Groupe de travail Réseau
Request for Comments : 3320
Catégorie : Sur la voie de la normalisation
janvier 2003
Traduction Claude Brière de L'Isle

R. Price, Siemens/Roke Manor
C. Bormann, TZI/Uni Bremen
J. Christoffersson & H. Hannu, Ericsson
Z. Liu, Nokia
J. Rosenberg, dynamicsoft

Compression de signalisation (SigComp)

Statut de ce mémoire
Le présent document spécifie un protocole Internet en cours de normalisation pour la communauté de l'Internet, et appelle à des discussions et des suggestions pour son amélioration. Prière de se reporter à l'édition actuelle du STD 1 "Normes des protocoles officiels de l'Internet" pour connaître l'état de normalisation et le statut de ce protocole. La distribution du présent mémoire n'est soumise à aucune restriction.

Résumé
Le présent document définit la compression de signalisation (SigComp, *Signaling Compression*) une solution pour compresser les messages générés par les protocoles d'application tels que le protocole d'initialisation de session (SIP, *Session Initiation Protocol*) [RFC 3261] et le protocole de flux directs en temps réel (RTSP, *Real Time Streaming Protocol*) [RFC 2326]. L'architecture et les prérequis de SigComp sont précisés, ainsi que le format du message SigComp.

Une fonction de décompression est fournie pour SigComp par une machine virtuelle de décompresseur universel (UDVM, *Universal Decompressor Virtual Machine*) optimisée pour la tâche d'application des algorithmes de décompression. L'UDVM peut être configurée pour comprendre le résultat de nombreux compresseurs bien connus tels que DEFLATE [RFC1951].

Table des matières

1. Introduction

De nombreux protocoles d'application utilisés pour les communications multimédia sont fondés sur le texte et conçus pour des liaisons à large bande passante. Il en résulte que les messages n'ont pas été optimisés en termes de taille. Par exemple, le message SIP typique va de quelques centaines d'octets à deux mille octets ou plus [RFC3261].

Dans l'utilisation prévue de ces protocoles dans les téléphones sans fil au titre des réseaux cellulaires de la 2,5G et de la 3G, les grandes tailles de message sont problématiques. Avec la connexité IP à faible débit, les délais de transmission sont significatifs. En prenant en compte les retransmissions, et le grand nombre de messages qui sont requis dans certains flux, l'établissement d'appel et l'invocation des caractéristiques sont gravement affectés. SigComp apporte un moyen d'éliminer ce problème en offrant une compression robuste, sans perte des messages d'application.

Le présent document précise l'architecture et les prérequis de la solution SigComp, le format du message SigComp et de la machine virtuelle de décompression universelle (UDVM) qui fournit la fonction de décompression.

SigComp est offert aux applications comme une couche entre l'application et le transport sous-jacent. Le service fourni est celui du transport sous-jacent plus la compression. SigComp prend en charge une large gamme de transports incluant TCP, UDP et SCTP [RFC2960].

2. Terminologie

Dans le présent document, les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", et "FACULTATIF" sont à interpréter comme décrit dans le BCP 14, [RFC2119].

Application : entité qui invoque SigComp et effectue les tâches suivantes :
1. fournir les messages d'application à l'expéditeur compresseur,
2. recevoir les messages décompressés de l'expéditeur décompresseur,
3. déterminer l'identifiant de compartiment pour un message décompressé.

Code d'octet *(Bytecode)* : code machine qui peut être exécuté par une machine virtuelle.

Compresseur : entité qui code les messages d'application en utilisant un certain algorithme de compression, et garde trace de l'état qui peut être utilisé pour la compression. Le compresseur est chargé de s'assurer que les messages qu'il génère peuvent être décompressés par l'UDVM distante.

Compresseur expéditeur : entité qui reçoit les messages d'application, invoque un compresseur, et transmet les message SigComp compressés résultants à un point d'extrémité distant.

Cycles UDVM : mesure de la quantité de "puissance de CPU" exigée pour exécuter une instruction d'UDVM (les plus simples instructions d'UDVM exigent un seul cycle d'UDVM). Une limite supérieure est placée au nombre de cycles d'UDVM qui peuvent être utilisés pour décompresser chaque bit dans un message SigComp.

Décompresseur expéditeur : entité qui reçoit les messages SigComp, invoque une UDVM, et transmet les messages décompressés résultants à l'application.

Point d'extrémité : instance d'une application, d'une couche SigComp, et d'une couche transport pour envoyer et/ou recevoir les messages SigComp.

Transport fondé sur le message : transport qui porte des données comme un ensemble de messages liés.

Compartiment : groupement spécifique d'une application de messages qui se rapportent à un point d'extrémité homologue. Selon le protocole de signalisation, ce groupement peut se rapporter à des concepts d'application tels que des "sessions", "dialogues", "connexions", ou "associations". L'application alloue la mémoire d'état compartiment par compartiment, et détermine quand un compartiment devrait être créé ou fermé.

Identifiant de compartiment : identifiant (dans un format choisi en local) qui référence de façon univoque un compartiment.

SigComp : solution globale de compression, comprenant le compresseur, une UDVM, les expéditeurs ; et un gardien d'état.

Message SigComp : message envoyé du compresseur expéditeur au décompresseur expéditeur. Dans le cas d'un transport fondé sur le message tel que UDP, un message SigComp correspond à exactement un datagramme. Pour un transport fondé sur le flux comme TCP, les messages SigComp sont séparés par des délimiteurs réservés.

Transport fondé sur le flux : transport qui porte des données dans un flux continu sans limite de message.

Transport : mécanisme pour passer les données entre deux points d'extrémité. SigComp est capable d'envoyer des messages sur une large gamme de transports incluant TCP, UDP et SCTP [RFC2960].

Machine virtuelle de décompresseur universel (UDVM, *Universal Decompressor Virtual Machine*) : architecture de machine décrite dans le présent document. L'UDVM est utilisée pour décompresser les messages SigComp.

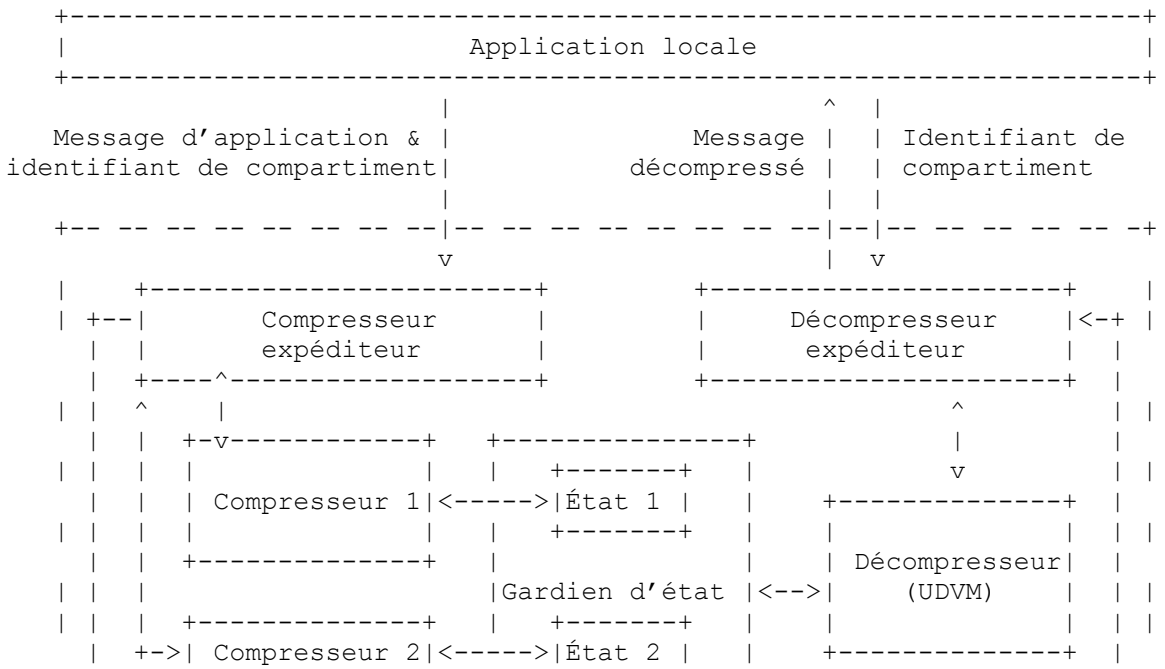
État : données sauvegardées pour être restituées par des messages SigComp ultérieurs.

Gardien d'état : entité chargée d'accéder et de mémoriser les informations d'état une fois que la permission est accordée par l'application.

Identifiant d'état : référence utilisée pour accéder à un élément d'état créé précédemment.

3. Architecture de SigComp

Dans l'architecture SigComp, la compression et la décompression sont effectuées à deux points d'extrémité communicants. La disposition d'un seul point d'extrémité est illustrée par la Figure 1 :



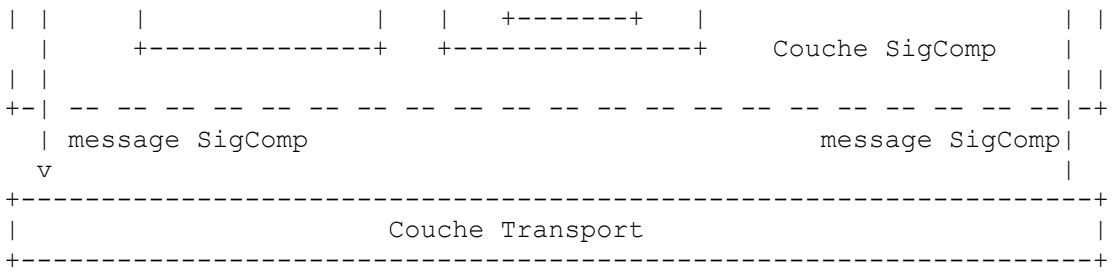


Figure 1 : Vue architecturale d'ensemble d'un point d'extrémité SigComp

Noter que SigComp est offert aux applications comme une couche entre l'application et le transport sous-jacent, et ainsi la Figure 1 est un point d'extrémité lorsque on la regarde du point de vue de la couche transport. Du point de vue des protocoles de couche d'application multi-bonds, SigComp est cependant appliqué bond par bond.

La couche SigComp est encore décomposée en les entités suivantes :

1. Compresseur expéditeur – interface pour l'application. L'application fournit au compresseur expéditeur un message d'application et un identifiant de compartiment (voir les détails au paragraphe 3.1). Le compresseur expéditeur invoque un compresseur particulier, qui retourne un message SigComp à transmettre au point d'extrémité distant.

2. Décompresseur expéditeur – interface de l'application. Le décompresseur expéditeur reçoit un message SigComp et invoque une instance de machine universelle de décompresseur virtuel (UDVM). Il transmet alors le message décompressé résultant à l'application, qui peut retourner un identifiant de compartiment si elle souhaite permettre que l'état du message soit sauvegardé.

3. Un ou plusieurs compresseurs – entités qui convertissent les messages d'application en messages SigComp. Des compresseurs distincts sont invoqués compartiment par compartiment, en utilisant les identifiants de compartiment fournis par l'application. Un compresseur reçoit un message d'application du compresseur expéditeur, compresse le message, et retourne un message SigComp au compresseur expéditeur. Chaque compresseur choisit un certain algorithme pour coder les données (par exemple, DEFLATE).

4. UDVM – entité qui décompresse les messages SigComp. Noter que comme SigComp peut fonctionner sur une couche de transport non sécurisée, une instance distincte de l'UDVM est invoquée message par message. Cependant, durant le processus de décompression, l'UDVM peut invoquer le gardien d'état pour accéder à l'état existant ou créer un nouvel état.

5. Gardien d'état – entité qui peut mémoriser et restituer l'état. L'état est l'ensemble d'informations qui est mémorisé entre les messages SigComp, évitant le besoin de charger les données message par message. Pour des raisons de sécurité, il n'est possible de créer un nouvel état qu'avec la permission de l'application. La création et la restitution d'état sont décrites plus en détails à la Section 6.

Lors de la compression d'un protocole d'application bidirectionnel, le choix d'utiliser SigComp peut être fait de façon indépendante dans les deux directions, et la compression dans une direction n'implique pas nécessairement la compression dans la direction inverse. De plus, même lorsque deux points d'extrémité communicants envoient des messages SigComp dans les deux directions, il n'est pas nécessaire d'utiliser le même algorithme de compression dans chaque direction.

Noter qu'un point d'extrémité SigComp peut décompresser des messages provenant de multiples points d'extrémité distants de différentes localisations d'un réseau, car l'architecture est conçue de façon à empêcher les messages SigComp provenant d'un point d'extrémité d'interférer avec les messages provenant d'un point d'extrémité différent. Une conséquence de ce choix de conception est qu'il est difficile à un utilisateur malveillant d'interrompre le fonctionnement de SigComp en insérant de faux messages compressés dans la couche transport.

3.1 Exigences pour l'application

Du point de vue d'une application, la couche SigComp apparaît comme un nouveau transport, avec un comportement similaire à celui du transport original utilisé pour les données non compressées (par exemple SigComp/UDP se comporte comme de l'UDP natif).

Les mécanismes pour découvrir si un point d'extrémité prend en charge SigComp sortent du domaine d'application du présent document.

Tous les messages SigComp contiennent un préfixe (les cinq bits de poids fort du premier octet sont réglés à un) qui ne se produit pas dans les messages de texte codés en UTF-8 [RFC2279], de sorte que pour les applications qui utilisent ce codage (ou le codage ASCII) il est possible de multiplexer les messages d'application non compressés et les messages SigComp sur le même accès. Les applications peuvent encore réserver un nouvel accès spécifique pour SigComp (par exemple, au titre du mécanisme de découverte).

Si un point d'extrémité particulier souhaite être à états pleins, il doit alors partager ses messages décompressés en "compartiments" dans lesquels l'état peut être sauvegardé. SigComp s'appuie sur l'application pour assurer ce partage. De sorte que pour les points d'extrémité à états pleins, une nouvelle interface est requise avec l'application afin de supporter les mécanismes d'authentification utilisés par l'application elle-même.

Lorsque l'application reçoit un message décompressé, elle transpose le message dans un certain compartiment et fournit l'identifiant de compartiment à SigComp. Chaque compartiment est alloué à un compresseur distinct ainsi qu'une certaine quantité de mémoire pour mémoriser les informations d'état, de sorte que l'application doit allouer des compartiments distincts aux différents points d'extrémité distants. Cependant, il est possible à un point d'extrémité local d'établir plusieurs compartiments qui se rapportent au même point d'extrémité distant (cela devrait être évité chaque fois que possible car c'est un gaspillage de mémoire et cela réduit le taux de compression global, mais cela ne cause pas une décompression incorrecte des messages). Dans ce cas, un fonctionnement fiable à états pleins n'est possible que si le décompresseur ne fourre pas plusieurs messages dans un seul compartiment alors que le compresseur s'attend à ce qu'ils soient alloués à des compartiments différents.

Le format exact de l'identifiant de compartiment est sans importance pourvu que des identifiants différents soient donnés aux différents compartiments.

Les applications qui souhaitent communiquer en utilisant SigComp à états pleins devraient utiliser un mécanisme d'authentification pour transposer en toute sécurité les messages décompressés en identifiants de compartiment. Elles devraient aussi se mettre d'accord sur des limites à la durée de vie d'un compartiment, pour éviter le cas où un point d'extrémité accède à des informations d'état qui ont déjà été supprimées.

3.2 Mécanisme de rétroaction de SigComp

Si un protocole de signalisation envoie des messages SigComp dans les deux directions et si il y a une relation biunivoque entre les compartiments établis par les applications des deux côtés ("compartiments homologues") les deux points d'extrémité peuvent coopérer plus étroitement. Dans ce cas, il est possible d'envoyer des informations de rétroaction qui surveillent le comportement d'un point d'extrémité et aident à améliorer le taux de compression global.

SigComp effectue les rétroaction s sur la base de la question/réponse, de sorte qu'un compresseur fait une demande de rétroaction et reçoit des données de rétroaction en réponse. La procédure de demande et de réponse d'informations de rétroaction dans SigComp est illustrée à la Figure 2 :

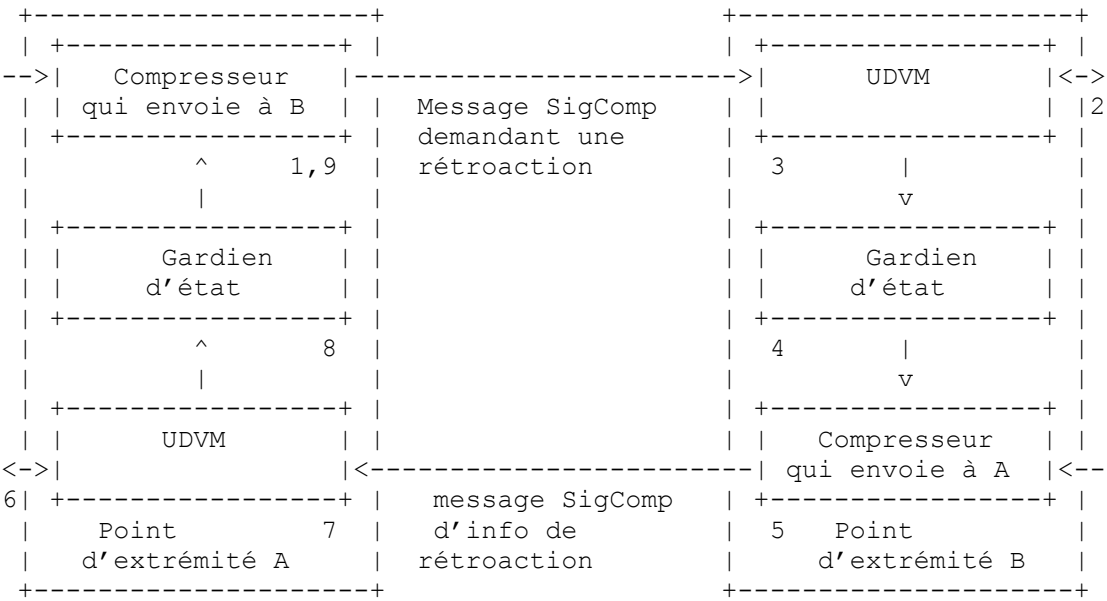


Figure 2 : Étapes impliquées dans la transmission de données de rétroaction

Les expéditeurs, la couche application et la couche transport sont omis du diagramme pour qu'il soit plus clair. Noter que les messages décompressés passent via le décompresseur expéditeur à l'application ; de plus, le messages SigComp transmis du compresseur à l'UDVM distante sont d'abord envoyés via le compresseur expéditeur, suivi par la couche transport et finalement le décompresseur expéditeur.

Les étapes pour demander et renvoyer les données de rétroaction sont décrites plus en détail ci-dessous :

1. Le compresseur qui envoie les messages au point d'extrémité B porte une demande de rétroaction sur un message SigComp.
2. Lorsque l'application reçoit le message décompressé, elle peut retourner l'identifiant de compartiment pour le message.
3. L'UDVM du point d'extrémité B transmet les données de rétroaction demandées au gardien d'état.
4. Si l'UDVM peut fournir un identifiant de compartiment valide, le gardien d'état transmet alors les données de rétroaction au compresseur approprié (à savoir le compresseur qui envoie au point d'extrémité A).
5. Le compresseur renvoie les données de rétroaction demandées au point d'extrémité A portées sur un message SigComp.
6. Lorsque l'application reçoit le message décompressé, elle peut retourner l'identifiant de compartiment pour le message.
7. L'UDVM du point d'extrémité A transmet les données de rétroaction au gardien d'état.
8. Si l'UDVM peut fournir un identifiant de compartiment valide, le gardien d'état transmet alors les données de rétroaction au compresseur approprié (à savoir le compresseur qui envoie au point d'extrémité B).
9. Le compresseur utilise les données de rétroaction envoyées.

Le rôle détaillé joué par chaque entité dans la transmission des données de rétroaction est expliqué dans les sections suivantes.

3.3 Paramètres de SigComp

L'utilisation d'une machine virtuelle pour la décompression présente l'avantage que presque toute la souplesse de la mise en œuvre réside dans les compresseurs SigComp. Lorsque il reçoit des messages SigComp, un point d'extrémité se comporte généralement d'une manière prévisible.

Noter cependant que les points d'extrémité qui mettent en œuvre SigComp vont normalement avoir une large gamme de capacités, dont chacune offre une quantité différente de mémoire de travail, de puissance de traitement, etc.. Afin de prendre en charge cette grande diversité des capacités des points d'extrémité, les paramètres suivants sont fournis pour modifier le comportement de SigComp lors de la réception des messages SigComp :

Taille_de_mémoire_de_décompression
Taille_de_mémoire_d'état
Cycles_par_bit
Version_SigComp
État_disponible_en_local (ensemble contenant 0, 1 ou plusieurs éléments d'état)

Chaque paramètre a une valeur minimum qui DOIT être offerte par tous les points d'extrémité SigComp receveurs. De plus, les points d'extrémité PEUVENT offrir des ressources supplémentaires si il en est de disponibles ; ces ressources peuvent être annoncées aux points d'extrémité distants en utilisant le mécanisme de rétroaction SigComp.

Des applications particulières peuvent aussi s'accorder à priori pour offrir des ressources supplémentaires obligatoires (par exemple, SigComp pour SIP offre un dictionnaire de phrases SIP communes comme élément d'état obligatoire).

Chacun des paramètres SigComp est décrit plus en détails ci-dessous.

3.3.1 Taille de mémoire et cycles UDVM

Le paramètre Taille_de_mémoire_de_décompression spécifie la quantité de mémoire disponible pour décompresser un message SigComp. (Noter que le terme "quantité de mémoire" est utilisé à un niveau conceptuel afin de spécifier le comportement du décompresseur et permettre une planification des ressources du côté du compresseur – une mise en œuvre pourrait exiger des quantités supplémentaires, limitées, de ressources réelles de mémoire, ou pourrait même organiser sa mémoire d'une façon complètement différente pour autant que cela ne cause pas d'échec de la décompression alors que le modèle conceptuel ne le ferait pas. Une portion de cette mémoire est utilisée pour mettre en mémoire tampon un message SigComp avant qu'il soit décompressé ; le reste est donné à l'UDVM. Noter que la mémoire est allouée sur la base du message et peut être reprise après que le message a été décompressé. Tous les points d'extrémité qui mettent en œuvre SigComp DOIVENT offrir une taille de mémoire de décompression d'au moins 2048 octets.

Le paramètre Taille_de_mémoire_d'état spécifie le nombre d'octets offerts à un compartiment particulier pour la création d'état. Ce paramètre est réglé à 0 si le point d'extrémité est sans état.

À la différence des autres paramètres SigComp, la Taille_de_mémoire_d'état est offerte par compartiment et peut varier pour différents compartiments. La mémoire pour un compartiment est reprise lorsque l'application détermine que le compartiment n'est plus nécessaire.

Le paramètre Cycles_par_bit spécifie le nombre de "cycles d'UDVM" disponibles pour décompresser chaque bit dans un message SigComp. Exécuter une instruction d'UDVM exige un certain nombre de cycles d'UDVM ; une liste complète d'instructions d'UDVM et leur coût en cycles d'UDVM se trouve à la Section 9. Un point d'extrémité DOIT offrir un minimum de 16 cycles par bit.

Chacune des valeurs des trois paramètres DOIT être choisie dans un ensemble limité donné ci-dessous, afin que les paramètres puissent être codés efficacement pour la transmission en utilisant le mécanisme de rétroaction de SigComp.

Le paramètre Cycles_par_bit est codé en utilisant 2 bits, tandis que Taille_mémoire_de_décompression et Taille_de_mémoire_d'état sont tous deux codés en utilisant 3 bits. Le codage des bits et leurs valeurs correspondantes sont les suivants :

Codage :	**Cycles_par_bit:**	**Codage :**	**Taille_de_mémoire_d'état (octets) :**
00	16	000	0
01	32	00	2048
10	64	010	4096
11	128	011	8192
		100	16384
		101	32768
		110	65536
		111	131072

La Taille_de_mémoire_de_décompression est codée de la même manière que la Taille_de_mémoire_d'état, excepté que le schéma binaire 000 ne peut pas être utilisé (car un point d'extrémité ne peut pas offrir une taille de mémoire de décompression de 0 octet).

3.3.2 Version SigComp

Le paramètre Version_SigComp spécifie si seule la version de base de SigComp est disponible, ou si est disponible une version mise à niveau qui offre des instructions supplémentaires, etc. Au sein de l'UDVM, il est disponible comme valeur de 2 octets, générée en étendant avec un zéro le paramètre de 1 octet de Version_SigComp (c'est-à-dire que le premier octet de la valeur de deux octets est toujours zéro). La version de base de SigComp est Version 0x01, qui est la version décrite dans le présent document.

Pour assurer la rétro compatibilité, si un message SigComp est décompressé avec succès par la version 0x01 de SigComp il sera alors bien décompressé par les versions améliorées. De même, si le message déclenche un échec de décompression manuelle (voir le paragraphe 8.7) il va aussi continuer de le faire.

Cependant, les messages qui causent un échec inattendu de décompression sur la version 0x01 de SigComp peuvent être décompressés avec succès par des versions améliorées.

La plus simple façon d'améliorer SigComp d'une manière rétro compatible est d'ajouter des instructions d'UDVM supplémentaires car cela ne va pas affecter la décompression des messages SigComp compatibles avec la version 0x01. Des adresses réservées dans la mémoire de l'UDVM (valeurs utiles, voir le paragraphe 7.2) pourront aussi être des valeurs allouées dans de futures versions de SigComp.

3.3.3 Éléments d'état disponibles localement

Un élément d'état SigComp est un élément de données qui est conservé entre les messages SigComp. Les éléments d'état peuvent être restitués et chargés dans la mémoire de l'UDVM au titre du processus de décompression, améliorant souvent de façon significative le taux de compression car les mêmes informations n'ont pas à être chargées message par message.
Chaque point d'extrémité conserve un ensemble d'éléments d'état où chaque élément est composé des informations suivantes :

Nom : **Type de données :**

state_identifier	valeur de 20 octets
state_length	valeur de 2 octets
state_address	valeur de 2 octets
state_instruction	valeur de 2 octets
minimum_access_length	valeur de 2 octets de 6 à 20 inclus
state_value	chaîne de state_length octets consécutifs

Les éléments d'état sont normalement créés à un point d'extrémité à la réussite de la décompression d'un message SigComp. Le compresseur distant qui envoie le message fait une demande de création d'état en invoquant l'instruction appropriée d'UDVM, et l'état est sauvegardé une fois que la permission est accordée par l'application.

Cependant, un point d'extrémité PEUT aussi souhaiter offrir un ensemble d'éléments d'état disponibles en local qui n'ont pas été chargés au titre d'un message SigComp. Par exemple, il peut offrir des algorithmes de décompression bien connus, des dictionnaires de phrases courantes utilisées dans un protocole de signalisation spécifique, etc.

Comme ces éléments d'état sont établis en local sans entrée de la part d'un point d'extrémité distant, ils sont très utiles si ils sont rendus publics afin qu'une large collection de points d'extrémité distants puissent déterminer les données contenues dans chaque élément d'état et comment elles peuvent être utilisées. D'autres documents Internet et des RFC pourront être publiés pour décrire les éléments d'état particuliers disponibles localement.

Bien qu'il n'y ait pas d'élément d'état disponible localement qui soit obligatoire pour chaque point d'extrémité SigComp, certains éléments d'état peuvent être rendus obligatoires dans un environnement spécifique (par exemple, le dictionnaire des phrases courantes pour un protocole de signalisation spécifique pourrait être rendu obligatoire pour l'utilisation de SigComp par ce protocole de signalisation). Aussi, des points d'extrémité distants peuvent indiquer leur intérêt à recevoir une liste de certains des éléments d'état disponibles en local à un point d'extrémité en utilisant le mécanisme de rétroaction SigComp.

C'est une décision locale d'un point d'extrémité de choisir quels éléments d'état disponible localement il annonce ; cette décision n'a pas d'influence sur l'interopérabilité, mais peut accroître ou diminuer l'efficacité de la compression réalisable entre les points d'extrémité.

4. Expéditeurs SigComp

Cette section définit le comportement des expéditeurs compresseur et décompresseur. La fonction de ces entités est de fournir une interface entre SigComp et son environnement, minimisant l'effort nécessaire pour intégrer SigComp dans une pile de protocoles existante.

4.1 Compresseur expéditeur

Le compresseur expéditeur reçoit les messages de l'application et passe la version compressée de chaque message à la couche transport.

Noter que SigComp invoque les compresseurs par compartiment, de sorte que lorsque l'application fournit un message à compresser, elle doit aussi fournir un identifiant de compartiment. Le compresseur expéditeur transmet le message d'application au bon compresseur sur la base de l'identifiant de compartiment (invoquant un nouveau compresseur si un nouvel identifiant de compartiment est rencontré). Le compresseur retourne un message SigComp qui peut être passé à la couche transport.

De plus, l'application devrait indiquer au compresseur expéditeur lorsque il souhaite clore un certain compartiment, afin que les ressources prises par le compresseur correspondant puissent être réclamées.

4.2 Décompresseur expéditeur

Le décompresseur expéditeur reçoit les messages de la couche transport et passe la version décompressée de chaque message à l'application.

Pour s'assurer que SigComp peut fonctionner sur une couche de transport non sécurisée, le décompresseur expéditeur invoque une nouvelle instance de l'UDVM pour chaque nouveau message SigComp. Les ressources pour l'UDVM sont libérées aussitôt que le message a été décompressé.

L'expéditeur NE DOIT PAS rendre plus d'un message SigComp disponible à une même instance de l'UDVM. En particulier, l'expéditeur NE DOIT PAS enchaîner deux messages SigComp pour former un seul message.

4.2.1 Stratégies de décompresseur expéditeur

Une fois que l'UDVM a été invoqué, il est initialisé en utilisant le message SigComp de la Section 7. Le message est alors décompressé par l'UDVM, retourné au décompresseur expéditeur, et passé à l'application receveuse. Noter que l'UDVM ne sait absolument pas si le transport sous-jacent est fondé sur le message ou si il est fondé sur le flux, et donc il sort toujours les données décompressées comme un flux. Il est de la responsabilité de l'expéditeur de fournir le message décompressé à l'application dans le format attendu (c'est-à-dire, comme un flux ou comme un message distinct, avec ses limites). L'expéditeur sait que la fin d'un message décompressé a été atteinte lorsque l'instruction d'UDVM END-MESSAGE est invoquée (voir au paragraphe 9.4.9).

Pour un transport fondé sur le flux, deux stratégies sont donc possibles pour le décompresseur expéditeur :

1) L'expéditeur collecte un message SigComp complet et invoque ensuite l'UDVM. L'avantage est que, même dans les mises en œuvre qui ont plusieurs flux compressés entrants, seule une instance de l'UDVM est nécessaire.

2) L'expéditeur collecte l'en-tête SigComp (voir la Section 7) et invoque l'UDVM ; l'UDVM reste active pendant que le reste du message arrive. L'avantage est qu'il n'est pas nécessaire de mettre en mémoire tampon le reste du message ; le message peut être décompressé lorsque il arrive, et tout résultat de décompression peut être immédiatement relayé à l'application.

En général, le choix de la stratégie utilisée appartient à la mise en œuvre. Cependant, le compresseur peut vouloir tirer parti de la seconde stratégie en s'attendant à ce qu'une partie du message d'application soit passée à l'application avant que le message SigComp soit terminé, par exemple, en gardant l'UDVM active tout en s'attendant à ce que l'application continue de recevoir des résultats décompressés. Cette approche ("mode continu") est contraire à certaines hypothèses du modèle de sécurité de SigComp et ne peut être utilisée que si le transport lui-même peut fournir la protection requise contre les attaques de déni de service. De plus, comme seule la stratégie 2 fonctionne dans cette approche, l'utilisation du mode continu exige un accord préalable entre les deux points d'extrémité.

4.2.2 Marquage d'enregistrement

Pour un transport fondé sur le flux, l'expéditeur délimite les messages en analysant le flux de données compressées à la recherche des instances de 0xFF et en effectuant les actions suivantes :

Vu dans le flux de données :	**Action:**
0xFF 00	un octet 0xFF dans le flux de données
0xFF 01	idem, mais l'octet suivant est entre guillemets (pourrait être un autre 0xFF)
:	:
0xFF 7F	idem, mais les 127 octets suivants sont entre guillemets
0xFF 80 à 0xFF FE	(réservé pour future normalisation)
0xFF FF	fin de message SigComp

Les combinaisons 0xFF01 à 0xFF7F sont utiles pour limiter le pire cas d'expansion du schéma de marquage d'enregistrement : les octets 1 (0xFF01) à 127 (0xFF7F) qui suivent la combinaison d'octets sont copiés littéralement par le décompresseur sans effectuer d'action particulière sur 0xFF. (Noter que 0xFF00 est juste un cas particulier de cela, où zéro octet suivant est copié littéralement.)

Dans UDVM version 0x01, toute occurrence des combinaisons 0xFF80 à 0xFFFE qui ne sont pas protégées par des guillemets cause l'échec de la décompression ; le décompresseur DEVRAIT dans ce cas clore le transport fondé sur le flux.

4.3 Retour d'un identifiant de compartiment

À réception d'un message décompressé, l'application peut fournir à l'expéditeur un identifiant de compartiment. Fournir cet identifiant donne la permission de ce qui suit :

1. Les éléments d'état accompagnant le message décompressé peuvent être sauvegardés en utilisant la mémoire d'état réservée pour le compartiment spécifié.
2. Les données de rétroaction qui accompagnent le message décompressé peuvent être réputées suffisamment fiables pour qu'elles puissent être utilisées lors de l'envoi de messages SigComp qui se rapportent à l'équivalent du compresseur pour le compartiment.

L'expéditeur passe l'identifiant de compartiment à l'UDVM, où il est utilisé selon l'instruction END-MESSAGE (voir le paragraphe 9.4.9).

L'application utilise un mécanisme d'authentification convenable pour déterminer si le message décompressé appartient à un compartiment légitime ou non. Si l'application échoue à authentifier le message avec une confiance suffisante pour permettre que l'état soit sauvegardé ou que les données de rétroaction soient de confiance, elle fournit une erreur "pas de compartiment valide" à l'expéditeur et l'UDVM se termine sans créer d'état ni transmettre de données de rétroaction.

5. Compresseur SigComp

Une caractéristique importante de SigComp est que la fonction de décompression est fournie par une machine virtuelle de décompression universelle (UDVM). Cela signifie que le compresseur peut choisir tout algorithme pour générer les messages compressés SigComp, et charger ensuite le code d'octet pour l'algorithme de décompression correspondant à l'UDVM au titre du message SigComp.

Pour aider à la mise en œuvre et à l'essai d'un point d'extrémité SigComp, d'autres documents et RFC Internet pourront être publiés pour décrire des algorithmes de compression particuliers.

L'exigence globale qui pèse sur le compresseur est celle de la transparence, c'est-à-dire que le compresseur NE DOIT PAS envoyer de code d'octet qui cause la décompression incorrecte d'un message SigComp par l'UDVM.

Les exigences plus spécifiques suivantes concernent aussi le compresseur (elles peuvent être considérées comme des instances particulières de l'exigence de transparence) :

1. Pour la robustesse, il est recommandé que le compresseur fournisse une forme de vérification d'intégrité (pas nécessairement de force cryptographique) sur le message d'application pour s'assurer du succès de la décompression. Une instruction d'UDVM est fournie pour la vérification du CRC ; aussi, une autre instruction peut être utilisée pour calculer un hachage cryptographique SHA-1.
2. Le compresseur DOIT s'assurer que le message peut être décompressé en utilisant les ressources disponibles au point d'extrémité distant.
3. Si le transport est fondé sur le message, alors le compresseur DOIT transposer chaque message d'application en exactement un message SigComp.
4. Si le transport est fondé sur le flux mais si l'application définit ses propres limites internes de message, le compresseur DEVRAIT alors transposer chaque message d'application en exactement un message SigComp.

Les limites de message devraient être conservées sur un transport fondé sur le flux afin que des dommages accidentels ou malveillants à un message SigComp n'affectent pas la décompression des messages suivants.

De plus, si le gardien d'état passe au compresseur des rétroactions demandées, elles DEVRAIENT alors être retournées dans le prochain message SigComp généré par le compresseur (sauf si le gardien d'état passe des rétroactions demandées depuis avant que les plus anciennes rétroactions ont été envoyées, auquel cas la plus ancienne rétroaction est supprimée).

S'il est présent, l'élément de rétroaction demandé DEVRAIT être copié sans modification dans le champ Élément_de_rétroaction_renvoyé fourni dans le message SigComp. Noter qu'il n'est pas nécessaire de transmettre plus d'une fois tout élément de rétroaction demandé.

Le compresseur DEVRAIT aussi charger les paramètres SigComp locaux chez le point d'extrémité distant, sauf si le point d'extrémité a indiqué qu'il ne souhaite pas recevoir ces paramètres ou si le compresseur détermine que les paramètres ont déjà été fournis avec succès (voir au paragraphe 5.1 les détails de la façon dont ce peut être réalisé). Les paramètres SigComp sont téléchargés dans la mémoire de l'UDVM au point d'extrémité distant comme décrit au paragraphe 9.4.9.

5.1 Assurer le succès de la décompression

Un compresseur DOIT être certain que toutes les données nécessaires pour décompresser un message SigComp sont disponibles au point d'extrémité receveur. Une façon de s'assurer de cela est d'envoyer toutes les informations nécessaires dans chaque message SigComp (y compris le code d'octet pour décompresser le message). Cependant, le taux de compression pour cette méthode sera relativement faible.

Pour obtenir le meilleur taux de compression global, le compresseur doit demander la création de nouveaux éléments d'état au point d'extrémité distant. Les informations sauvegardées dans ces éléments d'état peuvent alors être atteintes par les messages SigComp ultérieurs, évitant qu'il soit besoin de télécharger les données message par message.

Avant que le compresseur puisse accéder à l'état sauvegardé, il doit cependant s'assurer que le message SigComp qui porte la demande de création d'état est bien arrivé au point d'extrémité receveur. Pour un transport fiable (par exemple, TCP ou SCTP) ceci est garanti. Cependant, pour un transport non fiable, le compresseur doit lui-même fournir un mécanisme convenable (voir les détails dans la [RFC3321]).

Le compresseur doit aussi s'assurer que l'élément d'état auquel il souhaite accéder n'a pas été rejeté à cause d'un manque de mémoire d'état. Cela peut se faire en vérifiant le paramètre Taille_de_mémoire_d'état en utilisant le mécanisme de rétroaction SigComp (voir les détails au paragraphe 9.4.9).

5.2 Échec de compression

Le compresseur DEVRAIT faire tous les efforts nécessaires pour réussir à compresser un message d'application, mais dans certains cas, cela peut n'être pas possible (en particulier si les ressources sont rares au point d'extrémité receveur). Dans ce cas, on invoque un "échec de compression".

Si un échec de compression survient, le compresseur informe alors l'expéditeur et n'effectue pas d'autre action. L'expéditeur DOIT rapporter cet échec à l'application afin qu'elle puisse essayer d'autres méthodes pour livrer le message.

6. Traitement d'état et de rétroaction

Cette section définit le comportement du gardien d'état SigComp. La fonction du gardien d'état est de conserver les informations entre les messages SigComp reçus ; c'est la seule entité SigComp qui est capable d'assurer cette fonction, et il est donc particulièrement important du point de vue de la sécurité.

6.1 Création et accès d'état

Pour assurer la sécurité contre l'insertion malveillante ou la modification de messages SigComp, une instance distincte de l'UDVM est invoquée pour décompresser chaque message. Cela assure que les messages SigComp endommagés n'empêchent pas la réussite de la décompression des messages valides suivants.

Noter, cependant, que le taux global de compression est souvent significativement supérieur si les messages peuvent être compressés par rapport aux informations contenues dans les messages précédents. Pour cette raison, il est possible de créer des éléments d'état auxquels accéder lorsque un message ultérieur est décompressé. La création et l'accès à un état sont tous deux conçus comme sécurisés contre une altération malveillante des données compressées. L'UDVM peut seulement créer un élément d'état lorsque un message complet a été décompressé avec succès et que l'application a retourné un identifiant de compartiment sous lequel l'état peut être sauvegardé.

L'accès à l'état ne peut pas être protégé en s'appuyant seulement sur l'application, car le mécanisme d'authentification peut exiger des informations venant du message décompressé (qui bien sûr n'est pas disponible tant qu'on a pas eu accès à l'état). SigComp protège plutôt l'accès à l'état en créant un identifiant d'état qui est un hachage sur l'élément d'état à restituer. Cet Identifiant_d'état doit être fourni pour restituer un élément d'état à partir du gardien d'état.

Noter aussi que l'état n'est pas supprimé lorsqu'on y accède. De sorte que même si un envoyeur malveillant s'arrange pour accéder à des informations d'état, les messages compressés suivants relatifs à cet état peuvent encore être bien décompressés.

Chaque élément d'état contient un Identifiant_d'état qui est utilisé pour accéder à l'état. Un identifiant d'état peut être fourni dans l'en-tête du message SigComp pour initialiser l'UDVM (voir la Section 7) ; des éléments d'état supplémentaires peuvent être restitués en utilisant l'instruction STATE-ACCESS. L'UDVM peut aussi demander la création d'un nouvel élément d'état en utilisant les instructions STATE-CREATE et END-MESSAGE (voir les détails à la Section 9).

6.2 Gestion de mémoire

Le gardien d'état gère la mémoire d'état sur la base du compartiment. Chaque compartiment peut mémoriser l'état jusqu'à une certaine Taille_de_mémoire_d'état (où l'application peut allouer des valeurs différentes au paramètre Taille_de_mémoire_d'état de différents compartiments).

En plus de la mémorisation des éléments d'état eux-mêmes, le gardien d'état tient une liste des éléments d'état créés par chaque compartiment et s'assure qu'aucun compartiment n'excède la Taille_de_mémoire_d'état qui lui est allouée. Pour les besoins du calcul, chaque élément d'état est considéré coûter (Longueur_d'état + 64) octets.

Chaque instance de l'UDVM peut passer jusqu'à quatre demandes de création d'état au gardien d'état, ainsi que jusqu'à quatre demandes libres d'état (ces dernières sont des demandes de libération de la mémoire prise par un élément d'état dans un certain compartiment). Lorsque le gardien d'état reçoit une demande de création d'état de la part de l'UDVM, il suit les étapes suivantes :

1. Le gardien d'état DOIT rejeter toutes les demandes de création d'état qui ne sont pas accompagnées par un identifiant de compartiment valide, ou si il est alloué au compartiment 0 octets de mémoire d'état. Noter que si une demande de création d'état échoue à cause d'un manque de mémoire d'état, cela ne signifie alors pas que le message SigComp correspondant est endommagé ; les compresseurs vont souvent faire des demandes de création d'état dans le premier message SigComp d'un compartiment, avant d'avoir découvert la Taille_de_mémoire_d'état en utilisant le mécanisme de retour SigComp.

2. Si la demande de création d'état a besoin de plus de mémoire d'état que la Taille_de_mémoire_d'état totale pour le compartiment, le gardien d'état supprime tous les octets sauf le premier (Taille_de_mémoire_d'état - 64) de Valeur_d'état. Il règle Longueur_d'état à (Taille_de_mémoire_d'état - 64) et recalcule l'identifiant d'état comme défini au paragraphe 9.4.9.

3. Si la demande de création d'état contient un Identifiant_d'état qui existe déjà, le gardien d'état vérifie alors si l'élément d'état demandé est identique à l'élément d'état établi et compte la demande de création d'état comme réussie si c'est le cas. Sinon, la demande de création d'état est alors non réussie (bien que la probabilité que cela se produise soit infime).

4. Si la demande de création d'état excède la mémoire d'état allouée au compartiment, suffisamment d'éléments d'état créés par le même compartiment sont libérés jusqu'à ce qu'assez de mémoire soit disponible pour traiter le nouvel état. Lorsque un élément d'état est libéré, il est retiré de la liste des états créée par le compartiment et le coût en mémoire de l'élément d'état ne compte plus dans le coût total du compartiment. Noter, cependant, que des éléments d'état identiques peuvent être créés par plusieurs compartiments différents, et donc un élément d'état ne doit pas être supprimé physiquement jusqu'à ce que le gardien d'état détermine qu'il n'est plus exigé par aucun compartiment.

5. L'ordre dans lequel les éléments d'état existants sont libérés est déterminé par la Priorité_de_rétention_d'état, qui est obtenue lorsque les éléments d'état sont créés. La Priorité_de_rétention_d'état de 65535 est réservée pour les états disponibles en local ; ces états doivent toujours être libérés en premier. À part ce cas particulier, les états qui ont la plus faible Priorité_de_rétention_d'état sont toujours libérés en premier. Dans le cas d'égalité, c'est alors l'élément d'état créé en premier dans le compartiment qui est aussi le premier libéré.

Le paramètre Priorité_de_rétention_d'état est toujours mémorisé par compartiment au titre de la liste des éléments d'état créée par chaque compartiment. En particulier, le même élément d'état peut avoir plusieurs valeurs de priorité si il a été créé par plusieurs compartiments différents.

Noter que les éléments d'état disponibles en local (comme décrit au paragraphe 3.3.3) n'ont pas besoin d'être transposés dans un compartiment particulier. Cependant, si ils sont créés par compartiment, ils ne doivent pas interférer avec l'état créé à la demande du point d'extrémité distant. La Priorité_de_rétention_d'état spéciale de 65535 réservée pour les éléments d'état disponibles en local est destinée à s'assurer que c'est le cas.

L'UDVM peut aussi demander explicitement que le gardien d'état libère un élément d'état spécifique dans un compartiment. Dans ce cas, le gardien d'état supprime l'élément d'état de la liste des éléments d'état créée par le compartiment (comme précédemment, l'élément d'état ne doit pas être lui-même supprimé physiquement avant que le gardien d'état n'ait déterminé si il n'est plus exigé par un compartiment).

L'application devrait indiquer au gardien d'état quand elle souhaite fermer un certain compartiment, afin que les ressources prises par l'état correspondant puissent être reprises.

6.3 Données de retour

Le mécanisme de retour SigComp permet que des données de rétroaction soient reçues par une UDVM et transmises via le gardien d'état au bon compresseur.

Comme ces données de rétroaction sont conservées entre les messages SigComp, elles sont considérées comme faisant partie de l'état global et ne peuvent être transmises que si elles sont accompagnées par un identifiant de compartiment

valide. Si c'est le cas, le gardien d'état transmet alors les données de rétroaction au compresseur chargé de l'envoi des messages qui relèvent du compartiment homologue du compartiment spécifié.

7. Format de message SigComp

La présente section décrit le format du message SigComp et comment le message est utilisé pour initialiser la mémoire de l'UDVM.

Noter que le message SigComp n'est pas copié aussitôt qu'il arrive dans la mémoire de l'UDVM ; l'UDVM indique plutôt quand elle exige des données compressées en utilisant une instruction spécifique. Elle fait alors une pause et attend que les informations soient fournies avant d'exécuter l'instruction suivante. Cela signifie que l'UDVM peut commencer à décompresser un message SigComp avant que le message entier ait été reçu.

Une conséquence du comportement ci-dessus est que lorsque l'UDVM est invoquée, la taille de la mémoire UDVM dépend de ce que le transport utilisé pour fournir le message SigComp est fondé sur le flux ou sur le message. Si le transport est fondé sur le message, une mémoire suffisante doit être disponible pour mettre le message SigComp entier en mémoire tampon avant de le passer à l'UDVM. De sorte que si le message fait n octets, la taille de la mémoire de l'UDVM sera réglée à (Taille_de_mémoire_de_décompression - n) jusqu'à un maximum de 65536 octets.

Cependant, si le transport est fondé sur le flux, une taille fixe de mémoire tampon d'entrée est alors exigée pour s'accommoder du flux, indépendamment de la taille de chaque message SigComp. Donc, pour simplifier, la taille de la mémoire de l'UDVM est réglée à (Taille_de_mémoire_de_décompression / 2).

Comme une instance séparée de l'UDVM est invoquée message par message, chaque message SigComp doit indiquer explicitement l'algorithme de décompression choisi ainsi que toutes les informations supplémentaires qui sont nécessaires pour décompresser le message (par exemple, un ou plusieurs messages reçus précédemment, un dictionnaire des phrases SIP courantes, etc.). Ces informations peuvent être téléchargées au titre du message SigComp ou restituées à partir d'un élément d'état.

Un message SigComp prend une de ces deux formes selon qu'il accède à un élément d'état au point d'extrémité receveur. Les deux variantes d'un message SigComp sont données à la Figure 3. (Le bit T contrôle le format de l'élément de rétroaction renvoyé et est défini au paragraphe 7.1.)

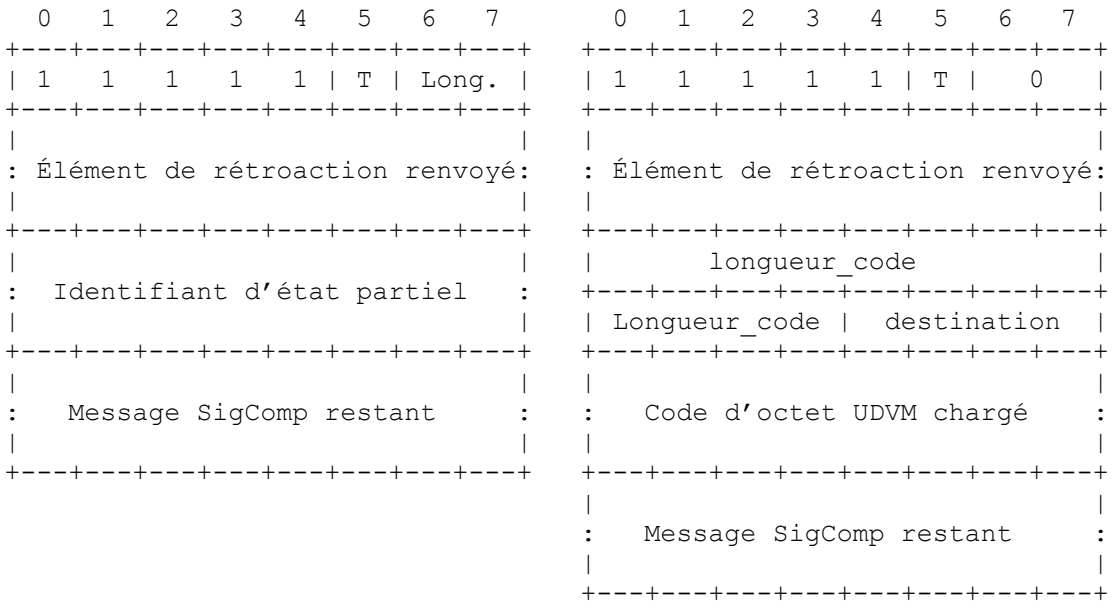


Figure 3 : Format d'un message SigComp

Un échec de décompression survient si le message SigComp est trop court pour contenir les champs attendus (voir les détails au paragraphe 8.7).

Sauf pour le champ "Message SigComp restant", on se réfère aux champs comme à "l'en-tête SigComp" (noter que cela peut inclure le code d'octet UDVM chargé).

7.1 Élément de rétroaction retourné

Pour les deux variantes du message SigComp, le bit T est réglé à 1 chaque fois que le message SigComp contient un élément de retour renvoyé. Le format de l'élément de retour renvoyé est illustré par la Figure 4.

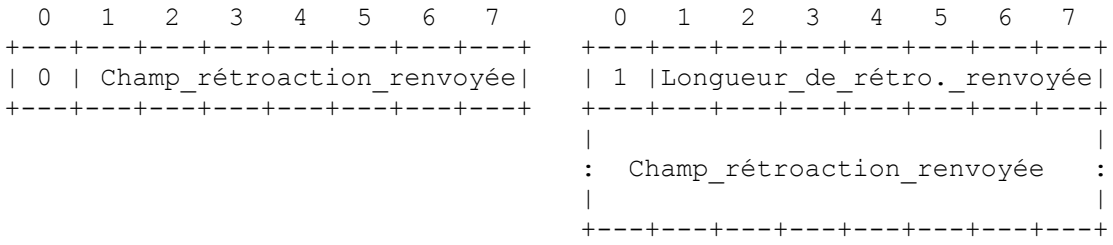


Figure 4 : Format d'un élément de rétroaction retourné

Noter que Longueur de rétroaction renvoyée spécifie la taille du champ de rétroaction renvoyée (de 0 à 127 octets). De sorte que la taille totale de l'élément de rétroaction renvoyé se tient entre 1 et 128 octets.
L'élément de rétroaction renvoyé n'est pas copié dans la mémoire de l'UDVM ; il est plutôt mis en mémoire tampon jusqu'à ce que l'UDVM ait réussi à décompresser le message SigComp. Il est alors transmis au gardien d'état avec le reste des données de rétroaction (voir les détails au paragraphe 9.4.9).

7.2 Accession à l'état mémorisé

Le champ Longueur du message SigComp détermine quels champs suivent l'éléments de rétroaction renvoyé. Si le champ Longueur est différent de zéro, le message SigComp contient alors un identifiant d'état pour accéder à un élément d'état au point d'extrémité receveur. Tous les éléments d'état incluent un identifiant d'état de 20 octets conformément au paragraphe 3.3.3, mais il est possible de ne transmettre que 6 octets de l'identifiant si l'envoyeur estime que c'est suffisant pour correspondre à un élément d'état unique chez le point d'extrémité receveur.

Le champ Longueur code le nombre d'octets transmis de la façon suivante :

Codage :	**Longueur d'un identifiant d'état partiel :**
01	6 octets
10	9 octets
11	12 octets

L'identifiant d'état partiel est passé au gardien d'état, qui le compare aux octets de poids fort de l'Identifiant_d'état dans chaque élément d'état actuellement mémorisé. Un échec de décompression survient si aucun élément d'état ne correspond ou si plus d'un élément d'état correspond. Un échec de décompression survient aussi si exactement un élément d'état correspond mais si l'élément d'état contient un paramètre Longueur_d'accès_minimum supérieur à la longueur de l'identifiant d'état partiel. Cela empêche en particulier un malveillant d'accéder à des éléments d'état sensibles par une évaluation en force brute de l'identifiant d'état. Si l'accès à un élément d'état réussit, la chaîne d'octets Valeur_d'état est copiée dans la mémoire d'UDVM en commençant par Adresse_d'état. Les 32 premiers octets de mémoire d'UDVM sont alors initialisés à des valeurs particulières comme l'illustre la Figure 5.

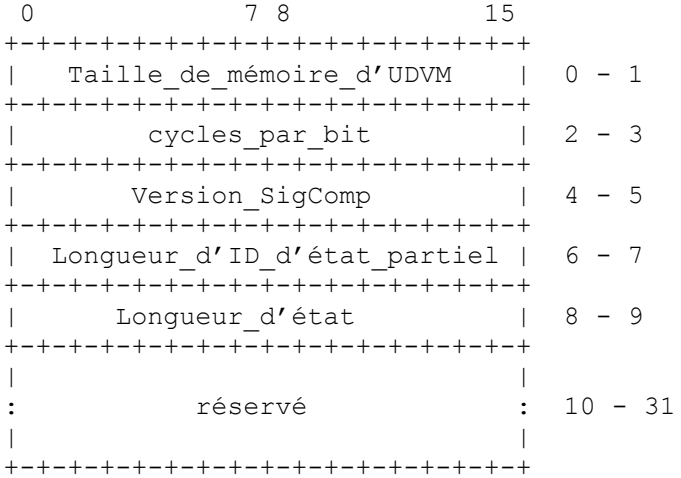


Figure 5 : Initialisation des valeurs utiles dans la mémoire UDVM

Les cinq premiers mots de deux octets sont initialisés pour contenir des valeurs qui pourraient être utiles au code d'octet UDVM (Valeurs utiles). Noter que ces valeurs ne sont que pour information et peuvent être outrepassées en exécutant le code d'octet UDVM sans aucun effet sur le point d'extrémité. Les bits de poids fort (MSB) de chaque mot de deux octets sont mémorisés avant les bits de moindre poids (LSB).

Les adresses 0 à 5 indiquent les ressources disponibles pour le point d'extrémité receveur. La taille de mémoire d'UDVM est exprimée en octets modulo 2^16, de sorte que, en particulier, elle est réglée à 0 si la taille de mémoire de l'UDVM est de 65536 octets. Les cycles_par_bit sont exprimés comme un entier de deux octets prenant les valeurs 16, 32, 64 ou 128. Version_SigComp est exprimé comme une valeur de deux octets comme indiqué au paragraphe 3.3.2.

Les adresses 6 à 9 sont initialisées à la longueur de l'identifiant d'état partiel, suivi par la Longueur_d'état provenant de l'élément d'état restitué. Tous deux sont exprimés comme des valeurs de deux octets.

Les adresses 10 à 31 sont réservées et sont initialisées à 0 pour la version 0x01 de SigComp. De futures versions de SigComp pourront utiliser ces localisations pour des valeurs utiles supplémentaires, et donc un décompresseur NE DOIT PAS s'appuyer sur ces valeurs qui sont à zéro.

Toute adresse restante dans la mémoire de l'UDVM qui n'est pas encore initialisée DOIT être réglée à 0.

L'UDVM commence alors à exécuter les instructions à l'adresse de mémoire contenue dans Instruction_d'état (qui fait partie de l'élément d'état restitué). Noter que le message SigComp restant est détenu par le décompresseur expéditeur jusqu'à ce qu'il soit demandé par l'UDVM.

(Noter que les valeurs utiles ne sont établies qu'au démarrage de l'UDVM ; il n'y a pas ensuite de signification particulière pour cette zone de mémoire. Cela signifie que le code d'octet de l'UDVM est libre d'utiliser ces localisations pour tout autre objet pour lequel une localisation de mémoire peut être utilisée ; il faut juste se souvenir qu'elles ne sont pas nécessairement initialisées à zéro.)

7.3 Chargement du code d'octet UDVM

Si le champ Longueur est réglé à 0, le code d'octet nécessaire pour décompresser le message SigComp est fourni au titre du message lui-même. Le champ de 12 bits Longueur_de_code spécifie la taille du code d'octet de l'UDVM chargé (de 0 à 4095 octets inclus) ; huit bits de poids fort sont dans le premier octet, suivis par quatre bits de moindre poids parmi les bits de poids fort dans le second octet. Les bits restants dans le second octet sont interprétés comme un champ de destination de quatre bits qui spécifie l'adresse de mémoire de début à laquelle le code d'octet est copié. Le champ Destination est codé comme suit :

Codage :	**Adresse de destination :**
0000	réservé
0001	2 * 64 = 128
0010	3 * 64 = 196
0011	4 * 64 = 256
:	:
1111	16 * 64 = 1024

Noter que le codage 0000 est réservé pour de futures versions SigComp, et cause un échec de décompression dans la version 0x01.

La mémoire d'UDVM est initialisée selon la Figure 5, sauf que les adresses 6 à 9 inclus sont réglées à 0 parce que aucun élément d'état n'a été atteint. L'UDVM commence alors l'exécution des instructions à l'adresse de mémoire spécifiée par le champ de destination. Comme ci-dessus, le message SigComp restant est détenu par le décompresseur expéditeur jusqu'à ce qu'il soit nécessaire pour l'UDVM.

8. Généralités sur l'UDVM

La fonction de décompression pour SigComp est fournie par une machine virtuelle de décompresseur universel (UDVM). L'UDVM est une machine virtuelle comme la machine virtuelle Java mais avec une différence clé : elle est seulement conçue pour les besoins du fonctionnement des algorithmes de décompression.

Le motif de la création de l'UDVM est de donner de la souplesse lors du choix de la façon de compresser un certain message d'application. Plutôt que de prendre un petit nombre d'algorithmes pré négociés, la mise en œuvre de compresseur est libre de retenir un algorithme de son choix. Les données compressées sont alors combinées avec un ensemble d'instructions d'UDVM qui permettent d'extraire les données d'origine, et le résultat est sorti comme un message SigComp. Comme l'UDVM est spécifiquement optimisée pour fonctionner avec des algorithmes de décompression, la taille de code d'un algorithme normal est petite (souvent moins de 100 octets). De plus, l'approche de l'UDVM n'ajoute pas de traitement supplémentaire significatif ou d'exigences de mémoire par rapport au fonctionnement d'un algorithme de décompression fixe préprogrammé.

La Figure 6 donne une vue détaillée des interfaces entre l'UDVM et son environnement.

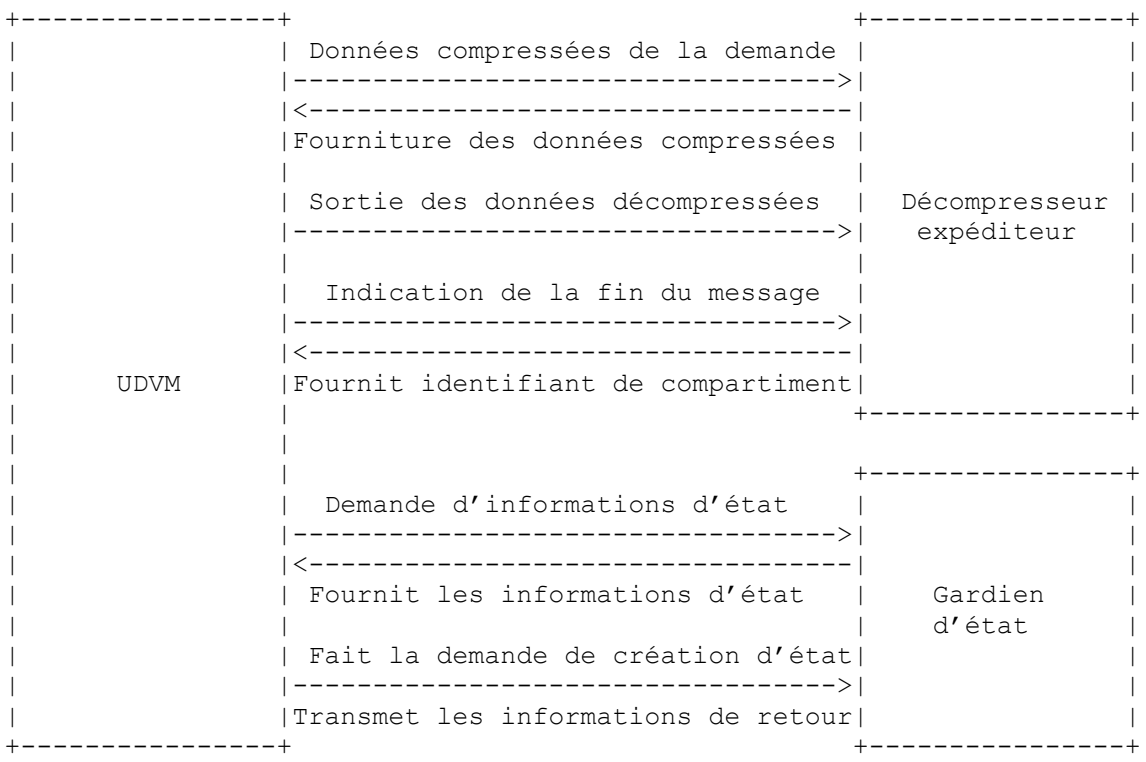


Figure 6 : Interfaces entre l'UDVM et son environnement

Noter qu'une fois l'UDVM initialisée, des données compressées supplémentaires et des informations d'état ne sont fournies qu'à la demande d'une instruction d'UDVM spécifique.

Cette section décrit les caractéristiques de base de l'UDVM incluant les registres UDVM et le format du code d'octet d'UDVM.

8.1 Registres UDVM

Les registres d'UDVM sont des mots de deux octets dans la mémoire d'UDVM qui ont des tâches particulières, par exemple de spécifier la localisation de la pile utilisée par les instructions CALL et RETURN.
Les registres d'UDVM sont illustrés par la Figure 7.

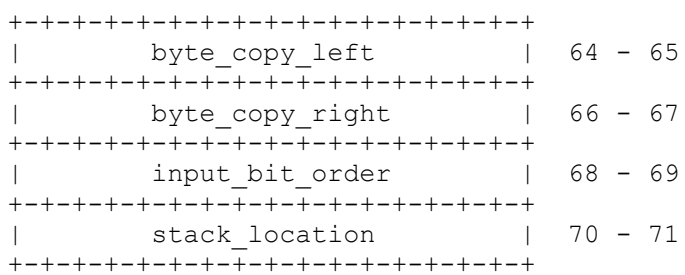


Figure 7 : Adresses de mémoire dans les registres UDVM

Les bits de poids fort (MSB) de chaque registre sont toujours mémorisés avant les bits de moindre poids (LSB). Ainsi, par exemple, les MSB de byte_copy_left sont mémorisés à l'adresse 64 alors que les LSB sont mémorisés à l'adresse 65.

L'utilisation de chaque registre UDVM est définie dans les paragraphes qui suivent.

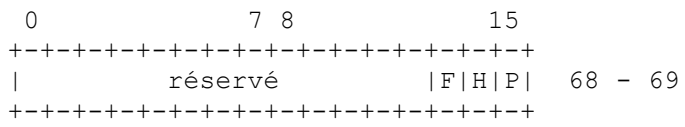
(Noter que les registres UDVM commencent à l'adresse 64, c'est-à-dire 32 octets après la zone réservée pour les valeurs utiles. L'intention est que le trou, c'est-à-dire, la zone entre l'adresse 32 et l'adresse 63, va souvent être utilisé comme bourrage initial *(scratch-pad)* qui est garanti à zéro au démarrage de l'UDVM et est efficacement adressable dans les types d'opérandes de référence ($) et multitype (%).)

8.2 Demande de données compressées supplémentaires

Le décompresseur expéditeur mémorise les données compressées provenant du message SigComp avant qu'elles soient demandées par l'UDVM via une des instructions INPUT. Lorsque le code d'octet UDVM est exécuté pour la première fois, l'expéditeur contient le message SigComp restant après que l'en-tête a été utilisé pour initialiser l'UDVM conformément à la Section 7.

Noter que les instructions INPUT-BITS et INPUT-HUFFMAN restituent un flux de bits compressés individuels provenant de l'expéditeur. Pour assurer la compatibilité au bit près avec divers algorithmes de compression bien connus, le registre input_bit_order *(ordre d'entrée des bits)* peut modifier l'ordre dans lequel les bits individuels sont passés au sein d'un octet.

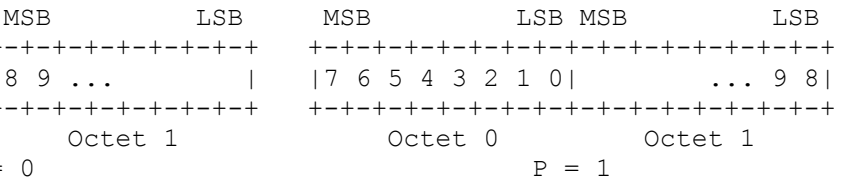
Le registre input_bit_order contient les trois fanions suivants :



Le bit P contrôle l'ordre dans lequel les bits sont passés de l'expéditeur aux instructions INPUT. S'il est à 0, il indique que les bits au sein d'un octet individuel sont passés aux instructions INPUT dans l'ordre du MSB au LSB. Si il est à 1, les bits sont passés du LSB au MSB.

Noter que le registre input_bit_order ne peut pas changer l'ordre dans lequel les octets eux-mêmes sont passés aux instructions INPUT (les octets sont toujours passés dans le même ordre que celui qu'ils ont dans le message SigComp).

Le diagramme suivant illustre l'ordre dans lequel les bits sont passés aux instructions INPUT pour les deux cas :



Noter qu'après une ou plusieurs instructions INPUT, l'expéditeur peut détenir une fraction d'un octet (ce qui était les LSB si P = 0, ou, les MSB si P = 1). Si une instruction INPUT est rencontrée et si le bit P a changé depuis la dernière instruction INPUT, toute fraction d'un octet encore détenue par l'expéditeur DOIT être éliminée (même si l'instruction INPUT ne demande aucun bit). Le premier bit passé à l'instruction INPUT est pris de l'octet suivant.

Lorsque une instruction INPUT demande n bits de données compressées, elle interprète les bits reçus comme un entier entre 0 et 2^n - 1. Le bit F et le bit H spécifient si les bits dans ces entiers sont considérés comme arrivant dans l'ordre du MSB au LSB (bit à 0) ou dans l'ordre du LSB au MSB (bit à 1).

Si le bit F est à 0, l'instruction INPUT-BITS interprète les bits reçus comme les MSB qui arrivent en premier, et si il est à 1, elle interprète les bits comme arrivant avec les LSB en premier. Le bit H effectue la même fonction pour l'instruction INPUT-HUFFMAN. Noter qu'il est possible de régler ces deux bits à des valeurs différentes afin d'utiliser des ordres de bits différents pour les deux instructions (certains algorithmes exigent en fait cela, par exemple, DEFLATE [RFC1951]). (Noter qu'il n'y a pas d'effet particulier à changer le bit F ou H entre les instructions INPUT, à la différence de la règle d'élimination pour le bit P décrite ci-dessus.)

Un échec de décompression survient si une instruction INPUT-BITS ou INPUT-HUFFMAN est rencontrée et si le registre input_bit_order ne se tient pas entre 0 et 7 inclus.

8.3 Pile UDVM

Certaines instructions d'UDVM utilisent une pile de mots de deux octets mémorisés à l'adresse de mémoire spécifiée par le mot de deux octets stack_location *(localisation de la pile)*. La pile contient les mots suivants :

Nom :	**Adresse de mémoire de début**
stack_fill	stack_location
stack[0]	stack_location + 2
stack[1]	stack_location + 4
stack[2]	stack_location + 6
:	:

La notation stack_location est une abréviation pour le contenu du registre stack_location, c'est-à-dire, le mot de deux octets situé en 70 et 71. La notation stack_fill est une abréviation pour le mot de deux octets situé à stack_location et stack_location+1. De même, la notation stack[n] est une abréviation pour le mot de deux octets situé à stack_location+2*n+2 et stack_location+2*n+3. (Comme toujours, l'arithmétique est modulo 2^16.)

La pile est utilisée par les instructions CALL, RETURN, PUSH et POP.

"Pousser" une valeur sur la pile est une abréviation pour copier la valeur dans stack[stack_fill] et ensuite augmenter stack_fill de 1. CALL et PUSH poussent les valeurs sur la pile.

"Faire sauter" une valeur de la pile est une abréviation pour diminuer stack_fill de 1, et ensuite utiliser la valeur mémorisée dans stack[stack_fill]. Un échec de décompression survient si stack_fill est à zéro au commencement d'une opération de saut. POP et RETURN font sauter les valeurs de la pile.

Pour ces deux opérations abstraites, l'UDVM prend d'abord note de la valeur actuelle de stack_location et utilise cette valeur pour les deux sous opérations (accéder à la pile et manipulation de stack_fill) c'est-à-dire que écraser stack_location dans le courant de l'opération n'a pas de conséquences pour l'opération.

8.4 Copie d'octet

Un certain nombre d'instructions de l'UDVM exigent qu'une chaîne d'octets soit copiée de et vers des zones de la mémoire de l'UDVM. Ce paragraphe définit comment devrait être effectuée l'opération de copie.

La chaîne d'octets est copiée en ordre ascendant d'adresse de mémoire, en respectant les limites établies par byte_copy_left et byte_copy_right. Plus précisément, si un octet est copié de/vers l'adresse m, alors le prochain octet est copié de/vers l'adresse n, où n est calculé comme suit :

Soit k := m + 1 (modulo 2^16)
Si k = byte_copy_right alors on a n := byte_copy_left, autrement on a n := k

Un échec de décompression survient si un octet est copié de/vers une adresse au delà de la mémoire de l'UDVM.

Noter que la chaîne d'octets est copiée un octet à la fois. En particulier, certains des derniers octets à copier peuvent eux-mêmes avoir été écrits dans la mémoire de l'UDVM par l'opération de copie d'octets en cours.

De même, il est possible qu'une opération de copie d'octet écrase l'instruction qui impliquait cette copie d'octet. Si cela arrive, l'opération de copie d'octet DOIT être achevée comme si l'instruction d'origine était encore en place dans la mémoire de l'UDVM (cela s'applique aussi si byte_copy_left ou byte_copy_right sont écrasés).

La copie d'octet est utilisée par les instructions d'UDVM suivantes : SHA-1, COPY, COPY-LITERAL, COPY-OFFSET, MEMSET, INPUT-BYTES, STATE-ACCESS, OUTPUT, END-MESSAGE

8.5 Opérandes d'instruction et code d'octet UDVM

Chacune des instructions d'UDVM dans un élément de code d'octet d'UDVM est représentée par un seul octet, suivi par 0, un ou plusieurs octets contenant les opérandes requis par l'instruction.

Durant l'exécution de l'instruction, l'UDVM va d'abord aller chercher le premier octet de l'instruction, déterminer le nombre et les types d'opérandes requis pour cette instruction, et décoder ensuite tous les opérandes à la suite avant de commencer à agir sur l'instruction. (Noter que les instructions d'UDVM ont été conçues de telle sorte que cette séquence reste conceptuelle dans les cas où elle résulterait en un fardeau déraisonnable pour la mise en œuvre.)

Pour réduire la taille du code d'octet d'UDVM normal, chaque opérande pour une instruction d'UDVM est compressé en utilisant un codage de longueur variable. L'objectif est de mémoriser les valeurs d'opérande les plus courantes en utilisant moins d'octets que pour les valeurs qui se produisent plus rarement.

Quatre différents types d'opérande sont disponibles : le littéral, la référence, le multitype et l'adresse. La Section 9 donne une liste complète des instructions d'UDVM et des types d'opérande qui suivent chaque instruction. Le code d'octet UDVM pour chaque type d'opérande est illustré de la Figure 8 à la Figure 10, avec les valeurs d'entiers représentées par le code d'octet.
Noter que les MSB dans le code d'octet sont illustrés comme précédant les LSB. Aussi, toute chaîne de bits marquée avec k "n" consécutifs est à interpréter comme un entier N de 0 à 2^k - 1 inclus (avec les MSB de n illustrés comme précédant les LSB).
La valeur d'entier décodée du code d'octet peut être interprétée de deux façons. Dans certains cas, elle est prise comme la valeur réelle de l'opérande. Dans d'autres cas, elle est prise comme étant une adresse mémoire à laquelle la valeur de deux octets de l'opérande peut être trouvée (les MSB se trouvent à l'adresse spécifiée, les LSB se trouvent à l'adresse suivante). Ces derniers cas sont notés par memory[X] où X est l'adresse et memory[X] est la valeur de deux octets qui commence à l'adresse X.

Le plus simple type d'opérande est le littéral (#), qui code un entier constant de 0 à 65535 inclus. Un opérande littéral peut exiger entre 1 et 3 octets selon sa valeur.

Code d'octet :	**Valeur d'opérande :**	**Gamme :**
0nnnnnnn	N	0 - 127
10nnnnnn nnnnnnnn	N	0 - 16383
11000000 nnnnnnnn nnnnnnnn	N	0 - 65535

Figure 8 : Code d'octet pour opérande littéral (#)

Le second type d'opérande est la référence ($), qui est toujours utilisée pour accéder à une valeur de deux octets située ailleurs dans la mémoire d'UDVM. Le code d'octet pour un opérande référence est décodé comme un entier constant de 0 à 65535 inclus, qui est interprété comme l'adresse mémoire qui contient la valeur réelle de l'opérande.

Code d'octet :	**Valeur d'opérande :**	**Gamme :**
0nnnnnnn	memory[2 * N]	0 - 65535
10nnnnnn nnnnnnnn	memory[2 * N]	0 - 65535
11000000 nnnnnnnn nnnnnnnn	memory[N]	0 - 65535

Figure 9 : Code d'octet pour opérande de référence ($)

Noter que la gamme de l'opérande référence est toujours 0 - 65535 indépendamment du nombre de bits utilisé pour coder la référence, parce que l'opérande se réfère toujours à une valeur de deux octets dans la mémoire.

La troisième sorte d'opérande est le multitype (%), qui peut être utilisé pour coder aussi bien des valeurs réelles que des adresses mémoire. L'opérande multitype offre aussi un codage efficace pour les petites valeurs d'entier (positives et négatives) et pour les puissances de 2.

Code d'octet :	**Valeur d'opérande :**	**Gamme :**
00nnnnnn	N	0 - 63
01nnnnnn	memory[2 * N]	0 - 65535
1000011n	2 ^ (N + 6)	64 , 128
10001nnn	2 ^ (N + 8)	256 , ... , 32768
111nnnnn	N + 65504	65504 - 65535
1001nnnn nnnnnnnn	N + 61440	61440 - 65535
101nnnnn nnnnnnnn	N	0 - 8191
110nnnnn nnnnnnnn	memory[N]	0 - 65535
10000000 nnnnnnnn nnnnnnnn	N	0 - 65535
10000001 nnnnnnnn nnnnnnnn	memory[N]	0 - 65535

Figure 10 : Code d'octet pour opérande multitype (%)

Le quatrième type d'opérande est l'adresse (@). Cet opérande est décodé comme un opérande multitype suivi par une autre étape : l'adresse mémoire de l'instruction d'UDVM qui contient l'opérande adresse est ajoutée pour obtenir la valeur d'opérande correcte. De sorte que si la valeur d'opérande de la Figure 10 est D, la valeur d'opérande réelle d'une adresse est calculée comme suit :

valeur_d'opérande = (adresse_mémoire_de_l'instruction + D) modulo 2^16

Les opérandes d'adresse sont toujours utilisés dans les instructions qui contrôlent les flux de programme, parce que ils assurent que le code d'octet UDVM est un code indépendant de la position (c'est-à-dire, il va fonctionner sans dépendre de l'endroit où il est placé dans la mémoire d'UDVM).

8.6 Cycles UDVM

Une fois que l'UDVM a été invoquée, elle exécute à la suite les unes des autres les instructions contenues dans sa mémoire sauf indication contraire (par exemple, lorsque l'UDVM rencontre une instruction JUMP). Si la prochaine instruction à exécuter se tient en dehors de la mémoire disponible, il survient alors un échec de décompression (voir le paragraphe 8.7).

Pour s'assurer qu'un message SigComp ne peut pas consommer trop de ressources de traitement, SigComp limite le nombre de "cycles UDVM" alloué à chaque message. Le nombre de cycles UDVM disponibles est initialisé à 1000 plus le nombre de bits de l'en-tête SigComp (comme décrit à la Section 7) ; cette somme est ensuite multipliée par cycles_par_bit. Chaque fois qu'une instruction est exécutée, le nombre de cycles UDVM disponibles est diminué de la quantité spécifiée à la Section 9. De plus, si l'UDVM réussit à demander n bits de données compressées en utilisant une des instructions INPUT, le nombre de cycles UDVM disponibles est alors augmenté de n * cycles_par_bit une fois que l'instruction a été exécutée.

Cela signifie que le nombre maximum de cycles UDVM disponibles pour traiter un message SigComp de n octets est donné par la formule :

maximum_cycles_UDVM = (8 * n + 1000) * cycles_par_bit

La raison pour laquelle ce total n'est pas alloué à l'UDVM quand elle est invoquée est que l'UDVM peut commencer à décompresser un message qui a été reçu seulement partiellement. De sorte que la taille totale du message peut n'être pas connue lorsque l'UDVM est initialisée. Noter que le nombre de cycles UDVM NE DOIT PAS être augmenté si une demande de données compressées supplémentaires échoue.

L'UDVM arrête d'exécuter les instructions quand elle rencontre une instruction END-MESSAGE ou si un échec de décompression survient (voir les détails au paragraphe 8.7).

8.7 Échec de décompression

Si un message compressé donné à l'UDVM est corrompu (accidentellement ou par malveillance) l'UDVM peut alors terminer avec un échec de décompression.

Les raisons d'un échec de décompression incluent ce qui suit :
1. un message SigComp contient un en-tête invalide selon la Section 7,
2. un message SigComp est plus grand que taille_de_mémoire_de_décompression,
3. une instruction coûte plus que le nombre de cycles UDVM restants,
4. l'UDVM tente de lire ou écrire à partir d'une adresse mémoire au delà de sa taille de mémoire,.
5. une instruction inconnue est rencontrée,
6. un opérande inconnu est rencontré,
7. une instruction est rencontrée qui ne peut pas être bien traitée par l'UDVM (par exemple une instruction RETURN alors qu'aucune instruction CALL n'a été rencontrée précédemment,
8. une demande d'accès à des informations d'état échoue,
9. un échec de décompression manuel est déclenché en utilisant l'instruction DECOMPRESSION-FAILURE.

Si un échec de décompression survient lors de la décompression d'un message, l'UDVM informe alors l'expéditeur et n'effectue pas d'autre action. Il est de la responsabilité de l'expéditeur de décider comment agir en cas d'échec de décompression. En général, un expéditeur DEVRAIT éliminer le message compressé (ou le flux compressé si le transport est fondé sur le flux) et toutes les données décompressées qui ont été sorties mais pas encore passées à l'application.

9. Ensemble d'instructions UDVM

L'UDVM comprend actuellement 36 instructions, choisies pour prendre en charge la gamme la plus large possible d'algorithmes de compression avec le minimum possible de redondance.

La Figure 11 fait la liste des différentes instructions et des valeurs du code d'octet utilisées pour coder les instructions. Le coût de chaque instruction en cycles UDVM est aussi donné :

Instruction :	**Valeur de code d'octet :**	**Coût en cycles UDVM :**
DECOMPRESSION-FAILURE	0	1
AND	1	1
OR	2	1
NOT	3	1
LSHIFT	4	1
RSHIFT	5	1
ADD	6	1
SUBTRACT	7	1
MULTIPLY	8	1
DIVIDE	9	1
REMAINDER	10	1
SORT-ASCENDING	11	1 + k * (plafond(log2(k)) + n)
SORT-DESCENDING	12	1 + k * (plafond(log2(k)) + n)
SHA-1	13	1 + longueur
LOAD	14	1
MULTILOAD	15	1 + n
PUSH	16	1
POP	17	1
COPY	18	1 + longueur
COPY-LITERAL	19	1 + longueur
COPY-OFFSET	20	1 + longueur
MEMSET	21	1 + longueur
JUMP	22	1
COMPARE	23	1
CALL	24	1
RETURN	25	1
SWITCH	26	1 + n
CRC	27	1 + longueur
INPUT-BYTES	28	1 + longueur
INPUT-BITS	29	1
INPUT-HUFFMAN	30	1 + n
STATE-ACCESS	31	1 + longueur_d'état
STATE-CREATE	32	1 + longueur_d'état
STATE-FREE	33	1
OUTPUT	34	1 + longueur_de_sortie
END-MESSAGE	35	1 + longueur_d'état

Figure 11 : Instructions UDVM et valeurs de code d'octet correspondantes

Chaque instruction d'UDVM coûte un minimum de un cycle UDVM. Certaines instructions peuvent coûter des cycles supplémentaires selon les valeurs des opérandes d'instruction. Les variables nommées dans les expressions de coût se réfèrent aux valeurs des opérandes d'instruction qui ont ces noms.

Noter que pour les instructions SORT *(trier)*, la formule plafond(log2(k)) calcule la plus petite valeur i telle que $k \leq 2^i$.

L'ensemble des instructions d'UDVM offre un mélange d'instructions de bas et de haut niveau. Les instructions de haut niveau peuvent toutes être émulées en utilisant des combinaisons d'instructions de bas niveau, mais quand on a le choix, il est généralement préférable d'utiliser une seule instruction plutôt qu'un grand nombre d'instructions d'usage général. Le code d'octet résultant sera plus compact (conduisant à un taux de compression global supérieur) et la décompression sera normalement plus rapide parce que la mise en œuvre des instructions de haut niveau peut être plus facilement optimisée.

Toutes les instructions sont codées sur un seul octet pour indiquer le type d'instruction, suivi par 0, un ou plusieurs octets contenant les opérandes requis par l'instruction. L'instruction spécifie lesquels des quatre types d'opérande du paragraphe 8.5 sont utilisés dans chaque cas. Par exemple l'instruction ADD *(ajouter)* est suivie par deux opérandes :

ADD ($operand_1, %operand_2)

Lorsque il est converti en code d'octet, le nombre d'octets requis par l'instruction ADD dépend de la valeur de chaque opérande, et si l'opérande multitype contient la valeur de l'opérande elle-même ou une adresse mémoire où se trouve la valeur réelle de l'opérande.

Chaque instruction est expliquée plus en détail ci-dessous.

Chaque fois que la description d'une instruction utilise l'expression "et ensuite", la sémantique prévue est que l'effet expliqué avant "et ensuite" est achevé avant que le travail sur l'effet expliqué après le "et ensuite" ne soit commencé.

9.1 Instructions mathématiques

Les instructions suivantes fournissent un certain nombre d'opérations mathématiques incluant la manipulation, l'arithmétique et le tri de bits.

9.1.1 Manipulation binaire

Les instructions AND *(et)*, OR *(ou)*, NOT *(non)*, LSHIFT *(glissement gauche)* et RSHIFT *(glissement droite)* fournissent une simple manipulation de bit sur des mots de deux octets.

AND ($operand_1, %operand_2)
OR ($operand_1, %operand_2)
NOT ($operand_1)
LSHIFT ($operand_1, %operand_2)
RSHIFT ($operand_1, %operand_2)

Après l'achèvement de l'opération, la valeur du premier opérande est écrasée par le résultat. (Noter que comme cet opérande est une référence, c'est le mot de 2 octets à l'adresse mémoire spécifiée par l'opérande qui est écrasé.)

Les définitions précises de LSHIFT et RSHIFT sont données ci-dessous. Noter que m et n sont les valeurs de deux octets codées par les opérandes, et que plancher(x) calcule le plus grand entier non supérieur à x :

LSHIFT (m, n) := m * 2^n (modulo 2^16)
RSHIFT (m, n) := plancher(m / 2^n)

9.1.2 Arithmétique

Les instructions ADD *(ajouter)*, SUBTRACT'*(retancher)*, MULTIPLY *(multiplier)*, DIVIDE *(diviser)* et REMAINDER *(reste)* effectuent l'arithmétique sur les mots de deux octets.

ADD ($operand_1, %operand_2)
SUBTRACT ($operand_1, %operand_2)
MULTIPLY ($operand_1, %operand_2)
DIVIDE ($operand_1, %operand_2)
REMAINDER ($operand_1, %operand_2)

Après l'achèvement de l'opération, la valeur du premier opérande est écrasée par le résultat.

La définition précise de chaque instruction est donnée ci-dessous :

ADD (m, n)	: = m + n (modulo 2^16)
SUBTRACT (m, n)	: = m - n (modulo 2^16)
MULTIPLY (m, n)	: = m * n (modulo 2^16)
DIVIDE (m, n)	: = plancher(m / n)
REMAINDER (m, n)	: = m - n * plancher(m / n)

Un échec de décompression survient si une instruction DIVIDE ou REMAINDER rencontre un operand_2 à zéro.

9.1.3 Tri

Les instructions SORT-ASCENDING *(tri ascendant)* et SORT-DESCENDING *(tri descendant)* trient des listes de mots de deux octets.

SORT-ASCENDING (%start, %n, %k)
SORT-DESCENDING (%start, %n, %k)

L'opérande start spécifie l'adresse mémoire de départ du bloc de données à trier.

Le bloc de données lui-même est divisé en n listes dont chacune contient k mots de 2 octets. L'instruction SORT-ASCENDING s'applique à une certaine permutation des listes, telle que la première liste soit triée en ordre ascendant (en traitant chaque mot de 2 octets comme un entier non signé). La même permutation est appliquée aux n listes, de sorte que les listes autres que la première ne seront pas nécessairement triées dans l'ordre.

Dans le cas où deux mots ont la même valeur, l'ordre d'origine de la liste est préservé.

Par exemple, la première liste pourrait contenir un ensemble d'entiers à trier tandis que la seconde liste pourrait être utilisée pour garder la trace de l'endroit où apparaissent les entiers dans la liste triée :

Avant le tri		**Après le tri**	
Liste 1	**Liste 2**	**Liste 1**	**Liste 2**
8	1	1	2
1	2	1	3
1	3	3	4
3	4	8	1

L'instruction SORT-DESCENDING se comporte comme ci-dessus, sauf que la première liste est triée en ordre descendant.

9.1.4 SHA-1

L'instruction SHA-1 calcule un hachage SHA-1 [RFC3174] de 20 octets sur la zone spécifiée de la mémoire UDVM.

SHA-1 (%position, %longueur, %destination)

Les opérandes position et longueur spécifient l'adresse mémoire de début et la longueur de la chaîne d'octet sur laquelle le hachage SHA-1 est calculé. Les règles de copie des octets sont appliquées conformément au paragraphe 8.4.

L'opérande destination donne l'adresse de début à laquelle le hachage résultant de 20 octets sera copié. Les règles de copie d'octet sont appliquées comme ci-dessus.

9.2 Instructions de gestion de mémoire

Les instructions suivantes sont utilisées pour établir la mémoire d'UDVM, et pour copier les chaînes d'octet d'une localisation de mémoire à une autre .

9.2.1 LOAD

L'instruction LOAD *(charger)* règle un mot de deux octets à une certaine valeur spécifiée. Le format d'une instruction LOAD est le suivant :

LOAD (%adresse, %valeur)

Le premier opérande spécifie l'adresse de début d'un mot de deux octets, tandis que le second opérande spécifie la valeur à charger dans ce mot. Comme d'habitude, les MSB sont mémorisés avant les LSB dans la mémoire d'UDVM.

9.2.2 MULTILOAD

L'instruction MULTILOAD établit un bloc contigu de mots de deux octets dans la mémoire d'UDVM aux valeurs spécifiées.

MULTILOAD (%adresse, #n, %valeur_0, ..., %valeur_n-1)

Le premier opérande spécifie l'adresse de début des mots de deux octets contigus, tandis que les opérandes valeur_0 à valeur_n-1 spécifient les valeurs à charger dans ces mots (dans le même ordre que celui qui apparaît dans l'instruction).

Un échec de décompression survient si l'ensemble de mots de deux octets établi par l'instruction se chevaucherait avec les localisations de mémoire détenues par l'instruction (incluant ses opérandes) elle-même, c'est-à-dire, si l'instruction était auto modificatrice. (Cette restriction rend plus simple la mise en œuvre de MULTILOAD étape par étape au lieu d'avoir à décoder tous les opérandes avant d'être capable de copier les données, comme l'implique le modèle conceptuel d'exécution d'instruction.)

9.2.3 PUSH et POP

Les instructions PUSH *(pousser)* et POP *(sauter)* lisent et écrivent sur la pile UDVM (comme défini au paragraphe 8.3).

PUSH (%valeur)
POP (%adresse)

L'instruction PUSH pousse la valeur spécifiée par son opérande sur la pile.

L'instruction POP fait sauter une valeur de la pile et copie ensuite la valeur à l'adresse mémoire spécifiée. (Noter que l'expression "et ensuite" implique que la copie de la valeur est sans conséquence pour l'opération sur la pile elle-même, qui se fait antérieurement.)

Voir au paragraphe 8.3 les conditions d'erreur possibles.

9.2.4 COPY

L'instruction COPY est utilisée pour copier une chaîne d'octets d'une partie de la mémoire UDVM à une autre.

COPY (%position, %longueur, %destination)

L'opérande position spécifie l'adresse mémoire du premier octet dans la chaîne à copier, et l'opérande longueur spécifie le nombre d'octets à copier.

L'opérande destination donne l'adresse à laquelle le premier octet de la chaîne sera copié.

La copie d'octets est effectuée selon les règles du paragraphe 8.4.

9.2.5 COPY-LITERAL

Une version modifiée de l'instruction COPY est donnée ci-dessous :

COPY-LITERAL (%position, %longueur, $destination)

L'instruction COPY-LITERAL se comporte comme une instruction COPY sauf qu'après l'achèvement de la copie, la valeur de l'opérande destination est remplacée par l'adresse à laquelle le prochain octet de données sera copié. Plus précisément, elle est remplacée par la valeur n, déduite selon le paragraphe 8.4 avec m réglé à l'adresse de destination du dernier octet à copier, s'il en est (c'est-à-dire, si la valeur de l'opérande longueur est à zéro, la valeur de l'opérande destination n'est pas changée).

9.2.6 COPY-OFFSET

Une autre version de l'instruction COPY-LITERAL est donnée ci-dessous :

COPY-OFFSET (%décalage, %longueur, $destination)

L'instruction COPY-OFFSET se comporte comme une instruction COPY-LITERAL sauf qu'un opérande offset *(décalage)* est donné au lieu d'un opérande position.

Pour déduire la valeur de l'opérande position, en commençant à l'adresse mémoire spécifiée par la destination, l'UDVM compte vers l'arrière un total de décalage d'adresses mémoire.

Si l'adresse mémoire spécifiée dans byte_copy_left est atteinte, l'adresse mémoire suivante est prise comme étant (byte_copy_right - 1) modulo 2^16.

L'instruction COPY-OFFSET se comporte alors comme une instruction COPY-LITERAL, en prenant la valeur de l'opérande position comme étant la dernière adresse mémoire atteinte dans l'étape ci-dessus.

9.2.7 MEMSET

L'instruction MEMSET initialise une zone de mémoire UDVM à une séquence de valeurs spécifiée. Le format d'une instruction MEMSET est le suivant :

MEMSET (%adresse, %longueur, %valeur_de_début, %décalage)

La séquence des valeurs utilisées par l'instruction MEMSET est spécifiée par la formule suivante :

Seq[n] := (valeur_de_début + n * décalage) modulo 256

Les valeurs Seq[0] à Seq[longueur - 1] inclus sont chacune interprétée comme un seul octet, et ensuite enchaînées pour former une chaîne d'octets où le premier octet a la valeur Seq[0], le second octet a la valeur Seq[1] et ainsi de suite jusqu'au dernier octet qui a la valeur Seq[longueur - 1].

La chaîne est ensuite copiée octet par octet dans la mémoire d'UDVM commençant à l'adresse mémoire spécifiée comme opérande à l'instruction MEMSET, en respectant les règles du paragraphe 8.4. (Noter que la chaîne d'octets peut écraser l'instruction MEMSET ou ses opérandes; comme expliqué au paragraphe 8.5, l'instruction MEMSET doit être exécutée comme si les opérandes d'origine étaient toujours en place dans la mémoire UDVM.)

9.3 Instructions de flux de programme

Les instructions suivantes altèrent le flux de code UDVM. Chaque instruction saute à une des adresses mémoire sur la base d'un certain critère spécifié.

Noter que certaines instructions entrée/sortie (voir au paragraphe 9.4) peuvent aussi altérer le flux de programme.

9.3.1 JUMP

L'instruction JUMP déplace l'exécution du programme à l'adresse mémoire spécifiée.

JUMP (@adresse)

Un échec de décompression survient si la valeur de l'opérande adresse va au delà de la taille globale de mémoire d'UDVM.

9.3.2 COMPARE

L'instruction COMPARE compare deux opérandes et saute ensuite à une des trois adresses mémoire spécifiées, selon le résultat.

COMPARE (%valeur_1, %valeur_2, @adresse_1, @adresse_2, @adresse_3)

Si valeur_1 < valeur_2, alors l'UDVM continue l'exécution de l'instruction à l'adresse mémoire spécifiée par adresse 1. Si valeur_1 = valeur_2 elle saute alors à l'adresse spécifiée par adresse_2. Si valeur_1 > valeur_2, elle saute alors à l'adresse spécifiée par adresse_3.

9.3.3 CALL et RETURN

Les instructions CALL *(invoquer)* et RETURN *(retourner)* permettent la prise en charge d'algorithmes de compression avec une structure incorporée.

CALL (@adresse)
RETURN

Les deux instructions utilisent la pile UDVM du paragraphe 8.3. Lorsque l'UDVM atteint une instruction CALL, elle trouve l'adresse mémoire de l'instruction qui suit immédiatement l'instruction CALL et elle pousse cette valeur de deux octets sur la pile, prête pour une restitution ultérieure. Elle continue ensuite l'exécution de l'instruction à l'adresse mémoire spécifiée par l'opérande adresse.

Lorsque l'UDVM atteint une instruction RETURN, elle fait sauter une valeur de la pile et continue ensuite l'exécution de l'instruction à l'adresse mémoire qui vient juste d'être sautée.

Voir les conditions d'erreur au paragraphe 8.3.

9.3.4 SWITCH

L'instruction SWITCH *(échanger)* effectue un bond conditionnel sur la base de la valeur d'un de ses opérandes.

SWITCH (#n, %j, @adresse_0, @adresse_1, ... , @adresse_n-1)

Lorsque elle rencontre une instruction SWITCH, l'UDVM lit la valeur de j. Elle continue ensuite l'exécution de l'instruction à l'adresse spécifiée par adresse j.

Un échec de décompression survient si j spécifie une valeur de n ou plus, ou si l'adresse est au delà de la taille globale de mémoire UDVM.

9.3.5 CRC

L'instruction CRC *(contrôle de redondance cyclique)* vérifie une chaîne d'octets en utilisant un CRC de deux octets.

CRC (%valeur, %position, %longueur, @adresse)

Le calcul réel du CRC est effectué en utilisant le générateur polynomial x^16 + x^12 + x^5 + 1, qui coincide avec la séquence de deux octets de vérification de (FCS, *Frame Check Sequence*) de PPP [RFC1662].

Les opérandes position et longueur définissent la chaîne d'octets sur laquelle le CRC est évalué. Les règles de copie d'octet sont appliquées selon le paragraphe 8.4.

La valeur du CRC est calculée exactement comme défini pour le calcul de la FCS de 16 bits dans la [RFC1662].

L'opérande valeur contient la valeur d'entier attendue du CRC de deux octets. Si le CRC calculé correspond à la valeur attendue, l'UDVM continue alors l'exécution d'instructions à l'instruction suivante. Autrement l'UDVM saute à l'adresse mémoire spécifiée par l'opérande adresse.

9.4 Instructions d'entrée/sortie

Les instructions suivantes permettent à l'UDVM de s'interfacer avec son environnement. Noter que dans l'architecture globale de SigComp toutes ces interfaces passent les données au décompresseur expéditeur ou au gardien d'état.

9.4.1 DECOMPRESSION-FAILURE

L'instruction DECOMPRESSION-FAILURE *(échec de décompression)* déclenche un échec de décompression manuel. C'est utile si le code d'octet UDVM découvre qu'il ne peut pas réussir à décompresser le message (par exemple, en utilisant l'instruction CRC).

Cette instruction n'a pas d'opérande.

9.4.2 INPUT-BYTES

L'instruction INPUT-BYTES *(entrer les octets)* demande un certain nombre d'octets de données compressées provenant du décompresseur expéditeur.

INPUT-BYTES (%longueur, %destination, @adresse)

L'opérande longueur indique le nombre d'octets demandé de données compressées, et l'opérande destination spécifie l'adresse mémoire de début où elles devraient être copiées. La copie d'octet est effectuée selon les règles du paragraphe 8.4.

Si l'instruction demande des données qui sont au delà de la fin du message SigComp, aucune donnée n'est retournée. L'UDVM déplace l'exécution de programme à l'adresse spécifiée par l'opérande adresse.

Si le INPUT-BYTES est rencontré après qu'une instruction INPUT-BITS ou INPUT-HUFFMAN a été utilisée, et si l'expéditeur détient actuellement une fraction d'un octet, la fraction DOIT alors être éliminée avant qu'aucune donnée ne soit passée à l'UDVM. Le premier octet à être passé est l'octet qui suit immédiatement les données éliminées.

9.4.3 INPUT-BITS

L'instruction INPUT-BITS demande un certain nombre de bits de données compressées au décompresseur expéditeur.

INPUT-BITS (%longueur, %destination, @adresse)

L'opérande longueur indique le nombre de bits demandés. Un échec de décompression survient si cet opérande n'est pas entre 0 et 16 inclus.

L'opérande destination spécifie l'adresse mémoire à laquelle les données compressées devraient être copiées. Noter que les bits demandés sont interprétés comme un entier de deux octets dans la gamme de 0 à 2^longueur - 1, comme expliqué au paragraphe 8.2.

Si l'instruction demande des données qui vont au delà de la fin du message SigComp, aucune donnée n'est retournée. L'UDVM passe l'exécution du programme à l'adresse spécifiée par l'opérande adresse.

9.4.4 INPUT-HUFFMAN

L'instruction INPUT-HUFFMAN demande un nombre variable de bits de données compressées au décompresseur expéditeur. L'instruction demande initialement un petit nombre de bits et compare le résultat à certains critères ; si les critères ne sont pas satisfaits, des bits supplémentaires sont alors demandés jusqu'à ce que les critères soient réalisés.

L'instruction INPUT-HUFFMAN est suivie par trois opérandes obligatoires plus n ensembles supplémentaires d'opérandes. Chaque ensemble supplémentaire contient quatre opérandes comme montré ci-dessous :

INPUT-HUFFMAN (%destination, @adresse, #n, %bits_1, %lower_bound_1, %upper_bound_1, %uncompressed_1, ... , %bits_n, %lower_bound_n, %upper_bound_n, %uncompressed_n)

Noter que si n = 0 alors l'instruction INPUT-HUFFMAN est ignorée et l'exécution du programme reprend à l'instruction suivante. Un échec de décompression survient si (bits_1 + ... + bits_n) > 16.

Dans tous les autres cas, le comportement de l'instruction INPUT-HUFFMAN est défini ci-dessous :

1. régler j := 1 et régler H := 0.
2. demander bits_j bits compressés. Interpréter les bits retournés comme un entier k de 0 à 2^bits_j - 1, comme expliqué au paragraphe 8.2.
3. régler H := H * 2^bits_j + k.
4. si des données sont demandées qui se tiennent au delà de la fin du message SigComp, terminer l'instruction INPUT-HUFFMAN et passer l'exécution du programme à l'adresse mémoire spécifiée par l'opérande adresse.
5. si (H < lower_bound_j) ou (H > upper_bound_j) alors régler j := j + 1. Puis revenir à l'étape 2, sauf si j > n auquel cas survient un échec de décompression.
6. copier (H + uncompressed_j - lower_bound_j) modulo 2^16 à l'adresse mémoire spécifiée par l'opérande destination.

9.4.5 STATE-ACCESS

L'instruction STATE-ACCESS restitue des informations d'état mémorisées précédemment.

STATE-ACCESS (%partial_identifier_start, %partial_identifier_length, %state_begin, %state_length, %state_address, %state_instruction)

Les opérandes partial_identifier_start et partial_identifier_length spécifient la localisation de l'identifiant d'état partiel utilisé pour restituer les informations d'état. Cet identifiant a la même fonction que l'identifiant d'état partiel transmis dans le message SigComp au paragraphe 7.2.

Un échec de décompression survient si partial_identifier_length ne se tient pas entre 6 et 20 inclus. Un échec de décompression survient aussi si aucun élément d'état correspondant à l'identifiant d'état partiel ne peut être trouvé, si plus d'un état correspond à l'identifiant partiel, ou si partial_identifier_length est inférieur au minimum_access_length de l'élément d'état correspondant. Autrement, un élément d'état est retourné du gardien d'état.

Si un des opérandes state_address, state_instruction ou state_length est réglé à 0, alors on prend à la place sa valeur dans l'élément d'état retourné.

Noter que lors du calcul du nombre de cycles d'UDVM, l'instruction STATE-ACCESS coûte (1 + state_length) cycles. La valeur de state_length DOIT être prise dans l'élément d'état retourné au cas où l'opérande state_length serait 0.

Les opérandes state_begin et state_length définissent l'octet de début et le nombre d'octets à copier de la state_value contenue dans l'élément d'état retourné. Un échec de décompression survient si les octets sont copiés d'au delà de la fin de la state_value. Noter qu'un échec de décompression surviendra toujours si l'opérande state_length est réglé à 0 et que l'opérande state_begin n'est pas à zéro.

L'opérande state_address contient une adresse mémoire UDVM. La portion demandée de state_value est copiée octet par octet à cette adresse mémoire en utilisant les règles du paragraphe 8.4.

L'exécution du programme reprend alors à l'adresse mémoire spécifiée par state_instruction, sauf si cette adresse est 0, auquel cas l'exécution du programme reprend à la prochaine instruction qui suit l'instruction STATE-ACCESS. Noter que ce dernier cas ne se produit que si l'opérande state_instruction et la valeur state_instruction provenant de l'état demandé sont tous deux réglés à 0.

9.4.6 STATE-CREATE

L'instruction STATE-CREATE demande la création d'un élément d'état au point d'extrémité receveur.

STATE-CREATE (%state_length, %state_address, %state_instruction, %minimum_access_length, %state_retention_priority)

Noter que le nouvel élément d'état ne peut pas être créé tant qu'un identifiant de compartiment valide n'a pas été retourné par l'application. Par conséquent, lorsque une instruction STATE-CREATE est rencontrée, l'UDVM met simplement en mémoire tampon les cinq opérandes fournis jusqu'à ce que l'instruction END-MESSAGE soit atteinte. Les étapes suivies à ce point sont décrites au paragraphe 9.4.9.

Un échec de décompression DOIT survenir si plus de quatre demandes de création d'état sont faites avant que soit rencontrée l'instruction END-MESSAGE. Un échec de décompression se produit aussi si le minimum_access_length ne se tient pas entre 6 et 20 inclus, ou si state_retention_priority est 65535.

9.4.7 STATE-FREE

L'instruction STATE-FREE informe le point d'extrémité receveur que l'envoyeur ne souhaite plus utiliser un élément d'état particulier.

STATE-FREE (%partial_identifier_start, %partial_identifier_length)

Noter que l'instruction STATE-FREE ne supprime pas automatiquement un élément d'état, mais réclame plutôt la mémoire prise par l'élément d'état au sein d'un certain compartiment, qui n'est généralement pas connu avant d'atteindre l'instruction END-MESSAGE. Ainsi, tout comme pour l'instruction STATE-CREATE, lorsque une instruction STATE-FREE est rencontrée, l'UDVM met simplement en mémoire tampon les deux opérandes fournis jusqu'à ce que soit atteinte l'instruction END-MESSAGE. Les étapes suivies à ce moment sont décrites au paragraphe 9.4.9.

Un échec de décompression DOIT survenir si plus de quatre demandes sans état sont faites avant que soit rencontrée l'instruction END-MESSAGE. Un échec de décompression survient aussi si partial_identifier_length ne se tient pas entre 6 et 20 inclus.

9.4.8 OUTPUT

L'instruction OUTPUT fournit des données décompressées avec succès à l'expéditeur.

OUTPUT (%output_start, %output_length)

Les opérandes définissent l'adresse mémoire de début et la longueur de la chaîne d'octets à fournir à l'expéditeur. Noter que l'instruction OUTPUT peut être utilisée pour sortir un message partiellement décompressé ; chaque fois que l'instruction est rencontrée, elle fournit une nouvelle chaîne d'octets que l'expéditeur ajoute à la fin de tous octets passés précédemment à l'expéditeur via l'instruction OUTPUT.

La chaîne de données est copiée octet par octet à partir de la mémoire d'UDVM en obéissant aux règles du paragraphe 8.4.

Un échec de décompression survient si le nombre cumulé d'octets fourni à l'expéditeur excède 65 536 octets.

Comme il y a une différence technique entre sortir un message décompressé de zéro octet, et ne pas sortir du tout de message décompressé, l'instruction OUTPUT a besoin de distinguer les deux cas. Et donc, si l'UDVM termine avant de rencontrer une instruction OUTPUT, elle est considérée comme n'ayant pas sorti de message décompressé. Si elle rencontre une ou plusieurs instructions OUTPUT, dont chacune fournit 0 octet de données à l'expéditeur, elle est alors considérée comme ayant sorti un message décompressé de zéro octet.

9.4.9 END-MESSAGE

L'instruction END-MESSAGE termine avec succès l'UDVM et transmet les demandes de création d'état et les demandes sans état au gardien d'état avec toutes les données de rétroaction fournies.

END-MESSAGE (%requested_feedback_location, %returned_parameters_location, %state_length, %state_address, %state_instruction, %minimum_access_length, %state_retention_priority)

Lorsque l'instruction END-MESSAGE est rencontrée, le décompresseur expéditeur indique à l'application qu'un message complet a été décompressé. L'application peut retourner un identifiant de compartiment, que l'UDVM transmet au gardien d'état avec les demandes de création d'état et les demandes sans état et toutes les données de rétroaction fournies.

Le message décompressé réel est sorti séparément en utilisant l'instruction OUTPUT ; cela conserve la mémoire chez l'UDVM parce qu'il n'y a pas besoin de mettre un message décompressé entier en mémoire tampon avant qu'il puisse être passé à l'expéditeur.

L'instruction END-MESSAGE peut passer jusqu'à quatre demandes de création d'état et jusqu'à quatre demandes sans état au gardien d'état. Les demandes sont passées au gardien d'état dans l'ordre dans lequel elles ont été faites ; en particulier, il est possible que les demandes de création d'état et les demandes sans état soit entrelacées.

Les demandes de création d'état sont faites par l'instruction STATE-CREATE. Noter cependant que l'instruction END-MESSAGE peut faire elle-même une demande de création d'état en utilisant les opérandes fournis. Si la minimum_access_length spécifiée ne se tient pas entre 6 et 20 inclus, ou si la state_retention_priority est de 65535, l'instruction END-MESSAGE échoue alors à faire une demande de création d'état de son propre chef (cependant, un échec de décompression n'intervient pas et les demandes de création d'état faites par l'instruction STATE-CREATE restent valides).

Noter qu'il y a une limite maximum de quatre demandes de création d'état par instance de l'UDVM. Donc, un échec de décompression survient si l'instruction END-MESSAGE fait une demande de création d'état et que quatre instances de l'instruction STATE-CREATE ont déjà été rencontrées.

Lors de la création d'un élément d'état, il est nécessaire de donner state_length, state_address, state_instruction et minimum_access_length ; ils sont fournis comme opérandes dans l'instruction STATE-CREATE (ou l'instruction END-MESSAGE). Un élément d'état complet exige aussi une state_value et un state_identifier, qui sont déduits comme suit :

L'octet UDVM copie une chaîne d'octets state_length de la mémoire UDVM commençant à state_address (en respectant les règles du paragraphe 8.4). C'est la state_value.

L'UDVM calcule alors un hachage SHA-1 de 20 octets [RFC3174] sur la chaîne d'octets formée de l'enchaînement de state_length, state_address, state_instruction, minimum_access_length et state_value (dans cet ordre). C'est le state_identifier.

La state_retention_priority ne fait pas partie de l'élément d'état lui-même, mais détermine plutôt l'ordre dans lequel l'état sera supprimé lorsque le compartiment excède sa mémoire d'état allouée. La state_retention_priority est fournie comme opérande dans l'instruction STATE-CREATE ou END-MESSAGE et est passée au gardien d'état au titre de chaque demande de création d'état.

Les demandes sans état sont faites par l'instruction STATE-FREE. Chaque instruction STATE-FREE fournit les valeurs partial_identifier_start et partial_identifier_length ; lorsque l'instruction END-MESSAGE est atteinte, ces valeurs sont utilisées pour copier les octets d'un identifiant d'état partiel à partir de la mémoire d'UDVM. Si aucun élément d'état correspondant à l'identifiant d'état partiel ne peut être trouvé ou si plus d'un élément d'état dans le compartiment correspond à l'identifiant d'état partiel, la demande sans état est alors ignorée (cela ne cause pas d'échec de décompression). Autrement, le gardien d'état libère l'élément d'état correspondant comme spécifié au paragraphe 6.2.

Aussi bien que la transmission des demandes de création d'état et des demandes sans état, l'instruction END-MESSAGE peut aussi passer des données de rétroaction au gardien d'état. Les données de rétroaction sont utilisées pour informer le point d'extrémité receveur des capacités du point d'extrémité envoyeur, qui peuvent aider à améliorer le taux de compression global et réduire les exigences de mémoire de travail des points d'extrémité.

Deux types de données de rétroaction sont disponibles : les rétroactions demandées et les rétroactions retournées. Le format des données de rétroaction demandées est donné à la Figure 12. Comme mentionné au paragraphe 3.2, les données de rétroaction demandées peuvent être utilisées pour influencer le contenu des données de rétroaction retournées dans la direction inverse.

Les données de rétroaction retournées sont elles-mêmes subdivisées en un élément de rétroaction retourné et en une liste de paramètres SigComp retournés. L'élément de rétroaction retourné est d'une importance suffisante pour garantir son propre champ dans l'en-tête SigComp comme décrit au paragraphe 7.1. Les paramètres SigComp retournés sont illustrés à la Figure 13.

Noter que les formats des figures 12 et 13 sont seulement pour la présentation locale des données de rétroaction sur l'interface entre l'UDVM et le gardien d'état. Les formats ne rendent obligatoire aucun bit sur le réseau ; le compresseur peut transmettre les données sous toute forme pourvu qu'elle soit chargée dans la mémoire d'UDVM à l'adresse correcte.

De plus, la responsabilité de s'assurer de la bonne arrivée des données de rétroaction sur un transport non fiable incombe à l'envoyeur. Le point d'extrémité receveur utilise toujours la dernière valeur reçue pour chaque champ dans les données de rétroaction, même si les valeurs sont périmées à cause d'une perte de paquet ou d'un déclassement.

Si l'opérande requested_feedback_location est à 0, aucune demande de rétroaction n'est faite ; autrement, il pointe sur l'adresse mémoire de début des données de rétroaction demandées comme le montre la Figure 12.

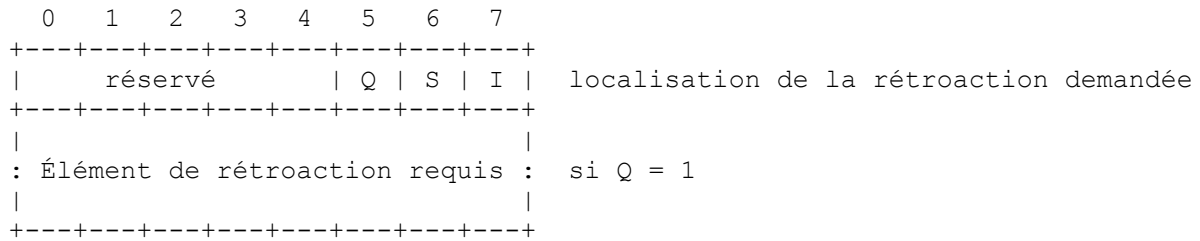


Figure 12 : Format des données de rétroaction demandées

Les bits réservés peuvent être utilisés dans de futures versions de SigComp, et sont réglés à 0 dans la version 0x01. Les valeurs non à zéro devraient être ignorées par le point d'extrémité receveur.

Le bit Q indique si un élément de rétroaction demandé est présent ou non. Le compresseur peut régler l'élément de rétroaction demandé à une valeur arbitraire, qui sera ensuite transmise sans modification dans la direction inverse comme un élément de rétroaction retourné. Voir à la Section 5 les détails de la façon dont l'élément de rétroaction demandé est retourné.

Le format de l'élément de rétroaction demandé est identique à celui de l'élément de rétroaction retourné illustré à la Figure 4.

Le compresseur règle le bit S à 1 si il ne souhaite pas (ou ne souhaite plus) sauvegarder les informations d'état au point d'extrémité receveur et aussi ne souhaite pas accéder aux informations d'état qu'il sauvegardait précédemment. Par

conséquent, si le bit S est envoyé à 1, le point d'extrémité receveur peut réclamer la mémoire d'état allouée au compresseur distant et régler taille_de_mémoire_d'état pour le compartiment à 0.

Le compresseur peut changer d'avis et ramener le bit S à 0 dans un message ultérieur. Cependant, le point d'extrémité receveur n'est pas obligé d'utiliser le taille_de_mémoire_d'état original pour le compartiment ; il peut choisir d'allouer moins de mémoire au compartiment ou éventuellement pas du tout.

De même, le compresseur règle le bit I à 1 si il ne souhaite pas (ou plus) accéder à un des éléments d'état disponibles en local offerts par le point d'extrémité receveur. Cela peut aider à conserver de la bande passante parce que la liste des éléments d'état disponibles en local n'a plus besoin d'être retournée dans la direction inverse. Il peut aussi conserver la mémoire au point d'extrémité receveur, car le gardien d'état peut supprimer tout élément d'état détectable en local qu'il détermine comme n'étant plus exigé par un point d'extrémité distant. Noter que le compresseur peut régler le bit I à nouveau à 0 dans un message ultérieur, mais il ne peut pas accéder à des éléments d'état disponibles en local qui étaient précédemment offerts par le point d'extrémité receveur sauf si ils sont ré-annoncés ensuite.

Si l'opérande localisation_des_paramètres_retournés est réglé à 0, aucun paramètre SigComp n'est alors retourné ; autrement, il pointe sur l'adresse mémoire de début des paramètres retournés comme le montre la Figure 13.

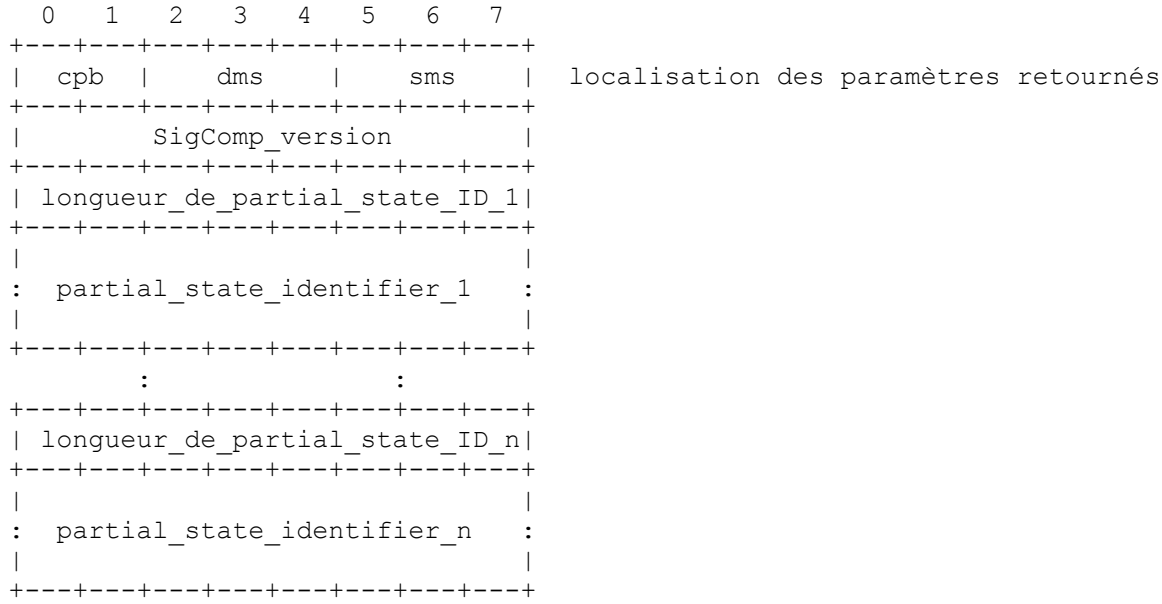


Figure 13 : Format des paramètres SigComp retournés

Le premier octet code les paramètres cycles_par_bit, taille_de_mémoire_de_décompression et taille_de_mémoire_d'état selon le paragraphe 3.3.1. L'octet peut être à 0 si les trois paramètres ne sont pas inclus dans les données de rétroaction. (Cela peut être utile pour économiser des bits dans le message compressé si le point d'extrémité distant est déjà satisfait que toutes les informations nécessaires aient atteint le point d'extrémité receveur du message.)

Le second octet code le SigComp_version selon le paragraphe 3.3.2. De même que le premier octet, le second octet peut être à 0 si le paramètre n'est pas inclus dans les données de rétroaction.

Les octets restants codent une liste d'identifiants d'état partiel pour les éléments d'état disponibles en local offerts par le point d'extrémité envoyeur. Chaque élément d'état est codé comme un champ de un octet, suivi par un identifiant d'état partiel qui contient autant d'octets qu'indiqué dans le champ Longueur. L'envoyeur peut choisir de n'envoyer que six octets si il pense que c'est suffisant pour que le receveur détermine quel élément d'état est offert.

La liste des identifiants d'état se termine par un octet dans la position où on attendrait le prochain champ Longueur, qui est réglé à une valeur inférieure à 6 ou supérieure à 20. Noter que des versions améliorées de SigComp pourront ajouter des éléments de données supplémentaires après le champ Longueur final.

10. Considérations pour la sécurité

10.1. Objectifs de sécurité

L'objectif de sécurité globale de l'architecture SigComp n'est pas de créer des risques qui s'ajoutent à ceux déjà présents dans les protocoles d'application. SigComp n'a aucune intention d'améliorer la sécurité de l'application, car elle peut toujours être circonvenue en n'utilisant pas la compression. Plus précisément, les objectifs de sécurité de haut niveau peuvent être décrits comme :

1. ne pas dégrader la sécurité du protocole d'application existant,
2. ne pas créer de nouveaux problèmes de sécurité,
3. ne pas entraver le déploiement de la sécurité d'application.

10.2 Risques pour la sécurité et contre mesures

Ce paragraphe identifie les risques potentiels pour la sécurité associés à SigComp, et explique comment chaque risque est minimisé par le schéma.

10.2.1 Risques pour la confidentialité

- Attaquer SigComp en furetant dans les états des autres utilisateurs : on accède à l'état en fournissant un identifiant d'état, qui est un hachage cryptographique de l'état référencé. Cela implique que le message de référence doit déjà connaître l'état. Pour appliquer cela, on ne peut pas accéder à un élément d'état sans fournir un minimum de 48 bits du hachage. Cela minimise aussi la probabilité d'une collision accidentelle d'état. Un compresseur peut, en utilisant l'opérande minimum_access_length des instructions STATE-CREATE et END-MESSAGE, augmenter le nombre de bits qui doivent être fournis pour accéder à l'état, augmentant la protection contre les attaques.

Généralement, le moyen d'obtenir la connaissance de l'identifiant d'état (par exemple, une attaque passive) donnera aussi facilement la connaissance de l'état référencé, de sorte qu'il n'en résulte pas de nouvelle faiblesse.

Un point d'extrémité doit traiter les identifiants d'état avec le même soin que l'état lui-même.

10.2.2 Risques pour l'intégrité

L'approche SigComp suppose qu'il y a une protection d'intégrité appropriée en dessous et/ou au dessus de la couche SigComp. Le mécanisme de création d'état fournit un potentiel de compromission supplémentaire de l'intégrité des messages ; cependant, cela sera très probablement détectable à la couche application.

- Attaquer SigComp en falsifiant l'état ou en faisant des changements non autorisés à l'état :
  L'état ne peut être détruit par un envoyeur malveillant que si il peut envoyer des messages que l'application identifie comme appartenant au même compartiment que celui sous lequel l'état a été créé ; cela ajoute des risques supplémentaires pour la sécurité lorsque l'application permet l'installation de l'état SigComp à partir d'un message où il n'aurait pas lui-même installé d'état.

Falsifier ou changer l'état n'est possible que si le hachage permet des collisions intentionnelles.

10.2.3 Risques de disponibilité (éviter la vulnérabilité aux attaques de déni de service)

- L'utilisation de SigComp comme outil pour une attaque de déni de service sur une autre cible : SigComp ne peut pas être facilement utilisé comme amplificateur d'une attaque par réflexion car il ne génère qu'un message décompressé par message compressé entrant. Ce message est ensuite passé à l'application ; l'utilité comme amplificateur de réflexion est donc limitée par l'utilité de l'application pour cet objet.

Cependant, on doit noter que SigComp peut être utilisé pour générer de plus grands messages comme entrée à l'application que ce qui doit être envoyé de l'envoyeur malveillant ; celui-ci peut donc envoyer de plus petits messages (avec une bande passante inférieure) que ce qui est livré à l'application. Selon les caractéristiques de réflexion de l'application, cela peut être considéré comme une forme adoucie d'amplification. L'application DOIT limiter le nombre de paquets réfléchis sur une cible potentielle - même si SigComp est utilisé pour générer une grande quantité d'informations à partir d'un petit paquet d'attaque entrant.

- Attaquer SigComp comme cible du déni de service en le remplissant d'état : un état excessif ne peut être installé par un envoyeur malveillant (ou un ensemble d'envoyeurs malveillants) qu'avec le consentement de l'application. Le système consistant en SigComp et l'application est donc approximativement aussi vulnérable que l'application elle-même, sauf

si il permet l'installation de l'état SigComp à partir d'un message où il n'aurait pas lui-même installé l'état d'application.

Si ceci est désirable pour augmenter le taux de compression, l'effet peut être atténué en faisant usage de rétroactions au niveau de l'application qui indiquent si l'état demandé a été effectivement installé – cela permet au système soumis à une attaque de dégrader en douceur en n'installant plus l'état de compresseur qui ne correspond plus à l'état de l'application.

Évidemment, si on utilise un transport fondé sur le flux, les flux eux-mêmes constituent l'état qui doit être traité de la même façon que l'application elle-même traiterait un transport fondé sur le flux ; si une application n'est pas équipée pour le transport fondé sur le flux, elle ne devrait pas permettre de connexions SigComp sur un transport fondé sur le flux. Pour l'autre usage de SigComp décrit comme "mode continu" au paragraphe 4.2.1, un attaquant pourrait créer un nombre quelconque d'UDVM sauf si il y a une protection contre le déni de service à un niveau inférieur (par exemple, en utilisant TLS dans les configurations appropriées).

- Attaquer l'UDVM en falsifiant l'état ou en faisant des changements non autorisés à l'état :

Ceci est couvert au paragraphe 10.2.2.

- Attaquer l'UDVM en lui envoyant un code de mise en boucle : l'application met une limite supérieure au nombre de "cycles d'UDVM" qui peuvent être utilisés par message compressé et par bit entré dans le message compressé. Le dommage infligé par l'envoi de paquets avec du code de mise en boucle est donc limité, bien que cela puisse quand même être substantiel si un grand nombre de cycles d'UDVM sont offerts par l'UDVM. Cependant, cela serait vrai pour tout décompresseur qui peut recevoir des paquets sur un transport non sûr.

11. Considérations relatives à l'IANA

SigComp exige un espace de nom d'un octet, SigComp_version, qui a été créé par l'IANA. Les nouvelles versions de SigComp doivent être rétro compatibles avec la version 0x01, décrite dans le présent document. Ajouter des instructions d'UDVM supplémentaires et allouer des valeurs aux adresses de mémoire d'UDVM réservées sont deux mises à niveau possibles pour lesquelles c'est le cas.

Suivant les politiques mentionnées dans la [RFC2434], la politique de l'IANA pour allouer une nouvelle valeur à SigComp_version exigera une action de normalisation. Les valeurs ne sont donc allouées que par des RFC sur la voie de la normalisation approuvées par l'IESG.

12. Remerciements

Merci à Abigail Surtees, Mark A West, Lawrence Conroy, Christian Schmidt, Max Riegel, Lars-Erik Jonsson, Stefan Forsgren, Krister Svanbro, Miguel Garcia, Christopher Clanton, Khiem Le, Ka Cheong Leung, et Robert Sugar pour leurs apports précieux et leur relecture.

13. Références

13.1 Références normatives

[RFC1662] W. Simpson, éditeur, "PPP en trames de style HDLC", STD 51, juillet 1994. *(Remplace la* RFC1549)

[RFC2119] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", BCP 14, mars 1997.

[RFC3174] D. Eastlake 3 et P. Jones, "Algorithme US de hachage sécurisé n° 1 (SHA1)", sept. 2001. *(Info, MàJ par* 4634 *et* 6234*)*

13.2 Références pour information

[RFC1951] P. Deutsch, "Spécification du format DEFLATE de données compressées, version 1.3", mai 1996.

[RFC2026] S. Bradner, "Le processus de normalisation de l'Internet -- Révision 3", (BCP0009) octobre 1996. *(Remplace* RFC1602, RFC1871) (*MàJ par* RFC3667, RFC3668, RFC3932, RFC3979, RFC3978, RFC5378, RFC6410)

[RFC2279] F. Yergeau, "UTF-8, un format de transformation de la norme ISO 10646", janvier 1998. (*Obsolète, voir* RFC3629) *(D.S.)*

[RFC2326] H. Schulzrinne, A. Rao et R. Lanphier, "Protocole de flux directs en temps réel (RTSP)", avril 1998.

[RFC2434] T. Narten et H. Alvestrand, "Lignes directrices pour la rédaction d'une section Considérations relatives à l'IANA dans les RFC", BCP 26, octobre, 1998. (*Rendue obsolète par la* RFC5226)

[RFC2960] R. Stewart et autres, "Protocole de transmission de commandes de flux", octobre 2000. (*Obsolète, voir* RFC4960) (*MàJ par* RFC3309) *(P.S.)*

[RFC3261] J. Rosenberg et autres, "SIP : Protocole d'initialisation de session", juin 2002. (*Mise à jour par* RFC3265, RFC3853, RFC4320, RFC4916, RFC5393, RFC6665)

[RFC3321] H. Hannu et autres, "Compression de signalisation (SigComp) - Opérations d'extension", janvier 2003. (*MàJ par* RFC4896) *(P.S.)*

14. Adresse des auteurs

Richard Price
Roke Manor Research Ltd
Romsey, Hants, SO51 0ZN
United Kingdom
téléphone : +44 1794 833681
mél : richard.price@roke.co.uk

Carsten Bormann
Universitaet Bremen TZI
Postfach 330440
D-28334 Bremen, Germany
téléphone : +49 421 218 7024
mél : cabo@tzi.org

Jan Christoffersson
Box 920
Ericsson AB
SE-971 28 Lulea, Sweden
téléphone : +46 920 20 28 40
mél : jan.christoffersson@epl.ericsson.se

Hans Hannu
Box 920
Ericsson AB
SE-971 28 Lulea, Sweden
téléphone : +46 920 20 21 84
mél : hans.hannu@epl.ericsson.se

Zhigang Liu
Nokia Research Center
6000 Connection Drive
Irving, TX 75039
téléphone : +1 972 894-5935
mél : zhigang.c.liu@nokia.com

Jonathan Rosenberg
dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936
mél : jdrosen@dynamicsoft.com

15. Déclaration complète de droits de reproduction

Remerciement
Le financement de la fonction d'édition des RFC est actuellement fourni par l'Internet Society.