International Tactical Radio Security Services API Specification

Document WINNF-TS-0011
(formerly WINNF-09-S-0011)

Version V1.0.0
11 February 2021

TERMS, CONDITIONS & NOTICES

This document has been prepared by the International Radio Security Services API Task Group to assist The Software Defined Radio Forum Inc. (or its successors or assigns, hereafter "the Forum"). It may be amended or withdrawn at a later time and it is not binding on any member of the Forum or of the International Radio Security Services API Task Group.

Contributors to this document that have submitted copyrighted materials (the Submission) to the Forum for use in this document retain copyright ownership of their original work, while at the same time granting the Forum a non-exclusive, irrevocable, worldwide, perpetual, royalty-free license under the Submitter's copyrights in the Submission to reproduce, distribute, publish, display, perform, and create derivative works of the Submission based on that original work for the purpose of developing this document under the Forum's own copyright.

Permission is granted to the Forum's participants to copy any portion of this document for legitimate purposes of the Forum. Copying for monetary gain or for other non-Forum related purposes is prohibited.

THIS DOCUMENT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY USE OF THIS SPECIFICATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE FORUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER, DIRECTLY OR INDIRECTLY, ARISING FROM THE USE OF THIS DOCUMENT.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the specification set forth in this document, and to provide supporting documentation.

The document was developed following the Forum's policy on restricted or controlled information (Policy 009) to ensure that that the document can be shared openly with other member organizations around the world. Additional Information on this policy can be found here: http://www.wirelessinnovation.org/page/Policies_and_Procedures. Although this document contains no restricted or controlled information, the specific implementation of concepts contained herein may be controlled under the laws of the country of origin for that implementation. Readers are encouraged, therefore, to consult with a cognizant authority prior to any further development.

Wireless Innovation Forum ™ and SDR Forum ™ are trademarks of the Software Defined Radio Forum Inc.

Table of Contents

List of Figures

List of Tables

International Tactical Radio Security Services API Specification

1 Introduction

The *International Tactical Radio Security Services (IRSS) API* standardizes a software security interface for use by the international tactical radio community. In particular, this API is targeted for deployment in tactical radio systems based on the Software Communication Architecture (SCA), though that is not necessarily a prerequisite for its use. In its current increment, the intent of this API is to promote waveform (WF) portability between various radio platforms that provide the API. As such, the focus of this API is on the security interfaces required to meet waveform needs. Although working systems require additional platform security interfaces to fulfill a number of needs, such as keyfill, security policies, etcetera, standardizing such interfaces does not add to waveform portability. Additionally, it is at the platform level where the variation is expected to be the highest across the international community, making such standardization difficult. As such, platform security interfaces will only be detailed where there is overlap with waveform security interfaces.

The IRSS API consists of several API service groups, as follows:

- The *control service group* details interfaces used to establish, configure, and otherwise manage channels for services provided by this API.
- The *Infosec service group* details interfaces for usage of cryptographic channels and TRANSEC channels. Cryptographic channels are used for transformation *(i.e. encryption/decrypt*ion) of user information between security domains or within a single security domain. TRANSEC channels are typically used to protect the protocol used for transmissions (compared with the traffic payload itself).
- The *bypass service group* details interfaces for usage of bypass channels used to transfer waveform control information between security domains without encryption.
- The *integrity and authentication service group* details interfaces for features such as generating hashes, generating message authentication codes (MACs), generating and verifying digital signatures, and generating random numbers.
- The *protocol service group* details interfaces that allow waveforms to interact with Cryptographic Applications (CAs), using a generic protocol to perform CA-specific functions. This allows specialized protocols or functions not addressed by the other IRSS APIs to be performed, such as asymmetric key negotiation, etc.

1.1 Overview

The contents of the document are laid out as follows:

- Section 1, *Introduction*, contains the introductory material regarding the overview, service layer description, modes, states and referenced documents of this document.
- Section 2, *Services*, specifies the interfaces for the component, port connections, and sequence diagrams.
- Section 3,

Service Primitives and Attributes, specifies the operations that are provided by the IRSS API.

- Section 4,

IDL details the IDL for the IRSS API.

- Section 5,

UML, depicts the component UML and details the data types used within the IRSS API.

1.2 Service Group Descriptions

1.2.1 Concepts and usage overview

While the IRSS API standardizes a number of interfaces for performing security functions on a radio set, by necessity the underlying algorithms and their specific configuration are intentionally generic. To bind this generic API to specific behaviors, the concept of *cryptographic applications (CAs)* is used. A CA provides a unique set of services that are specific for a particular cryptographic protocol or cryptographic algorithm. For example, an "AES CFB" CA is an example of a generic AES engine that streaming waveforms could utilize, while an "IPSEC" CA is an example of a complex CA that internally supports multiple algorithms with per-packet dynamic selection. How CAs become resident in a Cryptographic SubSystem (CSS) is implementation specific – some CSSs will be prebuilt with all CAs they support, while others may support the concept of installable CAs. Regardless of the means, a waveform references a CA by using a platform-specific ID.

Most IRSS services are employed using a multistep process as follows:

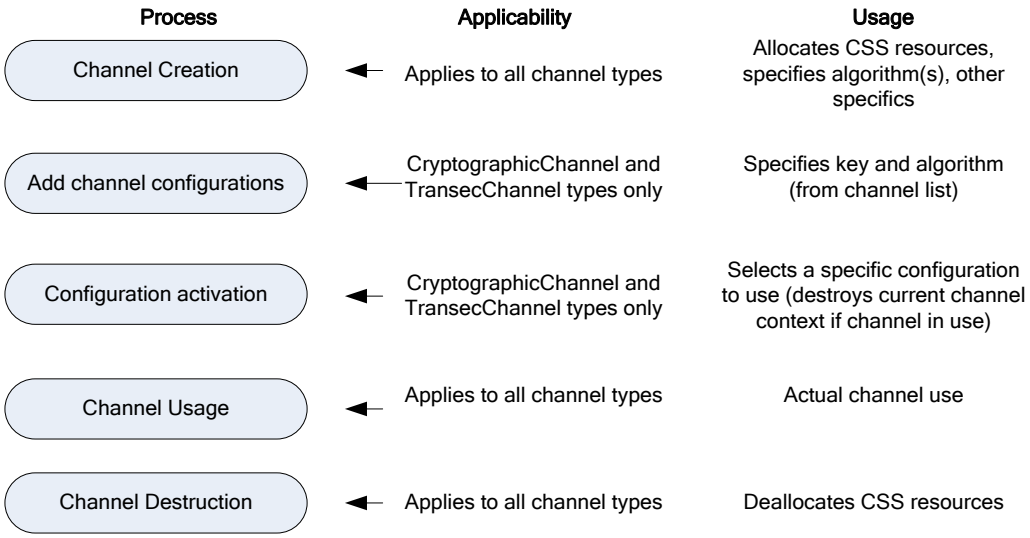

Figure 1: Channel lifecycle

1.2.2 Platform implementation of interfaces and operations

This standard defines a number of normative interfaces for waveform use to perform security functions relevant to the waveform. However, across the broad software-radio domain, there is no universal agreement or standardization on which specific functions should be performed by waveforms, platforms or possibly both. In the process of forming this standard, a variety of use-cases were examined, with the union of individual waveform needs considered in determining which operations and functions to include.

At the same time, the security requirements of individual Software Defined Radio (SDR) platforms may vary, and in the general case, it may be inappropriate for all operations to be available to the waveform. In such a case, several avenues are available to implementers realizing the interfaces in this specification:

- **Do not implement or connect a specific interface**: for example, some platforms do not allow a waveform to create or manage cryptographic channels – only use them. In this case, an implementer could choose to not implement the interface at all, or not connect the interface to a waveform port. In this case, presumably the platform would make the equivalent capability available to the platform instead, as channels still need to be created and managed, with the platform passing the channelIds to the waveform for use.
- **Dissallow one or more operations within an API:** In this case, the platform would either not implement one or more of the operations in an API, or, based on some platform policy (possibly waveform specific) disallow execution of the operation. In both cases an appropriate exception would be returned.

High assurance platforms typically implement a variety of policies that govern operation of the CSS and waveform's use of security services. As these policies tend to be very domain and country specific, this standard does not address them, even in a generic fashion. The implementation of any such policies is assumed to be a radio platform function and not needed by waveforms themselves.

1.2.3 IRSS API Port Definitions / Connections

Being a broad-spectrum standard, the IRSS API specifies a set of interfaces and their semantics without standardizing how these interfaces are allocated to components on a given radio set, nor how these components are distributed across security domains (SDs). SDs provide compartmentalization of information across cryptographic boundaries, with these boundaries being separated by a CSS.

In this section, several typical IRSS component port layouts are shown (single and double security domains), but many other configurations and topologies are possible, with dimensions spreading across multiple radio channels and multiple SDs. Each radio set implementing these APIs is expected to detail its specific port layout as part of its design documentation.

Figure 2 shows the port connections for an IRSS component in a single security domain. In this case, both plaintext (non-encrypted) and ciphertext (encrypted) information are presented through a common IRSS component. As a result, ports using the bypass module interfaces are not included as there is "nothing to bypass around". Contrast this to the double sided case, where the only way to send control data across the CSS is to use a bypass interface.

In the diagrams below and in this standard in general, port names are provided for reference only, and not normative.

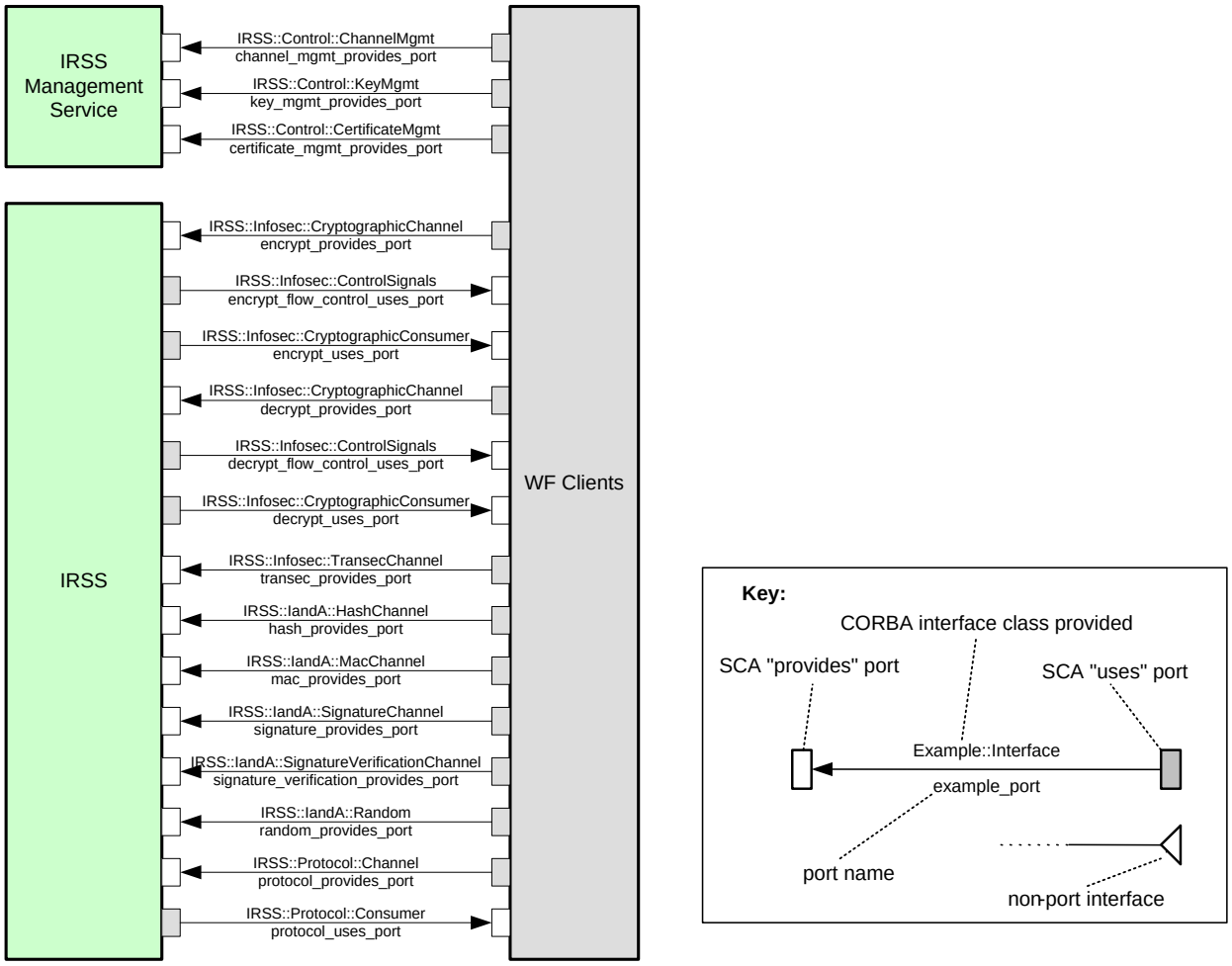

Figure 2 - IRSS Port Diagram - Single Security Domain

Figure 3 shows the port connections for an IRSS component in a typical high-assurance, dual security domain platform that realizes the IRSS API. In this case, the CSS formally separates plaintext (non-encrypted) information from ciphertext (encrypted) information. The waveform interfaces to this CSS are through two distinct components, each which implement some parts of the IRSS APIs. Note that in this example below, the ChannelManagement interface is presented to the plaintext side only. While this is typical, it is not normative, and other implementations could place this on the ciphertext side. Note that when compared with the single-sided arrangement in Figure 2, it becomes necessary to bypass control messages between plaintext and ciphertext sides. To support this, the Bypass interface is employed.

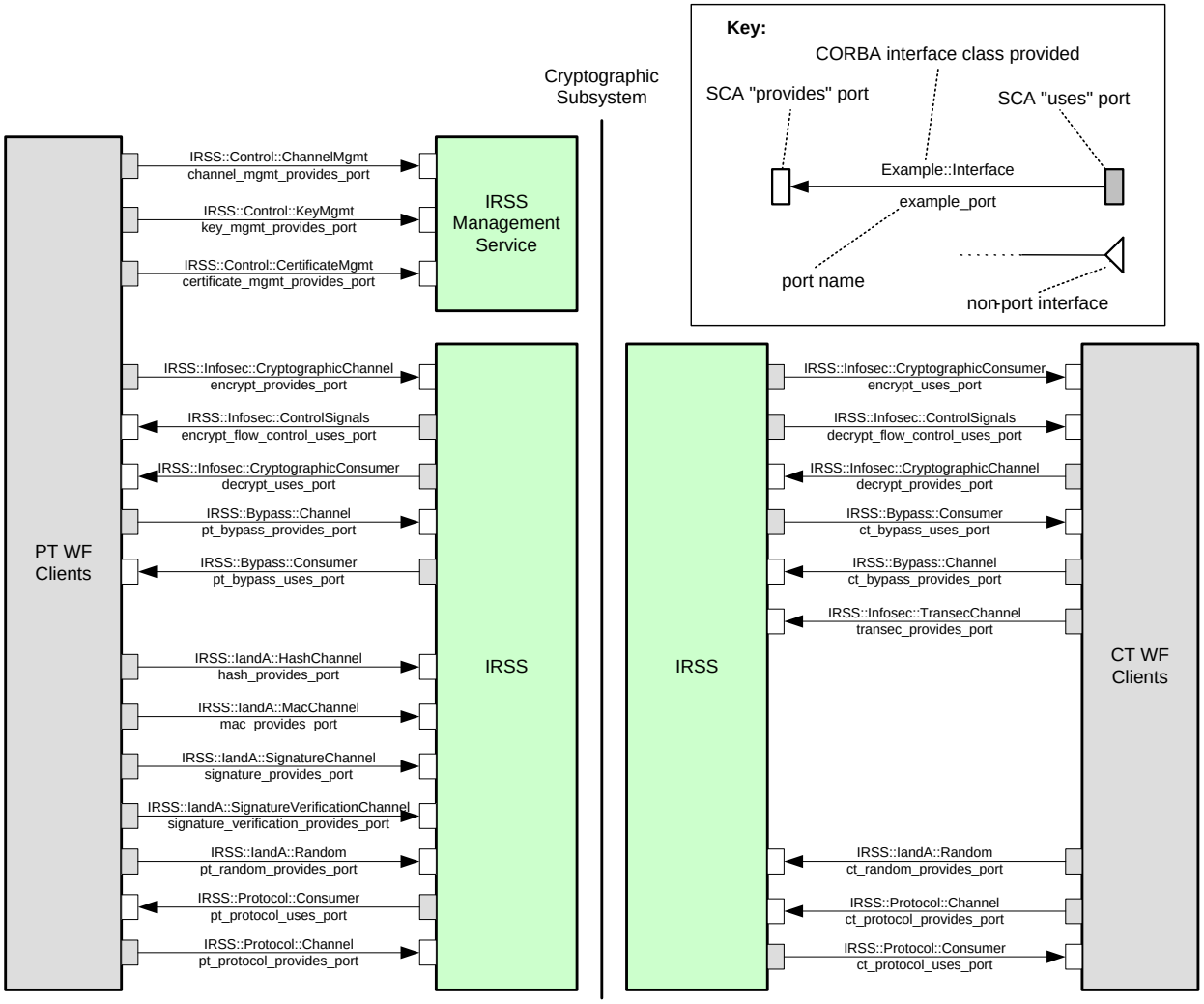

Figure 3 - IRSS Port Diagram - Two Security Domains

IRSS API Provides Port Definitions

channel_mgmt_provides_port is provided by the *IRSS* to allow a waveform or OE component to create, configure, and manage channels.

key_mgmt_provides_port is provided by the *IRSS* to allow a waveform or OE component to request key management operations.

certificate_mgmt_provides_port is provided by the *IRSS* to allow a waveform or OE component to retrieve and validate certificates.

encrypt_provides_port is provided by the *IRSS* to allow a waveform to request that a packet of data be encrypted by using the transform methods. Other methods allow the client to query the maximum packet and maximum payload sizes supported by the interface. The return values from the transform operations and the SpaceAvailable() method provide for flow control.

decrypt_provides_port is provided by the *IRSS* to allow a waveform to request that a packet of data be decrypted by using the transform methods. Other methods allow the client to query the maximum packet and maximum payload sizes supported by the interface. The return values from the transform operations and the SpaceAvailable() method provide for flow control.

transec_provides_port is provided by the *IRSS* to allow a client to encrypt or decrypt a TRANSEC payload. It also allows a client to generate keystream.

pt_bypass_provides_port and **ct_bypass_provides_port** are provided by the *IRSS* to allow a client to push a bypass message through the crypto module.

hash_provides_port is provided by the *IRSS* to allow a client to request the generation of a hash and have the hash returned.

mac_provides_port is provided by the *IRSS* to allow a client to request the computation of a MAC and have the MAC returned. It also allows a client to verify a MAC.

signature_provides_port is provided by the *IRSS* to allow a client to request the generation of a digital signature and have the signature returned.

signature_verification_provides_port is provided by the *IRSS* to allow a client to request the verification of a digital signature.

random_provides_port, pt_random_provides_port, and **ct_random_provides_port** are provided by the *IRSS* to allow a client to request the generation of true random numbers or pseudo random numbers.

protocol_provides_port, pt_protocol_provides_port, and **ct_protocol_provides_port** are provided by the *IRSS* to allow a client to push protocol messages to the IRSS.

IRSS API Uses Port Definitions

encrypt_uses_port is used by the *IRSS* to push data to a client after an encryption operation successfully completes. This port does not provide for any flow control.

encrypt_flow_control_uses_port is used by the *IRSS* to inform the client that the previously paused encryption flow may resume.

decrypt_uses_port is used by the *IRSS* to push data to a client after a decryption operation successfully completes. This port does not provide for any flow control.

decrypt_flow_control_uses_port is used by the *IRSS* to inform the client that the previously paused decryption flow may resume.

protocol_uses_port, pt_protocol_uses_port, and **ct_protocol_uses_port** are used by the *IRSS* to push protocol messages to a client.

pt_bypass_uses_port and **ct_bypass_uses_port** are used by the *IRSS* to push information that was bypassed through the crypto module to a client.

1.3 Modes of Service

Not applicable.

1.4 Service States

Not applicable.

1.5 Referenced Documents

[1] JTRS Standard, "Software Communications Architecture (SCA)," JPEO, Version 2.2.2.

[2] RFC 3280, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", IETF, http://www.ietf.org/rfc/rfc3280.txt

2 Services

2.1 Provide Services

Table 1 - IRSS API Uses Service Interface

Service Group	Port Name	Service (Interface Used)	Primitives (Used)
Bypass	ct_bypass_provides_port, pt_bypass_provides_port	IRSS::Bypass::Channel	PushBypass()
			GetMaxBypassSize()
Control	channel_mgmt_provides_port	IRSS::Control::ChannelMgmt	CreateCryptographicChannel()
			CreateTransecChannel()
			CreateBypassChannel()
			CreateHashChannel()
			CreateMacChannel()
			CreateSignatureChannel()
			CreateSignatureVerificationChannel()
			CreateProtocolChanel()
			DestroyChannel()
			AddCryptographicConfiguration()
			AddTransecConfiguration()
			RemoveConfiguration()
			ActivateConfiguration()
			DeactivateConfiguration()
	key_mgmt_provides_port	IRSS::Control::KeyMgmt	UpdateKey()
			UpdateKeyWithAlgorithm()
			GetUpdateCount()
			ZeroizeKey()

Service Group	Port Name	Service (Interface Used)	Primitives (Used)
	certificate_mgmt_provides_port	IRSS::Control::CertificateMgmt	RetrieveCertificate()
			GetCertificateIds()
			IsCertificateValid()
Infosec	encrypt_provides_port, decrypt_provides_port	IRSS:Infosec::CryptographicChannel	TransformPackets()
			TransformStream()
			GetMaxPayloadSize()
			GetMaxPacketSize()
			SpaceAvailable()
	transec_provides_port	IRSS::Infosec::TransecChannel	EncryptTransec()
			DecryptTransec()
			GenerateKeyStream()
			GetMaxPayloadSize()
IandA	hash_provides_port	IRSS::IandA::HashChannel	GetMaxDataSize()
			Reset()
			GetHash()
			PushData()
	mac_provides_port	IRSS::IandA::MacChannel	GetMaxDataSize()
			Reset()
			GetMac()
			IsMacValid()
			PushData()
	signature_provides_port	IRSS::IandA::SignatureChannel	GetMaxDataSize()
			Reset()
			GetSignature()
			PushData()
	signature_verification_provides_port	IRSS::IandA::SignatureVerificationChannel	GetMaxDataSize()
			Reset()
			IsSignatureValid()
			PushData()
	random_provides_port, ct_random_provides_port, pt_random_provides_port	IRSS::IandA::Random	GetPseudoRandom()
			GetRandom()
Protocol	protocol_provides_port, ct_protocol_provides_port, pt_protocol_provides_port	IRSS::Protocol::Channel	PushMessage()

2.2 Use Services

Table 2 - IRSS API Uses Service Interface

Service Group	Port Name	Service (Interface Used)	Primitives (Used)
Bypass	ct_bypass_uses_port, pt_bypass_uses_port	IRSS::Bypass::Consumer	PushBypass()
Infosec	encrypt_flow_control_uses_port, decrypt_flow_control_uses_port	IRSS::Infosec::ControlSignals	FlowResume()

Service Group	Port Name	Service (Interface Used)	Primitives (Used)
	encrypt_uses_port, decrypt_uses_port	IRSS::Infosec::CryptographicConsumer	Pushstream()
			PushPackets()
Protocol	protocol_uses_port, ct_protocol_uses_port, pt_protocol_uses_port	IRSS::Protocol::Consumer	PushMessage()

2.3 Interface Modules

2.3.1 IRSS::Control

2.3.1.1 IRSS::Control::CertificateMgmt Interface Description

The IRSS::Control::CertificateMgmt interface provides the means for waveforms to access certificates that are currently being managed by the IRSS, and to validate new certificates. A client can use GetCertificateIds() to retrieve the IDs for the certificates that have been loaded into, and are managed by, the IRSS. With these IDs the RetrieveCertificate() operation returns the public portion of the certificate (i.e. it does not include the private key). Waveform clients will also need to validate received certificates. Assuming the necessary trust anchors have been previously loaded onto the platform, a client can use ValidateCertificate() to pass in and validate a certificate received from a peer.

The IRSS::Control::CertificateMgmt interface is shown in Figure 4.

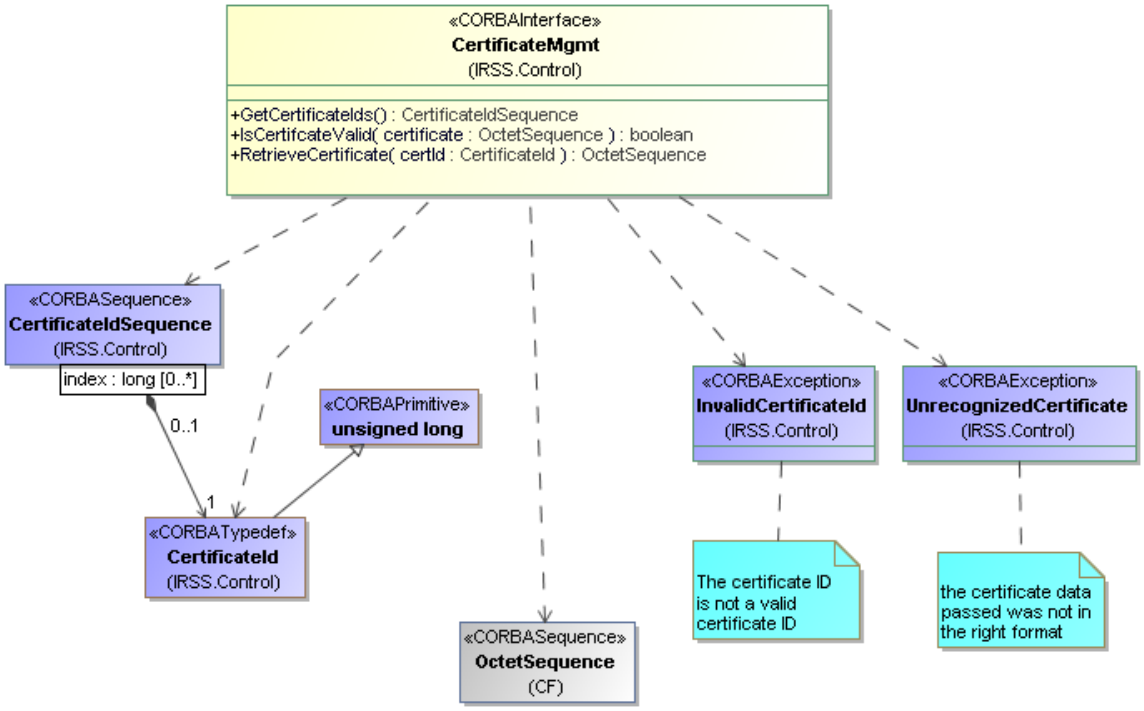

Figure 4 - Control::CertificateMgmt Interface

2.3.1.2 IRSS::Control::ChannelMgmt Interface Description

Many operations offered by the IRSS API are performed on *channels* which define a communication path between a waveform client and the CSS. Waveform clients use the IRSS::Control::ChannelMgmt interface to create and manage channels. There are various types of channels that clients can create:

- Cryptographic channels are used to transform (i.e. encrypt and decrypt) data
- Transec channels are used to cover protocol or other transmission information
- Bypass channels are used to bypass control information through the CSS
- Hash channels are used to generate a hash over data
- MAC channels are used to generate and verify a MAC over data,
- Signature channels are used to generate a signature over data
- Signature verification channels are used to verify a signature
- Protocol channels are used to send and receive protocol messages to/from the cryptographic subsystem (for example, as part of a key exchange protocol).

Channels are created on a specific crypto module[1] using specific endpoints that define the inputs and, where applicable, the outputs of the channel. The definition for an endpoint is implementation defined. For example, one could choose to use endpoints for each HW interface. Alternatively, one could choose to use endpoints for each API instance. When a waveform is ported between platforms, the values supplied to these parameters will in general need to be changed.

In many platforms, channel creation will allocate specific CSS resources for use, with subsequent deallocation of these resources on channel destruction. To be able to determine which resources are needed, specific channel types use the information supplied with the createXXX() operation – for example, for cryptographic channels, a list of all required cryptographic applications and duplexity is required. This pre-allocation guarantees that (in non-exceptional cases) once channel operation succeeds, all operations on the channel can be performed.

With the exception of Cryptographic channels and TRANSEC channels, channels are ready to use once created. Cryptographic channels and TRANSEC channels additionally need to be configured (via AddCryptographicConfiguration() or AddTransecConfiguration()) and activated (via ActivateConfiguration()) before they are ready to use. These operations allow fast switching of configurations within the lifecycle of a channel without risk of an allocation failure.

When created, cryptographic channels and TRANSEC channels establish a context which is shared between all the configurations on that channel. Switching between configurations on these channels (via ActivateConfiguration()) will destroy any previous state maintained for the channel and establish a new state for the new configuration. Multiple cryptographic/TRANSEC channels can be created between the same set of endpoints with each channel establishing its

[1] The concept of a crypto module, which typically refers to hardware function supporting cryptographic functions, is not standardized, and is considered platform-dependent. Some systems will have only one module, while others may have multiples.

own context[2]. Switching between channels will not destroy the state of the previous channel, allowing that state to be used further.

The lifecycle of channels are summarized in Figure 1, and the UML for the IRSS::Control::ChannelMgmt interface is shown in Figure 5.

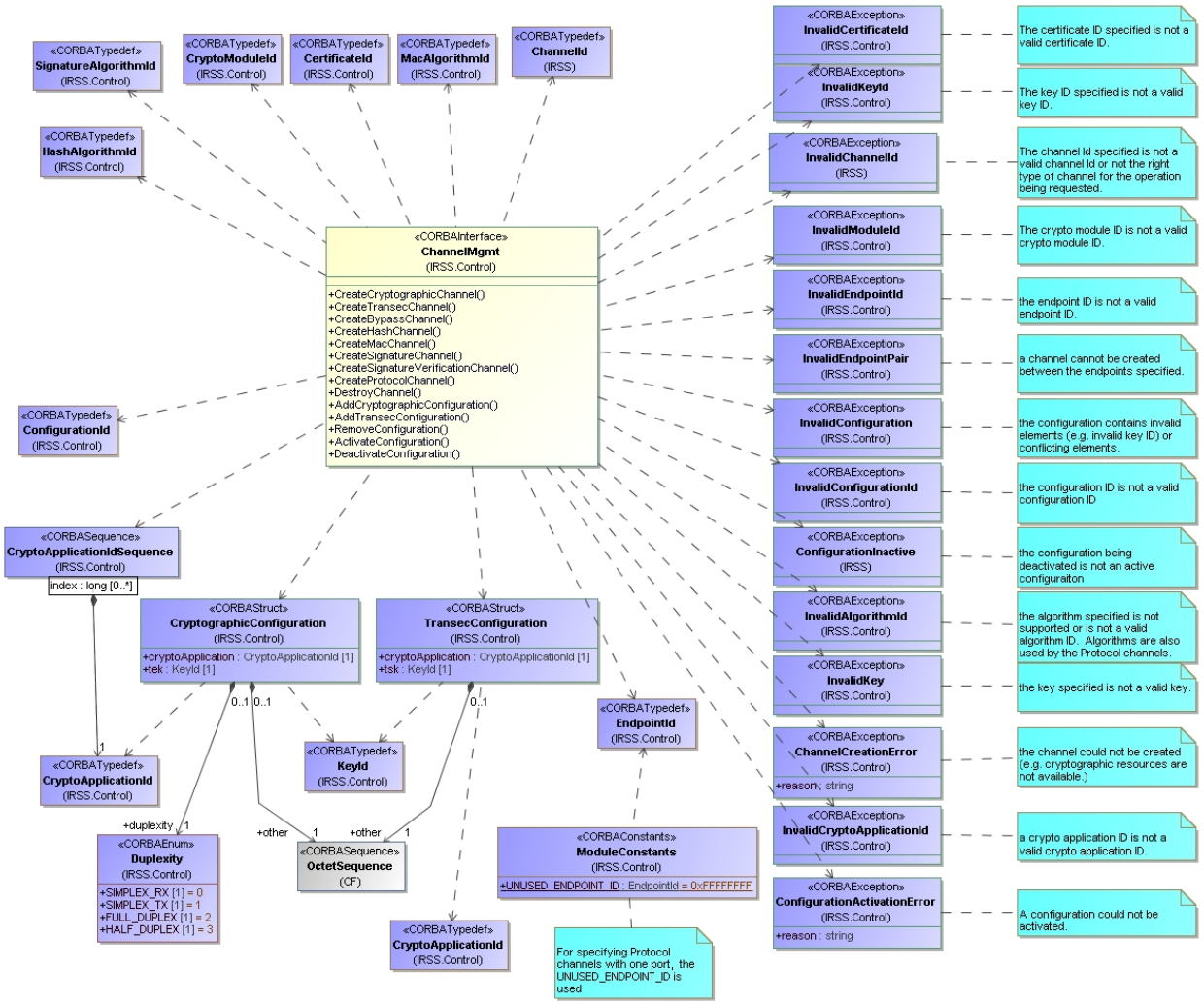

Figure 5 – IRSS::Control::ChannelMgmt[3]

2.3.1.3 IRSS::Control::KeyMgmt Interface Description

Waveform clients use the IRSS::Control::KeyMgmt interface to perform certain key management operations[4]. These operations include updating keys, getting their update counts, and zeroizing keys. The operation UpdateKey() uses a update algorithm implied by the specific

[2] Waveforms can create as many Cryptographic or TRANSEC channels as needed provided the CSS has sufficient cryptographic resources to allocate to each channel.
[3] In Figure 5, function signatures have been elided for brevity. Refer to section 3.4 for details.
[4] Additional key management operations, including the ability to load, store and tag keys are provided by platform interfaces not specified in this standard.

key, while UpdateKeyWithAlgorithm() is used in specific cases where multiple algorithms could be used to update a given key.

Waveforms can zeroize specific keys using the ZeroizeKey() operation.

The IRSS::Control::KeyMgmt interface is shown in Figure 6.

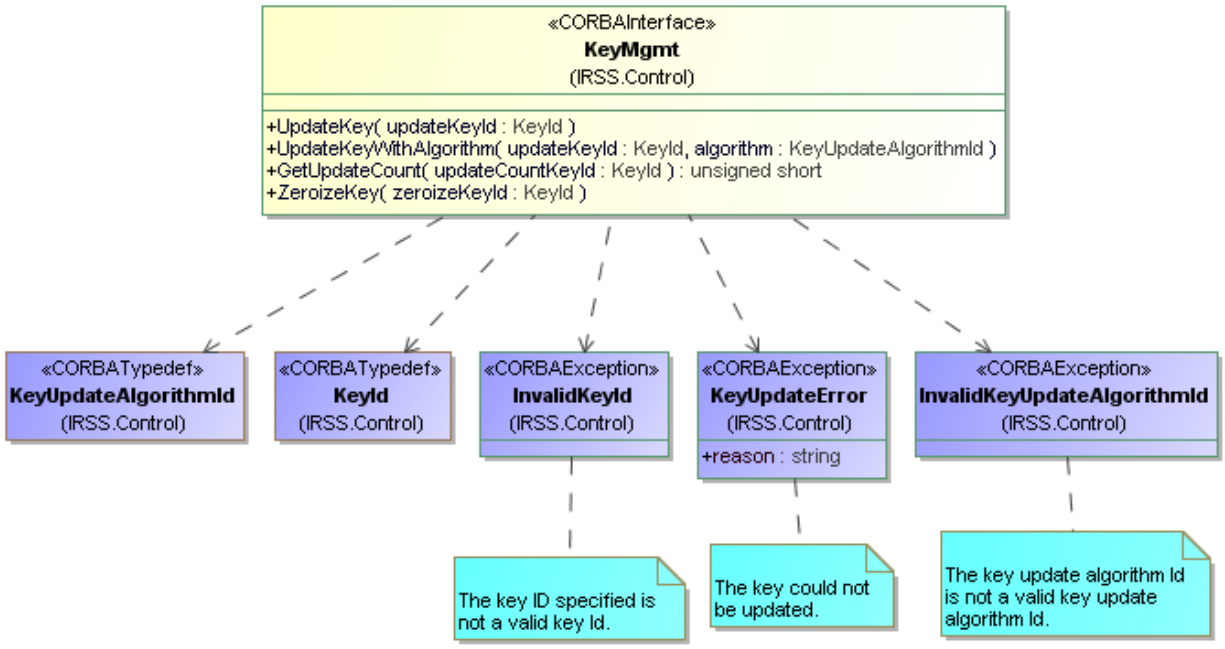

Figure 6 - Control::KeyMgmt Interface

2.3.2 IRSS::Infosec

2.3.2.1 IRSS::Infosec::CryptographicChannel Interface Description

IRSS::Infosec::CryptographicChannel provides an interface which clients use to submit data for encryption or decryption. The data itself consists of two sequences of octets, one containing the information to be transformed and an optional second sequence containing inline bypass information. The interface supports both stream traffic (using the TransformStream() operation) or network packet traffic (using TransformPacket() operation).

For each of these operations, there is a corresponding option to determine the maximum data length that may be submitted in a single call to the TransformStream() or TransformPacket() operations. For stream traffic, GetMaxPacketSize() returns the largest stream packet (i.e. the sum of the payload and bypass octet sequences) in octets that the IRSS can accept in a single TransformStream() call. For packet traffic, TransformPacket() allows multiple packets in a single call. Each individual packet (i.e. the sum of the payload and bypass octet sequences) must be less than or equal to GetMaxPacketSize() octets, while the sum of all packets in a single call shall be less than or equal to GetMaxPayloadSize().

The Transform operations and the SpaceAvailable() operation return a bool indicating if space is available for another transform request. True indicates that space is available for another transform request and false indicates that space is not available (i.e. flow pause). Once flow paused, the client should not push another packet until it receives a flow resume event through the IRSS::Infosec::ControlSignals interface or SpaceAvailable() returns True when queried.

The IRSS::Infosec::CryptographicChannel interface is shown in Figure 7.

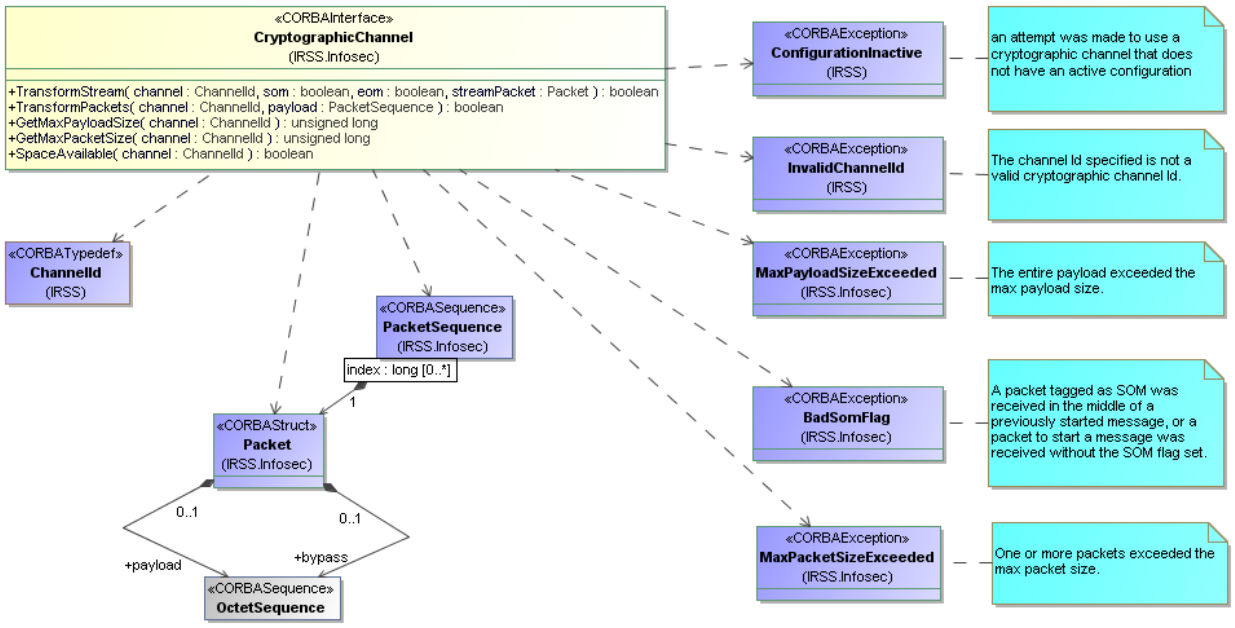

Figure 7 - IRSS::Infosec::CryptographicChannel Interface

2.3.2.2 IRSS::Infosec::CryptographicConsumer Interface Description

IRSS waveform clients implement the IRSS::Infosec::CryptographicConsumer interface to receive data encrypted / decrypted via the TransformStream() or TransformPacket() operations (presumably, but not necessarily in a different security domain). Flow control is not employed in the interface to the client, which is expected to be able to handle the received traffic, including any cryptographic preambles / postambles, etc. Any buffering needed as part of an overall system flow control protocol must be implemented within the client.

The IRSS::Infosec::CryptographicConsumer interface is shown in Figure 8.

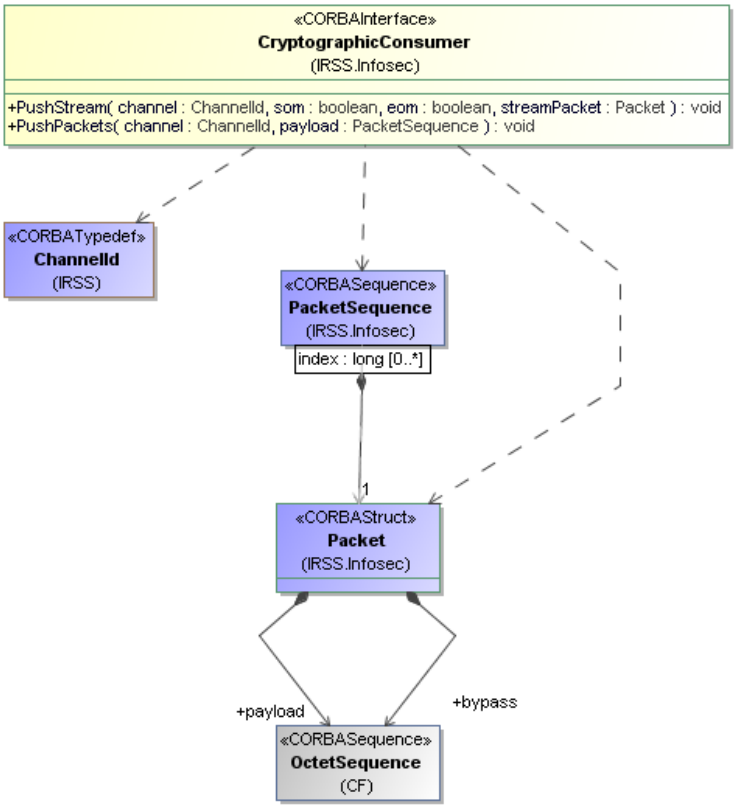

Figure 8 - IRSS::Infosec::CryptographicConsumer Interface

2.3.2.3 IRSS::Infosec::ControlSignals Interface Description

Flow control may be employed in the interface to the IRSS. A client can be flow paused after pushing a packet to the IRSS::Infosec::CryptographicChannel if that packet fills the queues managed by the IRSS. The IRSS::Infosec::ControlSignals interface is the mechanism that the IRSS uses to notify a client that flow can once again resume.

The IRSS::Infosec::ControlSignals interface is shown in Figure 9.

Figure 9 IRSS::Infosec::ControlSignals Interface

2.3.2.4 IRSS::Infosec::TransecChannel Interface Description

TRANSEC channels provide for TRANSEC encryption/decryption as well as keystream generation. The TRANSEC related operations must be seeded before use, via a seed parameter. On the first call to a TransecChannel operation, the seed (whose format is algorithm specific and not specified in this standard) shall be provided. This seed is used to initialize the appropriate

algorithm. On subsequent calls, if the provided seed length is 0, then the algorithm continues without reseeding. If the length is non-zero, than reseeding occurs. Clients pass seeds to the IRSS as CF::OctetSequences. However, a seed is not necessarily an integer multiple of 8 bits. Therefore, the number of seed *bits* must be passed to the IRSS as a separate parameter.

The IRSS::Infosec::TransecChannel interface is shown in Figure 10.

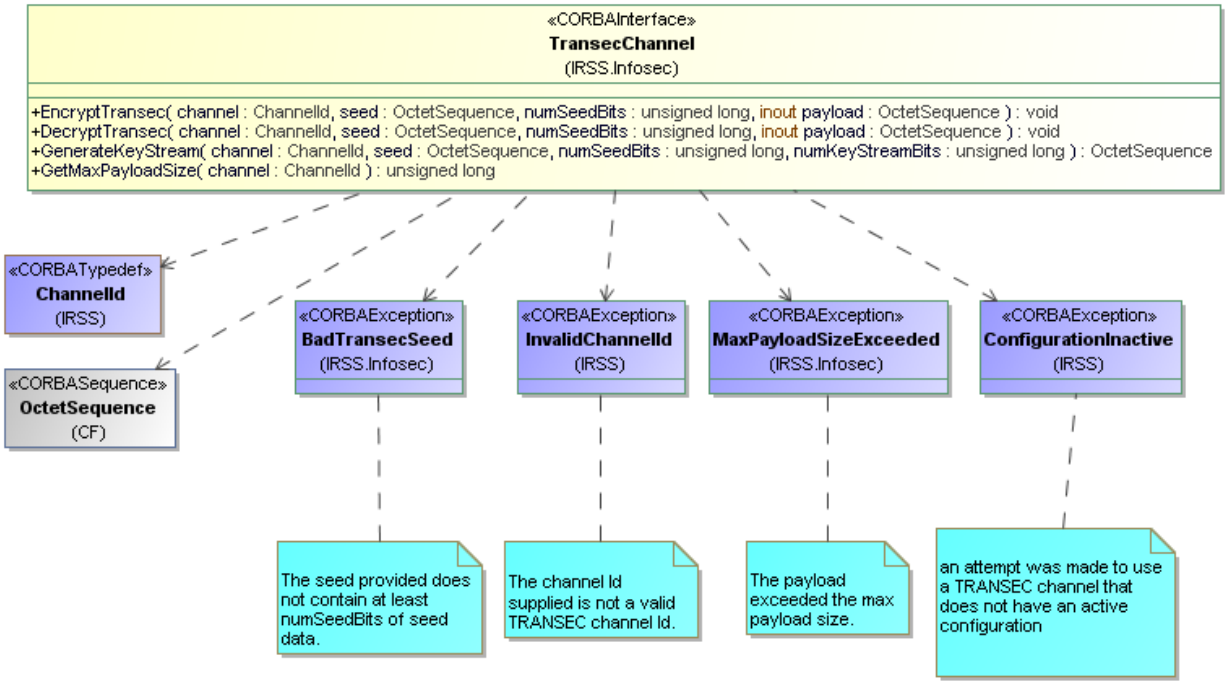

Figure 10 - IRSS::Infosec::TransecChannel Interface

2.3.3 IRSS::Bypass

The IRSS::Bypass::Channel and IRSS::Bypass::Consumer interfaces are shown in Figure 11. The combination of the two are used to move non-traffic data between security domains without encryption or other transformation, with the typical use in a system being to pass inter-component waveform control flows across the CSS divide.

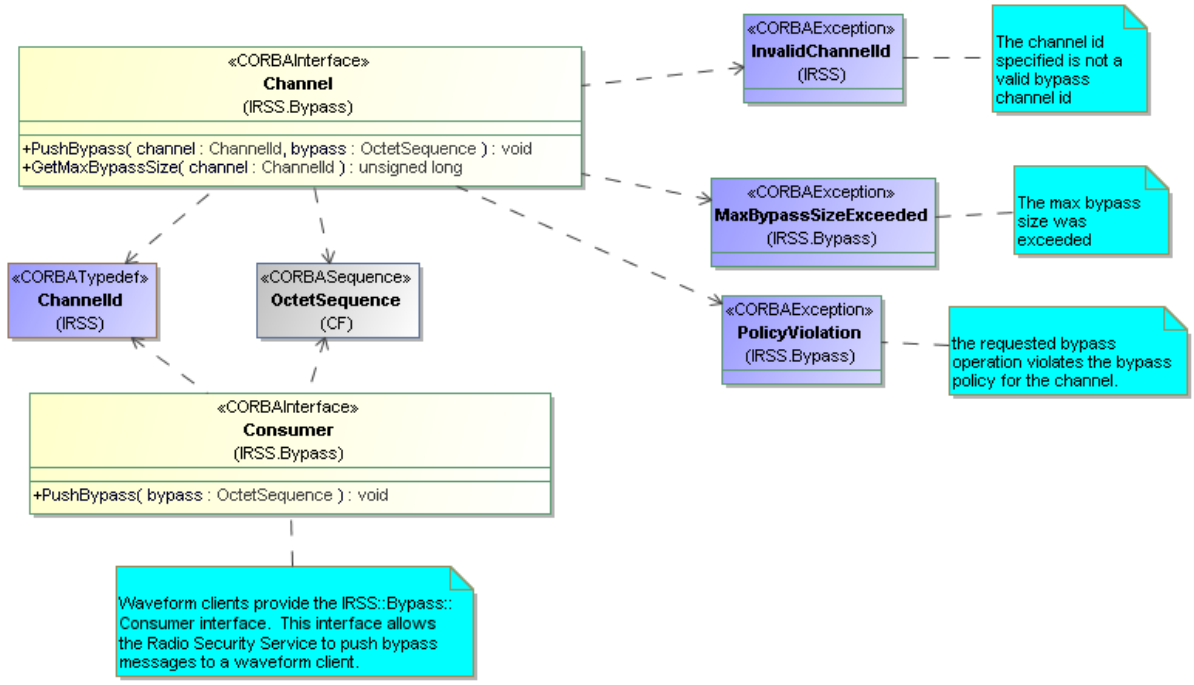

Figure 11 – IRSS::Bypass::Channel and IRSS::Bypass::Consumer Interfaces

2.3.3.1 IRSS::Bypass::Channel Interface Description

The IRSS provides the IRSS::Bypass::Channel interface. Waveforms use the interface to push bypass messages through the crypto module. Bypass traffic is expected to be low rate, and therefore, flow control is not built into the interface. However, there still exists a maximum bypass size allowed for any given bypass message, with an accessor being provided by the API for waveform clients to query the maximum bypass size. This maximum bypass size represents physical system limitations and not bypass policy restrictions (such policies are defined by the platform, and typically enforced by the cryptographic subsystem, but are not accessible by the standardized waveform APIs).

2.3.3.2 IRSS::Bypass::Consumer Interface Description

The IRSS::Bypass::Consumer interface is used by a waveform to receive bypass flows from the IRSS that were originated from a IRSS::Bypass::Channel interface in a different security domain. There are no inherent flow-control provisions supported by this interface – it is assumed that the stream is consumed by the waveform. This does not preclude the waveform from employing other mechanisms outside the range of this specification (e.g. waveform internal flows, etc).

2.3.4 IRSS::IandA

There are several different IRSS::IandA (Integrity and Authentication) channel types that are used to carry out common I&A functions. Except for random numbers (see below), the algorithms used to perform these functions are not standardized in this specification, but rather are provided by the IRSS implementation. When a waveform creates the IRSS::IandA channel

(see section 2.3.1.2), it specifies the desired algorithm. The IRSS::IandA interface UML is shown in Figure 12, with descriptions of the individual interfaces in the following subsections.

In addition, the IRSS::IandA module contains an interface related to the generation of random numbers. As the algorithm is standardized here (pseudorandom) or not applicable (true random), a channel concept is not used. This interface is described below in section 2.3.4.6.

2.3.4.1 IRSS::IandA::Channel Interface Description

Waveforms use the IRSS::IandA channels to perform a variety of I&A functions. In most cases, use of such channels require supplying the IRSS with a quantity of data (typically using multiple calls, as the data packet size is limited), and then when complete, asking for information back which constitutes, the hash, signature or MAC.

The IRSS::IandA::Channel interface is an abstract base interface that allows clients to push data to the IRSS. Data is pushed in chunks not to exceed the maximum data size as defined by GetMaxDataSize(). Actual concrete interfaces then specialize this interface with specific operations for retrieval of outputs. These are detailed in subsequent subsections.

Once one of the specialized concrete channels have been created, a client uses GetMaxDataSize() to find the maximum amount of data that can be passed in a call. They then can push multiple packets into a channel using PushData() When done, the specialized operations (see following sections) can be used to retrieve the results. Once done, a channel can be cleared and prepared for reuse using the Reset() operation. In this way, waveforms do not need to destroy and recreate the channel when multiple functions need to be accomplished.

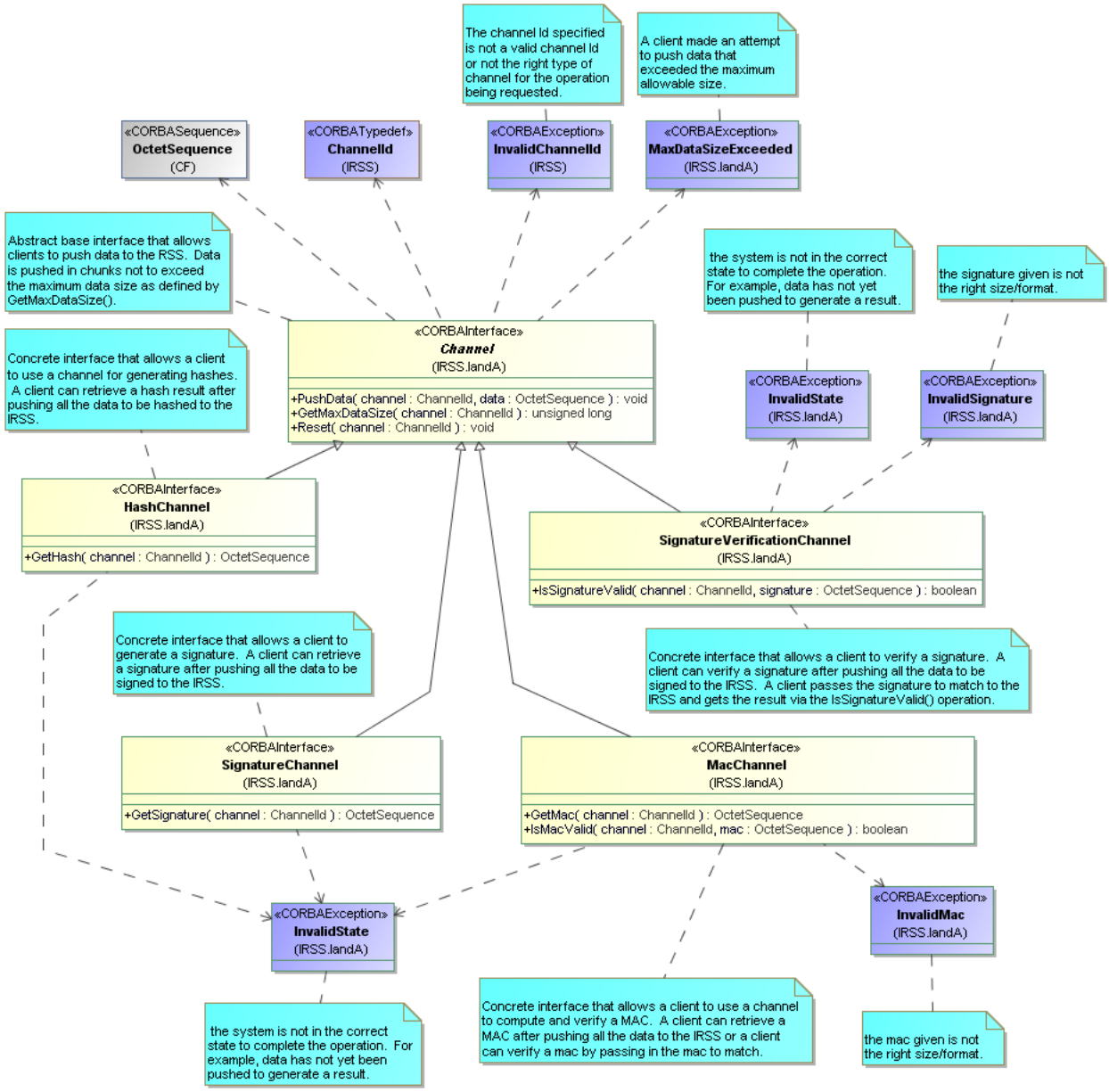

Figure 12 - IRSS::IandA::Channel Interfaces

2.3.4.2 IRSS::IandA::HashChannel Interface Description

IRSS::IandA::HashChannel is an interface that allows a client to use an IRSS::IandA::Channel for generating hashes. The hash is performed using the algorithm specified at channel creation. A client can retrieve a hash result after pushing all the data to be hashed to the IRSS.

2.3.4.3 IRSS::IandA::MacChannel Interface Description

IRSS::IandA::MacChannel is an interface that allows a client to use an IRSS::IandA::Channel to compute and verify a MAC. The MAC is performed using the algorithm specified at channel

creation. A client can retrieve a MAC after pushing all the data to the IRSS or a client can verify a MAC by passing in the MAC to match.

2.3.4.4 IRSS::IandA::SignatureChannel Interface Description

IRSS::IandA::SignatureChannel is an interface that allows a client to generate a signature. The signature is performed using the algorithm specified at channel creation. A client can retrieve a signature after pushing all the data to be signed to the IRSS.

2.3.4.5 IRSS::IandA::SignatureVerificationChannel Interface Description

IRSS::IandA::SignatureVerificationChannel is an interface that allows a client to verify a signature. A client can verify a signature after pushing all the data to be signed to the IRSS. A client passes the signature to match to the IRSS and gets the result via the IsSignatureValid() operation. The signature is performed using the algorithm specified at channel creation.

2.3.4.6 IRSS::IandA::Random Interface Description

IRSS::IandA::Random is an interface that can be used to generate true random numbers (via GetRandom()) or pseudo random numbers using a seed (via GetPseudoRandom()).

The IRSS::IandA::Random interface is shown in Figure 13.

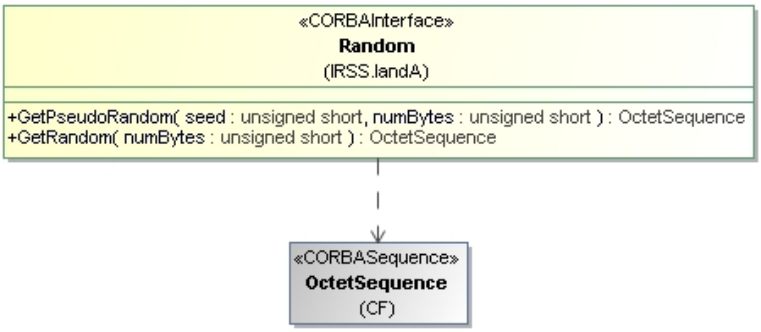

Figure 13 - IRSS::IandA::Random Interface

2.3.5 IRSS::Protocol

2.3.5.1 IRSS::Protocol::Channel Interface Description

Protocol channels, while typically used to exchange a series of algorithm-specific protocol messages to the IRSS[5], can be used in other ways as well – essentially providing a generic exchange between the waveform and the IRSS, which in turn is interpreted by the associated Cryptographic Application. Messages have a maximum size as defined by the protocol definition.

[5] An example would be when negotiating an asymmetric key for IPSEC, etc – where the IRSS is used to perform transformations in generating IPSEC messages.

The IRSS::Protocol::Channel interface is shown in Figure 14.

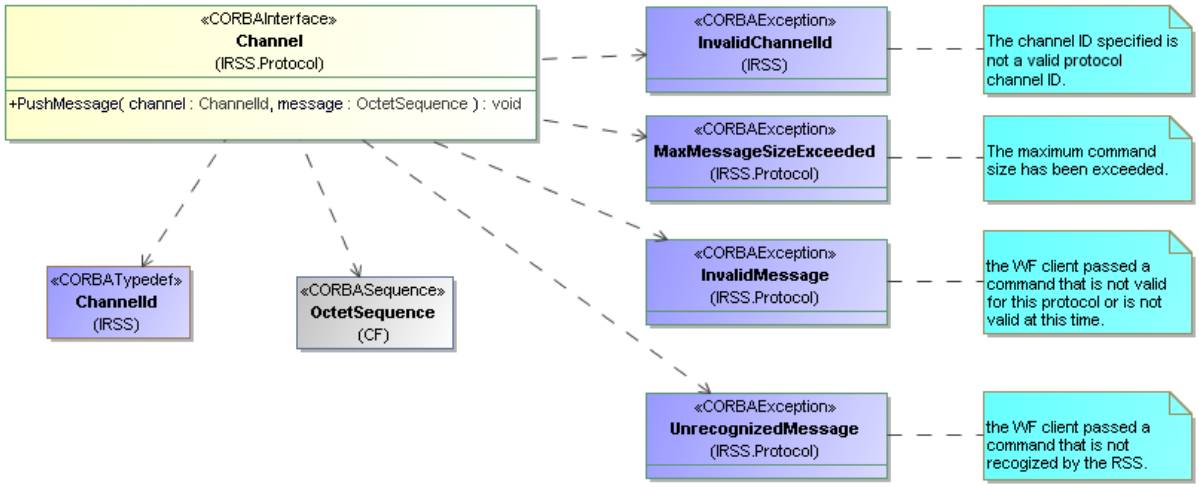

Figure 14 - IRSS::Protocol::Channel Interface

2.3.5.2 IRSS::Protocol::Consumer Interface Description

Waveform clients provide the IRSS::Protocol::Consumer interface. The IRSS uses this interface to push protocol messages to the client.

The IRSS::Protocol::Consumer interface is shown in Figure 15.

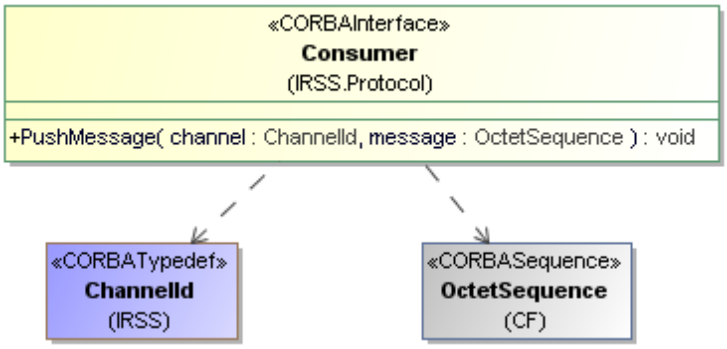

Figure 15 - IRSS::Protocol::Consumer Interface

2.4 Sequence Diagrams

2.4.1 Two Security Domain Cryptographic Channel

Description

This sequence diagram shows how to create and use a single cryptographic channel for encryption and decryption of packets in a two security domain implementation. The sequence includes the use of the flow control aspects of the API (see steps 11 – 14).

Note that the IRSS is shown as a single entity for simplicity. In a two security domain solution there would be an IRSS instance on the PT side and another on the CT side of the system.

Pre-conditions

The CSS has resources available to allow the creation of the cryptographic channel.

Post-conditions

The cryptographic channel is active and ready to process more data.

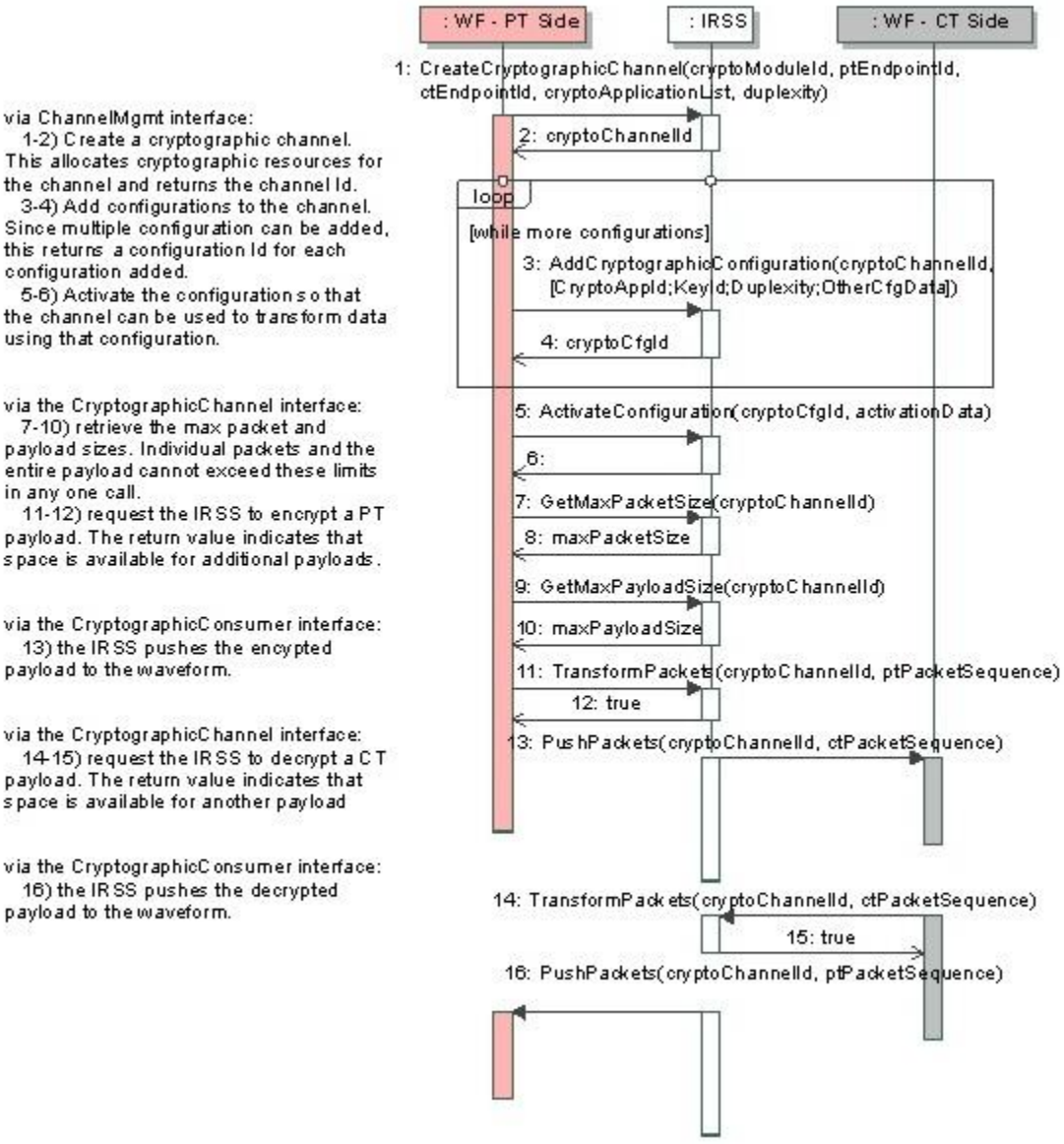

Figure 16 - Two Security Domain Cryptographic Channel Sequence Diagram

2.4.2 Single Security Domain Cryptographic Channel

Description

This sequence diagram shows how to create and use a single cryptographic channel for encryption and decryption of packets in a single security domain implementation. The IRSS will need to provide both PT and CT ports implementing the CryptographicChannel interface. The WF will need to provide both PT and CT ports implementing the CryptographicConsumer interface.

Pre-conditions

The CSS has resources available to allow the creation of the cryptographic channel.

Post-conditions

The cryptographic channel is active and ready to process more data.

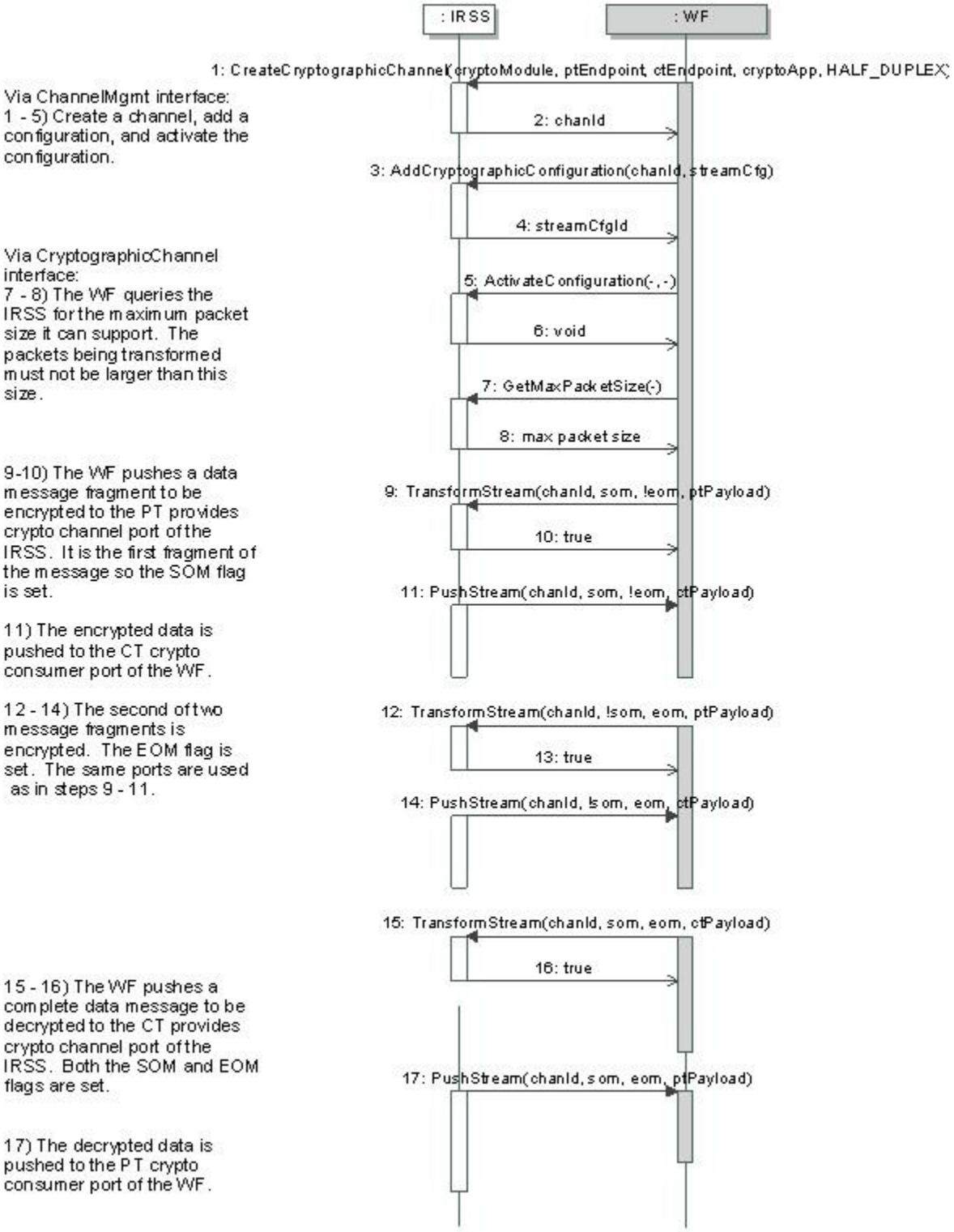

Figure 17 - Single Security Domain Cryptographic Channel Sequence Diagram

2.4.3 Stream Multi Channels

Description

This sequence diagram shows a waveform that needs to process two simultaneous incoming streams. Each stream may have its own algorithm and/or key. The waveform creates and configures a channel with a single configuration based on learning details of each incoming stream. The cryptographic state is kept with each cryptographic channel. This allows the two streams to be alternately processed through the crypto, each keeping its own overall message state.

Note that the IRSS is shown as a single entity for simplicity. In a two security domain solution there would be an IRSS instance on the PT side and another on the CT side of the system.

Pre-conditions

The CSS has resources available to allow the creation of the two cryptographic channels.

Post-conditions

The cryptographic channels have been destroyed, their state cleared, and their resources are available for use.

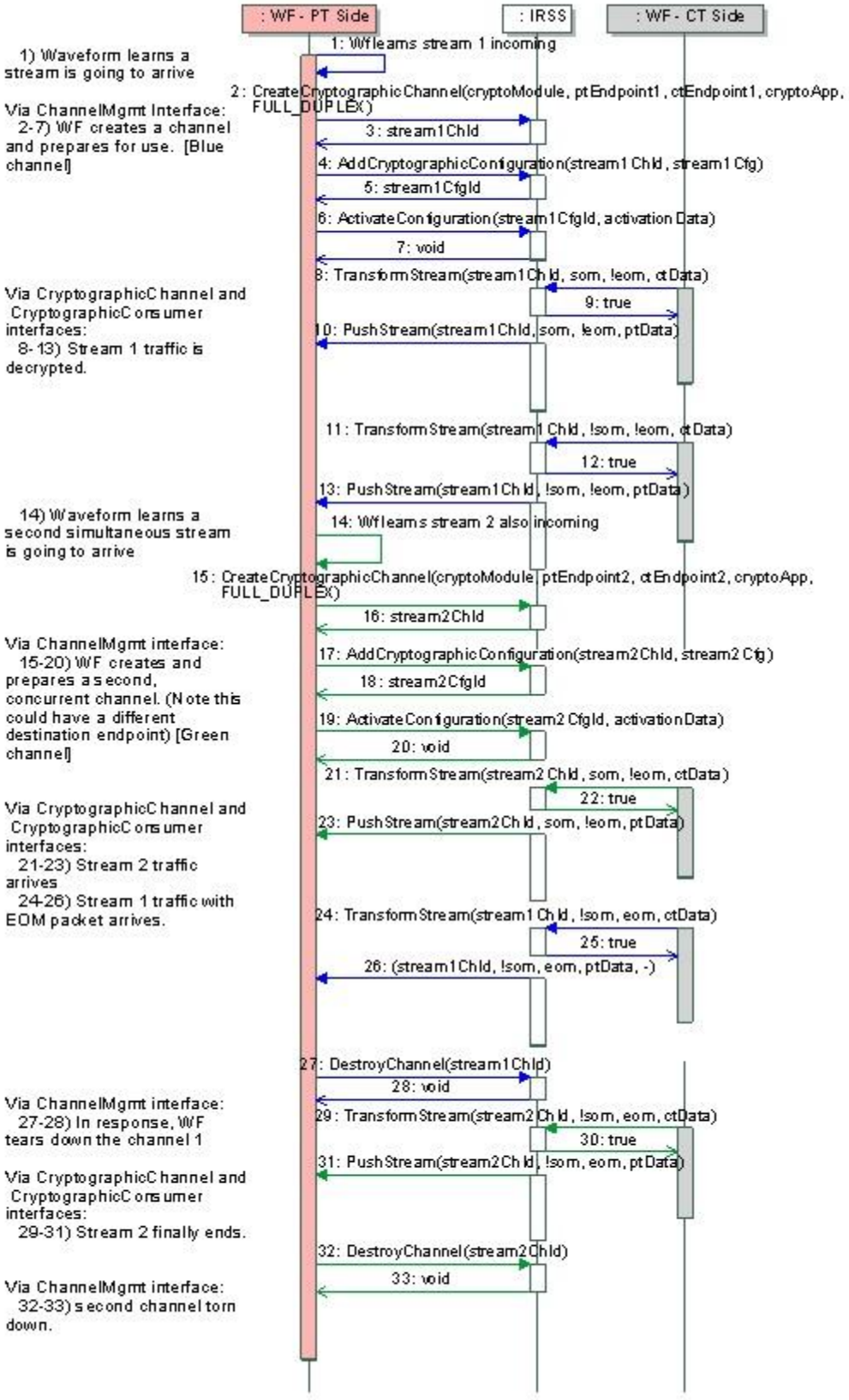

Figure 18 - Stream Multi Channels Sequence Diagram

2.4.4 TRANSEC - Encrypt/Decrypt

Description

This sequence diagram shows how to create and use a single TRANSEC channel for encryption and decryption to cover and uncover a data stream. The algorithm gets reinitialized whenever a new seed is passed in. The payload parameter of the encrypt and decrypt operations is an inout parameter.

Note that the IRSS is shown as a single entity for simplicity. In a two security domain solution there would be an IRSS instance on the PT side and another on the CT side of the system.

Pre-conditions

The CSS has resources available to allow the creation of the TRANSEC channel. The bypass channel has already been created and is available for the waveform to use.

Post-conditions

The TRANSEC channel is still active and able to encrypt and decrypt.

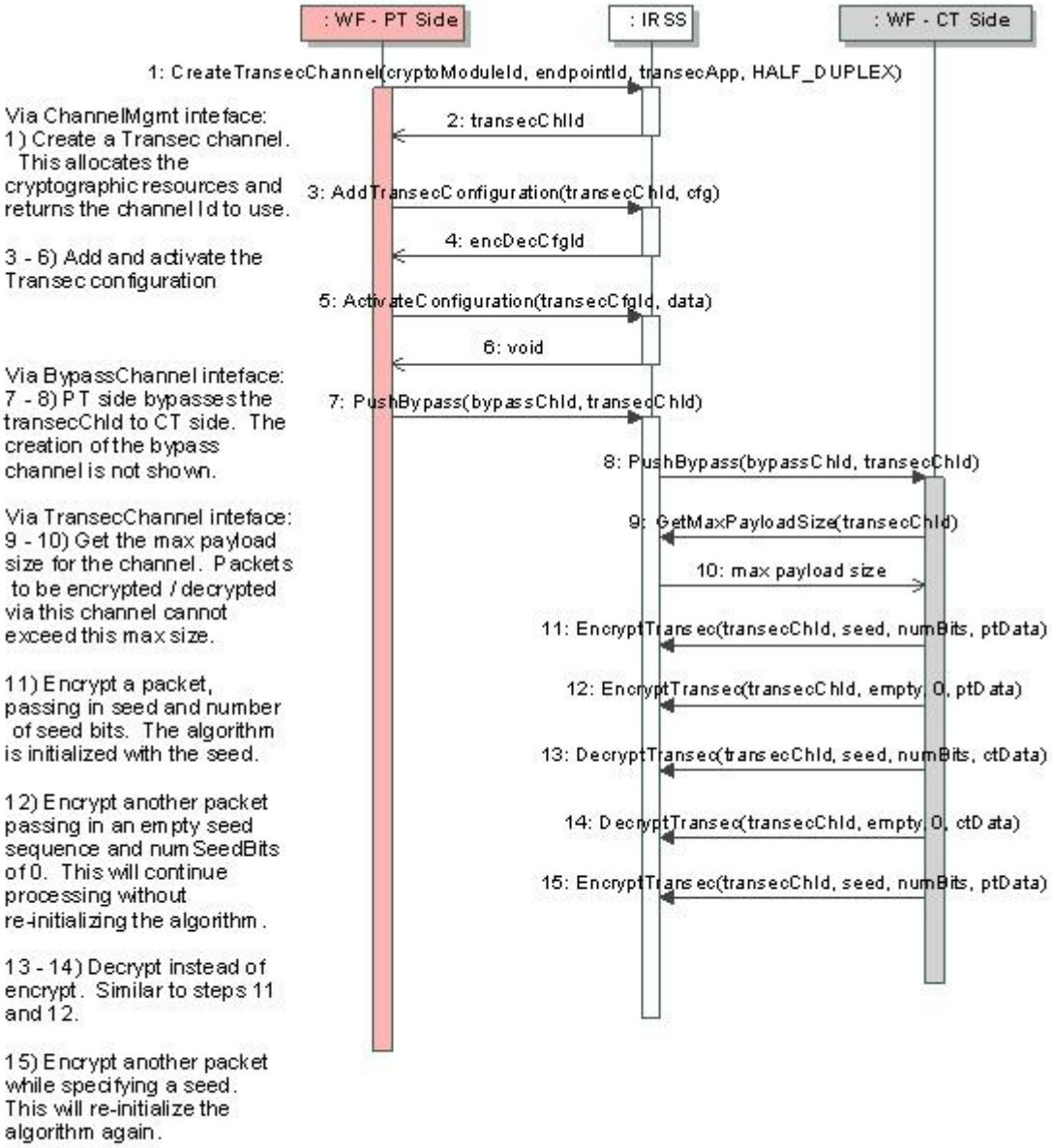

Figure 19 - TRANSEC - Encrypt/Decrypt Sequence Diagram

2.4.5 TRANSEC – Keystream

Description

This sequence diagram shows how to create and use a single TRANSEC channel for generation of key stream. The key stream generation algorithm gets reinitialized whenever a new seed is passed in. Both the size of the seed and the requested size of key stream to be generated are specified in number of bits. The seed itself and the returned key stream data are

CF::OctetSequences. The number of bytes in the returned key stream may be padded out to be a multiple of the algorithm's block size.

Note that the IRSS is shown as a single entity for simplicity. In a two security domain solution there would be an IRSS instance on the PT side and another on the CT side of the system.

Pre-conditions

The CSS has resources available to allow the creation of the TRANSEC channel. The bypass channel has already been created and is available for the waveform to use.

Post-conditions

The TRANSEC channel is still active and able generate keystream.

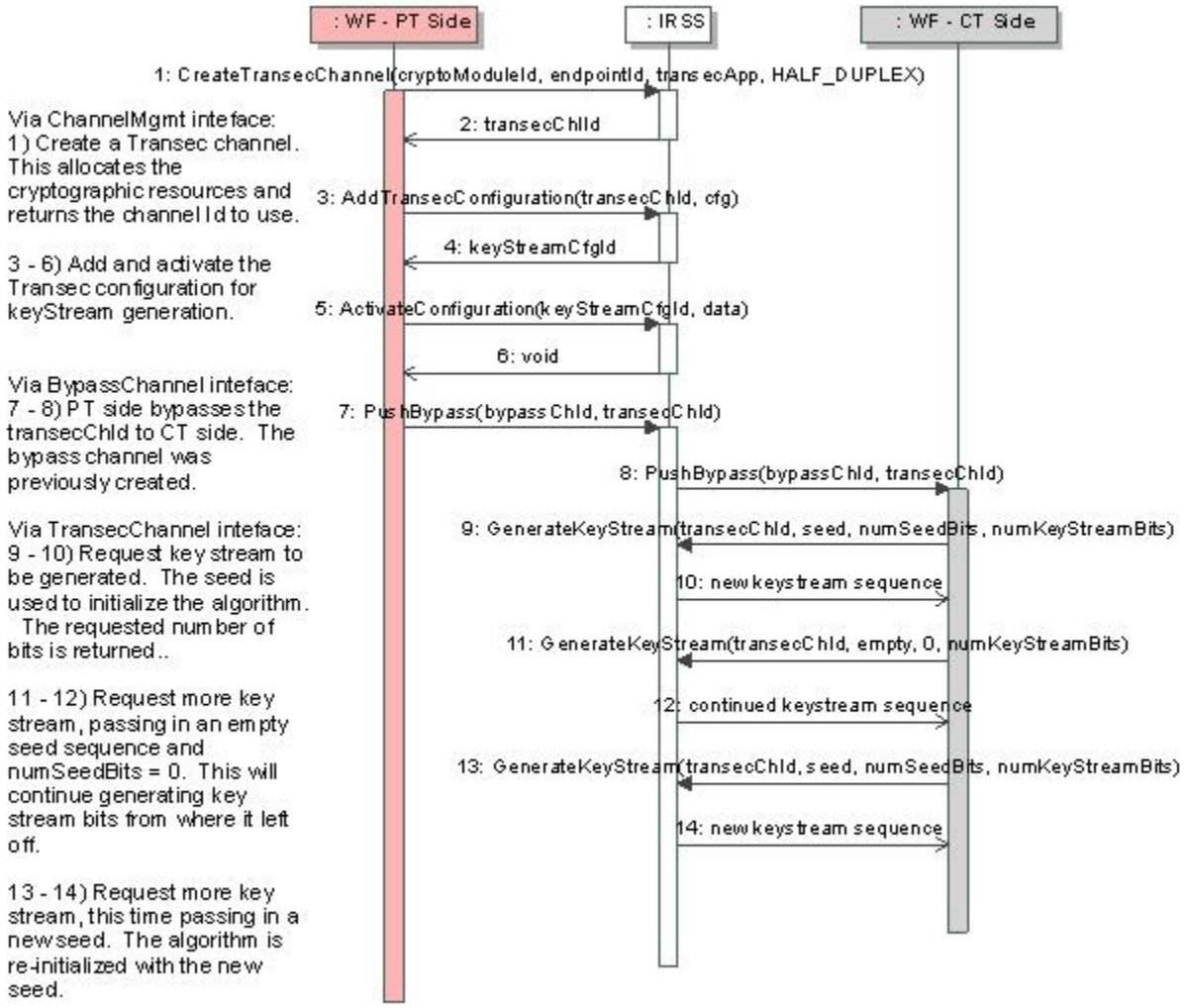

Figure 20 - TRANSEC - Keystream Sequence Diagram

2.4.6 Bypass Channels

Description

This example shows the full bypass channel lifecycle. A single bypass channel provides for bypassing of information sourced from security domain A and sunk to security domain B. To support two way bypass traffic between two security domains requires a pair of bypass channels. For this example PtToCt and CtToPt bypass channels are created and used. Each bypass message has to be smaller than the maximum size allowed on the platform. A client determines this value by calling GetMaxBypassSize(). The bypass policy being enforced by the cryptographic subsystem may impose further constraints on the bypass traffic.

Pre-conditions

The CSS has resources available to allow the creation of the bypass channel.

Post-conditions

The bypass channels have been destroyed and the resources have been released.

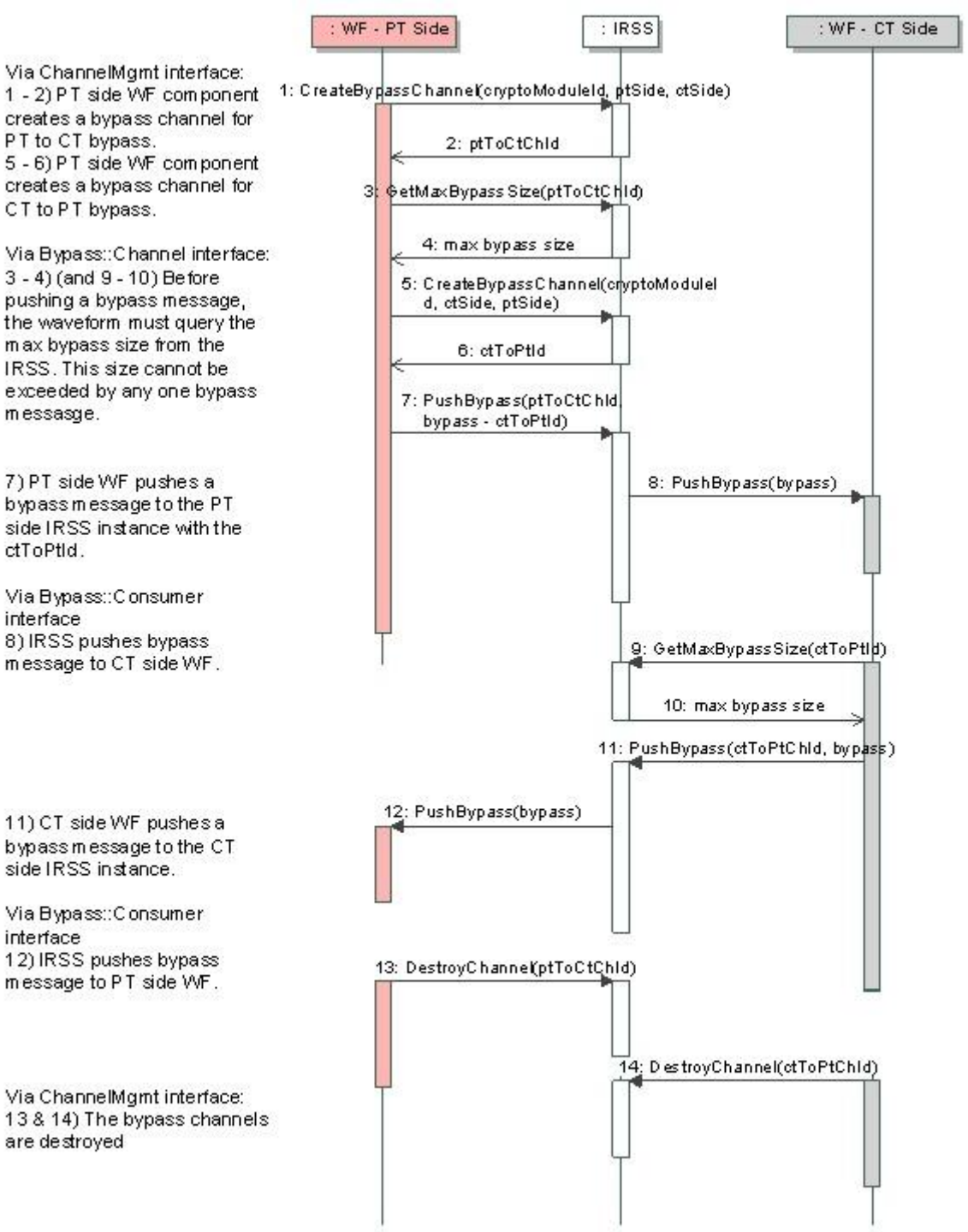

Figure 21 - Bypass Channels Sequence Diagram

2.4.7 Hash Channels

Description

At channel creation time, the hash algorithm (e.g. MD5, SHA-256, etc.) is selected along with the crypto module and endpoint. The waveform client needs to query for the maximum data size that can be handled by the channel since this value will be platform specific. The client then breaks up the data to be hashed into multiple chunks smaller than the maximum data size and pushes the data to the hash channel. The GetHash() method returns the hash of the data processed since the channel was created or last reset.

Pre-conditions

The CSS has resources available to allow the creation of a Hash Channel.

Post-conditions

The Hash Channel is active and ready to process more data.

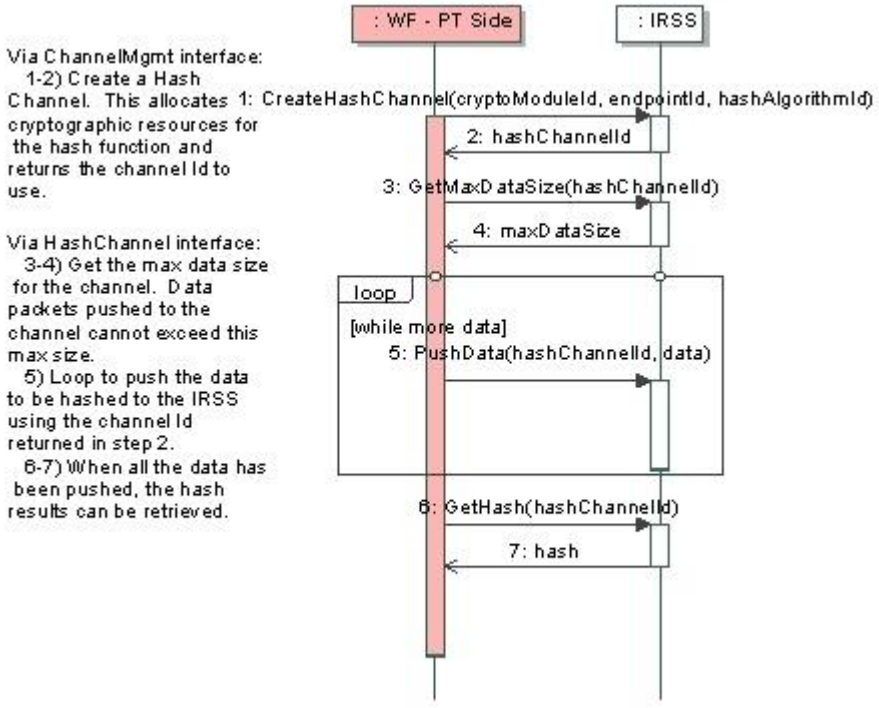

Figure 22 - HashChannel Sequence Diagram

2.4.8 Protocol

Description

This sequence diagram shows a possible example of Protocol channel usage. This example is a subset of required operations for generating a session key using IKE. After the channel is

created, protocol messages are passed between the WF component and IRSS via the PushMessage() method. This example is not a normative description of how an IKE protocol channel would work.

Pre-conditions

The CSS has resources available to allow the creation of the Protocol Channel.

Post-conditions

The Protocol Channel is still active and able to process messages.

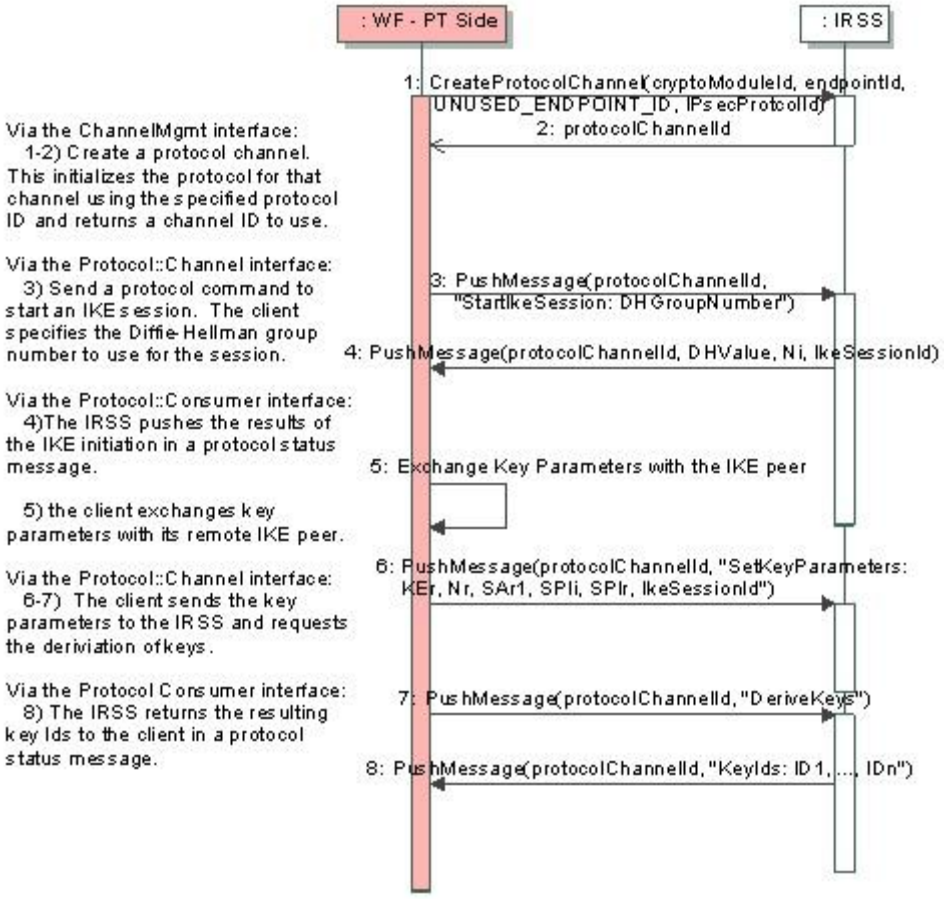

Figure 23 - Protocol Sequence Diagram

3 Service Primitives and Attributes

3.1 IRSS::Bypass::Channel

3.1.1 PushBypass Operation

This operation pushes bypass messages through the crypto module.

The maximum bypass message size allowed can be retrieved from the GetMaxBypassSize() operation.

Note: Bypass traffic is expected to be at a low rate, and therefore, flow control is not built into the interface.

3.1.1.1 Synopsis

void PushBypass(in IRSS::ChannelId channel, in CF::OctetSequence bypass)
raises(IRSS::InvalidChannelId, MaxBypassSizeExceeded, PolicyViolation);

3.1.1.2 Parameters

Parameter Name	**Type**	**Description**
channel	IRSS::ChannelId	Identifies the bypass channel
bypass	CF::OctetSequence	The bypass message to push

3.1.1.3 Return Value

None

3.1.1.4 Originator

Waveform clients

3.1.1.5 Exceptions

Exception	**Description**
IRSS::InvalidChannelId	The channel identifier specified is not a valid bypass channel identifier
MaxBypassSizeExceeded	The maximum bypass size was exceeded
PolicyViolation	The requested bypass operation violates the bypass policy for the channel

3.1.2 GetMaxBypassSize Operation

This operation allows waveform clients to retrieve a channel's maximum bypass size. This maximum bypass size represents physical system limitations and not bypass policy restrictions (as enforced by the cryptographic subsystem), which will likely be less than the physical system limitations.

3.1.2.1 Synopsis

unsigned long GetMaxBypassSize(in IRSS::ChannelId channel)
raises(IRSS::InvalidChannelId);

3.1.2.2 Parameters

Parameter Name	**Type**	**Description**
channel	IRSS::ChannelId	Identifies the bypass channel

3.1.2.3 Return Value

Type	**Description**	**Valid Range**
unsigned long	Maximum bypass message size in octets.	Channel dependent

3.1.2.4 Originator

Waveform clients

3.1.2.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid bypass channel identifier

3.2 IRSS::Bypass::Consumer

Waveform clients provide the IRSS::Bypass::Consumer interface

3.2.1 PushBypass Operation

This operation forwards a bypassed message back to a waveform client.

Note: Bypass traffic is expected to be at a low rate, and therefore, flow control is not built into the interface. A maximum message size allowed exists for any given bypass message.

3.2.1.1 Synopsis

void PushBypass(in CF::OctetSequence bypass);

3.2.1.2 Parameters

Parameter Name	**Type**	**Description**
bypass	CF::OctetSequence	The message that was bypassed.

3.2.1.3 Return Value

None

3.2.1.4 Originator

Radio Security Service

3.2.1.5 Exceptions

None

3.3 IRSS::Control::CertificateMgmt

Client interface provided by the IRSS for managing certificates by waveform clients.

3.3.1 RetrieveCertificate Operation

This operation returns the public portion of the requested certificate. It does not include the private key.

3.3.1.1 Synopsis

CF::OctetSequence RetrieveCertificate(in CertificateId certId) raises(InvalidCertificateId);

3.3.1.2 Parameters

Parameter Name	Type	Description
certId	CertificateId	The ID of the certificate being requested

3.3.1.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The certificate data.	Certificate data is returned in X.509v3 format as specified in RFC 3280.

3.3.1.4 Originator

Waveform clients

3.3.1.5 Exceptions

Exception	Description
InvalidCertificateId	The certificate ID is not a valid certificate ID

3.3.2 GetCertficateIds Operation

This operation retrieves all IDs for the certificates that have been loaded into, and are managed by, the IRSS.

3.3.2.1 Synopsis

CertificateIdSequence GetCertificateIds( );

3.3.2.2 Parameters

None

3.3.2.3 Return Value

Type	Description	Valid Range
CertificateIdSequence	The IDs of all valid certificates currently being managed	Platform dependent

3.3.2.4 Originator

Waveform clients

3.3.2.5 Exceptions

None

3.3.3 IsCertficateValid Operation

This operation checks if the certificate passed in is a valid certifcate. Possible reasons for a certificate being invalid include: a certificate does not trace to a known trust anchor, it is expired, the certificate has been revoked, etc.

3.3.3.1 Synopsis

boolean IsCertificateValid(in CF::OctetSequence certificate) raises(UnrecognizedCertificate);

3.3.3.2 Parameters

Parameter Name	Type	Description
certificate	CF::OctetSequence	The certificate data in X.509v3 format as specified in RFC 3280.

3.3.3.3 Return Value

Type	Description	Valid Range
boolean	Indicates whether the certificate is valid.	TRUE=The passed in certificate is valid FALSE=The passed in certificate is not valid

3.3.3.4 Originator

Waveform clients

3.3.3.5 Exceptions

Exception	Description
UnrecognizedCertificate	The passed in certificate data could not be recognized as a certificate

3.4 IRSS::Control::ChannelMgmt

Client interface provided by the IRSS for creating and managing channels.

3.4.1 CreateCryptographicChannel Operation

This operation creates a cryptographic channel for the purpose of encrypting and decrypting user data. Cryptographic resources are allocated at channel creation time. Before a cryptographic channel can be used it must be configured and activated. Cryptographic channels are configured by calling AddCryptographicConfiguration() (see 3.4.10) to add a configuration to the channel and then activated by calling ActivateConfiguration() (see 3.4.13) to activate it.

The channel duplexity is specified at creation time to allow the CSS to allocate resources for the channel. However, the actual duplexity used can vary with the active configuration. To ensure proper resource allocation, a waveform should specify the needed duplexity requiring the most CSS resources at channel creation time. For these purposes, duplexity can be ordered in increasing CSS resource requirements as follows:

- SIMPLEX_RX, SIMPLEX_TX (low)
- HALF_DUPLEX
- FULL_DUPLEX (high)

3.4.1.1 Synopsis

IRSS::ChannelId CreateCryptographicChannel(in CryptoModuleId cm, in EndpointId ptEndpoint, in EndpointId ctEndpoint, in CryptoApplicationIdSequence cryptoApps, in Duplexity channelDuplexity) raises(InvalidModuleId, InvalidEndpointId, InvalidEndpointPair, InvalidCryptoApplicationId, ChannelCreationError);

3.4.1.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
ptEndpoint	EndpointId	The number identifying the PT side crypto module access point
ctEndpoint	EndpointId	The number identifying the CT side crypto module access point
cryptoApps	CryptoApplicationIdSequence	The list of cryptographic application IDs that will be used on this channel
channelDuplexity	Duplexity	The duplexity usage requiring the most CSS resources.

3.4.1.3 Return Value

Type	**Description**	**Valid Range**
IRSS::ChannelId	The identifier of the cryptographic channel created	Platform dependent

3.4.1.4 Originator

Waveform clients

3.4.1.5 Exceptions

Exception	**Description**
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID
InvalidEndpointPair	A channel cannot be created between the endpoints specified
ChannelCreationError	The channel could not be created. This could be due to insufficient resources being available, an invalid combination of application IDs within the cryptoApps, or other reasons.
InvalidCryptoApplicationId	A crypto application ID is not a valid crypto application ID

3.4.2 CreateTransecChannel Operation

This operation creates a TRANSEC channel for the purpose of encrypting data for transmission. Cryptographic resources are allocated at channel creation time.

The channel duplexity is specified at creation time to allow the CSS to allocate resources for the channel. However, the actual duplexity used can vary with the active configuration. To ensure proper resource allocation, a waveform should specify the needed duplexity requiring the most CSS resources at channel creation time. For these purposes, duplexity can be ordered in increasing CSS resource requirements as follows:

- SIMPLEX_RX, SIMPLEX_TX (low)
- HALF_DUPLEX
- FULL_DUPLEX (high)

3.4.2.1 Synopsis

IRSS::ChannelId CreateTransecChannel(in CryptoModuleId cm, in EndpointId endpoint, in CryptoApplicationIdSequence cryptoApps, in Duplexity channelDuplexity) raises(InvalidModuleId, InvalidCryptoApplicationId, ChannelCreationError, InvalidEndpointId);

3.4.2.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
endpoint	EndpointId	The number identifying the crypto module access point
cryptoApps	CryptoApplicationIdSequence	The list of cryptographic application IDs that will be used on this channel
channelDuplexity	Duplexity	The duplexity usage requiring the most CSS resources.

3.4.2.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the TRANSEC channel created	Platform dependent

3.4.2.4 Originator

Waveform clients

3.4.2.5 Exceptions

Exception	Description
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID
ChannelCreationError	The channel could not be created. This could be due to insufficient resources being available, an invalid combination of application IDs within the cryptoApps, or other reasons.
InvalidCryptoApplicationId	A crypto application ID is not a valid crypto application ID

3.4.3 CreateBypassChannel Operation

This operation creates a bypass channel. Bypass channels are used to move control information through the cryptographic subsystem.

3.4.3.1 Synopsis

IRSS::ChannelId CreateBypassChannel(in CryptoModuleId cm, in EndpointId sourceEndpoint, in EndpointId destinationEndpoint) raises(ChannelCreationError, InvalidModuleId, InvalidEndpointId, InvalidEndpointPair);

3.4.3.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
sourceEndpoint	EndpointId	The number identifying the bypass channel's source crypto module access point
destinationEndpoint	EndpointId	The number identifying the bypass channel's destination crypto module access point

3.4.3.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the bypass channel created.	Platform dependent

3.4.3.4 Originator

Waveform clients

3.4.3.5 Exceptions

Exception	Description
InvalidEndpointId	The endpoint ID is not a valid endpoint ID
InvalidEndpointPair	A channel cannot be created between the endpoints specified
ChannelCreationError	The channel could not be created
InvalidModuleId	The crypto module ID is not a valid crypto module ID

3.4.4 CreateHashChannel Operation

This operation creates a hash channel. Hash channels are used to generate a hash on data that has already been pushed into the channel.

3.4.4.1 Synopsis

IRSS::ChannelId CreateHashChannel(in CryptoModuleId cm, in EndpointId inputEndpoint, in HashAlgorithmId hashAlogrithm) raises(ChannelCreationError, InvalidModuleId, InvalidEndpointId, InvalidAlgorithmId);

3.4.4.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module to create the channel in

inputEndpoint	EndpointId	The number identifying the hash channel's source crypto module access point
hashAlgorithm	HashAlgorithmId	The identifier of the Hash algorithm to use.

3.4.4.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the hash channel created.	Platform dependent

3.4.4.4 Originator

Waveform clients

3.4.4.5 Exceptions

Exception	Description
InvalidEndpointId	The endpoint ID is not a valid endpoint ID
InvalidAlgorithmId	The algorithm specified is not a supported hash algorithm or is not a valid algorithm ID
ChannelCreationError	The channel could not be created
InvalidModuleId	The crypto module ID is not a valid crypto module ID

3.4.5 CreateMacChannel Operation

This operation creates a MAC channel. MAC channels are used to generate a MAC for the data which has already been passed in.

3.4.5.1 Synopsis

IRSS::ChannelId CreateMacChannel(in CryptoModuleId cm, in EndpointId inputEndpoint, in MacAlgorithmId macAlogrithmId, in KeyId macKeyId) raises(InvalidKeyId, ChannelCreationError, InvalidAlgorithmId, InvalidModuleId, InvalidEndpointId);

3.4.5.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
inputEndpoint	EndpointId	The number identifying the MAC channel's input crypto module access point
macAlgorithmId	MacAlgorithmId	The identifier of the MAC algorithm to use.
macKeyId	KeyId	The identifier of the Key to use.

3.4.5.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the MAC channel created.	Platform dependent

3.4.5.4 Originator

Waveform clients

3.4.5.5 Exceptions

Exception	Description
InvalidKey	The key ID specified is not a valid key ID or does not specify a MAC key.
ChannelCreationError	The channel could not be created
InvalidAlgorithmId	The algorithm specified is not a supported MAC algorithm or is not a valid algorithm ID
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID

3.4.6 CreateSignatureChannel Operation

This operation creates a signature channel. Signature channels are used to generate a digital signature over data.

3.4.6.1 Synopsis

IRSS::ChannelId CreateSignatureChannel(in CryptoModuleId cm, in EndpointId inputEndpoint, in SignatureAlgorithmId algorithmId, in CertificateId certId) raises(InvalidCertificateId, ChannelCreationError, InvalidModuleId, InvalidEndpointId, InvalidAlgorithmId);

3.4.6.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
inputEndpoint	EndpointId	The number identifying the signature channel's access point into the crypto module
algorithmId	SignatureAlgorithmId	The identifier of the Signature algorithm to use.
certId	CertificateId	The identifier of the Certificate to use.

3.4.6.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the signature channel created	Platform dependent

3.4.6.4 Originator

Waveform clients

3.4.6.5 Exceptions

Exception	Description
InvalidCertificateId	The certificate ID specified is not a valid certificate ID
ChannelCreationError	The channel could not be created
InvalidAlgorithmId	The algorithm specified is not a supported signature algorithm or is not a valid algorithm ID
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID

3.4.7 CreateSignatureVerificationChannel Operation

This operation creates a signature verification channel. Signature verification channels are used to verify a digital signature.

3.4.7.1 Synopsis

IRSS::ChannelId CreateSignatureVerificationChannel(in CryptoModuleId cm, in EndpointId inputEndpoint, in SignatureAlgorithmId algorithmId, in CF::OctetSequence publicKey) raises(ChannelCreationError, InvalidModuleId, InvalidEndpointId, InvalidKey, InvalidAlgorithmId);

3.4.7.2 Parameters

Parameter Name	Type	Description
cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
inputEndpoint	EndpointId	The number identifying the signature verification channel's input crypto module access point
algorithmId	SignatureAlgorithmId	The identifier of the Signature algorithm to use.
publicKey	CF::OctetSequence	The Public key used to verify the signature.

3.4.7.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the signature verification channel created.	Platform dependent

3.4.7.4 Originator

Waveform clients

3.4.7.5 Exceptions

Exception	Description
InvalidKey	The key specified is not a valid key
ChannelCreationError	The channel could not be created
InvalidAlgorithmId	The algorithm specified is not a supported signature algorithm or is not a valid algorithm ID
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID

3.4.8 CreateProtocolChannel Operation

This operation creates a protocol channel. Protocol channels are used to send and receive protocol messages to and from the cryptographic subsystem. Protocol channels can be single sided with a single endpoint ID. For a single sided protocol channel, the constant UNUSED_ENDPOINT_ID should be passed in for either the ptEndpoint or ctEndpoint parameters. Providing both endpoints results in a protocol channel capable of handling input from one security domain, processing by the CSS, and results delivered to a different security domain.

3.4.8.1 Synopsis

IRSS::ChannelId CreateProtocolChannel(in CryptoModuleId cm, in EndpointId ptEndpoint, in EndpointId ctEndpoint, in CrptoApplicationId protocolApplicationId) raises(ChannelCreationError, InvalidModuleId, InvalidEndpointId, InvalidCryptographicApplicationId, InvalidEndpointPair);

3.4.8.2 Parameters

Parameter Name	Type	Description
Cm	CryptoModuleId	The identifier of the Cryptographic module in which to create the channel
ptEndpoint	EndpointId	The number identifying the protocol channel's PT side crypto module access point
ctEndpoint	EndpointId	The number identifying the protocol channel's CT side crypto module access point.

protocolApplicationId	CryptoApplicationId	ID specifying the crypto application that contains the desired protocol.

3.4.8.3 Return Value

Type	Description	Valid Range
IRSS::ChannelId	The identifier of the protocol channel created	Platform dependent

3.4.8.4 Originator

Waveform clients

3.4.8.5 Exceptions

Exception	Description
ChannelCreationError	The channel could not be created
InvalidCryptoApplicationId	The crypto application ID specified is not a supported crypto application or is not a valid crypto application ID
InvalidModuleId	The crypto module ID is not a valid crypto module ID
InvalidEndpointId	The endpoint ID is not a valid endpoint ID
InvalidEndpointPair	A channel cannot be created between the endpoints specified.

3.4.9 DestroyChannel Operation

This operation destroys a channel. Cryptographic resources allocated to the channel are returned to the system and the channel can no longer be used after this operation returns.

3.4.9.1 Synopsis

void DestroyChannel(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.4.9.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the channel to be destroyed

3.4.9.3 Return Value

None

3.4.9.4 Originator

Waveform clients

3.4.9.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier

3.4.10 AddCryptographicConfiguration Operation

This operation adds a configuration to a cryptographic channel using the parameters passed in. Multiple configurations can be added to a channel, but only one configuration may be active at any time.

3.4.10.1 Synopsis

ConfigurationId AddCryptographicConfiguration(in IRSS::ChannelId channel, in CryptographicConfiguration configuration) raises(IRSS::InvalidChannelId, InvalidConfiguration);

3.4.10.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to add the configuration to.
configuration	CryptographicConfiguration	The Cryptographic configuration to add.

3.4.10.3 Return Value

Type	Description	Valid Range
ConfigurationId	The identifier of the configuration added.	Platform dependent

3.4.10.4 Originator

Waveform clients

3.4.10.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a cryptographic channel.
InvalidConfiguration	The configuration contains invalid or conflicting elements

3.4.11 AddTransecConfiguration Operation

This operation adds a configuration to a TRANSEC channel using the parameters passed in. Multiple configurations can be added to a channel, but only one may be active at any time.

3.4.11.1 Synopsis

ConfigurationId AddTransecConfiguration(in IRSS::ChannelId channel, in TransecConfiguration configuration) raises(IRSS::InvalidChannelId, InvalidConfiguration);

3.4.11.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the TRANSEC channel to add the configuration to.
configuration	TransecConfiguration	The TRANSEC configuration to add.

3.4.11.3 Return Value

Type	Description	Valid Range
ConfigurationId	The identifier of the configuration added.	Platform dependent

3.4.11.4 Originator

Waveform clients

3.4.11.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a TRANSEC channel.
InvalidConfiguration	The configuration contains invalid or conflicting elements

3.4.12 RemoveConfiguration Operation

This operation removes a configuration from a cryptographic or TRANSEC channel.

3.4.12.1 Synopsis

void RemoveConfiguration(in ConfigurationId channelConfigId) raises(InvalidConfigurationId);

3.4.12.2 Parameters

Parameter Name	Type	Description
channelConfigId	ConfigurationId	The identifier of the configuration to remove.

3.4.12.3 Return Value

None

3.4.12.4 Originator

Waveform clients

3.4.12.5 Exceptions

Exception	Description
InvalidConfigurationId	The configuration ID is not a valid configuration ID

3.4.13 ActiviateConfiguration Operation

This operation activates a previously added configuration on a cryptographic or TRANSEC channel. Any cryptographic state from the prior configuration is cleared. Although a DeactivateConfiguration() (see 3.4.14) is defined, it is permissible to switch to a new configuration by calling ActivateConfiguration() without first deactivating the current configuration.

3.4.13.1 Synopsis

void ActivateConfiguration(in ConfigurationId channelConfigId, in CF::OctetSequence activationData) raises(InvalidConfigurationId, ConfigurationActivationError);

3.4.13.2 Parameters

Parameter Name	Type	Description
channelConfigId	ConfigurationId	The identifier of the configuration to activiate (the channel is implied by this identifier).
activationData	CF::OctetSequence	Optional control or configuration information for use with the configuration being activated. Note that most configuration is set via the AddCryptographicConfiguration() (see 3.4.10) and AddTransecConfiguration() (see 3.4.11) operations.

3.4.13.3 Return Value

None

3.4.13.4 Originator

Waveform clients

3.4.13.5 Exceptions

Exception	Description
InvalidConfigurationId	The configuration ID is not a valid configuration ID
ConfigurationActivationError	A configuration could not be activated

3.4.14 DeactivateConfiguration Operation

This operation deactivates an active configuration on a cryptographic or TRANSEC channel. Any cryptographic state of the channel is lost. The channel itself is not destroyed.

3.4.14.1 Synopsis

void DeactivateConfiguration(in ConfigurationId channelConfigId) raises(IRSS::ConfigurationInactive, InvalidConfigurationId);

3.4.14.2 Parameters

Parameter Name	Type	Description
channelConfigId	ConfigurationId	The identifier of the configuration to be deactivated.

3.4.14.3 Return Value

None

3.4.14.4 Originator

Waveform clients

3.4.14.5 Exceptions

Exception	Description
IRSS::ConfigurationInactive	The configuration being deactivated is not an active configuration
InvalidConfigurationId	The configuration ID is not a valid configuration ID

3.5 IRSS::Control::KeyMgmt

3.5.1 UpdateKey Operation

This operation generates a new key from the existing key using a key update algorithm. This operation is used to generate an updated key for a key type that has only one available update algorithm.

The existing key is replaced by the new key.

3.5.1.1 Synopsis

void UpdateKey(in KeyId updateKeyId) raises(InvalidKeyId, KeyUpdateError);

3.5.1.2 Parameters

Parameter Name	Type	Description
updateKeyId	KeyId	The ID of the key to be updated

3.5.1.3 Return Value

None

3.5.1.4 Originator

Waveform clients

3.5.1.5 Exceptions

Exception	Description
InvalidKeyId	The key ID specified is not a valid key ID
KeyUpdateError	The key could not be updated

3.5.2 UpdateKeyWithAlgorithm Operation

This operation generates a new key from the existing key using a key update algorithm. The algorithm must be specified. This operation is used to update a key that has more than one available update algorithm.

The existing key is replaced by the new key.

3.5.2.1 Synopsis

void UpdateKeyWithAlgorithm(in KeyId updateKeyId, in KeyUpdateAlgorithmId algorithm) raises(InvalidKeyId, KeyUpdateError, InvalidKeyUpdateAlgorithmId);

3.5.2.2 Parameters

Parameter Name	Type	Description
updateKeyId	KeyId	The ID of the key to be updated
algorithm	KeyUpdateAlgorithmId	The identifier of the algorithm to use for updating the key

3.5.2.3 Return Value

None

3.5.2.4 Originator

Waveform clients

3.5.2.5 Exceptions

Exception	Description
InvalidKeyId	The key ID specified is not a valid key ID
KeyUpdateError	The key could not be updated
InvalidKeyUpdateAlgorithmId	The key update algorithm ID is not a valid key update algorithm ID for this key

3.5.3 GetUpdateCount Operation

This operation returns the number of times a key has been updated.

3.5.3.1 Synopsis

unsigned short GetUpdateCount(in KeyId updateCountKeyId) raises(InvalidKeyId);

3.5.3.2 Parameters

Parameter Name	Type	Description
updateCountKeyId	KeyId	The ID of the key whose update count is being requested

3.5.3.3 Return Value

Type	Description	Valid Range
unsigned short	The update count of the key requested.	Platform dependent

3.5.3.4 Originator

Waveform clients

3.5.3.5 Exceptions

Exception	Description
InvalidKeyId	The key ID specified is not a valid key ID

3.5.4 ZeroizeKey Operation

This operation destroys the designated key.

3.5.4.1 Synopsis

void ZeroizeKey(in KeyId zeroizeKeyId) raises(InvalidKeyId);

3.5.4.2 Parameters

Parameter Name	Type	Description
zeroizeKeyId	KeyId	The identifier of the Key to zeroize.

3.5.4.3 Return Value

None

3.5.4.4 Originator

Waveform clients

3.5.4.5 Exceptions

Exception	Description
InvalidKeyId	The key ID specified is not a valid key ID

3.6 IRSS::IandA::Channel

An abstract base class that allows clients to push data to the IRSS.

3.6.1 PushData Operation

This operation pushes data to the specified channel where it will be processed by the algorithm configured for that channel. Data size must not exceed the maximum data size as defined by GetMaxDataSize() (see 3.6.2).

3.6.1.1 Synopsis

void PushData(in IRSS::ChannelId channel, in CF::OctetSequence data) raises(IRSS::InvalidChannelId,MaxDataSizeExceeded);

3.6.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the channel receiving the data
data	CF::OctetSequence	The data being pushed into the IRSS

3.6.1.3 Return Value

None

3.6.1.4 Originator

Waveform clients

3.6.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for an I&A channel.
MaxDataSizeExceeded	A client made an attempt to push data that exceeded the maximum allowable size

3.6.2 GetMaxDataSize Operation

This operation returns the maximum data size, in octets, allowed on the specified channel. Data pushed via PushData() operation (see 3.6.1) must not exceed this size.

3.6.2.1 Synopsis

unsigned long GetMaxDataSize(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.6.2.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the Channel to get the max data size for.

3.6.2.3 Return Value

Type	Description	Valid Range
unsigned long	Maximum data size in octets.	Channel dependent

3.6.2.4 Originator

Waveform clients

3.6.2.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for an I&A channel.

3.6.3 Reset Operation

This operation resets the state of a the IandA channel. The channel is still configured with the information provided at channel creation time. Any computed values from the algorithm operating on the data pushed in via the PushData() operation (see 3.6.1) are reset. This operation should be called before reusing a channel for a new data set.

3.6.3.1 Synopsis

void Reset(in IRSS::ChannelId channel) raises (IRSS::InvalidChannelId);

3.6.3.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the Channel to reset.

3.6.3.3 Return Value

None

3.6.3.4 Originator

Waveform clients

3.6.3.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for an I&A channel.

3.7 IRSS::IandA::HashChannel

3.7.1 GetHash Operation

This operation returns the hash of the data pushed to the channel since it was created or last reset.

3.7.1.1 Synopsis

CF::OctetSequence GetHash(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId, InvalidState);

3.7.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the hash channel to use.

3.7.1.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The hash	Algorithm dependent

3.7.1.4 Originator

Waveform clients

3.7.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a hash channel.
InvalidState	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to the channel.

3.8 IRSS::IandA::MacChannel

3.8.1 GetMac Operation

This operation returns the MAC of the data pushed to the channel since it was created or last reset.

3.8.1.1 Synopsis

CF::OctetSequence GetMac(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId, InvalidState);

3.8.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the MAC channel to use.

3.8.1.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The MAC	Algorithm dependent

3.8.1.4 Originator

Waveform clients

3.8.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a MAC channel.
InvalidState	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to the channel

3.8.2 IsMacValid Operation

This operation verifies a MAC. When this operation is invoked, the security subsystem compares the passed in MAC to the MAC it has calculated on the data pushed via PushData() (see 3.6.1) since the channel was created or last reset. The result of the comparison is returned, indicating if the client has a valid MAC or not.

3.8.2.1 Synopsis

boolean IsMacValid(in IRSS::ChannelId channel, in CF::OctetSequence mac) raises(IRSS::InvalidChannelId, InvalidState, InvalidMac);

3.8.2.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the MAC channel to use.
mac	CF::OctetSequence	The MAC to be verified

3.8.2.3 Return Value

Type	Description	Valid Range
boolean	Indicates whether the passed in MAC is a valid MAC.	TRUE=The data is a valid MAC FALSE=The data is not a valid MAC

3.8.2.4 Originator

Waveform clients

3.8.2.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a MAC channel.
InvalidState	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to the channel
InvalidMac	The MAC given is not in the right size or format

3.9 IRSS::IandA::SignatureChannel

3.9.1 GetSignature Operation

This operation returns the digital signature of the data pushed to the channel since it was created or last reset.

3.9.1.1 Synopsis

CF::OctetSequence GetSignature(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId, InvalidState);

3.9.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the signature channel to use.

3.9.1.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The digital signature	Algorithm dependent

3.9.1.4 Originator

Waveform clients

3.9.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a signature channel.
InvalidState	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to the channel.

3.10 IRSS::IandA::SignatureVerificationChannel

3.10.1 IsSignatureValid Operation

This operation verifies a signature. When this operation is invoked, the security subsystem compares the passed in signature to the signature it has calculated on the data pushed via PushData() (see 3.6.1) since the channel was created or last reset. The result of the comparison is returned, indicating if the client has a valid signature.

3.10.1.1 Synopsis

boolean IsSignatureValid(in IRSS::ChannelId channel, CF::OctetSequence signature) raises(IRSS::InvalidChannelId, InvalidState, InvalidSignature);

3.10.1.2 Parameters

Parameter Name	**Type**	**Description**
channel	IRSS::ChannelId	The identifier of the signature verification channel to use.
signature	CF::OctetSequence	The signature to be verified

3.10.1.3 Return Value

Type	**Description**	**Valid Range**
boolean	Indicates whether the passed in signature matches.	TRUE=The passed in signature matches what the security subsystem generated FALSE=The passed in signature does not match what the security subsystem generated

3.10.1.4 Originator

Waveform clients

3.10.1.5 Exceptions

Exception	**Description**
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a signature verification channel.
InvalidState	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to the channel
InvalidSignature	The passed in signature is not in the right size or format.

3.11 IRSS::IandA::Random

3.11.1 GetPseudoRandomOperation

This operation returns a pseudorandom number.

3.11.1.1 Synopsis

CF::OctetSequence GetPseudoRandom(in unsigned short seed, in unsigned short numBytes);

3.11.1.2 Parameters

Parameter Name	Type	Description
seed	unsigned short	Number used to initialize the pseudorandom number generator
numBytes	unsigned short	Length of random number in octets

3.11.1.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The pseudorandom number	0 to $2^{(8*numBytes)}$-1

3.11.1.4 Originator

Waveform clients

3.11.1.5 Exceptions

None

3.11.2 GetRandom Operation

This operation returns a true random number.

3.11.2.1 Synopsis

CF::OctetSequence GetRandom(in unsigned short numBytes);

3.11.2.2 Parameters

Parameter Name	Type	Description
numBytes	unsigned short	Size of random number being requested in octets

3.11.2.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The random number.	0 to $2^{(8*numBytes)}$-1

3.11.2.4 Originator

Waveform clients

3.11.2.5 Exceptions

None

3.12 IRSS::Infosec::CryptographicChannel

This interface is used by waveform clients for encryption and decryption. It supports streaming modes and packet modes.

Streams have traditionally been employed by circuit switched legacy waveforms. Messages are defined across multiple calls to the IRSS. Message boundaries are defined by flagging packets with start of message (SOM) and end of message (EOM) flags. Typically, the cryptographic application will prepend a cryptgraphic preamble to the first encrypted packet.

Networking waveforms would typically use packet mode. With packet mode operation, each packet is its own message with an implied SOM and EOM. Many packet based cryptographic applications will include an initialization vector (IV) with each packet.

3.12.1 TransformStream Operation

Clients use the TransformStream() operation to transform (i.e. encrypt or decrypt depending on the source and destination) messages, as part of a streaming protocol as described in 3.12, where each message consists of one or more packets delimited with SOM and EOM flags. Clients must identify the first packet of a message by asserting the som parameter and the last packet of a message by asserting the eom parameter. If a message consists of a single packet, then clients should assert both the som and eom parameters. After the security subsystem transforms the packet, it will be pushed to the consumer interface of the other endpoint of the channel via PushStream() (see 3.13.1).

The packet size cannot exceed the maximum packet size, returned by GetMaxPacketSize().

When TransformStream() returns false, this constitutes a flow pause state. The client should not send more packets until SpaceAvailable() returns true, or until it receives a flow resume event through the IRSS::Infosec::ControlSignals interface.

3.12.1.1 Synopsis

boolean TransformStream(in IRSS::ChannelId channel, in boolean som, in boolean eom, in Packet streamPacket) raises(IRSS::InvalidChannelId, MaxPacketSizeExceeded, BadSomFlag, IRSS::ConfigurationInactive);

3.12.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to use.
som	boolean	TRUE=The packet is the first packet of a message. FALSE=The packet is not the first packet of a message

eom	boolean	TRUE=The packet is the last packet of a message. FALSE=The packet is not the last packet of a message
streamPacket	Packet	The packet to transform

3.12.1.3 Return Value

Type	**Description**	**Valid Range**
boolean	Indicates whether there is any remaining available space in the designated channel.	TRUE=There is available space and the client can continue pushing packets. FALSE=There is not available space (i.e.flow paused) and the client should discontinue pushing packets until space becomes available.

3.12.1.4 Originator

Waveform clients

3.12.1.5 Exceptions

Exception	**Description**
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not the identifier for a cryptographic channel.
MaxPacketSizeExceeded	The packet exceeded the maximum packet size
BadSomFlag	A packet tagged as SOM was received in the middle of a previously started message, or a packet to start a message was received without the SOM flag set
IRSS::ConfigurationInactive	An attempt was made to use a cryptographic channel that does not have an active configuration

3.12.2 TransformPackets Operation

Clients use the TransformPackets() operation to transform (i.e. encrypt or decrypt depending on the source and destination) packets, as part of a networking protocol as described in 3.12, where each packet is considered a self-contained message with implied SOM and EOM flags. For efficiency reasons, this operation takes in a payload consisting of a sequence of packets, allowing for reduced overhead. After the security subsystem transforms the packets, they will be pushed to the consumer interface of the other endpoint of the channel via PushPackets() (see 3.13.2).

No packet in the sequence can exceed the maximum packet size, returned by GetMaxPacketSize().

The total size of all the packets cannot exceed the maximum payload size returned by GetMaxPayloadSize().

When TransformPackets() returns false, this constitutes a flow pause state. The client should not send more packets until SpaceAvailable() returns true, or it receives a flow resume event through the IRSS::Infosec::ControlSignals interface.

3.12.2.1 Synopsis

boolean TransformPackets(in IRSS::ChannelId channel, in PacketSequence payload) raises(IRSS::InvalidChannelId, MaxPayloadSizeExceeded, MaxPacketSizeExceeded, IRSS::ConfigurationInactive);

3.12.2.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to use.
payload	PacketSequence	A sequence of one or more packets to be transformed.

3.12.2.3 Return Value

Type	Description	Valid Range
boolean	Indicates whether there is any remaining available space in the designated channel.	TRUE=There is available space and the client can continue pushing payloads. FALSE=There is not available space (i.e.flow paused) and the client should discontinue pushing payloads until space becomes available.

3.12.2.4 Originator

Waveform clients

3.12.2.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a cryptographic channel identifier
MaxPacketSizeExceeded	One or more packets in the payload exceeded the maximum packet size
MaxPayloadSizeExceeded	The entire payload exceeded the maximum payload size
IRSS::ConfigurationInactive	An attempt was made to use a cryptographic channel that does not have an active configuration

3.12.3 GetMaxPayloadSize Operation

This operation returns the maximum payload in octets that the channel can accept.

This applies to the sum of the packets pushed to the channel via a TransformPacket() call (see 3.12.2).

3.12.3.1 Synopsis

unsigned long GetMaxPayloadSize(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.12.3.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to query

3.12.3.3 Return Value

Type	Description	Valid Range
unsigned long	Maximum payload size in octets.	Channel dependent

3.12.3.4 Originator

Waveform clients

3.12.3.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a cryptographic channel identifier.

3.12.4 GetMaxPacketSize Operation

This operation returns the maximum packet size the IRSS can accept in octets.

Clients should not pass packets to the IRSS, via TransformStream() (see 3.12.1) or TransformPacket() (see 3.12.2), that are larger than this size.

3.12.4.1 Synopsis

unsigned long GetMaxPacketSize(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.12.4.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to query

3.12.4.3 Return Value

Type	Description	Valid Range
unsigned long	Maximum packet size in octets.	Channel dependent

3.12.4.4 Originator

Waveform clients

3.12.4.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a cryptographic channel identifier.

3.12.5 SpaceAvailable Operation

This operation returns a boolean indicating whether there is any space available for a transform request.

If a false is returned, the client should not push another packet until it receives a flow resume event through the IRSS::Infosec::ControlSignals interface or a subsequent call to SpaceAvailable() returns true.

3.12.5.1 Synopsis

boolean SpaceAvailable(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.12.5.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel to query

3.12.5.3 Return Value

Type	Description	Valid Range
boolean	Indicates whether there is any remaining available space in the designated channel.	TRUE=There is available space and the client can continue pushing packets/payloads. FALSE=There is not available space (i.e.flow paused) and the client should discontinue pushing packets/payloads until space becomes available.

3.12.5.4 Originator

Waveform clients

3.12.5.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a cryptographic channel identifier.

3.13 IRSS::Infosec::CryptographicConsumer

Waveform clients provide the IRSS::CrytographicConsumer interface. The IRSS uses this interface to push data to a client after a transform operation successfully completes. Flow

control is not employed in the interface to the client. Any buffering needed as part of an overall system flow control protocol must be implemented within the client.

3.13.1 PushStream Operation

This operation pushes one packet of a message to the client, after a successful transform operation completes as part of a streaming protocol as described in 3.12, where each message consists of one or more packets delimited with SOM and EOM flags. The IRSS will identify the first packet of a message by asserting the som parameter and the last packet of a message by asserting the eom parameter. If a message consists of a single packet, then the IRSS will assert both the som and eom parameters.

3.13.1.1 Synopsis

void PushStream(in IRSS::ChannelId channel, in boolean som, in boolean eom, in Packet streamPacket);

3.13.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel used to transform the packet.
som	boolean	TRUE=The packet is the first packet of a message. FALSE=The packet is not the first packet of a message
eom	boolean	TRUE=The packet is the last packet of a message. FALSE=The packet is not the last packet of a message
streamPacket	Packet	The transformed packet

3.13.1.3 Return Value

None

3.13.1.4 Originator

IRSS

3.13.1.5 Exceptions

None

3.13.2 PushPackets Operation

This operation pushes a sequence of one or more packets of data to the client, after a successful transform operation completes as part of a networking protocol as described in 3.12, where each packet is considered a self-contained message with implied SOM and EOM flags.

3.13.2.1 Synopsis

void PushPackets(in IRSS::ChannelId channel, in PacketSequence payload);

3.13.2.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the cryptographic channel used to transform the packet(s).
payload	PacketSequence	The sequence of transformed packets

3.13.2.3 Return Value

None

3.13.2.4 Originator

IRSS

3.13.2.5 Exceptions

None

3.14 IRSS::Infosec::ControlSignals

Flow control may be employed in the IRSS::Infosec::CryptographicChannel interface to the IRSS.

A client can be flow paused after pushing a packet/payload to the IRSS::Infosec::CryptographicChannel if that packet/payload fills the queues managed by the IRSS. The ControlSignals interface is the mechanism that the IRSS uses to notify a client that flow can once again resume.

3.14.1 FlowResume Operation

The IRSS uses this operation to signal to the client that flow can resume.

3.14.1.1 Synopsis

oneway void FlowResume(in IRSS::ChannelId channel);

3.14.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the cryptographic channel where flow can be resumed

3.14.1.3 Return Value

None

3.14.1.4 Originator

IRSS

3.14.1.5 Exceptions

None

3.15 IRSS::Infosec::TransecChannel

3.15.1 EncryptTransec Operation

This operation encrypts the supplied payload using the activated configuration for the supplied channel.

The seed and its related parameter, numSeedBits, are optional. If not provided (i.e. numSeedBits is zero), the cryptographic subsystem continues the previously seeded encryption.

The payload cannot exceed the maximum payload size returned by GetMaxPayloadSize().

3.15.1.1 Synopsis

void EncryptTransec(in IRSS::ChannelId channel, in CF::OctetSequence seed, in unsigned long numSeedBits, inout CF::OctetSequence payload) raises(IRSS::InvalidChannelId, BadTransecSeed, IRSS::ConfigurationInactive, MaxPayloadSizeExceeded);

3.15.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the TRANSEC channel where encryption is being requested
seed	CF::OctetSequence	Optional parameter used to initialize the encryption algorithm.
numSeedBits	unsigned long	Length of seed in bits. A seed is not necessarily an integer multiple of 8 bits
payload	CF::OctetSequence	Data to be encrypted

3.15.1.3 Return Value

None

3.15.1.4 Originator

Waveform clients

3.15.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a TRANSEC channel identifier
BadTransecSeed	The seed provided does not contain at least numSeedBits of seed data or does not contain the number of seed bits required by the algorithm
IRSS::ConfigurationInactive	An attempt was made to use a TRANSEC channel that does not have an active configuration
MaxPayloadSizeExceeded	The payload exceeded the maximum payload size

3.15.2 DecryptTransec Operation

This operation decrypts the supplied payload using the active configuration for the supplied channel.

The seed and its related parameter, numSeedBits, are optional. If not provided (i.e. numSeedBits is zero), the cryptographic subsystem continues the previously seeded decryption.

The payload cannot exceed the maximum payload size returned by GetMaxPayloadSize().

3.15.2.1 Synopsis

void DecryptTransec(in IRSS::ChannelId channel, in CF::OctetSequence seed, in unsigned long numSeedBits, inout CF::OctetSequence payload) raises(IRSS::InvalidChannelId, BadTransecSeed, IRSS::ConfigurationInactive, MaxPayloadSizeExceeded);

3.15.2.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the TRANSEC channel where decryption is being requested
seed	CF::OctetSequence	Optional parameter used to initialize the decryption algorithm.
numSeedBits	unsigned long	Length of seed in bits. A seed is not necessarily an integer multiple of 8 bits
payload	CF::OctetSequence	Data to be decrypted

3.15.2.3 Return Value

None

3.15.2.4 Originator

Waveform clients

3.15.2.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a TRANSEC channel identifier
BadTransecSeed	The seed provided does not contain at least numSeedBits of seed data or does not contain the number of seed bits required by the algorithm
IRSS::ConfigurationInactive	An attempt was made to use a TRANSEC channel that does not have an active configuration
MaxPayloadSizeExceeded	The payload exceeded the maximum payload size.

3.15.3 GenerateKeyStream Operation

This operation provides TRANSEC cover to a waveform client's transmission by having the security subsystem generate a TRANSEC keystream. The waveform applies the keystream to its transmission information directly.

The seed and its related parameter, numSeedBits, are optional. If not provided (i.e. numSeedBits is zero), the cryptographic subsystem continues the previously seeded keystream.

3.15.3.1 Synopsis

CF::OctetSequence GenerateKeyStream(in IRSS::ChannelId channel, in CF::OctetSequence seed, in unsigned long numSeedBits, in unsigned long numKeyStreamBits) raises(IRSS::InvalidChannelId, BadTransecSeed, IRSS::ConfigurationInactive);

3.15.3.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the TRANSEC channel from which the TRANSEC keystream is being requested
seed	CF::OctetSequence	Optional parameter used to initialize the keystream algorithm
numSeedBits	unsigned long	Length of seed in bits. A seed is not necessarily an integer multiple of 8 bits
numKeyStreamBits	unsigned long	Length of keystream being requested in bits

3.15.3.3 Return Value

Type	Description	Valid Range
CF::OctetSequence	The generated TRANSEC keystream	Algorithm dependent

3.15.3.4 Originator

Waveform clients

3.15.3.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a TRANSEC channel identifier.
BadTransecSeed	The seed provided does not contain at least numSeedBits of seed data or does not contain the number of seed bits required by the algorithm.
IRSS::ConfigurationInactive	An attempt was made to use a TRANSEC channel that does not have an active configuration

3.15.4 GetMaxPayloadSize Operation

This operation returns the channel's maximum payload size in octets. The payloads used in the EncryptTransec() (see 3.15.1) and DecryptTransec() (see 3.15.2) operations should not exceed this size.

3.15.4.1 Synopsis

unsigned long GetMaxPayloadSize(in IRSS::ChannelId channel) raises(IRSS::InvalidChannelId);

3.15.4.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The identifier of the TRANSEC channel to query

3.15.4.3 Return Value

Type	Description	Valid Range
unsigned long	Maximum payload size in octets	Channel dependent

3.15.4.4 Originator

Waveform clients

3.15.4.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a TRANSEC channel identifier

3.16 IRSS::Protocol::Channel

Waveform clients use the IRSS::Protocol::Channel interface to push protocol messages to the IRSS.

Each protocol's specific message details are provided in external extension documents. Both the waveform and IRSS need to implement the protocol per the protocol definition.

3.16.1 PushMessage Operation

This operation pushes a message to the designated channel.

The maximum message size for a protocol is specified in the protocol definition.

3.16.1.1 Synopsis

void PushMessage(in IRSS::ChannelId channel, in CF::OctetSequence message) raises(IRSS::InvalidChannelId, MaxMessageSizeExceeded, InvalidMessage, UnrecognizedMessage);

3.16.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the protocol channel to push the message to.
message	CF::OctetSequence	The message to push

3.16.1.3 Return Value

None

3.16.1.4 Originator

Waveform clients

3.16.1.5 Exceptions

Exception	Description
IRSS::InvalidChannelId	The channel identifier specified is not a valid channel identifier or is not a protocol channel identifier
MaxMessageSizeExceeded	The message pushed exceeds the maximum message size defined by the protocol.
InvalidMessage	The waveform client passed a message that is not valid for this protocol or is not valid at this time
UnrecognizedMessage	The waveform client passed a message that is not recognized by the IRSS

3.17 IRSS::Protocol::Consumer

Waveform clients provide the IRSS::Protocol::Consumer interface. The IRSS uses this interface to push protocol messages to the client.

3.17.1 PushMessage Operation

This operation pushes protocol messages to waveform clients.

3.17.1.1 Synopsis

void PushMessage(in IRSS::ChannelId channel, in CF::OctetSequence message);

3.17.1.2 Parameters

Parameter Name	Type	Description
channel	IRSS::ChannelId	The ID of the protocol channel used to push the message
message	CF::OctetSequence	The protocol message being pushed

3.17.1.3 Return Value

None

3.17.1.4 Originator

IRSS.

3.17.1.5 Exceptions

None

4 IDL

The following idl files were generated by MagicDraw version 17 and compiled with OIS OrbExpress idl2cpp Version 3.0.0 (FC04).

4.1 Irss.idl

```
/**
* IRSS.idl
*
* Comments have been omitted from this file.
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum.
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _IRSS_idl
#define _IRSS_idl

module IRSS
{

    typedef unsigned long ChannelId;

    exception InvalidChannelId
    {

    };

    exception ConfigurationInactive
    {

    };

};
#endif
```

4.2 Bypass.idl

```
/**
* Bypass.idl
*
* Comments have been omitted from this file.
```

```
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _Bypass_idl
#define _Bypass_idl
#include "CF.idl"
#include "IRSS.idl"

module IRSS
{
    module Bypass
    {

        exception MaxBypassSizeExceeded
        {

        };

        interface Consumer
        {
            void PushBypass( in CF::OctetSequence bypass );

        };

        exception PolicyViolation
        {

        };

        interface Channel
        {
            void PushBypass( in IRSS::ChannelId channel,
                             in CF::OctetSequence bypass )
                    raises( IRSS::InvalidChannelId,
                            MaxBypassSizeExceeded,
                            PolicyViolation );
            unsigned long GetMaxBypassSize( in IRSS::ChannelId channel )
                    raises( IRSS::InvalidChannelId );

        };

    };

};
#endif
```

4.3 Control.idl

```
/**
* Control.idl
*
* Comments have been omitted from this file.
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _Control_idl
#define _Control_idl
#include "CF.idl"
#include "IRSS.idl"

module IRSS
{
    module Control
    {
        typedef unsigned long EndpointId;

        const IRSS::Control::EndpointId UNUSED_ENDPOINT_ID = 0xFFFFFFFF ;

        enum Duplexity
        {
            SIMPLEX_RX,
            SIMPLEX_TX,
            FULL_DUPLEX,
            HALF_DUPLEX

        };

        typedef unsigned long KeyId;

        typedef unsigned long CryptoModuleId;

        exception InvalidCertificateId
        {

        };

        exception ChannelCreationError
        {
            string reason;

        };

        exception ConfigurationActivationError
        {
```

```
        string reason;

    };

    exception InvalidAlgorithmId
    {

    };

    exception InvalidConfiguration
    {

    };

    exception InvalidConfigurationId
    {

    };

    exception InvalidCryptoApplicationId
    {

    };

    exception InvalidEndpointId
    {

    };

    exception InvalidEndpointPair
    {

    };

    exception InvalidKey
    {

    };

    exception InvalidKeyId
    {

    };

    exception InvalidKeyUpdateAlgorithmId
    {

    };

    exception InvalidModuleId
    {

    };

    exception KeyUpdateError
    {
        string reason;

    };

    exception UnrecognizedCertificate
    {

    };

    typedef unsigned long CertificateId;

    typedef unsigned long ConfigurationId;

    typedef unsigned long CryptoApplicationId;
```

```
typedef unsigned long HashAlgorithmId;

typedef unsigned long KeyUpdateAlgorithmId;

typedef unsigned long MacAlgorithmId;

typedef unsigned long SignatureAlgorithmId;

typedef sequence<CertificateId> CertificateIdSequence;

typedef sequence<CryptoApplicationId> CryptoApplicationIdSequence;

struct CryptographicConfiguration
{
    IRSS::Control::CryptoApplicationId cryptoApplication;
    IRSS::Control::KeyId tek;
    IRSS::Control::Duplexity duplexity;
    CF::OctetSequence other;

};

struct TransecConfiguration
{
    IRSS::Control::CryptoApplicationId cryptoApplication;
    IRSS::Control::KeyId tsk;
    CF::OctetSequence other;

};

interface CertificateMgmt
{
    CF::OctetSequence RetrieveCertificate( in CertificateId certId )
            raises( InvalidCertificateId );
    CertificateIdSequence GetCertificateIds(  );
    boolean IsCertifcateValid( in CF::OctetSequence certificate )
            raises( UnrecognizedCertificate );

};

interface KeyMgmt
{
    void UpdateKey( in KeyId updateKeyId )
        raises( InvalidKeyId, KeyUpdateError );
    void UpdateKeyWithAlgorithm(
            in KeyId updateKeyId,
            in KeyUpdateAlgorithmId algorithm )
          raises(
              InvalidKeyId,
              KeyUpdateError,
              InvalidKeyUpdateAlgorithmId );
    unsigned short GetUpdateCount( in KeyId updateCountKeyId )
        raises( InvalidKeyId );
    void ZeroizeKey( in KeyId zeroizeKeyId ) raises( InvalidKeyId );

};

interface ChannelMgmt
{
    IRSS::ChannelId CreateCryptographicChannel(
            in CryptoModuleId cm,
            in EndpointId ptEndpoint,
            in EndpointId ctEndpoint,
            in CryptoApplicationIdSequence cryptoApps,
            in Duplexity channelDuplexity )
        raises( InvalidModuleId,
        InvalidEndpointId,
        InvalidEndpointPair,
        InvalidCryptoApplicationId,
        ChannelCreationError );
```

```
IRSS::ChannelId CreateTransecChannel(
        in CryptoModuleId cm,
        in EndpointId endpoint,
        in CryptoApplicationIdSequence cryptoApps,
        in Duplexity channelDuplexity )
    raises( InvalidModuleId,
            InvalidCryptoApplicationId,
            ChannelCreationError,
            InvalidEndpointId );
IRSS::ChannelId CreateBypassChannel(
        in CryptoModuleId cm,
        in EndpointId sourceEndpoint,
        in EndpointId destinationEndpoint )
    raises( ChannelCreationError,
            InvalidModuleId,
            InvalidEndpointId,
            InvalidEndpointPair );
IRSS::ChannelId CreateHashChannel(
        in CryptoModuleId cm,
        in EndpointId inputEndpoint,
        in HashAlgorithmId hashAlogrithm )
    raises( ChannelCreationError,
            InvalidModuleId,
            InvalidEndpointId,
            InvalidAlgorithmId );
IRSS::ChannelId CreateMacChannel(
        in CryptoModuleId cm,
        in EndpointId inputEndpoint,
        in MacAlgorithmId macAlogrithmId,
        in KeyId macKeyId )
    raises( InvalidKeyId,
            ChannelCreationError,
            InvalidAlgorithmId,
            InvalidModuleId,
            InvalidEndpointId );
IRSS::ChannelId CreateSignatureChannel(
        in CryptoModuleId cm,
        in EndpointId inputEndpoint,
        in SignatureAlgorithmId algorithmId,
        in CertificateId certId )
    raises( InvalidCertificateId,
        ChannelCreationError,
        InvalidModuleId,
        InvalidEndpointId,
        InvalidAlgorithmId );
IRSS::ChannelId CreateSignatureVerificationChannel(
        in CryptoModuleId cm,
        in EndpointId inputEndpoint,
        in SignatureAlgorithmId algorithmId,
        in CF::OctetSequence publicKey )
    raises( ChannelCreationError,
            InvalidModuleId,
            InvalidEndpointId,
            InvalidKey,
            InvalidAlgorithmId );
IRSS::ChannelId CreateProtocolChannel(
        in CryptoModuleId cm,
        in EndpointId ptEndpoint,
        in EndpointId ctEndpoint,
        in CryptoApplicationId protocolApplicationlId )
    raises( ChannelCreationError,
        InvalidModuleId,
        InvalidEndpointId,
        InvalidCryptoApplicationId,
        InvalidEndpointPair );
void DestroyChannel(
        in IRSS::ChannelId channel )
    raises( IRSS::InvalidChannelId );
ConfigurationId AddCryptographicConfiguration(
        in IRSS::ChannelId channel,
```

```
                    in CryptographicConfiguration configuration )
                raises( IRSS::InvalidChannelId,
                        InvalidConfiguration );
            ConfigurationId AddTransecConfiguration(
                    in IRSS::ChannelId channel,
                    in TransecConfiguration configuration )
                raises( IRSS::InvalidChannelId,
                        InvalidConfiguration );
            void RemoveConfiguration(
                    in ConfigurationId channelConfigId )
                    raises( InvalidConfigurationId );
            void ActivateConfiguration(
                    in ConfigurationId channelConfigId,
                    in CF::OctetSequence activationData )
                    raises( InvalidConfigurationId,
                            ConfigurationActivationError );
            void DeactivateConfiguration(
                    in ConfigurationId channelConfigId )
                    raises( IRSS::ConfigurationInactive,
                    InvalidConfigurationId );

        };

    };

};
#endif
```

4.4 IandA.idl

```
/**
* IandA.idl
*
* Comments have been omitted from this file.
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _IandA_idl
#define _IandA_idl
#include "CF.idl"
#include "IRSS.idl"

module IRSS
{
    module IandA
    {
```

```
exception InvalidMac
{

};

exception InvalidSignature
{

};

exception InvalidState
{

};

exception MaxDataSizeExceeded
{

};

interface Random
{
    CF::OctetSequence GetPseudoRandom(
            in unsigned short seed,
            in unsigned short numBytes );
    CF::OctetSequence GetRandom(
            in unsigned short numBytes );

};

abstract interface Channel
{
    void PushData(
            in IRSS::ChannelId channel,
            in CF::OctetSequence data )
        raises( IRSS::InvalidChannelId,
                MaxDataSizeExceeded );
    unsigned long GetMaxDataSize( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId );
    void Reset( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId );

};

interface HashChannel : Channel
{
    CF::OctetSequence GetHash( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId, InvalidState );

};

interface SignatureChannel : Channel
{
    CF::OctetSequence GetSignature( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId, InvalidState );

};

interface MacChannel : Channel
{
    CF::OctetSequence GetMac( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId, InvalidState );
    boolean IsMacValid(
            in IRSS::ChannelId channel,
            in CF::OctetSequence mac )
        raises( IRSS::InvalidChannelId, InvalidState, InvalidMac );

};
```

```
        interface SignatureVerificationChannel : Channel
        {
            boolean IsSignatureValid(
                    in IRSS::ChannelId channel,
                    in CF::OctetSequence signature )
                raises( IRSS::InvalidChannelId,
                        InvalidState,
                        InvalidSignature );

        };

    };

};
#endif
```

4.5 Infosec.idl

```
/**
* Infosec.idl
*
* Comments have been omitted from this file.
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _Infosec_idl
#define _Infosec_idl
#include "CF.idl"
#include "IRSS.idl"

module IRSS
{
    module Infosec
    {

        exception MaxPayloadSizeExceeded
        {

        };

        exception MaxPacketSizeExceeded
        {

        };
```

```
exception BadSomFlag
{

};

exception BadTransecSeed
{

};

struct Packet
{
    CF::OctetSequence payload;
    CF::OctetSequence bypass;

};

interface ControlSignals
{
    oneway void FlowResume( in IRSS::ChannelId channel );

};

typedef sequence<Packet> PacketSequence;

interface CryptographicConsumer
{
    void PushStream(
            in IRSS::ChannelId channel,
            in boolean som,
            in boolean eom,
            in Packet streamPacket );
    void PushPackets(
            in IRSS::ChannelId channel,
            in PacketSequence payload );

};

interface TransecChannel
{
    void EncryptTransec(
            in IRSS::ChannelId channel,
            in CF::OctetSequence seed,
            in unsigned long numSeedBits,
            inout CF::OctetSequence payload )
        raises( IRSS::InvalidChannelId,
                BadTransecSeed,
                IRSS::ConfigurationInactive,
                MaxPayloadSizeExceeded );
    void DecryptTransec(
            in IRSS::ChannelId channel,
            in CF::OctetSequence seed,
            in unsigned long numSeedBits,
            inout CF::OctetSequence payload )
        raises( IRSS::InvalidChannelId,
                BadTransecSeed,
                IRSS::ConfigurationInactive,
                MaxPayloadSizeExceeded );
    CF::OctetSequence GenerateKeyStream(
            in IRSS::ChannelId channel,
            in CF::OctetSequence seed,
            in unsigned long numSeedBits,
            in unsigned long numKeyStreamBits )
        raises( IRSS::InvalidChannelId,
                BadTransecSeed,
                IRSS::ConfigurationInactive );
    unsigned long GetMaxPayloadSize( in IRSS::ChannelId channel )
        raises( IRSS::InvalidChannelId );
```

```
        };

        interface CryptographicChannel
        {
            boolean TransformStream(
                    in IRSS::ChannelId channel,
                    in boolean som,
                    in boolean eom,
                    in Packet streamPacket )
                raises( IRSS::InvalidChannelId,
                        MaxPacketSizeExceeded,
                        BadSomFlag,
                        IRSS::ConfigurationInactive );
            boolean TransformPackets(
                    in IRSS::ChannelId channel,
                    in PacketSequence payload )
                raises( IRSS::InvalidChannelId,
                        MaxPayloadSizeExceeded,
                        MaxPacketSizeExceeded,
                        IRSS::ConfigurationInactive );
            unsigned long GetMaxPayloadSize( in IRSS::ChannelId channel )
                raises( IRSS::InvalidChannelId );
            unsigned long GetMaxPacketSize( in IRSS::ChannelId channel )
                raises( IRSS::InvalidChannelId );
            boolean SpaceAvailable( in IRSS::ChannelId channel )
                raises( IRSS::InvalidChannelId );

        };

    };

};
#endif
```

4.6 Protocol.idl

```
/**
* Protocol.idl
*
* Comments have been omitted from this file.
* Please refer to the IRSS API Specification for details.
*
* Copyright:
* This document has been prepared by the members of the International Security
* Services API Task Group to assist The Software Defined Radio Forum Inc. (or
* its successors or assigns, hereafter "the Forum"). It may be amended or
* withdrawn at a later time and it is not binding on any member of the Forum
* or of the International Security Services API Task Group. Contributors to
* this document that have submitted copyrighted materials (the Submission) to
* the Forum for use in this document retain copyright ownership of their
* original work, while at the same time granting the Forum a non-exclusive,
* irrevocable, worldwide, perpetual, royalty-free license under the Submitter's
* copyrights in the Submission to reproduce, distribute, publish, display,
* perform, and create derivative works of the Submission based on that original
* work for the purpose of developing this document under the Forum's own
* copyright. Permission is granted to the Forum's participants to copy any
* portion of this document for legitimate purposes of the Forum.  Copying for
* monetary gain or for other non-Forum related purposes is prohibited.
*
* Permission is granted to the Forum's participants to copy any portion of this
* document for legitimate purposes of the Forum
*
*         (c) The Software Defined Radio Forum Inc. doing business as
*             The Wireless Innovation Forum
*         First fixed in 2011, all rights reserved.
*/
#ifndef _Protocol_idl
#define _Protocol_idl
#include "CF.idl"
```

```
#include "IRSS.idl"

module IRSS
{
    module Protocol
    {

        interface Consumer
        {
            void PushMessage(
                in IRSS::ChannelId channel,
                in CF::OctetSequence message );

        };

        exception InvalidMessage
        {

        };

        exception MaxMessageSizeExceeded
        {

        };

        exception UnrecognizedMessage
        {

        };

        interface Channel
        {
            void PushMessage(
                in IRSS::ChannelId channel,
                in CF::OctetSequence message )
                raises( IRSS::InvalidChannelId,
                        MaxMessageSizeExceeded,
                        InvalidMessage,
                        UnrecognizedMessage );

        };

    };

};
#endif
```

5 UML

In this document, most of the descriptive UML has been placed in section 2.3, and typical port structures of IRSS components are shown in section 1.2. The subsections below detail non-interface specifics that support the main interfaces.

5.1 Data Types

5.1.1 IRSS::ChannelId

The *ChannelId* identifies a communications channel for exchanging information between waveform components and the IRSS.

typedef unsigned long ChannelId;

5.1.2 IRSS::Control::ConfigurationId

The *ConfigurationId* identifies a channel configuration (Cryptographic or TRANSEC).

typedef unsigned long ConfigurationId;

5.1.3 IRSS::Control::CryptoApplicationId

The *CryptoApplicationId* identifies a cryptographic application (e.g. AES)

typedef unsigned long CryptoApplicationId;

5.1.4 IRSS::Control::KeyId

The *KeyId* identifies an individual key within the security subystem.

typedef unsigned long KeyId;

5.1.5 IRSS::Control::KeyUpdateAlgorithmId

The *KeyUpdateAlgorithmId* identifies an algorithm to be used when a key update is requested.

typedef unsigned long KeyUpdateAlgorithmId;

5.1.6 IRSS::Control::EndpointId

The *EndpointId* identifies an access point into a crypto module and is implementation defined. Examples of types of endpoints include: physical hardware interfaces into a crypto module, IRSS API instance, and IP address.

typedef unsigned long EndpointId;

5.1.7 IRSS::Control::CryptoModulelId

The *CryptoModulelId* identifies a crypto module.

typedef unsigned long CryptoModuleId;

5.1.8 IRSS::Control::CertificateId

The CertificateId identifies a specific certificate within the security subsystem.

typedef unsigned long CertificateId;

5.1.9 IRSS::Control::HashAlgorithmId

The *HashAlgorithmId* identifies a specific algorithm for generating hashes.

typedef unsigned long HashAlgorithmId;

5.1.10 IRSS::Control::MacAlgorithmId

The *MacAlgorithmId* identifies a specific algorithm for generating MACs.

typedef unsigned long MacAlgorithmId;

5.1.11 IRSS::Control::SignatureAlgorithmId

The *SignatureAlgorithmId* identifies a specific algorithm for computing digital signatures.

typedef unsigned long SignatureAlgorithmId;

5.1.12 IRSS::Control::CryptoApplicationIdSequence

The *CryptoApplicationIdSequence* identifies the ids of one or more crypto applications (e.g. AES, DES, …). Each sequence element is of type *CryptoApplicationId* (see 5.1.3).

typedef sequence<CryptoApplicationId> CryptoApplicationIdSequence;

5.1.13 IRSS::Control::CertificateIdSequence

The *CertificateIdSequence* identifies the ids of the one or more certificates. Each sequence element is of type *CertificateId* (see 5.1.8).

typedef sequence<CertificateId> CertificateIdSequence;

5.1.14 IRSS::Infosec::PacketSequence

The *PacketSequence* consists of one or more packets. Each sequence element is of type IRSS::Infosec::*Packet* (see 5.4.3).

typedef sequence<Packet> PacketSequence;

5.2 Enumerations

5.2.1 IRSS::Control::EndpointId

For specifying Protocol channels with one endpoint, the UNUSED_ENDPOINT_ID is used for the second endpoint parameter.

Const IRSS::Control::EndpointId UNUSED_ENDPOINT_ID = 0xFFFFFFFF;

5.2.2 IRSS::Control::Duplexity

The *Duplexity* enumeration defines the four types of directional communication.

```
enum Duplexity
{
        SIMPLEX_TX,
        SIMPLEX_RX,
        FULL_DUPLEX,
        HALF_DUPLEX
};
```

5.3 Exceptions

5.3.1 IRSS::InvalidChannelId

exception InvalidChannelId { };

Exception	Attributes	Description	Type
InvalidChannelId	N/A	The channel identifier specified is not a valid channel identifier	N/A

5.3.2 IRSS:ConfigurationInactive

exception ConfigurationInactive { };

Exception	Attributes	Description	Type
ConfigurationInactive	N/A	A client attempted to deactivate in inactive configuration or use a channel without and active configuration.	N/A

5.3.3 IRSS::Bypass::MaxBypassSizeExceeded

exception MaxBypassSizeExceeded { };

Exception	Attributes	Description	Type
MaxBypassSizeExceeded	N/A	The maximum bypass size was exceeded	N/A

5.3.4 IRSS::Bypass::PolicyViolation

exception PolicyViolation { };

Exception	Attributes	Description	Type
PolicyViolation	N/A	The requested operation violates the bypass policy for the channel.	N/A

5.3.5 IRSS::Control::InvalidCertificateId

exception InvalidCertificateId { };

Exception	Attributes	Description	Type
InvalidCertificateId	N/A	The certificate ID is not a valid certificate ID	N/A

5.3.6 IRSS::Control::ChannelCreationError

exception ChannelCreationError {string reason };

Exception	Attributes	Description	Type
ChannelCreationError	reason	The channel could not be created (e.g. cryptographic resources are not available.). The reason attribute contains the reason for the channel creation failure.	string

5.3.7 IRSS::Control::ConfigurationActivationError

exception ConfigurationActivationError {string reason };

Exception	Attributes	Description	Type
ConfigurationActivationError	reason	The configuration could not be activated. The reason attribute contains the reason for the activation failure.	string

5.3.8 IRSS::Control::InvalidAlgorithmId

exception InvalidAlgorithmId { };

Exception	Attributes	Description	Type
InvalidAlgorithmId	N/A	The algorithm specified is not supported or is not a valid algorithm ID.	N/A

5.3.9 IRSS::Control::InvalidConfiguration

exception InvalidConfiguration { };

Exception	Attributes	Description	Type
InvalidConfiguration	N/A	The configuration contains invalid elements (e.g. invalid key ID) or conflicting elements.	N/A

5.3.10 IRSS::Control::InvalidConfigurationId

exception InvalidConfigurationId { };

Exception	Attributes	Description	Type
InvalidConfigurationId	N/A	The configuration ID is not a valid configuration ID	N/A

5.3.11 IRSS::Control::InvalidCryptoApplicationId

exception InvalidCryptoApplicationId { };

Exception	Attributes	Description	Type
InvalidCryptoApplicationId	N/A	The cryptographic application ID is not a valid configuration ID	N/A

5.3.12 IRSS::Control::InvalidEndpointId

exception InvalidEndpointId { };

Exception	Attributes	Description	Type
InvalidEndpointId	N/A	The endpoint ID is not a valid endpoint ID.	N/A

5.3.13 IRSS::Control::InvalidEndpointPair

exception InvalidEndpointPair { };

Exception	Attributes	Description	Type
InvalidEndpointPair	N/A	A channel cannot be created between the endpoints specified.	N/A

5.3.14 IRSS::Control::InvalidKey

exception InvalidKey { };

Exception	Attributes	Description	Type
InvalidKey	N/A	The key ID specified is not a valid key ID.	N/A

5.3.15 IRSS::Control::InvalidKeyUpdateAlgorithmId

exception InvalidKeyUpdateAlgorithmId { };

Exception	Attributes	Description	Type
InvalidKeyUpdateAlgorithmId	N/A	The key update algorithm ID is not a valid update algorithm ID for this type of key	N/A

5.3.16 IRSS::Control::InvalidModuleId

exception InvalidModuleId { };

Exception	Attributes	Description	Type
InvalidModuleId	N/A	The crypto module ID is not a valid crypto module ID.	N/A

5.3.17 IRSS::Control::KeyUpdateError

exception KeyUpdateError { string reason };

Exception	Attributes	Description	Type
KeyUpdateError	reason	The key could not be updated. The reason attribute contains the reason for the key update failure.	string

5.3.18 IRSS::Control::UnrecognizedCertificate

exception UnrecognizedCertificate { };

Exception	Attributes	Description	Type
UnrecognizedCertificate	N/A	the certificate data passed was not in the right format	N/A

5.3.19 IRSS::IandA::InvalidMac

exception InvalidMac { };

Exception	Attributes	Description	Type
InvalidMac	N/A	The MAC given is not the right size/format.	N/A

5.3.20 IRSS::IandA::InvalidSignature

exception InvalidSignature { };

Exception	Attributes	Description	Type
InvalidSignature	N/A	The signature given is not the right size/format.	N/A

5.3.21 IRSS::IandA::InvalidState

exception InvalidState { };

Exception	Attributes	Description	Type
InvalidState	N/A	The system is not in the correct state to complete the operation. For example, data has not yet been pushed to generate a result.	N/A

5.3.22 IRSS::IandA::MaxDataSizeExceeded

exception MaxDataSizeExceeded { };

Exception	Attributes	Description	Type
MaxDataSizeExceeded	N/A	A client made an attempt to push data that exceeded the maximum allowable size.	N/A

5.3.23 IRSS::Infosec::MaxPayloadSizeExceeded

exception MaxPayloadSizeExceeded { };

Exception	Attributes	Description	Type
MaxPayloadSizeExceeded	N/A	The entire payload exceeded the maximum payload size.	N/A

5.3.24 IRSS::Infosec::MaxPacketSizeExceeded

exception MaxPacketSizeExceeded { };

Exception	Attributes	Description	Type
MaxPacketSizeExceeded	N/A	One or more packets exceeded the maximum packet size.	N/A

5.3.25 IRSS::Infosec::BadSomFlag

exception BadSomFlag { };

Exception	Attributes	Description	Type

Exception	Attributes	Description	Type
BadSomFlag	N/A	A packet tagged as SOM was received in the middle of a previously started message, or a packet to start a message was received without the SOM flag set.	N/A

5.3.26 IRSS::Infosec::BadTransecSeed

exception BadTransecSeed { };

Exception	Attributes	Description	Type
BadTransecSeed	N/A	The seed provided does not contain at least numSeedBits of seed data or does not contain the number of seed bits required by the algorithm.	N/A

5.3.27 IRSS::Protocol::InvalidMessage

exception InvalidMessage { };

Exception	Attributes	Description	Type
InvalidMessage	N/A	The client passed a message that is not valid for this protocol or is not valid at this time.	N/A

5.3.28 IRSS::Protocol::MaxMessageSizeExceeded

exception MaxMessageSizeExceeded { };

Exception	Attributes	Description	Type
MaxMessageSizeExceeded	N/A	The maximum message size has been exceeded.	N/A

5.3.29 IRSS::Protocol::UnrecognizedMessage

exception UnrecognizedMessage { };

Exception	Attributes	Description	Type
UnrecognizedMessage	N/A	The waveform client passed a message that is not recognized by the IRSS.	N/A

5.4 Structures

5.4.1 IRSS::Control::CryptographicConfiguration

The *CryptographicConfiguration* structure defines the configuration of the cryptographic channel.

```
struct CryptographicConfiguration
{
        IRSS::Control::CryptoApplicationId cryptoApplication;
        IRSS::Control::KeyId tek;
        IRSS::Control::Duplexity duplexity;
        CF::OctetSequence other;
};
```

Struct	Attributes	Description	Type	Valid Range
Cryptographic Configuration	*cryptoApplication*	A *CryptoApplicationId* identifies the cryptographic application.	See 5.1.3	Platform dependent
	tek	Key Identifier of the Traffic Encryption Key (TEK) to be used with this configuration. Some CAs may allow for the selection of tek on a packet by packet basis. These CAs would typically ignore this attribute and specify the key as part of metadata contained within the packet.	See 5.1.4	Platform dependent
	duplexity	Duplexity defines the type of directional communication.	See 5.2.2	Enumeration See 5.2.2
	other	(optional) Additional information needed as required for the configuration.	CF::Octet Sequence	configuration dependent

5.4.2 IRSS::Control::TransecConfiguration

```
struct TransecConfiguration
{
        IRSS::Control::CryptoApplicationId cryptoApplication;
        IRSS::Control::KeyId tsk;
        CF::OctetSequence other;
};
```

Struct	Attributes	Description	Type	Valid Range
TransecConfiguration	*cryptoApplication*	A *CryptoApplicationId* identifies the cryptographic application.	See 5.1.3	Platform dependent
	tsk	Key Identifier of the TRANSEC (TSK) key to be used with this configuration.	See 5.1.4	Platform dependent
	other	(optional) Additional information needed as required for the configuration.	CF::OctetSequence	configuration dependent

5.4.3 IRSS::Infosec::Packet

```
struct Packet
{
        CF::OctetSequence payload;
        CF::OctetSequence bypass;
};
```

Struct	Attributes	Description	Type	Valid Range
Packet	*payload*	The data that is to be transformed.	CF::OctetSequence	Cryptographic application dependent
	bypass	inline data that is to be bypassed.	CF::OctetSequence	Cryptographic application dependent

5.5 Unions

None

Appendix A ACRONYMS

CA	Cryptographic Application
CF	Core Framework
CSS	Cryptographic SubSystem
CT	Ciphertext
EOM	End of Message
IRSS	International Tactical Radio Security Service
IV	Initialization Vector
MAC	Message Authentication Code
OE	Operating Environment
PT	Plaintext
SCA	Software Communications Architecture
SD	Security Domain
SDR	Software Defined Radio
SOM	Start of Message
TRANSEC	TRANsmission SECurity
WF	Waveform