

CCHEF – Covert Channels Evaluation Framework User Manual

Version 0.1

Sebastian Zander {szander@swin.edu.au}

May 21, 2008

Abstract

Communication is not necessarily made secure by the use of encryption alone. The mere existence of communication is often enough to raise suspicion and trigger investigative actions. Covert channels aim to hide the very existence of the communication. The huge amount of data and vast number of different protocols in the Internet makes it ideal as a high-bandwidth vehicle for covert communications. Covert channels are hidden inside pre-existing overt communication by encoding additional semantics onto ‘normal’ behaviours of the overt channels. We have developed CCHEF – a flexible and extensible software framework for evaluating covert channels in network protocols. The framework is able to establish covert channels across real networks using real overt traffic, but can also emulate covert channels based on overt traffic previously collected in trace files. In this paper we describe how to use CCHEF.

1 Introduction

Often it is thought that the use of encryption is sufficient to secure communication. However, encryption only prevents unauthorised parties from decoding the communication. In many cases the simple existence of communication or changes in communication patterns, such as an increased message frequency, are enough to raise suspicion and reveal the onset of events.

Covert channels aim to hide the very existence of the communication. They hide within pre-existing (overt) communications channels by encoding additional semantics onto ‘normal’ behaviours of the overt channels. The huge amount of data and vast number of different protocols in the Internet makes it ideal as a high-bandwidth vehicle for covert communications [1]. The capacity of covert channels in computer networks has greatly increased because of new high-speed network technologies, and this trend is likely to continue.

Covert channels are primarily used to circumvent existing information security policies, to exfiltrate information from an organisation or country in a manner that does not raise suspicions of the network owners or operators. Although network covert channels may not be used frequently today, because of increased measures against ‘open channels’, such as the free transfer of memory sticks in and out of organisations, the use of covert channels in computer networks will increase in the near future.

We have developed a software framework to evaluate covert channels in network protocols called Covert Channels Evaluation Framework (CCHEF). The main goals of CCHEF are to evaluate the capacity, security and robustness of network covert channels. Since CCHEF creates network covert channels, it does also provide the traffic data for evaluating countermeasures against network covert channels – mechanisms for their elimination, capacity limitation or detection. Furthermore, CCHEF should be able to create traffic for testing existing firewalls or intrusion detection software.

CCHEF can be used in real networks with real overt traffic, but can also emulate covert channels using overt traffic from trace files. Usually testing with real traffic is restricted to controlled testbeds where it is almost impossible to generate a realistic traffic mix from a larger number of hosts. Therefore, CCHEF also runs on single hosts emulating covert channels based on overt traffic from trace files. CCHEF supports both covert channels hidden in data fields (storage channels) and the timing of packets (timing channels).

The architecture of CCHEF is very flexible and extensible. New covert channels modules can be added without the need to modify the framework itself. Furthermore, new mechanisms for covert channel security (authentication, encryption), framing and (reliable) transport can easily be added to CCHEF. The design and implementation of CCHEF is described in [2]. This document describes how to install, use and extend CCHEF.

Section 2 describes how to install CCHEF. Section 3 provides an overview of how CCHEF is configured to perform different tasks. Section 4 describes all the modules implemented and their configuration options. Section 5 gives some examples on how to use CCHEF with the example configuration files included in the distribution. Section 6 describes how CCHEF can be further extended with new modules.

2 Installation

Download the CCHEF distribution and unpack it. Inside the main CCHEF directory you will find the following directories.

config	files for autoconf
dist	files to create an RPM package
doc	documentation
etc	example configuration files
src/cchef	source code of CCHEF
src/compare	source code of compare
src/lib	general code
src/modules	source code of covert channel encoding/decoding modules

Install all the libraries CCHEF needs:

- libxml2 (<http://xmlsoft.org>)
- libpcap (<http://www.tcpdump.org>)
- libtrace (<http://http://research.wand.net.nz/software/libtrace.php>)
- libnfnetworklink (<http://www.netfilter.org/projects/libnfnetworklink/index.html>)
- libnetfilter_queue (http://www.netfilter.org/projects/libnetfilter_queue/index.html)
- GNU Scientific Library (<http://www.gnu.org/software/gsl/>)
- Reed Solomon Library (<http://www.ka9q.net/code/fec/>)

The *libtrace* or *libpcap* library is required for trace file support (*libtrace* is recommended). The *libnfnetworklink* and *libnetfilter* libraries are required for real network support (packet interception and re-injection). The *gsl* library is required for the TTLRandom noise module (see Section 4.8.2). The Reed Solomon library is required for optional Forward Error Correction (FEC) in the *Simple* transport module (see Section 4.6.1).

Then install CCHEF in three simple steps. (You probably need to be root when doing the last command.)

```
./configure
make
make install
```

For more information see the INSTALL file in the root directory.

3 Configuration

A XML configuration file controls the behaviour of CCHEF. The configuration file is divided into several parts:

- The main section defines general settings e.g. the name of the log file and how detailed the logging is.
- Module specifications define the modulation modules.
- Device specifications define the input/output devices.
- Encryption specifications define encryption/decryption techniques used for the covert data.
- The packet selection specifications define what packets are used to embed covert information.
- Framing specifications define how covert data is split into frames and how frames are decoded at the receiver.
- Transport specifications define how covert data is transported e.g. error detection and recovery.
- Noise specifications define if and how noise is simulated.
- Finally, the covert channel is defined by specifying the devices for the input and output of covert data, the source of overt packets used as cover, and the encryption, framing, transport, packet selection modules used.

In each part configuration information is specified as preferences (PREFs). Preferences have a name, a value and (optionally) a type specification. The following is an example for a preference:

```
<PREF NAME="VerboseLevel" TYPE="UInt8">4</PREF>
```

Table 1 shows the existing data types. The type is an optional attribute (used for checking the values of preferences). If no type is specified, the default type of string is assumed.

Table 1: Existing data types for configuration preferences

Type	Explanation
UInt8	8 bit unsigned integer
SInt8	8 bit signed integer
UInt16	16 bit unsigned integer
SInt16	16 bit signed integer
UInt32	32 bit unsigned integer
SInt32	32 bit signed integer
UInt64	64 bit unsigned integer
SInt64	64 bit signed integer
Float	Single precision floating point
Double	Double precision floating point
Bool	Boolean (yes or no)
String	Character string
IPAddr	IPv4 address in dotted decimal notation or domain name

The following shows a configuration file with only the main part and the covert channel definition. The other parts of the configuration are explained later.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONFIG SYSTEM "config.dtd">
<CONFIG>
<MAIN>
  <!-- general settings -->
  <PREF NAME="VerboseLevel" TYPE="UInt8">4</PREF>
  <PREF NAME="LogFile">/home/szander/src/cchef/install/var/log/cchef.log</PREF>
</MAIN>
<!-- modulation modules -->
<!-- devices -->
<!-- encryption -->
<!-- packet selection -->
<!-- framing -->
<!-- transport -->
<!-- noise simulation -->
<!-- covert channel -->
<CHANNEL NAME="channell">
  <PREF NAME="CovertIn">InFile</PREF>
  <PREF NAME="CovertOut">OutFile</PREF>
  <PREF NAME="Cover">NFQueue</PREF>
  <PREF NAME="PacketSelection">All</PREF>
  <PREF NAME="Cryptor">XOR</PREF>
  <PREF NAME="Framer">SOF</PREF>
  <PREF NAME="Transport">Simple</PREF>
  <PREF NAME="Noise">TTLRandom</PREF>
  <PREF NAME="Modules">tll</PREF>
</CHANNEL>
</CONFIG>
```

A channel must specify `Cover` and either `CovertIn` or `CovertOut` (unidirectional channel) or both (bi-directional channel) using device names of devices specified in the device section of the configuration. It must also specify at least one module under `Modules` using the module name(s) as specified in the module section of the configuration. Multiple modules can be specified by separating the names with spaces. If multiple modules are specified covert data is encoded by the modules in the order they are specified. This means sender and receiver configuration files must specify modules in the same order; otherwise the communication will not work.

`PacketSelection`, `Cryptor`, `Framer`, `Transport` and `Noise` specifications are optional. By default all overt packets will be used to encode covert data, and the `SOF` and `framer` and `Simple` transport modules will be used. By default no encryption will be used and no noise is simulated.

The following specifies a modulation module. Each module must have a unique name. The preferences are module-specific (see Section 4.3).

```
<MODULE NAME="ttl">
  <PREF NAME="BitsPerPacket">1</PREF>
  <PREF NAME="Delta">1</PREF>
</MODULE>
```

A device specification must have a unique name and have the `Type` preference set to an existing device (see Section 4.4). Other preferences are device-specific.

```
<DEV NAME="InFile">
  <PREF NAME="Type">File</PREF>
  <PREF NAME="Filename">send.txt</PREF>
</DEV>
```

An encryption/decryption specification must have a unique name and `Type` must be set to an existing module. Currently the only existing module is `XOR` (see Section 4.7).

```
<CRYPT NAME="XOR">
  <PREF NAME="Type">XOR</PREF>
  <PREF NAME="Key">geheim</PREF>
</CRYPT>
```

A framer specification must have a unique name and the `Type` preference must be set to an existing module. Currently two framing methods are implemented. The `SOF` framer uses a start of frame byte and bit stuffing whereas the `CRC` framer identifies frames based on a CRC32 checksum (see Section 4.5).

```
<FRAMER NAME="SOF">
  <PREF NAME="Type">SOF</PREF>
</FRAMER>
```

A transport specification must have a unique name and the `Type` preference must be set to an existing module. Currently the only transport module implemented is the `Simple` module (see Section 4.6).

```
<TRANSPORT NAME="Simple">
  <PREF NAME="Type">Simple</PREF>
  <PREF NAME="BlockSize">16</PREF>
  <PREF NAME="Parity">0</PREF>
</TRANSPORT>
```

The noise specification is optional and is only available for testing covert channels with overt data from trace files. A noise specification must have a unique name and the `Type` preference must be set to an existing noise module. Currently two noise modules exist for Time to Live (TTL) covert channels. The `TTL` module emulates noise based on TTLs from trace files and the `TTLRandom` module generates artificial noise based on a Gaussian distribution (see Section 4.8).

```
<NOISE NAME="TTLRandom">
  <PREF NAME="Type">TTLRandom</PREF>
  <PREF NAME="StdDev">0.5</PREF>
</NOISE>
```

The packet selector specification is optional. By default all available packets are used to embed covert data. A packet selection specification must have a unique name and the `Type` preference set to an existing module. Currently the `All` module selects all packets and the `Hash` module samples packets with specified probability (see Section 4.9).

```

<SELECTOR NAME="All">
  <PREF NAME="Type">All</PREF>
</SELECTOR>

```

The next section explains the various modules and their preferences.

4 Module Preferences

This section describes the configuration of all modules of CCHEF. For each module preferences are listed in alphabetical order with optional preferences enclosed in square brackets ([]).

4.1 Main

The main part of CCHEF provides some preferences related to the log file and the location of the modulation modules.

Preference	Type	Default	Meaning
LogFile	String	<install>/var/cchef/cchef.log	Directory and name of log file.
ModuleDir	String	<install>/lib/cchef	Directory where the modulation modules are located.
[VerboseLevel]	UInt8	0	Verbosity of the log file (0=lowest, 4=highest).

4.2 Channel

This is the definition of the covert channel. Currently an instance of CCHEF can only create a single channel.

Preference	Type	Default	Meaning
[BothDirections]	Boolean	No	This option only works if using overt traffic from a trace file. If set to yes CCHEF will encode (and possibly decode) covert data in both directions.
CovertIn	String		Device from where covert data to be send is read.
CovertOut	String		Device where received covert data is written to.
Cover	String		Device from where overt packets (used as cover) are intercepted (and possibly re-injected).
[Cryptor]	String		Encryption/decryption module to use. If this preference is unspecified there is no encryption.
[FastTrace]	Boolean	No	If set to yes an instance of CCHEF will send covert data and receive the same covert data straight away. Works only with overt data from trace files.
[Framer]	String	SOF	Framing module to be used.
Modules	String		A space-separated list of covert channel modules. At least one module must be specified. If multiple modules are specified the order must be the same at sender and receiver.
[PacketSelection]	String	All	Packet selection module to be used.
[Transport]	String	Simple	Transport module to be used.

4.3 Modulation Modules

This section describes the implemented modulation modules.

4.3.1 basettl

This module is actually not a modulation module. This module generates a list of flow IDs and the most common TTL values in both direction of the flow. This list can then be used as input for the *NoiseTTL* module (see Section 4.8.1).

Preference	Type	Default	Meaning
FileName	String		Name of the file where flow IDs and TTL values are written. The format of each line of the file is: flow ID, common forward TTL, common backward TTL.

4.3.2 flowlen

This module is actually not a modulation module. This module generates a file with flow length statistics for unidirectional flows.

Preference	Type	Default	Meaning
FileName	String		Name of the file where flow length statistics are written. The format of each line of the file is: number of packets, number of bytes.

4.3.3 ttl

This module encodes covert data in the Time to Live (TTL) field as described in [3].

Preference	Type	Default	Meaning
[BitsPerPacket]	UInt8	1	Number of covert bits encoded per overt packet.
[Delta]	UInt8	1	Difference between 0-bit and 1-bit.

4.3.4 ttlnew

This module encodes covert data in the TTL field as described in [4]. This module encodes covert data into packet inter-arrival times. The module has no options. overt bits are encoded in the difference between two consecutive TTL values similar to the Alternate Mark Inversion (AMI) code.

Preference	Type	Default	Meaning
[Delta]	UInt8	1	Difference between 0-bit and 1-bit.

4.3.5 ttlnew2

This module encodes covert data in the TTL field directly into bits of the TTL as described in [4].

Preference	Type	Default	Meaning
[BitsPerPacket]	UInt8	1	Number of covert bits encoded per overt packet.

4.3.6 ttlqu04

This module encodes covert data directly into the TTL field's last significant bit as described in [5]. The module has no options.

4.3.7 ttlqu041

This module encodes covert bits mapped to certain TTL values as described in [5].

Preference	Type	Default	Meaning
[BitsPerPacket]	UInt8	1	Number of covert bits encoded per overt packet.
[Delta]	UInt8	1	Difference between 0-bit and 1-bit.

4.3.8 ttlu05

This module encodes covert bits into the difference of subsequent TTL values as described in [6].

Preference	Type	Default	Meaning
[Delta]	UInt8	1	Difference between 0-bit and 1-bit.

4.3.9 ipid

This module encodes covert data into the IP identification (ID) header fields as described in [7]. The module has no options.

4.3.10 pktiat_fixed

This module encodes covert data into packet inter-arrival times. Fixed absolute delays are used to encode 0 bits (small delay) and 1 bits (large delay). The module has no options.

4.3.11 pktiat_modulo

This module encodes covert data into packet inter-arrival times using the encoding technique described in [8]. The module has no options.

4.4 Devices

This section describes the implemented devices.

4.4.1 DeviceNull

This device simply discards any output data i.e. it writes data to /dev/null. The module has no preferences.

4.4.2 DeviceRandom

This device reads uniform random (equal probability of 0 bits and 1 bits) input data.

Preference	Type	Default	Meaning
[Seed]	UInt32	time(NULL)	Seed value for random number generator

4.4.3 DeviceFile

This device reads and/or writes data from/to ASCII text files.

Preference	Type	Default	Meaning
[Append]	Boolean	Yes	Appends to existing file when writing (instead of creating a new file).
FileName	String		Name of the file to read from or write to.
[Repeat]	Boolean	No	Automatically re-opens the file if the end of file is reached and continues reading.

4.4.4 DeviceTun

This device opens a network tunnel interface and reads/writes IP packets from/to this interface. The tunnel device allows tunneling IP packets sent by any networked applications across the covert channel.

Preference	Type	Default	Meaning
IPAddress	IPAddr		IP address of the tunnel device.
Network	String		A string of the form: <i>IP/mask</i> . The network that is routed across the tunnel. Currently the mask is limited to /8, /16 or /24.

4.4.5 DevicePcap

This device reads packets from a trace file in tcpdump format or from a live network interface.

Preference	Type	Default	Meaning
[Device]	String	eth0	Name of the device (live capturing) or file name (trace file).
Filter	String		A filter rule to select the overt traffic of the form: [src_host SrcIP] [AND] [dst_host DestIP] [AND] [src_port number] [AND] [dst_port number] [AND] [ip_proto protocol].
[Online]	Boolean	Yes	Set to yes if live capturing otherwise no.
[Promiscuous]	Boolean	Yes	Set interface to promiscuous mode (live capturing only).
[RecvBufSize]	UInt32	65535 (OS dependent)	Size of the socket buffer.
[SnapSize]	UInt16	68	Number of bytes captured per packet (live capturing only).

4.4.6 DeviceLTrace

This device reads packets from trace files in tcpdump or ERF format or from a live network interface. It is based on the *libtrace* library [9].

Preference	Type	Default	Meaning
[Device]	String	eth0	Name of the device (live capturing) or file name (trace file).
Filter	String		A filter rule to select the overt traffic of the form: [src_host SrcIP] [AND] [dst_host DestIP] [AND] [src_port number] [AND] [dst_port number] [AND] [ip_proto protocol].
[Online]	Boolean	Yes	Set to yes if live capturing otherwise no.
[OutputDevice]	String		Name of the file to write to.
[Promiscuous]	Boolean	Yes	Set interface to promiscuous mode (live capturing only).
[RecvBufSize]	UInt32	65535 (OS dependent)	Size of the socket buffer.
[SnapSize]	UInt16	68	Number of bytes captured per packet (live capturing only).

4.4.7 DeviceNFQueue

This device intercepts packets in the Linux kernel via Netfilter rules specifying the queue target [10]. It also re-injects intercepted packets back into the kernel.

Preference	Type	Default	Meaning
Filter	String		A filter rule to select the overt traffic of the form: <i>table</i> : [src_host SrcIP] [AND] [dst_host DestIP]. The <i>table</i> is the iptables table name. The table name is INPUT, OUTPUT or FORWARD for incoming, outgoing or forwarded traffic respectively.

4.5 Framing

This section describes the available framing methods.

4.5.1 SOF

This framing technique is based on a start of frame (SOF) byte and bit stuffing as used by HDLC. The module has no preferences.

4.5.2 CRC

This framing technique is based on detecting frames via CRC32 checksums as used by ATM. The module has no preferences.

4.6 Transport

This section describes the different transport modules.

4.6.1 TransportSimple

This is a simple transport module implementing segmentation/reassembly, sequence numbers for detecting loss of segments at the receiver, and forward error correction (FEC) based on Reed-Solomon codes (FEC only works if CCHEF is compiled with the Reed Solomon library, see section 2).

Preference	Type	Default	Meaning
[BlockSize]	UInt8	8	Number of covert bytes per transport block.
[Parity]	UInt8	0	Number of parity bytes per transport block (Reed-Solomon error detection and correction).

4.7 Encryption

This section describes the existing encryption modules

4.7.1 XOR

The XOR module ‘encrypts’ covert data as XOR of the data and the shared secret. While this is not very secure it demonstrates the use of encryption modules.

Preference	Type	Default	Meaning
Key	String		Shared secret between covert sender and receiver.

4.8 Noise

This section describes the available noise modules.

4.8.1 TTL

This module simulates noise for TTL covert channels based on real TTL variation from trace files. Before this module can be used the *basettl* module must be used on the same trace to extract the most common TTL values per flow (see Section 4.3.1).

In a first step the trace file is pre-processed running the sender with the *basettl* modulation module. This module generates a list of all packet flows (identified by their flow ID) and their ‘normal’ TTL TTL_{norm} . We assume the ‘normal’ TTL to be the most common TTL value and every other TTL value is an error induced by TTL variation. The actual error simulation then works as follows. The TTL value prior to modulation TTL_{prior} is stored and the new TTL value (including noise) is set to:

$$TTL = TTL + (TTL_{prior} - TTL_{norm}).$$

Preference	Type	Default	Meaning
FileName	String		Name of the file with common TTL values per flow (generated by the <i>basettl</i> modulation module).

4.8.2 TTLRandom

This module simulates random noise for TTL covert channels. It adds a random offset to TTL values based on a Gaussian distribution. The standard deviation can be configured through the configuration file. The type of distribution can be easily changed to any of the distributions supported by the Gnu Scientific Library (GSL) [11].

Preference	Type	Default	Meaning
[Seed]	UInt32	time(NULL)	Seed value for random number generator.
[StdDev]	Float	0.0	Standard deviation of the Gaussian distribution.

4.9 Packet Selection

This section describes the available modules for selecting overt packets.

4.9.1 All

This module simply selects every packet and has no preferences.

4.9.2 Hash

This module selects a certain fraction of packets (depending on the specified probability).

Preference	Type	Default	Meaning
[Probability]	Float	1.0	Probability of selecting a packet.

5 Using CCHEF

This section provides a number of examples of how to use CCHEF. The `man` subdirectory of the CCHEF distribution contains `man` pages for `cchef` and `compare` describing all the command line options. The `etc` subdirectory of the CCHEF distribution contains the example configuration files.

5.1 Covert Channel Emulation based on Packet Trace

CCHEF can either be run in fast trace mode or normal trace mode. In normal trace mode CCHEF needs to be run twice. Firstly, CCHEF is run as the sender reading overt traffic from the trace file, encoding the covert channel and then generating a new trace file containing the original traffic modified by the covert channel. Secondly, CCHEF is run as the receiver reading the modified trace file and decoding the covert channel. Modify the example configuration files `trace-send.xml` and `trace-recv.xml` to specify input and output trace file names (and any other part of the configuration). Then run CCHEF as follows:

```
cchef -c trace-send.xml
cchef -c trace-recv.xml
```

In fast trace mode CCHEF is only run once, acting as combined sender/receiver. It encodes the covert channel into the overt traffic from the trace and then decodes the covert channel straight away. Because in fast mode no modified trace is created (the performance bottleneck in normal mode) it is almost twice as fast. Modify the configuration file `trace-sr.xml` to specify the input trace file (and any other part of the configuration). Then run CCHEF as follows:

```
cchef -c trace-sr.xml
```

In trace file mode CCHEF can encode/decode the covert channels into all traffic flows in both directions. It can also encode/decode the covert channel in only a subset of the traffic as specified by a filter preference and/or limit the encoding to one direction.

CCHEF will log all bits send and received on the payload (covert data only) and the transport layer (all encrypted bits send across the covert channel including covert data, transport data and framing data) [2]. The `compare` application can be used to compute bit error rates and block error rates. To compute bit error rates on the transport layer run:

```
compare -i BitsSendTransport -o BitsRecvTransport
```

To compute bit and block error rates on the payload layer run:

```
compare -i BitsSendPayload -o BitsRecvPayload -b <block size>
```

The block size specified must be identical to the block size configured in `trace-send.xml/trace-recv.xml` or `trace-sr.html`. Otherwise block error rates will be incorrect.

5.2 Text-based Covert Channel Across a Network

This scenario requires two PCs – one for the covert sender and one for the covert receiver. Modify `text-send.xml` and `text-recv.xml` to specify the IP addresses of the sender and receiver. Furthermore, at the sender a text file with the message to be send needs to be created and the file name must be specified in `text-send.xml`. Then on the sender run:

```
cchef -c text-send.xml
```

And on the receiver run:

```
cchef -c text-recv.xml
```

The covert channel will be embedded in traffic flowing between the sender and the receiver. The example configuration files are written for the case where covert sender and receiver are also the overt sender and receiver. However, it is not required that the overt traffic is actually generated by the covert sender and covert receiver themselves. They could act as middlemen and use traffic of unwitting users flowing across them (e.g. they are both routers). In this case both configuration files must be modified: replace `INPUT` and `OUTPUT` by `FORWARD` in the `Filter` preferences.

5.3 IP-Tunnel Covert Channel Across a Network

This scenario requires two PCs – one for the covert sender and one for the covert receiver. Modify the files `tun-peer1.xml` and `tun-peer2.xml` to specify the IP addresses of the covert sender and receiver and the network routed across the tunnel between them. Then on the covert sender run:

```
cchef -c tun-peer1.xml
```

And on the covert receiver run:

```
cchef -c tun-peer2.xml
```

In this scenario CCHEF creates a bi-directional covert channel (relying on overt traffic flowing in both directions). The covert sender and receiver can simultaneously send IP packets to each other over the covert channel (so actually instead of a sender and a receiver we have two equal peers).

6 Extending CCHEF

This section describes how to extend CCHEF by adding new modules. Before modifying CCHEF it is highly recommended to read and understand the design and architecture of CCHEF [2] and to study the source code documentation generated by Doxygen.

6.1 Modulation Modules

Modulation modules are implemented as shared libraries. The interface for modulation modules looks as follows:

```
int initMod(configItemList_t params);
int doneMod();
int initFlow(void **flowdata);
int doneFlow(void *flowdata);
int resetFlow( void *flowdata );
int timeout(mtimer_t *timer, void *flowdata);
int getTimers(mtimer_t **timers, void *flowdata);
int encode(BitBuffer *bb, pkt_t *pkt, void *flowdata);
int decode(BitBuffer *bb, pkt_t *pkt, void *flowdata);
```

To add a new modulation module use an existing module as template and modify the code. Each module must implement the `encode` (called for each overt packet in send direction) and `decode` (called for each overt packet in receive direction) functions. The `initMod` and `doneMod` function are for allocating/deallocating and initialising global state; module preference parsing is also done in `initMod`. The `initFlow`, `doneFlow` and `resetFlow` functions are for allocating/deallocating and initialising per-flow state. The `getTimers` function initialises module timers (if needed) and everytime a timer expires the `timeout` function is called.

In order to compile and install the new module `Makefile.am` in `src/modules` has to be modified. Add `newmod.la` under `lib_LTLIBRARIES` and add the following lines (where `newmod` is the name of the new module):

```

newmod_la_LDFLAGS = $(COMMON_LDFLAGS)
newmod_la_SOURCES = newmod.cc <other sources>
newmod_la_LIBADD = ${COMMON_LIBADD}

```

6.2 Devices

To add a new device, derive a new C++ class from the Device base class. Every framer must implement the following methods (besides constructor and destructor):

```

virtual void addFilter(string filter);
virtual void delFilter(string filter);
virtual int receive(char *buf, unsigned long len, int flags);
virtual int send(char *buf, unsigned long len, int flags);
virtual DeviceStats *getStats();
virtual int getFd();
virtual int getDevInfo();
virtual int isOnline();

```

The addFilter and delFilter methods set a filter and remove a filter. Filters are usually only used for devices that handle IP packets and are used to filter the input. The receive method is called when new data is read from the device. The send method is called when data is sent to the device. The getStats method returns statistics about the device (e.g. number of bytes/packet read). The getFD method is used by CCHEF to learn the file descriptor the device works on (if any). The getDevInfo method is used by CCHEF to query the type of device (byte or packet) and the read/write mode. Finally, the isOnline method is used to specify whether a device reads packets from a live network or from a trace file. The method only needs to be implemented for devices that handle bytes (and is not necessary for byte devices).

In order to make the new device available in configuration files, few lines of code need to be added in Channel.cc in the createDevice method (replace NewDevice by the actual name):

```

...
} else if (type == "NewDevice") {
    dev = new NewDevice(name, mode, <more parameters>);
} else
...

```

In order to compile the new device and link it with CCHEF all new source files need to be added under ccchef_SOURCES in src/ccchef/Makefile.am.

6.3 Framer

To add a new framer, derive a new C++ class from the Framer base class. Every framer must implement the following two methods (besides constructor and destructor):

```

virtual BufferBlock *send(BufferBlock *in, pkt_t *overtPkt);
virtual BufferBlock *receive(BitBuffer *in, pkt_t *overtPkt);

```

The send method is called every time an overt packet in send direction arrives. The receive method is called whenever an overt packet in receive direction arrives. In order to make the new framer available in configuration files, few lines of code need to be added in Channel.cc in the createFramer method (replace NewFramer by the actual name):

```

...
} else if (type == "NewFramer") {
    framer = new NewFramer(name, trans->getBlockSize(),
                           trans->getHeadroomSize(), <more parameters>);
} else
...

```

In order to compile the new framer and link it with CCHEF all new source files need to be added under ccchef_SOURCES in src/ccchef/Makefile.am.

6.4 Transport

To add a new transport module, derive a new C++ class from the `Transport` base class. Every transport module must implement the following methods (besides constructor and destructor):

```
virtual BufferBlock *send(BitBuffer *in, pkt_t *overtPkt);
virtual BufferBlock *receive(BufferBlock *in, pkt_t *overtPkt);
virtual unsigned short getPayloadSize();
virtual unsigned short getBlockSize();
virtual unsigned short getHeadroomSize();
```

The `send` method is called to generate new transport packets from the covert data read from the `CovertIn` device. The `receive` method is called each time a new frame has been received over the covert channel. The `getPayloadSize`, `getBlockSize` and `getHeadroomSize` methods must return the number of payload (covert) bytes per transport packet, the total size in bytes of transport packets, and the number of bytes the framer should leave as headroom. In order to make the new transport module available in configuration files, few lines of code need to be added in `Channel.cc` in the `createTransport` method (replace `NewTransport` by the actual name):

```
...
} else if (type == "NewTransport") {
    trans = new NewTransport(name, <more parameters>);
} else
...

```

In order to compile the new transport module and link it with CCHEF all new source files need to be added under `cchef_SOURCES` in `src/cchef/Makefile.am`.

6.5 Encryption

To add a new encryption module, derive a new C++ class from the `Crypt` base class. Every encryption module must implement the following two methods (besides constructor and destructor):

```
virtual BufferBlock *encrypt(BufferBlock *in, int flags);
virtual BufferBlock *decrypt(BufferBlock *in, int flags);
```

The `encrypt` method encrypts a buffer and the `decrypt` method decrypts a buffer. In order to make the new framer available in configuration files, few lines of code need to be added in `Channel.cc` in the `createCryptor` method (replace `NewCryptor` by the actual name):

```
...
} else if (type == "NewCryptor") {
    crypt = new NewCryptor(name, <more parameters>);
} else
...

```

In order to compile the new encryption module and link it with CCHEF all new source files need to be added under `cchef_SOURCES` in `src/cchef/Makefile.am`.

6.6 Noise

To add a new noise module, derive a new C++ class from the `Noise` base class. Every noise module must implement the following two methods (besides constructor and destructor):

```
virtual void pre(pkt_t *overtPkt);
virtual void post(pkt_t *overtPkt);
```

The `pre` method is called before the covert channel is embedded into the overt traffic and allows the noise module to learn the original characteristics of the overt traffic. The `post` method is called after the covert channel has been inserted in the overt traffic and here the module can create noise by changing header fields or packet timing. In order to make the new noise module available in configuration files, few lines of code need to be added in `Channel.cc` in the `createNoise` method (replace `NewNoise` by the actual name):

```
...
} else if (type == "NewNoise") {
    noise = new NewNoise(name, <more parameters>);
} else
...

```

In order to compile the new noise module and link it with CCHEF all new source files need to be added under `cchef_SOURCES` in `src/cchef/Makefile.am`.

6.7 Packet Selection

To add a new packet selection module, derive a new C++ class from the `PacketSelector` base class. Every packet selection module must implement the following method (besides constructor and destructor):

```
virtual int select(pkt_t *pkt);
```

The `select` method is called for every overt packet. It needs to return a 1 if the overt packet should be used as carrier for the covert channel or a 0 otherwise. In order to make the new packet selection module available in configuration files, few lines of code need to be added in `Channel.cc` in the `createSelector` method (replace `NewSelector` by the actual name):

```
...
} else if (type == "NewSelector") {
    sel = new NewSelector(name, <more parameters>);
} else
...

```

In order to compile the new packet selection module and link it with CCHEF all new source files need to be added under `cchef_SOURCES` in `src/cchef/Makefile.am`.

References

- [1] S. Zander, G. Armitage, P. Branch, "A Survey of Covert Channels and Countermeasures in Computer Network Protocols," *IEEE Communications Surveys and Tutorials*, vol. 9, pp. 44–57, October 2007.
- [2] S. Zander, G. Armitage, "CCHEF – Covert Channels Evaluation Framework Design and Implementation," Tech. Rep. 080530A, CAIA Technical Report, May 2008. <http://caia.swin.edu.au/reports/080530A/CAIA-TR-080530A.pdf>.
- [3] S. Zander, G. Armitage, P. Branch, "Covert Channels in the IP Time To Live Field," in *Proceedings of Australian Telecommunication Networks and Applications Conference (ATNAC)*, December 2006.
- [4] S. Zander, G. Armitage, P. Branch, "An Empirical Evaluation of IP Time To Live Covert Channels," in *Proceedings of IEEE International Conference on Networks (ICON)*, November 2007.
- [5] H. Qu, P. Su, D. Feng, "A Typical Noisy Covert Channel in the IP Protocol," in *Proceedings of 38th Annual International Carnahan Conference on Security Technology*, pp. 189–192, October 2004.
- [6] N. B. Lucena, G. Lewandowski, S. J. Chapin, "Covert Channels in IPv6," in *Proceedings of Privacy Enhancing Technologies (PET)*, pp. 147–166, May 2005.
- [7] C. H. Rowland, "Covert Channels in the TCP/IP Protocol Suite," *First Monday*, Peer Reviewed Journal on the Internet, July 1997.
- [8] G. Shah, A. Molina, M. Blaze, "Keyboards and Covert Channels," in *Proceedings of USENIX Security Symposium*, August 2006.
- [9] WAND Network Research Group, "libtrace – A Library for Trace Processing." <http://research.wand.net.nz/software/libtrace.php>.
- [10] H. Welte, "Netfilter Queue Library." http://www.netfilter.org/projects/libnetfilter_queue/.
- [11] GSL Team, "GSL – GNU Scientific Library." <http://www.gnu.org/software/gsl/>.