Network Working Group
Request for Comments: 3320
Category: Standards Track

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

January 2003

Signaling Compression (SigComp)

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document defines Signaling Compression (SigComp), a solution for compressing messages generated by application protocols such as the Session Initiation Protocol (SIP) (RFC 3261) and the Real Time Streaming Protocol (RTSP) (RFC 2326). The architecture and prerequisites of SigComp are outlined, along with the format of the SigComp message.

Decompression functionality for SigComp is provided by a Universal Decompressor Virtual Machine (UDVM) optimized for the task of running decompression algorithms. The UDVM can be configured to understand the output of many well-known compressors such as DEFLATE (RFC-1951).

Table of Contents

1. Introduction

Many application protocols used for multimedia communications are text-based and engineered for bandwidth rich links. As a result the messages have not been optimized in terms of size. For example, typical SIP messages range from a few hundred bytes up to two thousand bytes or more [RFC3261].

With the planned usage of these protocols in wireless handsets as part of 2.5G and 3G cellular networks, the large message size is problematic. With low-rate IP connectivity the transmission delays are significant. Taking into account retransmissions, and the multiplicity of messages that are required in some flows, call setup and feature invocation are adversely affected. SigComp provides a means to eliminate this problem by offering robust, lossless compression of application messages.

This document outlines the architecture and prerequisites of the SigComp solution, the format of the SigComp message and the Universal Decompressor Virtual Machine (UDVM) that provides decompression functionality.

SigComp is offered to applications as a layer between the application and an underlying transport. The service provided is that of the underlying transport plus compression. SigComp supports a wide range of transports including TCP, UDP and SCTP [RFC-2960].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC-2119].

Application

Entity that invokes SigComp and performs the following tasks:

1. Supplying application messages to the compressor dispatcher
2. Receiving decompressed messages from the decompressor dispatcher
3. Determining the compartment identifier for a decompressed message.

Bytecode

Machine code that can be executed by a virtual machine.

Compressor

Entity that encodes application messages using a certain compression algorithm, and keeps track of state that can be used for compression. The compressor is responsible for ensuring that the messages it generates can be decompressed by the remote UDVM.

Compressor Dispatcher

Entity that receives application messages, invokes a compressor, and forwards the resulting SigComp compressed messages to a remote endpoint.

UDVM Cycles

A measure of the amount of "CPU power" required to execute a UDVM instruction (the simplest UDVM instructions require a single UDVM cycle). An upper limit is placed on the number of UDVM cycles that can be used to decompress each bit in a SigComp message.

Decompressor Dispatcher

Entity that receives SigComp messages, invokes a UDVM, and forwards the resulting decompressed messages to the application.

Endpoint

One instance of an application, a SigComp layer, and a transport layer for sending and/or receiving SigComp messages.

Message-based Transport

A transport that carries data as a set of bounded messages.

Compartment

An application-specific grouping of messages that relate to a peer endpoint. Depending on the signaling protocol, this grouping may relate to application concepts such as "session", "dialog", "connection", or "association". The application allocates state memory on a per-compartment basis, and determines when a compartment should be created or closed.

Compartment Identifier

An identifier (in a locally chosen format) that uniquely references a compartment.

SigComp

The overall compression solution, comprising the compressor, UDVM, dispatchers and state handler.

SigComp Message

A message sent from the compressor dispatcher to the decompressor dispatcher. In case of a message-based transport such as UDP, a SigComp message corresponds to exactly one datagram. For a stream-based transport such as TCP, the SigComp messages are separated by reserved delimiters.

Stream-based transport

A transport that carries data as a continuous stream with no message boundaries.

Transport

Mechanism for passing data between two endpoints. SigComp is capable of sending messages over a wide range of transports including TCP, UDP and SCTP [RFC-2960].

Universal Decompressor Virtual Machine (UDVM)

The machine architecture described in this document. The UDVM is used to decompress SigComp messages.

State

Data saved for retrieval by later SigComp messages.

State Handler

Entity responsible for accessing and storing state information once permission is granted by the application.

State Identifier

Reference used to access a previously created item of state.

3. SigComp Architecture

In the SigComp architecture, compression and decompression is performed at two communicating endpoints. The layout of a single endpoint is illustrated in Figure 1:

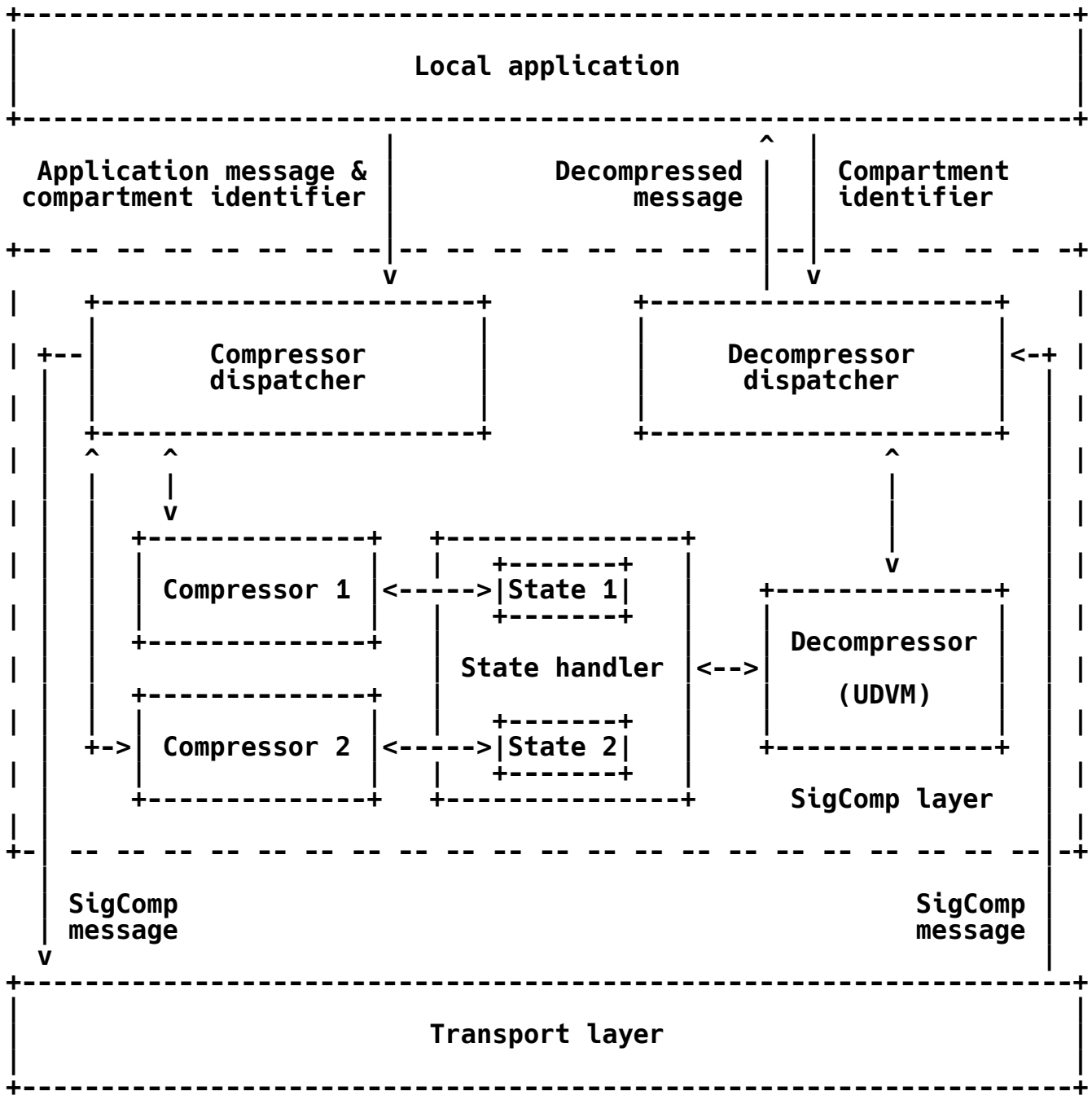


Figure 1: High-level architectural overview of one SigComp endpoint

Note that SigComp is offered to applications as a layer between the application and the underlying transport, and so Figure 1 is an endpoint when viewed from a transport layer perspective. From the perspective of multi-hop application layer protocols however, SigComp is applied on a per-hop basis.

The SigComp layer is further decomposed into the following entities:

1. Compressor dispatcher - the interface from the application. The application supplies the compressor dispatcher with an application message and a compartment identifier (see Section 3.1 for further details). The compressor dispatcher invokes a particular compressor, which returns a SigComp message to be forwarded to the remote endpoint.

2. Decompressor dispatcher - the interface towards the application. The decompressor dispatcher receives a SigComp message and invokes an instance of the Universal Decompressor Virtual Machine (UDVM). It then forwards the resulting decompressed message to the application, which may return a compartment identifier if it wishes to allow state to be saved for the message.

3. One or more compressors - the entities that convert application messages into SigComp messages. Distinct compressors are invoked on a per-compartment basis, using the compartment identifiers supplied by the application. A compressor receives an application message from the compressor dispatcher, compresses the message, and returns a SigComp message to the compressor dispatcher. Each compressor chooses a certain algorithm to encode the data (e.g., DEFLATE).

4. UDVM - the entity that decompresses SigComp messages. Note that since SigComp can run over an unsecured transport layer, a separate instance of the UDVM is invoked on a per-message basis. However, during the decompression process the UDVM may invoke the state handler to access existing state or create new state.

5. State handler - the entity that can store and retrieve state. State is information that is stored between SigComp messages, avoiding the need to upload the data on a per-message basis. For security purposes it is only possible to create new state with the permission of the application. State creation and retrieval are further described in Chapter 6.

When compressing a bidirectional application protocol the choice to use SigComp can be made independently in both directions, and compression in one direction does not necessarily imply compression in the reverse direction. Moreover, even when two communicating endpoints send SigComp messages in both directions, there is no need to use the same compression algorithm in each direction.

Note that a SigComp endpoint can decompress messages from multiple remote endpoints at different locations in a network, as the architecture is designed to prevent SigComp messages from one endpoint interfering with messages from a different endpoint. A consequence of this design choice is that it is difficult for a malicious user to disrupt SigComp operation by inserting false compressed messages on the transport layer.

3.1. Requirements on the Application

From an application perspective the SigComp layer appears as a new transport, with similar behavior to the original transport used to carry uncompressed data (for example SigComp/UDP behaves similarly to native UDP).

Mechanisms for discovering whether an endpoint supports SigComp are beyond the scope of this document.

All SigComp messages contain a prefix (the five most-significant bits of the first byte are set to one) that does not occur in UTF-8 encoded text messages [RFC-2279], so for applications which use this encoding (or ASCII encoding) it is possible to multiplex uncompressed application messages and SigComp messages on the same port. Applications can still reserve a new port specifically for SigComp however (e.g., as part of the discovery mechanism).

If a particular endpoint wishes to be stateful then it needs to partition its decompressed messages into "compartments" under which state can be saved. SigComp relies on the application to provide this partition. So for stateful endpoints a new interface is required to the application in order to leverage the authentication mechanisms used by the application itself.

When the application receives a decompressed message it maps the message to a certain compartment and supplies the compartment identifier to SigComp. Each compartment is allocated a separate compressor and a certain amount of memory to store state information, so the application must assign distinct compartments to distinct remote endpoints. However it is possible for a local endpoint to establish several compartments that relate to the same remote endpoint (this should be avoided where possible as it may waste

memory and reduce the overall compression ratio, but it does not cause messages to be incorrectly decompressed). In this case, reliable stateful operation is possible only if the decompressor does not lump several messages into one compartment when the compressor expected them to be assigned different compartments.

The exact format of the compartment identifier is unimportant provided that different identifiers are given to different compartments.

Applications that wish to communicate using SigComp in a stateful fashion should use an authentication mechanism to securely map decompressed messages to compartment identifiers. They should also agree on any limits to the lifetime of a compartment, to avoid the case where an endpoint accesses state information that has already been deleted.

3.2. SigComp feedback mechanism

If a signaling protocol sends SigComp messages in both directions and there is a one-to-one relationship between the compartments established by the applications on both ends ("peer compartments"), the two endpoints can cooperate more closely. In this case, it is possible to send feedback information that monitors the behavior of an endpoint and helps to improve the overall compression ratio. SigComp performs feedback on a request/response basis, so a compressor makes a feedback request and receives some feedback data in return. The procedure for requesting and returning feedback in SigComp is illustrated in Figure 2:

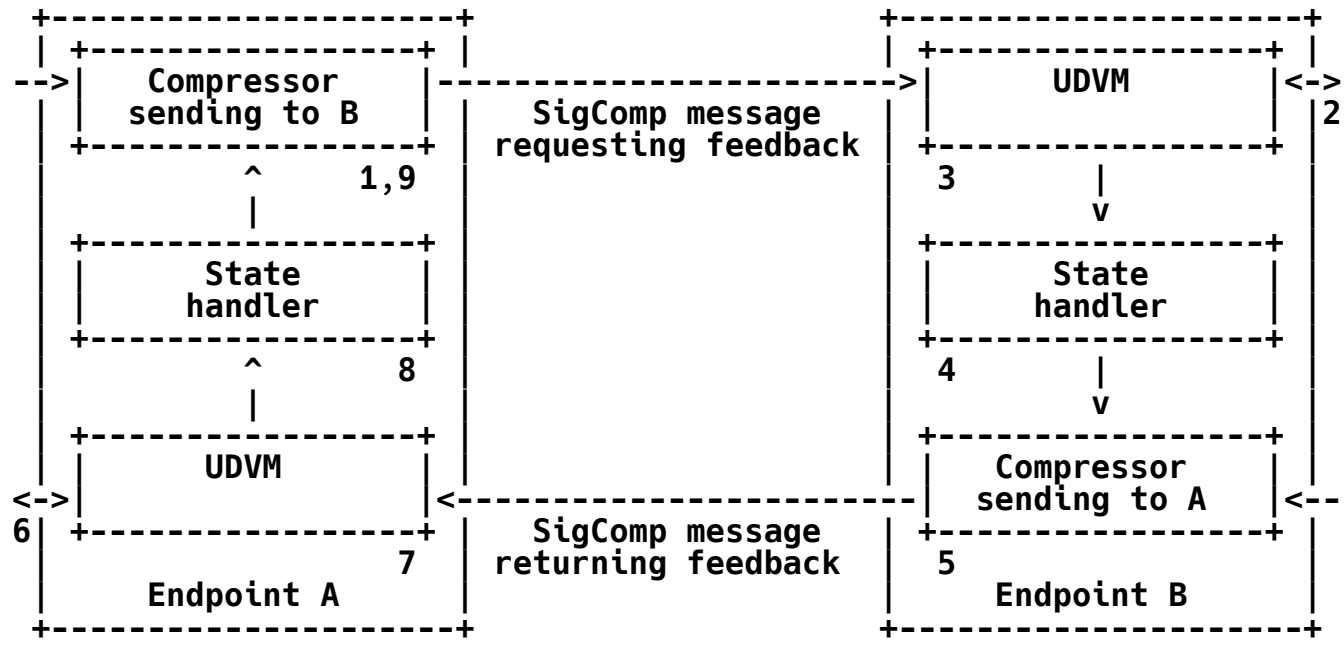


Figure 2: Steps involved in the transmission of feedback data

The dispatchers, the application and the transport layer are omitted from the diagram for clarity. Note that the decompressed messages pass via the decompressor dispatcher to the application; moreover the SigComp messages transmitted from the compressor to the remote UDVM are sent via first the compressor dispatcher, followed by the transport layer and finally the decompressor dispatcher.

The steps for requesting and returning feedback data are described in more detail below:

1. The compressor that sends messages to Endpoint B piggybacks a feedback request onto a SigComp message.

2. When the application receives the decompressed message, it may return the compartment identifier for the message.

3. The UDVM in Endpoint B forwards the requested feedback data to the state handler.

4. If the UDVM can supply a valid compartment identifier, then the state handler forwards the feedback data to the appropriate compressor (namely the compressor sending to Endpoint A).

5. The compressor returns the requested feedback data to Endpoint A piggybacked onto a SigComp message.

6. When the application receives the decompressed message, it may return the compartment identifier for the message.

7. The UDVM in Endpoint A forwards the returned feedback data to the state handler.

8. If the UDVM can supply a valid compartment identifier, then the state handler forwards the feedback data to the appropriate compressor (namely the compressor sending to Endpoint B).

9. The compressor makes use of the returned feedback data.

The detailed role played by each entity in the transmission of feedback data is explained in subsequent chapters.

3.3. SigComp Parameters

An advantage of using a virtual machine for decompression is that almost all of the implementation flexibility lies in the SigComp compressors. When receiving SigComp messages an endpoint generally behaves in a predictable manner.

Note however that endpoints implementing SigComp will typically have a wide range of capabilities, each offering a different amount of working memory, processing power etc. In order to support this wide variation in endpoint capabilities, the following parameters are provided to modify SigComp behavior when receiving SigComp messages:

decompression_memory_size
state_memory_size
cycles_per_bit
SigComp_version
locally available state (a set containing 0 or more state items)

Each parameter has a minimum value that MUST be offered by all receiving SigComp endpoints. Moreover, endpoints MAY offer additional resources if available; these resources can be advertised to remote endpoints using the SigComp feedback mechanism.

Particular applications may also agree a-priori to offer additional resources as mandatory (e.g., SigComp for SIP offers a dictionary of common SIP phrases as a mandatory state item).

Each of the SigComp parameters is described in greater detail below.

3.3.1. Memory Size and UDVM Cycles

The decompression_memory_size parameter specifies the amount of memory available to decompress one SigComp message. (Note that the term "amount of memory" is used on a conceptual level in order to specify decompressor behavior and allow resource planning on the side of the compressor -- an implementation could require additional, bounded amounts of actual memory resources or could even organize its memory in a completely different way as long as this does not cause decompression failures where the conceptual model would not.) A portion of this memory is used to buffer a SigComp message before it is decompressed; the remainder is given to the UDVM. Note that the memory is allocated on a per-message basis and can be reclaimed after the message has been decompressed. All endpoints implementing SigComp MUST offer a decompression_memory_size of at least 2048 bytes.

The state_memory_size parameter specifies the number of bytes offered to a particular compartment for the creation of state. This parameter is set to 0 if the endpoint is stateless.

Unlike the other SigComp parameters, the state_memory_size is offered on a per-compartment basis and may vary for different compartments. The memory for a compartment is reclaimed when the application determines that the compartment is no longer required.

The cycles_per_bit parameter specifies the number of "UDVM cycles" available to decompress each bit in a SigComp message. Executing a UDVM instruction requires a certain number of UDVM cycles; a complete list of UDVM instructions and their cost in UDVM cycles can be found in Chapter 9. An endpoint MUST offer a minimum of 16 cycles_per_bit.

Each of the three parameter values MUST be chosen from the limited set given below, so that the parameters can be efficiently encoded for transmission using the SigComp feedback mechanism.

The cycles_per_bit parameter is encoded using 2 bits, whilst the decompression_memory_size and state_memory_size are both encoded using 3 bits. The bit encodings and their corresponding values are as follows:

Encoding:	cycles_per_bit:	Encoding:	state_memory_size (bytes):
00	16	000	0
01	32	001	2048
10	64	010	4096
11	128	011	8192
		100	16384
		101	32768
		110	65536
		111	131072

The decompression_memory_size is encoded in the same manner as the state_memory_size, except that the bit pattern 000 cannot be used (as an endpoint cannot offer a decompression_memory_size of 0 bytes).

3.3.2. SigComp Version

The SigComp_version parameter specifies whether only the basic version of SigComp is available, or whether an upgraded version is available offering additional instructions etc. Within the UDVM, it is available as a 2-byte value, generated by zero-extending the 1-byte SigComp_version parameter (i.e., the first byte of the 2-byte value is always zero).

The basic version of SigComp is Version 0x01, which is the version described in this document.

To ensure backwards compatibility, if a SigComp message is successfully decompressed by Version 0x01 of SigComp then it will be successfully decompressed on upgraded versions. Similarly, if the message triggers a manual decompression failure (see Section 8.7), then it will also continue to do so.

However, messages that cause an unexpected decompression failure on Version 0x01 of SigComp may be successfully decompressed by upgraded versions.

The simplest way to upgrade SigComp in a backwards-compatible manner is to add additional UDVM instructions, as this will not affect the decompression of SigComp messages compatible with Version 0x01. Reserved addresses in the UDVM memory (Useful Values, see Section 7.2) may also be assigned values in future versions of SigComp.

3.3.3. Locally Available State Items

A SigComp state item is an item of data that is retained between SigComp messages. State items can be retrieved and loaded into the UDVM memory as part of the decompression process, often significantly improving the compression ratio as the same information does not have to be uploaded on a per-message basis.

Each endpoint maintains a set of state items where every item is composed of the following information:

Name:	Type of data:
state_identifier	20-byte value
state_length	2-byte value
state_address	2-byte value
state_instruction	2-byte value
minimum_access_length	2-byte value from 6 to 20 inclusive
state_value	String of state_length consecutive bytes

State items are typically created at an endpoint upon successful decompression of a SigComp message. The remote compressor sending the message makes a state creation request by invoking the appropriate UDVM instruction, and the state is saved once permission is granted by the application.

However, an endpoint MAY also wish to offer a set of locally available state items that have not been uploaded as part of a SigComp message. For example it might offer well-known decompression algorithms, dictionaries of common phrases used in a specific signaling protocol, etc.

Since these state items are established locally without input from a remote endpoint, they are most useful if publicly documented so that a wide collection of remote endpoints can determine the data contained in each state item and how it may be used. Further Internet Documents and RFCs may be published to describe particular locally available state items.

Although there are no locally available state items that are mandatory for every SigComp endpoint, certain state items can be made mandatory in a specific environment (e.g., the dictionary of common phrases for a specific signaling protocol could be made mandatory for that signaling protocol's usage of SigComp). Also, remote endpoints can indicate their interest in receiving a list of some of the state items available locally at an endpoint using the SigComp feedback mechanism.

It is a matter of local decision for an endpoint what items of locally available state it advertises; this decision has no influence on interoperability, but may increase or decrease the efficiency of the compression achievable between the endpoints.

4. SigComp Dispatchers

This chapter defines the behavior of the compressor and decompressor dispatcher. The function of these entities is to provide an interface between SigComp and its environment, minimizing the effort needed to integrate SigComp into an existing protocol stack.

4.1. Compressor Dispatcher

The compressor dispatcher receives messages from the application and passes the compressed version of each message to the transport layer.

Note that SigComp invokes compressors on a per-compartment basis, so when the application provides a message to be compressed it must also provide a compartment identifier. The compressor dispatcher forwards the application message to the correct compressor based on the compartment identifier (invoking a new compressor if a new compartment identifier is encountered). The compressor returns a SigComp message that can be passed to the transport layer.

Additionally, the application should indicate to the compressor dispatcher when it wishes to close a particular compartment, so that the resources taken by the corresponding compressor can be reclaimed.

4.2. Decompressor Dispatcher

The decompressor dispatcher receives messages from the transport layer and passes the decompressed version of each message to the application.

To ensure that SigComp can run over an unsecured transport layer, the decompressor dispatcher invokes a new instance of the UDVM for each new SigComp message. Resources for the UDVM are released as soon as the message has been decompressed.

The dispatcher MUST NOT make more than one SigComp message available to a given instance of the UDVM. In particular, the dispatcher MUST NOT concatenate two SigComp messages to form a single message.

4.2.1. Decompressor Dispatcher Strategies

Once the UDVM has been invoked it is initialized using the SigComp message of Chapter 7. The message is then decompressed by the UDVM, returned to the decompressor dispatcher, and passed on to the receiving application. Note that the UDVM has no awareness of whether the underlying transport is message-based or stream-based, and so it always outputs decompressed data as a stream. It is the responsibility of the dispatcher to provide the decompressed message to the application in the expected form (i.e., as a stream or as a distinct, bounded message). The dispatcher knows that the end of a decompressed message has been reached when the UDVM instruction END-MESSAGE is invoked (see Section 9.4.9).

For a stream-based transport, two strategies are therefore possible for the decompressor dispatcher:

1) The dispatcher collects a complete SigComp message and then invokes the UDVM. The advantage is that, even in implementations that have multiple incoming compressed streams, only one instance of the UDVM is ever required.

2) The dispatcher collects the SigComp header (see Section 7) and invokes the UDVM; the UDVM stays active while the rest of the message arrives. The advantage is that there is no need to buffer up the rest of the message; the message can be decompressed as it arrives, and any decompressed output can be relayed to the application immediately.

In general, which of the strategies is used is an implementation choice.

However, the compressor may want to take advantage of strategy 2 by expecting that some of the application message is passed on to the application before the SigComp message is terminated, e.g., by keeping the UDVM active while expecting the application to continuously receive decompressed output. This approach ("continuous mode") invalidates some assumptions of the SigComp security model and can only be used if the transport itself can provide the required protection against denial of service attacks. Also, since only strategy 2 works in this approach, the use of continuous mode requires previous agreement between the two endpoints.

4.2.2. Record Marking

For a stream-based transport, the dispatcher delimits messages by parsing the compressed data stream for instances of 0xFF and taking the following actions:

Occurs in data stream:	Action:
0xFF 00	one 0xFF byte in the data stream
0xFF 01	same, but the next byte is quoted (could be another 0xFF)
:	:
0xFF 7F	same, but the next 127 bytes are quoted
0xFF 80 to 0xFF FE	(reserved for future standardization)
0xFF FF	end of SigComp message

The combinations 0xFF01 to 0xFF7F are useful to limit the worst case expansion of the record marking scheme: the 1 (0xFF01) to 127 (0xFF7F) bytes following the byte combination are copied literally by the decompressor without taking any special action on 0xFF. (Note that 0xFF00 is just a special case of this, where zero following bytes are copied literally.)

In UDVM version 0x01, any occurrence of the combinations 0xFF80 to 0xFFFE that are not protected by quoting causes decompression failure; the decompressor SHOULD close the stream-based transport in this case.

4.3. Returning a Compartment Identifier

Upon receiving a decompressed message the application may supply the dispatcher with a compartment identifier. Supplying this identifier grants permission for the following:

1. Items of state accompanying the decompressed message can be saved using the state memory reserved for the specified compartment.

2. The feedback data accompanying the decompressed message can be trusted sufficiently that it can be used when sending SigComp messages that relate to the compressor's equivalent for the compartment.

The dispatcher passes the compartment identifier to the UDVM, where it is used as per the END-MESSAGE instruction (see Section 9.4.9).

The application uses a suitable authentication mechanism to determine whether the decompressed message belongs to a legitimate compartment or not. If the application fails to authenticate the message with sufficient confidence to allow state to be saved or feedback data to be trusted, it supplies a "no valid compartment" error to the dispatcher and the UDVM is terminated without creating any state or forwarding any feedback data.

5. SigComp Compressor

An important feature of SigComp is that decompression functionality is provided by a Universal Decompressor Virtual Machine (UDVM). This means that the compressor can choose any algorithm to generate compressed SigComp messages, and then upload bytecode for the corresponding decompression algorithm to the UDVM as part of the SigComp message.

To help with the implementation and testing of a SigComp endpoint, further Internet Documents and RFCs may be published to describe particular compression algorithms.

The overall requirement placed on the compressor is that of transparency, i.e., the compressor MUST NOT send bytecode which causes the UDVM to incorrectly decompress a given SigComp message.

The following more specific requirements are also placed on the compressor (they can be considered particular instances of the transparency requirement):

1. For robustness, it is recommended that the compressor supply some form of integrity check (not necessarily of cryptographic strength) over the application message to ensure that successful decompression has occurred. A UDVM instruction is provided for CRC verification; also, another instruction can be used to compute a SHA-1 cryptographic hash.

2. The compressor MUST ensure that the message can be decompressed using the resources available at the remote endpoint.

3. If the transport is message-based, then the compressor MUST map each application message to exactly one SigComp message.

4. If the transport is stream-based but the application defines its own internal message boundaries, then the compressor SHOULD map each application message to exactly one SigComp message.

Message boundaries should be preserved over a stream-based transport so that accidental or malicious damage to one SigComp message does not affect the decompression of subsequent messages.

Additionally, if the state handler passes some requested feedback to the compressor, then it SHOULD be returned in the next SigComp message generated by the compressor (unless the state handler passes some newer requested feedback before the older feedback has been sent, in which case the older feedback is deleted).

If present, the requested feedback item SHOULD be copied unmodified into the returned_feedback_item field provided in the SigComp message. Note that there is no need to transmit any requested feedback item more than once.

The compressor SHOULD also upload the local SigComp parameters to the remote endpoint, unless the endpoint has indicated that it does not wish to receive these parameters or the compressor determines that the parameters have already successfully arrived (see Section 5.1 for details of how this can be achieved). The SigComp parameters are uploaded to the UDVM memory at the remote endpoint as described in Section 9.4.9.

5.1. Ensuring Successful Decompression

A compressor MUST be certain that all of the data needed to decompress a SigComp message is available at the receiving endpoint. One way to ensure this is to send all of the needed information in every SigComp message (including bytecode to decompress the message). However, the compression ratio for this method will be relatively low.

To obtain the best overall compression ratio the compressor needs to request the creation of new state items at the remote endpoint. The information saved in these state items can then be accessed by later SigComp messages, avoiding the need to upload the data on a per-message basis.

Before the compressor can access saved state however, it must ensure that the SigComp message carrying the state creation request arrived successfully at the receiving endpoint. For a reliable transport (e.g., TCP or SCTP) this is guaranteed. For an unreliable transport however, the compressor must provide a suitable mechanism itself (see [RFC-3321] for further details).

The compressor must also ensure that the state item it wishes to access has not been rejected due to a lack of state memory. This can be accomplished by checking the state_memory_size parameter using the SigComp feedback mechanism (see Section 9.4.9 for further details).

5.2. Compression Failure

The compressor SHOULD make every effort to successfully compress an application message, but in certain cases this might not be possible (particularly if resources are scarce at the receiving endpoint). In this case a "compression failure" is called.

If a compression failure occurs then the compressor informs the dispatcher and takes no further action. The dispatcher MUST report this failure to the application so that it can try other methods to deliver the message.

6. State Handling and Feedback

This chapter defines the behavior of the SigComp state handler. The function of the state handler is to retain information between received SigComp messages; it is the only SigComp entity that is capable of this function, and so it is of particular importance from a security perspective.

6.1. Creating and Accessing State

To provide security against the malicious insertion or modification of SigComp messages, a separate instance of the UDVM is invoked to decompress each message. This ensures that damaged SigComp messages do not prevent the successful decompression of subsequent valid messages.

Note, however, that the overall compression ratio is often significantly higher if messages can be compressed relative to the information contained in previous messages. For this reason, it is possible to create state items for access when a later message is being decompressed. Both the creation and access of state are designed to be secure against malicious tampering with the compressed data. The UDVM can only create a state item when a complete message has been successfully decompressed and the application has returned a compartment identifier under which the state can be saved.

State access cannot be protected by relying on the application alone, since the authentication mechanism may require information from the decompressed message (which of course is not available until after the state has been accessed). Instead, SigComp protects state access by creating a state identifier that is a hash over the item of state to be retrieved. This state_identifier must be supplied to retrieve an item of state from the state handler.

Also note that state is not deleted when it is accessed. So even if a malicious sender manages to access some state information, subsequent messages compressed relative to this state can still be successfully decompressed.

Each state item contains a state_identifier that is used to access the state. One state identifier can be supplied in the SigComp message header to initialize the UDVM (see Chapter 7); additional state items can be retrieved using the STATE-ACCESS instruction. The

UDVM can also request the creation of a new state item by using the STATE-CREATE and END-MESSAGE instructions (see Chapter 9 for further details).

6.2. Memory Management

The state handler manages state memory on a per-compartment basis. Each compartment can store state up to a certain state_memory_size (where the application may assign different values for the state_memory_size parameter to different compartments).

As well as storing the state items themselves, the state handler maintains a list of the state items created by a particular compartment and ensures that no compartment exceeds its allocated state_memory_size. For the purpose of calculation, each state item is considered to cost (state_length + 64) bytes.

Each instance of the UDVM can pass up to four state creation requests to the state handler, as well as up to four state free requests (the latter are requests to free the memory taken by a state item in a certain compartment). When the state handler receives a state creation request from the UDVM it takes the following steps:

1. The state handler MUST reject all state creation requests that are not accompanied by a valid compartment identifier, or if the compartment is allocated 0 bytes of state memory. Note that if a state creation request fails due to lack of state memory then it does not mean that the corresponding SigComp message is damaged; compressors will often make state creation requests in the first SigComp message of a compartment, before they have discovered the state_memory_size using the SigComp feedback mechanism.

2. If the state creation request needs more state memory than the total state_memory_size for the compartment, the state handler deletes all but the first (state_memory_size - 64) bytes from the state_value. It sets the state_length to (state_memory_size - 64), and recalculates the state_identifier as defined in Section 9.4.9.

3. If the state creation request contains a state_identifier that already exists then the state handler checks whether the requested state item is identical to the established state item and counts the state creation request as successful if this is the case. If not then the state creation request is unsuccessful (although the probability that this will occur is vanishingly small).

4. If the state creation request exceeds the state memory allocated to the compartment, sufficient items of state created by the same compartment are freed until enough memory is available to accommodate the new state. When a state item is freed, it is removed from the list of states created by the compartment and the memory cost of the state item no longer counts towards the total cost for the compartment. Note, however, that identical state items may be created by several different compartments, so a state item must not be physically deleted unless the state handler determines that it is no longer required by any compartment.

5. The order in which the existing state items are freed is determined by the state_retention_priority, which is set when the state items are created. The state_retention_priority of 65535 is reserved for locally available states; these states must always be freed first. Apart from this special case, states with the lowest state_retention_priority are always freed first. In the event of a tie, then the state item created first in the compartment is also the first to be freed.

The state_retention_priority is always stored on a per-compartment basis as part of the list of state items created by each compartment. In particular, the same state item might have several priority values if it has been created by several different compartments.

Note that locally available state items (as described in Section 3.3.3) need not be mapped to any particular compartment. However, if they are created on a per-compartment basis, then they must not interfere with the state created at the request of the remote endpoint. The special state_retention_priority of 65535 is reserved for locally available state items to ensure that this is the case.

The UDVM may also explicitly request the state handler to free a specific state item in a compartment. In this case, the state handler deletes the state item from the list of state items created by the compartment (as before the state item itself must not be physically deleted unless the state handler determines that it is not longer required by any compartment).

The application should indicate to the state handler when it wishes to close a particular compartment, so that the resources taken by the corresponding state can be reclaimed.

6.3. Feedback Data

The SigComp feedback mechanism allows feedback data to be received by a UDVM and forwarded via the state handler to the correct compressor.

Since this feedback data is retained between SigComp messages, it is considered to be part of the overall state and can only be forwarded if accompanied by a valid compartment identifier. If this is the case, then the state handler forwards the feedback data to the compressor responsible for sending messages that pertain to the peer compartment of the specified compartment.

7. SigComp Message Format

This chapter describes the format of the SigComp message and how the message is used to initialize the UDVM memory.

Note that the SigComp message is not copied into the UDVM memory as soon as it arrives; instead, the UDVM indicates when it requires compressed data using a specific instruction. It then pauses and waits for the information to be supplied before executing the next instruction. This means that the UDVM can begin to decompress a SigComp message before the entire message has been received.

A consequence of the above behavior is that when the UDVM is invoked, the size of the UDVM memory depends on whether the transport used to provide the SigComp message is stream-based or message-based. If the transport is message-based then sufficient memory must be available to buffer the entire SigComp message before it is passed to the UDVM. So if the message is n bytes long, then the UDVM memory size is set to (decompression_memory_size - n), up to a maximum of 65536 bytes.

If the transport is stream-based however, then a fixed-size input buffer is required to accommodate the stream, independently of the size of each SigComp message. So, for simplicity, the UDVM memory size is set to (decompression_memory_size / 2).

As a separate instance of the UDVM is invoked on a per-message basis, each SigComp message must explicitly indicate its chosen decompression algorithm as well as any additional information that is needed to decompress the message (e.g., one or more previously received messages, a dictionary of common SIP phrases etc.). This information can either be uploaded as part of the SigComp message or retrieved from an item of state.

A SigComp message takes one of two forms depending on whether it accesses a state item at the receiving endpoint. The two variants of a SigComp message are given in Figure 3. (The T-bit controls the format of the returned feedback item and is defined in Section 7.1.)

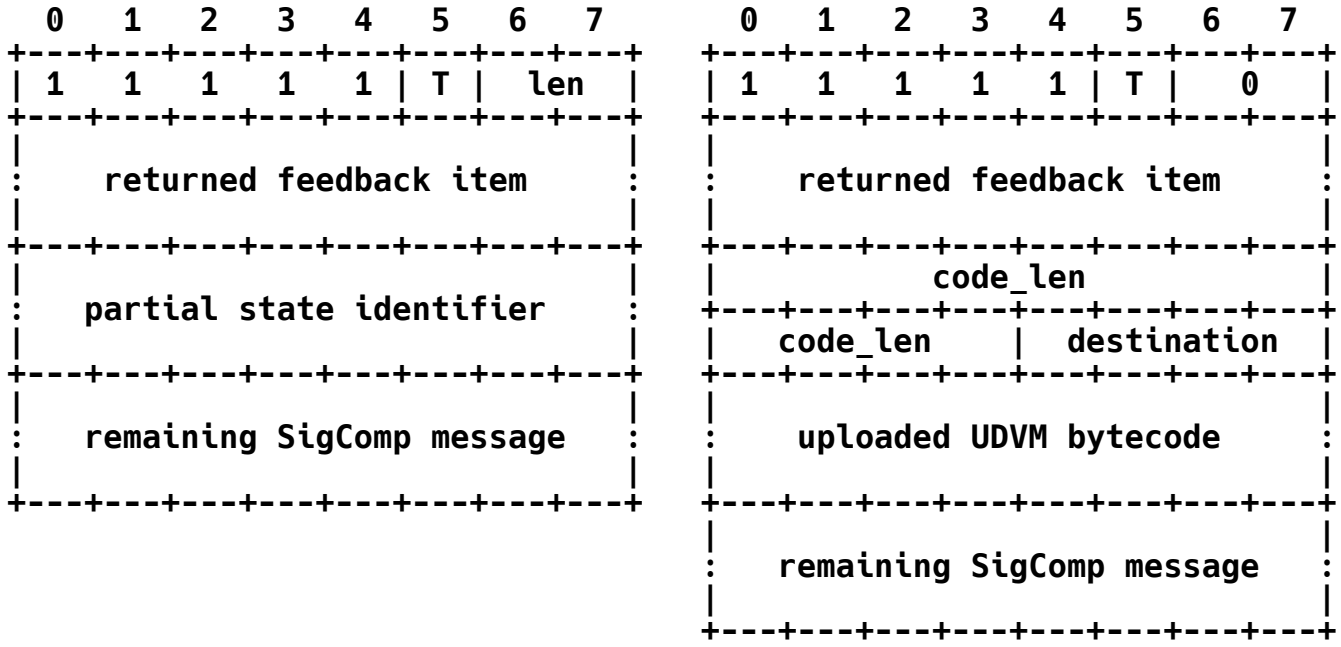


Figure 3: Format of a SigComp message

Decompression failure occurs if the SigComp message is too short to contain the expected fields (see Section 8.7 for further details).

The fields except for the "remaining SigComp message" are referred to as the "SigComp header" (note that this may include the uploaded UDVM bytecode).

7.1. Returned feedback item

For both variants of the SigComp message, the T-bit is set to 1 whenever the SigComp message contains a returned feedback item. The format of the returned feedback item is illustrated in Figure 4.

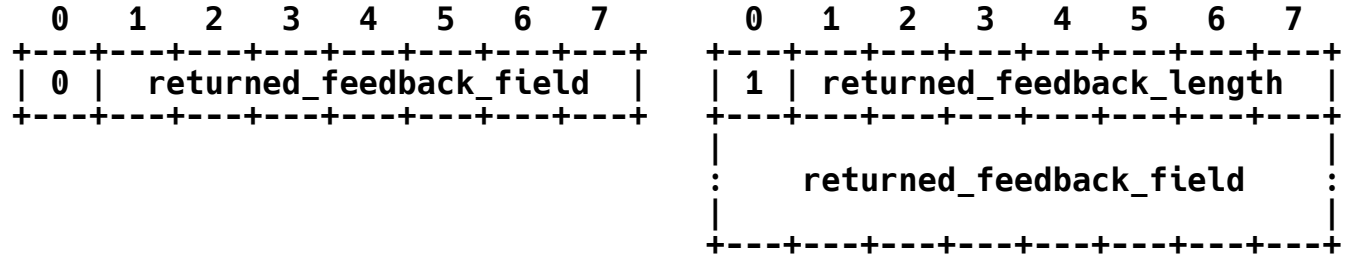


Figure 4: Format of returned feedback item

Note that the returned feedback length specifies the size of the returned feedback field (from 0 to 127 bytes). So the total size of the returned feedback item lies between 1 and 128 bytes.

The returned feedback item is not copied to the UDVM memory; instead, it is buffered until the UDVM has successfully decompressed the SigComp message. It is then forwarded to the state handler with the rest of the feedback data (see Section 9.4.9 for further details).

7.2. Accessing Stored State

The len field of the SigComp message determines which fields follow the returned feedback item. If the len field is non-zero, then the SigComp message contains a state identifier to access a state item at the receiving endpoint. All state items include a 20-byte state identifier as per Section 3.3.3, but it is possible to transmit as few as 6 bytes from the identifier if the sender believes that this is sufficient to match a unique state item at the receiving endpoint.

The len field encodes the number of transmitted bytes as follows:

Encoding:	Length of partial state identifier
01	6 bytes
10	9 bytes
11	12 bytes

The partial state identifier is passed to the state handler, which compares it with the most significant bytes of the state_identifier in every currently stored state item. Decompression failure occurs if no state item is matched or if more than one state item is matched.

Decompression failure also occurs if exactly one state item is matched but the state item contains a minimum_access_length greater than the length of the partial state identifier. This prevents especially sensitive state items from being accessed maliciously by brute force guessing of the state_identifier.

If a state item is successfully accessed then the state_value byte string is copied into the UDVM memory beginning at state_address.

The first 32 bytes of UDVM memory are then initialized to special values as illustrated in Figure 5.

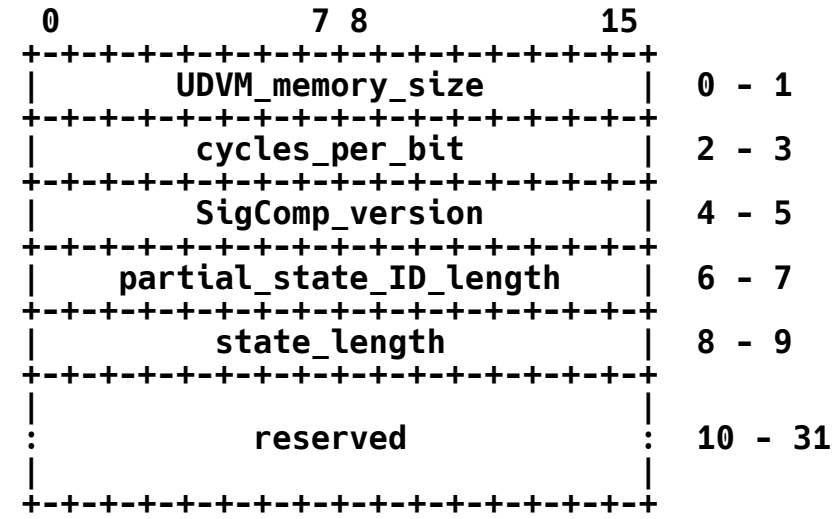


Figure 5: Initializing Useful Values in UDVM memory

The first five 2-byte words are initialized to contain some values that might be useful to the UDVM bytecode (Useful Values). Note that these values are for information only and can be overwritten when executing the UDVM bytecode without any effect on the endpoint. The MSBs of each 2-byte word are stored preceding the LSBs.

Addresses 0 to 5 indicate the resources available to the receiving endpoint. The UDVM memory size is expressed in bytes modulo 2^16, so in particular, it is set to 0 if the UDVM memory size is 65536 bytes. The cycles_per_bit is expressed as a 2-byte integer taking the value 16, 32, 64 or 128. The SigComp_version is expressed as a 2-byte value as per Section 3.3.2.

Addresses 6 to 9 are initialized to the length of the partial state identifier, followed by the state_length from the retrieved state item. Both are expressed as 2-byte values.

Addresses 10 to 31 are reserved and are initialized to 0 for Version 0x01 of SigComp. Future versions of SigComp can use these locations for additional Useful Values, so a decompressor MUST NOT rely on these values being zero.

Any remaining addresses in the UDVM memory that have not yet been initialized MUST be set to 0.

The UDVM then begins executing instructions at the memory address contained in state_instruction (which is part of the retrieved item of state). Note that the remaining SigComp message is held by the decompressor dispatcher until requested by the UDVM.

(Note that the Useful Values are only set at UDVM startup; there is no special significance to this memory area afterwards. This means that the UDVM bytecode is free to use these locations for any other purpose a memory location might be used for; it just has to be aware they are not necessarily initialized to zero.)

7.3. Uploading UDVM bytecode

If the len field is set to 0 then the bytecode needed to decompress the SigComp message is supplied as part of the message itself. The 12-bit code_len field specifies the size of the uploaded UDVM bytecode (from 0 to 4095 bytes inclusive); eight most significant bits are in the first byte, followed by the four least significant bits in the most significant bits in the second byte. The remaining bits in the second byte are interpreted as a 4-bit destination field that specifies the starting memory address to which the bytecode is copied. The destination field is encoded as follows:

Encoding:	Destination address:
0000	reserved
0001	2 * 64 = 128
0010	3 * 64 = 196
0011	4 * 64 = 256
:	:
1111	16 * 64 = 1024

Note that the encoding 0000 is reserved for future SigComp versions, and causes a decompression failure in Version 0x01.

The UDVM memory is initialized as per Figure 5, except that addresses 6 to 9 inclusive are set to 0 because no state item has been accessed. The UDVM then begins executing instructions at the memory address specified by the destination field. As above, the remaining SigComp message is held by the decompressor dispatcher until needed by the UDVM.

8. Overview of the UDVM

Decompression functionality for SigComp is provided by a Universal Decompressor Virtual Machine (UDVM). The UDVM is a virtual machine much like the Java Virtual Machine but with a key difference: it is designed solely for the purpose of running decompression algorithms.

The motivation for creating the UDVM is to provide flexibility when choosing how to compress a given application message. Rather than picking one of a small number of pre-negotiated algorithms, the compressor implementer has the freedom to select an algorithm of their choice. The compressed data is then combined with a set of UDVM instructions that allow the original data to be extracted, and the result is outputted as a SigComp message. Since the UDVM is optimized specifically for running decompression algorithms, the code size of a typical algorithm is small (often sub 100 bytes). Moreover, the UDVM approach does not add significant extra processing or memory requirements compared to running a fixed preprogrammed decompression algorithm.

Figure 6 gives a detailed view of the interfaces between the UDVM and its environment.

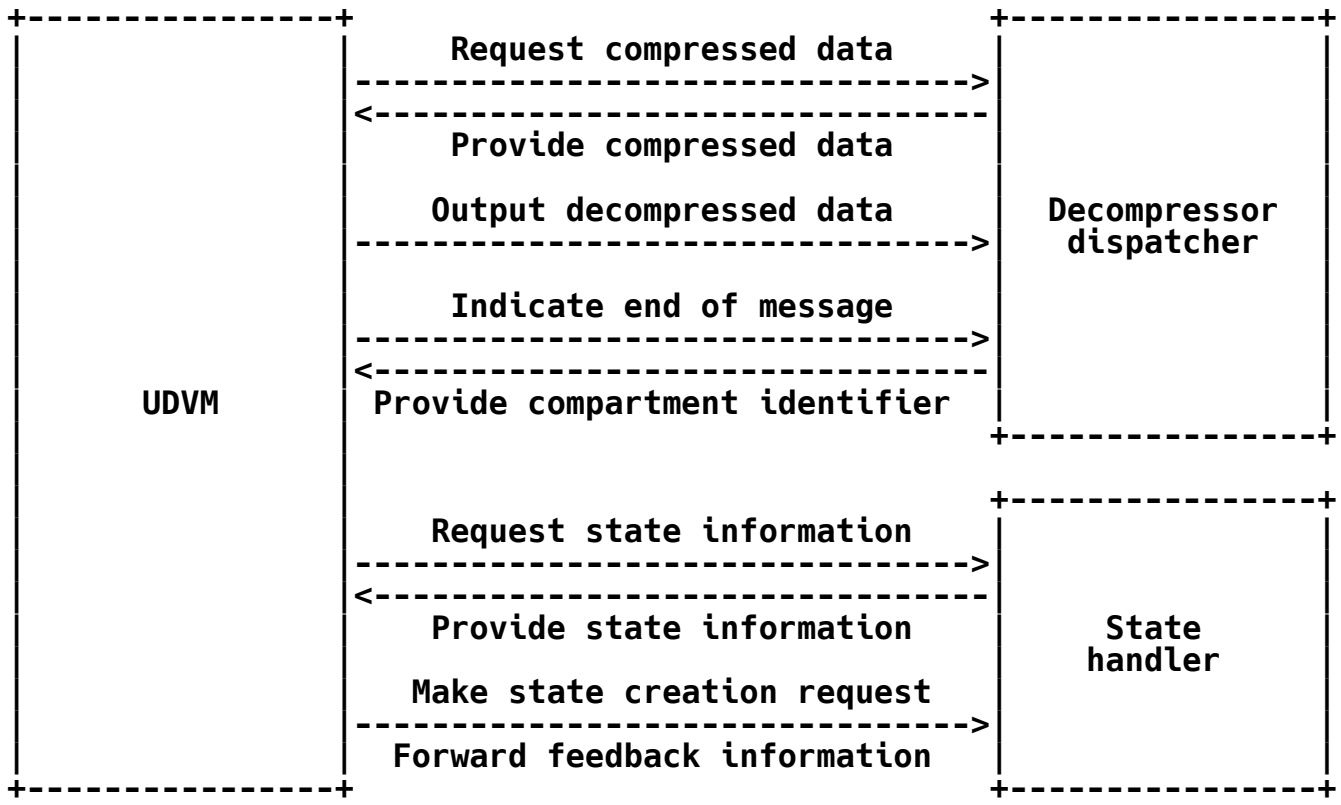


Figure 6: Interfaces between the UDVM and its environment

Note that once the UDVM has been initialized, additional compressed data and state information are only provided at the request of a specific UDVM instruction.

This chapter describes the basic features of the UDVM including the UDVM registers and the format of UDVM bytecode.

8.1. UDVM Registers

The UDVM registers are 2-byte words in the UDVM memory that have special tasks, for example specifying the location of the stack used by the CALL and RETURN instructions.

The UDVM registers are illustrated in Figure 7.

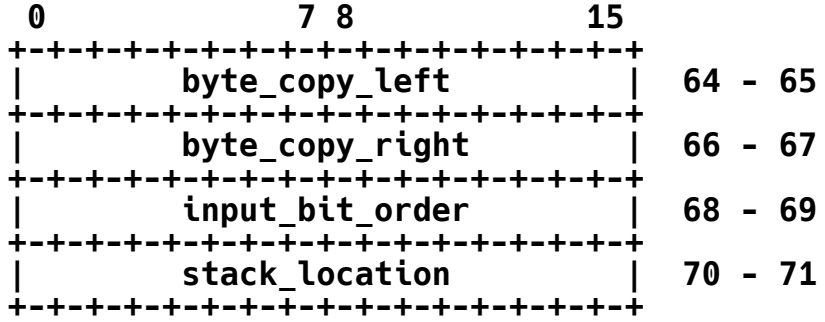


Figure 7: Memory addresses of the UDVM registers

The MSBs of each register are always stored before the LSBs. So, for example, the MSBs of byte_copy_left are stored at Address 64 whilst the LSBs are stored at Address 65.

The use of each UDVM register is defined in the following sections.

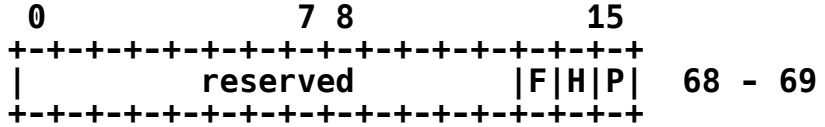
(Note that the UDVM registers start at Address 64, that is 32 bytes after the area reserved for Useful Values. The intention is that the gap, i.e., the area between Address 32 and Address 63, will often be used as scratch-pad memory that is guaranteed to be zero at UDVM startup and is efficiently addressable in operand types reference ($) and multitype (%).)

8.2. Requesting Additional Compressed Data

The decompressor dispatcher stores the compressed data from the SigComp message before it is requested by the UDVM via one of the INPUT instructions. When the UDVM bytecode is first executed, the dispatcher contains the remaining SigComp message after the header has been used to initialize the UDVM as per Chapter 7.

Note that the INPUT-BITS and INPUT-HUFFMAN instructions retrieve a stream of individual compressed bits from the dispatcher. To provide bitwise compatibility with various well-known compression algorithms, the input_bit_order register can modify the order in which individual bits are passed within a byte.

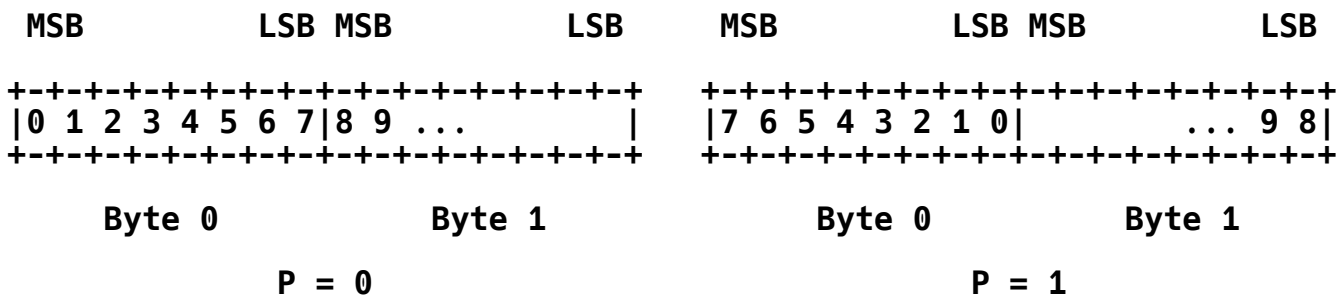
The input_bit_order register contains the following three flags:



The P-bit controls the order in which bits are passed from the dispatcher to the INPUT instructions. If set to 0, it indicates that the bits within an individual byte are passed to the INPUT instructions in MSB to LSB order. If it is set to 1, the bits are passed in LSB to MSB order.

Note that the input_bit_order register cannot change the order in which the bytes themselves are passed to the INPUT instructions (bytes are always passed in the same order as they occur in the SigComp message).

The following diagram illustrates the order in which bits are passed to the INPUT instructions for both cases:



Note that after one or more INPUT instructions the dispatcher may hold a fraction of a byte (what used to be the LSBs if P = 0, or, the MSBs, if P = 1). If an INPUT instruction is encountered and the P-bit has changed since the last INPUT instruction, any fraction of a byte still held by the dispatcher MUST be discarded (even if the INPUT instruction requests zero bits). The first bit passed to the INPUT instruction is taken from the subsequent byte.

When an INPUT instruction requests n bits of compressed data, it interprets the received bits as an integer between 0 and 2^n - 1. The F-bit and the H-bit specify whether the bits in these integers are considered to arrive in MSB to LSB order (bit set to 0) or in LSB to MSB order (bit set to 1).

If the F-bit is set to 0, the INPUT-BITS instruction interprets the received bits as arriving MSBs first, and if it is set to 1, it interprets the bits as arriving LSBs first. The H-bit performs the same function for the INPUT-HUFFMAN instruction. Note that it is possible to set these two bits to different values in order to use different bit orders for the two instructions (certain algorithms actually require this, e.g., DEFLATE [RFC-1951]). (Note that there are no special considerations for changing the F- or H-bit between INPUT instructions, unlike the discard rule for the P-bit described above.)

Decompression failure occurs if an INPUT-BITS or an INPUT-HUFFMAN instruction is encountered and the input_bit_order register does not lie between 0 and 7 inclusive.

8.3. UDVM Stack

Certain UDVM instructions make use of a stack of 2-byte words stored at the memory address specified by the 2-byte word stack_location. The stack contains the following words:

Name:	Starting memory address:
stack_fill	stack_location
stack[0]	stack_location + 2
stack[1]	stack_location + 4
stack[2]	stack_location + 6
:	:

The notation stack_location is an abbreviation for the contents of the stack_location register, i.e., the 2-byte word at locations 70 and 71. The notation stack_fill is an abbreviation for the 2-byte word at stack_location and stack_location+1. Similarly, the notation stack[n] is an abbreviation for the 2-byte word at stack_location+2*n+2 and stack_location+2*n+3. (As always, the arithmetic is modulo 2^16.)

The stack is used by the CALL, RETURN, PUSH and POP instructions.

"Pushing" a value on the stack is an abbreviation for copying the value to stack[stack_fill] and then increasing stack_fill by 1. CALL and PUSH push values on the stack.

"Popping" a value from the stack is an abbreviation for decreasing stack_fill by 1, and then using the value stored in stack[stack_fill]. Decompression failure occurs if stack_fill is zero at the commencement of a popping operation. POP and RETURN pop values from the stack.

For both of these abstract operations, the UDVM first takes note of the current value of stack_location and uses this value for both sub-operations (accessing the stack and manipulating stack_fill), i.e., overwriting stack_location in the course of the operation is inconsequential for the operation.

8.4. Byte copying

A number of UDVM instructions require a string of bytes to be copied to and from areas of the UDVM memory. This section defines how the byte copying operation should be performed.

The string of bytes is copied in ascending order of memory address, respecting the bounds set by byte_copy_left and byte_copy_right. More precisely, if a byte is copied from/to Address m then the next byte is copied from/to Address n where n is calculated as follows:

Set k := m + 1 (modulo 2^16)
If k = byte_copy_right then set n := byte_copy_left, else set n := k

Decompression failure occurs if a byte is copied from/to an address beyond the UDVM memory.

Note that the string of bytes is copied one byte at a time. In particular, some of the later bytes to be copied may themselves have been written into the UDVM memory by the byte copying operation currently being performed.

Equally, it is possible for a byte copying operation to overwrite the instruction that invoked the byte copy. If this occurs, then the byte copying operation MUST be completed as if the original instruction were still in place in the UDVM memory (this also applies if byte_copy_left or byte_copy_right are overwritten).

Byte copying is used by the following UDVM instructions:

SHA-1, COPY, COPY-LITERAL, COPY-OFFSET, MEMSET, INPUT-BYTES, STATE-ACCESS, OUTPUT, END-MESSAGE

8.5. Instruction operands and UDVM bytecode

Each of the UDVM instructions in a piece of UDVM bytecode is represented by a single byte, followed by 0 or more bytes containing the operands required by the instruction.

During instruction execution, conceptually the UDVM first fetches the first byte of the instruction, determines the number and types of operands required for this instruction, and then decodes all the operands in sequence before starting to act on the instruction. (Note that the UDVM instructions have been designed in such a way that this sequence remains conceptual in those cases where it would result in an unreasonable burden on the implementation.)

To reduce the size of typical UDVM bytecode, each operand for a UDVM instruction is compressed using variable-length encoding. The aim is to store more common operand values using fewer bytes than rarely occurring values.

Four different types of operand are available: the literal, the reference, the multitype and the address. Chapter 9 gives a complete list of UDVM instructions and the operand types that follow each instruction.

The UDVM bytecode for each operand type is illustrated in Figure 8 to Figure 10, together with the integer values represented by the bytecode.

Note that the MSBs in the bytecode are illustrated as preceding the LSBs. Also, any string of bits marked with k consecutive "n"s is to be interpreted as an integer N from 0 to 2^k - 1 inclusive (with the MSBs of n illustrated as preceding the LSBs).

The decoded integer value of the bytecode can be interpreted in two ways. In some cases it is taken to be the actual value of the operand. In other cases it is taken to be a memory address at which the 2-byte operand value can be found (MSBs found at the specified address, LSBs found at the following address). The latter cases are denoted by memory[X] where X is the address and memory[X] is the 2-byte value starting at Address X.

The simplest operand type is the literal (#), which encodes a constant integer from 0 to 65535 inclusive. A literal operand may require between 1 and 3 bytes depending on its value.

Bytecode:	Operand value:	Range:
0nnnnnnn	N	0 - 127
10nnnnnn nnnnnnnn	N	0 - 16383
11000000 nnnnnnnn nnnnnnnn	N	0 - 65535

Figure 8: Bytecode for a literal (#) operand

The second operand type is the reference ($), which is always used to access a 2-byte value located elsewhere in the UDVM memory. The bytecode for a reference operand is decoded to be a constant integer from 0 to 65535 inclusive, which is interpreted as the memory address containing the actual value of the operand.

Bytecode:	Operand value:	Range:
0nnnnnnn	memory[2 * N]	0 - 65535
10nnnnnn nnnnnnnn	memory[2 * N]	0 - 65535
11000000 nnnnnnnn nnnnnnnn	memory[N]	0 - 65535

Figure 9: Bytecode for a reference ($) operand

Note that the range of a reference operand is always 0 - 65535 independently of how many bits are used to encode the reference, because the operand always references a 2-byte value in the memory.

The third kind of operand is the multitype (%), which can be used to encode both actual values and memory addresses. The multitype operand also offers efficient encoding for small integer values (both positive and negative) and for powers of 2.

Bytecode:	Operand value:	Range:
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: Bytecode for a multitype (%) operand

The fourth operand type is the address (@). This operand is decoded as a multitype operand followed by a further step: the memory address of the UDVM instruction containing the address operand is added to obtain the correct operand value. So if the operand value from Figure 10 is D then the actual operand value of an address is calculated as follows:

operand_value = (memory_address_of_instruction + D) modulo 2^16

Address operands are always used in instructions that control program flow, because they ensure that the UDVM bytecode is position-independent code (i.e., it will run independently of where it is placed in the UDVM memory).

8.6. UDVM Cycles

Once the UDVM has been invoked it executes the instructions contained in its memory consecutively unless otherwise indicated (for example when the UDVM encounters a JUMP instruction). If the next instruction to be executed lies outside the available memory then decompression failure occurs (see Section 8.7).

To ensure that a SigComp message cannot consume excessive processing resources, SigComp limits the number of "UDVM cycles" allocated to each message. The number of available UDVM cycles is initialized to 1000 plus the number of bits in the SigComp header (as described in Section 7); this sum is then multiplied by cycles_per_bit. Each time an instruction is executed the number of available UDVM cycles is decreased by the amount specified in Chapter 9. Additionally, if the UDVM successfully requests n bits of compressed data using one of the INPUT instructions then the number of available UDVM cycles is increased by n * cycles_per_bit once the instruction has been executed.

This means that the maximum number of UDVM cycles available for processing an n-byte SigComp message is given by the formula:

```
maximum_UDVM_cycles = (8 * n + 1000) * cycles_per_bit
```

The reason that this total is not allocated to the UDVM when it is invoked is that the UDVM can begin to decompress a message that has only been partially received. So the total message size may not be known when the UDVM is initialized.

Note that the number of UDVM cycles MUST NOT be increased if a request for additional compressed data fails.

The UDVM stops executing instructions when it encounters an END-MESSAGE instruction or if decompression failure occurs (see Section 8.7 for further details).

8.7. Decompression Failure

If a compressed message given to the UDVM is corrupted (either accidentally or maliciously), then the UDVM may terminate with a decompression failure.

Reasons for decompression failure include the following:

1. A SigComp message contains an invalid header as per Chapter 7.

2. A SigComp message is larger than the decompression_memory_size.

3. An instruction costs more than the number of remaining UDVM cycles.

4. The UDVM attempts to read from or write to a memory address beyond its memory size.

5. An unknown instruction is encountered.

6. An unknown operand is encountered.

7. An instruction is encountered that cannot be processed successfully by the UDVM (for example a RETURN instruction when no CALL instruction has previously been encountered).

8. A request to access some state information fails.

9. A manual decompression failure is triggered using the DECOMPRESSION-FAILURE instruction.

If a decompression failure occurs when decompressing a message then the UDVM informs the dispatcher and takes no further action. It is the responsibility of the dispatcher to decide how to cope with the decompression failure. In general a dispatcher SHOULD discard the compressed message (or the compressed stream if the transport is stream-based) and any decompressed data that has been outputted but not yet passed to the application.

9. UDVM Instruction Set

The UDVM currently understands 36 instructions, chosen to support the widest possible range of compression algorithms with the minimum possible overhead.

Figure 11 lists the different instructions and the bytecode values used to encode the instructions. The cost of each instruction in UDVM cycles is also given:

Instruction:	Bytecode value:	Cost in UDVM cycles:
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 * (ceiling(log2(k)) + n)
SORT-DESCENDING	12	1 + k * (ceiling(log2(k)) + n)
SHA-1	13	1 + length
LOAD	14	1
MULTILOAD	15	1 + n
PUSH	16	1
POP	17	1
COPY	18	1 + length
COPY-LITERAL	19	1 + length
COPY-OFFSET	20	1 + length
MEMSET	21	1 + length
JUMP	22	1
COMPARE	23	1
CALL	24	1
RETURN	25	1
SWITCH	26	1 + n
CRC	27	1 + length
INPUT-BYTES	28	1 + length
INPUT-BITS	29	1
INPUT-HUFFMAN	30	1 + n
STATE-ACCESS	31	1 + state_length
STATE-CREATE	32	1 + state_length
STATE-FREE	33	1
OUTPUT	34	1 + output_length
END-MESSAGE	35	1 + state_length

Figure 11: UDVM instructions and corresponding bytecode values

Each UDVM instruction costs a minimum of 1 UDVM cycle. Certain instructions may cost additional cycles depending on the values of the instruction operands. Named variables in the cost expressions refer to the values of the instruction operands with these names.

Note that for the SORT instructions, the formula ceiling(log2(k)) calculates the smallest value i such that k <= 2^i.

The UDVM instruction set offers a mix of low-level and high-level instructions. The high-level instructions can all be emulated using combinations of low-level instructions, but given a choice it is generally preferable to use a single instruction rather than a large number of general-purpose instructions. The resulting bytecode will be more compact (leading to a higher overall compression ratio) and decompression will typically be faster because the implementation of the high-level instructions can be more easily optimized.

All instructions are encoded as a single byte to indicate the instruction type, followed by 0 or more bytes containing the operands required by the instruction. The instruction specifies which of the four operand types of Section 8.5 is used in each case. For example the ADD instruction is followed by two operands:

ADD ($operand_1, %operand_2)

When converted into bytecode the number of bytes required by the ADD instruction depends on the value of each operand, and whether the multitype operand contains the operand value itself or a memory address where the actual value of the operand can be found.

Each instruction is explained in more detail below.

Whenever the description of an instruction uses the expression "and then", the intended semantics is that the effect explained before "and then" is completed before work on the effect explained after the "and then" is commenced.

9.1. Mathematical Instructions

The following instructions provide a number of mathematical operations including bit manipulation, arithmetic and sorting.

9.1.1. Bit Manipulation

The AND, OR, NOT, LSHIFT and RSHIFT instructions provide simple bit manipulation on 2-byte words.

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

After the operation is complete, the value of the first operand is overwritten with the result. (Note that since this operand is a reference, it is the 2-byte word at the memory address specified by the operand that is overwritten.)

The precise definitions of LSHIFT and RSHIFT are given below. Note that m and n are the 2-byte values encoded by the operands, and that floor(x) calculates the largest integer not greater than x:

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

9.1.2. Arithmetic

The ADD, SUBTRACT, MULTIPLY, DIVIDE and REMAINDER instructions perform arithmetic on 2-byte words.

```
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)
```

After the operation is complete, the value of the first operand is overwritten with the result.

The precise definition of each instruction is given below:

```
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)    := floor(m / n)
REMAINDER (m, n) := m - n * floor(m / n)
```

Decompression failure occurs if a DIVIDE or REMAINDER instruction encounters an operand_2 that is zero.

9.1.3. Sorting

The SORT-ASCENDING and SORT-DESCENDING instructions sort lists of 2-byte words.

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

The start operand specifies the starting memory address of the block of data to be sorted.

The block of data itself is divided into n lists each containing k 2-byte words. The SORT-ASCENDING instruction applies a certain permutation to the lists, such that the first list is sorted into ascending order (treating each 2-byte word as an unsigned integer). The same permutation is applied to all n lists, so lists other than the first will not necessarily be sorted into order.

In the case that two words have the same value, the original ordering of the list is preserved.

For example, the first list might contain a set of integers to be sorted whilst the second list might be used to keep track of where the integers appear in the sorted list:

Before sorting		After sorting	
List 1	List 2	List 1	List 2
8	1	1	2
1	2	1	3
1	3	3	4
3	4	8	1

The SORT-DESCENDING instruction behaves as above, except that the first list is sorted into descending order.

9.1.4. SHA-1

The SHA-1 instruction calculates a 20-byte SHA-1 hash [RFC-3174] over the specified area of UDVM memory.

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

The position and length operands specify the starting memory address and the length of the byte string over which the SHA-1 hash is calculated. Byte copying rules are enforced as per Section 8.4.

The destination operand gives the starting address to which the resulting 20-byte hash will be copied. Byte copying rules are enforced as above.

9.2. Memory Management Instructions

The following instructions are used to set up the UDVM memory, and to copy byte strings from one memory location to another.

9.2.1. LOAD

The LOAD instruction sets a 2-byte word to a certain specified value. The format of a LOAD instruction is as follows:

LOAD (%address, %value)

The first operand specifies the starting address of a 2-byte word, whilst the second operand specifies the value to be loaded into this word. As usual, MSBs are stored before LSBs in the UDVM memory.

9.2.2. MULTILOAD

The MULTILOAD instruction sets a contiguous block of 2-byte words in the UDVM memory to specified values.

MULTILOAD (%address, #n, %value_0, ..., %value_n-1)

The first operand specifies the starting address of the contiguous 2-byte words, whilst the operands value_0 through to value_n-1 specify the values to load into these words (in the same order as they appear in the instruction).

Decompression failure occurs if the set of 2-byte words set by the instruction would overlap the memory locations held by the instruction (including its operands) itself, i.e., if the instruction would be self-modifying. (This restriction makes it simpler to implement MULTILOAD step-by-step instead of having to decode all operands before being able to copy data, as is implied by the conceptual model of instruction execution.)

9.2.3. PUSH and POP

The PUSH and POP instructions read from and write to the UDVM stack (as defined in Section 8.3).

PUSH (%value)
POP (%address)

The PUSH instruction pushes the value specified by its operand on the stack.

The POP instruction pops a value from the stack and then copies the value to the specified memory address. (Note that the expression "and then" implies that the copying of the value is inconsequential for the stack operation itself, which happens beforehand.)

See Section 8.3 for possible error conditions.

9.2.4. COPY

The COPY instruction is used to copy a string of bytes from one part of the UDVM memory to another.

COPY (%position, %length, %destination)

The position operand specifies the memory address of the first byte in the string to be copied, and the length operand specifies the number of bytes to be copied.

The destination operand gives the address to which the first byte in the string will be copied.

Byte copying is performed as per the rules of Section 8.4.

9.2.5. COPY-LITERAL

A modified version of the COPY instruction is given below:

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

The COPY-LITERAL instruction behaves as a COPY instruction except that after copying is completed, the value of the destination operand is replaced by the address to which the next byte of data would be copied. More precisely it is replaced by the value n, derived as per Section 8.4 with m set to the destination address of the last byte to be copied, if any (i.e., if the value of the length operand is zero, the value of the destination operand is not changed).

9.2.6. COPY-OFFSET

A further version of the COPY-LITERAL instruction is given below:

COPY-OFFSET (%offset, %length, $destination)

The COPY-OFFSET instruction behaves as a COPY-LITERAL instruction except that an offset operand is given instead of a position operand.

To derive the value of the position operand, starting at the memory address specified by destination, the UDVM counts backwards a total of offset memory addresses.

If the memory address specified in byte_copy_left is reached, the next memory address is taken to be (byte_copy_right - 1) modulo 2^16.

The COPY-OFFSET instruction then behaves as a COPY-LITERAL instruction, taking the value of the position operand to be the last memory address reached in the above step.

9.2.7. MEMSET

The MEMSET instruction initializes an area of UDVM memory to a specified sequence of values. The format of a MEMSET instruction is as follows:

MEMSET (%address, %length, %start_value, %offset)

The sequence of values used by the MEMSET instruction is specified by the following formula:

Seq[n] := (start_value + n * offset) modulo 256

The values Seq[0] to Seq[length - 1] inclusive are each interpreted as a single byte, and then concatenated to form a byte string where the first byte has value Seq[0], the second byte has value Seq[1] and so on up to the last byte which has value Seq[length - 1].

The string is then byte copied into the UDVM memory beginning at the memory address specified as an operand to the MEMSET instruction, obeying the rules of Section 8.4. (Note that the byte string may overwrite the MEMSET instruction or its operands; as explained in Section 8.5, the MEMSET instruction must be executed as if the original operands were still in place in the UDVM memory.)

9.3. Program Flow Instructions

The following instructions alter the flow of UDVM code. Each instruction jumps to one of a number of memory addresses based on a certain specified criterion.

Note that certain I/O instructions (see Section 9.4) can also alter program flow.

9.3.1. JUMP

The JUMP instruction moves program execution to the specified memory address.

JUMP (@address)

Decompression failure occurs if the value of the address operand lies beyond the overall UDVM memory size.

9.3.2. COMPARE

The COMPARE instruction compares two operands and then jumps to one of three specified memory addresses depending on the result.

COMPARE (%value_1, %value_2, @address_1, @address_2, @address_3)

If value_1 < value_2 then the UDVM continues instruction execution at the memory address specified by address 1. If value_1 = value_2 then it jumps to the address specified by address_2. If value_1 > value_2 then it jumps to the address specified by address_3.

9.3.3. CALL and RETURN

The CALL and RETURN instructions provide support for compression algorithms with a nested structure.

CALL (@address)
RETURN

Both instructions use the UDVM stack of Section 8.3. When the UDVM reaches a CALL instruction, it finds the memory address of the instruction immediately following the CALL instruction and pushes this 2-byte value on the stack, ready for later retrieval. It then continues instruction execution at the memory address specified by the address operand.

When the UDVM reaches a RETURN instruction it pops a value from the stack and then continues instruction execution at the memory address just popped.

See Section 8.3 for error conditions.

9.3.4. SWITCH

The SWITCH instruction performs a conditional jump based on the value of one of its operands.

SWITCH (#n, %j, @address_0, @address_1, ... , @address_n-1)

When a SWITCH instruction is encountered the UDVM reads the value of j. It then continues instruction execution at the address specified by address j.

Decompression failure occurs if j specifies a value of n or more, or if the address lies beyond the overall UDVM memory size.

9.3.5. CRC

The CRC instruction verifies a string of bytes using a 2-byte CRC.

CRC (%value, %position, %length, @address)

The actual CRC calculation is performed using the generator polynomial x^16 + x^12 + x^5 + 1, which coincides with the 2-byte Frame Check Sequence (FCS) of PPP [RFC-1662].

The position and length operands define the string of bytes over which the CRC is evaluated. Byte copying rules are enforced as per Section 8.4.

The CRC value is computed exactly as defined for the 16-bit FCS calculation in [RFC-1662].

The value operand contains the expected integer value of the 2-byte CRC. If the calculated CRC matches the expected value then the UDVM continues instruction execution at the following instruction. Otherwise the UDVM jumps to the memory address specified by the address operand.

9.4. I/O instructions

The following instructions allow the UDVM to interface with its environment. Note that in the overall SigComp architecture all of these interfaces pass to the decompressor dispatcher or to the state handler.

9.4.1. DECOMPRESSION-FAILURE

The DECOMPRESSION-FAILURE instruction triggers a manual decompression failure. This is useful if the UDVM bytecode discovers that it cannot successfully decompress the message (e.g., by using the CRC instruction).

This instruction has no operands.

9.4.2. INPUT-BYTES

The INPUT-BYTES instruction requests a certain number of bytes of compressed data from the decompressor dispatcher.

INPUT-BYTES (%length, %destination, @address)

The length operand indicates the requested number of bytes of compressed data, and the destination operand specifies the starting memory address to which they should be copied. Byte copying is performed as per the rules of Section 8.4.

If the instruction requests data that lies beyond the end of the SigComp message, no data is returned. Instead the UDVM moves program execution to the address specified by the address operand.

If the INPUT-BYTES is encountered after an INPUT-BITS or an INPUT-HUFFMAN instruction has been used, and the dispatcher currently holds a fraction of a byte, then the fraction MUST be discarded before any data is passed to the UDVM. The first byte to be passed is the byte immediately following the discarded data.

9.4.3. INPUT-BITS

The INPUT-BITS instruction requests a certain number of bits of compressed data from the decompressor dispatcher.

INPUT-BITS (%length, %destination, @address)

The length operand indicates the requested number of bits. Decompression failure occurs if this operand does not lie between 0 and 16 inclusive.

The destination operand specifies the memory address to which the compressed data should be copied. Note that the requested bits are interpreted as a 2-byte integer ranging from 0 to 2^length - 1, as explained in Section 8.2.

If the instruction requests data that lies beyond the end of the SigComp message, no data is returned. Instead the UDVM moves program execution to the address specified by the address operand.

9.4.4. INPUT-HUFFMAN

The INPUT-HUFFMAN instruction requests a variable number of bits of compressed data from the decompressor dispatcher. The instruction initially requests a small number of bits and compares the result against a certain criterion; if the criterion is not met, then additional bits are requested until the criterion is achieved.

The INPUT-HUFFMAN instruction is followed by three mandatory operands plus n additional sets of operands. Every additional set contains four operands as shown below:

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

Note that if n = 0 then the INPUT-HUFFMAN instruction is ignored and program execution resumes at the following instruction. Decompression failure occurs if (bits_1 + ... + bits_n) > 16.

In all other cases, the behavior of the INPUT-HUFFMAN instruction is defined below:

1. Set j := 1 and set H := 0.

2. Request bits_j compressed bits. Interpret the returned bits as an integer k from 0 to 2^bits_j - 1, as explained in Section 8.2.

3. Set H := H * 2^bits_j + k.

4. If data is requested that lies beyond the end of the SigComp message, terminate the INPUT-HUFFMAN instruction and move program execution to the memory address specified by the address operand.

5. If (H < lower_bound_j) or (H > upper_bound_j) then set j := j + 1. Then go back to Step 2, unless j > n in which case decompression failure occurs.

6. Copy (H + uncompressed_j - lower_bound_j) modulo 2^16 to the memory address specified by the destination operand.

9.4.5. STATE-ACCESS

The STATE-ACCESS instruction retrieves some previously stored state information.

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

The partial_identifier_start and partial_identifier_length operands specify the location of the partial state identifier used to retrieve the state information. This identifier has the same function as the partial state identifier transmitted in the SigComp message as per Section 7.2.

Decompression failure occurs if partial_identifier_length does not lie between 6 and 20 inclusive. Decompression failure also occurs if no state item matching the partial state identifier can be found, if

more than one state item matches the partial identifier, or if partial_identifier_length is less than the minimum_access_length of the matched state item. Otherwise, a state item is returned from the state handler.

If any of the operands state_address, state_instruction or state_length is set to 0 then its value is taken from the returned item of state instead.

Note that when calculating the number of UDVM cycles the STATE-ACCESS instruction costs (1 + state_length) cycles. The value of state_length MUST be taken from the returned item of state in the case that the state_length operand is set to 0.

The state_begin and state_length operands define the starting byte and number of bytes to copy from the state_value contained in the returned item of state. Decompression failure occurs if bytes are copied from beyond the end of the state_value. Note that decompression failure will always occur if the state_length operand is set to 0 but the state_begin operand is non-zero.

The state_address operand contains a UDVM memory address. The requested portion of the state_value is byte copied to this memory address using the rules of Section 8.4.

Program execution then resumes at the memory address specified by state_instruction, unless this address is 0 in which case program execution resumes at the next instruction following the STATE-ACCESS instruction. Note that the latter case only occurs if both the state_instruction operand and the state_instruction value from the requested state are set to 0.

9.4.6. STATE-CREATE

The STATE-CREATE instruction requests the creation of a state item at the receiving endpoint.

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

Note that the new state item cannot be created until a valid compartment identifier has been returned by the application. Consequently, when a STATE-CREATE instruction is encountered the UDVM simply buffers the five supplied operands until the END-MESSAGE instruction is reached. The steps taken at this point are described in Section 9.4.9.

Decompression failure MUST occur if more than four state creation requests are made before the END-MESSAGE instruction is encountered. Decompression failure also occurs if the minimum_access_length does not lie between 6 and 20 inclusive, or if the state_retention_priority is 65535.

9.4.7. STATE-FREE

The STATE-FREE instruction informs the receiving endpoint that the sender no longer wishes to use a particular state item.

STATE-FREE (%partial_identifier_start, %partial_identifier_length)

Note that the STATE-FREE instruction does not automatically delete a state item, but instead reclaims the memory taken by the state item within a certain compartment, which is generally not known before the END-MESSAGE instruction is reached. So just as for the STATE-CREATE instruction, when a STATE-FREE instruction is encountered the UDVM simply buffers the two supplied operands until the END-MESSAGE instruction is reached. The steps taken at this point are described in Section 9.4.9.

Decompression failure MUST occur if more than four state free requests are made before the END-MESSAGE instruction is encountered. Decompression failure also occurs if partial_identifier_length does not lie between 6 and 20 inclusive.

9.4.8. OUTPUT

The OUTPUT instruction provides successfully decompressed data to the dispatcher.

OUTPUT (%output_start, %output_length)

The operands define the starting memory address and length of the byte string to be provided to the dispatcher. Note that the OUTPUT instruction can be used to output a partially decompressed message; each time the instruction is encountered it provides a new byte string that the dispatcher appends to the end of any bytes previously passed to the dispatcher via the OUTPUT instruction.

The string of data is byte copied from the UDVM memory obeying the rules of Section 8.4.

Decompression failure occurs if the cumulative number of bytes provided to the dispatcher exceeds 65536 bytes.

Since there is technically a difference between outputting a 0-byte decompressed message, and not outputting a decompressed message at all, the OUTPUT instruction needs to distinguish between the two cases. Thus, if the UDVM terminates before encountering an OUTPUT instruction it is considered not to have outputted a decompressed message. If it encounters one or more OUTPUT instructions, each of which provides 0 bytes of data to the dispatcher, then it is considered to have outputted a 0-byte decompressed message.

9.4.9. END-MESSAGE

The END-MESSAGE instruction successfully terminates the UDVM and forwards the state creation and state free requests to the state handler together with any supplied feedback data.

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

When the END-MESSAGE instruction is encountered, the decompressor dispatcher indicates to the application that a complete message has been decompressed. The application may return a compartment identifier, which the UDVM forwards to the state handler together with the state creation and state free requests and any supplied feedback data.

The actual decompressed message is outputted separately using the OUTPUT instruction; this conserves memory at the UDVM because there is no need to buffer an entire decompressed message before it can be passed to the dispatcher.

The END-MESSAGE instruction may pass up to four state creation requests and up to four state free requests to the state handler. The requests are passed to the state handler in the same order as they are made; in particular it is possible for the state creation requests and the state free requests to be interleaved.

The state creation requests are made by the STATE-CREATE instruction. Note however that the END-MESSAGE can make one state creation request itself using the supplied operands. If the specified minimum_access_length does not lie between 6 and 20 inclusive, or if the state_retention_priority is 65535 then the END-MESSAGE instruction fails to make a state creation request of its own (however decompression failure does not occur and the state creation requests made by the STATE-CREATE instruction are still valid).

Note that there is a maximum limit of four state creation requests per instance of the UDVM. Therefore, decompression failure occurs if the END-MESSAGE instruction makes a state creation request and four instances of the STATE-CREATE instruction have already been encountered.

When creating a state item it is necessary to give the state_length, state address, state_instruction and minimum_access_length; these are supplied as operands in the STATE-CREATE instruction (or the END-MESSAGE instruction). A complete item of state also requires a state_value and a state_identifier, which are derived as follows:

The UDVM byte copies a string of state_length bytes from the UDVM memory beginning at state_address (obeying the rules of Section 8.4). This is the state_value.

The UDVM then calculates a 20-byte SHA-1 hash [RFC-3174] over the byte string formed by concatenating the state_length, state_address, state_instruction, minimum_access_length and state_value (in the order given). This is the state_identifier.

The state_retention_priority is not part of the state item itself, but instead determines the order in which state will be deleted when the compartment exceeds its allocated state memory. The state_retention_priority is supplied as an operand in the STATE-CREATE or END-MESSAGE instruction and is passed to the state handler as part of each state creation request.

The state free requests are made by the STATE-FREE instruction. Each STATE-FREE instruction supplies the values partial_identifier_start and partial_identifier_length; upon reaching the END-MESSAGE instruction these values are used to byte copy a partial state identifier from the UDVM memory. If no state item matching the partial state identifier can be found or if more than one state item in the compartment matches the partial state identifier, then the state free request is ignored (this does not cause decompression failure to occur). Otherwise, the state handler frees the matched state item as specified in Section 6.2.

As well as forwarding the state creation and state free requests, the END-MESSAGE instruction may also pass feedback data to the state handler. Feedback data is used to inform the receiving endpoint about the capabilities of the sending endpoint, which can help to improve the overall compression ratio and to reduce the working memory requirements of the endpoints.

Two types of feedback data are available: requested feedback and returned feedback. The format of the requested feedback data is given in Figure 12. As outlined in Section 3.2, the requested feedback data can be used to influence the contents of the returned feedback data in the reverse direction.

The returned feedback data is itself subdivided into a returned feedback item and a list of returned SigComp parameters. The returned feedback item is of sufficient importance to warrant its own field in the SigComp header as described in Section 7.1. The returned SigComp parameters are illustrated in Figure 13.

Note that the formats of Figure 12 and Figure 13 are only for local presentation of the feedback data on the interface between the UDVM and state handler. The formats do not mandate any bits on the wire; the compressor can transmit the data in any form provided that it is loaded into the UDVM memory at the correct addresses.

Moreover, the responsibility for ensuring that feedback data arrives successfully over an unreliable transport lies with the sender. The receiving endpoint always uses the last received value for each field in the feedback data, even if the values are out of date due to packet loss or misordering.

If the requested_feedback_location operand is set to 0, then no feedback request is made; otherwise, it points to the starting memory address of the requested feedback data as shown in Figure 12.

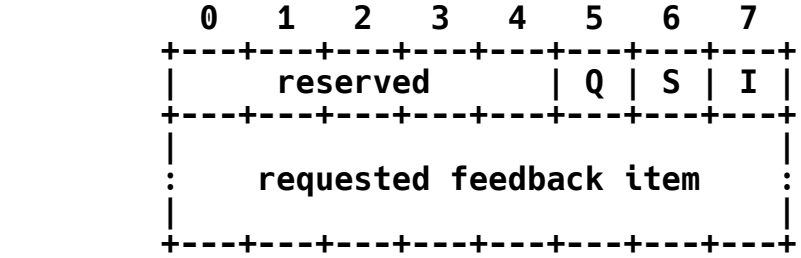


Figure 12: Format of requested feedback data

The reserved bits may be used in future versions of SigComp, and are set to 0 in Version 0x01. Non-zero values should be ignored by the receiving endpoint.

The Q-bit indicates whether a requested feedback item is present or not. The compressor can set the requested feedback item to an arbitrary value, which will then be transmitted unmodified in the reverse direction as a returned feedback item. See Chapter 5 for further details of how the requested feedback item is returned.

The format of the requested feedback item is identical to the format of the returned feedback item illustrated in Figure 4.

The compressor sets the S-bit to 1 if it does not wish (or no longer wishes) to save state information at the receiving endpoint and also does not wish to access state information that it has previously saved. Consequently, if the S-bit is set to 1 then the receiving endpoint can reclaim the state memory allocated to the remote compressor and set the state_memory_size for the compartment to 0.

The compressor may change its mind and switch the S-bit back to 0 in a later message. However, the receiving endpoint is under no obligation to use the original state_memory_size for the compartment; it may choose to allocate less memory to the compartment or possibly none at all.

Similarly the compressor sets the I-bit to 1 if it does not wish (or no longer wishes) to access any of the locally available state items offered by the receiving endpoint. This can help to conserve bandwidth because the list of locally available state items no longer needs to be returned in the reverse direction. It may also conserve memory at the receiving endpoint, as the state handler can delete any locally available state items that it determines are no longer required by any remote endpoint. Note that the compressor can set the I-bit back to 0 in a later message, but it cannot access any locally available state items that were previously offered by the receiving endpoint unless they are subsequently re-announced.

If the returned_parameters_location operand is set to 0, then no SigComp parameters are returned; otherwise, it points to the starting memory address of the returned parameters as shown in Figure 13.

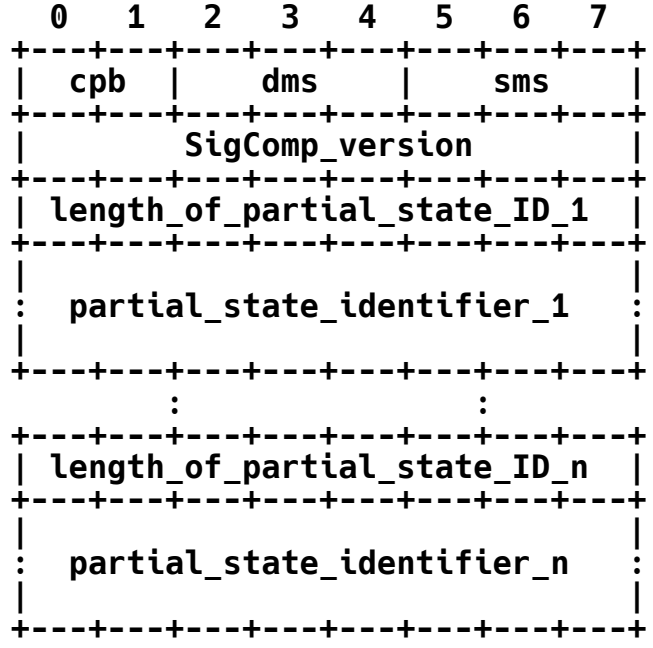


Figure 13: Format of returned SigComp parameters

The first byte encodes the SigComp parameters cycles_per_bit, decompression_memory_size and state_memory_size as per Section 3.3.1. The byte can be set to 0 if the three parameters are not included in the feedback data. (This may be useful to save bits in the compressed message if the remote endpoint is already satisfied all necessary information has reached the endpoint receiving the message.)

The second byte encodes the SigComp_version as per Section 3.3.2. Similar to the first byte, the second byte can be set to 0 if the parameter is not included in the feedback data.

The remaining bytes encode a list of partial state identifiers for the locally available state items offered by the sending endpoint. Each state item is encoded as a 1-byte length field, followed by a partial state identifier containing as many bytes as indicated in the length field. The sender can choose to send as few as 6 bytes if it believes that this is sufficient for the receiver to determine which state item is being offered.

The list of state identifiers is terminated by a byte in the position where the next length field would be expected that is set to a value below 6 or above 20. Note that upgraded SigComp versions may append additional items of data after the final length field.

10. Security Considerations

10.1. Security Goals

The overall security goal of the SigComp architecture is to not create risks that are in addition to those already present in the application protocols. There is no intention for SigComp to enhance the security of the application, as it always can be circumvented by not using compression. More specifically, the high-level security goals can be described as:

1. Do not worsen security of existing application protocol

2. Do not create any new security issues

3. Do not hinder deployment of application security.

10.2. Security Risks and Mitigation

This section identifies the potential security risks associated with SigComp, and explains how each risk is minimized by the scheme.

10.2.1. Confidentiality Risks

- Attacking SigComp by snooping into state of other users:

State is accessed by supplying a state identifier, which is a cryptographic hash of the state being referenced. This implies that the referencing message already needs knowledge about the state. To enforce this, a state item cannot be accessed without supplying a minimum of 48 bits from the hash. This also minimizes the probability of an accidental state collision. A compressor can, using the minimum_access_length operand of the STATE-CREATE and END-MESSAGE instructions, increase the number of bits that need to be supplied to access the state, increasing the protection against attacks.

Generally, ways to obtain knowledge about the state identifier (e.g., passive attacks) will also easily provide knowledge about the referenced state, so no new vulnerability results.

An endpoint needs to handle state identifiers with the same care it would handle the state itself.

10.2.2. Integrity Risks

The SigComp approach assumes that there is appropriate integrity protection below and/or above the SigComp layer. The state creation mechanism provides some additional potential to compromise the integrity of the messages; however, this would most likely be detectable at the application layer.

- Attacking SigComp by faking state or making unauthorized changes to state:

State cannot be destroyed by a malicious sender unless it can send messages that the application identifies as belonging to the same compartment the state was created under; this adds additional security risks only when the application allows the installation of SigComp state from a message where it would not have installed state itself.

Faking or changing state is only possible if the hash allows intentional collision.

10.2.3. Availability Risks (Avoiding DoS Vulnerabilities)

- Use of SigComp as a tool in a DoS attack to another target:

SigComp cannot easily be used as an amplifier in a reflection attack, as it only generates one decompressed message per incoming compressed message. This message is then handed to the application; the utility as a reflection amplifier is therefore limited by the utility of the application for this purpose.

However, it must be noted that SigComp can be used to generate larger messages as input to the application than have to be sent from the malicious sender; this therefore can send smaller messages (at a lower bandwidth) than are delivered to the application. Depending on the reflection characteristics of the application, this can be considered a mild form of amplification. The application MUST limit the number of packets reflected to a potential target - even if SigComp is used to generate a large amount of information from a small incoming attack packet.

- Attacking SigComp as the DoS target by filling it with state:

Excessive state can only be installed by a malicious sender (or a set of malicious senders) with the consent of the application. The system consisting of SigComp and application is thus approximately as vulnerable as the application itself, unless it allows the installation of SigComp state from a message where it would not have installed application state itself.

If this is desirable to increase the compression ratio, the effect can be mitigated by making use of feedback at the application level that indicates whether the state requested was actually installed - this allows a system under attack to gracefully degrade by no longer installing compressor state that is not matched by application state.

Obviously, if a stream-based transport is used, the streams themselves constitute state that has to be handled in the same way that the application itself would handle a stream-based transport; if an application is not equipped for stream-based transport, it should not allow SigComp connections on a stream-based transport. For the alternative SigComp usage described as "continuous mode" in Section 4.2.1, an attacker could create any number of active UDVMs unless there is some DoS protection at a lower level (e.g., by using TLS in appropriate configurations).

- Attacking the UDVM by faking state or making unauthorized changes to state:

This is covered in Section 10.2.2.

- Attacking the UDVM by sending it looping code:

The application sets an upper limit to the number of "UDVM cycles" that can be used per compressed message and per input bit in the compressed message. The damage inflicted by sending packets with looping code is therefore limited, although this may still be substantial if a large number of UDVM cycles are offered by the UDVM. However, this would be true for any decompressor that can receive packets over an unsecured transport.

11. IANA Considerations

SigComp requires a 1-byte name space, the SigComp_version, which has been created by the IANA. Upgraded versions of SigComp must be backwards-compatible with Version 0x01, described in this document. Adding additional UDVM instructions and assigning values to the reserved UDVM memory addresses are two possible upgrades for which this is the case.

Following the policies outlined in [RFC-2434], the IANA policy for assigning a new value for the SigComp_version shall require a Standards Action. Values are thus assigned only for Standards Track RFCs approved by the IESG.

12. Acknowledgements

Thanks to

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
Robert Sugar

for valuable input and review.

13. References

13.1. Normative References

[RFC-1662] Simpson, W., "PPP in HDLC-like Framing", STD 51, RFC 1662, July 1994.

[RFC-2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC-3174] Eastlake, 3rd, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001.

13.2. Informative References

[RFC-1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC-2026] Bradner, S., "The Internet Standards Process - Revision 3", BCP 9, RFC 2026, October 1996.

[RFC-2279] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

[RFC-2326] Schulzrinne, H., Rao, A. and R. Lanphier, "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998.

[RFC-2434] Alvestrand, H. and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC-2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwartzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

[RFC-3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC-3321] Hannu, H., Christoffersson, J., Forsgren, S., Leung, K.-C., Liu, Z. and R. Price, "Signaling Compression (SigComp) - Extended Operations", RFC 3321, January 2003.

14. Authors' Addresses

Richard Price
Roke Manor Research Ltd
Romsey, Hants, SO51 0ZN
United Kingdom

Phone: +44 1794 833681
EMail: richard.price@roke.co.uk

Carsten Bormann
Universitaet Bremen TZI
Postfach 330440
D-28334 Bremen, Germany

Phone: +49 421 218 7024
EMail: cabo@tzi.org

Jan Christoffersson
Box 920
Ericsson AB
SE-971 28 Lulea, Sweden

Phone: +46 920 20 28 40
EMail: jan.christoffersson@epl.ericsson.se

Hans Hannu
Box 920
Ericsson AB
SE-971 28 Lulea, Sweden

Phone: +46 920 20 21 84
EMail: hans.hannu@epl.ericsson.se

Zhigang Liu
Nokia Research Center
6000 Connection Drive
Irving, TX 75039

Phone: +1 972 894-5935
EMail: zhigang.c.liu@nokia.com

Jonathan Rosenberg
dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936

EMail: jdrosen@dynamicsoft.com

15. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.