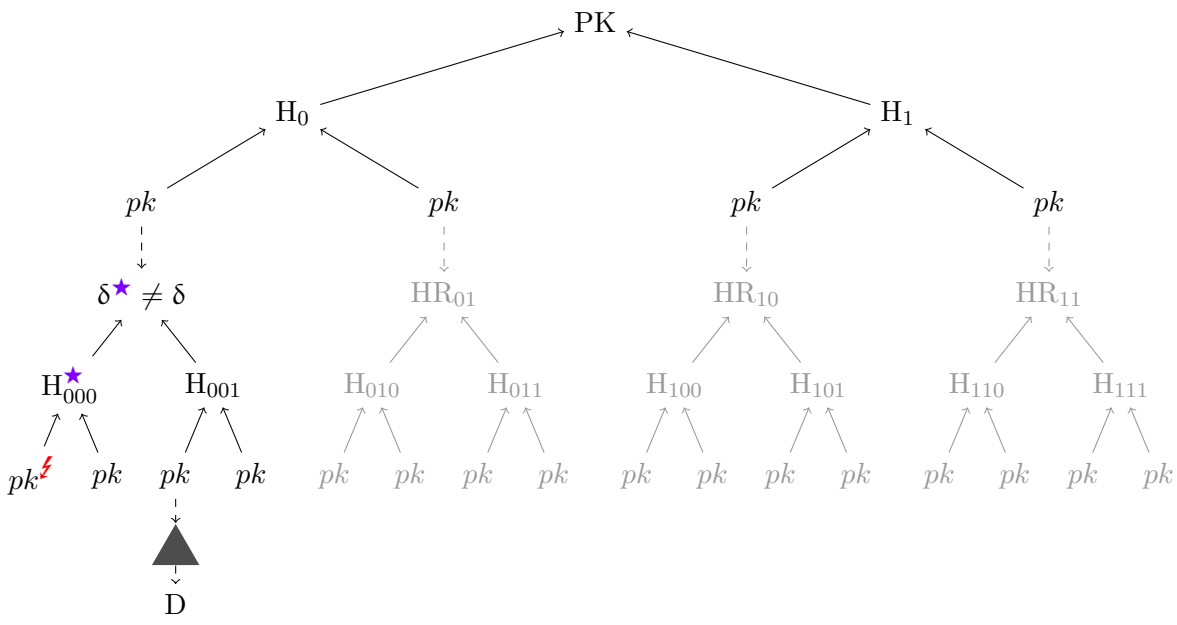

FIGURE I.6 – Le principe de l'attaque. La partie de l'arbre qui est reconstruite pendant la signature est en noir. La faute est symbolisée par un éclair, les éléments modifiés en conséquence par une étoile.

provoquer de faute directe sur les nœuds dont le calcul doit être refait par le vérificateur de la signature, mais, ceux-ci écartés, tous les autres peuvent être fautés pour mener à bien l'attaque (soulignons à cette fin qu'une feuille erronée peut être la conséquence d'une compression erronée). En d'autres termes, on peut fauter tous les nœuds qui se situent « en-dessous » du chemin d'authentification, tandis qu'on ne peut pas fauter ceux qui se trouvent « au-dessus » (voir la figure I.7). Cette restriction n'est pas très contraignante : sur les $32 \times 66 + 31 = 2\,143$ hachés nécessaires pour calculer δ — il en faut, en effet, soixante-six pour compresser une clef publique WOTS+, qui sont au nombre de trente-deux, auxquels il faut ajouter les trente-et-un hachés nécessaires pour calculer, en ayant les feuilles, la racine de l'arbre —, seuls $66 + 5 = 71$ d'entre eux ne sont pas exploitables pour notre attaque.

Supposons à présent qu'un attaquant ait fait signer un message m et qu'il soit parvenu à injecter une faute au moment adéquat. Soit $\boldsymbol{\sigma}^\star$ la signature fautée qu'il recueille. Pour en déduire $\delta^\star$, il lui suffit d'exécuter dessus l'algorithme de vérification qui doit calculer cette racine intermédiaire pour conclure. Une fois $\delta^\star$ obtenue, il calcule le vecteur $\boldsymbol{b}$ correspondant dans WOTS+ en suivant l'algorithme de signature donné en I.1.2. Il repère alors tous les indices i tels que $b_i = 0$ et récupère $\sigma_{d-1,i}$ qui vaut $\check{s}_i$ en vertu de l'algorithme de signature de WOTS+. S'il n'a pas collecté tous les $\check{s}_1, \ldots, \check{s}_\ell$, il conserve ceux qu'il a déjà obtenus et recommence : il redemande une signature de m, la faute (les fautes doivent être différentes les unes des autres), en déduit une nouvelle racine fautée, et ainsi de suite. L'algorithme 1 résume l'attaque.

Maintenant que l'attaquant sait retrouver une clef privée WOTS+ de l'étage $d-1$, il n'est pas difficile d'en retrouver *toutes* les clefs privées. En effet, d'une part, les $2^{h/d} = 32$ clefs ont toutes la même probabilité d'intervenir dans la signature d'un message m pris au hasard ; d'autre part, la signature de m contient, à travers la chaîne $i||j$, le numéro

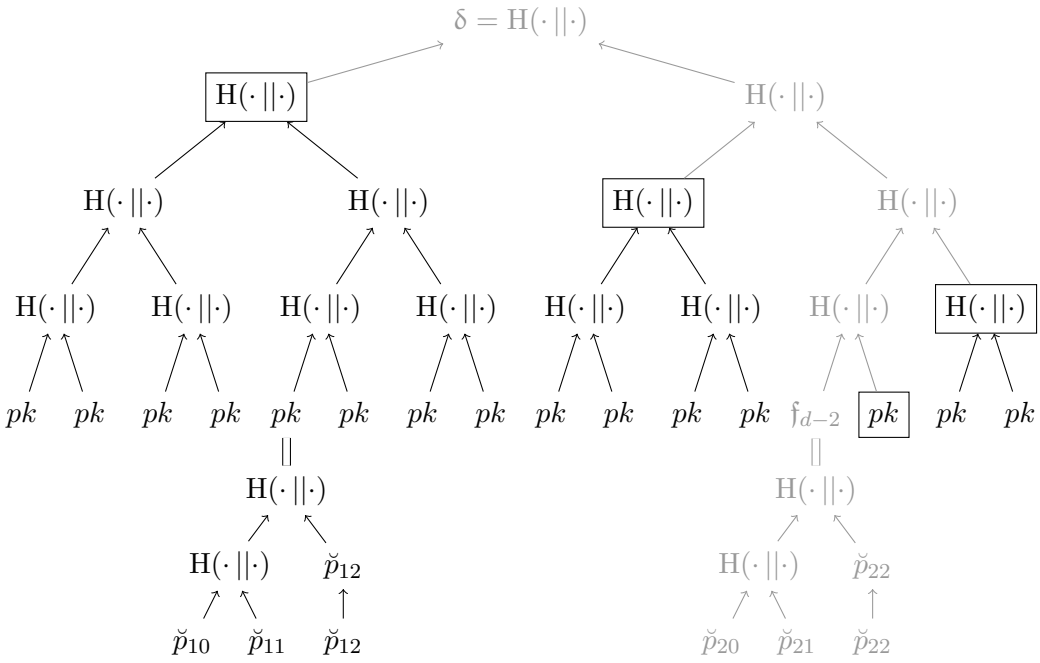

FIGURE I.7 – Localisations possibles de la faute dans le cadre de la stratégie 1. La figure représente l'arbre de Merkle dont δ est la racine, ainsi que deux des arbres de compression de ses feuilles. Les nœuds constituant $\mathrm{A}(\mathfrak{f}_{d-2})$ sont encadrés. Les fautes peuvent être provoquées avec succès dans la partie noire, mais pas dans la partie grise. On a omis les masques pour la lisibilité.

de la clef utilisée dans la signature. Ainsi, l'attaquant peut savoir, après avoir demandé une signature, quelle est la clef qu'il s'apprête à découvrir. S'il la détient déjà, il n'a qu'à sélectionner un autre message aléatoire et réessayer. L'algorithme 2 résume l'attaque.

Remarquons pour finir que les signatures fautées produites par cette attaque sont valides. En effet, $\boldsymbol{\sigma}^\star_{d-1}$ est la signature valide de $\delta^\star$, qui est calculée à partir de $\mathrm{A}(\mathfrak{f}_{d-2})^\star$, qui est donné dans la signature totale $\boldsymbol{\sigma}^\star$. Par ailleurs, tous les autres éléments de $\boldsymbol{\sigma}^\star$ sont corrects. Par suite, l'algorithme de vérification accepte $\boldsymbol{\sigma}^\star$.

Il s'agit à présent de déterminer le nombre de signatures fautées nécessaires pour retrouver une clef privée WOTS+ entière.

I.2.2 Nombre de signatures requis

Dans le but de calculer le nombre moyen de signatures que nécessite cette attaque, on fait l'hypothèse que la fonction H se comporte comme un oracle aléatoire. Cette hypothèse a pour conséquence immédiate que, lorsqu'on faute le calcul de δ, $\delta^\star$ prend une valeur uniformément aléatoire entre 0 et $2^\lambda - 1$. Par suite, chacun des ℓ_1 morceaux de $\log w$ bits de $\delta^\star$ peut être vu comme une variable aléatoire suivant la loi uniforme sur $[\![0, w-1]\!]$. Chaque b_i pour $1 \leqslant i \leqslant \ell_1$ vaut donc 0 avec probabilité $\frac{1}{w}$.

La somme de contrôle κ suit alors la loi d'une somme de $\ell_1 = 64$ variables aléatoires suivant la loi uniforme sur $[\![0, w-1]\!]$, qu'on approchera, grâce au théorème central limite,

Entrées : un message m quelconque.
Sorties : l'une des clefs privées WOTS+ de l'étage $d-1$ du super-arbre.

1 $\boldsymbol{sk} \leftarrow [\ell$ coordonnées à $0]$
2 partiesSKRetrouvées $\leftarrow 0$
3 $\boldsymbol{\sigma} = (id, \ldots, \mathrm{A}(\mathfrak{f}_{d-2}), \boldsymbol{\sigma}_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1})) \leftarrow \mathsf{SignatureNormale}(m)$
4 $\mathfrak{A} \leftarrow \mathrm{A}(\mathfrak{f}_{d-2})$ `/* on conserve ce chemin d'authentification en mémoire pour pouvoir le comparer aux fautés qu'on obtiendra après */`
5 **Répéter**
6 Exécuter l'algorithme de vérification sur $\boldsymbol{\sigma}$ pour obtenir δ
7 De δ déduire le vecteur $\boldsymbol{b}$ calculé par l'algorithme de signature de WOTS+
8 **Pour** $1 \leqslant i \leqslant \ell$ **faire**
9 **Si** $b_i = 0$ **alors**
10 $\boldsymbol{sk}[i] \leftarrow \sigma_{d-1,i}$
11 partiesSKRetrouvées $\leftarrow$ partiesSKRetrouvées $+1$
12 **FinSi**
13 **FinPour**
14 **Si** partiesSKRetrouvées $= \ell$ **alors**
15 Sortir de la boucle
16 **FinSi**
17 **Répéter**
18 $\boldsymbol{\sigma}^\star = (id, \ldots, \mathrm{A}(\mathfrak{f}_{d-2})^\star, \boldsymbol{\sigma}^\star_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1})) \leftarrow \mathsf{SignatureFautée}(m)$
19 **TantQue** $\mathrm{A}(\mathfrak{f}_{d-2})^\star = \mathfrak{A}$
20 $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma}^\star$
21 **TantQue** *vrai*
22 **Renvoyer** $(id, \boldsymbol{sk})$ `/* de` id `on déduit la position dans son arbre de la clef visée */`

Algorithme 1 : attaque avec recalcul de la racine fautée permettant de trouver une clef privée WOTS+ de l'étage $d-1$.

Entrées : néant
Sorties : toutes les clefs privées WOTS+ de l'étage $d-1$ du super-arbre

1 listeDesNuméros $\leftarrow$ [$2^{h/d}$ coordonnées à 0]
2 listeDesSK $\leftarrow$ [$2^{h/d}$ coordonnées à []]
3 **Pour** $0 \leqslant j < 2^{h/d}$ **faire**
4 | **Répéter**
5 | | $m \overset{\mathrm{R}}{\leftarrow} \mathcal{U}(\{0,1\}^\lambda)$ `/* une chaîne aléatoire de λ bits */`
6 | | $\boldsymbol{\sigma} = (id, \ldots, \mathrm{A}(\mathfrak{f}_{d-2}), \boldsymbol{\sigma}_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1})) \leftarrow \mathsf{SignatureNormale}(m)$
7 | | $n \leftarrow$ les h/d bits de poids fort de id `/* c'est ainsi qu'on déduit du numéro de la feuille utilisée à l'étage 0 celui de la feuille utilisée à l'étage` $d-1$`; cf [BHH+14] */`
8 | **TantQue** $n \in$ listeDesNuméros
9 | $(id, \boldsymbol{sk}) \leftarrow \mathsf{Algorithme1}(m)$
10 | $n \leftarrow$ les h/d bits de poids fort de id
11 | listeDesSK$[n] \leftarrow \boldsymbol{sk}$
12 **FinPour**
13 **Renvoyer** listeDesSK

Algorithme 2 : attaque avec recalcul de la racine fautée permettant de trouver toutes les clefs privées WOTS+ de l'étage $d-1$.

par une loi normale. Les paramètres de cette loi normale sont :

$$\mu = \ell_1 \frac{w-1}{2}, \sigma^2 = \ell_1 \frac{w^2-1}{12}.$$

Posons qu'on exprime κ en base w selon la convention *big-endian*. $b_i = 0$ pour $\ell_1 < i \leqslant \ell_2$ signifie alors que $\kappa \mod w^j < w^{j-1}$ avec $j = i - \ell_1$. Cet évènement a pour probabilité :

$$\begin{aligned}\mathbb{P}(\kappa \mod w^j < w^{j-1}) &= \sum_{q=0}^{\lfloor \ell_1(w-1)/w^j \rfloor} \mathbb{P}(qw^j \leqslant \kappa < qw^j + w^{j-1}) \\ &= \sum_{q=0}^{\lfloor \ell_1(w-1)/w^j \rfloor} \sum_{z=qw^j}^{qw^j+w^{j-1}-1} \mathbb{P}(\kappa = z)\end{aligned}$$

avec :

$$\mathbb{P}(\kappa = z) = \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right) \Big/ \sum_{n\in\mathbb{Z}} \exp\left(-\frac{(n-\mu)^2}{2\sigma^2}\right).$$

On obtient (on rappelle qu'on a alors $\ell_1 = 64$ et $w = 16$) : $\mathbb{P}(b_{65} = 0) \approx 1/w, \mathbb{P}(b_{66} = 0) \approx 0{,}098$ et $\mathbb{P}(b_{67} = 0) \approx 2^{-30{,}7}$. Cette dernière valeur est intéressante. Elle a deux conséquences immédiates : la première, c'est qu'il faut demander en moyenne $2^{30{,}7}$ signatures fautées pour retrouver $\check{s}_{67}$, ce qui est beaucoup ; la seconde, c'est qu'on n'a besoin de $\check{s}_{67}$ pour signer une racine qu'avec probabilité $2^{-30{,}7}$. Donc en ce qui concerne $\check{s}_{67}$, on peut se contenter de ne retrouver que $\mathrm{H}(\check{s}_{67} \oplus r_1)$ — noter qu'alors l'algorithme 1 doit être adapté. Compte tenu de la très forte probabilité de retrouver cette valeur ($\mathbb{P}(b_{67} = 1) \approx 0{,}80$), on supposera dans la suite que le nombre de signatures moyen qu'on calculera pour retrouver $\check{s}_1, \ldots, \check{s}_{66}$ est suffisant pour trouver $\mathrm{H}(\check{s}_{67} \oplus r_1)$ avec probabilité très proche de 1. Nous verrons que cette supposition est vérifiée en pratique.

Pour finir, on s'appuie sur les valeurs de $\mathbb{P}(b_{65} = 0)$ et $\mathbb{P}(b_{66} = 0)$ pour justifier cette approximation qu'on fera dans la suite : on assimilera $b_1, b_2, \ldots, b_{66}$ à 66 variables aléatoires indépendantes suivant la loi uniforme sur $[\![0, w-1]\!]$.

Intéressons-nous maintenant au nombre de signatures requis en moyenne pour mener à bien l'attaque. Soit X la variable aléatoire qui prend comme valeur le nombre de signatures demandées pour retrouver $\boldsymbol{sk}$ (sauf $\check{s}_{67}$). Notre problème se ramène alors à calculer $\mathbb{E}(\mathrm{X})$.

Soit $\{\boldsymbol{\sigma}^{(1)\star}, \boldsymbol{\sigma}^{(2)\star}, \ldots, \boldsymbol{\sigma}^{(n)\star}\}$ l'ensemble des n signatures fautées que l'attaquant a collectées. Soit encore :

$$\mathrm{V}_j^{(n)} := \left\{ \sigma_{d-1,j}^{(i)\star} \mid 1 \leqslant i \leqslant n \right\},$$

c'est-à-dire l'ensemble des valeurs prises par la j^{e} coordonnée des $\boldsymbol{\sigma}_{d-1}^{(i)\star}$ récupérées. On définit ensuite l'évènement $\mathrm{B}_n := \{\exists 1 \leqslant j < \ell \text{ tq } \check{s}_j \notin \mathrm{V}_j^{(n)}\}$.

$\mathrm{X} = n$ signifie qu'il existe $1 \leqslant j < \ell$ tel que $\check{s}_j \notin \mathrm{V}_j^{(n-1)}$ mais que $\check{s}_j \in \mathrm{V}_j^{(n)}$ pour tout $1 \leqslant j < \ell$. En d'autre termes, l'évènement $\{\mathrm{X} = n\}$ est égal à l'évènement $\mathrm{B}_{n-1} \cap \overline{\mathrm{B}_n}$. Par suite :

$$\begin{aligned}\mathbb{P}(\mathrm{X} = n) &= \mathbb{P}(\mathrm{B}_{n-1} \cap \overline{\mathrm{B}_n}) \\ &= \mathbb{P}(\mathrm{B}_{n-1}) + \mathbb{P}(\overline{\mathrm{B}_n}) - \mathbb{P}(\mathrm{B}_{n-1} \cup \overline{\mathrm{B}_n}).\end{aligned}$$

Remarquons maintenant que B_n implique B_{n-1}, et que $\overline{\mathrm{B}_{n-1}}$ implique $\overline{\mathrm{B}_n}$. Il s'ensuit que $\mathbb{P}(\mathrm{B}_{n-1} \cup \overline{\mathrm{B}_n}) = 1$, donc que :

$$\mathbb{P}(\mathrm{X} = n) = \mathbb{P}(\overline{\mathrm{B}_n}) - \mathbb{P}(\overline{\mathrm{B}_{n-1}}).$$

Reste donc à calculer $\mathbb{P}(\overline{\mathrm{B}_n})$. Puisque les coordonnées des $\sigma_{d-1}^{\star}$ sont deux-à-deux indépendantes et de même loi uniforme sur $[\![0, w-1]\!]$ par hypothèse, on a :

$$\begin{aligned}\mathbb{P}(\overline{\mathrm{B}_n}) &= \prod_{j=1}^{\ell-1} \mathbb{P}\left(\check{s}_j \in \mathrm{V}_j^{(n)}\right) = \mathbb{P}\left(\check{s}_1 \in \mathrm{V}_1^{(n)}\right)^{\ell-1} = \left(1 - \mathbb{P}\left(\check{s}_1 \notin \mathrm{V}_1^{(n)}\right)\right)^{\ell-1} \\ &= \left(1 - \left(\frac{w-1}{w}\right)^n\right)^{\ell-1}.\end{aligned}$$

On obtient donc finalement :

$$\mathbb{P}(\mathrm{X} = n) = \left(1 - \left(\frac{w-1}{w}\right)^n\right)^{\ell-1} - \left(1 - \left(\frac{w-1}{w}\right)^{n-1}\right)^{\ell-1}$$

Les paramètres de SPHINCS-256 ($w = 16$, $\ell = 67$) conduisent à $\mathbb{E}(\mathrm{X}) \approx 74{,}5$. Remarquons que c'est largement suffisant pour trouver $\mathrm{H}(\check{s}_{67} \oplus r_1)$ (la probabilité de retrouver ce haché au bout de 74,5 signatures est strictement supérieure à $1 - 2^{-172}$).

Une clef privée WOTS+ de SPHINCS-256 peut donc être retrouvée en demandant 74,5 signatures fautées en moyenne.

I.2.3 Seconde stratégie

Description de l'attaque

Une autre stratégie ne nécessitant pas de connaître la valeur de $\delta^\star$, donc sans restriction sur la faute infligée si l'on s'autorise à recevoir des signatures invalides, peut être

envisagée. Elle consiste à réunir, pour chaque coordonnée $\sigma^\star_{d-1,j}$ de $\boldsymbol{\sigma}^\star_{d-1}$, les w valeurs $c_0(\check{s}_i, \boldsymbol{r}), \ldots, c_{w-1}(\check{s}_i, \boldsymbol{r})$ qu'elle peut prendre, afin d'en déduire la valeur de $\check{s}_i$.

Pour cela, on commence par préparer ℓ ensembles vides $\mathrm{V}_1, \ldots, \mathrm{V}_\ell$. Ensuite, on demande et faute la signature d'un message m quelconque. On récupère la signature $\boldsymbol{\sigma}^\star_{d-1}$ qu'elle contient et, remarquant que, par définition, $\sigma_{d-1,i} = c_k(\check{s}_i, \boldsymbol{r})$ pour un certain k inconnu, on place $\sigma_{d-1,i}$ dans V_i, et cela pour tout $1 \leqslant i \leqslant \ell$. S'il existe un ensemble V_j de cardinal plus petit que w, on recommence : on demande et faute (les fautes doivent être différentes les unes des autres) la signature du même message m, on récupère la $\boldsymbol{\sigma}^\star_{d-1}$ qu'elle contient, et ainsi de suite.

Arrivé à ce point, il nous faut distinguer, au sein de chaque ensemble V_i, le très convoité $c_0(\check{s}_i, \boldsymbol{r}) = \check{s}_i$ des autres $c_j(\check{s}_i, \boldsymbol{r})$. Il suffit de remarquer pour cela que $\check{s}_i$ est le seul élément de V_i à vérifier que $\mathrm{H}(\check{s}_i \oplus r_1) \in \mathrm{V}_i$, puisque, par définition, $\mathrm{H}(\check{s}_i \oplus r_1) = c_1(\check{s}_i, \boldsymbol{r})$ qui se trouve bien lui aussi dans V_i. Donc, pour dégager $\check{s}_i$ des autres éléments de V_i, on détermine si un élément $v \in \mathrm{V}_i$ est tel que $\mathrm{H}(v \oplus r_1) \in \mathrm{V}_i$: si c'est le cas, alors $v = \check{s}_i$, sinon, on prend un autre $v \in \mathrm{V}_i$ et on fait passer le test à ce nouveau v. Après avoir fait cela pour tous les ensembles V_i, on possède bien toute la clef privée WOTS+.

L'attaque est détaillée dans l'algorithme 3.

Entrées : un message m quelconque.
Sorties : l'une des clefs privées WOTS+ de l'étage $d-1$ du super-arbre.

$sk \leftarrow [\ell$ coordonnées à $0]$
Pour $1 \leqslant j \leqslant \ell$ **faire**
  $\mathrm{V}_j \leftarrow []$
FinPour
Répéter
  $\sigma^\star = (id, \ldots, \mathrm{A}(\mathfrak{f}_{d-2})^\star, \sigma^\star_{d-1}, \mathrm{A}(\mathfrak{f}_{d-1})) \leftarrow \mathsf{SignatureFautée}(m)$
  Pour $1 \leqslant j \leqslant \ell$ **faire**
    Si $\sigma^\star_{d-1,j} \notin \mathrm{V}_j$ **alors**
      $\mathrm{V}_j \leftarrow \mathrm{V}_j \cup [\sigma^\star_{d-1,j}]$
    FinSi
  FinPour
TantQue $\exists j \in [\![1, \ell]\!]$ *tq* $\mathrm{card}(\mathrm{V}_j) \neq w$
Pour $1 \leqslant j \leqslant \ell$ **faire**
  Pour $1 \leqslant i \leqslant w$ **faire**
    Si $\mathrm{H}(\mathrm{V}_j[i] \oplus r_1) \in \mathrm{V}_j$ **alors**
      `/* `r_1`: premier masque pour WOTS+, cf I.1.2 */`
      $sk[j] \leftarrow \mathrm{V}_j[i]$
      Sortir de la boucle
    FinSi
  FinPour
FinPour
Renvoyer (id, sk) `/* de` id `on déduit la position dans son arbre de la clef visée */`

Algorithme 3 : seconde stratégie d'attaque.

Bien sûr, l'algorithme 3 peut être facilement étendu (sur le modèle de l'algorithme 2 par exemple) pour trouver toutes les clefs privées WOTS+ de l'étage $d-1$.

Toutefois, pour pouvoir réaliser l'attaque, il faut que $b_i = 0$ ou $b_i = 1$ raisonnablement souvent pour tout $1 \leqslant i < \ell$; en effet, vu la faible probabilité que $b_\ell = 0$, la condition devient pour $i = \ell$: $b_i = 1$ ou $b_i = 2$ raisonnablement souvent — noter que l'algorithme 3 doit être adapté pour en tenir compte. On a déjà vu (cf I.2.2) que cette condition était satisfaite pour $1 \leqslant i \leqslant \ell_1$. Un calcul similaire montre que $\mathbb{P}(b_{65} = 1) \approx 1/w$, $\mathbb{P}(b_{66} = 1) \approx 0{,}056$ et $\mathbb{P}(b_{67} = 2) \approx 0{,}20$. Par suite, la condition que nous venons d'énoncer est vraie.

Nombre de signatures requis

On reconduit l'hypothèse qu'on a déjà faite en I.2.2 selon laquelle $b_1, \ldots, b_{66}$ se comportent comme 66 variables aléatoires suivant la loi uniforme sur $[\![0, w-1]\!]$. On suppose de plus, comme en I.2.2, que le nombre de signatures requis pour trouver $\breve{s}_1, \ldots, \breve{s}_{66}$ est suffisant pour trouver $\mathrm{H}(\breve{s}_{67} \oplus r_1)$ avec une probabilité très proche de 1.

Le nombre de signatures moyen que requiert cette stratégie s'élève à $\mathbb{E}(\mathrm{Y})$ où Y est la variable aléatoire qui prend pour valeur le nombre de signatures fautées qu'il faut demander pour disposer, pour chaque $\sigma^\star_{d-1,j}$, $1 \leqslant j < \ell$, des w valeurs qu'elle peut prendre.

En reprenant les notations de la section I.2.2, Y est donc la variable aléatoire qui prend comme valeur le plus petit entier n tel que $\operatorname{card} \mathrm{V}_j^{(n)} = w$ pour tout $1 \leqslant j < \ell$. Soit l'évènement $\mathrm{K}_n := \{\forall\, 1 \leqslant j < \ell, \operatorname{card} \mathrm{V}_j^{(n)} = w\}$. Un raisonnement similaire à celui qu'on a suivi en I.2.2 pour calculer $\mathbb{P}(\mathrm{X} = n)$ conduit à :

$$\begin{aligned}\mathbb{P}(\mathrm{Y} = n) &= \mathbb{P}(\mathrm{K}_n \cap \overline{\mathrm{K}_{n-1}}) \\ &= \mathbb{P}(\mathrm{K}_n) + \mathbb{P}(\overline{\mathrm{K}_{n-1}}) - \mathbb{P}(\mathrm{K}_n \cup \overline{\mathrm{K}_{n-1}}) \\ &= \mathbb{P}(\mathrm{K}_n) - \mathbb{P}(\mathrm{K}_{n-1}).\end{aligned}$$

De notre hypothèse sur les b_i s'ensuit que :

$$\mathbb{P}(\mathrm{K}_n) = \prod_{j=1}^{\ell-1} \mathbb{P}\left(\operatorname{card} \mathrm{V}_j^{(n)} = w\right) = \mathbb{P}\left(\operatorname{card} \mathrm{V}_1^{(n)} = w\right)^{\ell-1}.$$

Cette dernière probabilité peut être calculée par dénombrement des listes, de longueur $n \geqslant w$, contenant exactement $j \leqslant w$ éléments distincts pris parmi w : il y en a C_w^j fois le nombre de listes de longueur n contenant les j éléments d'un ensemble de cardinal j. Ce dernier nombre vaut $\gamma_{j,n}$ défini par récurrence ainsi pour tous entiers $n \geqslant j \geqslant 2$:

$$\gamma_{j,n} = j^n - \sum_{k=1}^{j-1} \mathrm{C}_j^k \gamma_{k,n},\ \gamma_{1,n} = 1$$

c'est-à-dire le nombre de listes de longueur n à valeurs dans un ensemble de cardinal j, moins le nombre de listes de longueur n contenant un élément de cet ensemble, moins le nombre de listes de longueur n contenant deux éléments distincts de cet ensemble, et ainsi de suite. Pour contrôler ce résultat, on peut vérifier que $\sum_{j=1}^{w} \mathrm{C}_w^j \gamma_{j,n} = w^n$, c'est-à-dire le nombre de listes de longueur n issues d'un ensemble de cardinal w. On en conclut que :

$$\mathbb{P}\left(\operatorname{card} \mathrm{V}_1^{(n)} = w\right) = \begin{cases} 0 & \text{si } n < w\ ; \\ \frac{1}{w^n}\gamma_{w,n} & \text{sinon}\ ; \end{cases}$$

donc que, si $n \geqslant w$:

$$\mathbb{P}(\mathrm{Y} = n) = \left(\frac{1}{w^n}\gamma_{w,n}\right)^{\ell-1} - \left(\frac{1}{w^{n-1}}\gamma_{w,n-1}\right)^{\ell-1}$$

et $\mathbb{P}(\mathrm{Y} = n) = 0$ si $n < w$. On en déduit, avec les paramètres pour SPHINCS-256, que $\mathbb{E}(\mathrm{Y}) \approx 117{,}3$. Remarquons que cela conduit à une probabilité de retrouver $\mathrm{H}(\breve{s}_{67} \oplus r_1)$ strictement supérieure à $1 - 2^{-37}$.

Utiliser cette seconde stratégie coûte donc plus cher en signatures que la première, mais, d'une part, elle n'impose pas de contrainte sur la localisation de la faute, d'autre part, elle requiert relativement peu de calculs après la récupération des signatures, alors que l'autre nécessite une vérification par signature demandée ainsi que la recomposition du vecteur $\boldsymbol{b}$ associé à $\delta^\star$ (cf I.2.1). Ceci dit, le coût de ces opérations est nettement moins élevé que celui d'une signature (35 fois moins pour la vérification [BHH+15]).

Noter de plus qu'en pratique, obtenir 117,3 signatures en moyenne demande des précautions de réalisation, car notre hypothèse selon laquelle b_{66} se comporte comme une variable aléatoire uniforme est trop forte (en effet, $\mathbb{P}(4 \leqslant b_{66} \leqslant 7) \approx 0{,}01$). Suivre naïvement l'algorithme 3 peut donc faire inutilement grimper ce nombre à plusieurs centaines. Nous avons procédé autrement dans nos simulations pour nous limiter à 117,1 signatures en moyenne (voir la section I.2.4).

I.2.4 Simulations de l'attaque

Nous avons réalisé des simulations des deux stratégies exposées dans les sections I.2.1 et I.2.3 afin de corroborer nos résultats. Nous nous sommes appuyés sur l'implémentation de SPHINCS [BHH+14] écrite par les auteurs du schéma. Les fichiers sources sont livrés avec la version électronique du rapport et se trouvent dans le dossier `implems/sphincs`.

L'attaque a été simulée de la façon suivante. Nous avons modifié la fonction de signature (`crypto_sign_sphincs`) pour qu'elle prenne en entrée un booléen devant valoir 1 pour simuler une faute. Si ce booléen vaut 1 et si l'algorithme de signature s'apprête à construire l'étage $d-2$ du super-arbre, alors `crypto_sign_sphincs` demande à la fonction chargée de calculer les chemins d'authentification (`compute_authpath_wots`) de fauter le prochain. Cette fonction va alors sélectionner au hasard une feuille de l'arbre de Merkle parmi les trente-et-une possibles — car nous n'avons implémenté qu'un modèle de faute pour les deux stratégies, c'est donc celui de la stratégie 1 (section I.2.1 ou figure I.7), plus contraignant, que nous avons mis en œuvre. Elle va ensuite demander à la fonction chargée de compresser les clefs publiques WOTS+ (`l_tree`) de fauter le calcul de la feuille sélectionnée. Pour faire cela, `l_tree` choisit au hasard un appel à la fonction de hachage et modifie au hasard un octet aléatoire de l'entrée de la fonction de hachage. C'est ainsi que la faute est produite.

Noter que, pour simplifier l'implémentation, `compute_authpath_wots` ne faute pas les nœuds de l'arbre de Merkle lui-même, mais le résultat serait identique.

Pour l'implémentation des attaques elles-mêmes, nous avons suivi les algorithmes 1 et 3 en les adaptant au fait que nous ne cherchions pas $\breve{s}_{67}$ mais $\mathrm{H}(\breve{s}_{67} \oplus r_1)$. De plus, dans la cadre de la stratégie 2, on cesse de demander des signatures fautées dès lors qu'on a retrouvé $\breve{s}_1, \ldots, \breve{s}_{65}$, supposant que cela est suffisant pour obtenir au moins une fois $b_{66} = 0$ et $b_{66} = 1$. Les essais que nous avons menés ont révélé, d'une part, que le nombre de signatures demandées moyen était proche de 117,1 (ce qui est bien le nombre attendu

lorsqu'on remplace, dans la formule pour $\mathbb{P}(\mathrm{Y} = n)$, $\ell - 1$ par $\ell - 2$), d'autre part que le taux d'échec est très faible puisque nous n'avons échoué que 12 fois à deviner le $\check{s}_{66}$ correct sur les 7 000 essais effectués, ce qui est en ligne avec le taux d'échec théorique qui vaut environ $\mathbb{P}(b_{66} \neq 0)^{117} + \mathbb{P}(b_{66} \neq 1)^{117} - \mathbb{P}(\{b_{66} \neq 0\} \cap \{b_{66} \neq 1\})^{117} \approx 0{,}001$.

Notre machine de test fonctionnait sous Ubuntu 16.04 LTS et disposait d'un processeur Inter Core i3-6100 CPU à 3,7 GHz et de 8 Go de RAM. Sous cette configuration, la stratégie 1 nous a livré une clef privée WOTS+ en environ six secondes et la stratégie 2 appliquée comme nous l'avons décrit ci-dessus en environ dix secondes. Noter qu'aucune des deux attaques ne requiert plus de 120 ko de mémoire.

Nous avons également mis en œuvre l'algorithme 2 qui nous a permis de retrouver toutes les clefs privées WOTS+ de l'étage $d - 1$ du super-arbre visé — qui était différent à chaque essai, puisque nous la clef privée générale $(\check{\mathrm{S}}_1, \check{\mathrm{S}}_2)$ était fixée aléatoirement à chaque fois. Nous sommes ainsi parvenu à forger des signatures valides pour des messages arbitraires : d'abord, nous avons lancé `attack`, qui, en appliquant l'algorithme 2, produit en un peu plus de trois minutes un fichier appelé `wots_private_key` qui contient les 32 clefs privées WOTS+ de l'étage $d - 1$ d'un super-arbre obtenues grâce à notre attaque ; ensuite, nous avons écrit un message dans le fichier source du programme de falsification, `forge.c`, puis avons lancé ce programme, qui forge la signature du message donné et affiche à l'écran la sortie de l'algorithme de *vérification*. L'exécution de `forge.c` nécessite moins d'une seconde sur notre machine de test.

I.2.5 Remarques sur la portée de l'attaque

Nous avons eu l'occasion d'échanger sur cette attaque avec Andreas Hülsing et Tanja Lange, deux des coauteurs de l'article fondateur de SPHINCS [BHH+15]. Ils ont mis en lumière les deux points dont nous allons discuter maintenant et les en remercions vivement.

En introduction de cette section nous annoncions que l'attaquant capable de récupérer toutes les clefs privées de l'étage $d-1$ avait le pouvoir de remplacer le super-arbre légitime par le sien propre, donc de falsifier la signature de n'importe quel message. En fait, la falsification d'un message aléatoire ne nécessite pas que l'adversaire dispose de tout l'arbre, mais seulement de l'une des branches. En effet, la chaîne $i||j$ qui détermine le numéro de la feuille signataire du super-arbre ne fait l'objet d'aucune négociation ni d'aucun contrôle de la part du vérificateur de la signature ; elle est entièrement calculée par le signataire (voir I.1.2). Le signataire malhonnête a donc toute liberté pour choisir la feuille du super-arbre qui lui convient, en particulier une du sous-arbre qu'il maîtrise.

De surcroît, il est possible de s'assurer la maîtrise d'un sous-arbre grâce à une unique faute. Ceci est la conséquence de l'analyse menée par L. Bruinderink et A. Hülsing [BH16] de la résistance effective de WOTS+ à une attaque à deux messages aléatoires. Il en ressort qu'avec les paramètres choisis dans SPHINCS, un attaquant qui possède les signatures de deux messages différents réalisées par la même clef privée a probabilité supérieure à 1/2 de trouver, après 2^{34} essais, un message pour lequel il sache falsifier la signature. Cela signifie que, connaissant la signature WOTS+ légitime d'une racine δ et la signature de $\delta^\star$ — dans le cadre de la stratégie 1, car il faut connaître le vecteur $\boldsymbol{b}$ découlant de $\delta^\star$ pour pouvoir réaliser la falsification envisagée dans [BH16] —, un attaquant n'a qu'à essayer en moyenne 2^{35} arbres de Merkle avant d'en trouver un qu'il sache authentifier. Certes, d'un point de vue pratique, cela fait beaucoup eu égard au nombre de hachés que requiert le calcul d'un seul arbre (près de $2^{15,1}$, sans compter le coût du pseudo-aléa), mais toutes ces opérations sont facilement parallélisables — à titre de comparaison, la collision sur SHA-1

publiée par M. Stevens et coll. [SBK+17] a nécessité le calcul de près de 2^{63} hachés. Aussi cette attaque en une seule faute doit-elle être considérée, même si, ayant été suggérée très tardivement, nous n'avons pas pu la tester.

Ces deux remarques combinées réduisent à seulement 1 le nombre de fautes suffisant pour assurer à l'attaquant la possibilité de signer un message arbitraire, au lieu des quelques 2 400 que requiert l'algorithme 2. Aussi l'attaque peut-elle être menée à la fois par un adversaire disposant de peu de moyens pour provoquer des fautes mais d'une bonne puissance de calcul, et par un adversaire disposant des facultés inverses.

I.2.6 Contremesures

Les mesures envisageables pour protéger SPHINCS de cette attaque sont de deux natures : matérielles, c'est-à-dire qu'elles sont appliquées au composant qui effectue la signature, et logicielles, celles qui nécessitent qu'on modifie l'implémentation du schéma. [MKAA16] propose une contremesure matérielle concernant les arbres de hachage : elle consiste à permuter deux paires de nœuds (les nœuds « fils uniques » étant ignorés) avant de calculer chacune leur nœud père. Afin de diminuer le surcoût en temps, [MKAA16] suggère de diviser le circuit en deux parties, dans l'idée de pouvoir envoyer dans la première partie du circuit l'entrée $n + 1$ tandis que la seconde partie du circuit finit de calculer la sortie pour l'entrée n. Cette suggestion implique toutefois un surcoût en espace.

En termes logiciels, la principale contremesure qu'on propose reste classiquement de calculer deux fois la signature et de comparer les résultats avant de fournir un résultat : si les deux signatures sont identiques, on la renvoie, sinon on renvoie une erreur. Cette contremesure est efficace contre les attaquants qui n'ont pas la capacité de provoquer deux fautes aux effets identiques. Elle a néanmoins deux inconvénients immédiats : elle double le temps de signature et nécessite de conserver en mémoire un exemplaire de la signature. Celle-ci occupant 41 ko pour SPHINCS-256, c'est inenvisageable pour une puce disposant de ses seules ressources. Il y a toutefois un moyen de réduire ces 41 ko à seulement 32 octets : puisque l'attaque décrite ici cible en premier lieu les racines des arbres de Merkle, on peut, avant d'en signer une, la stocker et la recalculer, interrompant le calcul de la signature dès que deux résultats sensés être identiques sont différents. Cette façon de procéder est suggérée par [HRS16a] qui fragmente le calcul de la signature selon les étages du super-arbre pour pouvoir calculer une signature SPHINCS dans seulement 16 ko de RAM.

Cette solution règle le problème du surcoût en mémoire mais laisse entier celui du surcoût en temps — ou presque, car il est inutile de calculer deux fois la clef publique en fin de signature. Par ailleurs, comme la contremesure physique discutée plus haut, elle complique considérablement l'attaque mais n'assure pas l'immunité de SPHINCS face à un attaquant capable d'injecter deux fois la même faute.

On a cherché d'autres contremesures spécifiques à SPHINCS. Une contremesure qui fonctionne pour protéger les clefs de l'étage $d - 1$ (qu'on visait dans les sections I.2.1 et I.2.3) consiste, lors du calcul de la clef publique de SPHINCS, à y adjoindre les (mettons) 32 bits de poids faible de chacune des racines de l'étage $d-2$. Lors de la signature, le signataire doit vérifier que les bits de poids faible de la racine qu'il signe se trouvent bien dans sa clef publique ; dans le cas contraire, il renvoie une erreur. Ce procédé est presque gratuit lors de la signature et augmente sensiblement la complexité à la fois de l'attaque et de la falsification. Toutefois, cette contremesure, d'une part, n'est que d'une efficacité limitée si l'adversaire peut paralléliser son attaque, d'autre part, ne peut rien pour protéger les clefs d'un étage inférieur au $(d - 1)$-ème.

Une contremesure additionnelle consisterait alors à rendre les racines d'un étage dépendantes de l'étage en-dessous, afin qu'une faute affectant un étage quelconque se propage jusqu'au sommet de l'arbre où elle y serait détectée par ce système. Malheureusement, nous n'y sommes pas parvenu à un coût acceptable.

I.3 Conclusion de la partie

Nous avons présenté la première attaque par injection de faute contre le schéma de signature SPHINCS. Cette attaque requiert peu de signatures fautées et une seule faute par signature pour permettre à l'adversaire de signer n'importe quel message. Elle est par ailleurs discrète, les signatures fautées qu'elle demande peuvant être rendues valides sans contrainte importante. L'attaque peut de surcroît réussir avec une seule signature fautée, en contrepartie d'une puissance de calcul importante mais envisageable.

Elle sera l'objet d'un article en cours de rédaction que nous prévoyons de soumettre à PQCrypto-2018. Nous y traiterons également très probablement de la possibilité d'établir un compromis entre le nombre de fautes à injecter et la quantité de calculs nécessaire pour trouver une racine dont on puisse falsifier la signature WOTS+.

Nous n'avons toutefois pas résolu le problème des contremesures applicables. Établir une contremesure efficace à l'attaque décrite dans toute cette partie reste donc une question ouverte.

Partie II

Implémentation sécurisée d'un générateur pseudo-aléatoire gaussien

L'échantillonnage de lois gaussiennes discrètes est un problème que l'on rencontre dans la cryptographie basée sur les réseaux. Étant donnés une base courte d'un réseau Λ et un élément $c \in \mathbb{R}^{\dim \Lambda}$, le problème consiste à engendrer un élément aléatoire $v \in \Lambda$ selon une distribution gaussienne de centre c sur les éléments de Λ. Utilisé pour la première fois en 2008 par C. Gentry et coll. [GPV08] pour obtenir des schémas de chiffrement basé sur l'identité (IBE) ou des schémas de signature, l'échantillonnage d'une loi normale sur un réseau est devenu une étape essentielle d'un certain nombre de schémas, entre autres de l'IBE proposé par S. Agrawal et coll. [ABB10] ou de la signature de X. Boyen [Boy10].

Cet échantillonnage est réalisé au moyen, principalement, des algorithmes dus à P. Klein [Kle00] — utilisé dans un but cryptographique pour la première fois dans [GPV08] — et à C. Peikert [Pei10]. Or, tous deux nécessitent de savoir échantillonner une loi normale discrète sur $\mathbb{Z}$. Plusieurs propositions ont été faites pour y parvenir, par exemple celles de [BCG+14, DDLL13, DG14, GPV08, Kar16, Pei10] ou la très récente proposition de D. Micciancio et M. Walter [MW17]. Celle-ci semble la plus prometteuse pour deux raisons : en premier lieu, le générateur pseudo-aléatoire (GPA) de [MW17] est au moins aussi rapide que ceux cités précédemment, et, en second lieu, il peut être rendu, avec un faible surcoût, constant en temps, ce qui le protège des attaques par mesure du temps d'exécution.

Toutefois, ce GPA ne fonctionne que si l'écart-type de la loi à échantillonner est supérieur à $\sqrt{2}\eta$, où η est une constante appelée *smoothing parameter* [MR07]. Or, les écarts-types utilisés dans le schéma de signature développé à TCS auquel est destiné ce GPA restent confinés dans l'intervalle $[1, 1{,}37] \cdot \eta$, ce qui est hors de portée du GPA de [MW17].

La seconde partie du stage a donc été consacrée à la construction d'un GPA qui fonctionne pour des écarts-types inférieurs à $\sqrt{2}\eta$ et qui rivalise en vitesse et en sécurité avec celui de [MW17]. Il devait notamment répondre aux contraintes de sécurité suivantes :

- **contrainte M :** son niveau de précision numérique doit être suffisant pour que la loi normale échantillonnée soit indiscernable de la loi normale parfaite ;
- **contrainte TS :** le temps d'exécution du GPA ne doit pas faire fuiter d'information sur les sorties qu'il renvoie ;
- **contrainte TE :** le temps d'exécution du GPA ne doit pas faire fuiter d'information sur les données qu'il prend en entrée, à savoir la moyenne et l'écart-type de la loi

échantillonnée.

Un premier GPA proposé par Thomas Prest et Thomas Ricosset répondait aux contraintes **M** et **TE**, mais pas à la contrainte **TS**. Notre travail a consisté à proposer des solutions pour qu'il fonctionne en temps indépendant de la sortie qu'il renvoie.

Le GPA obtenu à la fin de la partie répond aux trois contraintes fixées ci-dessus et fournit jusqu'à 14,5 millions d'échantillons par seconde, ce qui fait de lui, à notre connaissance, l'un des meilleurs échantillonneurs gaussiens sur $\mathbb{Z}$.

Organisation de la partie. La section II.1 est consacrée à la description du GPA de T. Prest et T. Ricosset. Son implémentation de principe ayant été réalisée sans souci de sécurité physique, on étudie cette question dans la section II.2. On propose ensuite dans la section II.3 une solution pour prévenir les risques de fuites identifiés dans la section précédente et on présente les performances du nouveau GPA. On discute brièvement des risques résiduels avant de conclure ce travail dans la section II.4.

Notations. Dans toute cette partie, on notera :

- pour une fonction f définie sur un ensemble E fini ou dénombrable, $f(\mathrm{E}) = \sum_{x\in\mathrm{E}} f(x)$ si cette somme est absolument convergente ;
- $\rho_{\mu,\sigma}(z) = \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right)$;
- $\mathcal{D}_{\mu,\sigma} = \rho_{\mu,\sigma}/\rho_{\mu,\sigma}(\mathbb{Z})$ la densité de probabilité gaussienne définie sur $\mathbb{Z}$ de moyenne μ et d'écart-type σ [1] ;
- $\mathcal{U}(\mathrm{I})$ pour la densité de la loi uniforme sur l'ensemble I ;
- $\mathcal{B}(p)$ pour la densité de la loi de Bernoulli de paramètre p ;
- $x \xleftarrow{\mathrm{R}} f$ le tirage aléatoire d'un élement x selon la loi de densité f.

Enfin, on confondra parfois, pour la concision du discours, une distribution et sa densité.

II.1 Description algorithmique du GPA

Dans cette section, on décrit, sans preuve de correction, le GPA initialement proposé par Thomas Prest et Thomas Ricosset. Il tire largement parti, pour son fonctionnement, du fait qu'il ait été conçu pour des lois normales d'écart-type inférieur à une borne connue lors de l'implémentation, à savoir, dans le cas d'usage qui a motivé ce travail, $1{,}37\,\eta$. Cette borne est notée σ_0 et a été fixée, pour tous nos tests, à $\sigma_0 = 2$ — on a pris $\eta \approx 1{,}3$, en suivant le raisonnement de [Pre17, section 4.4].

Il est également important de souligner que, pour son implémentation, le GPA repose sur le résultat de T. Prest [Pre17, section 3.3] qui affirme que 37 bits de précision relative sont suffisants pour obtenir une sécurité théorique de 256 bits. Nous utiliserons donc 64 bits pour nous ramener à une précision standard.

Avant de détailler l'algorithme du GPA, on fait un bref rappel sur la méthode d'échantillonnage par rejet, sur laquelle il repose.

1. En fait, pas exactement, mais la différence est négligeable en pratique : cf [MR04, lemme 4.2]

II.1.1 Principe de l'échantillonnage par rejet

La méthode d'échantillonnage par rejet est décrite par J. von Neumann en 1951 [vN51]. Le problème est de simuler une v.a. X (ici donc sur $\mathbb{Z}$) suivant une loi de probabilité de densité f. Supposons qu'il existe une densité de probabilité g telle que, d'une part, on sache simuler une v.a. qui suive la loi de densité g, et telle que, d'autre part, il existe une constante $\kappa \in \mathbb{R}$ telle que $f \leqslant \kappa g$. Dans la suite, par souci de simplicité, on entendra par *distribution d'appui* une telle distribution g.

L'algorithme d'échantillonnage par rejet (algorithme 4) consiste à tirer un élément $z \in \mathbb{Z}$ selon la loi de densité g, puis un élément u selon la loi de densité $\mathcal{U}([0,1])$, et cela tant que $u \geqslant f(z)/(\kappa g(z))$. L'algorithme renvoie la valeur z qui a fait échouer le test.

Cet algorithme renvoie donc $x \in \mathbb{Z}$ avec probabilité $t \cdot \mathbb{P}(z = x) \cdot \mathbb{P}\left(u \leqslant \frac{f(x)}{\kappa g(x)}\right) = t \cdot f(x)/\kappa$ où t est le nombre moyen de tours effectués par l'algorithme. Comme la boucle a probabilité $\sum_{y \in \mathbb{Z}} \mathbb{P}(z = y) \cdot \mathbb{P}\left(u \leqslant \frac{f(y)}{\kappa g(y)}\right) = 1/\kappa$ de s'arrêter à la fin de chaque tour, $t = \kappa$. Par suite, la probabilité que l'algorithme 4 renvoie $x \in \mathbb{Z}$ est $f(x)$.

Entrées : les densités f et g.
Sorties : $x \in \mathbb{Z}$ avec probabilité $f(x)$.

1 **Répéter**
2 $\quad z \xleftarrow{\mathrm{R}} g$
3 $\quad u \xleftarrow{\mathrm{R}} \mathcal{U}([0,1])$
4 **TantQue** $u \geqslant \frac{f(z)}{\kappa g(z)}$
5 **Renvoyer** z

Algorithme 4 : algorithme d'échantillonnage par rejet.

La rapidité d'un algorithme d'échantillonnage par rejet dépend donc du choix de la distribution d'appui, dont la simulation doit être efficace et qui doit en même temps permettre de choisir κ aussi proche de 1 que possible pour réduire la complexité de l'algorithme 4, qui est proportionnelle à κ.

II.1.2 SamplerZ

SamplerZ échantillonne une distribution de densité $\mathcal{D}_{\mu,\sigma}$ avec[2] $\mu \in [0,1[$ et $\sigma \in]0, \sigma_0]$ avec $\sigma_0 \in \mathbb{R}$ arrêté *avant* l'implémentation. L'échantillonnage est effectué par rejet avec une distribution d'appui dont la densité est :

$$\mathcal{G} : z \mapsto \frac{1}{2\rho_{0,\sigma_0}(\mathbb{N})}(\mathbb{1}_{\{z \leqslant 0\}} \cdot \rho_{0,\sigma_0}(z) + \mathbb{1}_{\{z \geqslant 1\}} \cdot \rho_{1,\sigma_0}(z)).$$

Cette distribution a été choisie car elle est triviale à échantillonner dès lors que l'on sait échantillonner la distribution de densité $\mathcal{G}^+ : z \mapsto \frac{1}{\rho_{0,\sigma_0}(\mathbb{N})}\mathbb{1}_{\{z \geqslant 0\}} \cdot \rho_{0,\sigma_0}(z)$.

Échantillonnage de $\mathcal{G}$. Supposons que l'on sache échantillonner $\mathcal{G}^+$. L'échantillonnage de $\mathcal{G}$ peut alors être ensuite réalisé ainsi : soit $x \xleftarrow{\mathrm{R}} \mathcal{G}^+$ et $b \xleftarrow{\mathrm{R}} \mathcal{B}(1/2)$, on calcule $z =$

2. SamplerZ est ensuite trivialement étendu à $\mu \in \mathbb{R}$ en ajoutant à ses sorties la partie entière de μ puisque $\mathcal{D}_{\mu,\sigma}(z) = \mathcal{D}_{\mu+1,\sigma}(z+1)$ pour tout $z \in \mathbb{Z}$.

$b+(2b-1)x$ comme échantillon d'une v.a. suivant la loi de densité $\mathcal{G}$. Ceci peut se déduire du fait que, pour tout $z \in \mathbb{Z}$, on ait $\mathcal{G}(z) = \mathcal{G}(1-z)$.

L'échantillonnage de $\mathcal{G}$ se ramène donc essentiellement à échantillonner $\mathcal{G}^+$. Pour ce faire, l'algorithme utilisé est le CoDFSampler proposé par T. Prest [Pre17], qui utilise non pas la densité $\mathcal{G}^+$ directement mais la densité conditionnelle $\mathrm{CoDF}_{\mathcal{G}^+}$ de la loi de densité $\mathcal{G}^+$:

$$\mathrm{CoDF}_{\mathcal{G}^+}(z) = \mathcal{G}^+(z) / \sum_{k \geqslant z} \mathcal{G}^+(k).$$

CoDFSampler (cf algorithme 5, qui reproduit [Pre17, algorithme 1]) commence par initialiser un compteur z à 0 et par demander u selon la loi uniforme sur $[0,1]$. Il redemande un nouveau u et incrémente z tant que $u \geqslant \mathrm{CoDF}_{\mathcal{G}^+}(z)$. Dès que cette condition n'est plus remplie, il renvoie le compteur z.

Entrées : une table précalculée des $\mathrm{CoDF}_{\mathcal{G}^+}(i)$.
Sorties : $z \in \mathbb{Z}$ avec probabilité $\mathcal{G}^+(z)$.

1 $z \leftarrow 0$
2 $u \xleftarrow{\mathrm{R}} \mathcal{U}([0,1])$
3 **TantQue** $u \geqslant \mathrm{CoDF}_{\mathcal{G}^+}(z)$ **faire**
4 $\quad z \leftarrow z+1$
5 $\quad u \xleftarrow{\mathrm{R}} \mathcal{U}([0,1])$
6 **FinTantQue**
7 **Renvoyer** z

Algorithme 5 : CoDFSampler.

Échantillonnage de $\mathcal{D}_{\mu,\sigma}$. On souhaite réaliser l'échantillonnage de $\mathcal{D}_{\mu,\sigma}$ par rejet en utilisant $\mathcal{G}$ comme distribution d'appui. Notons que $\mathcal{D}_{\mu,\sigma}/\mathcal{G}$ est borné supérieurement sur $\mathbb{Z}$ par $2\rho_{0,\sigma_0}(\mathbb{N})/\rho_{\mu,\sigma}(\mathbb{Z})$. En effet,

$$\frac{\mathcal{D}_{\mu,\sigma}(z)}{\mathcal{G}(z)} = \frac{2\rho_{0,\sigma_0}(\mathbb{N})}{\rho_{\mu,\sigma}(\mathbb{Z})} \times \begin{cases} \exp\left(-\dfrac{(z-\mu)^2}{2\sigma^2} + \dfrac{(z-1)^2}{2\sigma_0^2}\right) & \text{si } z > 0\ ; \\ \exp\left(-\dfrac{(z-\mu)^2}{2\sigma^2} + \dfrac{z^2}{2\sigma_0^2}\right) & \text{si } z \leqslant 0. \end{cases} \tag{II.1}$$

Les arguments des exponentielles sont donc, compte tenu du fait que $0 \leqslant \mu < 1$ et $\sigma \leqslant \sigma_0$, négatifs ou nuls pour tout $z > 0$ et pour tout $z \leqslant 0$ respectivement ; d'où la conclusion. Il s'ensuit que toute valeur $\kappa \geqslant 2\rho_{0,\sigma_0}(\mathbb{N})/\rho_{\mu,\sigma}(\mathbb{Z})$ convient pour le rejet ; en particulier, on peut le choisir égal à cette borne, ce qui nous permet de nous débarrasser des constantes de normalisation lors des calculs de probabilité dont il sera question lors du rejet.

À ce stade, SamplerZ peut être résumé ainsi :

1. $z \leftarrow$ CoDFSampler($\mathcal{G}$) ;
2. $b \xleftarrow{\mathrm{R}} \mathcal{B}(1/2)$; $z \leftarrow b + (2b-1)x$ (jusque là, on a échantillonné $\mathcal{G}$) ;
3. $u \xleftarrow{\mathrm{R}} \mathcal{U}([0,1])$;

4. si $u \geqslant \exp\left(-\frac{(z-\mu)^2}{2\sigma^2} + \frac{(z-b)^2}{2\sigma_0^2}\right)$, retourner au 1 (étape de rejet).

Cet algorithme est toutefois handicapé par l'exponentielle dont le calcul n'est pas souhaitable en environnement contraint [DG14, section 4.1]. On a donc recours à un autre algorithme, BerExp (algorithme 7), qui prend en entrée $x \geqslant 0$ et échantillonne $\mathcal{B}(\mathrm{e}^{-x})$. BerExp fait lui-même appel à SmallBerExp (algorithme 8), qui échantillonne $\mathcal{B}(\mathrm{e}^{-x})$ pour $x < 1$. Finalement, l'algorithme SamplerZ étudié est celui donné par l'algorithme 6.

Remarquons que SamplerZ peut être décomposé en deux parties : une première, *offline*, qui consiste à échantillonner $\mathcal{G}^+$ grâce à CoDFSampler, la seconde, *online* qui consiste à tirer b et à appeler BerExp (cf algorithme 7). Mais $\mathcal{G}^+$ étant de moyenne et d'écart-type fixés, elle ne dépend pas des entrées de SamplerZ ; elle peut donc être échantillonnée *avant* d'appeler SamplerZ, ce qui en améliore la vitesse en contrepartie d'un peu de mémoire. Nous nous servirons de ce fait de façon cruciale lorsque nous chercherons à rendre le temps d'exécution de SamplerZ indépendant de sa valeur de sortie dans le section II.3.

Entrées : une moyenne $\mu \in [0, 1[$ et un écart-type $\sigma \leqslant \sigma_0$.
Sorties : $z \in \mathbb{Z}$ avec probabilité $\mathcal{D}_{\mu,\sigma}(z)$.

1 $b \leftarrow 0$
2 **TantQue** $b = 0$ **faire**
3 $\quad z \leftarrow \mathsf{CoDFSampler}(\mathcal{G})$
4 $\quad b \xleftarrow{\mathrm{R}} \mathcal{B}(1/2)$
5 $\quad z \leftarrow z + b - 2z(1-b)$
6 $\quad b \leftarrow \mathsf{BerExp}\left(\frac{(z-\mu)^2}{2\sigma^2} - \frac{(z-b)^2}{2\sigma_0^2}\right)$
7 **FinTantQue**
8 **Renvoyer** z

Algorithme 6 : SamplerZ.

Entrées : $x \in \mathbb{R}_+$.
Sorties : un bit tiré selon la loi de densité $\mathcal{B}(\mathrm{e}^{-x})$.

1 $q \leftarrow \lfloor x/\ln 2 \rfloor$
2 $b_1 \xleftarrow{\mathrm{R}} \mathcal{B}(2^{-q})$
3 $b_2 \leftarrow \mathsf{SmallBerExp}(x - q)$
4 **Renvoyer** $b_1 \wedge b_2$

Algorithme 7 : BerExp.

II.2 Vulnérabilités initiales

Objectifs de la section. Le GPA que nous cherchons à développer doit répondre aux trois contraintes énoncées en introduction à cette partie II. Nous devons donc vérifier si, en l'état, SamplerZ y répond ou pas.

En ce qui concerne la contrainte **M** relative à la sécurité théorique du GPA, il sera admis que SamplerZ y satisfait. L'objectif de cette section sera donc de :

Entrées : $x \in [0, 1[$ et une table $\mathrm{E} = \left[\mathrm{e}^{-j2^{-8i}}\right]$ avec $i \in [\![1, 4]\!]$ et $j \in [\![0, 255]\!]$.
Sorties : un bit tiré selon la loi de densité $\mathcal{B}(\mathrm{e}^{-x})$.

1 Écrire $x = y + \sum_{i=1}^{4} x_i \cdot 2^{-8i}$ avec $x_i \in [\![0, 255]\!]$ et $y < 2^{-32}$
2 $p \leftarrow (1 - y) \prod_{i=1}^{4} \mathrm{E}[i, x_i]$
3 $u \xleftarrow{\mathrm{R}} \mathcal{U}([0, 1])$
4 **Renvoyer** $u < p$

Algorithme 8 : SmallBerExp.

— confirmer ou infirmer le fait que SamplerZ fonctionne en temps indépendant des valeurs de ses sorties (contrainte **TS** de l'introduction) ;
— confirmer ou infirmer le fait que SamplerZ fonctionne en temps indépendant des valeurs de ses entrées, à savoir la moyenne μ et l'écart-type σ (contrainte **TE** de l'introduction).

Hypothèses de travail. Pour étudier la sécurité de SamplerZ vis-à-vis de son temps d'exécution, on fait les hypothèses suivantes :
— l'attaquant est seulement capable de mesurer le temps d'exécution de SamplerZ ;
— les opérations arithmétiques et les échantillonnages de $\mathcal{B}$ et de $\mathcal{U}$ sont constants en temps ;
— l'accès aux éléments d'une table est fait en temps constant [3].

Observation liminaire. Rappelons le fonctionnement général de SamplerZ (cf algorithme 6) : sur une moyenne μ et un écart-type σ :

1. il demande en premier lieu un nombre z à CoDFSampler, algorithme qui ne dépend ni de μ, ni de σ ;
2. puis il demande un échantillon b de $\mathcal{B}(1/2)$;
3. il effectue ensuite quelques opérations arithmétiques sur z et b ;
4. il appelle enfin BerExp, dépendant de μ, σ, z et b, qui renvoie un bit ;
5. si ce bit est nul, il retourne au point 1, sinon il renvoie l'échantillon que les quatre étapes précédentes ont permis de calculer.

D'après nos hypothèses de travail, les étapes 2 et 3 sont effectués en temps constant et ne donnent donc pas lieu à des fuites. Il en va de même de l'étape 4, puisque BerExp et la fonction SmallBerExp qu'il appelle n'effectuent que des échantillonnages de $\mathcal{B}$ ou $\mathcal{U}$ et des opérations arithmétiques sans faire intervenir de boucle — SmallBerExp accède aussi à une table, opération qu'on a également supposée constante en temps dans les hypothèses. Les éventuelles fuites dues au temps d'exécution doivent donc être cherchées dans CoDFSampler (étape 1) et dans la boucle conditionnelle (étape 5).

3. Cette hypothèse doit être regardée avec prudence, car [Ber05] montre qu'elle est fausse et l'exploite avec succès pour attaquer AES. Toutefois, elle n'aura d'intérêt que pour SmallBerExp, dont la table peut être, comme le propose [Pre17, annexe A], substituée à un calcul d'approximants de Padé [Pad99] — calcul constant en temps et rapide — si l'hypothèse s'avérait en pratique trop forte.

II.2.1 Variations du temps d'exécution dues à la boucle conditionnelle

Pour étudier les variations du temps d'exécution dues à la boucle conditionnelle, supposons que CoDFSampler s'exécute en temps indépendant de μ, de σ et de la valeur de sa sortie z — on peut faire cette hypothèse car, si elle était fausse, il serait possible de remplacer l'appel à CoDFSampler par un accès à une table de sorties de CoDFSampler précalculées, accès dont la durée serait indépendante des trois valeurs μ, σ et z. En combinaison avec notre observation liminaire, le temps d'exécution d'un tour de boucle est ainsi décorrélé, d'une part, de μ et de σ — c'est-à-dire des entrées de SamplerZ — et, d'autre part, de l'échantillon z — c'est-à-dire de la sortie potentielle de SamplerZ. Reste à voir l'influence de chacun d'eux sur la probabilité de ne pas procéder à un nouveau tour de boucle, c'est-à-dire de terminer l'algorithme.

L'échantillon z. L'échantillon z influe sur la probabilité de terminer l'algorithme (cf ligne 6 de l'algorithme 6). Toutefois, cette influence ne constitue pas une fuite : en effet, ou bien z est rejeté, auquel cas l'attaquant n'a aucun intérêt à le connaître, ou bien il est accepté, auquel cas l'attaquant peut uniquement en déduire que z est distribué selon une loi gaussienne, ce qu'il sait déjà. Par suite, le temps d'exécution de SamplerZ ne donne aucune information à l'attaquant sur la valeur de l'échantillon z accepté.

La moyenne μ. En ce qui concerne la moyenne, celle-ci étant fixe, il est possible que sa valeur fuite grâce à l'observation par l'attaquant du nombre de tours moyen qu'effectue SamplerZ avant de renvoyer un échantillon. Ce nombre de tours moyen vaut $\kappa = 2\rho_{0,\sigma_0}(\mathbb{N})/\rho_{\mu,\sigma}(\mathbb{Z})$. Mais [GPV08, lemme 2.6] montre que, si $\sigma > \eta$ (le *smooothing parameter*), alors $\rho_{\mu,\sigma}(\mathbb{Z}) \approx \rho_{0,\sigma}(\mathbb{Z})$ pour tout $\mu \in \mathbb{R}$. Par suite, en restreignant les cas d'usage de SamplerZ à $\sigma > \eta$ — ce qui est le cas dans toute la littérature publique en cryptographie —, μ n'a essentiellement pas d'influence sur la probabilité de terminer l'algorithme.

L'écart-type σ. Tout comme la moyenne, la valeur de l'écart-type est susceptible de fuiter par l'observation du nombre de tours moyen qu'effectue SamplerZ. C'est d'ailleurs le cas, comme nous l'avons observé en travaillant sur les algorithmes de la section II.1. Nous n'avons toutefois pas cherché à éliminer absolument cette relation entre σ et le temps d'exécution de SamplerZ pour plusieurs raisons liées au contexte d'utilisation :

- dans le cas de l'algorithme de Klein, l'entrée σ est différente à chaque appel de SamplerZ, il est donc impossible d'estimer un temps d'exécution moyen pour en déduire un écart-type d'entrée ;
- toujours dans le cas de l'algorithme de Klein, à supposer que l'attaquant parvienne à trouver exactement σ, il n'est pas certain, compte-tenu du fait qu'il connaissait déjà l'intervalle restreint dans lequel il se trouvait, qu'il puisse exploiter cette information de façon utile, quoique cela n'ait pas encore été étudié ;
- dans le cas de l'algorithme de Peikert, SamplerZ est destiné à être appelé plusieurs centaines de fois à la suite, ce qui aurait pour effet de rendre difficilement discernables les différentes exécutions de la fonction.

Dans tous les cas, si la variation du temps en fonction de σ est inacceptable, on peut suggérer cette contremesure : on peut montrer que $\rho_{0,\sigma}(\mathbb{Z}) \approx \sigma\sqrt{2\pi}(1 + \exp(-2\pi^2\sigma^2))$, donc on a $\rho_{\mu,\sigma}(\mathbb{Z}) \approx \sigma\sqrt{2\pi}(1 + \exp(-2\pi^2\sigma^2))$ en vertu de [GPV08, lemme 2.6]. La probabilité d'effectuer un nouveau tour de boucle est donc essentiellement proportionnelle à σ.

Il suffit donc de multiplier cette probabilité par $\sigma_{\min}/\sigma$ où $\sigma_{\min} \in]0, \sigma]$ pour perdre cette dépendance, en contrepartie d'un taux de rejet plus important. Rappelons à l'appui de cette suggestion que l'indépendance du temps d'exécution de SamplerZ vis-à-vis de μ est conditionné au fait que σ soit minoré par le *smoothing parameter*, ce qui permet d'avoir recours à un tel $\sigma_{\min}$.

In fine, il ressort de la discussion de cette sous-section que, si CoDFSampler s'exécute en temps indépendant de μ, de σ et de la valeur de sa sortie, alors SamplerZ s'exécute en temps indépendant de μ, de la valeur de sa sortie et de σ si nécessaire. Reste donc à voir si l'hypothèse concernant CoDFSampler est vraie.

II.2.2 Temps d'exécution de CoDFSampler

En fonction de μ et σ. Puisque CoDFSampler échantillonne une loi normale de moyenne et d'écart-type fixés à l'implémentation, il est totalement indépendant des valeurs de μ et de σ.

En fonction de sa sortie. CoDFSampler, pour renvoyer un entier z, fait appel à $z+1$ échantillons de $\mathcal{U}([0,1])$. Le temps d'exécution de CoDFSampler est donc proportionnel à celui de l'échantillonneur de $\mathcal{U}([0,1])$, ce qui permet de retrouver immédiatement z si l'on sait mesurer finement ce temps. Comme nous n'avions pas de matériel pour essayer d'observer cette vulnérabilité potentielle (en particulier, les fonctions `clock()` du langage C et `time.clock()` de Python, qui repose celle du C, se sont montré trop imprécises pour estimer le temps d'exécution), nous avons montré la faille de principe en appliquant, sous Unix, l'outil `callgrind` du logiciel `Valgrind` au petit programme C suivant :

```
int main() {
    printf("%ld", CoDFSampler());
    return 0;
}
```

où `CoDFSampler()` est la fonction fournie en annexe A. Elle suit à la lettre l'algorithme 5, si ce n'est que, pour ne pas avoir à manipuler des nombres décimaux, les éléments précalculés de la table, qui sont des décimaux entre 0 et 1, sont ramenés à des nombres entiers de 64 bits par multiplication par 2^{64}, puis troncature à l'unité — on rappelle que 64 bits de précision sont suffisants pour assurer la sécurité (condition **M** de l'introduction) du GPA. De même, elle fait appel à une fonction, `rng_uint64()`, qui échantillonne $\mathcal{U}([\![0, 2^{64}-1]\!])$ et non $\mathcal{U}([0,1])$.

L'outil `callgrind` produit un fichier assez peu digeste qui, une fois interprété (dans notre cas par le logiciel `kcachegrind`), nous permet de savoir combien de fois `rng_uint64()` a été appelée par `CoDFSampler()`, donc de deviner la valeur affichée à l'écran. Rendre la sortie de CoDFSampler indépendante du nombre d'échantillonnages de $\mathcal{U}([0,1])$ est donc primordial. La résolution de ce problème est l'objet de la section suivante.

II.3 Révision de CoDFSampler

Le but de cette section est de rendre le temps d'exécution de CoDFSampler indépendant de sa sortie. En fait, la solution que nous développerons dans la suite consiste, d'une part, à engendrer successivement un grand nombre N d'échantillons, afin que la génération

d'un ensemble de N échantillons requière un temps proche de N fois le temps moyen d'exécution de CoDFSampler ; d'autre part, à engendrer plusieurs échantillons à la fois, afin de réduire, puisque l'approche précédente ne résout pas notre problème du point de vue de la génération d'un seul échantillon, la fuite d'information liée au nombre de tours de boucle de CoDFSampler — et accessoirement d'engendrer les échantillons plus vite. L'algorithme qui résulte de ces modifications a été appelé CoDFArray. On décrira en premier lieu le fonctionnement de CoDF_Para, qui engendre huit échantillons à la fois, puis on déterminera le nombre de fois qu'il doit être appelé par CoDFArray pour que celui-ci fonctionne en temps *constant*.

Dans la suite, on désigne par $c = (c_7c_6 \dots c_0)_{2^8}$ et $d = (d_7d_6 \dots d_0)_{2^8}$ deux entiers de 64 bits.

II.3.1 Construction de CoDF_Para

On rappelle que l'implémentation de CoDFSampler donnée en annexe A n'utilise que des entiers de 64 bits et non des décimaux. CoDF_Para va utiliser cela. L'idée directrice en est de voir un entier de 64 bits comme huit entiers de 8 bits et d'effectuer, grâce aux opérations sur 64 bits, des opérations sur ces huit nombres de 8 bits en parallèle. À la fin de CoDF_Para, ces huit nombres seront huit échantillons d'une v.a. suivant une loi de densité $\tilde{\mathcal{G}}^+$ définie par l'égalité II.2.

Nous avons donc deux points à traiter pour transformer CoDFSampler (algorithme 5) en CoDF_Para :

— effectuer en même temps huit comparaisons sur les entiers c et d, c'est-à-dire déterminer simultanément pour tous les $0 \leqslant i \leqslant 7$ si $c_i > d_i$;

— s'assurer que l'incrémentation de certains échantillons en cours de production continue tandis que celle des échantillons prêts s'est bien arrêtée.

Comparaisons simultanées. Les comparaisons simultanées constituent le point le plus délicat. Dans CoDFSampler, on compare un entier pseudo-aléatoire de 64 bits avec un entier tabulé de 64 bits également. Les arguments de [Pre17] nous assurent que cela est suffisant pour échantillonner $\mathcal{G}^+$ de façon mathématiquement sûre (c'est-à-dire sans créer de biais statistique exploitable entre la distribution $\mathcal{G}^+$ théorique et celle qui résulte de l'échantillonnage). Dans la procédure de comparaison que nous nous sommes imposée, nous ne pouvons comparer que des entiers de 8 bits, et cela n'est plus suffisant pour échantillonner $\mathcal{G}^+$ de façon mathématiquement sûre.

Ceci dit, le choix de $\mathcal{G}$ (duquel vient le problème d'échantillonnage de $\mathcal{G}^+$) comme distribution d'appui n'est le fruit que d'un compromis temps-mémoire dont on discutera en II.3.3 et qui sacrifiait un peu de temps pour économiser un peu de mémoire. Pour parvenir à nos fins, nous allons devoir choisir une autre distribution, légèrement différente, au prix du compromis temps-mémoire inverse, comme nous le verrons également en II.3.3.

Notons $\tilde{\mathcal{G}}^+$ la densité issue de la fonction de densité conditionnelle :

$$\mathrm{CoDF}_{\tilde{\mathcal{G}}^+} = \lfloor 2^8 \cdot \mathrm{CoDF}_{\mathcal{G}^+} \rfloor / 2^8 \tag{II.2}$$

c'est-à-dire que $\mathrm{CoDF}_{\tilde{\mathcal{G}}^+}$ est égale à $\mathrm{CoDF}_{\mathcal{G}^+}$ tronquée aux huit bits de poids fort de sa mantisse. Dans CoDF_Para, la densité $\mathcal{G}^+$ est remplacée par $\tilde{\mathcal{G}}^+$: ainsi, les entiers que nous avons à comparer sont désormais de taille 8 bits. Ceci établi, le problème de la comparaison simultanée est résolu par la fonction multicompare (algorithme 9) qui repose sur le fait que

l'ordre des deux entiers c et d est fixé par le bit de poids le plus fort parmi ceux qui diffèrent entre ceux de c et ceux de d. L'algorithme est le suivant :

1. on calcule $x = c \oplus d$; ainsi un bit de x vaudra 1 si, et seulement si, le bit de même poids de c est différent de celui de d ;
2. on recherche dans x le bit à 1 de poids le plus fort et on effectue un ET logique entre lui et le bit de d de même poids ;
3. si le résultat vaut 1, alors $d > c$, dans le cas contraire, $d \leqslant c$.

Entrées : deux entiers c et d de 64 bits
Sorties : un entier $l = (l_7 l_6 \ldots l_0)_{2^8}$ tel que $l_i = 0$ si $c_i \geqslant d_i$ et $l_i = 1$ sinon.

```
cxord ← c ⊕ d
masque ← 0x8080808080808080
cmp ← 0
Pour i allant de 7 à 0 faire
    y ← cxord ∧ masque
    cmp ← cmp ⊕ ((y ∧ d) >> i)
    masque ← y ⊕ masque
    masque ← masque >> 1
FinPour
Renvoyer cmp
```

Algorithme 9 : multicompare.

Incrémentation différentiée. Le second problème, celui de l'incrémentation différentiée, est résolu en exploitant la sortie de multicompare. Soit $s = 0$, $a = 0\text{x}0101010101010101$ et $y = \mathsf{multicompare}(c, d)$. y est un nombre de huit octets valant 0 ou 1, et, précisément, valant 1 si, et seulement si, $c_i < d_i$. Les octets nuls du nombre $x = a \wedge \neg y$ étant ceux non-nuls de y, l'octet x_i de x est nul si, et seulement si, $c_i < d_i$. À ce stade, posons $s \leftarrow s + x$ et $a \leftarrow x$. En demandant cette fois $y = \mathsf{multicompare}(c', d')$, et en recalculant $x = a \wedge \neg y$, l'octet x_i est non-nul si, et seulement si, $c_i \geqslant d_i$ et $c'_i \geqslant d'_i$. Si l'on fait à nouveau $s \leftarrow s + x$, alors l'octet s_i vaut 2 si, et seulement si, $c_i \geqslant d_i$ et $c'_i \geqslant d'_i$.

En appliquant cette observation de façon itérative, si $((c^{(j)}, d^{(j)}))_{1 \leqslant j \leqslant n}$ est la suite des arguments de multicompare, on montre qu'un octet s_i de s est incrémenté après le k^{e} appel à multicompare si, et seulement si, l'expression $\bigwedge_{1 \leqslant j \leqslant k} \{c_i^{(j)} \geqslant d_i^{(j)}\}$ est vraie. Ceci prouve, lorsque les $c^{(j)}$ sont des entiers aléatoires et les $d^{(j)}$ une concaténation de huit fois les huit bits de poids forts de $\mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(j)$, la correction de CoDF_Para (algorithme 10).

II.3.2 Construction de CoDFArray

On rappelle qu'on veut que CoDFArray engendre successivement un grand nombre d'échantillons, de sorte que son temps d'exécution puisse être fixé à un surcoût aussi faible que possible. Formellement, s'il faut M tours en moyenne à CoDF_Para pour donner une sortie (c'est-à-dire M appels à multicompare), on souhaite qu'en $\lceil \mathrm{M} n \beta \rceil$ appels à multicompare en tout, avec $\beta \geqslant 1$ aussi petit que possible, CoDFArray nous fournisse au moins $\mathrm{N} = 8n$ échantillons avec probabilité $1 - \varepsilon$. Pour construire l'algorithme CoDFArray, il faut donc déterminer les valeurs de M et de β (cette dernière en fonction de ε).

Entrées : une table précalculée des $\mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(i)$, dont les valeurs sont ramenées à des entiers entre 0 et $2^8 - 1$.
Sorties : un entier de 64 bits dont chaque octet est un entier $z \in \mathbb{Z}$ avec probabilité $\tilde{\mathcal{G}}^+(z)$.

1 cond-arret $\leftarrow$ 0x0101010101010101
2 $i \leftarrow 0$
3 échantillons $\leftarrow 0$
4 **TantQue** cond-arret $\neq 0$ **faire**
5 $\quad u \overset{\mathrm{R}}{\leftarrow} \mathcal{U}([\![0, 2^{64} - 1]\!])$
6 $\quad v \leftarrow \mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(i) || \ldots || \mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(i)$ `/* chaque octet de` v `est une copie de` $\mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(i)$ `*/`
7 $\quad$ cmp $\leftarrow$ multicompare(u, v)
8 $\quad$ cond-arret $\leftarrow$ cond-arret $\wedge \neg$cmp
9 $\quad$ échantillons $\leftarrow$ échantillons + cond-arret
10 $\quad i \leftarrow i + 1$
11 **FinTantQue**
12 **Renvoyer** échantillons

Algorithme 10 : CoDF_Para

Avant tout, pour ne pas alourdir les écritures, il est convenu une fois pour toutes que, à défaut de précision explicite, les sommes, produits, unions et intersections indicées vont de 1 à 8.

Calcul du nombre moyen de tours fait par CoDF_Para. Le nombre de tours faits par CoDF_Para est égal au maximum des échantillons engendrés, plus un. Soit X la v.a. qui prend pour valeur ce maximum. Soit s l'entier renvoyé par CoDF_Para et s_i chacun de ses huit octets. Soit un entier $m > 0$. Soit $\mathrm{A} = \{\forall i, s_i \leqslant m\}$ et $\mathrm{B} = \{\exists i \text{ tq } s_i = m\}$. Alors :

$$\begin{aligned}\mathbb{P}(\mathrm{X} = m) = \mathbb{P}(\mathrm{A} \cap \mathrm{B}) &= \mathbb{P}(\mathrm{A})\mathbb{P}_{\mathrm{A}}(\mathrm{B}) \\ &= \mathbb{P}(\mathrm{A}) - \mathbb{P}(\mathrm{A})\mathbb{P}_{\mathrm{A}}(\overline{\mathrm{B}}) \\ &= \mathbb{P}(\mathrm{A}) - \mathbb{P}(\mathrm{A} \cap \overline{\mathrm{B}}) \\ &= \tilde{\mathcal{S}}(m)^8 - \tilde{\mathcal{S}}(m-1)^8,\end{aligned}$$

où $\tilde{\mathcal{S}}(z) = \sum_{i=0}^{z} \tilde{\mathcal{G}}^+(i)$.

Si $m = 0$, alors, puisque les huit s_i sont deux-à-deux indépendants et puisque $\mathbb{P}(s_i = 0) = \tilde{\mathcal{G}}^+(0)$, on a $\mathbb{P}(\mathrm{X} = m) = \tilde{\mathcal{G}}^+(m)^8$.

Ainsi, en moyenne, CoDF_Para fait $\mathrm{M} = 1 + \mathbb{E}(\mathrm{X}) \approx 4{,}36$ tours par appel.

Détermination de β. On cherche $\beta \geqslant 1$ le plus petit possible tel qu'on obtienne, avec probabilité $1 - \varepsilon$, un nombre d'entiers supérieur ou égal à n en exactement $\lceil \mathrm{M}n\beta \rceil$ appels à multicompare.

Soit les événements $\mathrm{A}_{k,n} = \{$après k tours, CoDF_Para a livré au moins n entiers$\}$ et $\mathrm{B}_{k,n} = \{$après k tours, CoDF_Para a livré exactement n entiers$\}$. On a $\mathrm{A}_{k,n} = \bigcup_{i \geqslant n} \mathrm{B}_{k,i}$, ce qui, étant donné que les $\mathrm{B}_{k,i}$ sont deux-à-deux distincts et que l'on ne peut obtenir plus

$N = 8n$	$-\log_2 \varepsilon$	β	mémoire requise (en octets)
1 024	64	1,220	190×8
	80	1,250	200×8
	100	1,280	210×8
	128	1,325	220×8
2 048	64	1,155	350×8
	80	1,175	360×8
	100	1,195	360×8
	128	1,225	380×8

TABLE II.1 – valeur de β en fonction de N et de ε.

de k échantillons en k tours, conduit à :

$$\mathbb{P}(A_{k,n}) = \sum_{i=n}^{k} \mathbb{P}(B_{k,i}).$$

Reste donc à déterminer ces $\mathbb{P}(B_{k,i})$.

Vu que CoDF_Para doit effectuer au moins un tour avant de renvoyer un entier, on a $\mathbb{P}(B_{0,0}) = 1$ et $\mathbb{P}(B_{0,n}) = 0$ pour tout $n > 0$. Ce cas étant réglé, on suppose à présent que $k \geqslant 1$.

Introduisons l'événement $T_i = \{\text{attendre } i \text{ tours pour obtenir un entier}\}$. Remarquons qu'on peut écrire :

$$B_{k,n+1} = \bigcup_{i=1}^{k} T_i \cap B_{k-i,n}$$

et que les T_i sont disjoints, donc que les $T_i \cap B_{k-i,n}$ le sont aussi. De plus, puisque chaque exécution de CoDF_Para est indépendante des autres (en particulier celle correspondant à T_i est indépendante de celles nécessaires pour vérifier $B_{k-i,n}$), T_i et $B_{k-i,n}$ sont indépendants. Il s'ensuit que :

$$\mathbb{P}(B_{k,n+1}) = \sum_{i=1}^{k} \mathbb{P}(T_i)\mathbb{P}(B_{k-i,n}).$$

Initialisons cette suite : $B_{k,0}$ est vrai signifie qu'après le k^{e} tour, aucun entier n'a été renvoyé, c'est-à-dire que le maximum des échantillons en cours de génération vaut au moins k (car CoDF_Para s'arrête dès que le maximum de ces échantillons vaut $k-1$ à l'issue du tour k). Par suite $\mathbb{P}(B_{k,0}) = \mathbb{P}(X \geqslant k)$. Par ailleurs $T_i = \{X = i-1\}$, donc $\mathbb{P}(T_i) = \mathbb{P}(X = i-1)$.

On est désormais en mesure de calculer $\mathbb{P}(A_{k,n})$. Le tableau II.1 fournit quelques valeurs pour β en fontion de la valeur de ε exigée et du nombre d'échantillons désiré. Il donne en plus la taille de la table requise pour stocker tous les échantillons produits avec probabilité $1-\varepsilon$. Cette taille est égale à huit fois le plus petit ν tel que $\mathbb{P}(B_{\lceil n M \beta \rceil, \nu}) < \varepsilon$ (on a considéré seulement des nombres ronds lors de la recherche des ν).

CoDFArray. L'algorithme 11 décrit CoDFArray tel qu'il résulte des choix faits dans cette partie. Pour la lisibilité, on a fait le choix de l'écrire dans le cas particulier $N = 2\,048$ et $\varepsilon = 2^{-64}$, mais il n'est pas difficile l'adapter à n'importe quel choix de paramètres N et ε.

Entrées : une table précalculée des $\mathrm{CoDF}_{\tilde{\mathcal{G}}^+}(i)$ sous la forme d'entiers entre 0 et $2^8 - 1$.

Sorties : avec probabilité $1 - 2^{-64}$, un tableau d'au moins $\mathrm{N} = 2\,048$ échantillons d'une v.a. suivant la loi de densité $\tilde{\mathcal{G}}^+$.

```
1  sortie ← [350 × 8 octets à 0]   /* c'est la taille de la table donnée par la
     ligne correspondante de la table II.1 */
2  cond-arret ← 0x0101010101010101
3  cpt ← 0
4  j ← 0
5  i ← 0
6  TantQue cpt < ⌈nMβ⌉ = 1 291 faire
7      Si cond-arret = 0 alors
8          cond-arret ← 0x0101010101010101
9          i ← 0
10         j ← j + 8
11     Sinon
12         cond-arret ← cond-arret
13         i ← i
14         j ← j + 0
15     FinSi
16     u ←R U([[0, 2^64 − 1]])
17     v ← CoDF_G̃+(i)|| ... ||CoDF_G̃+(i)
18     cmp ← multicompare(u, v)
19     cond-arret ← cond-arret ∧ ¬cmp
20     sortie[j] ← sortie[j] + cond-arret     /* l'opération concerne en fait huit
         cases adjacentes */
21     i ← i + 1
22     cpt ← cpt + 1
23 FinTantQue
24 Renvoyer sortie
```

Algorithme 11 : CoDFArray, dans le cas particulier $\mathrm{N} = 2\,048$ et $\varepsilon = 2^{-64}$.

II.3.3 Impact de CoDFArray sur SamplerZ

Le passage de CoDFSampler à CoDFArray a eu deux conséquences importantes. La première d'entre elles est le calcul « tout-en-un » de plusieurs centaines d'échantillons alors que CoDFSampler n'en produisait qu'un à la fois. La seconde est la modification, lors du passage de l'un à l'autre algorithme, de la distribution échantillonnée. L'objet de cette section est d'apporter à SamplerZ les rectifications que ces conséquences imposent et de les discuter.

Précalcul des échantillons. Dans l'algorithme 6, la boucle `tantque` demande en premier lieu un échantillon de $\mathcal{G}^+$ à CoDFSampler. Cet appel est remplacé par le passage en paramètre à SamplerZ d'une table d'échantillons précalculée par CoDFArray, auprès de laquelle la boucle se fournit en échantillons. La pertinence de cette proposition tient au fait que, dans le contexte d'utilisation des algorithmes de Peikert ou de Klein, SamplerZ est appelé plusieurs centaines de fois à la suite. Il est bien évident qu'il est hors de prix d'appeler CoDFArray à chaque exécution de SamplerZ. Si celui-ci doit être utilisé ponctuellement, ou si l'environnement est très contraint en mémoire, une solution peut être de faire appel à CoDF_Para une fois tous les $t \leqslant 8$ tours de boucle, quoique la sécurité vis-à-vis des *timing-attacks* de cette suggestion, si elle est *a priori* meilleure que celle offerte par l'utilisation de CoDFSampler, reste une question ouverte.

Modification de la distribution. La modification de la distribution doit être prise en compte dans la façon dont est effectuée l'étape de rejet dans SamplerZ (ligne 6 de l'algorithme 6). Cette ligne provenait du quotient de la distribution visée, à savoir $\mathcal{D}_{\mu,\sigma}$, par la distribution d'appui, à savoir $\mathcal{G}$ (voir la section II.1.2). Or, si on utilise CoDFArray, la distribution d'appui n'est plus $\mathcal{G}$ mais $\tilde{\mathcal{G}} : z \mapsto \frac{1}{2}(\mathbb{1}_{\{z \geqslant 1\}} \cdot \tilde{\mathcal{G}}^+(z-1) + \mathbb{1}_{\{z \leqslant 0\}} \cdot \tilde{\mathcal{G}}^+(-z))$ pour tout $z \in \mathbb{Z}$. Comme nous voulons toujours faire appel à BerExp pour effectuer le rejet, nous devons rectifier son argument.

Pour réaliser cette correction, on suit le raisonnement suivant. Il ressort de la section II.1.1 qu'on peut échantillonner $\mathcal{D}_{\mu,\sigma}$ à partir de la distribution d'appui $\tilde{\mathcal{G}}$ s'il existe une constante $\kappa \in]0, +\infty[$ telle que :

$$\frac{1}{\kappa} \cdot \frac{\mathcal{D}_{\mu,\sigma}}{\tilde{\mathcal{G}}} \leqslant 1.$$

On cherche donc une telle constante.

Soit $\check{\mathcal{G}} = 2\rho_{0,\sigma_0}(\mathbb{Z}) \cdot \mathcal{G}$. On a déjà souligné (cf égalité II.1) que $\mathcal{D}_{\mu,\sigma} \leqslant \check{\mathcal{G}}$, c'est-à-dire que :

$$\frac{\mathcal{D}_{\mu,\sigma}}{\mathcal{G} \cdot \check{\mathcal{G}}(\mathbb{Z})} \leqslant 1.$$

Soit maintenant $m = \min_{z \in \mathbb{Z}} \tilde{\mathcal{G}}/\mathcal{G}$, alors :

$$m \cdot \frac{\mathcal{D}_{\mu,\sigma}}{\tilde{\mathcal{G}} \cdot \check{\mathcal{G}}(\mathbb{Z})} \leqslant \frac{\mathcal{D}_{\mu,\sigma}}{\mathcal{G} \cdot \check{\mathcal{G}}(\mathbb{Z})} \leqslant 1. \tag{II.3}$$

Ceci nous garantit l'existence de la constante κ qu'on cherche. Pour la choisir, on a deux critères : la prendre la plus petite possible pour limiter le taux de rejet, et, pour exprimer

cette probabilité sous la forme de l'exponentielle la plus simple possible (dans l'objectif d'utiliser BerExp), faire en sorte qu'elle nous débarrasse des constantes de normalisation.

On peut remarquer que l'égalité II.1 dit en fait que $\rho_{\mu,\sigma} \leqslant \check{\mathcal{G}}$, donc, en suivant le même raisonnement que précédemment, il se trouve que :

$$m \cdot \frac{\rho_{\mu,\sigma}}{\tilde{\mathcal{G}} \cdot \check{\mathcal{G}}(\mathbb{Z})} \leqslant 1,$$

c'est-à-dire qu'on a :

$$m \cdot \rho_{\mu,\sigma}(\mathbb{Z}) \cdot \frac{\mathcal{D}_{\mu,\sigma}}{\tilde{\mathcal{G}} \cdot \check{\mathcal{G}}(\mathbb{Z})} \leqslant 1$$

On répond donc à nos deux objectifs en choisissant en particulier :

$$\kappa = \frac{\check{\mathcal{G}}(\mathbb{Z})}{m \cdot \rho_{\mu,\sigma}(\mathbb{Z})}.$$

Finalement, la nouvelle probabilité de rejet s'écrit :

$$\exp\left(-\frac{(z-\mu)^2}{2\sigma^2} - \ln\left(\frac{\check{\mathcal{G}}(\mathbb{Z})}{m}\tilde{\mathcal{G}}(z)\right)\right).$$

La ligne 6 de l'algorithme 6 doit donc être réécrite ainsi :

$$b \leftarrow \mathsf{BerExp}\left(\frac{(z-\mu)^2}{2\sigma^2} + \ln\left(\frac{\check{\mathcal{G}}(\mathbb{Z})}{m}\tilde{\mathcal{G}}(z)\right)\right).$$

Pour gagner en vitesse, les $\ln(\check{\mathcal{G}}(\mathbb{Z})\tilde{\mathcal{G}}(z)/m)$ peuvent être précalculés et mis en table. Il est à noter que la taille d'une telle table reste modeste. En effet, on a souligné au début de cette partie (cf section II.1) qu'une précision de calcul limitée à 64 bits est suffisante pour assurer la sécurité mathématique du GPA. Or, pour $\sigma_0 = 2$, $\check{\mathcal{G}}(\mathbb{Z})\tilde{\mathcal{G}}(z)/m < 2^{-64}$ pour $|z| > 20$. À cela s'ajoute le fait que $\tilde{\mathcal{G}}(z) = \tilde{\mathcal{G}}(1-z)$, ce qui permet de ne précalculer et stocker que les valeurs de $\ln(\check{\mathcal{G}}(\mathbb{Z})\tilde{\mathcal{G}}(z)/m)$ pour $z > 0$. Finalement, la table précalculée ne contient, pour $\sigma_0 = 2$, que dix-neuf nombres flottants double précision.

L'algorithme 12 reprend toutes les modifications qui ont été apportées à SamplerZ par les deux paragraphes précédents.

II.3.4 Performances de SamplerZ

Environnement et méthode de mesure. La vitesse du SamplerZ qui résulte de cette section a été mesurée sur l'implémentation fournie avec la version électronique de ce rapport et présente dans le dossier `implems/samplerZ`. Elle a été estimée sous Unix en utilisant la fonction `time` du terminal. Le processeur de travail était un Intel Core i3-6100 CPU de fréquence 3,7 GHz. Le tableau II.2 consigne les résultats de ces mesures. Il donne également la vitesse des fonctions appelées par `SamplerZ`.

Les mesures ont été réalisées avec (sauf pour `CoDFArray`) et sans précalcul de l'aléa. Les estimations avec précalcul de l'aléa sont donc indépendantes du GPA uniforme utilisé, ce qui permet de s'approcher du temps consommé par les seules opérations des fonctions

Entrées : une moyenne $\mu \in [0, 1[$ et un écart-type $\sigma \leqslant \sigma_0$, une table $\mathcal{A}$ d'échantillons de $\tilde{\mathcal{G}}^+$ fournie par CoDFArray, une table α des $\ln(\check{\mathcal{G}}(\mathbb{Z})\tilde{\mathcal{G}}(z)/m)$ précalculés.
Sorties : $z \in \mathbb{Z}$ avec probabilité $\mathcal{D}_{\mu,\sigma}(z)$.

1 $b \leftarrow 0$
2 $i \leftarrow 0$
3 **TantQue** $b = 0$ **faire**
4 $\quad x \leftarrow \mathcal{A}[i]$
5 $\quad b \xleftarrow{R} \mathcal{B}(1/2)$
6 $\quad z \leftarrow x + b - 2x(1 - b)$
7 $\quad b \leftarrow \mathsf{BerExp}\left(\frac{(z-\mu)^2}{2\sigma^2} + \alpha[x]\right)$
8 $\quad i \leftarrow i + 1$
9 **FinTantQue**
10 **Renvoyer** z

Algorithme 12 : SamplerZ adapté à CoDFArray.

Aléa précalculé	Algorithme	Vitesse (échantillons par seconde)
Non	`CoDFArray`	54 000 000
Non	`SmallBerExp`	29 500 000
	`BerExp`	15 500 000
	`SamplerZ`	8 500 000
Oui	`SmallBerExp`	75 000 000
	`BerExp`	46 000 000
	`SamplerZ`	14 500 000

TABLE II.2 – Vitesses atteintes par nos implémentations.

mesurées. Le GPA uniforme utilisé pour obtenir les vitesses sans précalcul de l'aléa est celui implémenté par J. Schanck pour CPQREF [4], à base de Salsa20.

La constante σ_0 a été fixée à 2, μ à 1 et σ à 1,8. Pour les paramètres de `CoDFArray`, on a pris ceux de l'algorithme 11.

À noter que, pour `CoDFArray`, le chiffre est le résultat de la multiplication du nombre de tableaux obtenu en une seconde par le nombre d'échantillons *demandé*, et non celui effectivement obtenu en moyenne, qui a été observé près de 15% supérieur. Nous avons fait ce choix car les échantillons supplémentaires obtenus n'ont pas vocation à être utilisés par `SamplerZ`, qui prend en entrée un tableau de taille le nombre d'échantillons demandé.

On souligne également que le chiffre donné pour `SamplerZ` n'inclut pas la génération de la table des échantillons de $\tilde{\mathcal{G}}^+$.

Comparaison avec [MW17]. Le GPA de [MW17] fournit, selon leurs estimations, près de 10^7 échantillons par seconde à paramètres comparables. Toutefois, [MW17] ne précise pas le cadre dans lequel cette estimation a été réalisée, en particulier le processeur

4. Disponible à de multiples endroits sur Internet, entre autres : `https://github.com/floodyberry/supercop/tree/master/crypto_sign/ntrumls401x/ref/fastrandombytes.c` [consulté le 22/08/2017].

de travail. Le code qui a servi à réaliser leurs mesures a été publié trop tard pour que nous puissions inclure dans ce rapport une comparaison des performances des deux GPA mesurées sur la même machine. On relève également que [MW17] donne la vitesse du GPA *non protégé* des attaques par mesure du temps d'exécution, tandis que notre GPA l'est.

II.3.5 Risques résiduels identifiés

Le GPA résultant de l'algorithme 12 ouvre de nouvelles questions sur lesquelles le temps ne nous a pas permis de nous pencher.

D'une part, que se passe-t-il si, par exemple pour des raisons liées à la mémoire disponible, CoDF_Para est préféré à CoDFArray et appelé pendant l'exécution de SamplerZ ? Il n'est pas certain que les informations que CoDF_Para laisse fuiter — à savoir le maximum en valeur absolue de huit échantillons successifs — soient inexploitables par un attaquant.

D'autre part, il reste le préoccupant problème des *cache-attacks*. Menées contre la table α, elles sont susceptible de livrer la sortie de SamplerZ, et l'application de mesures de prévention de ces attaques comme celles suggérées par [BHLY16, BCNS15] risque d'entraîner un surcoût en temps préjudiciable. Compte-tenu de la taille très réduite de la table, il peut être envisageable, pour la soustraire aux regards, de la placer dans la pile du processeur avant le premier appel à SamplerZ et de la dépiler après le dernier.

Par ailleurs, les *cache-attacks* menacent également CoDFArray. Les accès de l'algorithme à la table de la distribution conditionnelle donne à l'observateur des informations directes sur la valeur du maximum en valeur absolue de huit échantillons. La même information peut être obtenue par une attaque par *branch-tracing*, qui consiste à identifier tous les brachements conditionnels effectués par l'algorithme visé. Nous en avons mené une, exposée en annexe B, contre le SamplerZ résultant de l'algorithme 6, que, faute de temps, nous n'avons pas pu adapter au nouveau SamplerZ (l'algorithme 12), mais il est probable qu'elle nous aurait livré la valeur de ce maximum. Là encore, il faudrait s'interroger sur le risque que cela représente pour le schéma qui utilise le GPA.

II.4 Conclusion de la partie

Cette partie s'achève sur un GPA qui remplit tous les objectifs qui lui avaient été assignés : d'une part, il est mathématiquement sûr et son temps d'exécution est indépendant des ses entrées et de sa sortie ; d'autre part, sa vitesse n'a pas eu trop à souffrir des protections qui lui ont été ajoutées. Ses performances sont proches de celles des meilleurs GPA publiés à ce jour et j'ai bon espoir que nous parvenions à les améliorer encore d'ici à la soumission prévue d'un article à son sujet à PQCrypto-2018.

Il reste toutefois un certain nombre de vulnérabilités potentielles desquelles il faudrait prémunir le GPA, notamment celles ouvrant la voie aux *cache-attacks*. Il faudrait étudier l'impact d'un certain nombre de propositions qui ont déjà été faites pour s'en protéger (par exemple [BCNS15]) sur les performances du GPA.

Conclusion générale

Mon stage chez Thales Communications & Security a pris fin le 31 août 2017 et a duré près de six mois au cours desquels j'ai pu étudier la sécurité physique des schémas post-quantiques selon deux axes complémentaires.

Le premier axe est l'axe offensif, qui m'a donné l'opportunité de proposer une attaque par injection de faute inédite contre le schéma de signature SPHINCS. Cette attaque requiert peu de signatures fautées et une seule faute par signature pour permettre à l'adversaire de signer n'importe quel message. Elle est par ailleurs discrète, les signatures fautées qu'elle demande peuvant être rendues valides sans contrainte importante. L'attaque peut de surcroît réussir avec une seule signature fautée, en contrepartie d'une puissance de calcul importante mais envisageable. Malgré ces bons résultats initiaux, cette proposition peut être considérablement améliorée et cette amélioration sera l'objet d'un article que nous soumettrons à PQCrypto-2018.

Le second axe est l'axe défensif : j'ai participé à la protection d'un générateur pseudo-aléatoire qui remplit tous les objectifs qui lui avaient été assignés : d'une part, il est mathématiquement sûr et son temps d'exécution est indépendant des ses entrées et de sa sortie ; d'autre part, ses performances sont proches de celles des meilleurs générateurs publiés à ce jour et j'ai bon espoir que nous parvenions à les améliorer encore d'ici à la soumission prévue d'un article à son sujet à PQCrypto-2018.

Lorsque je me suis assis pour la première fois derrière mon bureau, j'étais désabusé et frustré de voir la cryptanalyse des schémas asymétriques attendre, impuissante, que nos connaissances en mathématiques progressent pour arriver à ses fins. Au point que j'en étais arrivé à me détourner de toute la cryptographie asymétrique. Ce stage m'a littéralement réconcilié avec elle. Aujourd'hui, après six mois d'un stage qui s'est concentré sur les attaques physiques, je vois tous les schémas, symétriques comme asymétriques, comme des sources intarissables d'opportunités pour les attaquants et de défis pour tous ceux qui cherchent à les protéger.

Et je trouve cela particulièrement stimulant ! J'ai pris grand plaisir à observer les algorithmes sur lesquels j'ai travaillé et à imaginer contre eux une série d'attaques par canaux auxiliaires plus ou moins farfelues. J'ai été également très content d'avoir à réaliser autant d'implémentation et à imaginer des algorithmes pour résoudre certains des problèmes qui se sont posés. En particulier, j'ai beaucoup apprécié apprendre sur le tas le fonctionnement de l'assembleur qui, si je n'en connais pas encore toutes les conventions, ce qui nous obligera peut-être à revoir ce fragment de code, nous a rendu de grands services lorsqu'il s'est agi de donner un coup de fouet à notre générateur. J'ai également appris bon nombre de choses en Python, en C et sur le fonctionnement d'Ubuntu qui seront bien utiles dans ma vie future.

À côté de tous ces aspects électroniques et informatiques, j'ai été agréablement surpris de la quantité et de la diversité des mathématiques que j'ai eu à mobiliser pour parvenir à mes fins. Certes, c'est surtout, on l'a vu tout au long du rapport, les probabilités qui ont dominé ce travail mathématique, mais elle m'ont permis de renouer avec l'analyse qui m'est très chère et qui, d'ailleurs, s'est amusée à me jouer bien des tours lorsque j'appliquais mes vieux réflexes d'analyste sur des fonctions définies sur $\mathbb{Z}$ comme si elles l'étaient sur $\mathbb{R}$.

En bref, ce stage m'a montré que les attaques physiques répondaient exactement à ce que je recherchais en postulant au master Cryptis de Limoges : des mathématiques avancées ayant des applications très pratiques dans un domaine qui me motive.

C'est pour cela que je suis heureux de rejoindre — temporairement, hélas ! — les équipes du CESTI de Thales qui se trouve à Toulouse, et pour lequel je réaliserai des tests d'intrusion. J'espère y apprendre d'une part, quelles attaques sont réalistes contre telles ou telles opérations et, d'autre part, bien entendu, de nouvelles attaques — dont j'ai pu entendre le nom de certaines sans avoir eu le temps de mettre un concept en face — avec, accessoirement, les contremesures y afférentes.

Je n'y retrouverai certes pas l'intégralité de ce qui m'a plu dans le stage, notamment le côté recherche et développement, mais cette phase d'apprentissage pratique m'a paru indispensable pour pouvoir peut-être renouer avec lui dans le futur.

Bibliographie

[ABB10] S. Agrawal, D. Boneh et X. Boyen – « Lattice basis delegation in fixed dimension and shorter-ciphertext hierarchical IBE », *Advances in Cryptology – CRYPTO-2010* (T. Rabin, éd.), LNCS, vol. 6223, Springer, août 2010, p. 98–115.
✈ Cité pages 2 et 27.

[ADPS15] E. Alkim, L. Ducas, T. Pöppelmann et P. Schwabe – « Post-quantum key exchange - a new hope », Cryptology ePrint Archive, rapport 2015/1092, 2015, http://eprint.iacr.org/2015/1092.
✈ Cité page 2.

[BCG+14] J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing et P. Weiden – « Discrete Ziggurat : A time-memory trade-off for sampling from a Gaussian distribution over the integers », *SAC 2013 : 20th Annual International Workshop on Selected Areas in Cryptography* (T. Lange, K. Lauter et P. Lisonek, éds.), LNCS, vol. 8282, Springer, 2014, p. 402–417.
✈ Cité page 27.

[BCGD+06] J. Buchmann, L. C. Coronado García, E. Dahmen, M. Döring et E. Klintsevich – « CMSS – an improved Merkle signature scheme », *Progress in Cryptography – INDOCRYPT 2006* (R. Barua et T. Lange, éds.), LNCS, vol. 4329, Springer, 2006, p. 349–363.
✈ Cité page 5.

[BCNS15] J. W. Bos, C. Costello, M. Naehrig et D. Stebila – « Post-quantum key exchange for the TLS protocol from the ring learning with errors problem », *2015 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 2015, p. 553–570.
✈ Cité page 43.

[BCO04] E. Brier, C. Clavier et F. Olivier – « Correlation power analysis with a leakage model », *Cryptographic Hardware and Embedded Systems – CHES 2004* (M. Joye et J.-J. Quisquater, éds.), LNCS, vol. 3156, Springer Berlin Heidelberg, 2004, p. 16–29.
✈ Cité page 1.

[BDH11] J. Buchmann, E. Dahmen et A. Hülsing – « XMSS - a practical forward secure signature scheme based on minimal security assumptions », *Post-Quantum Cryptography* (B.-Y. Yang, éd.), LNCS, vol. 7071, Springer, 2011,

p. 117–129.
✈ Cité pages 5 et 9.

[BDK+07] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya et C. Vuillaume – « Merkle signatures with virtually unlimited signature capacity », *Applied Cryptography and Network Security* (J. Katz et M. Yung, éds.), LNCS, vol. 4521, Springer, 2007, p. 31–45.
✈ Cité pages 5 et 9.

[BDL01] D. Boneh, R. A. DeMillo et R. J. Lipton – « On the importance of eliminating errors in cryptographic computations », *Journal of Cryptology* **14** (2001), no. 2, p. 101–119.
✈ Cité page 2.

[Ber05] D. J. Bernstein – « Cache-timings attacks on AES », 2005, http ://cr.yp.to/antiforgery/cachetiming-20050414.pdf [consulté le 14 août 2017].
✈ Cité page 32.

[BH16] L. G. Bruinderink et A. Hülsing – « "Oops, I did it again" – Security of one-time signatures under two-message attacks », Cryptology ePrint Archive, rapport 2016/1042, 2016, http://eprint.iacr.org/2016/1042 [consulté le 3 mai 2017].
✈ Cité page 24.

[BHH+14] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe et Z. Wilcox-O'Hearn – Implémentation de SPHINCS, 2014, disponible à l'adresse http://sphincs.cr.yp.to/software.html [consulté le 10 avril 2017].
✈ Cité pages 5, 14, 15, 19 et 23.

[BHH+15] — , « SPHINCS : practical stateless hash-based signatures », *Advances in Cryptology EUROCRYPT 2015 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques* (E. Oswald et M. Fischlin, éds.), LNCS, vol. 9056, Springer, 2015, p. 368–397.
✈ Cité pages 2, 5, 6, 15, 23 et 24.

[BHLV17] D. J. Bernstein, N. Heninger, P. Lou et L. Valenta – « Post-quantum rsa », Cryptology ePrint Archive, rapport 2017/351, 2017, http://eprint.iacr.org/2017/351.
✈ Cité page 2.

[BHLY16] L. G. Bruinderink, A. Hülsing, T. Lange et Y. Yarom – « Flush, Gauss, and reload - A cache attack on the BLISS lattice-based signature scheme », *Cryptographic Hardware and Embedded Systems – CHES 2016* (B. Gierlichs et A. Y. Poschmann, éds.), LNCS, vol. 9813, Springer, 2016, p. 323–345.
✈ Cité pages 2 et 43.

[BHT98] G. Brassard, P. Høyer et A. Tapp – « Quantum cryptanalysis of hash and claw-free functions », *LATIN'98 : Theoretical Informatics* (C. L. Lucchesi et A. V. Moura, éds.), LNCS, vol. 1380, Springer, 1998.
✈ Cité page 9.

[BMM00] I. Biehl, B. Meyer et V. Müller – « Differential fault attacks on elliptic curve cryptosystems », *Advances in Cryptology – CRYPTO 2000* (M. Bellare, éd.), LNCS, vol. 1880, Springer, 2000, p. 131–146.
✈ Cité page 2.

[Boy10] X. Boyen – « Lattice mixing and vanishing trapdoors : A framework for fully secure short signatures and more », *PKC 2010 : 13th International Conference on Theory and Practice of Public Key Cryptography* (P. Q. Nguyen et D. Pointcheval, éds.), LNCS, vol. 6056, Springer, mai 2010, p. 499–517.
✈ Cité pages 2 et 27.

[CFS01] N. Courtois, M. Finiasz et N. Sendrier – « How to achieve a McEliece-based digital signature scheme », *Advances in Cryptology – ASIACRYPT 2001* (C. Boyd, éd.), LNCS, vol. 2248, Springer, 2001, p. 157–174.
✈ Cité page 5.

[DDLL13] L. Ducas, A. Durmus, T. Lepoint et V. Lyubashevsky – « Lattice signatures and bimodal gaussians », *CRYPTO 2013, Part I* (R. Canetti et J. A. Garay, éds.), LNCS, vol. 8042, Springer, 2013, p. 40–56.
✈ Cité pages 3, 5 et 27.

[DG14] N. C. Dwarakanath et S. D. Galbraith – « Sampling from discrete Gaussians for lattice-based cryptography », *Applicable Algebra in Engineering, Communications and Computing* **25** (2014), no. 3, p. 159–180.
✈ Cité pages 27 et 31.

[DOTV08] E. Dahmen, K. Okeya, T. Takagi et C. Vuillaume – « Digital signatures out of second-preimage resistant hash functions », *Post-Quantum Cryptography* (J. Buchmann et J. Ding, éds.), LNCS, vol. 5299, Springer, 2008, p. 109–123.
✈ Cité pages 5 et 9.

[EFGT17] T. Espitau, P.-A. Fouque, B. Gérard et M. Tibouchi – « Side-channel attacks on BLISS lattice-based signatures – exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers », Cryptology ePrint Archive, rapport 2017/583, 2017, http://eprint.iacr.org/2017/583.
✈ Cité pages 3 et 55.

[GKT10] C. Giraud, E. Knudsen et M. Tunstall – « Improved fault analysis of signature schemes », *Smart Card Research and Advanced Application*, Springer, 2010, p. 164–181.
✈ Cité page 2.

[Gol86] O. Goldreich – « Two remarks concerning the Goldwasser-Micali-Rivest signature scheme », *Advances in Cryptology - CRYPTO '86* (A. M. Odlyzko, éd.), LNCS, vol. 263, Springer, 1986, p. 104–110.
✈ Cité pages 5, 6 et 7.

[GPV08] C. Gentry, C. Peikert et V. Vaikuntanathan – « Trapdoors for hard lattices and new cryptographic constructions », *40th ACM STOC* (R. E. Ladner et C. Dwork, éds.), ACM Press, mai 2008, p. 197–206.
✈ Cité pages 2, 27 et 33.

[Gro96] L. K. Grover – « A fast quantum mechanical algorithm for database search », *Proceedings of the 28th annual ACM Symposium on Theory of Computing*, 1996, p. 212–219.
✈ Cité page 9.

[GRSZ14] P. Gaborit, O. Ruatta, J. Schrek et G. Zémor – « RankSign : an efficient signature algorithm based on rank metric », *PQCrypto 2014* (M. Mosca, éd.), LNCS, vol. 8772, Springer, Heidelberg, 2014, p. 88–107.
✈ Cité pages 3 et 5.

[HRB13] A. Hülsing, L. Rausch et J. Buchmann – « Optimal Parameters for XMSS^{MT} », *Security Engineering and Intelligence Informatics* (A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl et L. Xu, éds.), LNCS, vol. 8128, Springer, 2013, p. 194–208.
✈ Cité pages 5, 9 et 11.

[HRG+15] P. Hodgers, F. Regazzoni, R. Gilmore, C. Moora et T. Oder – « State-of-the-art in physical side-channel attacks and resistant technologies », Tech. report, SAFECrypto, 2015, disponible sur https ://www.safecrypto.eu/wp-content/uploads/2015/02/SAFEcrypto_D7.1-Approved.pdf [consulté le 28 août 2017].
✈ Cité page 2.

[HRS16a] A. Hülsing, J. Rijneveld et P. Schwabe – « ARMed SPHINCS computing a 41KB signature in 16KB of RAM », *Public-Key Cryptography - PKC 2016 : 19th IACR International Conference on Practice and Theory in Pulbic-Key Cryptography* (Taipei, Taïwan) (C.-M. Cheng, K.-M. Chung, G. Persiano et B.-Y. Yang, éds.), LNCS, vol. 9614, Springer, 2016, p. 446–470.
✈ Cité pages 5 et 25.

[HRS16b] A. Hülsing, J. Rijneveld et F. Song – « Mitigating multi-target attacks in hash-based signatures », *Public-Key Cryptography - PKC 2016* (C.-M. Cheng, K.-M. Chung, G. Persiano et B.-Y. Yang, éds.), LNCS, vol. 9614, Springer, 2016.
✈ Cité page 14.

[Hül13] A. Hülsing – « W-OTS+ – shorter signatures for hash-based signature schemes », *Progress in Cryptology - AFRICACRYPT 2013* (A. Youssef,

A. Nitaj et A.-E. Hassanien, éds.), LNCS, vol. 7918, Springer, 2013, p. 173–188.
✈ Cité page 10.

[Kar16] C. F. F. Karney – « Sampling Exactly from the Normal Distribution », *ACM Transactions on Mathematical Software* **42** (2016), no. 1, article n° 3.
✈ Cité page 27.

[KJJ99] P. Kocher, J. Jaffe et B. Jun – « Differential power analysis », *Advances in Cryptology (CRYPTO '99), 19th Annual Internation Cryptology Conference* (Santa Barbara, Californie, États-Unis) (M. J. Wiener, éd.), LNCS, vol. 1666, Springer, 1999, p. 388–397.
✈ Cité pages 1 et 2.

[Kle00] P. N. Klein – « Finding the closest lattice vector when it's unusually close », *11th Annual ACM-SIAM Symposium on Discrete Algorithms* (D. B. Shmoys, éd.), ACM-SIAM, janvier 2000, p. 937–941.
✈ Cité page 27.

[Koc96] P. Kocher – « Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems », *Advances in Cryptology (CRYPTO '96)* (N. Koblitz, éd.), LNCS, vol. 1109, Springer, 1996, p. 104–113.
✈ Cité pages 1 et 2.

[Mer90] R. C. Merkle – « A certified digital signature », *Advances in Cryptology — CRYPTO' 89* (New York, NY) (G. Brassard, éd.), LNCS, vol. 435, Springer New York, 1990, p. 218–238.
✈ Cité page 6.

[MKAA16] M. Mozaffari-Kermani, R. Azarderakhsh et A. Aghaie – « Fault detection architectures for post-quantum cryptographic stateless hash-based secure signatures benchmarcked on ASIC », *ACM Transactions on Embedded Computing Systems* **16** (2016), no. 2, article 59.
✈ Cité page 25.

[MR04] D. Micciancio et O. Regev – « Worst-case to average-case reductions based on Gaussian measures », *45th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 2004, p. 372–381.
✈ Cité page 28.

[MR07] — , « Worst-case to average-case reductions based on Gaussian measures », *SIAM Journal on Computing* **37** (2007), no. 1, p. 267–302.
✈ Cité page 27.

[MW17] D. Micciancio et M. Walter – « Gaussian sampling over the integers : Efficient, generic, constant-time », Cryptology ePrint Archive, rapport 2017/259, 2017, http ://eprint.iacr.org/2017/259.
✈ Cité pages 27, 42 et 43.

[Pad99] H. Padé – « Mémoire sur les développements en fractions continues de la fonction exponentielle », *Annales scientifiques de l'École normale supérieure* **16** (1899), p. 395–426.
✈ Cité page 32.

[Pei10] C. Peikert – « An efficient and parallel Gaussian sampler for lattices », *Advances in Cryptology – CRYPTO-2010* (T. Rabin, éd.), LNCS, vol. 6223, Springer, août 2010, p. 80–97.
✈ Cité page 27.

[PPM17] R. Primas, P. Pessl et S. Mangard – « Single-trace side-channel attacks on masked lattice-based encryption », Cryptology ePrint Archive, rapport 2017/594, 2017, http://eprint.iacr.org/2017/594.
✈ Cité page 2.

[Pre17] T. Prest – « Sharper bounds in lattice-based cryptography using Rényi divergence », Cryptology ePrint Archive, rapport 2017/480, 2017, http://eprint.iacr.org/2017/480.
✈ Cité pages 3, 28, 30, 32 et 35.

[QS01] J.-J. Quisquater et D. Samyde – « ElectroMagnetic Analysis (EMA) : Measures and Countermeasures for Smart Cards », *Smart Card Programming and Security* (I. Attali et T. Jensen, éds.), LNCS, vol. 2140, Springer Berlin Heidelberg, 2001.
✈ Cité page 2.

[Rom90] J. Rompel – « One-way functions are necessary and sufficient for secure signatures », *22th Annual ACM Symposium on Theory of Computing*, ACM Press, 1990, p. 387–394.
✈ Cité page 5.

[RR02] L. Reyzin et N. Reyzin – « Better than BiBa : Short one-time signatures with fast signing and verifying », *Information Security and Privacy 2002* (L. Batten et J. Seberry, éds.), LNCS, vol. 2384, Springer, 2002, p. 1–47.
✈ Cité page 11.

[SBK+17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini et Y. Markov – « The first collision for full SHA-1 », *Advances in Cryptology – CRYPTO 2017*, LNCS, Springer, 2017, p. 570–596.
✈ Cité page 25.

[Sho94] P. W. Shor – « Algorithms for quantum computation : Discrete logarithms and factoring », *35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 1994, p. 124–134.
✈ Cité page 2.

[vN51] J. von Neumann – *Various techniques used in connection with random digits. Monte Carlo methods*, National Bureau of Standards, 1951.
✈ Cité page 29.

Annexe A

Implémentation initiale du GPA

```
/*
 * rng_uint64() échantillonne U([0,2^64 - 1]) sur les entiers.
 * rng_uint64b(m,n) renvoie un entier aléatoire compris entre m et n, n exclu.
 * rng_bool() renvoie un octet nul avec probabilité 1/2, sinon un octet non-nul.
 * HalfGaussCoDF et exp_tables sont des tables de valeurs entières précalculées.
 * SIGMA0 est la valeur fixée de \sigma_0.
 */

int64_t CoDFSampler() {
    int64_t i = 0;
    uint64_t u = rng_uint64();
    while(u > HalfGaussCoDF[i]) {
        u = rng_uint64();
        i++;
    }
    return i;
}

uint8_t SmallBerExp(double x) {
    uint64_t u = (uint64_t)round((~0ull)*x);
    uint64_t p, q, ps, qs;
    p = exp_tables[0][u >> 56];
    for (int i = 2; i < 5; i++) {
        q = exp_tables[i-1][(u >> (64 - (i << 3))) & 0xff];
        ps = p >> 32;
        qs = q >> 32;
        p = ps*qs + (((p & 0xffffffff)*qs) >> 32) + (((q & 0xffffffff)*ps) >> 32);
    }
    return(rng_uint64() < p);
}

uint8_t BerExp(double x) {
    uint64_t q = (uint64_t)(floor(x/LOG2));
    uint8_t b = rng_uint64b(0,(1<<q))?0:1;
```

```
    return SmallBerExp(x-(double)q*LOG2) & b;
}

int64_t SamplerZ(double mu, double sigma) {
    /*
    * mu peut être un réel quelconque.
    */
    int64_t z, q = (int64_t)(floor(mu));
    double x, r = mu - (double)q;
    uint8_t b;
    while(1) {
        z = CoDFSampler();
        b = rng_bool() ? 0 : 1;
        z = z + b - 2*z*(1-b);
        x = (double)(z-r)*(z-r)/(2*sigma*sigma) -
            (double)(z-b)*(z-b)/(2*SIGMA0*SIGMA0);
        b = BerExp(x);
        if (b) return z+q;
    }
}
```

Annexe B

Attaque par *branch-tracing* contre SamplerZ

Nous avons mené contre l'implémentation de SamplerZ fournie à l'annexe A une attaque par *branch-tracing*, sur le modèle de celle exposée par Espitau et coll. dans [EFGT17]. Cette attaque nous a livré toutes les sorties de CoDFSampler sur lesquelles BerExp a renvoyé 1.

Description de l'attaque. Il est expliqué dans [EFGT17, section 3.4] que les processeurs Intel récents sont équipés d'un instrument appelé *Branch Trace Store* qui enregistre diverses informations (dont les adresses d'origine et de destination) de toutes les instructions de branchement effectivement réalisées lors de l'exécution d'un processus. Sous Linux, cet instrument peut être utilisé sans droit administrateur pour enregistrer les traces laissées par un processus de l'utilisateur courant. Il suffit d'exécuter en ligne de commande l'une de ces deux instructions :

```
perf record -e branches:u -c 1 -d -p <PID>
perf record -e branches:u -c 1 -d ./<exec>
```

où `<PID>` est l'identifiant du processus visé et `<exec>` le nom de l'exécutable visé. Cette commande produit un (très gros) fichier binaire qui peut être interprété par la commande suivante :

```
perf script -F ip,addr
```

qui affiche des lignes de la forme :

```
adresse_origine => adresse_destination
```

ces adresses étant celles d'instructions de branchement. En particulier, le *branch-tracing* permet de savoir quand une fonction a été appelée, et par quelle fonction. C'est de cela dont nous allons nous servir.

Cadre et préparation de l'attaque. L'attaque a été menée sur un ordinateur fonctionnant sous Ubuntu 16.04 LTS et possédant un processeur Intel Core i3-6100 CPU qui dispose bien d'un *Branch Trace Store*. Elle peut être reproduite grâce aux fichers sources joints avec la version électronique de ce rapport (dossier `implems/branch-tracing`). Nous

avons attaqué l'exécutable issu de `brtracing.c` qui, en substance, demande une sortie de `SamplerZ` (dont l'implémentation est donnée en A) et l'affiche à l'écran.

L'idée de l'attaque est la suivante : d'une part, à chaque fois que `SamplerZ` rejette un échantillon de `CoDFSampler`, il le rappelle pour lui en demander un nouvel. Par conséquent, c'est l'échantillon livré par le dernier appel à `CoDFSampler` qui, au signe près, sera renvoyé par `SamplerZ`. C'est donc ce dernier appel que nous devons repérer et étudier. D'autre part, `CoDFSampler` fait appel $z+1$ fois à la fonction `rng_uint64`, où z est la valeur renvoyée par `CoDFSampler`. Il nous suffira donc de compter le nombre de fois où `rng_uint64` est appelé par le dernier `CoDFSampler` pour connaître, au signe près, la sortie de `SamplerZ`.

La première chose à faire est donc de récupérer les adresses des fonctions `CoDFSampler` et `rng_uint64`. Pour cela, nous avons désassemblé l'exécutable au moyen de la commande Linux `objdump -d <exec>`. Les lignes qui nous intéressent sont de cette forme :

```
0000000000401b35 <CoDFSampler>:
```

On apprend ainsi que l'adresse de `CoDFSampler` est 0x401b35 (le fichier produit par `perf script` ne fait pas figurer les 0 inutiles). Ces adresses sont reportées manuellement dans le fichier `br_tracing.py` dont le script recherche, à partir de ces adresses et du fichier obtenu de `perf script`, le dernier appel à `CoDFSampler` et compte le nombre d'appels à `rng_uint64` qu'il effectue. Il en déduit les deux valeurs possibles pour la sortie de `SamplerZ`.

Résultats. Le script `attaque-bt.sh` automatise les opérations de production et d'interprétation des données de `perf`. Son lancement affiche à l'écran une chaîne de la forme :

```
Sortie SamplerZ = -1
[ perf record: Woken up 4 times to write data ]
[ perf record: Captured and wrote 0.883 MB perf.data (19209 samples) ]
Sorties possibles: -1 et 2
```

c'est-à-dire que `SamplerZ` a en réalité renvoyé -1 et que, grâce au *branch-tracing*, on aurait pu, sans connaître cette valeur réelle, affirmer que `SamplerZ` avait renvoyé ou -1 ou 2 (c'est-à-dire que CoDFSampler a renvoyé 1). Cette attaque permet donc de deviner avec probabilité $1/2$ la sortie de SamplerZ.

Elle semble toutefois d'intérêt assez théorique : la commande `perf` qu'elle utilise doit être appelée sur la même machine et par le même utilisateur que celui qui exécute le programme visé, ce qui la rend difficile à mettre en œuvre par un attaquant en pratique. Elle provoque de plus un ralentissement notable du système qui ne passe pas inaperçu de la victime. En revanche, dans les cas où elle serait envisageable, elle constitue une menace sérieuse, car il semble difficile de s'en prémunir. La solution la plus simple pour ce faire est peut-être d'empêcher l'accès à un tel instrument aux utilisateurs sans droits administrateur.