Matthieu Rambaud

Cryptographie appliquée, Télécom Paris

27/9/2023

No-face offering tokens (from studio Ghibli)

1 Objectif principal et introduction. **2**

2 Définitions générales et modèle **8**

3 Outils **14**

4 Cryptographie symétrique **20**
4.1 Message authentication code (MAC). 20
4.2 Symmetric encryption scheme 24
4.3 Authenticated symmetric encryption scheme 29

5 Cryptographie asymétrique **32**
5.1 Le authenticated key exchange de Diffie-Hellman entre peers pré-spécifiés 33
5.2 Public key encryption scheme (PKE) 34
5.3 (Digital) signature scheme 39

6 Attention au vocabulaire utilisé en pratique **42**

7 Key exchanges over insecure channels: TLS 1.3 & PKE-based **43**
7.1 Solution basique: signature-based Diffie-Hellman, et ses limitations 44

7.2 Deux anonymes ayant échangé une clé peuvent-ils se prouver leur identité? 50
7.3 Encryption-based KE (1RTT, confidential identification, post-specified peers) 53

8 Davantage de sécurité et d'anonymité **56**

9 Lightweight et embarqué. **57**
9.1 Attaques de type *side channel* 57

1 Objectif principal et introduction.

Un des objectifs du cours est de décrire des solutions permettant de communiquer de façon sécurisée via un réseau insécurisé.

1.1 Objectif principal: implémenter des secure channels.

On formalise cet objectif de la façon suivante. On considère deux machines, appelées Alice et Bob. Elles sont éloignées et n'ont jamais interagi avant. L'adversaire $\mathcal{A}$ connait leur état initial et les programmes qu'elles utilisent. Toutes les communications entre elles passent par un réseau contrôlé par l'adversaire $\mathcal{A}$. Un tel réseau est modélisé de la façon la plus pessimiste possible, sous le nom de *insecure channel*, comme formalisé dans la Definition 6. Notamment $\mathcal{A}$ peut lire tous les messages envoyés, les bloquer pendant une durée arbitraire, et envoyer à Bob ou Alice des messages de son choix comme s'ils avaient été envoyés depuis Alice ou Bob. L'objectif est d'implémenter un *secure channel* entre Alice et Bob, au sens de la Definition 1, uniquement à l'aide d'envois de messages sur le insecure channel. Plus précisément, un *algorithme* est une suite d'instructions faisables par une machine. Il prend en entrée des inputs à partir d'une interface utilisateur, appellée "API". Il renvoie des outputs sur une autre interface utilisateur, appelée aussi API. Un *protocole* est un algorithme qui comporte, en plus, des instructions consistant à envoyer ou recevoir des messages sur un channel (insecure, ou authenticated Definition 7).

Définition 1 ((vanilla) Secure channel unidirectionnel)**.** On dit qu'un protocole entre Alice et Bob *implémente un secure channel unidirectionnel vanilla* s'il offre les API suivantes. Alice a une API Send et Bob une API $\mathsf{Receive}$. Elles apportent les mêmes garanties que la ressource idéalisée suivante. Send prend en input des chaînes de bits, appelées *plaintexts*. Lorsqu'un plaintext P est mis en input de Send, l'adversaire $\mathcal{A}$ en est informé, et est informé de sa longueur $|P|$. Puis, au bout d'un certain temps contrôlé par l'adversaire $\mathcal{A}$, à préciser, P est téléporté en output de l'API $\mathsf{Receive}$. Ĉes garanties doivent être vérifiées *même si l'adversaire $\mathcal{A}$ a un accès arbitraire à ces API.*

La Definition 1 implique que le protocole apporte de nombreuses garanties. On les détaille dans Sections 1.3 to 1.5. Ces paragraphes donnent des exemples qui montrent qu'il est facile de faire, sans s'en rendre compte, des protocoles qui ne les respectent pas. Comme décrit dans Section 5.2.1 et prouvé dans [CDN15, pp. 4.2.2–4.2.6], il est possible d'implémenter correctement un secure channel d'une façon relativement simple mais inefficace, à partir des deux ingrédients suivants. Le premier est un canal de communication public qui garantit que les interlocuteurs restent les mêmes tout au long de la discussion (Definition 7), avec en option la possibilité qu'ils s'identifient l'un l'autre (à préciser). Par exemple, se parler dans un lieu public. Le deuxième consiste en des calculs, connus sous le nom de *public key encryption*, dont le principe remonte aux années 70. Avec les mêmes outils, ou, plus efficacement avec d'autres ingrédients (Section 4.3 "authenticated symmetric encryption" et Section 7 "authenticated key exchange"), il est pos-

sible d'implémenter des secure channels, en sens inverse l'un de l'autre, avec la garantie supplémentaire que leurs extrémités arrivent aux mêmes endroits.

Définition 2 (Secure channel vanilla (bidirectionnel) de Alice vers Bob ([BS23, §21.9.1])). C'est un protocole qui implémente *à la fois* un (vanilla) secure channel undirectionnel de Alice vers Bob: ($\mathsf{Send}_{A\to B}$, $\mathsf{Receive}_{A\to B}$) et un autre de Bob vers Alice: ($\mathsf{Send}_{B\to A}$, $\mathsf{Receive}_{B\to A}$).

1.2 Les métadonnées sont rendues publiques, contre-mesures ?

Considérons d'abord pour simplifier un adversaire $\mathcal{A}$ qui n'a aucun accès aux API. Alors, le secure channel révèle quand même à l'adversaire la taille de tous les plaintexts envoyés, le moment de l'envoi, et les machines expéditrice et réceptrice. En ce sens, il est *semi confidentiel*.

Garder confidentielles la taille et la date. Une méthode simple mais coûteuse pour les cacher, consiste à envoyer un flux continu de plaintexts de taille fixe. Pour envoyer de l'information utile, la mettre dans ce flux de plaintexts, et le reste du temps, écrire la même phrase en boucle dans ces plaintexts.

Garder confidentiels qui est l'émetteur et le récepteur On verra dans Section 7.2 une nouveauté technique dans le protocole de secure channel appelé "TLS 1.3", qui apporte la garantie suivante. Il s'agit que le *contenu* des messages, envoyés sur le insecure channel par Alice et Bob, lorsqu'ils implémentent un secure channel entre eux avec TLS 1.3, n'apprend aucune information à l'adversaire $\mathcal{A}$ sur le fait que leurs expéditeurs sont Alice et Bob. On verra dans Section 7.3.3 que cette même garantie peut être apportée par de possibles remplaçants de TLS 1.3 compatibles aux futurs standards. Sans ingrédient supplémentaire, cette garantie n'apporte rien puisque le insecure channel révèle à $\mathcal{A}$ les machines expéditrice et réceptrice des messages, i.e., Alice et Bob. *Supposons* en outre qu'Alice arrive à passer par une machine "faux-nez", P, d'apparence anodine, pour envoyer et recevoir ses messages, alors il en résulte que TLS 1.3 garantit l'anonymité d'Alice. Un exemple simple de tel faux-nez est appelé "serveur DNS" (Exercice 53). Mais, comme remarqué dans [BS23, §21.10 p900], Bob ne peut a priori pas utiliser cette solution pour cacher son identité s'il est un serveur web. En effet, dans la réalité, les messages des clients de Bob, par exemple Alice, doivent contenir "Bob" en en-tête (techniquement: le *server name indication*, ou SNI, de Bob). Enfin, comment implémenter un faux-nez qui ne sache pas, lui-même, que c'est Alice qui se cache derrière (au cas où il souhaiterait vendre cette information à l'adversaire)? Dans Section 8.0.3 on verra une technique répondant à ces deux problèmes, et implémentée sous le nom de "the onion routing (TOR)". Le nom "onion" peut se voir comme le fait qu'Alice superpose des faux-nez, tel que chacun ne connaisse que ses voisins immédiats.

1.3 Les utilisations passées et futures n'enlèvent rien à la confidentialité

On considère maintenant un adversaire $\mathcal{A}$ qui a un accès arbitraire aux API. Cet accès modélise les liens (influence ou connaissance) que $\mathcal{A}$ peut avoir sur les *utilisateurs* des APIs des machines Alice et Bob. Dans ce cours on ne jugera pas ces liens, on les modélisera simplement comme un accès de $\mathcal{A}$ aux API. Qu'est-ce que la Definition 2 peut bien garantir contre un tel adversaire ? Elle apporte par exemple la garantie modélisée par le scénario suivant, que l'on pourrait qualifier de "crash-test", appelé formellement "security game" ("semantic" ou "ind-CPA". On considère un adversaire $\mathcal{A}$ qui a eu jusqu'à maintenant un arbitraire aux API d'envoi et de réception. Puis, il tourne le dos pendant qu'un utilisateur utilise Alice pour Send un plaintext P, et qu'un utilisateur Receive P sur Bob. Puis, $\mathcal{A}$ reprend exclusivement la main sur les API, sans

interagir avec les autres utilisateurs. Alors, $\mathcal{A}$ n'en apprendra jamais plus sur P qu'il n'en savait déjà. En creux, on voit qu'un secure channel a un comportement idéal en tant que ressource, mais qu'il ne protège pas contre eux-mêmes des utilisateurs influençables par l'adversaire, ou contre elles-mêmes des machines qui donneraient accès à plus que leurs API (voir Section 1.8). Le communication channel des Japonais pendant WWII n'apportait pas cette garantie, comme le montre cet exemple ([Ole21]): "In the spring of 1942, Japanese intercepts began to make references to a pending operation in which the objective was designated as 'AF'. Rochefort and Captain Layton believed 'AF' might be Midway since they had seen 'A' designators assigned to locations in the Hawaiian Islands. [...] How to convince the doubters in Washington? Via submarine cable, which could not be intercepted, Honolulu sent a message to Midway instructing it to radio Pearl Harbor in the clear that the salt-water evaporators on the base had broken down. Two days later, on May 22, a Japanese message was intercepted that reported "AF" was running out of fresh drinking water." En effet cet exemple se formalise comme le fait que les Américains avaient le dos tourné pendant qu'un plaintext P a été envoyé sur l'API d'envoi des Japonais. Puis à un moment les Américains ont eu un accès à l'API d'envoi des Japonais. Concrètement, ils se sont débrouillés pour que le plaintext "Midway", choisi par eux mêmes, soit mis en input de Send. *Juste avant l'input à* Send, ils n'étaient pas encore certains du contenu de P. Alors que *juste après cet input à* Send, ils ont obtenu l'information que le contenu de P était égal à "Midway". Conclusion: c'est bien le *channel* des Japonais a leaké l'information sur P. Précisément, cette information leur a été donnée par un pur effet de l'*implémentation* de l'API d'envoi sur channel, qui a consisté en l'émission d'un message public. On est donc bien dans les conditions du *semantic security game*. Conclusion: le protocole des Japonais n'implémentait pas un secure channel. En fait, cette faille est un exemple très général. Techniquement, avec le vocabulaire de Section 4.2.1, l'Exercice 28 montre qu'un algorithme qui vérifie la spécification appelée *symmetric encryption*, est forcément *randomisé*, car sinon il ne pourrait pas résister au semantic security game (on appelle *secrecy* la garantie d'avoir cette résistance). Or celui des Japonais, appelé "JN-25", était au contraire déterministe.

1.4 Authentication encore meilleure qu'authenticated: non-identity-misbinding.

Un secure channel apporte une garantie dite *d'authentication*. C'est à dire que, pour peu que Alice ait la certitude l'autre extrémité de son secure channel soit bien reliée à Cyrano (en anticipant sur Sections 1.6 and 1.7), alors, elle a la garantie que tout ce qu'elle reçoit sur ce channel a bien été émis par Cyrano. Un channel qui n'aurait que cette garantie, mais sans confidentialité, s'appelle *authenticated channel* (Section 2.3). De façon non-intuitive, il y a pourtant une garantie d'authentication supplémentaire qui est apportée par la spécification de secure channel Definition 1, mais pas par un authenticated channel. Il s'agit de la résistance à une catégorie d'attaques couramment appelée *identity-misbinding* ([Kra03] et [BS23, §21.2.1]) ou *source-substitution* ([MVV96]). Prenons l'exemple d'une lettre manuscrite $\widetilde{m}$ d'un individu appelé Cyrano. Alors, il ne fait aucun doute que l'*objet* $\widetilde{m}$ a été produit par Cyrano (tâches d'encre laissées par le nez sur la feuille etc.). Appelons m le texte de cette lettre, formellement: la donnée utile. Alors, l'envoi de cette lettre $\widetilde{m}$ de Cyrano à Alice implémente un authenticated channel. C'est à dire, que, quand Alice reçoit $\widetilde{m}$, elle a la garantie que, à un moment donné, Cyrano a mis m en input de sa plume. Mais si cette lettre $\widetilde{m}$ est publique, alors en elle-même, elle ne prouve pas que Cyrano soit l'*auteur* de m. En effet, n'importe quel Christian lisant $\widetilde{m}$ pourrait la recopier. C'est à dire: mettre m en input de sa plume, et produire une lettre manuscrite $\overline{m}$. L'envoi de $\overline{m}$ de Christian à Alice constitue tout aurant un authenticated channel. C'est à dire, que, quand Alice reçoit $\overline{m}$, elle a la garantie que, à un moment donné, Christian a mis m en input de sa plume.

Par contre, si Cyrano transmettait la donnée m à Alice par *un protocole de secure channel*, alors on aurait la garantie supplémentaire suivante, appelée *non-identity-misbinding*. Dans notre contexte, elle apporte

notamment la garantie suivante dans le scénario suivant. Considérons une machine Christian qui est en fait l'adversaire $\mathcal{A}$, qui contrôle le insecure channel. Alice est une machine qui exécute un protocole de secure channel, avec Christian comme correspondant (au sens précis de Sections 1.6 and 1.7), sur ce insecure channel. Notamment, Alice a une API $\mathsf{Receive}_C$ dédiée à Christian. On considère Cyrano et Alice aussi reliés par un protocole de secure channel. On suppose que Cyrano est capable de générer un certain type de données, comme m, que Christian est incapable de générer. On suppose en outre que ni Cyrano ni Alice n'aient de lien avec Christian, en dehors de leurs envois-réceptions sur le insecure channel. Alors, il est garanti que Alice n'output jamais une donnée de type m sur son API $\mathsf{Receive}_C$. On ne formalisera pas davantage cette garantie, puisque c'est sous-produit de la Definition 1. Par contre, on donnera des exemples de méthodes naives qui échouent à apporter cette garantie: Exercice 41 (Encrypt-then-authenticate), ou Exercice 45 (Sign-then-Encrypt). On évoquera dans Section 5.2.5 et Section 5.3.2 comment les réparer, mais avec un outil hors-programme (ind-CPA public-key encryption). Heureusement, les méthodes de Section 7 sont à la fois plus efficaces et n'ont pas besoin de cet outil.

1.5 Conservation des propriétés pour des exécutions en concurrence ou en sous-routine

La Definition 1 est une simplification informelle de *implémenter la fonctionalité idéale de secure message transfer*, au sens de [Can01] et, implicitement, de [BS23, §13.7.2]. Comme prouvé dans [CDN15, p. 4.2.7], la Definition 1 implique la garantie que les protocoles conservent leurs spécifications mêmes lorsqu'ils sont *composés* à d'autres protocoles, e.g., en *concurrence* ou appelés en *sous-routine* par des protocoles de plus haut niveau, etc.. Des protocoles qui perdraient leurs spécifications lorsqu'ils ne sont plus exécutés en isolation, s'exposent à des *cross-protocol attacks* (comme [Bri+21] sur TLS). L'exemple décrit dans Section 5.2.5 montre qu'il est facile de se tromper et d'implémenter un secure channel qui fonctionne en isolation, mais qui échoue lorsqu'il y a plusieurs instances en parallèle. Une subtilité avec les garanties de composabilité sont que, d'une certaine manière, les spécifications de sécurité des protocoles de haut niveau, tolèrent en fait un adversaire moins intrusif que les spécifications de sécurité individuelles de ses composantes. L'Exercice 27 en donne un exemple, commenté juste après en Item (iv).

1.6 Pourquoi il est en fait dangereux d'utiliser de simples vanilla secure channels

Supposons que toutes les machines du monde soient reliées deux à deux par des vanilla secure channels. Ces channels vérifient simplement la Definition 2, en particulier, n'apportent aucune garantie d'identité de la machine correspondante (en un sens à définir). Ils ne garantissent que l'unicité du correspondant, et la confidentialité. Ces channels n'ont pas de dénomination globale. Dit autrement, la Definition 2 ne fournit aucun signe distinctif qui dirait quelle est l'API Send, parmi toutes celles d'Alice, est reliée par un channel à quelle API Receive, parmi toutes celles de Bob. Pour forcer le trait, supposons que les deux machines Alice et Bob soient côte à côte dans la même pièce, et qu'un *même* utilisateur ait accès à toutes leur API Send et Receive. Faisons l'hypothèse que cet utilisateur utilise sur Alice l'API d'un secure channel bidirectionnel qui se présente sous le nom de sc, et sur Bob l'API d'un secure channel bidirectionnel qui se présente sous le nom de sc′. Supposons que ces deux API affichent le comportement prévu par la Definition 2, plus particulièrement, tout message $\mathsf{Send}_{\mathsf{sc}}$ chez Alice s'affiche sur l'interface $\mathsf{Receive}_{\mathsf{sc}'}$ chez Bob, et inversement. Alors, avec uniquement ces API en boîte noire, il n'existe *aucune* méthode permettant à l'utilisateur de s'assurer que sc et sc′ sont bien les extrémités du même secure channel. Les temps de communication ne sont pas exploitables car l'adversaire a un contrôle total sur le réseau. L'utilisateur ne pourra jamais exclure qu'il soit en fait dans la situation suivante, appelée *Man-in-the-middle*:

$$\text{Alice} \xleftrightarrow{\mathsf{sc}} \mathcal{A} \xleftrightarrow{\mathsf{sc}'} \text{Bob} \quad . \tag{1}$$

où son correspondant au bout de sc est en fait l'adversaire $\mathcal{A}$, lui-même relié à Bob via sc', et $\mathcal{A}$ relaie tous les plaintexts envoyés entre Alice et Bob. Un scénario classique est celui où $\mathcal{A}$ se fait passer pour le point d'accès au wifi Bob [Fen+23]. A priori TLS 1.2 évitait cette incertitude avec une méthode d'attribution implicite d'un *identifiant global* (ID), a priori distinct, à chaque secure channel créé sur terre. Une telle méthode d'attribution s'appelle un *channel binding* (Section 7.2.1, [BS23, §21.8]). Mais l'attaque de [Bha+14] avait provoqué la situation catastrophe que les deux channels sc et sc' avaient le même ID. L'attaque dite "triple handshake" de [Bha+14] contre TLS 1.2 met Alice (un serveur) dans une situation où elle a un secure channel sc avec un correspondant C (un "client"), contrôlé par $\mathcal{A}$, tandis que le vrai client Bob était relié à C par un channel avec le même ID:

$$\text{Alice} \overset{\mathsf{sc}}{\longleftrightarrow} C \overset{\mathsf{sc}}{\longleftrightarrow} \text{Bob} \quad .$$

//Techniquement, $\mathcal{A}$ avait déclenché une reprise de connexion sur les deux channels (une *session resumption*). Suite à la reprise, TLS 1.2 avait attribué le même ID aux deux. Techniquement, ce nouvel ID était défini comme une donnée appelée "verify_data", qui se trouvait être égale sur les deux channels TLS 1.3 a corrigé ce problème en quelque sorte en suivant partiellement [BPR00, Remark 1], c'est à dire en définissant l'ID comme égal au hash de tous les messages échangés pendant la construction du secure channel. Ainsi, Alice et Bob étaient en quelque sorte tombés d'accord sur le fait qu'ils communiquaient sur un secure channel ayant le même ID, et en avaient faussement déduit qu'ils étaient les correspondants respectifs l'un de l'autre.
En fait, la situation de (1) était celle d'un *insecure channel* entre Alice et Bob. On va voir que ce support permet à Alice et Bob de construire un secure channel entre eux, qu'on peut voir de façon imagée comme un "tunnel". C'est exactement ce que la victime Alice a fait avec Bob à la fin de l'attaque. Techniquement, Alice a fait une *renégociation de connexion* avec Bob. Cela n'a pas empêché que, même après cette renégociation, elle a (au moins temporairement) attribué à Bob tous les plaintexts qu'elle avait reçus sur le secure channel sc, donc envoyés par $\mathcal{A}$, avant cette renégociation. C'était donc une situation de *identity-misbinding*, qui est interdite par les spécifications (Section 1.4), donc cela prouve que TLS 1.2 échouait à implémenter des secure channels.

1.7 Comment distinguer, à distance, deux machines identiques?

Ce que montre l'exemple au début de Section 1.6, c'est qu'on a besoin d'un ingrédient *en plus* qu'un secure channel, permettant à une machine, Alice, de distinguer *à distance* si son secure channel est bien relié à Bob. Le problème n'est pas simple puisque notre modèle suppose que l'adversaire $\mathcal{A}$ a a priori le pouvoir de faire l'attaque Man in the middle (Equation (1)). En effet, il a une connaissance parfaite des machines et des logiciels utilisés, en possède des copies, etc. Il est avéré qu'il est faisable d'implémenter un tel adversaire en pratique. Il est commun de trouver des copies identiques de serveurs, permettant du fishing. Également, l'attaque [Ber05] crée une copie identique d'un serveur victime et de son implémentéation d'un circuit cryptographique (AES), qui a les mêmes temps de calcul que la victime, au 1/10 de milliseconde près. (Une exploitation de cette similarité lui a permis, simplement en mesurant des différences de temps d'exécution, d'apprendre à distance une information aussi fine que le n-ième byte secret mis en input d'une 128-lookup table par la victime: voir Section 9.1.1).
Nous verrons dans Section 5.1 une catégorie d'algorithmes, appelée *cryptographie asymétrique*, qui permettent, entre autres, de distinguer deux machines à distance. Elle consiste en des algorithmes, dont le point de départ est de faire générer à une machine une donnée secrète, connue de elle seule et appelée *clé secrète*. Cette donnée secrète, est en quelque sorte, la solution à un problème public, appelé *clé publique*, dont elle publie l'énoncé. Par définition, la machine ayant la connaissance de la clé secrète d'une clé publique est appelée le *owner* de la clé publique. La conséquence est que cela *brise la symétrie entre les machines*, elles

ne sont plus interchangeables. La capacité à résoudre le problème posé par la clé publique, est une façon d'*identifier* son owner. Par exemple sur les blockchains, la capacité à générer une signature reconnue comme valide pour une clé publique, est synonyme de *ownership* de l'identité définie par cette clé publique (et de l'argent qui va avec).

Définition 3 (Identités, authentication)**.** Soit id_Q une propriété, on dit qu'une machine Q a l'identité id_Q si elle a cette propriété.

Les deux exemples principaux d'identité sont: (i) soit une public verification key fixée vk, la propriété d'être capable de générer, *sur n'importe quelle donnée* m, une signature valide pour vk; (ii) connaître une clé secrète symétrique $\mathsf{k_m}$.

Définition 4. Une système de *(non-interactive) message authentication*, par rapport à une identité id_Q consiste en deux APIs. L'une, appelée Sign_{id_Q}, est accessible à toute machine possédant l'identité id_Q: elle prend en input n'importe quelle donnée m, et output une donnée $\widetilde{m}$ qu'on appelle *authenticated* m. La deuxième, appelée *vérification*, prend en input $(id_Q, \widetilde{m})$ et renvoie (id_Q, m) ***si et seulement si*** m ***a été mise dans le passé*** en input de l'API Sign_{id_Q}, et reject sinon. Cette garantie est appellée *unforgeability*. Il est imposé qu'elle tienne quand bien même l'adversaire aurait accès à l'API Sign_{id_Q}.

On peut donc maintenant préciser l'objectif sous une forme plus utile.

Définition 5 (Secure channel (uni- ou bi-directionnel) *avec identification one sided / two sided*)**.** On dit qu'un secure channel entre Alice et Bob (uni- ou bi-directionnel) apporte de *l'identification one-sided pour l'identité* id_Q si, en outre, il retourne l'output id_Q à Alice *si et seulement si* son correspondant (Bob) possède l'identité id_Q.
On dira qu'il apporte de l'*identification two-sided pour les identités* id_Q *et* id_P, ou *mutual*, si, en outre, il retourne l'output id_P à Bob *si et seulement si* son correspondant (Alice) possède l'identité id_P.

Dans TLS, one-sided s'appelle: connexion d'un client Alice anonyme au serveur Bob. Dans TLS, lorsque Alice avait commencé en anonyme puis continue en two-sided, cela s'appelle: une renégociation de la connexion du client Alice au serveur Bob. Cela permet à Bob d'associer à Alice toutes les actions que Alice avait effectuées lorsqu'elle était anonyme.
Les solutions de base que nous verrons dans Section 5.1 et Section 7.3.1 permettent à Alice d'établir un secure channel avec le owner d'une certaine identité id_Q. Appelons Bob ce dernier. Cette solution de base a l'inconvénient qu'elle nécessite qu'Alice connaisse a priori id_Q, et oblige Bob à révéler *publiquement* que le owner de id_Q va faire un secure channel avec Alice. On verra qu'en fait le protocole TLS 1.3 [IET18b][BS23, §21.10] *permet* à Bob de ne révéler id_Q qu'à son correspondant. Encore mieux: il permet à Alice de *ne pas* révéler son identité id_P à son correspondant, avant d'avoir reçu la confirmation que son correspondant a bien l'id_Q.

1.8 Liens maximaux tolérés avec l'adversaire pour y arriver, conséquences des spécifications

Les solutions pour implémenter un secure channel, ne fonctionnent que si les machines Alice et Bob n'ont pas plus de *liens* avec l'adversaire que ceux qu'on va spécifier. *Liens* signifient *influence* et/ou *leakage*, au sens suivant:

- on appelle ([Can01]) *influence* la capacité de l'adversaire $\mathcal{A}$ à modifier l'état interne, les inputs ou les outputs, d'une ressource ou bien d'une machine;

- on appelle *leakage* ([CDN15, §4]), ou *fuite*, toute information donnée à $\mathcal{A}$ sur l'état interne, d'une ressource ou bien d'une machine.

Dans ce cours on ne juge pas de l'origine ni des responsables d'une influence ou d'un leakage. Pour pouvoir obtenir leurs API, Alice et Bob doivent exécuter un protocole, appelé *handshake*. C'est un protocole, donc il peut donc être fait par Alice et Bob entièrement à distance. Les liens maximaux tolérés avec l'adversaire $\mathcal{A}$ sont les suivants [1]:

- $\mathcal{A}$ connait parfaitement l'état initial de Alice et de Bob (Section 2.6);
- à l'issue du handshake, $\mathcal{A}$ peut bénéficier en outre d'un accès illimité aux API d'envoi et de réception de Alice et Bob (Figure 4);
- enfin (Definition 6) l'adversaire a une vue et un contrôle total sur le insecure channel.

Tout autre canal de leakage ou d'influence que ceux ci-dessus, entre $\mathcal{A}$ et les machines Alice et Bob, est appelé un *side-channel*. Les spécifications ne tiennent pas pour une machine Alice qui subit un side-channel: on appelle *corrompue* une telle machine. On donne des exemples de side-channels dans Section 9.1, et d'attaques qui les exploitent. Une machine non corrompue est appelée *honnête*. Dans les exercices ou les exemples, lorsqu'on parle d'une machine *corrompue* sans plus de précisions, on sous-entend l'hypothèse plus pessimiste possible, i.e., qu'elle est complètement contrôlée par l'adversaire.

2 Définitions générales et modèle

On ne revient pas sur les définitions déjà données dans Section 1: *secure channel* sans ou avec authentication (Definitions 2 and 5), *influence*, *leakage* et *identités* (Definition 3).

2.1 Notations

Additions/soustractions modulo 2 = " $\oplus$ " = XOR entre bits. Un bit est un élément de $(\mathbb{Z}/2\mathbb{Z}, \underset{\text{mod } 2}{+})$. C'est l'ensemble $\{0,1\}$ avec la loi d'addition et de multiplication modulo 2. On rappelle que $\underset{\text{mod } 2}{-} = \underset{\text{mod } 2}{+}$.

(2) On appelle cette opération *XOR*, et on la note: " $\oplus$ " $:= \underset{\text{mod } 2}{+}$

Suites ou vecteurs de bits $u = [u_0, \ldots, u_{\beta-1}] \in \{1,0\}^\beta$. Cette notation signifie que u est une *suite*, aussi appelée *vecteur*, de β bits. La notation $u[i]$ désigne la coordonnée indexée par i: u_i. Donc il s'agit de la $(i+1)$-ème dans l'exemple précédent où les indices de u commencent à 0, et la i-ème si au contraire ils commencent à 1 ou si l'indexation n'est pas spécifiée. On appelle aussi β la *taille* du vecteur u. On note $\{0,1\}^\beta$ l'ensemble des vecteurs de β bits, et $\{1,0\}^*$ l'ensemble des vecteurs de taille quelconque.

Concaténation de chaînes de bits ou de caractères u et v on utilisera de façon interchangeable les deux notations possibles $u\|v$ et (u,v) pour désigner une concaténation. La deuxième notation signifie en plus être un vecteur, e.g., si $w = (u,v)$ alors $w[2] = v$.

[1] [BS23, §13.7.2] "In fact, we can just assume there is a single attacker who orchestrates the behavior of all the corrupt users and completely controls the network. Moreover, this attacker may have some knowledge of or influence over messages sent by honest users, and may have some knowledge of messages received by honest users"

Les vecteurs de $\beta = 64$ or 128 bits sont appelés "blocks". "Key size" de $\kappa = 128$ or 256 bits. Le blockcipher AES prend en input des suites de $\beta = 128$ bits, appelées "blocks". Le paramètre β est différent dans d'autres blockciphers, par exemple $\beta = 64$ pour "ChaCha20" ([IET18b, §9.1]).
On utilisera principalement des symmetric keys ($\mathsf{k}_{\mathsf{ed}} \in \{0,1\}^\kappa$, $\mathsf{k}_{\mathsf{m}} \in \{0,1\}^\kappa$) égales à des suites de bits. Leur longueur, κ est appelée *key size*, elle est typiquement de $\kappa = 128$ ou 256 bits, suivant le niveau de sécurité.

XOR $\oplus$ entre vecteurs de bits. Soient $u = [u_0, \ldots, u_{\beta-1}] \in \{1,0\}^\beta$ et $v = [v_0, \ldots, v_{\beta-1}] \in \{1,0\}^\beta$ deux vecteurs, ou suites, de β bits. On note:

$$\begin{vmatrix} u_0 \\ u_1 \\ \vdots \\ u_{\beta-1} \end{vmatrix} \oplus \begin{vmatrix} v_0 \\ v_1 \\ \vdots \\ v_{\beta-1} \end{vmatrix} := \begin{vmatrix} u_0 \oplus v_0 \\ u_1 \oplus v_1 \\ \vdots \\ u_{\beta-1} \oplus v_{\beta-1} \end{vmatrix} \tag{3}$$

le XOR coordonnée par coordonnée. C'est donc égal à leur XOR: $u \oplus v$ au sens usuel.

Développement binaire: $\{0,1\}^\beta \longleftrightarrow$ nombre dans $\mathbb{Z}/2^\beta\mathbb{Z}$. Soit $x = [x_0, \ldots, x_{\beta-1}] \in \{0,1\}^\beta$, il représente le nombre $\langle x \rangle := x_0 + 2.x_1 + \cdots + x_{\beta-1}.2^{\beta-1} \in \mathbb{Z}/2^\beta\mathbb{Z}$, où "+" désigne $\underset{\mathrm{mod}\ 2^\beta}{+}$ l'addition modulo 2^β. Réciproquement soit un nombre $y \in \mathbb{Z}/2^\beta\mathbb{Z}$, il est représenté par un unique *développement binaire* $[y_0, \ldots y_{\beta-1}] \in \{0,1\}^\beta$, c'est à dire tel que $y = y_0 + y_1.2 + \cdots + y_{\beta-1}.2^{\beta-1}$

Le nombre d'éléments dans un ensemble E est noté $|E|$. On appelle aussi $|E|$ le *cardinal* de E. On notera aussi parfois $|u|$ la taille d'un vecteur de bits u. Donc avec cette notation, $u \in \{0,1\}^{|u|}$.
$s \in \{0,1\}^\kappa$ bitstring of specified length signifie que s est une chaîne de bits de longueur κ.
$s \in \{0,1\}^*$ bitstring of unspecified length signifie que s est une chaîne de bits de longueur non spécifiée. Soit elle déterminée de façon implicite d'après le contexte, par exemple la longueur d'une donnée en input d'une fonction de hachage. Soit sa valeur est spécifiée d'avance, parmi plusieurs possibilités, par exemple 128 ou 256 pour la taille d'une clé symétrique, etc.
$r \overset{\$}{\leftarrow} E$ secure random generation signifie tirer un élément uniformément au hasard dans l'ensemble E. Pour l'implémenter, il faut soit tirer environ $\log(|E|)$ fois à pile ou face, soit utiliser une méthode approchée déterministe, mais *obligatoirement* parmi celles décrites dans Sections 3.4 and 3.5.

2.2 Insecure communication channel

La Definition 6 le modèle d'adversaire le plus courant, qui est celui d'un adversaire qui a une vue et un contrôle total sur le réseau. Dit autrement, on formalise le réseau comme une ressource d'envoi et de réception de messages, qui offre à l'adversaire une vue et un contrôle total. On trouve ce modèle sous différentes formes: dans [BR93][2], [BS23, §13.7.2] et [CK01] ("unauthenticated links adversarial model (UM)").

Définition 6 (Insecure communication channel)**.** C'est une ressource accessible par deux machines, Alice et Bob, qui leur permet d'envoyer et de recevoir des chaînes de bits, appelées *messages*, de l'un vers l'autre. L'adversaire $\mathcal{A}$ contrôle le channel, voit tous les messages envoyés sur le channel, peut en faire une copie, peut bloquer ou injecter des messages de son choix.

[2] In particular, the adversary can read the messages produced by the parties, provide messages of her own to them, modify messages before they reach their destination, and delay messages or replay them.

Pour pouvoir implémenter un secure channel qui délivrerait tous les plaintexts envoyés au maximum après un délai Δ, on aura besoin de l'hypothèse que $\mathcal{A}$ débloque les messages envoyés sur le insecure channel au maximum au bout de Δ. Si $\Delta = \infty$, cela veut dire que l'influence de $\mathcal{A}$ sur le insecure channel est totale. Par exemple, supposons que $\Delta = 1$ mois. Alice envoie $m = \texttt{tigre du Bengale}$ sur le channel. $\mathcal{A}$ bloque m. Il injecte les deux messages $m' = \texttt{tigre du}$ et $m'' = \texttt{Bengale}$, qui sont délivrés à la suite à Bob. *Aucun des deux* n'est le message m envoyé par Alice. Puis un mois plus tard, $\mathcal{A}$ débloque m et l'intercale entre deux autres messages $m_1 = \texttt{Le}$ et $m_2 = \texttt{tue le sambar.}$, qu'il injecte. Bob reçoit donc trois messages à la suite: m_1, m et m_2. En particulier il a reçu m le message envoyé, donc l'hypothèse de $\Delta = 1$ mois est respectée. //Concrètement, l'hypothèse que les messages seraient bloqués au maximum pendant $\Delta = 1$ mois, peut s'implémenter si Alice a les ressources nécessaires pour délivrer m à Bob en une semaine, et qu'elle mobilise des ressources de plus en plus importantes pour délivrer m tant qu'elle n'a pas eu d'accusé de réception. Il est possible d'implémenter cette preuve de réception avec les techniques d'*authentication* de messages, que l'on va décrire. ***La définition n'entraîne pas que $\mathcal{A}$ connaisse l'identité de Alice et Bob, par exemple il peut s'agir d'un canal de communication public anonyme.***

2.3 (Public) authenticated channel.

Un *public authenticated channel* est une ressource idéalisée intermédiaire entre insecure et secure channel. Il peut se voir comme un insecure channel sur lequel $\mathcal{A}$ ne pourrait pas injecter de messages //ni changer l'ordre de réception par Bob par rapport à l'ordre d'envoi des messages par Alice, idem pour l'autre sens Bob $\rightarrow$ Bob. Il peut aussi se voir comme un secure channel qui leake de surcroît le contenu des messages à $\mathcal{A}$, i.e., sans secrecy. On considère une paire de machines Alice et Bob, avec chacune une identité: id_P et id_Q. On considère l'un des deux cas de figure:

- $id_P = id_Q$ //par exemple: égale à la connaissance d'une clé secrète symétrique ssk et Alice et Bob sont les seuls owners de cette identité;
- ou $id_P \neq id_Q$, et Alice est le seul owner de id_P, idem pour Bob et id_Q//par exemple, id_P peut impliquer (i) la connaissance d'une secret signature key associée à une certaine public verification key vk, et/ou, (ii) dans AKE1 (Section 7.3.1): la connaissance d'une secret decryption key associée à une certaine ek.

Définition 7 ((Public) authenticated channel unidirectionnel/bidirectionnel, vanilla / avec one-sided / two-sided identification)**.** On dit que Alice et Bob *implémentent un vanilla public authenticated channel unidirectionnel* si elles offrent à leurs utilisateurs les mêmes API (Send chez Alice et Receive chez Bob) que dans la définition d'un secure channel. Les garanties sont les mêmes, sauf que le leakage est total. Concrètement, l'adversaire $\mathcal{A}$ est en outre informé du contenu de tout ce qui est mis en input de Send. C'est pourquoi on appellera ces inputs simplement des *messages* (et pas des plaintexts).
Les variantes avec identification one-sided / two-sided identities, sont les mêmes.

Concrètement, si Bob a accès à un vanilla authenticated channel anonyme et qu'il reçoit deux messages sur son API Receive, alors il a la garantie qu'ils ont été Sent par la machine à l'autre bout du channel. En outre, si le insecure channel avec lequel on va l'implémenter délivre tous les messages au bout d'un temps fini, alors $\mathsf{Receive}_{\mathrm{P}\rightarrow\mathrm{B}}$ délivrera *tous* les messages qui ont été $\mathsf{Send}_{\mathrm{A}\rightarrow\mathrm{Q}}$, chacun au bout d'un temps fini contrôlé par $\mathcal{A}$. Dans la version one-sided pour (Bob, id_Q), le channel garantit en outre à Alice que son correspondant a l'identité id_Q. C'est ce qui se produit dans TLS 1.3 quand un client, Alice, reste en mode anonyme vis à vis du serveur Bob.

2.4 Synchronous secure channel

Un upgrade de secure channel, ou de authenticated channel, s'appelle *synchronous* secure/authenticated, channel ([Can01]). Il garantit en plus que Bob a lu le dernier plaintext/message envoyé par Alice, lorsque Alice reçoit un nouveau plaintext/message de Bob (idem pour Bob $\rightarrow$ Alice). Il est facile de l'implémenter directement à partir d'un secure/authenticated channel, sans outils supplémentaires. Il suffit que Alice et Bob ajoutent des accusés de réception à la fin de leurs plaintexts/messages, attendent un accusé de réception de leur dernier plaintext/message envoyé avant de parler, et ignorent tout plaintext/message reçu ne contenant pas un accusé de réception de leur dernier plaintext/message envoyé.

2.5 Key exchange.

Définition 8 (Key exchange avec identification two-sided entre *pre*-specified peers)**.** C'est un protocole prévu pour être exécuté par deux machines avec la syntaxe et les garanties suivantes. Les identités des deux machines sont *pré-spécifiées*, par exemple Alice et Bob. Précisément, on appelle *session chez Alice avec peer Bob* une exécution du protocole faite par Alice avec comme paramètre (Alice,Bob). Un tel protocole, $\mathsf{KeyExch}$, doit garantir que:

Consistency si Alice et Bob sont honnêtes, i.e., n'ont pas de side-channel, et finissent tous les deux une session paramétrée (Alice,Bob), ils output tous les deux une même chaîne de bits ssk de longueur préspécifiée, appelée *(secret) symmetric session key*

Key indistinguishability elle est définie par le scénario suivant, qu'on pourrait appeler "crash-test", couramment appelé *security game*: soit une machine Alice honnête qui output ssk à l'issue d'une session paramétrée par (Alice,Bob), avec Bob l'identité d'une machine honnête. On autorise l'adversaire $\mathcal{A}$ à demander qu'on lui montre *toutes les autres* ssk_i *output par des machines honnêtes au cours d'autres sessions* (y compris incluant Alice ou Bob et/ou avec un peer corrompu) *sauf la session en cours de Bob paramétrée par (Alice,Bob), si elle existe.* Quand bien même cela, $\mathcal{A}$ n'arrive pas à distinguer cette situation de celle où un tiers de confiance avait généré uniformément $\mathsf{ssk} \xleftarrow{\$} \{0,1\}^*$ et l'avait donnée secrètement à Alice.

Remark 9 (Definition 8 $\Rightarrow$ *implicit authentication* $\Rightarrow$ non-identity-misbinding)**.** La Definition 8 apporte donc, comme sous-produit, la garantie suivante appelée *implicit authentication* par [SFW20]. Il s'agit que *si* Alice output ssk à l'issue d'une session (Alice,Christian), alors *si* une machine honnête devait un jour output ssk au cours d'une session quelconque, alors nécessairement (sauf probabilité négligeable) (i) cette machine honnête est Christian (ii) et il a output ssk à l'issue de la session (Alice,Christian). Donc par exemple, cette garantie exclut qu'une machine honnête Cyrano$\neq$Christian puisse output la même $\mathsf{ssk}' = \mathsf{ssk}$, e.g., au cours d'une session (Alice,Cyrano) (sauf probabilité négligeable que la ssk' de cette session soit tirée aléatoirement égale à ssk). Cela vaut *quand bien même* Alice n'aurait jamais reçu output dans la session (Alice,Cyrano). De façon plus surprenante, cela vaut dans le #scénario où (a) Christian serait corrompu par l'adversaire $\mathcal{A}$, et (b) $\mathcal{A}$ (ni Christian) n'auraient jamais appris ssk. En anticipant sur la suite, la *implicit authentication* évite donc le #scénario précédent où Alice recevrait un auth-ciphertext c généré par Cyrano avec ssk, et attribuerait à tort son plaintext m à Christian. Ce #scénario serait donc un identity-misbinding au sens de Section 1.4: Alice attribue un plaintext m à une machine Christian, alors que Christian et l'adversaire sont bien incapables de générer m (en particulier, ils ne peuvent même pas espérer l'obtenir de c puisqu'ils n'ont pas la ssk pour décrypter c). Un tel #scénario est décrit dans Section 7.3.2, obtenu par une attaque contre une mauvaise implémentation de $\mathsf{KeyExch}$, qui ne vérifie donc pas Definition 8. À

noter que cette implémentation est dans le modèle plus exigeant de peers *post*-spécifiés (Definition 10), où ce type d'attaque est plus difficile à prévenir.
On peut maintenant se demander pourquoi la implicit authentication est-elle bien garantie par la Definition 8? Considérons une tentative d'implémentation de KeyExch qui autoriserait le scénario où Alice output la même ssk dans deux sessions: (Alice,Christian), (Alice,Cyrano), considérons un adversaire $\mathcal{A}$ avec le pouvoir autorisé par le security game de la "key-indistinguishability" de Definition 8. Cet adversaire demande à apprendre la session key ssk de (Alice,Christian). Ainsi, il en déduit que la session key de (Alice,Cyrano) est ssk, sans avoir demandé qu'on la lui montre. Il gagnerait donc le security game, donc l'implémentation ne respecterait en fait pas la Definition 8.
On peut enfin relever une contradiction apparente avec certaines spécifications essentiellement équivalentes, de type [CK02b]. Elles rendent explicite qu'il est toléré que $\mathcal{A}$ peut *forcer la valeur de la* ssk *output par Alice* vis à vis d'un peer corrompu Christian. C'est même exactement ce qui se produit lors d'un encryption-based key-exchange (Section 7.3.1) avec un peer (responder) Christian corrompu. Qu'est-ce qui empêcherait alors $\mathcal{A}$ de forcer la réutilisation de la même ssk que dans une session (Alice,Cyrano), ce qui ruinerait la implicit authentication ? En fait ça ne peut pas arriver puisque, tel que spécifié par [CK02b] (et implémenté dans Section 7.3.1), l'adversaire $\mathcal{A}$ ne peut forcer l'utilisation d'une ssk par Alice *que si* $\mathcal{A}$ *connaît la valeur* ssk. Donc, dans le security game, comme $\mathcal{A}$ n'a aucune information sur la valeur de la ssk tirée au hasard pour la session (Alice,Cyrano), il est incapable de forcer sa réutilisation.

Dans le cas d'une session entre Alice et une machine Christian corrompue, la valeur de la ssk output par Alice est potentiellement influencée par l'adversaire $\mathcal{A}$. Mais, grâce à la garantie de la Definition 8, la valeur de cette ssk ne pourra jamais être corrélée (encore moins égale) aux ssk_i de sessions entre machines honnêtes.
Enfin, dans TLS 1.3, l'output ssk est en fait appelé *master secret*, de laquelle sont déduites les session keys. Si un client Alice utilise ce type de protocole pour se connecter à un serveur Bob, elle aura forcément la contrainte que Bob doit apprendre une identité d'Alice avant de commencer sa session, puisque l'identité d'Alice est un paramètre de la session. Mais ce n'est pas une vraie contrainte puisque, comme on le verra dans Section 7.1.3 Figure 12, Alice peut potentiellement générer une identité anonyme, par exemple une paire de clés de signature $(\mathsf{vk}_\mathrm{A}, \mathsf{sk}_A)$, et envoyer publiquement vk_A à Bob dans son premier message. À la réception de ce message, Bob, s'il le souhaite, initialisera une session entre lui et un peer d'identité vk_A. La catégorie suivante de key exchanges n'oblige pas Alice à se créer une quelque identité que ce soit.

Définition 10 (Key exchange avec identification one-sided d'un correspondant pré-spécifié)**.** C'est la variante où seul l'un des participants, par exemple Alice, doit prendre en paramètre préspécifié l'identité du peer, par exemple Bob.

Cette définition est suggérée dans [SFW20]: "We can restrict to the case that only identities of authenticating partners are known, e.g. if a client remains anonymous in a TLS connection." D'un autre côté, cette définition n'exclut pas que le premier message d'Alice contienne l'identité du peer de la session, donc de Bob. Par exemple, si Alice ne connait que l'adresse IP de Bob, ce n'est pas une identité qui caractérise totalement Bob. Alice n'aura donc la garantie, faible, que d'avoir réalisé un key exchange avec une *certaine* machine ayant cette IP. Une identité qui caractérise Bob est par exemple une URL (ou un nom de domaine) ou une public signature verification key. Si Alice et Bob exécutent un tel one-sided key exchange, Bob obtient malgré tout, par définition, une certaine garantie sur l'identité d'Alice. Précisément, la garantie est que, si son peer (qu'il ne sait pas forcément être Alice) est honnête, alors à l'issue, son peer et lui ont tous les deux l'identité id_{ssk}: celle qui consiste à *connaître* ssk. C'est formalisé dans [SFW20, §3.1] comme un cas particulier d'*implicit authentication*. On verra un exemple dans Remark 48.

Un protocole qui garantit à Bob de recevoir la garantie que son peer a output, avant d'output lui-même, est appelée *key confirmation* dans [Fis+16]. On laissera cette question de côté, une méthode générique apportant cette garantie est décrite dans [Kra05].

Remark 11. Que se passe-t-il si Alice met en paramètre une "mauvaise" identité, par exemple Bob' au lieu de Bob, dans une session ? Cela peut arriver pour plusieurs raisons, par exemple, Bob lui aurait au préalable donné une autre identité que la sienne. Pour savoir ce qu'il va se passer il suffit d'appliquer la Definition 8. *Soit* (1) il existe vraiment une machine honnête d'identité Bob', et elle aussi initie un key-exchange avec Alice, par exemple, s'il s'avère qu'elle reçoit la demande d'Alice. Alors, a l'issue, Alice et Bob' ont donc les garanties apportées par Definition 8 (même ssk, tirée aléatoirement uniformément indépendamment des autres sessions). *Soit* (2) il n'existe aucune machine honnête d'identité Bob', alors quand bien même Alice output une ssk' (par exemple, suite à l'interaction avec une machine corrompue jouant une partition, appelée Bob', avec elle), elle a quand même la garantie que ssk' est indépendante de toutes celles output entre peers honnêtes.

On passe enfin à la dernière catégorie de key-exchanges, qui ne nécessitent *aucune connaissance a priori des participants sur leurs identités respectives*. Il peut donc être initié *directement*, par deux machines Alice et Bob reliées par un insecure channel. Contrairement à Definition 10, Alice n'a *pas besoin* de recueillir au préalable l'information de l'identité qu'elle pense être celle de Bob, afin de pouvoir la mettre en paramètre de son premier message. Mais, au vu de l'exemple Section 1.6 appelé "Man in the middle", un tel key-exchange n'aurait a priori que peu d'intérêt s'il ne renvoyait que l'output égal à une ssk. C'est pourquoi il renvoie également un *deuxième* output, égal à l'identité du peer.

Définition 12 (Key exchange avec identification two(or one-) sided entre peers *post-spécifiés*)**.** C'est un protocole prévu pour être exécuté entre deux machines. Il est garanti que:

Consistency si Alice et Bob sont honnêtes, i.e., n'ont pas de side-channel, et finissent chacun de leur côté une session où Alice output (Bob,ssk) et Bob output (Alice,ssk'), alors nécessairement $\mathsf{ssk} = \mathsf{ssk}'$;

Key indistinguishability elle est définie par le scénario suivant, qu'on pourrait appeler "crash-test", couramment appelé *security game*: soit une machine Alice honnête qui output (Bob,ssk) à l'issue d'une session, avec Bob l'identité d'une machine honnête. On autorise l'adversaire $\mathcal{A}$ à demander qu'on lui montre *toutes les autres* ssk_i *output par des machines honnêtes au cours d'autres sessions* (y compris incluant Alice ou Bob et/ou avec un peer corrompu), *sauf la session en cours de Bob avec Alice (en un sens précis de session ID), si elle existe.* Quand bien même cela, $\mathcal{A}$ n'arrive pas à distinguer cette situation de celle où un tiers de confiance avait généré uniformément $\mathsf{ssk} \xleftarrow{\$} \{0,1\}^*$ et l'avait donnée secrètement à Alice.

Ce type de key-exchange a été formalisé pour la première fois dans [CK02a]. Puis, le premier correct à 100% a été décrit dans [Kra03] (appelé "Sigma", qui remonte à 1995, lui-même une adaptation du STS de 1987). C'est en fait essentiellement celui repris dans TLS 1.3, qui apporte donc ces garanties. En bonus, le contenu des messages publics envoyés dans [Kra03] et TLS 1.3 n'apprend rien sur l'identité des peers. Cela n'aurait pas été possible avec un protocole où le premier message d'Alice aurait inclus l'identité de Bob.

Remark 13 (Definition 12 $\Rightarrow$ *implicit authentication* $\Rightarrow$ non-identity-misbinding)**.** La Definition 12 implique en particulier une garantie d'"implicit authentication", comme dans la Remark 9 mais adaptée à ce contexte de *post*-specified peers. En gros, sauf probabilité négligeable, il ne peut exister deux machines

honnêtes distinctes qui output la même ssk mais n'output pas les identités l'une de l'autre. L'argument pour le démontrer est le même que celui de la Remark 9: s'il arrivait que Alice output (Christian,ssk) et (Cyrano,ssk) avec Cyrano honnête, alors avec les pouvoirs autorisés dans le security game, l'adversaire demanderait à ce qu'on lui révèle l'output d'Alice (Christian,ssk) , et en déduirait donc son output (Cyrano,ssk) sans l'avoir demandé: il gagnerait donc le security game.

Dans certains cas, par exemple TLS 1.3, le protocole donne des instructions différentes à chaque machine. On appelle ces instructions des *rôles*. Par exemple, dans le cas où l'un des deux rôles attend de recevoir un message de l'autre avant de commencer, il est appelé *responder*. L'autre rôle (celui qui envoie le premier message) est alors appelé *initiator*.
Dans [BS23, pages 885-890] il est donné une Définition/spécification plus longue de Definition 12. Elle est plus orientée "utilisateur final". Mais pour la comprendre, il faut avant avoir compris la notion de *symmetric authenticated encryption* (Section 4.3). En gros, la Definition/specification de [BS23, pages 885-890] demande la garantie que si Alice output (ssk, id_R), puis que ssk est utilisée par Alice pour faire AuthSymEnc/AuthSymDec, alors cela implémente un secure channel avec un correspondant d'identité id_R.

Remark 14 (Multiple sessions, session IDs)**.** On laissera de côté les questions de sessions multiples entre les mêmes peers Alice et Bob. Par exemple, dans les Remarks de [CK01, §5.2] et [JKL04], une session ID unique entre Alice et Bob est définie comme

2.6 Modèle: temps d'exécution constant, leakage autorisé de tout le harware et software.

La plupart des algorithmes qu'on décrit ne *vérifient pas*, tels quels, les spécifications. La raison est qu'ils ne mettent en général pas le même temps d'exécution pour deux inputs différents de même longueur. On décrit dans Section 9.1 des exemples d'attaque sur certaines implémentations, qui exploitent des différences de temps d'exécution. On fait donc le disclaimer que les algorithmes décrits dans ce cours vérifient les spécifications s'ils sont compilés en des algorithmes dont le temps d'exécution ne dépend que de la longueur des inputs. Par abus de langage, on dit qu'ils sont en *temps constant.* //D'un côté cette contrainte est parfois superflue pour une définition "standalone" de la sécurité, notamment pour les algorithmes qui ne déclenchent pas de messages à la fin (Dec, Verify etc.). D'un autre côté, pour arriver à des garanties de composabilité [Can01], même ces derniers doivent terminer en temps constant.
On renforce maintenant les spécifications. Il est admis depuis Kerckhoffs qu'un bon cryptosystème doit résister à un adversaire $\mathcal{A}$ qui aurait une information préliminaire totale sur les machines. Cela inclut l'implémentation de l'algorithme utilisé, et la connaissance de tout le hardware. Donc on supposera implicitement que toute machine, à l'instant t où elle est initialisée, leake entièrement son état initial à $\mathcal{A}$: hardware et software. Lorsqu'on dit qu'une machine n'a pas de canal d'influence et de leakage avec $\mathcal{A}$, on veut dire que, *après* cet instant t où tout a leaké, elle ne leake rien et ne subit pas d'influence.

3 Outils

3.1 Fonctions de hachage et vérification d'intégrité

Une fonction de hachage est un algorithme déterministe qui prend en input une suite de bits de longueur arbitraire et output une suite de bits de taille spécifiée par l'utilisateur, par exemple 256 pour SHA256.

$$H : \{0,1\}^* \longrightarrow \{0,1\}^{256} \tag{4}$$

Les propriétés demandées dépendent des applications. //Les théoriciens demandent des spécifications très fortes pour pouvoir apporter des garanties sur certains de leurs utilisations. Par exemple la preuve de sécurité d'une signature EdDSA, suppose que H se comporterait comme un tiers de confiance qui, pour chaque nouvel input demandé par une machine sur terre, tirerait 256 bis d'output au hasard (hypothèse dite de random oracle). Pour le cas d'usage de l'Example 15, on a juste besoin de la spécification *collision freeness*. Elle garantit que même si toutes les machines sur terre unissaient leur puissance et leurs connaissances, elles ne pourraient pas produire x, x' distincts tels que $H(x) = H(x')$, sauf avec probabilité négligeable //cette garantie implique donc celle appelée "2nd-preimage resilience". Donc cette spécification implique que: soient deux x, x' produits par des machines (par exemple des humains), alors

$$H(x) = H(x') \implies x = x' \text{ sauf avec probabilité négligeable.} \tag{5}$$

Example 15 (Vérification d'intégrité)**.** Bob cherche à obtenir un document x donc il connait seulement le hash: $h_x = H(x)$. Lorsque Bob obtient un document x', il teste si $H(x') =? h_x$. *Si* l'égalité est vérifiée alors l'Equation (5) lui garantit que $x' = x$, donc il output x'. Si l'égalité n'est pas vérifiée alors nécessairement $x' \neq x$ donc il rejette x' (et continue de chercher).
On est dans cette situation par exemple si Bob est un juge qui chercher à faire exécuter un contrat dont une des clauses fait référence à un document x dont elle donne seulement le hash h_x. Si un témoin apporte à Bob un document x' tel que $H(x') = x$, alors cela apporte la preuve à Bob que x' est bien le document x auquel la clause faisait référence. Donc Bob rend son jugement en considérant que le contrat fait référence à $x' = x$. C'est entre autres ce travail que font les mineurs / valideurs sur une blockchain.

//Voici un ordre de grandeur de la probabilité d'échec de l'Equation (5). Les machines du réseau Bitcoin calculent au total 3.10^{20} valeurs de hash par seconde, soit $a = 2.10^{28}$ en un an. Donc leur probabilité de trouver une collision en un an est de $O(a^2/2^{256}) = 10^{-21}$ //par le paradoxe des anniversaires, en idéalisant H comme un random oracle. Donc c'est 13 ordres de grandeur plus faible que la probabilité que la terre soit détruite par un astéroide dans l'année qui suit, qui est de $O(10^{-8})$.

3.2 Pseudorandom functions (PRF)

Définition 16 (Pseudorandom function PRF[Lin06, Definition 3.24] [BS23, Definition 3.1])**.** API: c'est une fonction déterministe $\mathrm{PRF}(k, x) \to y \in \{0,1\}^\beta$ qui prend en input une *key* $k \in \{0,1\}^\kappa$, avec typiquement $\kappa = 128$ ou 256, et une input value $x \in \{0,1\}^\beta$, où β est un paramètre typiquement 128 (AES) ou 64 (Chacha20). Il output $y \in \{0,1\}^\beta$.
Elle vérifie la propriété que si k est tirée uniformément au hasard, alors pour tout input x qui serait choisi par l'adversaire $\mathcal{A}$ lui-même, il serait incapable de distinguer l'output y d'une suite de bits aléatoire uniforme de même longueur β.

Une suite de β bits est appelée *bloc*. Une PRF qui a un ıinverse public et efficacement calculable est appelée *blockcipher*. La condition précédente s'exprime comme: soit $\mathrm{BC}(k, x) \to y \in \{0,1\}^\beta$ un blockcipher, il existe une fonction publique $\mathrm{BC}^{-1}(k, y) \to x \in \{0,1\}^\beta$ telle que

$$\mathrm{BC}^{-1}\big(k, \mathrm{BC}(k, x)\big) = x \quad \forall k \in \{0,1\}^\kappa,\ x \in \{0,1\}^\beta$$

Mauvais exemples: DES n'est plus un PRF pour les adversaires d'aujourd'hui [BS23, p. 4.2.1], seule une variante (triple-DES) est encore considérée comme telle. [IET18a, p. 2.3]: "Poly1305 is not a suitable choice for a PRF." car elle est biaisée. Mais elle sert quand même à fabriquer un message authentication tag d'une façon qui est autorisée dans TLS 1.3 [IET18b].
Exemples: AES (qui est en plus un blockcipher), et ChaCha20, sont les deux PRFs autorisées dans les implémentations de TLS 1.3 [IET18b].

3.3 Pseudorandom generators (PRG)

Définition 17 (PRG[Lin06, Definition 3.14] [BS23, Definition 3.1])**.** API: c'est une fonction déterministe $\mathrm{PRG}(\$)^L \to \{0,1\}^L$ qui prend en input une *seed*, $\$ \in \{0,1\}^\kappa$, avec typiquement $\kappa = 128$ ou 256, et output une chaîne de L bits, où L est spécifié par l'utilisateur.
Elle vérifie la propriété que si \$ est tiré uniformément au hasard et si L en dessous d'une taille maximale, appelée *expansion factor*, alors si on montrait l'output R à l'adversaire $\mathcal{A}$ (mais pas \$), il ne serait pas capable de le distinguer d'une suite de bits aléatoire uniforme de même taille.

Une méthode classique pour implémenter un PRG à partir d'une PRF est recommandée par le RFC [IET05, §6.2.1]: [3] En input une clé k, output PRF_k sur les valeurs prises par un compteur. Elle est prouvée dans [Lin06, Exercice 3.16] (a fortiori [Lin06, Thm. 3.33]).

3.4 Secure random bits generation $r \xleftarrow{\$} \{0,1\}^\ell$.

Cette instruction signifie: tirer $r \in \{0,1\}^\ell$ une chaîne de ℓ bits uniformément au hasard, en suivant la procédure générale suivante. Elle est recommandée par les RFC [IET05, §6] [4] Elle utilise comme ingrédient un *pseudorandom generator* (PRG). Un PRG est une fonction déterministe, qui doit satisfaire les spécifications dans Section 3.3. La procédure permet de retourner une chaîne de 2^L bits uniformes aléatoires, pour L inférieur à la taille maximale précisée par le mode d'emploi du PRG.

(i) tirer une *seed* $\$ \in \{0,1\}^{256}$ uniformément au hasard;

(ii) output $R \leftarrow \mathrm{PRG}(\$)^L$, puis *supprimer* \$.

Si besoin de plus de bits que L, répéter la procédure. Si besoin d'exécuter plusieurs instructions $r_i \xleftarrow{\$} \{0,1\}^{\ell_i}$ de taille totale $L := \ell_1 + \cdots + \ell_n$, générer $R \leftarrow \mathrm{PRG}(\$)^L$ en une fois avec la méthode précédente, puis découper R en sous-chaînes de bits $r_1, \ldots, r_n$ de tailles $\ell_1 + \cdots + \ell_n = L$, et output r_i pour chaque instruction. À noter que l'ANSSI recommande cette méthode avec trois instantiations plus particulières [ANS21, §7.2, R23]. Le PRG utilisé n'apparaît pas de façon explicite dans leurs descriptions détaillées. Par exemple, pour celle appelée CTR-DRBG, le PRG est en gros: $\mathsf{k}_{\mathsf{ed}} \to [\mathrm{PRF}(\mathsf{k}_{\mathsf{ed}}, ctr), \ldots, \mathrm{PRF}(\mathsf{k}_{\mathsf{ed}}, ctr + \ell/128)]$.
La nouvelle implémentation de getrandom() de Linux depuis 5.17 [man23; Don22; Inf22] ne suit pas exactement la méthode. Elle renvoie PRG(\$), où \$ est une seed choisie par Linux de façon pas uniforme indépendante des autres seeds. Concrètement, le temps est découpé en intervalles de 5 minutes $[t, t+5[$, tels que toutes les seeds générées dans $[t, t+5[$ sont calculées de façon déterministe par rapport à une donnée en mémoire à t appelée 'entropy pool'. Sa taille minimum est de 256bits. En pratique, la méthode de génération des seeds de Linux fait qu'un adversaire qui ne verrait pas l'état interne de la machine, et dont la puissance de calcul ne permettrait pas de casser les standards de authenticated encryption du RFC, ne détectera pas la différence. En fait Linux déroge à la méthode même après t+5 puisque l'entropy pool

[3] One way to produce a strong sequence is to take a seed value and hash the quantities produced by concatenating the seed with successive integers, or the like, and then to mask the values obtained so as to limit the amount of generator state available to the adversary.

It may also be possible to use an "encryption" algorithm with a random key and seed value to encrypt successive integers, as in counter (CTR) mode encryption.

[4] When a seed has sufficient entropy, from input as described in Section 3 and possibly de-skewed and mixed as described in Sections 4 and 5, one can algorithmically extend that seed to produce a large number of cryptographically-strong random quantities. Such algorithms are platform independent and can operate in the same fashion on any computer. For the algorithms to be secure, their input and internal workings must be protected from adversarial observation. The design of such pseudo-random number generation algorithms, like the design of symmetric encryption algorithms, is not a task for amateurs.

n'est renouvelée qu'en partie. Cette dérogation supplémentaire permet de résister à un adversaire qui contrôlerait toutes les sources d'input de l'entropy pool à l'instant t+5 (mais qui ne verrait pas l'entropy pool ele même)
Exemples de conséquences quand la procédure *n'est pas* suivie:

Output $R \leftarrow \mathrm{PRG}(\$)^L$ pour L plus grand que prévu par le mode d'emploi. Un exemple d'implémentation de PRG est la méthode de Section 3.3 appliquée à la PRF ChaCha20 en mode compteur. Si on considère les paramètres autorisés dans TLS 1.3 [IET18b], qui sont ceux décrits dans le RFC [IET18a, §2.4], le mode d'emploi [IET18a, §2.8] dit qu'il ne faut pas appliquer ces paramètres et ce mode pour générer plus de $L = 2^{32} - 1$ blocs de 64 bytes chacun avec la même seed (= paire (key,nonce) dans leur contexte). Si un utilisateur ne suit pas cette règle, à partir du 2^{32}-ième bloc généré les blocs reviennent à l'identique. Deux protections: dans libSodium il y a un garde-fou à 2^{32} blocs d'output [Lib22], tandis que l'implémentation de getrandom() dans Linux 5.18 [man23; Don22; Inf22] change les paramètres de ChaCha20 pour aller jusqu'à 2^{64} blocs [Tso22a] //d'une façon différente de celle proposée par le RFC [IET18a, p22 1.]. //Plus généralement, il existe des études qui estiment le coût des attaques vs les paramètres choisis [Ber15; LP17]. Ces deux références donnent des bornes pessimistes car considèrent une autre utilisation que simplement PRG.

Restaurer ou copier (fork) une machine à partir d'un snapshot contenant une seed déjà utilisée. Soit une copie de l'état d'une machine (un *snapshot*), telle qu'il est possible d'en déduire facilement des seeds déjà utilisées par cette machine. Alors, conserver cette copie est donc contraire à la procédure. Voir [Eve+14], et les related works, pour des attaques sur des VMs créées à partir de snapshots de VMs existantes. Elles exploitent le fait que les seeds dans Linux sont générées de façon déterministe à partir l'entropy pool à l'instant t, et donc qu'un snapshot en t détermine toutes les seeds dans $[t, t+5[$. Quand bien même l'adversaire n'aurait pas accès au snapshot, on a la conséquence que la machine père et fils génèrent exactement les mêmes seeds, et donc le même aléa, pendant les premières secondes suivant le fork. Or, on verra par exemple que AES-GCM et les signatures de Schnorr sont complètement cassées en cas de réutilisation de la même paire (nonce, clé).
La version 5.18 de Linux [Don22; Cor22] introduit une contre-mesure: lorsqu'une nouvelle VM est créée à partir d'un snapshot, l'hyperviseur peut lui attribuer une nouvelle ID, dont le hash est injecté dans son entropy pool. //D'un côté, cela ne rajoute pas plus d'entropie que n'en contient cette ID. De l'autre, dans l'hypothèse où $\mathcal{A}$ ne connaîtrait pas l'état de l'entropy pool et où l'extracteur de seeds serait un random oracle, alors cette injection suffit pour faire que les seeds du fils soient indépendantes de celles du père, quand bien même $\mathcal{A}$ aurait choisi le nouvel ID. Une autre contre-mesure est l'utilisation de la fonction RNDADDENTROPY par l'utilisateur, qui lui permet d'injecter des données de son choix dans l'entropy pool. //À noter que l'accès à celle-ci semble être restreint aux administrateurs depuis Linux 5.18: [Inf22] vs [Tso22b]. //L'entropy pool dans Linux change progressivement d'état, et une nouvelle seed en est extraite toutes les 5 minutes. Cette progressivité, qui est donc contraire à la procédure, évite qu'un adversaire qui prendrait le contrôle de toutes les sources d'entropie à un instant t, puisse complètement contrôler une seed créée à un instant t. Ne pas la vider à chaque seed permet aussi d'éviter à l'utilisateur d'avoir à attendre qu'elle se reremplisse, comme c'était le cas avec dev/random avant Linux 5.6.

Ne pas tirer uniformément les bits de la seed. Une machine Linux qui vient d'être créée et n'aurait pas assez de sources d'entropie, commence par remplir son entropy pool avec une méthode heuristique appelée "the Linus Jitter dance" [Don22]. Cette attaque [Hen+12] de récupération de clés, exploitait l'existence d'un grand nombre de machines qui initialisaient leur clé RSA juste après leur création, avec des seeds parfois égales. Donc statistiquement il existait des paires de machines qui tiraient la même valeur p pour l'un des facteurs premiers de leurs $N \neq N'$, $N = pq$, $N' = pq'$. Donc le $PGCD(N, N')$ renvoyait p.

3.5 Secure random generation $r \overset{\$}{\leftarrow} [0, \ldots, N]$.

L'instruction notée $r \overset{\$}{\leftarrow} [0, \ldots, N]$ signifie: tirer un nombre au hasard uniformément dans $[0, \ldots, N]$. Plus généralement, l'instruction notée $r \overset{\$}{\leftarrow} \mathscr{D}$ signifie: tirer un élément au hasard dans un ensemble avec une loi de probabilité, c'est à dire une distribution, $\mathscr{D}$. La méthode est de d'abord générer une suite de bits aléatoire uniforme, avec la méthode précédente et assez grande, puis lui appliquer une fonction déterministe publique fixe spécifiée selon $\mathscr{D}$. Par exemple, [Ava+21, Algorithm 2] spécifie la fonction à appliquer pour générer une variable gaussienne.

Exercice 18. Combien de bits tirer, et quelle fonction appliquer, pour générer $r \overset{\$}{\leftarrow} [0, \ldots, N]$? L'ANSSI [ANS21, §7.3, R34] recommande au choix la technique "par rejet" [FIPS186, Appendix B.1.2] ou celle par utilisation d'aléa additionel [FIPS186, Appendix B.1.1].

3.6 Courbes elliptiques

Soit $p > 3$ un nombre premier. Soit $W(X, Y) = Y^2 - (X^3 + a.X + b)$ un polynôme à deux variables à coefficients $a, b \in \mathbb{Z}/p\mathbb{Z}$ //tels que $4a^3 + 27b^2 \neq 0$. Notamment, la démonstration de la Prop III 1.4 dans Silverman (lire $\Delta \neq 0$ dans (i)) montre que c'est une condition suffisante pour tout p, voir aussi [Pap14, slide 15]. Cependant pour $p = 2, 3$ elle est trop stricte, aucune courbe elliptique n'est de cette forme dans le cas $p = 2$. On appelle "points de la courbe elliptique d'équation W" l'ensemble suivant:

$$E := \{(x \in \mathbb{Z}/p\mathbb{Z}, y \in \mathbb{Z}/p\mathbb{Z}),\ , W(x, y) = 0\ \} \cup \text{“}O\text{”} \tag{6}$$

Donc dans la classe E, par définition il y a deux types d'objets: des paires $(x, y) \in (\mathbb{Z}/q\mathbb{Z})^2$, et un élément supplémentaire qui est le symbole spécial "O". Ce dernier est parfois appelé "le point à l'infini".

Exercice 19. soit $Z/11Z$ et $W = y^2 - (x^3 - 2x)$ Combien d'éléments contient E ? //Dans la slide 24 de [Cos18] il dénombre 11 points (u, v) tels que $W(u, v) = 0$ Auquel s'ajoute le symbole spécial $O \in E$. Conclusion : $|E| = 12$.

Définition 20. Loi $P + Q \leftarrow (P, Q)$ ([BS23, §15.2] "the addition law")
On considère une courbe elliptique E définie par le polynôme $W = y^2 - (x^3 + a.x + b)$ à coefficients dans $\mathbb{Z}/p\mathbb{Z}$. Soient P et Q deux points de E. Décrivons le calcul qui retourne le point $P + Q \in E \leftarrow (P, Q)$ Il faut distinguer selon 4 cas:
0) P ou Q est égal à "O". Alors $P +$ "O" $= P$ $\forall P \in E$, idem pour Q. C'est à dire que "O" est *l'élément neutre de* $(E, +, \text{“}O\text{”})$.
On suppose maintenant que ni P ni Q n'est "O". Ce sont donc des paires de chiffres dans Z/pZ: $P = (x_P, y_P) \in E$ et $Q = (x_Q, y_Q) \in E$.
1) si $x_P \neq x_Q$ l'output est une paire de chiffres: $P + Q = (x_{R'}, y_{R'})$ qui se calcule de la façon suivante. Calculer $s := \frac{y_P - y_Q}{x_P - x_Q}$ //la "slope" = "pente de la droite (P,Q)". Output la paire $\{x_{R'} = s^2 - x_P - x_Q, y_{R'} = -(y_P + s.(x_{R'} - x_P))\}$ //mnémotechnique: $y_R = y_P + slope \times (x_R - x_P)$; $y_{R'} \leftarrow -y_R$
2) $x_P = x_Q$ et $y_P = -y_Q$ alors output $R =$ "O".
3) $x_P = x_Q$ et $y_P = y_Q$ et $y_P \neq 0$ (sinon si $y_P = 0$ alors on est dans le cas 2). On est donc dans une situation où $P = Q$. Définissons le calcul de $R' := P + P = [2].P$, noté (x_{2P}, y_{2P}). $s \leftarrow \dfrac{3x_P^2 + a}{2y_P}$; output $(x_{2P} = s^2 - 2x_P, y_{2P} = -y_P + s.(x_P - x_{2P}))$

Exercice 21. Même E que dans l'Exercice 19. Calculer $(x_P = 5, y_P = 7) + (x_Q = 8, y_Q = 10)$. Nous sommes dans la situation 1) On calcule donc la slope $s = (7-10)/(5-8) = -3/(-3) = 1$ [Exercice: que vaut 3/2 dans $\mathbb{Z}/11\mathbb{Z}$? C'est égal à 3.2^{-1}, avec 2^{-1} = le nombre tel que $2.2^{-1} = 1 = 6$. Conclusion: $3/2 = 3.6 = 18 = 7 \bmod 11$]. Output l'élément $x_{R'} = 1^2 - 5 - 8; y_{R'} = (x_{R'} = -12 = 10 \bmod 11, y_{R'} = -(y_P + s.(x_R - x_P)) = -(7 + 1.(10-5)) = -12 = 10 \bmod 11)$.

3.7 Compress: arrondi *modulo q* à 0 ou à $q/2$.

On suppose pour simplifier que q est divisible par 4. Intuitivement, Compress est la fonction qui prend en input $x \in \mathbb{Z}/q\mathbb{Z}$ et output: 0 si x est plus proche de 0 que de $\frac{q}{2}$ modulo q; ou 1 sinon. C'est illustré sur la Figure 1. Si on met en input un polynôme, alors elle applique Compress (Definition 22) sur chaque coefficient. Concrètement:

Définition 22 (fonction Compress)**.**

$$\text{Compress}(x \in \mathbb{Z}/q\mathbb{Z}) : \text{ output } 0 \text{ si } x \in \left]-\frac{q}{4}, \frac{q}{4}\right[\bmod q = \left[0, \frac{q}{4}\right[\cup \left]\frac{3q}{4}, q\right[; \tag{7}$$

$$\text{et } 1 \text{ sinon, i.e., si } x \in \left[\frac{q}{4}, \frac{3q}{4}\right] . \tag{8}$$

$$\text{Compress}(P = \sum_{i=0}^{n-1} P_i X^i) : \text{ output } \sum_{i=0}^{n-1} \text{Compress}(P_i) X^i . \tag{9}$$

3.8 Autres définitions équivalentes de la fonction Compress.

Au cas où cela pourrait être utile, on donne deux autres définitions équivalentes de la fonction Compress. Notamment, la deuxième (Equation (11)) est celle spécifiée sous une forme plus générale dans Kyber ([Ava+21]).
La première définition est celle donnée en intuition au début de Section 3.7, il reste à donner un sens rigoureux à "plus proche que.." modulo q.

Définition 23 (Distance, proximité entre deux chiffres modulo q)**.** Soient $x, y \in \mathbb{Z}/q\mathbb{Z}$, on définit la *distance de x à y dans $\mathbb{Z}/q\mathbb{Z}$, notée $|y - x|_{\mathbb{Z}/q\mathbb{Z}}$, comme la longueur du plus court chemin de x à y dans $\mathbb{Z}/q\mathbb{Z}$.* Concrètement:

- si $|y - x| \leqslant \frac{q}{2}$ alors $|x, y|_{\mathbb{Z}/q\mathbb{Z}} = |y - x|$;
- et sinon si $|y - x| > \frac{q}{2}$, alors $|x, y|_{\mathbb{Z}/q\mathbb{Z}} = q - (y - x)$ //en effet, si par exemple $x < y$, alors le chemin le plus court part de x vers la gauche, arrivé en 0 il boucle sur q, et arrive donc sur y par la droite. Sinon si $x > y$, c'est le chemin qui part de y vers la gauche, etc..

On dit que "x est plus proche de y que de z modulo q" si $|y - x|_{\mathbb{Z}/q\mathbb{Z}} < |z - x|_{\mathbb{Z}/q\mathbb{Z}}$.

Une définition équivalente est la suivante, qui est suggérée dans la spécification [Ava+21]. Suivant [Bos+17, §2.2], on note:

$$\text{"}x \bmod^{\pm} q\text{" l'unique nombre dans } \left]-\frac{q}{2}, \frac{q}{2}\right] \text{ congru à } x \text{ modulo } q \text{ (donc } x \text{ si } x \in [0, \frac{q}{2}], \text{ et } x - q \text{ sinon).} \tag{10}$$

Alors, avec cette notation, $|y - x|_{\mathbb{Z}/q\mathbb{Z}} = |(y - x) \bmod^{\pm} q|$.

Pour la deuxième définition, on a en plus besoin de la notation: soit $x \in \mathbb{R}$ un nombre réel, alors $\lceil x \rfloor$ désigne l'entier le plus proche de x; par convention l'entier supérieur en cas d'égalité.

$$\mathsf{Compress}(x) := \left\lceil \frac{x}{q/2} \right\rfloor \mod 2 \quad \text{(c'est une division réelle, pas } \bmod q\text{)} \tag{11}$$

4 Cryptographie symétrique

Ce sont des algorithmes utilisables par toute paire de machines, Alice et Bob, ayant accès à un key exchange KeyExch (Section 2.5), et à un channel (à préciser selon les cas).

4.1 Message authentication code (MAC).

4.1.1 Comment bien l'utiliser pour implémenter un authenticated channel de Alice vers Bob

Un MAC ([Lin06, Def. 4.1], [BS23, Def. 6.1]) consiste en deux algorithmes MAC.Sign et MAC.Verify qui respectent les spécifications de la Figure 1. Une conséquence de ces spécifications est qu'il est possible d'implémenter un authenticated channel entre deux machines Alice et Bob (Definition 7), dès qu'elles ont accès à un key exchange KeyExch (Section 2.5) et à un insecure channel. Le protocole vérifie les spécifications d'un authenticated channel vis à vis d'Alice si elle ne subit pas de side-channel (Section 1.8), i.e., autre que l'utilisation par $\mathcal{A}$ de son API d'envoi ($\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$). De même, le protocole vérifie les spécifications d'un authenticated channel vis à vis de Bob si Bob ne subit pas de side-channel, i.e., autre que l'utilisation par $\mathcal{A}$ de son API de réception ($\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$). On commence par décrire un protocole qui implémente un authenticated channel unidirectionnel de Alice vers Bob. Il offre 2 APIs, appelées ($\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$,$\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$). En fait, le protocole décrit n'exclut pas que Bob $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$ plusieurs fois la même donnée envoyée une seule fois par Alice. Donc il ne vérifie pas tout à fait la spécification d'un secure channel. Il peut se compiler facilement en en secure channel avec l'une des méthodes décrites dans Remark 24 ci-dessous.

- Alice et Bob appellent tous les deux KeyExch et reçoivent une *(secret symmetric) MAC key*: k_{m};
- pour $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ une donnée publique M à Bob, Alice génère un *tag* $t \leftarrow \mathsf{MAC}.\mathrm{Sign}(\mathsf{k}_{\mathrm{m}}, M)$ et envoie publiquement (M, t) à Bob via le insecure channel;
- quand Bob reçoit un (M', t'), il teste si $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k}_{\mathrm{m}}, M', t') = \mathsf{accept}$; *Si* c'est le cas, alors l'*unforgeability* (ci-dessous) lui garantit que la donnée M' a bien été mise, à un moment donné, en input de $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k}_{\mathrm{m}}, \bullet)$ de Alice ou de Bob (les deux API sont égales) [car par hypothèse Bob et Alice n'utilisent que leur API MAC.Sign pour générer tout ce qu'ils envoient sur le insecure channel, donc on est bien dans les conditions du security game.]. Donc Bob output M' sur l'API $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$. Sinon, il ignore M'.

Dans Section 4.1.3 on montre deux méthodes possibles pour, à partir de ce protocole simple, implémenter un authenticated channel bidirectionnel. On montre aussi qu'il est facile de se tromper.

Remark 24 (Ne recevoir qu'une fois (Replay attacks))**.** Les API précédentes: $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ chez Alice et $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$ chez Bob n'implémentent pas tout à fait un authenticated channel puisque, si l'adversaire injecte vers Bob une copie d'un (M, t) déjà envoyé, Bob va output une deuxième fois M. On compile donc les API précédentes, en des API qui assurent que Bob va output sur $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ une seule fois chaque messages envoyé par Alice. Soit une donnée M en input de $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$. Par simplicité, soit i le nombre de données qui ont été mises en input de $\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$, chez Bob, jusqu'à maintenant. Alors, Alice génère la concaténation $M_{i+1} = (\text{“(i+1)-ème donnée”} \| M)$. Puis elle envoie M_{i+1} via le $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ du protocole

précédent. Lorsque Bob reçoit M_{i+1} via le $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ du protocole précédent, avec l'en-tête "(i+1)-ème donnée", il attend d'avoir reçu tous les messages numérotés par les $i \in [1,\dots,i]$, puis output de M_{i+1} sur son API de réception. Évidemment en pratique, le compteur i peut être remplacé par tout autre identifiant unique de message prédéfini entre Alice et Bob. Par exemple, il peut s'agir du numéro d'instance du protocole, suivi par exemple du hash des messages envoyés et reçus jusqu'à maintenant: c'est ce qui est fait dans TLS 1.3 [Dow+21, Figure 2].
Cette méthode est suggérée dans la [BCK98, Remark 1]. Ils font la remarque qu'elle est dangereuse si Bob ne stocke pas la valeur du compteur avec Alice. De même, si Alice et Bob définissent des numéros d'instance "à la volée", c'est à dire basés sur certaines données contenues dans les messages échangés, alors il y a des risques qu'ils se retrouvent avec un numéro d'instance déjà utilisé. Une attaque sur TLS 1.2 qui exploite ce scénario est décrite dans [Bha+14] ("triple handshake", qui conduit à une identity-misbinding de messages reçus de l'adversaire attribués à tort à une machine honnête), et une autre dans [LS17] (deux instances entre machines honnêtes utilisant la même ssk). Notamment, cela ouvre la porte à des attaques appelées *replay*, dans lesquelles Bob accepte de nouveau des données déjà reçues de Alice dans le passé. Pour ces raisons, [BCK98, Remark 1] mettent en avant une méthode coûteuse mais qui ne nécessite aucune mémoire du passé par Alice et Bob. Pour chaque donnée m reçue, Bob la met en attente le temps d'envoyer à Alice une demande de confirmation d'envoi de m avec un numéro de demande N_m qu'il a généré aléatoirement. Puis, lorsqu'il reçoit la signature d'Alice sur N_m, Bob fait $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$ de m. On peut remarquer que cette méthode n'informe pas Bob de l'ordre d'envoi, donc si besoin cette information peut être ajoutée par Alice.

4.1.2 Spécification, exemples, mauvaise utilisation

Message authentication code (MAC)

API: $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, M) \to t$: en input $\mathsf{k_m} \in \{0,1\}^\kappa$ une *(secret symmetric) MAC key* et $M \in \{0,1\}^*$ une *donnée* quelconque, c'est à dire une suite de bits de longueur arbitraire, en output t un *authentication tag*.
$\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, M, t) \to \mathsf{accept}$ or reject : en input $\mathsf{k_m}$ une (secret symmetric) MAC key, $M \in \{0,1\}^*$ une donnée et t un tag, output accept ou reject.

Correctness: $\forall\, \mathsf{k_m}, \forall\, M$: $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, M, \mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, M)) = \mathsf{accept}$;

Unforgeability: Elle se définit à l'aide du security game suivant. Soient Alice une machine qui ne subit aucune emprise de $\mathcal{A}$ hormis le leakage de son état initial (Section 2.6). Elle génère $\mathsf{k_m} \xleftarrow{\$} \{0,1\}^\kappa$. Désormais, elle offre également à l'adversaire $\mathcal{A}$ un accès illimité à l' API $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet)$. C'est à dire que $\mathcal{A}$ n'a pas accès à la $\mathsf{k_m}$ mais que, en input une donnée M' de son choix, appelée *requête*, il récupère $t' \leftarrow \mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, M')$ en output. Il peut recommencer pour autant de *requêtes* $M'_1, M'_2, \dots$ qu'il souhaite. $\mathcal{A}$ a aussi accès à l'API $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, \bullet, \bullet)$ ([BS23, Thm 6.1]). Alors, $\mathcal{A}$ est incapable de produire une paire donnée-tag (M,t) telle que $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, M, t) = \mathsf{accept}$ alors que M ne serait pas dans les liste des requêtes précédentes $M'_1, M'_2, \dots$.

Figure 1: Définition d'un MAC.

Il est facile de voir que l'unforgeability reste valable dans un contexte multi-utilisateurs, au sens suivant. Soient des machines Alice, Bob etc., qui auraient en commun la même symmetric MAC key $\mathsf{k_m}$, et tel que *aucune d'entre elles*, ni le mécanisme qui a généré la clé $\mathsf{k_m} \xleftarrow{\$} \{0,1\}^\kappa$ (par exemple un $\mathsf{KeyExch}$ idéalisé), n'a de side-channel avec $\mathcal{A}$. Alors, l'unforgeability reste vraie même si $\mathcal{A}$ a accès à toutes leurs API (s'en

convaincre en les modélisant toutes comme une même machine, et vu qu'elles ont toutes la même API $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet)$). Un exemple de MAC est le standard HMAC (RFC 2104, [BS23, §8.7.2]), encore utilisé dans le handshake de TLS 1.3. Un autre exemple est le MAC de "Carter-Wegman" [BS23, §7.4], qu'on retrouve sous une forme tordue dans le standard AES-GCM (Section 4.3.3). Dans ces deux exemples, la fonction $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, M, t)$ est simplement celle qui vérifie l'égalité $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, M) =? t$ et renvoie accept si elle est vraie.

Exercice 25 (Dictionary attack)**.** Alice (un serveur) et Bob (un client) connaissent tous les deux une chaîne de caractères secrète x choisi au hasard par Bob dans un ensemble total X de taille 2^{48}. On appelle parfois x un "mot de passe", et X le "dictionnaire" des mots de passe. Alice souhaite un système permettant à Bob de lui envoyer des messages authenticated par un tag généré avec x. Elle s'impose deux contraintes: (i) utiliser en sous-routine une fonction MAC qui remplit la condition Figure 1, notamment, qui prend en input une clé $\mathsf{k_m} \in \{0,1\}^\kappa$ de taille standard ($\kappa = 128$ ou 256). (ii) faire en sorte qu'elle n'ait pas à stocker la valeur de x, mais seulement un hash (sinon ce serait une mauvaise pratique). Pour remplir ses contraintes, elle définit le protocole suivant. Il est paramétré par $H : \{0,1\}^{48} \to \{0,1\}^\kappa$ une fonction déterministe avec toutes les propriétés imaginables de préimage résistance, d'extraction d'entropie, etc. Par exemple la fonction HKDF ([BS23, p. 8.10.5]) qui est utilisée dans TLS 1.3. Elle spécifie pour Bob: $\forall m$, $\mathsf{MAC}.\mathrm{Sign}(x, m)$ retourne $t \leftarrow \mathsf{MAC}.\mathrm{Sign}(H(x), m)$. Alice de son côté stocke seulement $h = H(x)$. Elle spécifie pour elle-même: en input (m, t), si $\mathsf{MAC}.\mathrm{Verify}(h, m) =$ accept alors elle output accept, sinon reject. Ce protocole garantit-il l'*unforgeability*, au sens de Figure 1 ?

L'attaque décrite dans l'exercice précédent est inévitable dès que la taille du dictionnaire est assez courte. Par exemple, on pourrait imaginer qu'Alice prenne une contre-mesure consistant à ignorer tous les messages d'un certain expéditeur prétendant être Bob dès l'instant où elle reject un tagged message (m, t) qu'elle reçoit de lui. L'adversaire $\mathcal{A}$ peut contourner cette mesure en recontactant à chaque fois Alice de la part d'une machine anonyme différente. Alice sera bien obligée de vérifier tous les (m, t) reçus de chacune de ces machines, car il se pourrait très bien que l'un des expéditeurs soit Bob. Une fois que $\mathcal{A}$ aura trouvé un x tel que Alice renvoie accept sur les tags générés avec x sur deux m, m' différents, il est quasi-certain que x est le mot de passe de Bob. Il pourra donc complètement se faire passer pour Bob.

4.1.3 Comment échouer (et réussir de deux façons) à implémenter un authenticated channel bidirectionnel

Le protocole simple de Section 4.1.1 implémente un authenticated channel unidirectionnel de Bob vers Alice au sens de Definition 7, il est donc prouvé ([Can01],[CDN15, §4]) que ses spécifications sont préservées lorsqu'il est utilisé en concurrence, en sous-routine etc. Donc, pour implémenter un authenticated channel bidirectionnel, il suffit d'utiliser en plus une autre instance de ce protocole, donc cette fois qui fournit les API $\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$ chez Alice et $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ chez Bob. En particulier, il faut réexécuter la première étape, qui consiste à appeler KeyExch, c'est à dire qu'il faut *générer une autre clé de MAC pour le sens Bob→Alice*. C'est en gros ce qui est fait dans TLS 1.3, à ceci près que les deux clés sont en fait déterminées à partir du même "master secret". On décrit maintenant ce qui peut se produire de mal lorsque Alice et Bob, au contraire, *réutilisent* la même clé $\mathsf{k_m}$ pour les deux sens (avec une solution correcte à la fin malgré tout).

Exercice 26 (Un tag valide permet-il, en soi, de distinguer l'émetteur ?)**.** (a) Alice et Bob implémentent un authenticated channel unidirectionnel avec le protocole de Section 4.1.1. Soit $\mathsf{k_m}$ la MAC key utilisée. En outre, ils implémentent des APIs dans l'autre sens Bob→Alice: $\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$, $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ en utilisant la méthode d'envoi-réception du protocole de Section 4.1.1, *mais en utilisant la même MAC key* k_m. Décrire

une attaque qui montre que Alice et Bob n'ont en fait même plus de authenticated channel unidirectionnel, ni dans un sens ni dans l'autre. Indice: décrire une attaque permettant que Bob output un m sur $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$, que Alice n'aurait en fait jamais $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$.
(b) Ils essaient de réparer le protocole précédent de la façon suivante. Pour $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ une donnée m, Alice envoie désormais la concaténation (m, t, "from Alice"), toujours avec $t \leftarrow \mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, m)$. Pour $\mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}$, Bob vérifie désormais la condition supplémentaire que le message reçu est bien de la forme (m, t, "from Alice"). Idem pour l'autre sens ($\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$,$\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$). Même question.

La solution correcte pour implémenter un authenticated channel bidirectionnel avec une seule $\mathsf{k_m}$, est de réparer la question (b) de l'exercice précédent en générant le tag non pas sur m, mais sur la concaténation de "from Alice" avec m. Ajouter également "to Bob" s'ils sont plus de deux machines à utiliser la $\mathsf{k_m}$.

Exercice 27 (Alice peut-elle correctement vérifier si une donnée a été authentifiée par Bob, si elle-même a subi une influence side-channel dans le passé ?)**.** On considère Alice et Bob qui implémentent un authenticated channel bidirectionnel ($\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}, \mathsf{Receive}_{\mathrm{A}\to\mathrm{B}}, \mathsf{Send}_{\mathrm{B}\to\mathrm{A}}, \mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$), en utilisant la solution correcte ci-dessus (celle qui utilise la même clé $\mathsf{k_m}$, avec les en-têtes "from Alice" etc.). On suppose en outre que Alice a subi, au début du protocole, l'influence side-channel que l'adversaire a eu un accès à son API de $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet)$ (en plus de son accès autorisé aux API $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$ $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ de Alice). Maintenant, Alice ne subit plus cette influence.
(a) Sans faire d'autre hypothèse, est-il possible que, dans le passé ou dans le futur, les API $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet)$ et $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, \bullet)$ de Alice et Bob aient un comportement qui ne respecte pas la spécification Figure 1 d'un MAC ?
(b) Sans faire d'autre hypothèse, décrire une attaque qui pourrait se produire à partir de maintenant (pas seulement à l'époque où Alice était corrompue), et qui produit un comportement des API $\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$, $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ qui ne respecte pas la spécification Definition 7 d'un authenticated channel de Bob vers Alice. Indice: décrire un scénario où Alice output une donnée m sur $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$, alors qu'elle n'a jamais été prise en input de $\mathsf{Send}_{\mathrm{B}\to\mathrm{A}}$.

Une contre-mesure qui anéantit l'effet de l'attaque (b) est d'implémenter un authenticated channel bidirectionnel comme décrit au début de Section 4.1.3. C'est à dire, avec une MAC key $\mathsf{k_{mab}}$ pour Alice$\to$Bob et une autre $\mathsf{k_{mba}}$ pour Bob$\to$Alice. C'est ce qui est fait dans TLS 1.3. Évidemment, cette contre-mesure ne peut rien contre un adversaire qui apprendrait directement la clé $\mathsf{k_{mba}}$ stockée par Alice (qui l'utilise pour faire $\mathsf{MAC}.\mathrm{Verify}(\mathsf{k_{mba}}, \bullet)$).
L'exercice Exercice 27 illustre un nombre de points pas forcément intuitifs

(i) Un adversaire qui apprend des clés d'authentication stockées chez Alice, peut les utiliser contre Alice elle-même, en se faisant passer pour Bob. De façon spectaculaire, ce type d'attaques se produit aussi contre de (mauvais) protocoles asymétriques. Elles ont été décrites pour la première fois par Just-Vaudenay [JV96, §3.1], elles s'appellent *key compromise impersonation (KCI) attacks* [Kra05; Men05].

(ii) une autre contre-mesure consiste pour Alice et Bob, à un certain moment, à updater la (les) MAC keys de façon déterministe et non-interactive: $\mathsf{k_m} \leftarrow H(\mathsf{k_m})$. H est en gros une fonction de hachage. Précisément, dans TLS 1.3 il s'agit de la fonction HKDF. Cet update est fait pour de longues sessions, il s'appelle *update traffic keys* (plus précisément, ce qui est updaté dans TLS sont les clés de authenticated symmetric encryption Section 4.3). Elle garantit que l'attaque ne pourra plus avoir lieu à partir du moment où cette opération est faite, cela apporte donc un certain niveau de *post-compromise security*;

(iii) les deux contre-mesures précédentes n'aident en rien si $\mathcal{A}$ a appris la ou les MAC keys chez Alice. Cela montre une limitation inhérente à la vérification d'authentication avec la cryptographie symétrique: c'est

celle que le vérificateur doit connaître un secret, et perd toute garantie si l'adversaire a appris ce secret. Dans le cadre de l'identity verification, cette limitation est formalisée comme de la "weak security" dans [BS23, Definition 18.9].

(iv) la question (a) vs la (b) illustrent que les spécifications d'un protocole de haut niveau, tel que un authenticated channel, font l'hypothèse que l'adversaire a seulement accès aux API de haut niveau, et pas aux API de ses sous-composantes. Alors même que la spécification de chaque sous-composante, prise individuellement, par exemple $(\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet), \mathsf{MAC}.\mathrm{Verify}(\mathsf{k_m}, \bullet))$, est prévue pour résister à une utilisation arbitraire de son API par l'adversaire.

4.2 Symmetric encryption scheme

4.2.1 Définition simplifiée. Utilisation correcte.

Un symmetric encryption scheme ([Lin06, §3.22][BS23, §5.3] simplifiés) est deux algorithmes: SymEnc et SymDec qui respectent les spécifications dans la Figure 2. Il permet à une paire de machines, Alice et Bob, d'implémenter un secure channel (uni- ou bi-directionnel) entre elles dès qu'elles ont accès à un KeyExch et à un *authenticated channel* (uni- ou bi-directionnel) (Definition 7). Le protocole vérifie les spécifications d'un secure channel vis à vis d'Alice si elle ne subit pas de side-channel, i.e., autre que les leakages et influences tolérés dans (Section 1.8). On rappelle qu'il s'agit du leakage des états initiaux, du contrôle de $\mathcal{A}$ sur le insecure channel, et de l'utilisation par $\mathcal{A}$ des API d'Alice (d'envoi $\mathsf{Send}_{\mathrm{A}\to\mathrm{B}}$, et aussi de réception $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$ si implémentation d'un channel bidirectionnel). De même pour Bob. La méthode d'implémentation est la suivante. On la décrit pour le cas seulement unidirectionnel de Alice vers Bob.

- Alice et Bob appellent tous les deux KeyExch et reçoivent une (secret symmetric) encryption-decryption key $\mathsf{k_{ed}}$.
- Pour envoyer un plaintext secret p à Bob, Alice génère un *ciphertext* $c \leftarrow \mathsf{SymEnc}(\mathsf{k_{ed}}, p)$ qu'elle envoie à Bob sur le *authenticated channel*.
- Quand Bob reçoit un ciphertext c', le authenticated channel lui garantit que c'est Alice qui l'a envoyé. La *correctness* (ci-dessous) lui garantit que le plaintext $p' \leftarrow \mathsf{Dec}(\mathsf{k_{ed}}, c')$ est celui que Alice a utilisé pour générer c'. En conclusion, Bob a la garantie que p' est un plaintext que Alice a souhaité lui transmettre, donc il output p'.

Même si ce n'est pas requis par la spécification d'un secure channel, on remarque que cette implémentation règle automatiquement les problèmes d'émetteur, d'ordre, de doublons etc. grâce à l'utilisation du authenticated channel. En particulier, Alice et Bob pourraient donc utiliser la même clé, $\mathsf{k_{ed}}$, pour envoyer plaintexts dans l'autre sens Bob→Alice. Il se trouve que TLS 1.3 fait, au contraire, le choix d'utiliser une clé différente $\mathsf{k_{ed}}'$ pour implémenter le secure channel dans l'autre sens. La garantie de *secrecy* s'exprime par la résistance à un scénario appelé *security game* et formalisé dans la Figure 2. En gros elle garantit qu'un adversaire $\mathcal{A}$, qui pourrait faire un nombre illimité de requêtes aux API $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$ de Alice et de Bob, mais en boîte noire donc sans connaître ssk, est incapable de deviner le plaintext P d'un ciphertext c généré dans son dos. Plus précisément, c ne lui apprend aucune information *supplémentaire* sur p au delà de ce qu'il connaissait déjà.
$\mathcal{A}$ ne bénéficie pas plus que: (i) son influence totale sur les délais de transmission de messages sur le authenticated channel, (ii) de sa vue complète des messages envoyés sur ce channel, (iii) et de son pouvoir de déclencher l'envoi de ciphertexts par Alice et Bob sur ce channel via son accès à l'API $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$.

Enfin, pour implémenter le authenticated channel requis à partir du KeyExch, il suffit d'un insecure channel, grâce à l'utilisation d'un MAC, comme expliqué dans la Section 4.1. En compilant cela, on obtient la construction explicitée dans la Section 4.3.2.

Symmetric encryption scheme

API: $\mathsf{SymEnc}(\mathsf{k_{ed}}, P; \$) \rightarrow c$: en input $\mathsf{k_{ed}}$ une *(secret symmetric) encryption-decryption key* et P un *plaintext*, en output c un ciphertext.
$\mathsf{SymDec}(\mathsf{k_{ed}}, c') \rightarrow P'$: en input $\mathsf{k_{ed}}$ une *(secret symmetric) encryption-decryption key* et c' un ciphertext, en output P' un plaintext;

Correctness: $\forall$ $\mathsf{k_{ed}}$, $\forall$ P: $\mathsf{SymDec}(\mathsf{k_{ed}}, \mathsf{SymEnc}(\mathsf{k_{ed}}, P)) = P$;

Secrecy (IND-CPA, informelle): Elle se définit à l'aide du scénario suivant, appelé *semantic security game*. Soit une machine Alice qui ne subit pas de side-channel, et qui génère $\mathsf{k_{ed}} \xleftarrow{\$} \{0,1\}^\kappa$. Puis, Alice offre en outre à l'adversaire $\mathcal{A}$ un accès illimité à l'API $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$. C'est à dire que $\mathcal{A}$ n'a pas accès à la $\mathsf{k_{ed}}$ mais que, en input un plaintext P' de son choix, il récupère $c' \leftarrow \mathsf{SymEnc}(\mathsf{k_{ed}}, P')$ en output. Il peut recommencer pour autant de plaintexts $P'_1, P'_2, \ldots$ qu'il souhaite. À un moment, $\mathcal{A}$ choisit deux plaintexts P_0, P_1, les donne à Alice puis lui tourne le dos. Alice tire à pile ou face et génère le *challenge ciphertext* $c \leftarrow \mathsf{SymEnc}(\mathsf{k_{ed}}, P_i)$ pour l'un des deux plaintexts $i \in \{0,1\}$, et montre c à $\mathcal{A}$. À la fin du game, $\mathcal{A}$ donne son guess sur lequel des deux plaintexts P_0, P_1 utilisés pour générer c. Alors, la probabilité de guess correct de $\mathcal{A}$ est $1/2$, c'est dire qu'il ne gagne pas mieux qu'en tirant son guess à pile ou face.

Figure 2: La secrecy reste garantie dans un contexte multi-utilisateurs, e.g., avec Alice et Bob qui auraient obtenu une $\mathsf{k_{ed}}$ commune d'un KeyExch, de même que décrit dans la légende de la Figure 1

//Commentaires sur la secrecy. Le "taux de guess correct moyen" signifie: en moyenne en recommençant le game un grand nombre de fois, avec une nouvelle $\mathsf{k_{ed}}$ à chaque fois. En fait on peut montrer que c'est équivalent à une notion plus forte de secrecy, qui garantit que $\mathcal{A}$ n'apprend rien sur la corrélation entre plusieurs plaintexts envoyés au cours d'une même discussion. Ça se formalise comme un game où Alice tire à pile ou face une fois pour toutes un bit $b \in \{L, R\}$ ("left or right"). $\mathcal{A}$ envoie des paires de challenges à plusieurs reprises: (P^i_L, P^i_R) et à chaque fois Alice génère un ciphertext de P^i_b dans son dos. Alors $\mathcal{A}$ ne sait pas deviner b.
//Le pouvoir d'utilisation de l'API par $\mathcal{A}$ modélise l'influence totale que $\mathcal{A}$ peut avoir sur les *utilisateurs* d'Alice et Bob, dès lors que ces utilisateurs eux-mêmes ne bénéficient pas de plus d'influence sur, ou de leakage de, Alice et Bob, qu'un simple accès à leurs API (identiques) $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$ //et du leakage de leur état initial (Section 2.6). On remarque que, d'une certaine manière, le security game autorise implicitement $\mathcal{A}$ à accéder aussi aux API $\mathsf{SymDec}(\mathsf{k_{ed}}, \bullet)$ sur des inputs c' qui auraient été générés correctement avec l'API $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$, et qui ne soient pas égaux à l'un des challenge ciphertexts. En effet le security game permet de supposer, sans perte de généralité, que $\mathcal{A}$ a un contrôle total sur tous les utilisateurs de Alice et Bob n'ayant accès qu'à API $\mathsf{SymEnc}(\mathsf{k_{ed}}, \bullet)$, donc qu'il connait déjà leurs plaintexts, donc qu'en fait $\mathsf{SymDec}(\mathsf{k_{ed}}, \bullet)$ ne lui apprend rien de plus. La notion de sécurité CCA permet des security games dans lesquels $\mathcal{A}$ peut aussi mettre des inputs possiblement mal formés dans l'API $\mathsf{SymDec}(\mathsf{k_{ed}}, \bullet)$.

Exercice 28. (a) ([Lin06, Thm 3.21]) Montrer que SymEnc ne peut pas être déterministe, sinon il n'y aurait pas de secrecy. Indice: montrer que $\mathcal{A}$ aurait un taux de succès de 100% au security game. S'inspirer de l'exemple de Section 1.3.
(b) Montrer qu'un symmetric encryption scheme ne peut pas vérifier l'égalité (MAUVAIS) pour une cer-

taine k_{ed}:

(MAUVAIS) $$\forall c\ ,\ c \leftarrow \mathsf{SymEnc}(\mathsf{k}_{\mathrm{ed}}, \mathsf{SymDec}(\mathsf{k}_{\mathrm{ed}}, c))\ .$$

Indice: montrer qu'il serait alors déterministe.

Examples 29. Exemple de SymEnc déterministe donc mauvais: l'utilisation en mode ECB de n'importe quelle PRF. Exemple de SymEnc déterministe donc mauvais: celui des Japonais pendant WWII (Section 1.3).

4.2.2 Exemple de AES-CTR

On décrit dans la Figure 3 une implémentation d'un symmetric encryption scheme. Elle s'appelle *randomized counter mode* ([Lin06, Thm 3.33]), elle utilise comme ingrédient n'importe quelle PRF (Section 3.2). On la présente instanciée avec la PRF AES, et dans la version basique où la valeur initiale du compteur est choisie complètement aléatoirement ([BS23, §5.4.2]).

AES-CTR

$\mathsf{SymEnc}(k, P; \$)$**:** Parser le plaintext en une suite de blocks de 128 bits: $P = [P_1, \ldots, P_n]$ ($P_i \in \{0,1\}^{128}$);

- générer $r \xleftarrow{\$} \{0,1\}^{128}$ //c'est de là que vient la randomisation;
- pour tout $x \in \mathbb{Z}/2^{128}\mathbb{Z}$, on note $\langle x \rangle \in \{0,1\}^{128}$ le développement binaire de x sur 128bits (Section 2.1). On définit les valeurs successives: $\langle r \rangle, \langle r+1 \rangle, \langle r+2 \rangle \ldots$, appelées *counter*, à l'aide de *l'addition* $\underset{(\mathrm{mod}\ 2^{128})}{+}$ *dans* $\mathbb{Z}/2^{128}\mathbb{Z}$ //et non pas XOR;
- output le ciphertext: $c = [\, r\ ,\ \mathsf{AES}\big(k, \langle r+1 \rangle\big) \oplus P_1,\ \ldots,\ \mathsf{AES}\big(k, \langle r+n \rangle\big) \oplus P_n]$ //il fait $n+1$ blocks.

$\mathsf{SymDec}(k, c)$**:** est la même opération à l'envers: parser le ciphertext en une suite de blocs de 128 bits $c = [r, c_1, \ldots, c_n]$;

- output le plaintext: $P = [\mathsf{AES}\big(k, \langle r+1 \rangle\big) \oplus c_1,\ \ldots,\ \mathsf{AES}\big(k, \langle r+n \rangle\big) \oplus c_n]$ //il fait donc n blocks.

Figure 3: Implémentation d'un symmetric encryption scheme avec la méthode randomized counter mode, dans le cas particulier d'une valeur initiale de compteur r tirée complètement au hasard et avec la PRF AES. On rappelle (Section 2.1) que $\oplus$ = XOR entre vecteurs de 128 bits ($\{0,1\}^{128}$), aussi appelés *blocks*.

Quelques remarques. Dans certains cas il est précisé une méthode pour choisir r, dans ce contexte il est appelé *initialisation vector*. Dans ces méthodes il est précisé qu'il est *interdit* d'utiliser deux fois le même initialisation vector. Par exemple car cela révélerait le XOR de deux plaintexts, par exemple aussi à cause de la "forbidden attack" [Jou06, §3] décrite plus bas. On peut voir AES-CTR comme un exemple particulier d'un type d'implémentation de SymEnc, appelé *stream cipher*. Dans cet exemple, la fonction qui génère le *key stream* est $(k, r) \rightarrow [\mathsf{AES}\big(k, \langle r \rangle + 1\big)\ ,\ \mathsf{AES}\big(k, \langle r \rangle + 2\ , \ldots]$, qui est destiné à être XORé avec le plaintext.

Exercice 30. *(XOR-malléabilité)*
Soit un ciphertext $c = [r, c_1]$ qui a été généré comme $\mathsf{SymEnc}(k, P_1)$ avec P_1 un plaintext de taille 128 bits. Soit P_1' un plaintext. Vous êtes l'adversaire, vous connaissez seulement $[P_1, c, P_1']$. Calculer le ciphertext $c' = [r', c_1']$, constitué de 2 blocs de 128 bits, tel que $\mathsf{SymDec}(k, c') = P_1'$.

Objectif: trouver $[r', c'_1]$ tel que $\mathsf{SymDec}(k, [r', c'_1]) = P'_1$.
On exprime ce qu'on cherche:

(12) $$P'_1 = \mathsf{SymDec}(k, [r', c'_1]) := \mathsf{AES}(k, \langle r' \rangle + 1) \oplus c'_1 \ .$$

On exprime ce qu'on connait:

(13) $$c_1 = \mathsf{AES}(k, \langle r \rangle + 1) \oplus P_1$$

On va chercher à exprimer la quantité inconnue: $\mathsf{AES}(k, \langle r' \rangle + 1)$ en fonction de ce qu'on connait.
Comme on ne connait (indirectement) que $\mathsf{AES}(k, \langle r \rangle + 1)$, on fait donc le choix $r' := r$.
On exprime directement $\mathsf{AES}(k, \langle r \rangle + 1)$ en fonction de l'Equation (13), en la XORant par P_1 à gauche et à droite: $c_1 \oplus P_1 = \mathsf{AES}(k, \langle r \rangle + 1)$.
En remplaçant cette expression l'Equation (12), on obtient $P'_1 = c_1 \oplus P_1 \oplus c'_1$. On va isoler la quantité qu'on cherche, c'_1, seule à droite de l'égalité, en XORant à gauche et à droite par $c_1 \oplus P_1$. On obtient: $c_1 \oplus P_1 \oplus P'_1 = c'_1$ ce qui répond à la question.
Application numérique: calculer c'_1 connaissant $P_1 = (\underbrace{0\ldots0}_{123 \text{ bits}}\underbrace{01010}_{5 \text{ bits}})$, $c_1 = (\underbrace{0\ldots0}_{123 \text{ bits}}\underbrace{01111}_{5 \text{ bits}})$, $P'_1 = (\underbrace{0\ldots0}_{123 \text{ bits}}\underbrace{00011}_{5 \text{ bits}})$.

Morale: n'utiliser un symmetric encryption scheme que sur un authenticated channel.

4.2.3 Exemple du symmetric Regev

Les paramètres publics sont des nombres entiers n et q. On suppose pour simplifier que q est divisible par 4. Par exemple pour 128bits de sécurité, [Mel19] recommandent $n \geq 1024$ et $q \geq 2^{29}$. Toutes les additions et multiplications sont modulo q. On définit le produit scalaire entre deux vecteurs $\overrightarrow{x} = (x_1, \ldots, x_n) \in (\mathbb{Z}/q\mathbb{Z})^n$ et $\overrightarrow{y} = (y_1, \ldots, y_n)$, tous les deux de taille n et à coordonnées dans $\mathbb{Z}/q\mathbb{Z}$:

$$\langle \overrightarrow{x}, \overrightarrow{y} \rangle = x_1y_1 + x_2y_2 + \cdots + x_ny_n = \sum_{i=1}^{n} x_iy_i \in \mathbb{Z}/q\mathbb{Z}$$

On définit un symmetric encryption scheme, que l'on appelle "Regev", puisqu'il s'obtient comme une simplification du public key encryption scheme de [Reg09, §4]. Il est souvent utilisé comme cas d'école ([PS17], [Mic20, LWE], [Wu22, §8.3] et semble même implémenté par [Zam22]) pour introduire aux futurs standards post-quantiques //c'est aussi un cas d'école de *additively homomorphic encrytion scheme*, voir par exemple [Ben+11, §2.1] pour sa généralisation facile à des plaintexts dans Z/pZ.

Public parameters: q grand, on le suppose divisible par 4 pour simplifier, n grand.

Une encryption-decryption key est de la forme $\overrightarrow{\mathsf{s}} \in (\mathbb{Z}/q\mathbb{Z})^n$;

$\mathsf{SymEnc}(\overrightarrow{s}\ , m \in \{0,1\}; \$)$: générer $\overrightarrow{a} \xleftarrow{\$} (\mathbb{Z}/q\mathbb{Z})^n$;

- générer $e \xleftarrow{\$} \mathbb{Z}$ de *petite* taille: au maximum $e \in \left]\frac{-q}{4}, \frac{q}{4}\right[\bmod q = \left[0, \frac{q}{4}\right[\cup \left]\frac{3q}{4}, q\right[$;
- Output la *paire* $c := \left(\overrightarrow{a}, b := \langle \overrightarrow{a}, \overrightarrow{s} \rangle + e + \frac{q}{2} \in \mathbb{Z}/q\mathbb{Z}\right)$

$\mathsf{SymDec}\left(\overrightarrow{s} \in (\mathbb{Z}/q\mathbb{Z})^n\ , c' = \left(\overrightarrow{a'} \in (\mathbb{Z}/q\mathbb{Z})^n, b' \in \mathbb{Z}/q\mathbb{Z}\right)\right)$:

- $\widetilde{M'} \leftarrow b' - \langle \overrightarrow{a'}, \overrightarrow{s} \rangle$ //le plaintext noisy et dilaté;
- output $m' \leftarrow \mathsf{Compress}(\widetilde{M'})$ //c'est à dire: 0 si $\widetilde{M'_i} \in \left]\overline{-\frac{q}{4}}, \overline{\frac{q}{4}}\right[$ et 1 sinon, i.e., si $\widetilde{M'} \in \left[\frac{q}{4}, \frac{3q}{4}\right]$.

Exercice 31. Montrer la correctness, i.e., pour tout $m \in \{0,1\}$ et $\overrightarrow{s} \in (\mathbb{Z}/q\mathbb{Z})^n$, $c_m \leftarrow \mathsf{SymEnc}(\overrightarrow{s}, m; \$)$ alors $\mathsf{SymDec}(\overrightarrow{s}, c_m) = m$.

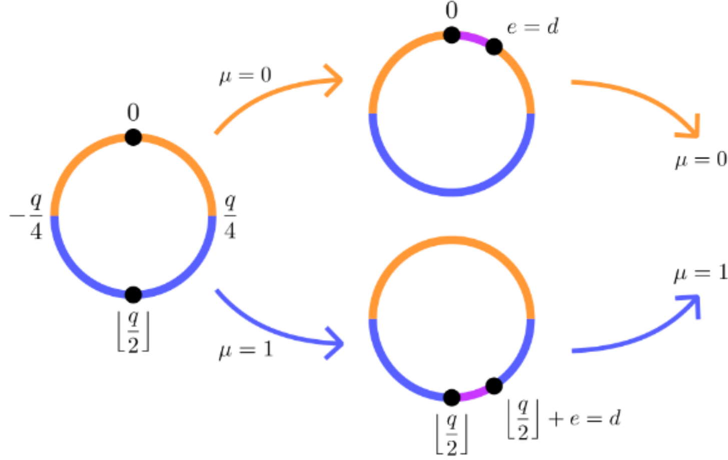

Figure 1: Illustration de la decryption dans symmetric Regev (from [Wu22, §8.3]). La lettre d symbolise $\widetilde{M}'$ le plaintext noisy et dilaté. $-\frac{q}{4}$ signifie en fait $\overline{-\frac{q}{4}} = \frac{3q}{4}$

4.2.4 Exemple du *symmetric LPR.*

On décrit une simplification de [LPR13, §8.3] //En fait, ce SymEnc est folklore depuis [LPR10, p4] puisque c'est en quelque sorte une simplification directe du public key encryption scheme qu'ils suggèrent. C'est un cas d'école pour introduire Kyber (Section 5.2.6). Il a aussi un intérêt en soi puisque il supporte des additions et des multiplications homomorphes //une variante est proposée dans [BV11, p5], et la description rigoureuse dans [LPR13, §8.3], en toute généralité pour des coefficients dans $\mathbb{Z}/p\mathbb{Z}$. Les paramètres publics sont q et n (grands, à préciser). Pour simplifier on suppose que q est divisible par 2. Soit $R_q := \frac{(\mathbb{Z}/q\mathbb{Z})[X]}{X^n+1}$ avec la loi de multiplication modulo X^n+1. Une symmetric encryption-decryption key est un élément $\mathsf{k}_{\mathsf{ed}} = \mathsf{s} \in R_q$

Paramètres: q grand, supposé divisible par 4 pour simplifier. $n = 256$. $R_q := \frac{(\mathbb{Z}/q\mathbb{Z})[X]}{X^n+1}$.

SymEnc(s, *m*; \$): pour $m = \sum_{i=0}^{n-1} m_i.X^i \in R_q$ à coefficients $m_i \in \{0,1\}$:

- générer $\mathsf{a} \xleftarrow{\$} R_q$;
- générer $e \leftarrow R_q$ de *petite* taille: au maximum $e \in \left]-\frac{q}{4}, \frac{q}{4}\right[\bmod q = \left[0, \frac{q}{4}\right[\cup \left]\frac{3q}{4}, q\right[$;
- output le ciphertext égal à la *paire* $c = \left(\mathsf{a}\,,\, \mathsf{a.s} + e + \frac{q}{2}m\right)$.

SymDec(s, *c*′ = (a′, *b*′)): calculer le plaintext noisy et dilaté $\widetilde{M}' := b' - \mathsf{s.a}$;

- output $\mathsf{Compress}(\widetilde{M}')$ //c'est à dire: soit $\widetilde{M}' = \sum_{i=0}^{n-1} = \widetilde{M}'_i.X^i$; alors pour chaque $i \in [0, \ldots, n-1]$: set $m'_i \leftarrow 0$ si $\widetilde{M}'_i \in \left]-\frac{q}{4}, \frac{q}{4}\right[\bmod q = \left[0, \frac{q}{4}\right[\cup \left]\frac{3q}{4}, q\right[$ et $m'_i \leftarrow 1$ sinon, i.e., si $\widetilde{M}'_i \in \left[\frac{q}{4}, \frac{3q}{4}\right]$.

Exercice 32. Bob a en input le plaintext $m = 0+1.X+0.X^2+\cdots+0.X^{n-1}$. Il génère $c \leftarrow \mathsf{SymEnc}(\mathsf{s}, m)$ avec un noise dont certains coefficients sont trop grands: $e = \frac{4q}{5} + 0.X + \frac{q}{3}X^2 + 0.X^3 + \cdots + 0.X^{n-1}$. Calculer $\mathsf{SymDec}(\mathsf{s}, c)$.

4.3 Authenticated symmetric encryption scheme

4.3.1 Définition simplifiée et utilisation avec KeyExch pour implémenter un secure channel.

Un authenticated symmetric encryption scheme ([BS23, §9.2] simplifiée) est deux algorithmes: AuthSymEnc et AuthSymDec qui vérifient les spécifications de la Figure 4. Avec cet outil, le protocole suivant permet à une paire de machines Alice et Bob d'implémenter un secure channel *unidirectionnel*, dès qu'elles ont accès à un KeyExch (Section 2.5) et à un insecure channel (Definition 6). Le protocole vérifie les spécifications d'un secure channel vis à vis d'Alice si elle ne subit pas de side-channel (Section 1.8). De même, le protocole vérifie les spécifications d'un secure channel vis à vis de Bob si il ne subit pas de side-channel.

Setup Alice et Bob appellent tous les deux KeyExch et reçoivent une *(secret) symmetric session key* ssk $\in \{0,1\}^\kappa$. Ils sont maintenant les *co-owners* de ssk. Alors, le protocole suivant implémente un secure channel entre eux. Par construction, il leur apporte en outre la garantie avec que leur correspondant est co-owner de la ssk. Si en outre Alice et Bob utilisent un KeyExch avec de l'identification (Section 2.5) one-sided, par exemple relativement à l'identité id_Q pour Bob, alors Alice a la garantie que son correspondant a l'identité id_Q. Dans la suite on considère que l'identification est two-sided, relativement à des identités notées id_P et id_Q.

$\mathsf{Send}_{\mathbf{A}\to\mathbf{Q}}(p)$ Pour implémenter communiquer un plaintext secret p à Bob (dont elle sait seulement qu'il a l'identité id_Q), Alice génère un *auth-ciphertext* $c \leftarrow \mathsf{AuthSymEnc}(\mathsf{ssk}, p; \$)$ qu'elle envoie à Bob sur le insecure channel.

$\mathsf{Receive}_{\mathbf{P}\to\mathbf{B}}$ Quand Bob reçoit un auth-ciphertext c', il calcule $\mathsf{AuthSymDec}(\mathsf{ssk}, c')$. Si l'output est reject, la *correctness* (ci dessous) entraîne que c' n'a pas été généré par l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet; \$)$, donc il output reject et ignore c'. Si l'output est au contraire un plaintext, p', la *ciphertext integrity* (ci-dessous) lui garantit que c' a été généré en utilisant l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet; \$)$. En outre, p a forcément été mis en input de l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet; \$)$ chez Alice (qu'il sait seulement être d'identité id_P), et non celle chez lui, puisqu'il ne s'en sert jamais. En outre, la *correctness* lui garantit que p' est l'input qui a été mis dans l'API pour générer c'. Donc Bob output p'.

Cette implémentation n'exclut pas que Bob puisse output plusieurs fois un plaintext qui n'a été $\mathsf{Sent}_{\mathrm{A}\to\mathrm{Q}}$ qu'une seule fois par Alice, ou des plaintexts $\mathsf{Sent}_{\mathrm{A}\to\mathrm{Q}}$ par Alice pas dans le même ordre. Ces détails se règlent facilement, comme expliqué dans la Remark 24.
TLS 1.3 impose des contraintes dans l'implémentation ci-dessus, appelées *record protocol* ([BS23, §9.8]). Notamment, elles rendent déterministes certains choix d'aléa dans $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet; \$)$. Techniquement il s'agit de la valeur initiale du counter, il est imposé qu'elle soit incrémentée à chaque nouvel envoi de plaintext.

Remark 33 (Better-than-advertized secrecy: ind-CCA)**.** En fait, il est prouvé dans [BS23, Theorem 9.1] que n'importe quel symmetric authenticated encryption scheme, i.e., vérifiant les spécifiations de Figure 4, résiste en fait à des security games de secrecy et de ciphertext integrity strictement plus difficiles. Il s'agit des mêmes security games, sauf qu'on tolère en plus que $\mathcal{A}$ ait aussi accès à l'API $\mathsf{AuthSymDec}(\mathsf{ssk}, \bullet)$, sauf

Authenticated symmetric encryption

API: $\mathsf{AuthSymEnc}(\mathsf{ssk}, P; \$) \to c$: en input ssk une symmetric session key et P un plaintext, en output c un *auth-ciphertext*.
$\mathsf{AuthSymDec}(\mathsf{ssk}, c') \to \{\mathsf{reject} \text{ or } P'\}$: en input ssk une symmetric session key et c' un auth-ciphertext, en output: soit reject, soit P' un plaintext.

Correctness: idem ($\forall$ ssk, $\forall$ P: $\mathsf{AuthSymDec}(\mathsf{ssk}, \mathsf{AuthSymEnc}(\mathsf{ssk}, P)) = P$);

Secrecy: idem que pour un symmetric encryption scheme (accès à $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet)$);

Ciphertext integrity: on considère le security game suivant. On considère une machine, Alice, qui n'a pas de side-channel, et génère $\mathsf{ssk} \xleftarrow{\$} \{0,1\}^*$. Puis, elle donne accès à $\mathcal{A}$ à l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet)$ (donc sans lui montrer directement ssk). Alors $\mathcal{A}$ est incapable de produire un auth-ciphertext c qu'il n'aurait pas obtenu comme output de l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet)$, et tel que $\mathsf{AuthSymDec}(\mathsf{ssk}, c) \neq \mathsf{reject}$.

Figure 4

bien sûr pour l'input égal au auth-ciphertext donné en challenge par Alice. L'intuition de cette résistance est que, soit un auth-ciphertext c que $\mathcal{A}$ n'a pas lui-même généré comme requête à $\mathsf{AuthSymEnc}(\mathsf{k_{ed}}, \bullet)$, alors la ciphertext integrity implique que $\mathsf{AuthSymDec}(\mathsf{k_{ed}}, c) = \mathsf{reject}$. Donc une requête de $\mathsf{AuthSymDec}(\mathsf{k_{ed}}, \bullet)$ sur tout ciphertext c ne va rien lui apprendre de plus que ce qu'il savait déjà, puisque soit il connaissait déjà le plaintext p, soit non et alors il sait déjà que la requête va lui renvoyer reject. Ce niveau de résistance plus élevé est appelé "ind-CCA" (l'analogue asymétrique, hors programme, est discuté dans Section 5.2.4).

Pour implémenter un secure channel bidirectionnel, on peut utiliser l'une des deux méthodes suivantes. Ce sont les analogues exactes de celles décrites dans Section 4.1.3 pour implémenter un authenticated channel bidirectionnel avec un MAC, donc on ne les détaille pas plus. La première consiste pour Alice et Bob à appeler une deuxième fois KeyExch, et donc à recevoir une deuxième clé, ssk', qu'ils utilisent pour implémenter un secure channel unidirectionnel dans l'autre sens Bob→Alice. C'est plus ou moins la méthode de TLS 1.3. La deuxième consiste pour Alice et Bob à concaténer leurs plaintexts avec "from Alice", resp., "from Bob" avant de les mettre en input de $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet)$. Comme on l'a vu dans Exercice 27, la deuxième méthode ne garantit pas d'authentication dans un modèle plus défavorable, où l'adversaire aurait un accès direct à l'API $\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet)$ (et pas seulement aux API $\mathsf{Send}_{\mathrm{A}\to\mathrm{Q}}$ et $\mathsf{Send}_{\mathrm{B}\to\mathrm{P}}$).

Exercice 34 (Non malléabilité)**.** On a décrit un protocole au début Section 4.3.1. On a affirmé qu'il implémente un secure channel, mais on ne l'a pas prouvé. Le but de l'exercice est de prouver une partie de cette affirmation, donc, sans partir du principe que l'affirmation est vraie. Précisément: prouver que le protocole implémente un channel qui est *authenticated* (on rappelle que cela fait partie des conditions requises pour être secure).
Indice: montrer que (contrairement à Exercice 30), Bob ne peut pas output un plaintext qui n'aurait pas été $\mathsf{Sent}_{\mathrm{A}\to\mathrm{B}}$ par Alice.

4.3.2 Exemple: la classe d'implémentations *Encrypt-then-Mac*

On décrit dans la Figure 5 une classe d'implémentations particulière de authenticated symmetric encryption scheme, appelée *Encrypt-then-Mac* [BS23, Thm 9.2]. Elle utilise comme ingrédients n'importe quel symmetric encryption scheme $(\mathsf{SymEnc}, \mathsf{SymDec})$ et MAC $(\mathsf{MAC.Sign}, \mathsf{MAC.Verify})$.

Encrypt-then-MAC

Ingrédients. Utilise en sous-routine un MAC (MAC.Sign, MAC.Verify) et un symmetric encryption scheme (SymEnc, SymDec).

Keys Les deux participants parsent ssk en $(\mathsf{k_m} \in \{0,1\}^\kappa, \mathsf{k_{ed}} \in \{0,1\}^\kappa)$.

Implémentation de AuthSymEnc(ssk, P): $c \leftarrow \mathsf{SymEnc}(\mathsf{k_{ed}}, P)$, $t \leftarrow \mathsf{MAC.Sign}(\mathsf{k_m}, c)$. Output le auth-ciphertext (c, t), qu'on appellera aussi dans cet exemple un *tagged ciphertext.*

AuthSymDec(ssk, (c', t')): Calculer *d'abord* $\mathsf{MAC.Verify}(\mathsf{k_m}, c', t')$. Si ça renvoie reject//c'est donc que (c', t') n'a pas été généré par un owner de ssk, output reject et ignorer (c', t'). Si ça renvoie accept//c'est donc que (c', t') a été généré par un owner de ssk, output le plaintext $P' = \mathsf{SymDec}(\mathsf{ssk}, c')$.

Figure 5: Définition simplifiée de la classe d'implémentations particulière de authenticated symmetric encryption, appellée *Encrypt-then-Mac*

//Attention ([BS23, §9.4.1.1]) "A common mistake when implementing encrypt-then-MAC is to use the same key for the [SymEnc] and the MAC [c'est à dire choisir $\mathsf{k_{ed}} = \mathsf{k_m}$]. The resulting system need not provide authenticated encryption and can be insecure, as shown in Exercise 9.8. In the proof of Theorem 9.2 we relied on the fact that the two keys are chosen independently." C'est pourquoi dans Encrypt-then-MAC il faut parser ssk en $(\mathsf{k_m}, \mathsf{k_{ed}})$ et utiliser $\mathsf{k_m}$ pour le MAC et $\mathsf{k_{ed}}$ pour SymEnc.

//Une autre classe d'implémentations de authenticated symmetric encryption, qui est dangereuse, s'appelle "Mac-then-Encrypt". Elle consiste à générer un tag non pas sur le ciphertext, mais sur le plaintext: $t \leftarrow \mathsf{MAC.Sign}(\mathsf{ssk}, P)$, puis à générer un ciphertext c de la concaténation $P \| t$. Cela oblige Bob à d'abord *utiliser la clé secrète* ssk *pour déchiffrer* c *et récupérer le tag* t, avant de pouvoir tester (de nouveau avec la ssk) si (c, t) est accept ou reject. Voir Section 9.1 pour un aperçu d'attaques sur cette classe: POODLE contre SSL 3.0 ([BS23, §9.4.2]), et Lucky13 contre TLS 1.0, 1.1 et 1.2 [AP13].

4.3.3 Exemple: AES-GCM

AES-GCM est le seul authenticated encryption scheme qui est obligatoire dans TLS 1.3 [IET18b, §9.1]. TLS 1.3 utilise la version standardisée par le NIST en 2007 dans [NIS07]. On commence par des rappels. Soit $\mathrm{GF}(2^{128}) = \mathbb{Z}/2\mathbb{Z}[x]/D$, avec la multiplication modulo D, pour $D = x^{128} + x^7 + x^2 + x + 1$.

Exercice 35. Soit $Q = x^{66} + x^{65}$, calculer $\overline{Q.x^{64}}$.
Solution: $= \overline{x^{130} + x^{129}}$. On retranche le multiple de D: $x^2.D$ pour faire baisser le degré:
$= \overline{x^{129} - (x^9 + x^4 + x^3 + x^2)}$.
Il faut retrancher *encore* un multiple de D: $x.D$ pour *encore* faire baisser le degré:
$= \overline{x^9 + x^4 + x^3 + x^2 - (x^8 + x^3 + x^2 + x)}$
$= \overline{x^9 + x^8 + x^4 + x}$
Ce qu'il y a sous la barre est de degré $\leqslant 128 - 1 = 127$ donc égal à son propre reste modulo D donc $= x^9 + x^8 + x^4 + x$

On décrit maintenant une version simplifiée de AES-GCM. La version complète permet en plus d'authentifier en même temps un message en clair annexé au ciphertext, par exemple des métadonnées. Sa structure ressemble de loin à Encrypt-then-MAC appliqué à AES-CTR. En fait elle s'en écarte sur plusieurs points, détaillés plus bas. On le présente donc en bloc.

Clés parser ssk en $\mathsf{k_{ed}}$. Calculer $H := \mathsf{AES}(\mathsf{k_{ed}}, 0)$ la *hash key*;

AuthSymEnc$\big((\mathsf{k}_{\mathsf{ed}}, H), P = [P_1, \dots, P_n]; \$\big)$:

- Output le ciphertext obtenu en utilisant l'algorithme AES-CTR (Section 4.2.2): $c = [r, c_1, \dots, c_n] \leftarrow \mathsf{SymEnc}(\mathsf{k}_{\mathsf{ed}}, P; \$)$;
- Interpréter H et chacun des vecteurs de 128 bits: $c_1, \dots, c_n$ comme des éléments dans $\mathrm{GF}(2^{128})$, c'est à dire des polynômes de degré $\leqslant 128 - 1 = 127$. Calculer $GHASH = \sum_{i=1}^{n} c^i H^{n-i+1}$ Output le message authentication tag: $t := \mathsf{AES}(\mathsf{k}_{\mathsf{ed}}, r) + GHASH$;

AuthSymDec$\big((\mathsf{k}_{\mathsf{ed}}, H), c' = [r', c'_1, \dots, c'_n], t'\big)$: recalculer $GHASH' = \sum_{i=1}^{n} c'^i H^{n-i+1}$ et tester l'égalité $t' \ =?\ \mathsf{AES}(\mathsf{k}_{\mathsf{ed}}, r') + GHASH'$. Si elle n'est pas vérifiée, output reject. Si elle l'est, output avec le même algorithme que AES-CTR: $m' \leftarrow \mathsf{SymDec}(\mathsf{k}_{\mathsf{ed}}, c')$.

//La version décrite utilise le AES-CTR tel que décrit dans Section 4.2.2, c'est à dire avec un compteur r choisi complètement aléatoirement. C'est autorisé dans [NIS07, p. 8.2.2]. Une référence non-officielle mais plus lisible de AES-GCM est [Jou06, §2] modulo quelques changements depuis 2006 dans la méthode de génération de r, qu'on ne discute pas. Les écarts par rapport à Encrypt-then-MAC sont les suivants. Tout d'abord la clé de symmetric encryption, k_{ed}, est aussi utilisée pour calculer le authentication tag t. Cela se produit lorsque k_{ed} est utilisée pour construire la hash key H, et de nouveau à l'étape finale du calcul de t. C'est donc une (double) dérogation à la règle qui demande que les clés utilisées pour l'encryption et le MAC soient indépendantes. De telles dérogations sont parfois attaquables ([BS23, §9.4.1.1]). D'autre part, il n'est pas connu si le MAC utilisé répond bien aux spécifications d'un MAC en général. Il le serait si *tout* le ciphertext était mis en input du calcul de $GHASH$ (ce serait alors le MAC de Carter-Wegman). Or ce n'est pas le cas puisque le counter r n'est mis en input //ce point n'est pas explicite dans [BS23, §9.7], il sera corrigé dans la prochaine version. Donc cela semble proche d'une construction facilement attaquable ([BS23, §9.4.1.1]). Cet écart est rattrapé en quelque sorte par le fait que le counter r est réutilisé dans le calcul final du tag t. On remarque que cette réutilisation est en quelque sorte elle-même une dérogation //techniquement, t est utilisé comme nonce dans le MAC de Carter-Wegman. Cela ne respecte donc pas la spécification de Carter-Wegman ([BS23, §7.4]), qui est que le nonce doit être généré indépendemment de la donnée en input du MAC. Malgré tous ces écarts, il est prouvé dans [IOM12] que AES-GCM est bien un authenticated encryption scheme.
La "forbidden attack" de [Jou06, §3] s'applique à une Alice qui génère mal r au hasard, au point de réutiliser le même r. Soient t, t' les deux tags publics produits, on obtient que $GHASH + GHASH' - t - t' = 0$. Donc on voit que l'évaluation en H d'un polynôme public est nulle, i.e., H est une racine. Dans [Jou06, §3] il est expliqué comment $\mathcal{A}$ peut retrouver la bonne hash key H parmi les racines. À partir de H, $\mathcal{A}$ peut fabriquer un tag valide sur *n'importe quel ciphertext de son choix*, y compris formé avec un autre compteur que r. En particulier, si $\mathcal{A}$ observe un ciphertext public c dont il connaîtrait le plaintext (par exemple "bonjour"), il peut appliquer l'Exercice 30 pour fabriquer un ciphertext c' qui va se déchiffrer en un plaintext choisi par $\mathcal{A}$. A l'aide du H, il fabrique un tag t' sur c' qui sera accept par Bob. Dans [Jou06, §3] il décrit aussi une modification de cette attaque qui permet d'attaquer la version du NIST avant 2006 de AES-GCM, même contre des Alice qui ne réutilisent pas le même r.

5 Cryptographie asymétrique

Ce sont des algorithmes qui ne nécessitent pas, pour répondre à leurs spécifications, que les participants connaissent au préalable le même secret en commun.

5.1 Le authenticated key exchange de Diffie-Hellman entre peers pré-spécifiés

L'algorithme de la Figure 6 décrit un protocole de key exchange entre des peers pré-définis Alice et Bob, au sens de la Definition 8. Il nécessite comme setup que Alice et Bob soient reliées par un authenticated channel (soit anonyme, soit relativement à des identités qu'elles possèdent, e.g., id_P et/ou id_Q). Les paramètres publics sont un groupe $(\mathbb{G}, +, 0)$ de cardinal connu $|\mathbb{G}| = q$, et un générateur $G \in \mathbb{G}$. On note, pour tout $u \in \mathbb{Z}$ et $H \in \mathbb{G}$: $[u].H := \underbrace{H + \cdots + H}_{u \text{ fois}}$. On rappelle que *générateur* signifie que $\mathbb{G} = \{[u].G \ , \ u \in \mathbb{Z}/q\mathbb{Z}\}$.

On le décrit d'abord dans sa version basique.

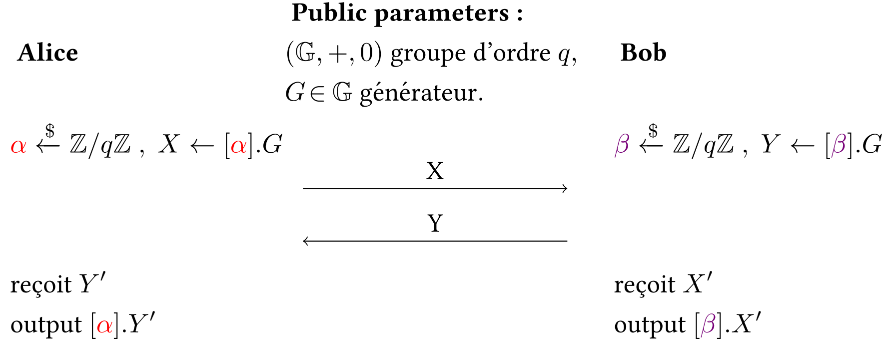

Figure 6: Key-exchange de Diffie Hellman sur un *authenticated channel* entre deux machines Alice et Bob. Dans le cas particulier authenticated anonyme, elles n'ont pas forcément d'information a priori l'une sur l'autre. Les messages peuvent être envoyés en parallèle (comme dans [JKL04]), i.e., celui de Bob sans attendre celui d'Alice, idem pour Alice→Bob. Le authenticated channel garantit que le X' reçu par Bob est égal au X envoyé par Alice, idem que le Y' reçu par Alice est bien le Y envoyé par Bob.

Le authenticated channel garantit que le X' reçu par Bob est égal au X envoyé par Alice, et idem pour le $Y = Y'$. Donc, les outputs sont les mêmes: $[\alpha].Y = [\alpha].[\beta].G = [\beta].X$. Cet output commun est appelé *master secret* dans TLS. Il n'est a priori pas un aléa uniforme, donc Alice et Bob lui appliquent, chacun, une *key derivation function* (KDF) pour en déduire la (les) clés secrètes symmétriques ssk. Une KDF est une fonction déterministe, qui se comporte en gros comme une fonction de hachage, avec des propriétés supplémentaires (capacité à extraire de l'aléa), et une taille d'output ajustable. Celle utilisée dans TLS s'appelle "HKDF" ([BS23, §8.10.5]). Pour que Diffie-Hellman satisfasse la propriété d'un key exchange, il faut que l'adversaire $\mathcal{A}$ n'apprenne pas l'output malgré sa connaissance des messages X et Y publics. Il faut donc, en particulier, qu'il ne soit pas capable de calculer les logs discrets de X et Y en base G, qui sont α et β. Le site de l'ANSSI [ANS21, §5.3] conseille des paramètres réputés donner cette garantie.

Proposition 36. *Sous l'hypothèse que les paramètres apportent bien la garantie que $\mathcal{A}$ n'apprend rien sur* ssk *sur la base de X et Y, alors Diffie-Hellman (Figure 6), lorsqu'il est réalisé sur un* authenticated channel *(vanilla , ou avec identification one/two-sided relativement aux identités id_P et/ou id_Q), implémente un key exchange (vanilla , ou avec identification one ou two-sided (Definition 8) relativement à id_P et/ou id_Q).*

En particulier si le authenticated channel utilisé n'est que vanilla , alors a priori vanilla Diffie-Hellman n'implémente donc qu'un vanilla KeyExch, donc le secure channel obtenu en utilisant ssk ne sera que vanilla . Donc a priori, comme expliqué dans Section 1.6, sauf ingrédients supplémentaires, Alice et Bob ne pourront jamais exclure qu'ils ne soient pas dans la situation "man-in-the-middle". Enfin, si Alice et

Bob exécutent Diffie-Hellman tel quel sur un *insecure* channel, alors c'est encore plus compliqué: on y reviendra dans Sections 7.1 and 7.2.

5.2 Public key encryption scheme (PKE)

5.2.1 Introduction: implémentation d'un secure channel avec Encrypt-then-authenticate. Problème si plusieurs émetteurs.

Un public key encryption scheme (PKE) ([BS23, §11.2] simplifiée) est trois algorithmes: KeyGen, Enc et Dec avec les spécifications dans la Figure 7 Patrick a accès à un authenticated channel avec Quiterie. Quiterie peut être n'importe quelle quidam pour Patrick, et réciproquement, i.e., ils peuvent n'avoir jamais interagi ni n'ont de (secret) symmetric session key en commun. Par exemple, ils se rencontrent dans un lieu public. Le protocole suivant permet d'implémenter un secure channel unidirectionnel de Quiterie vers Patrick.

- Patrick exécute $(\mathsf{dk}, \mathsf{ek}) \leftarrow \mathsf{KeyGen}(\$)$;
- il envoie (publiquement) sa *public encryption key* ek à Quiterie;
- Pour Send un plaintext P à Patrick, Quiterie génère un *ciphertext* $c \leftarrow \mathsf{Enc}(\mathsf{ek}, P; \$)$ et envoie c sur le authenticated channel;
- Patrick, lorsqu'il reçoit un c' de Quiterie, utilise la *secret decryption key* dk pour calculer le plaintext $P' = \mathsf{Dec}(\mathsf{dk}, c')$. Il output P' sur l'API Receive.

Il est prouvé, dans [CDN15, §4.2.2-§4.2.6], que ce protocole conserve toutes les garanties d'un secure channel unidirectionnel même lorsqu'il est composé de façon arbitraire avec d'autres protocoles. En particulier, cette garantie permet à Patrick d'exécuter plusieurs instances de ce protocole avec plusieurs Quiterie$_i$ arbitraires, possiblement en parallèle, telles que chaque instance i implémente bien un secure channel de $Quiterie_i$ vers Patrick. Mais qui dit plusieurs instances, dit en particulier obligation pour Patrick de refaire la première étape pour chaque instance distincte. C'est à dire qu'il doit générer une paire de clés *distincte* $(\mathsf{dk}_i, \mathsf{ek}_i)$ pour chaque Quiterie$_i$ Donc ce protocole oblige Patrick à stocker (on dit "gérer") une paire de clés distincte pour chaque Quiterie$_i$. En outre, ce protocole impose un échange préliminaire avec chaque sender distinct Quiterie$_i$. C'est à dire que, avant de pouvoir envoyer des plaintexts à Patrick, Quiterie$_i$ doit lui demander de générer une nouvelle paire de clés, puis attendre de recevoir ek_i. On aimerait donc une solution qui enlèverait les deux contraintes. C'est à dire qu'elle soit aussi simple pour Patrick que de générer une paire de clés $(\mathsf{sk}, \mathsf{ek})$ une fois pour toutes, et d'afficher sur son front ek lorsqu'il se promène en public. Il pourrait donc recevoir des plaintexts de toute Quiterie$_i$ qu'il croiserait, sans jamais devoir générer ni lui envoyer quoi que ce soit. On verra deux exemples de fausses solutions naïves, dans Exercice 41 et Exercice 45, qui échouent à implémenter des secure channels depuis plusieurs senders. Les attaques permettent à chaque fois un hijacking de l'origine du plaintext (Section 1.4). On expliquera dans Section 5.2.5 et Section 5.3.2 comment transformer ces échecs en solutions. Mais le prix à payer sera l'utilisation d'un PKE avec des spécifications strictement plus fortes (en particulier non-malléabilité), que celles données dans Figure 7 ("semantic security", a.k.a. ind-CPA).

5.2.2 Définition/specification simplifiée, nécessité de randomisation

Exercice 37. b) ([BS23, §11.3.1]) Montrer que Enc ne peut pas être déterministe, sinon il n'y aurait pas de secrecy. Exemple de mauvais Enc déterministe: le RSA sans random padding de [Bis18], cf aussi [Bit19].
a) Montrer qu'un public key encryption scheme ne peut pas vérifier l'égalité (MAUVAIS) pour une certaine

Public key encryption scheme (PKE)

API: $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\$)$: un algorithme randomisé qui renvoie en output ek une *public encryption key*, et dk une *secret decryption key*;
$c \leftarrow \mathsf{Enc}(\mathsf{ek}, P\,;\$)$: en input ek une public encryption key et P un plaintext, en output c un ciphertext;
$P' \leftarrow \mathsf{Dec}(\mathsf{dk}, c')$: en input dk une secret decryption key et c' un plaintext, en output P' un plaintext ou reject.

Correctness: $\forall(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\$)$, $\forall P$, $\mathsf{Dec}(\mathsf{dk}, \mathsf{Enc}(\mathsf{ek}, P\,;\$)) = P$.

Secrecy (ind-CPA) On considère le scénario suivant, appelé *semantic security game* (ou chosen plaintext attack (CPA) game). Soit Alice une machine sans side-channel qui génère $(\mathsf{ek}, \mathsf{dk}) \stackrel{\$}{\leftarrow} \mathsf{KeyGen}(\$)$ et donne ek à l'adversaire $\mathcal{A}$. Puis, $\mathcal{A}$ choisit lui-même deux plaintexts (P_0, P_1) et les donne à Alice. Alice génère un ciphertext $c \leftarrow \mathsf{Enc}(\mathsf{ek}, P_i)$ de l'un des deux ($i \in \{0,1\}$) sans dire lequel à $\mathcal{A}$ et lui montre c. Alors, $\mathcal{A}$ ne sait même pas reconnaître s'il a été produit comme $c \leftarrow \mathsf{Enc}(\mathsf{ek}, P_0; \$)$ ou $c \leftarrow \mathsf{Enc}(\mathsf{ek}, P_1; \$)$ //quand bien même $\mathcal{A}$, puisqu'il connait ek, est capable de se générer beaucoup de ciphertexts $c \leftarrow \mathsf{Enc}(\mathsf{ek}, P_i; \$)$ de P_0 et de P_1.

Figure 7: Spécifications d'un PKE

paire $(\mathsf{ek}, \mathsf{dk})$:

(MAUVAIS) $$\forall c\,,\ \$\ ,\ c \leftarrow \mathsf{Enc}(\mathsf{ek}, \mathsf{Dec}(\mathsf{dk}, c); \$)\ ,$$

sinon il serait déterministe.

5.2.3 Exemples

Hybrid encryption À partir de n'importe quel PKE $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ destiné à des plaintexts courts, il est possible de le compiler en un autre, potentiellement plus efficace sur des plaintexts longs, que la simple répétition de Enc sur chaque morceau de plaintext. Cette construction est parfois appelée "hybrid" encryption. Elle utilise n'importe quel symmetric encryption scheme $(\mathsf{SymEnc}, \mathsf{SymDec})$. Pour $\mathsf{Enc}(\mathsf{ek}, m; \$)$: générer $\mathsf{k}_{\mathsf{ed}} \stackrel{\$}{\leftarrow} \{0,1\}^*$ et $c \leftarrow \mathsf{SymEnc}(\mathsf{k}_{\mathsf{ed}}, m; \$)$, ouput la paire $\big(\mathsf{Enc}(\mathsf{ek}, \mathsf{k}_{\mathsf{ed}}; \$), c\big)$. Pour $\mathsf{Dec}(\mathsf{dk}, (c_k, c))$: $\mathsf{k}_{\mathsf{ed}} \leftarrow \mathsf{Dec}(\mathsf{dk}, c_k)$, output $\mathsf{Dec}(\mathsf{k}_{\mathsf{ed}}, c)$. La construction a un intérêt pour les longs plaintexts, car les algorithmes de symmetric encryption sont environ 1000x plus rapides que ceux de public key encryption. Cette construction est prouvée être un PKE dans [BS23, exercice 11.9] (sous des hypothèses même plus faibles sur PKE, appelées "key encapsulation"). On voit qu'elle est au moins aussi faible que le SymEnc utilisé. Concrètement, s'il s'agit de AES-CTR, alors les ciphertexts sont malléables (Exercice 30). On verra comment résoudre ce problème dans Section 5.2.4.

Elgamal Le PKE de Elgamal peut être décrit, en caricaturant, comme Diffie-Hellman (Section 5.1) avec une syntaxe différente. Le message X de Alice est sa ek, et α sa dk. Pour Enc un plaintext $P \in \mathbb{G}$, Bob génère $\beta \stackrel{\$}{\leftarrow} \mathbb{Z}/q\mathbb{Z}$ et $Y \leftarrow [\beta].G$ comme dans Diffie-Hellman, et output le ciphertext constitué de la paire $c := (Y\ ;\ P + [\beta].X)$. On voit donc que le plaintext P est masqué par la quantité secrète $[\beta].X = [\alpha].Y$ connue à la fois de Alice et de Bob. Ainsi, pour Dec, il suffit à Alice de retirer ce masque. Une variante de Elgamal s'applique à des plaintexts $x \in \mathbb{Z}/q\mathbb{Z}$ de petite taille. Elle consiste

à utiliser Elgamal sur $M := [x].G$. Alice extrait alors le log discret de M pour retrouver x. Cette variante est en cours de standardisation [ISO19]. Enfin une dernière variante ([BS23, §11.5]), qui est plus efficace sur des plaintexts longs, consiste à utiliser la quantité secrète $[\beta].X$ comme seed d'un PRG pour générer un long masque. Comme $[\beta].X$ n'est pas uniformément distribué et n'a pas la bonne longueur, il faut d'abord lui appliquer une *key derivation function* (KDF [BS23, §8.10]) pour en extraire une seed. C'est donc très proche de l'exemple de hybrid encryption ci-dessus. En caricaturant, c'est la méthode utilisée dans TLS 1.3: le $[\beta].X$ de Diffie-Hellman s'appelle le *master secret*, et les clés de session en sont extraites avec une KDF.

RSA est un PKE interdit depuis TLS 1.3.

le PKE **de Regev** ([Reg09, §4]) peut se décrire comme obtenu par une compilation du symmetric encryption scheme décrit dans Section 4.2.3. De façon alternative, il peut se décrire de façon plus compacte en notation matricielle. Sa variante la plus récente semble être [Ben+11, §2.1], qui prend notamment des inputs dans $\mathbb{Z}/p\mathbb{Z}$ (et plus seulement dans $\{0,1\}$). Il permet des additions homomorphes et est post-quantique, mais il y a plus efficace depuis, par exemple [LPR10] (Section 5.2.6).

5.2.4 Hors programme mais nécessaire: ind-CCA, en particulier *non-malléabilité*.

Définition 38 (ind-CCA ([BS23, §12]))**.** On dit qu'un PKE a un niveau de secrecy qui résiste aux *chosen ciphertext attacks*, si, dans le security game définissant la secrecy dans la Figure 7, l'adversaire $\mathcal{A}$ a accès, en outre, à l'API $\mathsf{Dec}(\mathsf{dk}, \bullet)$ de Alice, sauf bien sûr pour l'input égal au c donné en challenge par Alice. On appelle *ind-CCA* ce niveau de sécurité, qui est donc plus élevé que celui de la Figure 7.

Un sous-produit de cette nouvelle définition renforcée, est qu'elle exclut les attaques du type de celle décrite dans Section 5.2.3

Exercice 39 (ind-CCA⇒ non-malleable)**.** Montrer qu'un $\mathrm{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ qui est ind-CCA est forcément *non-malléable*, par exemple au sens suivant. Soit une machine Alice sans side-channel qui exécute $(\mathsf{ek}, \mathsf{dk}) \xleftarrow{\$} \mathsf{KeyGen}$. Montrer qu'il ait impossible que l'adversaire $\mathcal{A}$ ait un algorithme $\mathscr{M}$ permettant la chose suivante. Pour toute machine Cyrano sans side-channel qui génère secrètement un plaintext m, crée la concaténation (h, m) avec, par exemple, le header h=“`généré par Cyrano`”, en génère un ciphertext $c \leftarrow \mathsf{Enc}(\mathsf{ek}, (h, m); \$)$, qu'elle donne à $\mathcal{A}$, alors $\mathscr{M}(c) = c'$ tel que $\mathsf{Dec}(\mathsf{dk}, c') =$ (“`généré par Christian`”,m). Indice: voir [BS23, §12.2.1].

Évidemment, même ind-CCA, un PKE ne permet pas d'authentication:

Exercice 40. Question subsidiaire: supposons que la machine Cyrano concatène *tous* les plaintexts qu'elle met en input de $\mathsf{Enc}(ek, \bullet)$ avec ce même header h=“`généré par Cyrano`”. Soit un ciphertext c tel que $\mathsf{Dec}(\mathsf{sk}, c) = (h, m)$ avec h=“`généré par Cyrano`”, décrire un scénario où (h, m) n'a jamais été mis en input de $\mathsf{Enc}(ek, \bullet)$ par Cyrano.

Il est possible de renforcer de même la définition de symmetric encryption. Précisément, on peut rendre le Figure 2 plus facile pour $\mathcal{A}$ en supposant qu'il a aussi accès à l'API $\mathsf{Dec}(\mathsf{k}_{\mathsf{ed}}, \bullet)$ sauf bien sûr pour l'input égal au challenge ciphertext c. En fait, il est prouvé dans [BS23, Theorem 9.1] que authenticated symmetric encryption satisfait automatiquement cette définition renforcée (cf Remark 33). Des exemples de ind-CCA PKE sont:

la construction générale de [BS23, §12.3]. Par exemple, *instanciée* en utilisant RSA en *sous-routine*, tel que décrit dans [BS23, §12.3.1];

Cramer-Shoup. La variante de Elgamal décrite dans [BS23, §12.4] (de [CS01]);

Fujisaki-Okamoto si plaintexts uniformes. Tout PKE peut être compilé en un plus résistant, au sens où il a une privacy qui résiste à CCA *seulement pour des plaintexts aléatoires uniformes.* Cette notion de sécurité intermédiaire est appelée "key encapsulation mechanism" (KEM) ([BS23, exercice 12.5]), elle est due à [CS01]. Un tel PKE est donc utilisable par Alice avec en input une symmetric key, elle aurait généré uniformément. La méthode de compilation est décrite dans [HHK17, Fig. 12].

Futur standard: KyberKEM *si plaintexts uniformes.* Un exemple de KEM-CCA, obtenu avec la compilation précédente et donc uniquement utilisable pour des plaintexts égaux à des clés, est décrit dans [Bos+17, Algos. 6-8] sous le nom de "Kyber KEM", et plus récemment dans [Ava+21] sous le nom de "KyberCCA".

CCA hybrid encryption. De même que dans Section 5.2.3, mais instanciée à la fois avec un ind-CCA PKE et un ind-CCA symmetric encryption scheme, par exemple, un authenticated symmetric encryption scheme (cf Remark 33). Cette construction est due à [CS01, §7.3], où il est remarqué qu'elle fonctionne même si le PKE a seulement le niveau de sécurité intermédiaire appelé "KEM" (ci-dessus). Intuitivement c'est raisonnable puisqu'il n'est destiné qu'à être utilisé sur un plaintext égal à une symmetric key générée aléatoirement uniformément. Il est proposé dans [BS23, exercice 12.5] de prouver la sécurité de cette construction.

5.2.5 Comment passer à l'échelle dans Encrypt-then-Authenticate. Comment échouer facilement.

Exercice 41. On considère Alice qui souhaite implémenter un secure channel unidirectionnel depuis chacun de ses correspondants. Les correspondants sont reliés à Alice par des (public) authenticated channels unidirectionnels. Ils sont two-sided authenticated, et garantissent toutes les informations imaginables sur les identités des correspondants, au sens de la Definition 7. Alice souhaite n'avoir à générer qu'*une seule paire de clés* pour tous ses correspondants, contrairement à la méthode de Section 5.2.1. Elle leur propose donc le protocole suivant. Il utilise un public key encryption scheme (PKE). Lorsqu'Alice reçoit un ciphertext c sur un channel, qui lui garantit que l'expéditeur est une certaine machine Bob, elle calcule le plaintext $P \leftarrow \mathsf{Dec}(\mathsf{sk}_A, c)$. Puis elle output la paire (Bob,P). Dit autrement: elle output P sur son API $\mathsf{Receive}_{\mathrm{B}\to\mathrm{A}}$, qui est celle censée implémenter la réception des plaintexts de Bob.
(a) Supposons que le encryption scheme ait toutes les propriétés imaginables, par exemple être non-malléable (Exercice 39). Décrire un scénario où Alice a d'autres correspondants, et où Bob est corrompu, dans lequel ce protocole échoue à vérifier la Definition 2 d'un secure channel avec Bob. On pourra s'inspirer de l'attaque décrite dans Section 7.3.2, à partir de [BS23, §21.2.1 Variation 5].
(b) Alice ajoute la correction suivante au protocole: chaque expéditeur Bob doit désormais concaténer chaque plaintext P la chaîne de caractères "from Bob", puis générer un ciphertext $\widetilde{c}$ de la concaténation $\widetilde{P}$, puis procéder comme avant. Supposons cette fois que le encryption scheme soit malléable. C'est par exemple le cas d'un hybrid encryption avec AES-CTR, tel que décrit dans Section 5.2.3. Même question //La réponse est dans [BS23, §12.2.1] (voir aussi [BS23, §21.2.1 Variation 6]).

La solution correcte pour implémenter des secure channels avec Encrypt-then-Authenticate, avec plusieurs senders mais une seule paire de clés, consiste à combiner les questions (a) et (b) de l'Exercice 41. Concrètement, Alice doit utiliser un ind-CCA encryption scheme (hors-programme, voir Section 5.2.4), *et* ajouter l'identité de l'expéditeur aux plaintexts. C'est prouvé dans [BS23, Theorem 13.8], dans le cas particulier où les authenticated channels sont implémentés avec un digital signature scheme, comme décrit dans Section 5.3.1.

5.2.6 Exemple: "Kyber"

Un PKE appelé "Kyber" [Ava+21] a été accepté par le NIST en 2022 pour être un nouveau standard. Il s'agit essentiellement du PKE proposé par [LPR10, p4] //plus généralisé en Module-LWE, plus avec des clés et des ciphertexts compressés par arrondi ([Pei09, eq. 2.1]), plus compilé en (CCA) KEM par une complexification de [HHK17, Fig. 12], plus avec une génération d'aléa spécifique, minus particularisé à $p = 2$. Dans la Figure 8 on décrit le PKE de [LPR10, p5], avec les paramètres de plaintext de Kyber, sous le nom de "Mini Kyber".

Mini Kyber

Paramètres: q grand, supposé divisible par 4 pour simplifier. $n = 256$. $R_q := \frac{(\mathbb{Z}/q\mathbb{Z})[X]}{X^n + 1}$.

KeyGen($): - output $\mathsf{dk} := \mathsf{s} \xleftarrow{\$} R_q$ avec des *petits* coefficients;

- $\mathsf{a} \xleftarrow{\$} R_q$;
- $e \xleftarrow{\$} R_q$ avec des *petits* coefficients //le "key noise";
- output $\mathsf{ek} := (\mathsf{a}\ ,\ \mathsf{a}.\mathsf{s} + e)$.

Enc(ek = (a, t), m ; $): pour $m = \sum_{i=0}^{n-1} m_i.X^i \in R_q$ à coefficients $m_i \in \{0, 1\}$:

- $r, e_1, e_2 \xleftarrow{\$} R_q$ avec des *petits* coefficients //les "encryption noises";
- $u \leftarrow \mathsf{a}.r + e_1$;
- $v \leftarrow \mathsf{t}.r + e_2 + \frac{q}{2}.m$;
- output $c \leftarrow (u, v)$.

Dec(s , c′ = (u′, v′)): $\widetilde{M'} \leftarrow v' - \mathsf{s}.u'$ "le plaintext noisy et dilaté" //appelé "point de couleur" en cours;

- output $\mathsf{Compress}(\widetilde{M'})$ (Definition 22)//c'est à dire: soit $\widetilde{M'} = \sum_{i=0}^{n-1} = \widetilde{M'_i}.X^i$; alors pour chaque $i \in [0, \dots, n-1]$: set $m'_i \leftarrow 0$ si $\widetilde{M'_i} \in \left]-\frac{q}{4}, \frac{q}{4}\right[\bmod q = \left[0, \frac{q}{4}\right[\cup \left]\frac{3q}{4}, q\right[$ et $m'_i \leftarrow 1$ sinon, i.e., si $\widetilde{M'_i} \in [\frac{q}{4}, \frac{3q}{4}]$.

Figure 8: "Mini Kyber" est une particularisation de [LPR10, p5] avec des plaintexts dans $\{0, 1\}^n$. Un plaintext $m = (m_i)_{i \in [0,\dots,n-1]}$ est interprété comme le polynôme $m = \sum_{i=0}^{n-1} m_i X^i \in (\mathbb{Z}/q\mathbb{Z})[X]/(X^n + 1)$ à coefficients $m_i \in \{0, 1\}$. Ci-dessus on a fait directement cette interprétation. Dans Kyber, $q = 3329$. Dans Kyber, "*petit*" signifie, en particulier, être dans l'intervalle $[-\mu, \mu] \bmod q$, avec $\mu = 2$ ou 3 ([Ava+21, Algorithm 2]). Cependant, pour avoir le même niveau de privacy que dans Kyber, il faudrait augmenter les valeurs de q et μ dans Mini Kyber //car Kyber bénéficie d'un paramètre en plus, appelé "k".

L'intuition de la correctness est la suivante. Supposons ici, pour simplifier, que tous les "noises": s, e, e_1, e_2 soient nuls. Alors lorsqu'elle calcule KeyGen($), Alice output $\big(\mathsf{s}, (\mathsf{a}, \mathsf{t} = \mathsf{a}.\mathsf{s})\big)$. Et lorsqu'il calcule $\mathsf{Enc}((\mathsf{a}, \mathsf{t}), m, \$)$, Bob output $(\mathsf{a}.r\ ,\ \mathsf{t}.r + \frac{q}{2}.m)$. Le terme de droite est donc égal au plaintext dilaté $M = \frac{q}{2}.m$ masqué par la quantité secrète $\mathsf{t}.r$. C'est précisément cette quantité secrète qui est retirée par Alice en faisant Dec: $\mathsf{Dec}(\mathsf{s}\ ,\ (\mathsf{a}.r, \mathsf{t}.r + \frac{q}{2}.m)) = \mathsf{t}.r + \frac{q}{2}.m - \mathsf{s}(\mathsf{a}.r) = \frac{q}{2}.m$. L'idée à retenir est que le masque secret $\mathsf{t}.r = (\mathsf{a}.\mathsf{s}).r = \mathsf{s}.(\mathsf{a}.r)$ est à la fois connu de Alice et de Bob. Comme le remarquent [LPR10, p4], c'est donc la même situation que dans Diffie-Hellman (Section 5.1) ou Elgamal (Section 5.2.3), aux "noises" près.

Exercice 42 (cf [LPR10, p4], qui est un cas simple de [Bos+17, Thm. 1])**.** Prouver la correctness de Mini Kyber sous l'hypothèse que les paramètres sont choisis tels que, pour tout tirage de $\mathsf{s}, e, r, e_1, e_2$, alors

$$\text{tous les coefficients de } r.e - \mathsf{s}.e_1 + e_2 \text{ sont dans } \left]\overline{-\frac{q}{4}}, \overline{\frac{q}{4}}\right[\mod q \ . \tag{14}$$

Voici une reformulation de la correctness, au cas où cela pourrait aider. Pour tout polynôme $Q = \sum_{i=0}^{n-1} Q_i X^i \in R_q$, on note $||Q||_\infty := \max_i |Q_i|_{\mathbb{Z}/q\mathbb{Z}}$, c'est à dire le max des distances à $0 \mod q$ des coefficients (Definition 23). Alors, l'Equation (14) se reformule comme: $||r.e - \mathsf{s}.e_1 + e_2||_\infty < \frac{q}{4}$.

Exercice 43. Supposons que Bob génère deux ciphertexts, c et c', avec Mini Kyber, sous la même encryption key $\mathsf{ek} = (\mathsf{a}, \mathsf{t})$ de Alice. Bob a tiré les mêmes randomnesses $r = r'$ pour les générer (c'est mal). Vous êtes l'adversaire, vous ne connaissez pas les plaintexts m et m', décrire un algorithme qui output $m - m'$ à partir de c et c', sous l'hypothèse que tous les coefficients de $r.e - \mathsf{s}.e_1 + e_2$, et aussi ceux de $r'.e' - \mathsf{s}.e'_1 + e'_2$, sont dans $\left]\overline{-\frac{q}{8}}, \overline{\frac{q}{8}}\right[\mod q$.

5.3 (Digital) signature scheme

Un digital signature scheme consiste en trois algorithmes KeyGen, Sign et Verify qui vérifient les spécifications de la Figure 9. Ils s'utilisent de la façon suivante. Soit une machine sur laquelle l'adversaire $\mathcal{A}$ n'a aucune emprise, à part une connaissance parfaite de son état initial, qui exécute KeyGen($). Cela lui renvoie une paire de clés $(\mathsf{sk}, \mathsf{vk})$: une *(secret) signature key* et une *(public) verification key*. Désormais, elle est donc équipée de l'API de signature: $\mathsf{Sign}(\mathsf{sk}, \bullet)$. Désormais, cette API doit être *son seul lien avec le reste du monde* //ce qui ne l'empêche pas d'être incorporée, en boîte noire, à un système plus vaste etc.. On dit désormais que cette machine est le *owner* de vk. En input une donnée quelconque m, l'API renvoie une *signature* σ sur m. La garantie de *correctness* ci-dessous est que toute personne Quiterie qui connaîtrait vk, lorsque elle exécute $\mathsf{Verify}(\mathsf{vk}, m, \sigma)$, cela lui renvoie accept: c'est à dire que *σ est une signature sur m reconnue comme valide pour la clé* vk. En outre, si Quiterie observe qu'une signature σ' est reconnue comme valide sur une donnée m' pour la clé vk, alors par la *unforgeability* ci-dessous, cela lui garantit que m' a forcément été mise en input de l'API $\mathsf{Sign}(\mathsf{sk}, \bullet)$. Concrètement, quand bien même $\mathcal{A}$ réussirait à faire en sorte qu'un chèque m à son nom de 100$ soit mis en input de l'API $\mathsf{Sign}(\mathsf{sk}, \bullet)$ de W. Buffet, il n'arrivera pas à forger une signature valide pour la vk de W. Buffet sur un chèque m' à son nom de 1000$, si un tel chèque n'a jamais été mis en input de cette API.

5.3.1 Utilisation correcte pour implémenter un authenticated channel.

Alice et Bob génèrent chacun une paire de clés: $(\mathsf{sk}_P, \mathsf{vk}_\mathrm{P}) \leftarrow \mathsf{KeyGen}(\$)$ et $(\mathsf{sk}_Q, \mathsf{vk}_\mathrm{Q}) \leftarrow \mathsf{KeyGen}(\$)$. Désormais, Alice a l'identité qu'elle est *owner* de vk_P, de même Bob est owner de vk_Q. Le protocole suivant implémente un authenticated channel *entre les owners de* vk_P *et de* vk_Q. Par simplicité on appelle ces owners Alice et Bob. Mais en réalité, si par exemple l'adversaire a bénéficié d'un leakage de sk_Q, cela fait aussi de lui un owner de sk_Q.

- Pour $\mathsf{Send}_{\mathrm{A}\to\mathrm{Q}}(m)$ au owner de vk_Q, Alice génère la concaténation $\widetilde{m} = (\text{to } \mathsf{vk}_\mathrm{Q}, m)$ et envoie $\big(\widetilde{m}, \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_P, \widetilde{m})\big)$ sur le insecure channel;
- Pour $\mathsf{Receive}_{\mathrm{P}\to\mathrm{B}}$, à chaque fois que Bob reçoit une paire $\big(\widetilde{m'} = (\text{to } \mathsf{vk}_\mathrm{Q}, m'), \sigma'\big)$, il teste si $\mathsf{Verify}(\mathsf{vk}_\mathrm{P}, \widetilde{m'}, \sigma') = ?\mathsf{accept}$ et si c'est le cas il output m' ;
- De même pour Bob $\to$ Alice.

Digital signature scheme

API: $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\$)$: un algorithme randomisé qui renvoie en output sk une *(secret) signing key*, et vk une *(public) verification key*;
$\mathsf{Sign}(\mathsf{sk}, M) \rightarrow \sigma$: en input sk une signing key et $M \in \{0,1\}^*$ une donnée quelconque, output σ une *signature sur* M;
$\mathsf{Verify}(\mathsf{vk}, M, \sigma) \rightarrow \{\mathsf{accept\ or\ reject}\}$: en input vk une verification key, M une donnée quelconque et σ une signature, output accept ou reject. Si l'output est accept, on dit que σ est une signature reconnue comme *valide* sur M pour la clé vk.

Correctness: $\forall(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\$)$ et $\forall M$, alors $\mathsf{Verify}\Big(\mathsf{vk}, M, \mathsf{Sign}\big(\mathsf{sk}, M\big)\Big) = \mathsf{true}$ //cela signifie que si le owner de vk génère une signature σ sur M avec sa signature key, alors il a la garantie que σ sera reconnue comme une signature valide sur M pour vk.

Unforgeability: Soit une machine sans canal d'influence ou de leakage avec $\mathcal{A}$//hormis son état initial (Section 2.6) qui exécute $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\$)$. Désormais elle est le *owner* de vk, et elle autorise $\mathcal{A}$ à utiliser son API de signature: $\mathsf{Sign}(\mathsf{sk}, \bullet)$. Cela veut dire que $\mathcal{A}$ peut mettre en input des données "*requêtes*" $M'_1, M'_2, \ldots$ quelconques et récupérer les signatures $\sigma'_1, \sigma'_2, \ldots$ sur ces données. Le owner n'a aucun autre canal d'influence ou de leakage. Concrètement, la seule information que $\mathcal{A}$ a sur sk est ce que lui renvoie l'API. Alors, $\mathcal{A}$ est incapable de générer (M, σ) avec M une donnée qui n'aurait pas été mise en requête, et σ qui serait reconnue comme valide sur M pour vk.

Figure 9: Spécifications d'un digital signature scheme

[BCK98, Figure 1] On renvoie Remark 24 pour les méthodes garantissant que Bob n'output qu'une fois chaque donnée m envoyée par Alice.

Exercice 44. Considérons une variation dégradée du protocole, dans lequel le destinataire vk_A ne serait pas ajouté dans l'en tête des messages. Montrer que, pour peu que Alice ait plusieurs destinataires, alors cette variation n'implémente pas des authenticated channels.

5.3.2 Comment implémenter un secure channel avec Sign-then-(PK)Encrypt (et échouer de deux façons)

On considère le protocole Figure 10 entre Alice et plusieurs machines $S_1, \ldots, S_N$, appelées "senders". Les senders sont reliés à Alice par des (public) insecure channels. Le protocole utilise un digital signature scheme. Il est supposé que Alice a accès à un "annuaire des senders sans signature verification key en double", c'est à dire à un tableau $\{(S_i, \mathsf{vk}_\mathsf{i}) \ : i \in [N]\}$ tel que pour chaque i, *si* S_i est honnête alors il est le owner de vk_i, et tel que, dans tous les cas, *toutes les* vk_i *sont distinctes*. Le protocole utilise un public key encryption scheme (PKE) qui doit vérifier la *condition supplémentaire* appelée "ind-CCA" (hors programme, Section 5.2.4). En particulier, il doit être *non-malléable*, donc pas comme le mauvais exemple de hybrid encryption avec AES-CTR donné dans Section 5.2.3. Dans [BS23, Theorem 13.8] il est prouvé que, sous toutes ces hypothèses, le protocole Figure 10 implémente bien un secure channel.

Exercice 45. (a) [PKE malléable $\rightarrow$ identity-misbinding] Supposons que le encryption scheme soit malléable. C'est par exemple le cas d'un hybrid encryption avec AES-CTR, tel que décrit dans Section 5.2.3. Expliquer pourquoi ce protocole échoue à vérifier la Definition 2 de secure channels avec chacun des senders. Indice: décrire un scénario où un des senders, S_1, est *corrompu*, et l'adversaire fait une attaque résultant en un

Sign-then-encrypt (multi-senders)

Participants Senders $S_1, \ldots, S_N$ et Alice;

Setup Alice génère $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\$)$ et donne ek à tous les senders;

$\mathsf{Send}_{i\to A}(p)$ Générer une signature sur p: $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, p)$, et envoyer $c \leftarrow \mathsf{Enc}(\mathsf{ek}, (\sigma_i, p); \$)$ à Alice via le insecure channel;

$\mathsf{Receive}_{i\to A}$ Lorsque Alice reçoit un ciphertext c elle calcule le plaintext $\mathsf{Dec}(\mathsf{dk}_A, c)$. S'il est de la forme (σ_i, p), avec σ_i une signature sur p qui est valide pour la verification key vk_i telle que l'entrée $(S_i$,$\mathsf{vk}_i)$ est bien dans l'annuaire, alors elle output la paire $(S_i$,$p)$. Dit autrement: elle output p sur son API $\mathsf{Receive}_{i\to A}$, qui est celle censée implémenter la réception des plaintexts de S_i.

Figure 10: Implémentation simultanée de secure channel unidirectionnels vers Alice: méthode "Sign-then-(PK)Encrypt" avec: un ind-CCA PKE, et un annuaire des signature verification keys des senders tel qu'il n'y ait pas de vk_i en double.

identity-misbinding (Section 1.4) au profit de S_1 et au détriment d'un autre sender honnête S_2. Indice précis: on pourra s'inspirer de l'attaque décrite dans [BS23, §21.2.1 Variation 6].
(b) [Key hijacking $\to$ identity-misbinding] Supposons que le tableau *contienne la clé d'un sender honnête en double*. Expliquer pourquoi ce "protocole élargi" n'implémente pas des secure channels depuis ce sender honnête. Par exemple: S_2 est honnête et S_1 corrompu, avec dans le tableau: (S_1, vk_2) et (S_2, vk_2). Décrire un scénario de identity-misbinding (Section 1.4).
(c) L'attaque de (b) s'applique-t-elle à Encrypt-then-authenticate (Section 5.2.5) implémenté avec des digital signatures? Précisément: les authenticated channels depuis les senders sont simplement implémentés comme dans Section 5.3.1, c'est à dire, chaque envoyeur Q génère une signature sur son ciphertext et l'envoie à Alice en plus du ciphertext.

5.3.3 Exemple de RSA

Soient p et q deux nombres premiers. Soit $N = pq$. On note $\left(\mathbb{Z}/N\mathbb{Z}\right)^*$ l'ensemble des éléments de $\mathbb{Z}/N\mathbb{Z}$ qui ont un inverse pour la loi de multiplication $\underset{\text{mod } N}{\times}$. On appelle qu'il y en a $(p-1)(q-1)$.
Ainsi, par le théorème de Lagrange, pour tout $x \in \left(\mathbb{Z}/N\mathbb{Z}\right)^*$, on a l'égalité

$$x^{(p-1)(q-1)} = 1 \bmod N \; . \tag{15}$$

On note $\left(\frac{\mathbb{Z}}{(p-1)(q-1)\mathbb{Z}}\right)^*$ l'ensemble des éléments de $\frac{\mathbb{Z}}{(p-1)(q-1)\mathbb{Z}}$ qui ont un inverse pour la loi de multiplication $\underset{\text{mod } (p-1)(q-1)}{\times}$.

Définition 46 (RSA signature). $\mathsf{KeyGen}(\$)$:

- $p, q \overset{\$}{\leftarrow} [1, \ldots, 2^{2048}]$ deux très grands nombres premiers distincts au hasard, où le paramètre 2^{2048} dépend du niveau d'unforgeability souhaité; calculer $N := pq$;
- $\mathsf{v} \overset{\$}{\leftarrow} \left(\frac{\mathbb{Z}}{(p-1)(q-1)\mathbb{Z}}\right)^*$; //et certainement pas $\mathsf{v} = 3$ ([Fin08], [BS23, §13.6.1] voir aussi [Bit19]).
- Output $\mathsf{vk} := \{N, \mathsf{v}\}$.

- Calculer $(p-1)(q-1)$; //aussi parfois appelé "indicatrice d'Euler de N", et noté "$\phi(N)$". Ne montrer $(p-1)(q-1)$ à personne. En dépit de la notation $\phi(N)$, cette quantité n'est calculable en pratique par aucun adversaire $\mathcal{A}$ qui ne connait que le chiffre public N, dès lors que les facteurs secrets p et q sont très grands.
- calculer $\mathsf{s} \in \left(\frac{\mathbb{Z}}{(p-1)(q-1)\mathbb{Z}}\right)^*$ égal à l'inverse de v pour la multiplication modulo $(p-1)(q-1)$. //s se calcule par exemple avec l'algorithme d'Euclide.. Cela signifie par définition que:

$$\overline{\mathsf{s}.\mathsf{v}} = 1 \mod (p-1)(q-1) \; ; \tag{16}$$

  //La notation $\overline{\mathsf{s}.\mathsf{v}}$, utilisée en cours de rappels de mathématiques, signifie le *reste de la division euclidienne* modulo m, où m est suivant le contexte, ici $m = (p-1)(q-1)$. C'est par définition l'unique nombre dans $[0, \ldots, (p-1)(q-1)-1]$ congru à $\mathsf{s}.\mathsf{v}$ modulo $(p-1)(q-1)$. $\overline{\mathsf{s}.\mathsf{v}}$ n'est donc pas égal, en général, à $\mathsf{s}.\mathsf{v}$. //[Une fois que s est calculé, effacer les données secrètes intermédiaires $(p-1)(q-1)$, p, q. Éventuellement, p, q peuvent être conservés pour accélérer les multiplications $\mod N$.];
- Output $\mathsf{sk} := \{N, \mathsf{s}\}$. //La sécurité de RSA est basée sur le fait qu'aucun adversaire ne sait calculer $\phi(N) = (p-1)(q-1)$ à partir de N. Car s'il savait faire, il pourrait ensuite facilement calculer l'inverse de e modulo $(p-1)(q-1)$, égal à la signing key.

$\mathsf{Sign}(\mathsf{sk} = (N, \mathsf{s}), m)$: output $H(m)^{\mathsf{s}} \pmod N$ //H une fonction de hachage.

$\mathsf{Verify}(\mathsf{vk} = (N, \mathsf{v}), m, \sigma)$: si $\sigma^{\mathsf{v}} \mod N = H(m)$, output accept, sinon output reject.

Preuve de la correctness: $\sigma^{\mathsf{v}} = (H(m)^{\mathsf{s}})^{\mathsf{v}} = H(m)^{\mathsf{s}\,\mathsf{v}}$. Or $\mathsf{s}\,\mathsf{v} \equiv 1 \mod (p-1)(q-1)$ donc il existe $\lambda \in \mathbb{Z}$ tel que $\mathsf{s}\,\mathsf{v} = 1 + \lambda(p-1)(q-1)$. Donc $H(m)^{\mathsf{s}\,\mathsf{v}} = H(m)^1 \, (m^{(p-1)(q-1)})^{\lambda}$. Or $H(m)^{(p-1)(q-1)} \equiv 1 \in (\mathbb{Z}/N\mathbb{Z})^*$ par le théorème de Lagrange.

Exercice 47. Soit le MAUVAIS signature scheme RSA où on a enlevé les hashs: $\mathsf{MSign}(\mathsf{sk} = (N, \mathsf{s}), m) := m^{\mathsf{s}}$; et $\mathrm{MVerify}(\mathsf{vk}, m, \sigma)$: output accept si $\sigma^{\mathsf{v}} = m$ et reject sinon. Soient les signatures valides dans $\mathbb{Z}/35\mathbb{Z}$ pour la clé de vérification $\mathsf{vk} = (N = 35, \mathsf{v} = 17)$: $\sigma_1 = 32$ sur $m_1 = 2$; $\sigma_2 = 33$ sur $m_2 = 3$.
a) Calculer une signature valide sur $m = 6$ pour la même vk.
//I] exprimer ce que l'on cherche: σ tel que $\sigma^{\mathsf{v}} = m$.
II] exprimer ce que l'on connait: (indice 1) $\overline{32^{\mathsf{v}}} = m_1$ et (indice 1) $\overline{33^{\mathsf{v}}} = m_2$.
III] exprimer m en fonction de ce qu'on connait: $m = m_1.m_2$.
IV] combiner les équations que l'on connait //concrètement cela veut dire appliquer la formule que si on a deux égalités $a = b$ et $c = d$, alors $a.c = b.d$: pour faire apparaître ce que l'on cherche: $m = m_1.m_2 = \overline{32^{\mathsf{v}}.33^{\mathsf{v}}} = \overline{(-3).(-2)}^{\mathsf{v}} = 5^{\mathsf{v}}$ Donc $\sigma = 5$ est la signature cherchée.
b) Calculer une signature valide sur $m' = 8$ pour la même vk.
//I] Ce que l'on cherche: σ' tel que $\sigma'^{\mathsf{v}} = m'$. II] déjà fait. III] exprimer m en fonction de ce que l'on connait: $m = m_1^3$. IV] combiner les équations que l'on connait pour faire apparaître ce que l'on cherche: $m = (m_1)^3 = \overline{32^{\mathsf{v}}}^3 = \overline{(-3)^3}^{\mathsf{v}} = \overline{-27}^{\mathsf{v}} = 8^{\mathsf{v}}$ Donc $\sigma' = 8$ est la signature cherchée.

6 Attention au vocabulaire utilisé en pratique

- Les mots intégrité, *integrity* ([Lin06, p. 4.1.1] [BS23, §9.2]), *authenticated* et *authenticity* ([Lin06, p. 4.1.1] [IOM12]) peuvent tous prendre deux sens très différents. Dans le premier sens ([Lin06, p. 4.1.1], [BS23, §9.2], [IOM12]), on dit qu'une donnée m est intègre ou authenticated, par définition

si elle a été authenticated par une ou des machines désignées, par exemple Bob. Précisément, cela veut dire que m a été mis en input d'une API d'authentication de Bob (selon les cas: MAC.Sign ou AuthSymEnc ou Sign). Sous certaines conditions, formalisées par divers security games selon les cas, l'output de ces API en fournit la preuve (selon les cas: un tag ou un authenticated ciphertext ou une signature). À noter que dans le cas symétrique (MAC.Verify ou AuthSymDec), cette preuve n'est vérifiable que par les machines qui connaissent la clé secrète utilisée par Bob pour authenticate.

Dans le deuxième sens (Section 3.1), on dit qu'une donnée m est intègre si elle est égale à une donnée désignée comme point de référence. Par exemple le point de référence peut être une version antérieure, un original, ou encore une donnée dont on ne connait avec certitude que le hash.

- to encrypt se traduit par *chiffrer*, encryption par *chiffrement*;
- un ciphertext se traduit par *un chiffré*;
- un plaintext se traduit par *un clair*. Il est souvent aussi traduit par *message*, ou *message secret*. (Mnémotechnique: le "secret" est ce qu'on n'a pas envie de montrer en "plain" / "clair" à l'Adversaire)
- Attention: ce même mot de *message* désigne aussi la donnée publique envoyée sur le réseau de communication.
- *signer un message* x: est un double abus de langage. Il signifie: générer une signature σ sur la chaîne de caractères x. Il signifie parfois aussi créer la concaténation $x||\sigma$ et l'envoyer par message public. Suivant le contexte, x peut être un plaintext, un ciphertext, une public encryption key etc.
- (public) encryption key ek: on la trouve souvent sous le nom de *public key*, notée pk, qui se traduit par *clé publique*. Attention: *public key* désigne aussi souvent une *(public) signature verification key* vk.
- secret decryption key: on la trouve souvent sous le nom de *secret key* et notée sk, qui se traduit par *clé secrète*. Attention: *secret key* désigne parfois aussi une *secret signature key*, qu'on note sk dans ce cours.

7 Key exchanges over insecure channels: TLS 1.3 & PKE-based

On rappelle l'objectif fixé dans Section 1.7, qui est d'implémenter un secure channel entre deux machines Alice et Bob, *avec identification mutuelle*.
En réalité, un client Alice typique ne souhaite pas se connecter à une machine précise Bob. Elle cherche simplement à se connecter à une machine dont elle a la garantie qu'elle possède une identité, qu'on note id_{Bob}, et qui est définie par "appartenir au propriétaire du nom de domaine bob.com" Mais par simplicité, dans la suite, on fera comme si Alice souhaitait se connecter précisément à la machine Bob. On considérera donc l'identité très précise, notée aussi Bob, définie comme "être la machine Bob".
Également, dans ce scénario, Bob n'est pas au courant que Alice souhaite se connecter à lui, ni ne connaît son identité, avant d'avoir au reçu au moins un message d'Alice. On commence par présenter une solution simple dans Section 7.1.1, qui suppose que Alice *connaisse a priori la signature verification key de Bob*. Puis on introduit le modèle des certificate authorities (Section 7.1.2), et comment l'utiliser pour résoudre ce problème.
Puis, dans Section 7.2, on donnera la technique générale de TLS 1.3 permettant à Alice d'implémenter un secure channel avec identification de Bob *sans aucune information a priori sur son identité*, et le tout en un

seul aller-retour; ainsi qu'une méthode pour Bob et Alice de ne révéler leurs identités que l'un à l'autre. On l'adaptera, dans Section 7.3, dans le contexte des futurs standards de *encryption-based* key exchange.

7.1 Solution basique: signature-based Diffie-Hellman, et ses limitations

On décrit d'abord une solution simple, qu'on appelle "Signature-based-Diffie-Hellman", qui s'obtient en compilant des outils déjà vus. Dans Section 4.3.1 on a vu que deux ressources suffisent pour implémenter un secure channel avec identification (one- or two-sided) entre Alice et Bob: un key exchange avec identification (Definition 8), et un insecure channel. Puis, pour implémenter le key exchange requis, le *authenticated Diffie-Hellman* de Section 5.1 Figure 6 nécessite comme préalable un authenticated channel avec identification. Cette implémentation impose donc que les peers (Alice et Bob) soient pré-spécifiés, c'est à dire comme les deux correspondants du authenticated channel. Comment implémenter le authenticated channel requis, sans KeyExch, donc sans cryptographie symétrique? On ne connait que la méthode de Section 5.3.1. Elle suppose que Alice *connaisse* à l'avance une verification key $\mathsf{vk_B}$ de Bob. Concrètement, Alice ajoute "to $\mathsf{vk_B}$" (ou "to Bob") à chacun de ses messages, et n'accepte de message de Bob que s'il est accompagné d'une signature valide pour $\mathsf{vk_B}$.

7.1.1 Plain signature-based Diffie-Hellman

Le protocole suivant implémente un Key exchange avec authentification two-sided entre peers pré-spécifiés Alice et Bob qui *connaissent d'avance* les vérification keys $\mathsf{vk_A}$, $\mathsf{vk_B}$ l'un de l'autre.

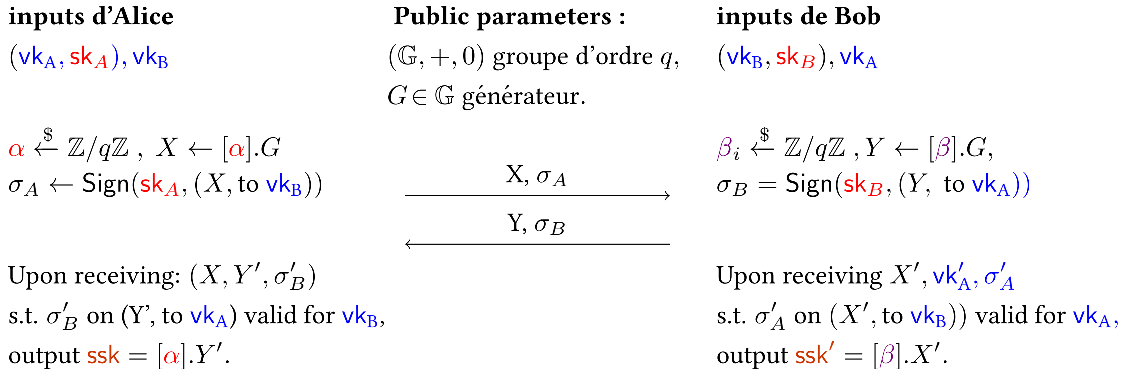

Figure 11: Plain signature-based Diffie Hellman over insecure channel. Il est obtenu en instanciant Section 5.1 Figure 6 avec le authenticated channel implémenté par des signatures, suivant la méthode de Section 5.3.1. Il a donc les mêmes garanties que celui de la Figure 6, i.e., satisfait la Definition 8. Il s'agit de celui de [CK01, Figure 4] avec les simplifications suivantes. (i) Bob n'attend pas de confirmation d'Alice avant d'output (pas de "key confirmation" au sens de [Kra05; SFW20]). (ii) pas de session ID explicite. Il peut par exemple être constitué de (X, Y) (voir Remark 14); (iii) méthode d'authentication par signature sans interaction (contrairement à [CK01, Figure 4], héritée de [BCK98]). En effet, suivant la Remark 14, on a considéré que (X, Y) joue le rôle de session ID, ce qui est suffisant puisque Alice et Bob n'envoient qu'un seul message par session.

Ce protocole a la contrainte qu'il ne doit être utilisé par Alice que si elle connaît avec certitude la $\mathsf{vk_B}$ de Bob, sinon elle risque de se retrouver à réaliser un key exchange avec un man-in-the-middle en le prenant pour Bob.

7.1.2 Peut-on garantir l'identité de l'owner d'une verification key? Modèle des CA.

Une *certificate authority* (CA) peut être vue, de façon idéalisée, comme une entité qui apporte à Alice la garantie qu'une certaine signature verification key est bien owned par Bob. CA est, elle-même, owner d'une public signature key vk_{CA} qu'on supposera connue d'Alice. Par exemple, la liste des vk_{CA} connues d'un navigateur Firefox sont visibles en cliquant sur "afficher les certificats", dans les paramètres. On fera l'hypothèse que toutes les affirmations signées par vk_{CA} sont vraies. On dit qu'un *certificat* d'une identité "Bob" pour la clé vk_{Q}, est la donnée d'une signature σ_{CA}, valide pour vk_{CA}, sur la chaîne de caractères "Bob est owner de vk_{Q}". En pratique, il n'est pas évident d'implémenter, à des prix attractifs, une CA qui dirait toujours la vérité. C'est pourquoi il n'est pas exclu qu'une CA fasse mal son travail. On peut diviser en trois catégories les certificats qui ne disent pas la vérité. On les classe de la faute la plus grave à la moins grave.

Rogue certificate $\rightarrow$ identity theft C'est un certificat qui dit qu'une machine *honnête* Bob est owner d'une clé vk_{Q}, alors que Bob n'est pas le owner de vk_{Q}. La conséquence est que, si vk_{Q}est en réalité owned par une machine Q corrompue par l'adversaire $\mathcal{A}$, alors ce dernier pourra créer des signatures valides pour vk_{Q}sur toute donnée arbitraire. Or, le certificat attribue toutes ces signatures à Bob. C'est donc une *usurpation de l'identité de l'honnête Bob*, cela s'appelle couramment *identity theft*. De vieux exemples de rogue certificates sont donnés dans [BS23, p. 13.8.1].

Key hijacking $\rightarrow$ identity-misbinding C'est un certificat qui dit qu'une machine *corrompue* Q est owner d'une certaine clé vk_{Q}, alors que vk_{Q}est en réalité owned par une machine honnête Bob. La conséquence est qu'une signature de Bob sera attribuée à Q par le certificat. Cela ne permet heureusement pas à Q d'usurper l'identité de Bob, puisque Q ne connait pas la secret signature key sk_{Q} de Bob. L'exercice Exercice 45 (b) est un exemple très simple qui montre comment un key hijacking peut être exploité pour faire un identity-misbinding. Un exemples plus élaboré, avec d'autres mauvaises circonstances aidantes, est donné dans [BS23, §21.2.1 variation 3]. Leurs notations sont différentes ($Q \rightarrow R, Bob \rightarrow P$ et Alice$\rightarrow Q$), on garde ici les nôtres. Il aboutit à la conséquence qu'un certain $\mathsf{KeyExch}$ entre Alice et Bob échoue à garantir ses propriétés. Précisément, à l'issue, Alice pense que le coowner de ssk est Q alors qu'il est en réalité Bob: cela ne respecte donc pas la définition/spécification d'un key exchange (Definition 8). Dans une telle situation, les spécifications de authenticated encryption ne garantissent plus rien. Concrètement, la conséquence est que les auth-ciphertexts créés par Bob, lorsqu'Alice les reçoit, alors elle les crédite à Q. Or, par exemple si le plaintext p d'un certain auth-ciphertext c est *inconnu* de Q, alors il est incapable d'avoir mis p en input de $\mathsf{AuthSymEnc}$. Donc dans cet exemple, si la spécification de *ciphertext integrity* de symmetric authenticated encryption (Figure 4) avait été respectée, jamais Alice n'aurait crédité p à Q. Dit auterment, c'est un exemple de *identity-misbinding* (Section 1.4), donc la spécification de secure channel entre Alice et Bob est mise en défaut.

Rogue key C'est un certificat qui dit qu'une machine *corrompue* Q est owner d'une certaine clé vk_{Q}, alors qu'elle ne l'est pas, et que *aucune machine honnête* n'est owner de cette clé. Une attaque décrite dans [Boneh-Drijvers-Neven, AC'18], exploite que Q arrive à obtenir un certificat affirmant faussement qu'il est owner d'une vk_{Q}. C'est en réalité en "rogue key", au sens où elle a été mal formée par Q, au point qu'il ne connait même pas de secret signature key correspondante. Concrètement, Q a créé vk_{Q}, de façon déterministe, uniquement à partir de clés publiques d'un ensemble E de machines honnêtes, donc sans connaître aucune secret signature key sk_{Q}. Il n'a donc pas du tout appliqué la fonction $\mathsf{KeyGen}(\$)$. L'exploitation est que, quand cette bitstring est mise en input d'un algorithme

dit de "multisignature" sur une donnée quelconque m, il en ressort un objet (une "multi-signature") qui a la même validité que si tous les membres de l'ensemble $E \cup Q$ avaient unanimement signé m, alors même qu'aucun membre de E n'a signé m. Il existe plusieurs contre-mesures possibles à cette attaque, qui tolèrent des certificats sur des rogue keys.

Une CA qui crée ne vérifie pas que Q est bien owner de vk_Q, et donc émet potentiellement des certificats pour Q sur un rogue/hijacked key vk_Q, pourrait être appelée *paresseuse.* C'est typiquement le cas d'une CA implémentée par une blockchain sur laquelle toute personne pourrait publier n'importe quelle clé en son nom. Une telle CA est couramment modélisée par la ressource idéalisée suivante, appelée aussi *bulletin board* ([CSV16, Fig. 3] [BS23, §21.12]).

7.1.3 (ua,uk)-DH: Diffie-Hellman with unknown activator and unknown responder's key

En supposant le modèle CA, on cherche maintenant à enlever la contrainte observée à la fin de Section 7.1.1. Il s'agit du fait que Alice et Bob doivent connaître leurs signature verification key respectives pour pouvoir exécuter Figure 11. Une façon naïve serait qu'Alice et Bob se demandent mutuellement des certificats sur leurs signature verification keys avant de commencer, mais ça prendrait donc un aller-retour préliminaire. On va faire mieux. Dans la Figure 12 on décrit un key-exchange avec peers préspécifiés, qui ne nécessite d'Alice "que" la connaissance a priori de l'identité de Bob écrite sur l'un des certificats de Bob. En outre, le protocole a une flexibilité supplémentaire par rapport au plain signature-based Diffie Hellman de Figure 11. C'est le fait qu'il est exécutable même si Bob ne sait pas a priori que Alice veut réaliser un key exchange avec lui.
Pour permettre ces deux améliorations au plain signature-based Diffie Hellman de Figure 11, sans augmenter le nombre d'allers-retours, on compile ce dernier de la façon suivante, décrite dans la Figure 12. Alice, dans son premier message, remplace "to vk_B" par "to Bob", car elle ne connaît pas encore vk_B. Lorsque Bob reçoit un premier message de ce type, signé par une certaine vk_A quelle qu'elle soit, il commence une session de Figure 11 avec vk_A. En quelque sorte, cette instruction de commencer une session est "meta", elle ne fait pas partie du key-exchange proprement dit. Cependant comme cette situation est typique, il est de coutûme de considérer qu'elle fait partie du key-exchange: on dit alors que la owner de vk_A (Alice) est l'*initiator* et Bob le *responder.* À partir de là, la différence avec Figure 11 est que Bob annexe à son message un certificat Cert(Bob, vk_B) pour sa clé vk_B, afin que sa correspondante l'apprenne. De même, Alice n'accepte le message de Bob *que* si la signature est valide pour la vk_B contenue dans le certificat *et* si le certificat est au *même* nom, Bob, que celui qu'elle a envoyé dans son premier message.

Remark 48 (Variation: one-sided identification only)**.** On considère la variante de Figure 12 où Alice n'a pas besoin d'utiliser une signature key pair $(\mathsf{vk}_\mathrm{A}, \mathsf{sk}_A)$ ni de signer son message. Alice est néanmoins implicitement caractérisée comme étant la seule machine connaissant le α secret tel que $[\alpha].G = X$. Elle est donc, en quelque sorte, la owner de X. La session ouverte par Bob est alors (X,Bob). Cette variante apporte donc de l'identification one-sided de Bob envers Alice, au sens de la Definition 10. Par contre il n'apporte pas d'identification de Alice à Bob (mis à part, implicitement, être la owner de X, puis de la ssk output).

7.1.4 Limitations, et solutions sous-optimales

Comme remarqué dans la dernière phrase de Section 7.1.3, le signature-based Diffie-Hellman amélioré de Figure 12 souffre quand même des contraintes suivantes. D'abord, Bob ne répond au message d'Alice *que* s'il possède un certificat au nom l'identité, Bob, contenue dans le message d'Alice: Cert(Bob, vk_B).

inputs d'Alice	**Public parameters :**	**inputs de Bob**
$(\mathsf{vk}_{\mathrm{A}}, \mathsf{sk}_{A})$, "Bob"	$(\mathbb{G}, +, 0)$ groupe d'ordre q, $G \in \mathbb{G}$ générateur.	$(\mathsf{vk}_{\mathrm{B}}, \mathsf{sk}_{B})$
$\alpha \xleftarrow{\$} \mathbb{Z}/q\mathbb{Z}$, $X \leftarrow [\alpha].G$ $\sigma_A \leftarrow \mathsf{Sign}(\mathsf{sk}_A, (X, \text{"to Bob"}))$	X, vk_{A}, σ_A $\longrightarrow$	Upon receiving $X', \mathsf{vk}'_{\mathrm{A}}, \sigma'_A$ with σ'_A on $(X', \text{"to Bob"})$ valid for $\mathsf{vk}'_{\mathrm{A}}$ **open session ($\mathsf{vk}'_{\mathrm{A}}$,Bob)**
Upon receiving: (X, Y', σ'_B) with σ'_B on (X,Y', to vk_{A}) valid for $\mathsf{vk}'_{\mathrm{B}}$ output $\mathsf{ssk} = [\alpha].Y'$	$\longleftarrow$ Y, σ_B, Cert(Bob,$\mathsf{vk}'_{\mathrm{B}}$)	$\beta \xleftarrow{\$} \mathbb{Z}/q\mathbb{Z}$, $Y \leftarrow [\beta].G, \sigma_B = \mathsf{Sign}(\mathsf{sk}_B, (X', Y, \text{ to } \mathsf{vk}'_{\mathrm{A}}))$ output $\mathsf{ssk}' = [\beta].X'$

Figure 12: Signature-based Diffie Hellman over insecure channel, avec *initiator* Alice. Raffinement par rapport à Figure 11 et [CK01, Figure 4]: on n'a pas supposé que Bob connaît d'avance la vk_{A} d'Alice, ni même son identité. Donc, on a spécifié que Bob initie une session dès qu'il reçoit un message de demande de session vis à vis de toute signature verification key, quelle qu'elle soit, par exemple $\mathsf{vk}'_{\mathrm{A}}$ (pas forcément vk_{A}). Donc par exemple si le message reçu par Bob est celui d'Alice, il va ouvrir une session (vk_{A},Bob). Formellement, la session d'Alice sera aussi (vk_{A},Bob), puisqu'elle l'exécute en tant que owner de vk_{A}, mais on la notera parfois (Alice,Bob) par simplicité. *Si* Alice outputs, alors c'est que Bob a initié une session (vk_{A},Bob) *et* a donc reçu son X (pas un autre), donc les outputs sont égaux pour la même raison que dans Section 5.1. Dans tous les cas on a les mêmes garanties que dans Section 5.1. En détail: si Alice et honnête et outputs ssk, alors, étant donné que c'est une session (Alice,Bob), (i) la seule façon qu'une autre machine honnête puisse output ssk est que cette machine soit Bob (et donc que Bob soit honnête); (ii) si c'est le cas, Alice et Bob bénéficient alors de la Key-indistinguishability pour ssk. On rappelle également que les garanties de la Definition 8 s'appliquent même si Bob est honnête et se retrouve à initier une session relativement à une autre $\mathsf{vk}'_{\mathrm{A}}$. Par exemple, toujours en supposant Alice honnête, la Definition 8 garantit que même si l'adversaire apprend la ssk' output par Bob dans une session (Bob,$\mathsf{vk}'_{\mathrm{A}}$), il n'apprendra jamais la ssk output par Bob dans la session (Bob,vk_{A}) (la même que celle output par Alice dans sa session (vk_{A},Bob)). Pour s'en convaincre, supposons par exemple que l'owner de $\mathsf{vk}'_{\mathrm{A}}$ est corrompue et a réutilisé le X envoyé par Alice vis à vis de Bob. Alors, sous les hypothèses cryptographiques de la Proposition 36, ni l'adversaire (ni cette owner) seront incapables de deviner α, donc de deviner la ssk (Alice,Bob). Les autres simplifications par rapport à [CK01, Figure 4] sont les mêmes que dans Figure 11.

Puis, lorsqu'Alice reçoit une réponse avec une signature valide pour une clé vk_{B} liée à une identité dans un certificat: Cert(Bob,vk_{B}), Alice n'output *que* si l'identité sur le certificat est bien *égale* à celle qu'elle a envoyé dans son premier message, e.g., Bob. Donc si Alice se trompe sur l'identité de Bob dans son premier message, le protocole ne produit aucun output (mais au moins il n'enfreint pas les spécifications de la Definition 8). Ces règles ne peuvent pas être relâchées d'une manière ou d'une autre, sinon le protocole s'expose aux attaques suivantes.

Accept certificate for another Bob' Supposons que Alice a ouvert une session avec un paramètre une certaine identité de peer: Bob (donc son premier message contient "to Bob"). Supposons maintenant qu'elle accepte quand même une réponse contenant une signature valide pour une certaine $\mathsf{vk}'_{\mathrm{B}}$, liée par un certificat à une certaine identité, Bob', qui ne serait *pas* celle d'une machine honnête d'identité Bob. Concrètement, Alice ne connait pas assez précisément l'identité de Bob pour distinguer que Bob' n'est pas Bob. Alors la spécification de la Definition 8 est mise en défaut dans les sous-scénarios suivants (ni exhaustifs, ni mutuellement exclusifs).

(1) Si Bob' est une machine honnête $\rightarrow$ identity-misbinding , par exemple qui a répondu trop diligemment à Alice, i.e., a ouvert une session (Bob',Alice) en dépit du fait que le message d'Alice était pour Bob. Le résultat est que Bob' va output une clé, ssk, dans une session (Bob',Alice), alors que Alice va output la même clé ssk relativement à la session (Alice,Bob). Donc cela ne respecte pas le sous-propriété de *implicit authentication* de la Remark 9. La conséquence est que Alice va attribuer les messages de Bob' à Bob (identity-misbinding), sans même que Bob' ni Bob ne l'aient voulu.

(2) Si Bob' est une machine corrompue

(a) Si Bob' a généré $Y' = [\beta'].G$ connaissant β' alors l'adversaire $\mathcal{A}$ apprend donc la $\mathsf{ssk}' = [\alpha].[\beta'].G$ output par Alice relativement à la session (Alice,Bob) qu'elle a ouverte entre elle et la machine honnête Bob. Donc ça ne respecte pas la spécification Key indistinguishability de la Definition 8. Concrètement, une telle situation où $\mathcal{A}$ apprend la clé d'une session entre machines honnêtes (Alice,Bob) s'appelle *key exposure*, quand bien même Bob n'est pas au courant de cette ssk. L'exploitation dans ce cas précis est que Alice va attribuer à Bob des plaintexts arbitrairement choisis par Bob'. Bob' réalise donc une usurpation de l'identité de Bob. Cela s'appelle aussi *identity theft*, ou encore *impersonation*, de Bob.

(b) Si Bob a envoyé un Y différent du Y' reçu par Alice de Bob' (puis output) alors, quand bien même Bob' ne connaîtrait pas Y', il résulte que les ssk et ssk' output par Bob et Alice dans leurs sessions respectives (Alice,Bob) ne sont pas les mêmes, donc cela ne respecte pas la *consistency*.

Change one's mind on the peer si maintenant Alice décide au contraire de changer à la volée l'intitulé de la session: (Alice,Bob) vers (Alice,Bob'), l'exercice Exercice 49 montre qu'elle s'expose à des attaques d'identity-misbinding.

Le protocole Figure 12 n'est donc pas applicable si Alice ne connait pas à l'avance l'identité de Bob, par exemple si Bob est un objet volant. Une solution serait qu'Alice demande à Bob de lui donner son identité, mais cela ajouterait un aller-retour. On souhaiterait donc tout faire en un seul aller-retour, c'est à dire un key-exchange avec identification entre peers qui ne connaissent pas a priori, au sens de la Definition 12. Le problème n'est pas simple, puisque l'attaque de l'Exercice 49 montre qu'une adaptation naïve de Figure 12 échoue à remplir cet objectif.

Exercice 49. On considère la insecure variation suivante du protocole Figure 12, décrite dans la Figure 13. Alice n'a plus en input l'identité de Bob. Elle n'inclut plus "to Bob" dans son message. De même, Bob ouvre une session sans vérifier si la signature reçue s'applique à un message contenant "to Bob". Enfin, de même Alice accepte le message reçu dès que le X est celui qu'elle a envoyé et que la signature est valide pour un certificat reçu, par exemple Cert(Bob',$\mathsf{vk}'_{\mathrm{B}}$), pour toute identité Bob' que ce soit. Elle output alors (Bob',$\mathsf{ssk} = [\alpha].Y'$), où Bob' est l'identité marquée sur le certificat. Décrire une attaque conduisant à ce que Alice outputs (Bob', ssk) et Bob outputs $(\mathsf{vk}_{\mathrm{A}}, \mathsf{ssk})$, donc avec la vk_{A} d'Alice et la même ssk. Expliquer en quoi ce protocole ne respecte donc pas la Definition 12 (indice: Remark 13).

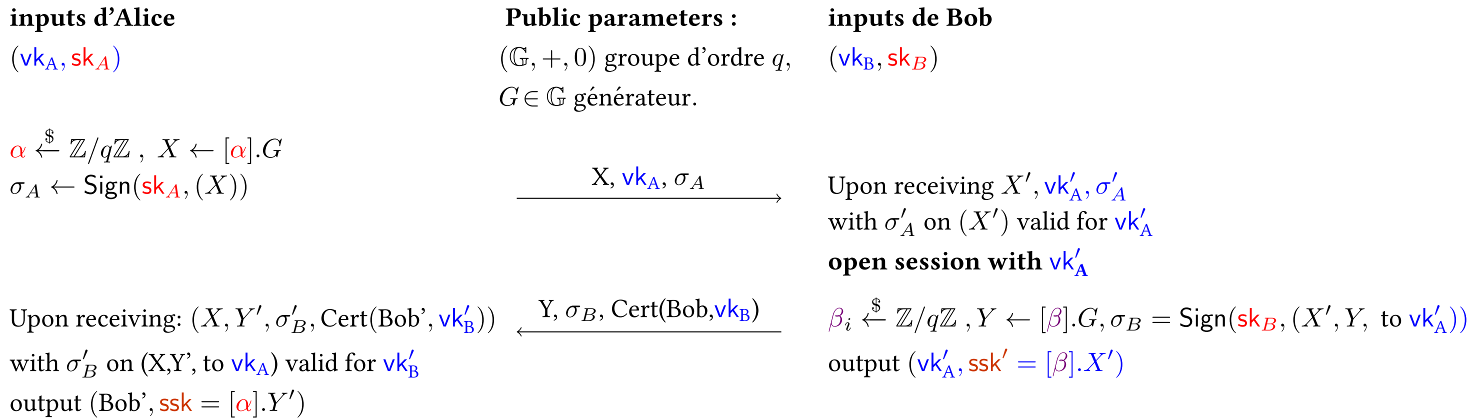

Figure 13: Insecure adaptation of signature-based Diffie Hellman (Figure 12), when misused for *post*-specified peers. (One-sided version of "BADH" from [Kra03, §3.1], credited to [DOW92]).

Enfin, même on a la limitation supplémentaire que dans tous les protocoles considérés jusqu'à maintenant, le contenu des messages envoyés sur le insecure channel révèle à $\mathcal{A}$ que Alice et Bob ont implémenté un secure channel entre eux. Par exemple, même dans le insecure Figure 13: le premier message contient une signature valide pour la verification key d'Alice vk_{A}. Donc *il en apprend autant à $\mathcal{A}$ qu'à Bob sur l'identité d'Alice* (c'est à dire: rien si Alice a généré vk_{A} pour l'occasion, ou, s'il s'agit au contraire d'une clé de long terme publiquement associée à Alice, lui apprend que c'est Alice). Ensuite, le message envoyé par Bob contient une signature valide pour la clé vk_{B} contenue dans le certificat pour Bob, donc apprend que le signeur est Bob, et en outre apprend qu'il est destiné à l'owner de la vk_{A}.
D'un autre côté, si Bob utilisait lui aussi une vk_{B} créée pour l'occasion pour cacher son identité, il resterait le problème pour Alice et Bob de se prouver leur véritable identité l'un à l'autre à travers le secure channel qu'ils viennent de créer. L'Exercice 50 montre que c'est possible en un aller retour. L'exercice est applicable à la situation où: Alice a demandé à Bob une identité, Bob a utilisé une key pair $(\mathsf{sk}_Q, \mathsf{vk}_{\mathrm{Q}}) \xleftarrow{\$} \mathsf{KeyGen}(\$)$ pour l'occasion, puis Alice et Bob ont exécuté Figure 12 relativement à l'identité vk_{Q}(et n'importe quelle identité vk_{A} pour Alice, on ne s'en préoccupe pas). L'exercice suggère donc un protocole permettant (i) à l'owner de vk_{Q}de convaincre Alice qu'il est l'owner de la vk_{B} de Bob, s'il est vraiment Bob, et (ii) garantissant que Alice ne le croira pas s'il n'est en fait pas Bob.

Exercice 50. Considérons deux (public) signature verification keys vk_{Q} et vk_{B} et la donnée des deux signatures suivantes: σ_Q une signature valide pour vk_{Q} sur: $m_Q =$"je suis la machine owner de vk_{B} "; et σ_B une signature valide pour vk_{B} sur: $m_B =$"je suis la machine owner de vk ".

(a) Que peut-on en déduire si l'owner Bob de $\mathsf{vk_B}$ est unique, et que c'est une machine:

- conforme aux spécifications de Section 2.6 et de Section 5.3. C'est à dire: sans side-channel, autre que la possible utilisation par $\mathcal{A}$ de son(ses) API(s) de signature;
- et telle que son API de signature n'accepte en input que des phrases vraies.

(b) Supposons que l'owner (ou l'un des owners) Q de $\mathsf{vk_Q}$qui soit corrompu, au sens où il ne vérifierait pas les conditions de la question (a). Peut-on être dans les conditions de la question (a) ? //indice: cette question est la contraposée de la réponse à la question (a).

La condition dans (a) sur l'API de signature de Bob, qui n'accepte que des phrases vraies, sera implémentée dans Section 7.2.2 et Section 7.3.1 (modulo la situation différente), en faisant utiliser cette API par un certain protocole de plus haut niveau exécuté par Bob. Comme remarqué dans Exercice 27, le modèle est alors que l'adversaire ne peut utiliser que l'API de ce protocole de haut niveau, pas l'API de signature de Bob lui-même.
En conclusion, on obtient le protocole suivant permettant à Alice de réaliser un key exchange au sens de Definition 12, i.e., *sans connaître l'identité de Bob à l'avance*, et de permettre à Alice et Bob de s'identifier mutuellement sans que le contenu de leurs messages publics révèle qui ils sont. D'abord, Alice se génère une paire de clés éphémère $(\mathsf{sk}_P, \mathsf{vk_P})$. Puis elle envoie une demande à Bob, signée par $\mathsf{vk_P}$ (et $\mathsf{vk_P}$en pièce jointe) où elle lui demande une verification key. Bob se crée une paire de clés éphémère: $(\mathsf{sk}_Q, \mathsf{vk_Q})$ et envoie la $\mathsf{vk_Q}$à Alice. Puis Alice et Bob réalisent un plain signature-based Diffie-Hellman (Figure 11). Et enfin ils se prouvent leur identité à travers le secure channel, en utilisant la technique de l'Exercice 50. Ce protocole est *sous-optimal* car il coûte un *aller retour préliminaire*, et oblige Alice et Bob à générer tous les deux des signature key pairs éphémères. Ils doivent en outre utiliser ces dernières *en plus* de leurs key pairs de long terme, i.e., celles inscrites sur leurs certificats et donc qui leur servent à prouver leur identité.

7.2 Deux anonymes ayant échangé une clé peuvent-ils se prouver leur identité?

On cherche toujours à résoudre le problème où Alice et Bob, qui *ne se connaissent pas à l'avance*, ne veulent pas que le *contenu* des messages, qu'ils envoient sur le insecure channel, révèle à l'adversaire qu'ils sont en train d'implémenter un secure channel entre eux. Ils veulent néanmoins se prouver leur identité (au moins Bob à Alice), donc de façon confidentielle. On cherche maintenant à faire plus efficace que la méthode générique suggérée dans Section 7.1.4. On rappelle que cette méthode consistait à ce que: Alice génère une key pair temporaire, demande à Bob son identité, Bob génère une key pair $(\mathsf{sk}_Q, \mathsf{vk_Q})$ temporaire et donne $\mathsf{vk_Q}$ à Alice, puis Alice et Bob exécutent le signature-based Diffie-Hellman Figure 12 relativement à ces key pairs, obtiennent une ssk leur permettant d'implémenter un secure channel avec $\mathsf{AuthSymEnc}$, puis enfin exécutent l'Exercice 50 à travers le secure channel pour se prouver leurs identités.
On va alléger la partie publique de Diffie-Hellman à l'extrême: on enlève toutes les signatures. Désormais, la seule garantie apportée à Alice est que *si* une machine honnête a output relativement à son X et au Y qu'elle a reçu, alors cette machine honnête et elle ont la même ssk, donc sont reliées par un secure channel implémenté par $(\mathsf{AuthSymEnc}(\mathsf{ssk}, \bullet), \mathsf{AuthSymDec}(\mathsf{ssk}, \bullet))$. Comment, dans cette situation, Alice peut-elle arriver à obtenir la garantie que ce co-owner de ssk soit bien Bob, juste en communiquant avec lui à travers le secure channel implémenté avec la ssk? La discussion de Section 1.6 montre que, si elle utilise ce channel comme une ressource en boîte noire, elle n'aura aucun espoir de distinguer Bob d'un man-in-the-middle. Si une solution existe, elle doit donc réutiliser des informations obtenues au cours du key-exchange, par exemple, la ssk elle-même.

7.2.1 Éviter man-in-the-middle avec des (binding) session IDs

Pour attaquer le problème, on va le simplifier: on ne demande plus la privacy, i.e., Alice est par un vanilla authenticated channel à un certain Q anonyme. Dans le cas qui nous intéresse, ce authenticated channel est implémenté avec une ssk commune, donc par exemple avec la méthode de Sections 4.1.1 and 4.1.3, qui utilise les MACs. On rappelle que la garantie apportée par le authenticated channel est que, qui que Q soit réellement, il est par définition l'émetteur et le récepteur unique de tous les messages envoyés et reçus par Alice sur ce channel. Par exemple, sur la Figure 2 de gauche, Q à gauche et Alice à droite, se retrouvent côte à côte dans un lieu public. Pour simplifier, on suppose que Alice a reçu un certificat (Bob,$\mathsf{vk_B}$) et Q lui affirme (donc *via le authenticated channel*) être Bob. Donc il ne reste plus à Alice qu'à déterminer oui ou non si Q a l'identité de Bob, c'est à dire est capable de générer des signatures valides pour la public signature verification key $\mathsf{vk_B}$ de Bob.
Une tentative naïve consiste à faire le protocole scolaire suivant, dit de *(interactive) identification* ([BS23, §18.6.1.1]). Alice envoie un message public r à Q. Par exemple sur la Figure 2 de gauche, r est la phrase prononcée. Alice s'attend à ce que son correspondant réponde par une signature σ sur r valide pour $\mathsf{vk_B}$ (on précise que ce n'était pas dans le scénario de l'image). En particulier, r doit avoir être généré uniformément de taille suffisamment grande. Sinon on risque la situation où Q serait en réalité différent de Bob, et aurait appris σ car Bob l'aurait générée dans un autre contexte. Mais même avec un r suffisamment grand, Q peut néanmoins passer la vérification, pour la raison générale expliquée dans Section 1.6. Concrètement, comme décrit dans [DOW92, §2], Q se ferait passer pour Alice auprès du vrai Bob, qu'on peut imaginer juste à gauche à l'extérieur du cadre de l'image, obtiendrait σ sur r, et le relaierait à Alice.

Figure 2: Left: Schoolbook interactive identification ([BS23, §18.6.1.1], Q-in-the-middle-prone). Right: session-ID-based non-interactive identification.

Cette tentative échoue car elle tombe dans le travers rappelé à la fin de l'introduction de Section 7.2, c'est à dire qu'elle utilise le authenticated channel en boîte noire. On rappelle qu'on a suggéré qu'il est possible de résoudre le problème si le authenticated channel a un trait caractéristique, que au moins Alice et Q connaissent, et qui le distingue de tous les autres. Par exemple cela s'appelle un *channel binding* dans [BS23, §21.8], ou "session ID" dans [BPR00, Remark 1]. On l'appellera *(binding) session ID* pour insister sur le fait qu'il doit avoir la propriété que deux channels ne peuvent avoir le même. La solution est alors qu'Alice attende de recevoir une *signature de Bob*, non pas sur r, mais sur le session ID du authenticated

channel anonyme entre Q et elle. En fait, c'était déjà l'idée utilisée dans l'Exercice 50, même si la situation était plus simple (la $\mathsf{vk_Q}$ *était* l'identité de Q, donc c'était encore plus binding). Par exemple, pour le channel de la Figure 2 de droite, un (binding) session ID pourrait être constitué par: l'apparence de Alice telle que Q la décrit, ainsi que l'heure et le lieu etc. Dans le cas qui nous intéresse, Alice et Q sont reliés par un vanilla secure channel implémenté par symmetric authenticated encryption, avec une clé ssk qu'ils ont reçue d'un KeyExch (ou, si le channel est public, un simple MAC avec une clé $\mathsf{k_m}$). Donc, dans ce cas, il y a plusieurs possibilités de session ID (1) un auth-ciphertext (ou un simple MAC) d'une valeur par défaut, e.g., de 0; (2) la paire (X, Y) de Diffie-Hellman utilisée pour le key-exchange ayant conduit à ssk. C'est cette dernière solution qui est retenue par TLS 1.3.

7.2.2 Putting it together: TLS 1.3 avec one-sided identification confidentielle

On commence par dérouler, dans Figure 14, la solution suggérée dans Section 7.2.1 dans le cas le plus simple: one-sided authentication de Bob uniquement, et sans anonymité des messages. Néanmoins, cette solution résout déjà le problème posé par la mauvaise implémentation de l'Exercice 49. Il s'agit de la simplification en one-sided de "Sigma" ("Sign-and-MAC" [Kra03]), qui est l'ancêtre direct de TLS 1.3.

inputs d'Alice	**Public parameters :**	**inputs de Bob**
$(\mathsf{vk_A}, \mathsf{sk}_A)$	$(\mathbb{G}, +, 0)$ groupe d'ordre q, $G \in \mathbb{G}$ générateur, H une KDF	$(\mathsf{vk_B}, \mathsf{sk}_B)$
$\alpha \xleftarrow{\$} \mathbb{Z}/q\mathbb{Z}$, $X \leftarrow [\alpha].G$	$\xrightarrow{\quad X \quad}$	Upon receiving X' , $\beta_i \xleftarrow{\$} \mathbb{Z}/q\mathbb{Z}$, $Y \leftarrow [\beta].G$, output $(\mathsf{ssk}' = [\beta].X')$ $\sigma_B = \mathsf{Sign}\big(\mathsf{sk}_B, (X', Y)\big)$ $\mathsf{k_m} \leftarrow H(\mathsf{ssk}')$ $t_B \leftarrow \mathsf{MAC.Sign}_{\mathsf{k_m}}\big(\mathrm{Cert}(\mathrm{Bob}, \mathsf{vk_B})\big)$
Upon receiving: $(Y', \sigma'_B, t'_B, \mathrm{Cert}(\mathrm{Bob}, \mathsf{vk'_B}))$ s.t. σ'_B on (X,Y') valid for $\mathsf{vk'_B}$ and s.t. $\mathsf{MAC.Verify}(\mathsf{k_m}, \mathrm{Cert}(\mathrm{Bob}, \mathsf{vk'_B}), t'_B) = \mathsf{accept}$ output (Bob', $\mathsf{ssk} = [\alpha].Y'$)	$\xleftarrow{\quad Y, \sigma_B, t_B, \mathrm{Cert}(\mathrm{Bob}, \mathsf{vk_B}) \quad}$	

Figure 14: Diffie-Hellman-based key-exchange for *post*-specified peers with one-sided identification (Definition 12): one-sided version of "Sigma", from [Kra03, §5.1]. In TLS 1.3, H is the (deterministic) key derivation function "HKDF" ([BS23, §8.10.5]).

Pour anonymiser l'identité de Bob, il lui suffit d'envoyer le contenu non-anonyme de son message: (σ_B, t_B, Cert(Bob,$\mathsf{vk_B}$)) sous la forme d'un ciphertext encrypté sous une clé symétrique $\mathsf{k_{ed}}$ dérivée de ssk, par exemple: $(\mathsf{k_m}, \mathsf{k_{ed}}) \leftarrow H(\mathsf{ssk}')$: c'est la version one-sided de "Sigma-I" ([Kra03, §5.2]). Enfin pour permettre à Alice de prouver son identité à Bob de façon confidentielle, il lui suffit de faire de même dans un 3e message en réponse à Bob, mais cette fois avec un *authenticated* symmetric encryption scheme.

Exercice 51 (malléability$\rightarrow$ identity-misbinding)**.** On considère une version dégradée du protocole "Sigma-I" ([Kra03, §5.2]) où Alice et Bob n'utilisent que AES-CTR pour le 3e message, d'Alice, qui est donc

malléable. Décrire une attaque conduisant à un output d'Alice égal à (Bob,ssk) et de Bob égal à (Alice',ssk) *avec la même* ssk. Indice: l'attaque de [Kra03] sur STS, ou [BS23, §21.2] Variation 6.
Expliquer en quoi ce protocole ne respecte donc pas la Definition 12 (indice: Remark 13).

Remark 52. Dans TLS 1.3, les ciphertexts des messages de Bob et d'Alice sont spécifiés être encryptés avec un *authenticated* symmetric encryption scheme. Donc cela joue déjà le rôle de MAC, il semble donc en fait inutile d'envoyer en plus les message authentication tags t_B et t_A. Ce design sous-optimal est sans doute hérité de "Sigma-I" ([Kra03, §5.2]).

TLS 1.3 est décrit dans [BS23, §21.10] en version lisible, et dans [Dow+21] dans la version avec tous les détails mais néanmoins plus lisible que le RFC [IET18b].

7.3 Encryption-based KE (1RTT, confidential identification, post-specified peers)

Le *encryption-based* key-exchange est une classe d'implémentations de key exchange, qui est encore plus simple à décrire que Diffie-Hellman. En résumé, Alice génère une ssk, puis envoie à Bob un ciphertext de cette ssk encrypté avec la clé publique de Bob. En fait, on va voir dans Section 7.3.2 quelques exemples de [BS23, §21.2], qui montrent que cette simplicité augmente en fait les possibilités d'attaques. La raison est le choix de la clé ne dépend maintenant que du message de Bob, plus de celui d'Alice. Donc cela explique pourquoi les implémentations correctes, présentées dans Section 7.3.1 et Section 7.3.3, sont en fait plus compliquées que cette description. Certains des futurs standards sont prévus pour jouer le rôle du public key encryption scheme utilisé dans ce key exchange. C'est par exemple le cas de "Kyber" (Section 5.2.6), qui proposaient d'ailleurs un autre encryption-based key-exchange dans [Bos+17, §5].

7.3.1 AKE1: encryption-based key-exchange avec post-specified peers, en 1 aller-retour

On décrit maintenant le (encryption-based) authenticated key exchange de [BS23, §21.2], appelé AKE1, dans le modèle de *static security*, où les participants ne deviennent jamais corrompus dans le futur. Il a l'avantage de ne pas imposer que les peers aient une quelconque information préalable l'un sur l'autre (Definition 12). Il ne prend qu'un aller-retour de messages, mais a l'inconvénient que les identités des peers sont révélées par le contenu des messages publics.
Une particularité est que le PKE doit vérifier strictement plus que la spécification Figure 7 dans Section 5.2.2 Figure 7. Notamment, la 6e "insecure variation", décrite dans [BS23, §21.2.1], considère le mauvais où AKE1 est instancié avec un PKE malléable. Ils décrivent une attaque qui conduit à une usurpation d'identité, qui est exactement la même que celle décrite dans Exercices 41 and 45. Un tel exemple de PKE malléable est le "hybrid encryption" avec AES-CTR (Section 5.2.3).

7.3.2 Insecure variation de AKE1: encrypt r instead of Bob's identity $\rightarrow$ identity-misbinding

On considère la insecure variation de AKE1, décrite dans la Figure 16, et appelée "Variation 5" dans [BS23, §21.2.1]. Bob n'inclut plus son identité "Bob" dans le ciphertext c, il inclut à la place r. De même, Alice ne vérifie plus que la décryption de c est de la forme (ssk,"Bob"). Elle vérifie à la place si elle est de la forme (ssk,r), avec le r qu'elle avait envoyé dans le premier message. Le reste ne change pas: Alice parse le certificat Cert(Bob,vk_{B}), vérifie que la signature σ est valide pour la verification key vk_{B} contenue dans ce certificat, et si oui, elle output (ssk,"Bob"), où Bob est l'identité contenue dans ce même certificat.
Elle s'expose à l'attaque suivante. Soit R une machine corrompue par l'adversaire, qui a légitimement généré une paire de signature keys $(\mathsf{vk}_R, \mathsf{sk}_R) \xleftarrow{\$} \mathsf{KeyGen}(\$)$ et légitimement obtenu un certificat sur vk_R

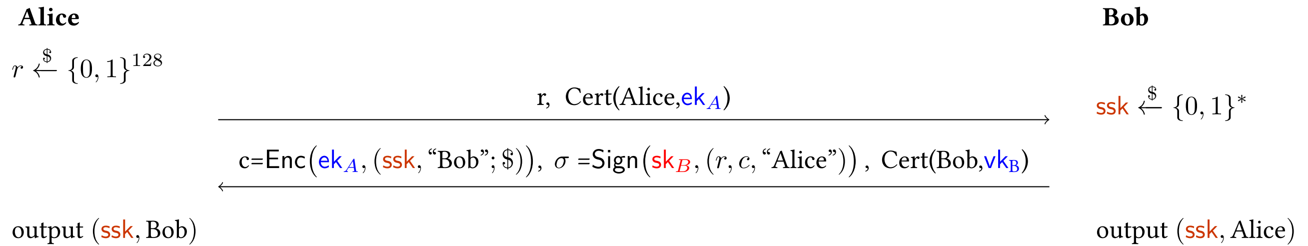

Figure 15: Le “encryption-based” authenticated key-exchange AKE1 de [BS23, §21.2]. Il a comme paramètres publics un public key encryption scheme (PKE) et un signature scheme. La notation Cert(Alice,ek_A) désigne un certificat de la “certification authority”, CA, qui atteste que Alice est l’owner de la public encryption key ek_A, c’est à dire que Alice est bien la seule personne à connaître la secret decryption key dk_A. La notation Cert(Bob,vk_B) désigne un certificat de CA qui atteste que Bob est bien l’owner de la public signature verification key vk_B. Chaque participant, Alice et Bob, verifie le certificat qu’il reçoit. En outre, Alice verifie la signature σ par rapport à la verification key vk_B de Bob, elle vérifie aussi que $\mathsf{Dec}(\mathsf{dk}_A, c)$ est bien un plaintext de la forme (ssk,“Bob”).

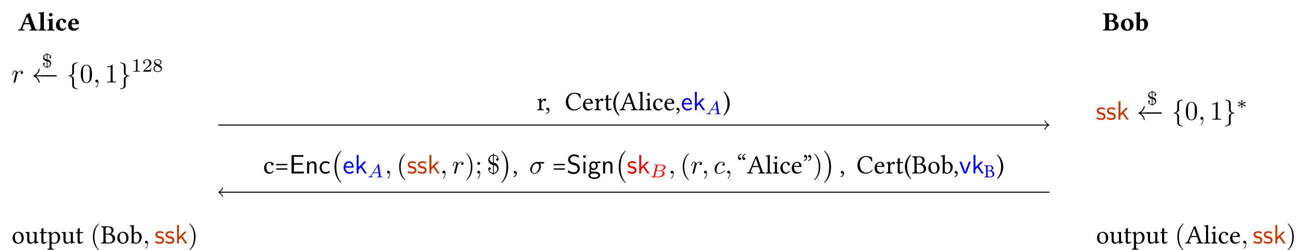

Figure 16: Insecure variation de AKE1, où Bob n’inclut plus son identité “Bob” dans le plaintext de c, mais r à la place, et de même Alice ne vérifie plus si cette identité est présente dans la décryption de c, mais vérifie si r est présent à la place.

pour son identité: Cert(R,vk_R). L'adversaire bloque le message envoyé par Bob, et injecte à la place le message (c_R, σ_R,Cert(R,vk_R)) comme décrit dans la Figure 17:

Alice | **Bob**

$r \xleftarrow{\$} \{0,1\}^{128}$

$\xrightarrow{\text{r, Cert(Alice,} \mathsf{ek}_A)}$ $\mathsf{ssk} \xleftarrow{\$} \{0,1\}^*$

$\xleftarrow{c_R = \mathsf{Enc}(\mathsf{ek}_A, (\mathsf{ssk}, r); \$),\ \sigma_R = \mathsf{Sign}(\mathsf{sk_R}, (r, c, \text{“Alice”}))\ ,\ \text{Cert(R,}\mathsf{vk_R})}$ output (Alice, ssk)

output (R, ssk)

Figure 17: Attaque sur l'insecure variation de AKE1 de la Figure 16.

Où R est une machine corrompue contrôlée par l'adversaire. La conséquence est que Alice output (R, ssk) alors que Bob output (Alice,ssk) pour la même ssk. Or, une garantie apportée par $\mathsf{KeyExch}$ avec peers post-spécifiés (Definition 12) est en particulier l'“implicit authentication”. Elle garantit que, sauf probabilité négligeable, il ne peut exister deux machines honnêtes distinctes qui reçoivent la même ssk mais n'output pas les identités l'une de l'autre. L'attaque précédente montre que ce protocole ne satisfait pas cette garantie, il ne respecte donc pas la Definition 12.
Comme expliqué dans Remark 9, l'attaque précédente peut-être exploitée pour faire du identity misbinding (Alice crédite à R les auth-ciphertexts générés par Bob).
À noter qu'une insecure variation encore pire est décrite dans [BS23, Variation 4], où il n'est même pas requis que Bob encrypte r. Elle autorise une exploitation légèrement plus grave, appelée *replay attack*. On considère Alice et Bob qui ont eu une *ancienne* ession de key exchange paramétrée par r il y a longtemps, à l'issue de laquelle ils ont output ssk puis l'ont utilisée pour communiquer via $\mathsf{AuthSymEnc}$. Puis, Alice ouvre une nouvelle session de key exchange paramétrée avec r'. La même attaque permet à R de devenir le peer d'Alice dans cette nouvelle session, et permet de forcer l'output par Alice de la même ssk. Ainsi, r peut donc réenvoyer à Alice des auth-ciphertexts générés par Bob au cours de l'*ancienne* session.

7.3.3 AKE4 Encryption-based key-exchange avec identification *confidentielle*

Le protocole AKE4 est un authenticated key exchange, toujours avec post-specified peers (Definition 12), décrit dans [BS23, §21.5]. Pour simplifier, sa version avec identification seulement one-sided, donc de Bob, est décrite dans [BS23, Figure 21.9]. Par rapport à AKE1, les messages de Bob ne révèlent son identité qu'à Alice. Un avantage supplémentaire par rapport à AKE1 est que le PKE dans AKE4 peut être n'importe lequel vérifiant la spécification Figure 7, en particulier, peut même être malléable. Intuitivement, la raison est que Alice utilise désormais une public encryption key créée pour l'occasion, donc qu'elle n'utilisera que pour une seule session. On voit en particulier que cela évite l'attaque décrite dans [BS23, §21.2] sur la “variation 6” de AKE1 avec malléabilité. Donc par exemple, le futur standard de PKE “Kyber.CCAKEM” [Ava+21], qui est CCA (Section 5.2.4) semble donc sur-spécifié pour un usage dans AKE4, la version plus simple et malléable “Kyber.CPA” semble suffire. L'autre différence pour passer de AKE1 à AKE4 est simplement le fait que Bob envoie ses données non-anonymes sous la forme de *authenticated*-symmetric ciphertexts avec ssk (plus précisément, une autre ssk_1 dédiée à cet usage, dont il envoie également le ciphertext à Alice. Ce dédoublement de clés n'est pas forcément nécessaire, comme discuté dans [Brz+13]).

8 Davantage de sécurité et d'anonymité

8.0.1 Forward secrecy

Ce nom est un faux ami. Il s'agit d'une garantie qui vaut si une certaine machine, Bob, est honnête au moins jusqu'au moment où une certaine action spécifique est faite. La garantie est que si Bob devient corrompu après cette action, alors, l'état interne de Bob n'apprend aucune information supplémentaire à l'adversaire sur les plaintexts envoyés à Bob *avant* cette action. Évidemment, rien ne peut empêcher l'adversaire d'apprendre les plaintexts que Bob aurait pu garder en mémoire au moment où il devient corrompu. C'est pour cela qu'on a supposé que les machines effacent les plaintexts reçus après les avoir output. Le type d'action à réaliser porte le nom vague de *key rotation*. Plus précisément, dans le cas de (authenticated) symmetric encryption, l'action est analogue celle décrite dans Exercice 27 Item (ii). Elle consiste pour Alice et Bob à, chacun, updater la session key de façon déterministe $\mathsf{ssk} \leftarrow H(\mathsf{ssk})$ (et donc effacer l'ancienne). Cette opération s'appelle un *roll forward* de la (des) traffic key(s). Dans TLS 1.3 H est la fonction HKDF, cette action est faite au cours de longues sessions. Dans le cas de public key encryption, l'action consiste simplement pour Bob à effacer son ancienne paire de clés et à générer (et faire enregistrer) une nouvelle paire de clés. Par exemple si Bob est un serveur Tor (un "Tor service"), il génère tous les jours une nouvelle encryption key (voir ci-dessous pour son usage).

8.0.2 Post-compromise security

Il s'agit d'une garantie qui vaut si une certaine machine, par exemple Alice, avait été corrompue (à définir) puis est maintenant décorrompue. Alors, après qu'une action spécifique (à définir) est réalisée, il est garanti que tous les protocoles de authenticated et de secure channels auxquels Alice participe, même ceux initiés avant l'action, retrouveront leur comportement attendu. Par exemple, dans Exercice 27, la "corruption" de Alice dans le passé a consisté en un accès de l'adversaire à son API $\mathsf{MAC}.\mathrm{Sign}(\mathsf{k_m}, \bullet)$. L'action spécifique à faire, décrite dans Item (ii), était un *roll forward* de la MAC key $\mathsf{k_m}$. Donc curieusement c'est la même action que celle à faire, dans Section 8.0.1, pour garantir la forward secrecy, alors même qu'on parle ici de corruption dans le passé, et pas dans le futur. En réalité, comme observé dans Exercice 27 Item (iii), un roll-forward ne sert à rien si la corruption d'Alice était totale, c'est à dire si $\mathcal{A}$ avait appris la MAC key $\mathsf{k_m}$ elle-même. Par exemple dans le cas de public key encryption et d'un digital signature scheme, l'action consiste à générer et faire enregistrer une nouvelle paire de clés. Elle consiste en outre à demander à CA de révoquer le certificat sur l'ancienne signature key.

8.0.3 Anonymité (DNS et Tor)

On a vu des key exchanges tels que le *contenu* des messages, envoyés par Alice et Bob sur le insecure channel, ne révèlent pas le fait qu'ils sont en train d'implémenter un secure channel entre eux (TLS 1.3 et AKE4 Section 7.3.3). Leur reste à masquer le fait qu'ils sont les expéditeurs et les récepteurs de ces messages. Un DNS est une machine à laquelle est reliée Alice par un secure channel, et qu'elle utilise pour communiquer avec Bob de la façon suivante.

Exercice 53 (DNS)**.**

$$\text{Alice} \xleftrightarrow{\mathsf{sc}} P_1 \xleftrightarrow{\text{insecure}} \text{Bob} \quad . \tag{17}$$

On suppose de P_1 envoie à Bob tous les messages que Alice lui donne via le secure channel sc, et transfère à Alice tous les messages qu'il reçoit de Bob. Alice envoie à P_1 et reçoit de P_1 des messages réalisant un

key exchange de elle et Bob avec identification confidentielle two sided. Cela implémente-t-il un secure channel entre P_1 et Bob ?

L'exercice précédent montre que un DNS *ne joue pas* le rôle d'un man in the middle entre Alice et Bob, au sens de Section 1.6. Notamment, le DNS P_1 n'apprend rien du contenu des échanges ultérieurs entre Alice et Bob. Il n'en demeure pas moins qu'il sait que Alice échange avec Bob. Il pourrait donc la trahir en donnant cette information à l'adversaire. Un réseau dit de "onion routers", par exemple implémenté par "Tor" ([DMS04]), amoindrit ce risque avec une cascade de DNS. Précisément, au lieu de contacter directement Bob, Alice ordonne à P_1 de contacter une autre machine P_2, appelée *onion router*, puis exécute avec P_2 un key exchange *via* les messages forwardés par P_1.

$$\text{Alice} \xleftrightarrow{\mathsf{sc}} P_1 \xleftrightarrow{\text{insecure}} P_2 \quad . \tag{18}$$

Sous l'hypothèse que P_1 continue de forwarder les messages que Alice lui donne via sc, l'exercice précédent montre que cela implémente donc un secure channel, appelé sc_2, entre Alice et P_2. Puis Alice recommence et ordonne à P_2, *via* sc_2, de contacter un troisième P_3 etc. À ce stade P_1, quand bien même il relaie tous les messages entre Alice et P_2 via sc, *ne sait donc même pas qui est* P_3, grâce à la propriété que sc_2 est un secure channel. Puis, arrivée à un certain P_m, Alice envoie un ciphertext à Bob via P_n, encrypté avec la clé publique de Bob, dont le contenu informe Bob qu'Alice est joignable à un autre P'_m (qu'elle contrôle via une autre cascade de onion routers). P'_m est appelé le "rendez-vous". Bob implémente donc un secure channel avec P'_m, lui aussi via un certain une cascade de onion routers (donc via un certain Q_n).

9 Lightweight et embarqué.

"A pig's got to fly" (from studio Ghibli)

9.1 Attaques de type *side channel*

Ce sont des attaques qui exploitent des implémentations qui ne respectent pas les spécifications, et/ou le fait que les machines victimes leakent plus d'information, ou subissent plus d'influence, que ne le prévoient les spécifications.

9.1.1 Time

exploitent le fait que des implémentations de victimes ne respectent pas la spécification que le temps d'exécution ne dépend que de taille des inputs. Par exemple dans [Ber05], la machine victime a une implémentation de AES dont le temps d'exécution varie en fonction de l'input. L'attaque de [Ber05] permet de récupérer la clé secrète k que la victime (un serveur) utilise pour communiquer avec de l'authenticated symmetric encryption, avec un client tiers. L'adversaire $\mathcal{A}$ mesure les temps d'exécution du AES de la machine victime sur plusieurs inputs de son choix, de la façon suivante. Il envoie des requêtes de AuthSymDec à la machine victime, pour des ciphertexts de son choix. Il obtient l'information de combien de temps la machine victime a mis pour exécuter chaque AuthSymDec (dans son attaque, il l'apprend car la victime est programmée pour répondre un message après avoir fini). A priori cette influence et ce leakage sont de type chosen ciphertexts (CCA), donc non couverts par nos spécifications (de type CPA seulement). Mais en réalité, d'une certaine façon cette attaque est *neutralisée par nos spécifications*. En effet on a spécifié que les algorithmes terminent en temps constant sur des inputs de taille donnée. Donc, *si* l'implémentation de la victime était correcte, l'adversaire $\mathcal{A}$ pourrait *entièrement prédire* les temps d'exécution observés, à partir de sa connaissance totale du harware et sofware de la victime (Section 2.6). Donc, cette attaque n'apprendrait rien à $\mathcal{A}$ qu'il ne sait déjà.
En détail, $\mathcal{A}$ possède une copie du programme de la victime qui calcule AES, et l'exécute sur une copie du même hardware. Il fixe une clé arbitraire k', et mesure le temps pris pour calculer $AES(k', r')$ sur plusieurs inputs r' de son choix. Plus précisément, il fait varier le premier byte r'_1 de r', jusqu'à trouver celui qui maximise le temps d'évaluation de $AES(k', r')$. Puis, $\mathcal{A}$ obtient l'information du premier byte r_1 de r qui maximise le temps d'évaluation de $AES(k, r)$. Il arrive à obtenir cette information sans connaître k de la façon suivante: il envoie plusieurs auth-ciphertexts requêtes à la victime avec un nonce fixe r sauf le premier byte r_1 qu'il fait varier. Le byte r_1 qui maximise le temps de la réponse de la victime est celui cherché. De ces deux informations, il déduit immédiatement le premier byte de la clé k ([Ber05, p. 3]). Puis il recommence pour les 15 autres bytes, et en déduit la totalité de k. De façon remarquable, cette attaque récupère directement la clé k et pas seulement une round key, car elle exploite la structure simple du premier round du key scheduling.

9.1.2 accept/reject

Ciblent une victime qui applique Dec ou SymDec. La victime leake de l'information sur la valeur de l'output (de Dec ou SymDec) y compris sur des input ciphertexts mal formés. Ce leakage fait sauter la garantie de secrecy, puisqu'il n'est pas prévu par le security game. Un premier exemple est une vicime qui *pourrait* ne pas leaker, mais le fait. Par exemple dans [APW09], la victime SymDec avant même d'avoir vérifié le MAC. Donc possiblement elle SymDec des inputs mal formés, i.e., qui n'ont pas été générés par l'API AuthSymEnc donc n'ont pas été produits par SymEnc. Puis, elle a un comportement qui dépend de l'output. Cette différence de comportement est exploitée dans [APW09] pour retrouver les premier bits de plaintext chaque bloc de 128 bits de ciphertexts.
Un deuxième exemple est lorsque le leakage est inévitable, et donc le PKE devrait respecter la spécification CCA (hors programme). Les attaques exploitent le fait que le PKE utilisé n'est pas CCA. Concrètement, CCA garantit la secrecy dans des security games où $\mathcal{A}$ bénéficie d'une influence et d'un leakage plus forts, au sens suivant. $\mathcal{A}$ peut envoyer des ciphertexts c requêtes à la victime choisis par lui, possiblement mal formés. Puis la victime calcule $y \leftarrow \mathsf{Dec}(sk, c)$ et donne de l'information à $\mathcal{A}$ sur y et/ou le temps de calcul //avec nos spécifications de temps de calcul fixe, le leakage du temps de calcul est autorisé de base, puisque par définition il n'apprend rien de nouveau à $\mathcal{A}$ (Section 2.6).. Pour qu'un public key encryption puisse être

utlisé pour faire un encryption-based key exchange, il doit satisfaire une spécification de type CCA. En effet lorsque la victime Bob reçoit de Alice un ciphertext c, elle s'attend à ce que c ait été généré comme $c \leftarrow \mathsf{Enc}(\mathsf{ek}, \mathsf{ssk})$, avec ek la encryption key de Bob et ssk la symmetric session key choisie par Alice. Donc, Bob calcule $y \leftarrow \mathsf{Dec}(\mathsf{dk}, c)$. Si y est un plaintext, alors Bob l'utilise en guise de ssk pour communiquer avec Alice. Si $y =$ reject, Bob aura forcément un autre comportement, puisque reject n'est pas utilisable comme ssk. Donc, cette différence de comportement est une information si y est un plaintext (accept) ou s'il est reject. Dans l'attaque de Bleichenbacher 1998 [BS23, §12.8.3], l'attaquant $\mathcal{A}$ joue le rôle d'une Alice anonyme dans un encryption-based key exchange utilisant le public key encryption scheme (PKE) appelé "RSA avec le padding PKCS #1 v1.5". Cette attaque exploite que ce PKE *n'est pas* CCA.
$\mathcal{A}$ créée le ciphertext c' de son choix, et obtient de la victime S l'information si $\mathsf{Dec}(\mathsf{dk}, c')$ commence (accept) ou non (reject) par 0x0002 //Il s'agit du préfixe fixe du padding de PKCS #1 v1.5. Un padding plus moderne de RSA, OAEP, évite ce problème. Des futurs standards, comme Kyber, évitent entièrement ce problème puisqu'ils sont CCA. Dans ce modèle, l'algorithme de Bleichenbacher permet à $\mathcal{A}$ d'obtenir, après plusieurs requêtes de ce type, la valeur de $\mathsf{Dec}(\mathsf{dk}, c)$ pour un ciphertext c de son choix. L'algorithme de Bleichenbacher a été utilisé dans les attaques réelles [Mey+14; BSY18; BSY18; Ron+19]. Notamment, celle de [BSY18] en 2017 contre un serveur S de Facebook, a réussi à produire une signature sur le document de leur choix, qui est reconnue valide pour la clé du certificat https de facebook.com //cette attaque exploite la mauvaise implémentation de ce serveur S, qui utilisait la même clé RSA pour Sign et pour Dec. Le document en question était la chaîne de caractères ("`We hacked Facebook with a Bleichenbacher Oracle (JS/HB)".`

9.1.3 accept/reject + malware

Un deuxième type d'attaques exploitant le side channel accept/reject sont celles de POODLE contre SSL 3.0 ([BS23, §9.4.2]), et celle de Lucky13 contre TLS 1.0, 1.1 et 1.2 [AP13]. La victime est un client C qui communique avec S. Chaque plaintext envoyé par C à S contient une chaîne de bits fixe appellée cookie, le but de $\mathcal{A}$ est de connaître sa valeur. Dans leur modèle, pour simplifier, $\mathcal{A}$ peut convaincre C (i) de se connecter à S, i.e., faire un KeyExch pour obtenir une ssk commune avec S (que ne connaît pas $\mathcal{A}$), (ii) et générer un auth-ciphertext $c' \leftarrow \mathsf{AuthSymEnc}(\mathsf{ssk}, (P \| \mathsf{cookie}))$ pour un P bien choisi par $\mathcal{A}$ (qui ne connait pas non plus cookie) //dans leur attaque, $\mathcal{A}$ y arrive en faisant exécuter un malware au browser de C, par exemple un javascript. Puis $\mathcal{A}$ crée plusieurs modifications c'_i de c, les envoie vers S sur le insecure channel utilisé par C et S, et apprend si C a accept ou reject c. En recommençant plusieurs fois $\mathcal{A}$ peut apprendre la valeur de cookie. À noter qu'à chaque reject, S ferme la connexion, ce qui oblige $\mathcal{A}$ à convaincre C d'en ouvrir une nouvelle avec S. La ssk est donc différente à chaque fois mais ça n'empêche pas l'attaque. À noter que ces attaques exploitent la *mauvaise* classe d'implémentations de authenticated symmetric encryption, appelée "Mac-then-Encrypt", qui obligeait C à SymDec avant de pouvoir MAC.Verify.

9.1.4 Power

on suppose que $\mathcal{A}$ reçoit le leakage de la puissance de calcul utilisée à chaque instant par le programme victime [Gue+22], par exemple déduite d'une mesure de consommation d'énergie, par exemple via le rayonnement électromagnétique.

9.1.5 Cache

on suppose que $\mathcal{A}$ reçoit le leakage des hit/miss des accès au cache du programme victime, et/ou peut écrire lui même des données dans le cache accédé par le programme victime, puis observer combien de temps ces écritures y restent. Par exemple en 2016 sur Amazon EC2, une VM contrôlée par $\mathcal{A}$ pouvait détecter si

elle était hébergée sur la même machine physique que la VM victime cible, puis arrivait à retrouver la clé RSA secrète de la victime [İnc+16]. De nombreux exemples dans [Ron+19, §II H].

9.1.6 Autres

Observer la lumière émise par les fibres optiques [Jai23].

9.1.7 Couplages électromagnétiques.

screaming channels: le rayonnement EM d'un composant protégé fuite vers un amplificateur chargé de l'émission radio. *Row-hammer:* lecture répétée de deux lignes de mémoire prenant en sandwich une ligne au milieu. Cela finit par injecter des fautes dans la ligne du milieu, par couplage EM. *CLK screw:* variations brusques de la tension d'alimentation qui induisent des fautes dans le composant.

References

[ANS21] ANSSI. *guide de sélection d'algorithmes cryptographiques*. `https://www.ssi.gouv.fr/uploads/2021/03/anssi-guide-selectioncrypto-1.0.pdf`. 2021.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *IEEE Symposium on Security and Privacy*. 2013.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. "Plaintext Recovery Attacks against SSH". In: *IEEE Symposium on Security and Privacy*. 2009.

[Ava+21] Roberto Maria Avanzi et al. *Kyber Algorithm Specifications And Supporting Documentation 3.02*. `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf`. 2021.

[BCK98] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols". In: *STOC*. 1998.

[Ben+11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. "Semi-homomorphic Encryption and Multiparty Computation". In: *EUROCRYPT*. 2011.

[Ber05] Daniel Bernstein. *Cache-timing attacks on AES*. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. 2005.

[Ber15] Daniel Bernstein. *Break a dozen secret keys, get a million more for free*. `https://blog.cr.yp.to/20151120-batchattacks.html`. 2015.

[Bha+14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *IEEE Symposium on Security and Privacy, SP*. `https://www.mitls.org/pages/attacks/3SHAKE`. 2014.

[Bis18] Matt Bishop. *Computer Security, 2nd Edition*. 2018.

[Bit19] Ben Perez @ Trail of Bits. *F** RSA*. `https://www.youtube.com/watch?v=lElHzac8DDI`. 2019.

[Bos+17] Joppe Bos et al. "CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM". In: *IEEE Euro S & P*. 2017.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. "Authenticated Key Exchange Secure against Dictionary Attacks". In: *EUROCRYPT*. 2000.

[BR93] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution". In: *CRYPTO*. 1993.

[Bri+21] Marcus Brinkmann et al. In: *Usenix, Black Hat and RWC (!)* `https://alpaca-attack.com/`, see also `https://thehackernews.com/2021/06/new-tls-attack-lets-attackers-launch.html`. 2021.

[Brz+13] Christina Brzuska, Marc Fischlin, Nigel P. Smart, Bogdan Warinschi, and Stephen C. Williams. "Less is more: relaxed yet composable security notions for key exchange". In: *Int. J. Inf. Sec.* (2013).

[BS23] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Version 0.6 Jan 2023. 2023.

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". In: *USENIX Security 18*. 2018.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages". In: *CRYPTO*. 2011.

[Can01] Ran Canetti. "Universally composable security: A New Paradigm for Cryptographic Protocols". In: *FOCS*. Eprint version updated February 11, 2020. 2001.

[CDN15] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[CK01] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *Eurocrypt*. long version `https://eprint.iacr.org/2001/040`. 2001.

[CK02a] Ran Canetti and Hugo Krawczyk. "Security Analysis of IKE's Signature-Based Key-Exchange Protocol". In: *CRYPTO*. 2002.

[CK02b] Ran Canetti and Hugo Krawczyk. "Universally Composable Notions of Key Exchange and Secure Channels". In: *EUROCRYPT*. 2002.

[Cor22] Jonathan Corbet. *Understanding random number generators, and their limitations, in Linux*. `https://lwn.net/Articles/887207/`. 2022.

[Cos18] Craig Costello. *A gentle introduction to elliptic curve cryptography*. `https://summerschool-croatia.cs.ru.nl/2018/slides/Introductiontoellipticcurvecryptography.pdf`. 2018.

[CS01] Ronald Cramer and Victor Shoup. "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack". In: `https://eprint.iacr.org/2001/108`. 2001. URL: `https://eprint.iacr.org/2001/108`.

[CSV16] Ran Canetti, Daniel Shahaf, and Margarita Vald. "Universally Composable Authentication and Key-Exchange with Global PKI". In: *PKC*. 2016.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. "Tor: The Second-Generation Onion Router". In: *USENIX*. 2004.

[Don22] Jason A. Donenfeld (zx2c4). *Random number generator enhancements for Linux 5.17 and 5.18*. `https://www.zx2c4.com/projects/linux-rng-5.17-5.18/`. 2022.

[Dow+21] Benjamin Dowling, Marc Fischlin, Felix GUnther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol". In: *Journal of Cryptology* (2021).

[DOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. "Authentication and Authenticated Key Exchanges". In: *Des. Codes Cryptogr.* (1992).

[Eve+14] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael M. Swift. "Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG". In: *IEEE Symposium on Security and Privacy.* 2014.

[Fen+23] X. Feng, Q. Li, K. Sun, Y. Yang, and K. Xu. "Man-in-the-Middle Attacks without Rogue AP: When WPAs Meet ICMP Redirects". In: *IEEE (SP).* 2023.

[Fin08] Hal Finney. *Bleichenbacher's RSA signature forgery based on implementation error.* https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/. 2008.

[Fis+16] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3". In: *IEEE (SP).* 2016.

[Gue+22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. "The Hidden Parallelepiped Is Back Again: Power Analysis Attacks on Falcon". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022 (2022).

[Hen+12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices". In: *USENIX.* 2012.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography.* Nov 2021 corrected version of 2017/604 eprint. 2017.

[IET05] IETF. *Randomness Requirements for Security.* https://www.rfc-editor.org/rfc/rfc4086. 2005.

[IET18a] IETF. *ChaCha20 and Poly1305 for IETF Protocols.* https://www.rfc-editor.org/rfc/rfc8439. 2018.

[IET18b] IETF. *The Transport Layer Security (TLS) Protocol Version 1.3.* https://www.rfc-editor.org/rfc/rfc8446. 2018.

[İnc+16] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud". In: *CHES.* 2016.

[Inf22] Federal Office for Information Security. *Documentation and Analysis of the Linux Random Number Generator.* https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG EN V5 0.pdf?blob=publicationFile&v=3. 2022.

[IOM12] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. "Breaking and Repairing GCM Security Proofs". In: *Crypto.* 2012.

[ISO19] ISO/IEC. *ISO/IEC 18033-6:2019 IT Security techniques — Encryption algorithms — Part 6: Homomorphic encryption.* 2019. URL: %5Curl%7Bhttps://www.iso.org/obp/ui/#iso:std:iso-iec:18033:-6:ed-1:v1:en%7D.

[Jai23] Philippe Jaillon. “Canaux cachés en pleine lumière ou comment la lumière illuminant une fibre optique pourrait permettre de retrouver les clefs de chiffrement des communication”. In: *Webinaire Risques&Cybersécurité@IMT*. Institut Mines-Télécom. Paris, France, 2023. URL: `https://hal.science/hal-04004214`.

[JKL04] Ik Rae Jeong, Jonathan Katz, and Dong Hoon Lee. “One-Round Protocols for Two-Party Authenticated Key Exchange”. In: *ACNS*. 2004.

[Jou06] Antoine Joux. *Authentication Failures in NIST version of GCM*. 2006.

[JV96] Mike Just and Serge Vaudenay. “Authenticated Multi-Party Key Agreement”. In: *ASIACRYPT*. 1996.

[Kra03] Hugo Krawczyk. “SIGMA: The ‘SIGn-and-MAc’ Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols”. In: *CRYPTO*. 2003.

[Kra05] Hugo Krawczyk. “HMQV: A High-Performance Secure Diffie-Hellman Protocol”. In: *CRYPTO*. 2005.

[Lib22] LibSodium. *Generating random data*. `https://libsodium.gitbook.io/doc/generatingrandomdata`. 2022.

[Lin06] Katz & Lindell. *Introduction to modern cryptography*. 2006.

[LP17] Atul Luykx and Kenneth G. Paterson. *Limits on Authenticated Encryption Use in TLS*. 2017.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On Ideal Lattices and Learning with Errors over Rings”. In: *EUROCRYPT*. 2010.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “A Toolkit for Ring-LWE Cryptography”. In: 2013.

[LS17] Yong Li and Sven Schäge. “No-Match Attacks and Robust Partnering Definitions: Defining Trivial Attacks for Security Protocols is Not Trivial”. In: *CCS*. 2017.

[man23] Linux manual. *getrandom(2)*. `https://man7.org/linux/man-pages/man2/getrandom.2.html`. 2023.

[Mel19] Martin Albrecht and Melissa Chase and Hao Chen and Jintai Ding and Shafi Goldwasser and Sergey Gorbunov and Shai Halevi and Jeffrey Hoffstein and Kim Laine and Kristin Lauter and Satya Lokam and Daniele Micciancio and Dustin Moody and Travis Morrison and Amit Sahai and Vinod Vaikuntanathan. *Homomorphic Encryption Standard*. Iacr eprint. 2019.

[Men05] Alfred Menezes. *Another look at HMQV*. Cryptology ePrint Archive, Paper 2005/205. 2005. URL: `https://eprint.iacr.org/2005/205`.

[Mey+14] Christopher Meyer et al. “Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks”. In: *USENIX*. 2014.

[Mic20] Daniele Micciancio. *CSE208: Advanced Cryptography*. `https://cims.nyu.edu/regev/papers/lwesurvey.pdf`. 2020.

[MVV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[NIS07] Morris Dworkin @ NIST. *SP 800-38D Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf`. 2007.

[Ole21] Peter C. Oleson. "The Breaking of JN-25 and its Impact in the War Against Japan". In: *Journal of U.S. Intelligence Studies* (2021). https://www.afio.com/publications/OLESON WIMAD Breaking of JN-25 from AFIO Intelligencer Vol26 No2 WinterSpring2021.pdf.

[Pap14] Francesco Pappalardi. *ELLIPTIC CURVES OVER FINITE FIELDS*. http://www.mat.uniroma3.it/users/pappa/missions/slides/HCMC20153.pdf. 2014.

[Pei09] Chris Peikert. "Public-key cryptosystems from the worst-case shortest vector problem". In: Full version of STOC : https://web.eecs.umich.edu/cpeikert/pubs/svpcrypto.pdf. 2009.

[PS17] A. Pellet–Mary and D. Stehlé. *Tutorial 3: PRG and Symmetric encryption schemes*. https://apelletm.pages.math.cnrs.fr/page-perso/documents/enseignement/Tdscrypto/TD03.pdf. 2017.

[Reg09] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *J. ACM* (2009).

[Ron+19] Eyal Ronen et al. "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations". In: *IEEE Security and Privacy (SP)*. 2019.

[SFW20] Cyprien Delpech de Saint Guilhem, Marc Fischlin, and Bogdan Warinschi. "Authentication in Key-Exchange: Definitions, Relations and Composition". In: *IEEE CSF*. 2020.

[Tso22a] Theodore Ts'o. *01-04 5:55 reply to [PATCH v2] random: avoid superfluous call to RDRAND in CRNG extraction*. https://lore.kernel.org/lkml/20211230165052.2698-1-Jason@zx2c4.com/t/. 2022.

[Tso22b] Theodore Ts'o. *03-22 [PATCH] random: allow writes to /dev/urandom to influence fast init*. https://lwn.net/ml/linux-kernel/YjqVemCkZCU1pOzj@mit.edu/. 2022.

[Wu22] David Wu. *Draft lecture notes on lattice-based cryptography*. https://www.cs.utexas.edu/dwu4/courses/sp22/static/scribe/notes.pdf. 2022.

[Zam22] Zama. *Encrypting with LWE*. https://docs.zama.ai/concrete/v/0.1/encrypting-and-decrypting/encrypting. 2022.