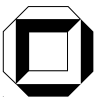

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

PktAnon Manual

Generic Framework for Profile-based Traffic Anonymization

build date June 26, 2008

Institute of Telematics
University of Karlsruhe

Christoph P. Mayer, Thomas Gamer, Dr. Marcus Schöller

Contact: mayer@tm.uka.de
Web: www.tm.uka.de/pktanon

Contents

1. Introduction

Computer network researchers, system engineers and network operators have an increasing need for network traces. These are necessary to build and evaluate communication systems. This ranges from developing intrusion detection systems over evaluating network protocols or system design decisions, up to education in network security. Unfortunately, availability of real-world traces is very scarce, mainly due to privacy and security concerns. Making recorded data anonymous helps to mitigate this problem. Available anonymization systems, however, do not provide sufficient flexibility, extensibility or ease of use.

Therefore, we developed *PktAnon*—a generic framework for network traffic anonymization. PktAnon aims at enabling wide-spread use of realistic network traffic traces by providing fine-grained, profile-based anonymization of network traffic traces. It can easily be configured by anonymization profiles, which are used to define how the network traffic should be made anonymous. Such profiles ensure an easy adaptation of the information actually being made anonymous to different environments or local legislation. Furthermore, our framework supports flexible application of arbitrary anonymization primitives to every protocol field. Due to its extensibility our framework provides an easy incorporation of new anonymity-enhancing techniques, too. Additionally, it prevents accidental disclosure of private data by applying a technique called defensive transformation. Finally, it can be used for online as well as offline anonymization of network traffic.

Further reading

The following publications covering PktAnon exist:

- **PktAnon: A Generic Framework for Profile-based Traffic Anonymization**, Thomas Gamer, Christoph P. Mayer and Marcus Schöller, PIK Praxis der Informationsverarbeitung und Kommunikation, 2/2008.
- **Datenschutzkonforme Anonymisierung von Datenverkehr auf einem Vermittlungssystem**, Thomas Gamer, Christoph P. Mayer and Marcus Schöller, Vortrag auf dem 2. Essener Workshop "'Neue Herausforderungen in der Netzsicherheit"' an der Universität Duisburg/Essen, October 2006.

- **Datenschutzkonforme Anonymisierung von Datenverkehr auf einem Vermittlungssystem**, Christoph P. Mayer, Bachelor Thesis, Institute of Telematics, University of Karlsruhe, July 2006.

Disclaimer

PktAnon comes with no warranty! Use it at your own risk. Please note that PktAnon is currently under development and not in a final state.

2. Anonymization primitives

PktAnon supports a large number of predefined anonymization primitives which can be mapped to protocol attributes using XML-based profiles (see Section 4). Extending PktAnon by additional anonymization primitives is described in Section 6.2.

Table 2.1 shows the currently available anonymization primitives and their according parameters that have to be specified. Every anonymization primitive can be applied to every protocol attribute, therefore allowing high flexibility in defining anonymization profiles. The available anonymization profiles in the following are explained in alphabetical order.

PktAnon is able to recalculate all length and checksum fields when reassembling packets after anonymization. Therefore, the resulting network trace is completely well-formed. This holds true even if the AnonShorten primitive is applied on some protocol attributes.

PktAnon additionally supports chaining of anonymization primitives. This way, special handling e. g. for broadcast IP and MAC addresses can be achieved. Considering n linearly chained anonymization primitives $a_1, \ldots, a_n$, the anonymization primitive a_{i+1} is called only if the anonymization primitive a_i returns true. Otherwise, processing of the primitive chain is aborted.

AnonIdentity

Every supported protocol attribute must be assigned an anonymization primitive in the anonymization profile in order to prevent configuration errors where protocol attributes are simply forgotten to be specified. Thus, protocol attributes that are meant not to be made anonymous also need to have an anonymization primitive assigned. In this case the AnonIdentity primitive is suitable since it just preserves the original data.

AnonBroadcastHandler

This special handler is used to prevent anonymization of broadcast IP and MAC addresses. Addresses having all bits set to 1 are identified as broadcast addresses

in a generic manner. Since PktAnon can handle chains of anonymization primitives for a single protocol attribute, application of this primitive in front of the normal anonymization primitives will prevent anonymization of broadcast addresses and perform anonymization on all other addresses.

AnonShorten

This anonymization primitive can be used to shorten or completely remove optional protocol attributes. IP Options are a common example. A further example is payload data. Such data can be removed by application of the AnonShorten primitive. The new length of the data must be specified in the anonymization profile. If 0 is specified as new length data is completely removed. This anonymization will fail if the protocol item is not optional and therefore, can not be shortened.

AnonConstOverwrite

This anonymization primitive overwrites each byte of the data with a constant byte value. This value must be specified in the anonymization profile.

AnonContinuousChar

In contrast to the AnonConstOverwrite anonymization primitive that overwrites each byte of the complete data with a constant value, this primitive uses a continuously incremented byte value. Therefore, the output is a series of bytes, each byte having an incremented decimal value.

AnonRandomize

This anonymization primitive replaces each byte of the data with random byte values.

AnonShuffle

This anonymization primitive manipulates the data by shuffling the single bytes of the data randomly. Therefore, the original bytes are preserved but the order of the bytes is manipulated.

AnonWhitenoise

This anonymization primitive adds noise on bit-level. The complete data is interpreted as an array of bits. The strength value specified in the anonymization profile, defines the percentage of bits that are randomly chosen and toggled in steps of 10%, i. e. a value of 3 toggles 30% of the data bits randomly.

AnonBytewiseHashSha1

This anonymization primitive applies a hash function on each byte of the data using SHA1. The resulting hash value in this case is truncated to a length of one byte.

AnonHashSha1

This anonymization primitive applies a hash function on the complete data using SHA1. The resulting hash value is truncated to the actual length of the buffer.

AnonBytewiseHashHmacSha1

This anonymization primitive applies an HMAC-SHA1 hash on each byte of the data. A key for the HMAC must be specified in the anonymization profile. The resulting hash value in this case is truncated to a length of one byte.

AnonHashHmacSha1

The AnonHashHmacSha1 anonymization is similar to the AnonBytewiseHashSha1 anonymization primitive. It, however, hashes the complete buffer using HMAC-SHA1 instead of each single byte. The HMAC needs a key which must be specified in the anonymizationprofile. The resulting hash value is truncated to the actual length of the buffer.

AnonCryptoPan

This is a prefix-preserving anonymization that was developed by Fan et al [1]. It needs an input key for the Rijandel encryption scheme. This anonymization primitive has the property to preserve the prefixes of any two addresses on a bit-wise basis.

[1] Jinliang Fan and Jun Xu and Mostafa H. Ammar and Sue Moon, Cryptography-based Prefix-preserving Anonymization, http://www-static.cc.gatech.edu/computing/Networking/projects/cryptopan, 2004

Name	Parameters	Description
`AnonIdentity`	-	Preserve original data.
`AnonBroadcastHandler`	-	Preserve broadcast IP or MAC addresses.
`AnonShorten`	`newlen`	Cut the buffer to the given length.
`AnonConstOverwrite`	`anonval`	Overwrite every byte with the provided value, given in hex (e. g. 0x00).
`AnonContinuousChar`	-	Overwrite every byte with continuous values.
`AnonRandomize`	-	Overwrite each byte with a random value
`AnonShuffle`	-	Shuffle the bytes of the buffer randomly.
`AnonWhitenoise`	`strength`	Apply bit-based noise of strength between 1 and 10.
`AnonBytewiseHashSha1`		Hash every byte separately with SHA1. The
`AnonHashSha1`	-	Hash the complete buffer with SHA1.
`AnonBytewiseHashHmacSha1`	`key`	Hash every byte separately with HMAC-SHA1. The tt key parameter is needed as key input for the HMAC.
`AnonHashHmacSha1`	`key`	Hash the complete buffer with HMAC-SHA1. The tt key parameter is needed as key input for the HMAC.
`AnonCryptoPan`	`key`	Prefix-preserving anonymization. The `key` parameter is needed for the for Rijandel algorithm used inside the prefix-preserving anonymization.

Table 2.1: Overview of anonymization primitives in PktAnon

3. Supported protocols

PktAnon supports a large number of network protocols. Every protocol attribute can be made anonymous using one of the anonymization primitives described in Section 2. Table 3.1 shows a complete list of protocol and attribute names that currently can be used in an anonymization profile.

Protocol	Attribute	Description
EthernetPacket	MacSource	the source mac address
	MacDest	the destination mac address
	MacType	the type attribute which specified the next higher protocol
ArpPacket	ArpHardwaretp	hardware type
	ArpPrototp	protocol type
	ArpHardwareaddrlen	hardware address length
	ArpProtoaddlen	protocol address length
	ArpOpcode	opcode
	ArpSourcemac	source mac address
	ArpSourceip	source ip address
	ArpDestmac	destination mac address
	ArpDestip	destination ip address
IpPacket	IpTos	type of service
	IpIdent	identifier
	IpFragoffset	fragmentation offset
	IpFlags	ip flags
	IpTtl	time to live
	IpSourceip	source ip address
	IpDestip	destination ip address
	IpOptions	ip options
Ipv6Packet	Ipv6Trafficclass	traffic class
	Ipv6Flowlabel	flow label
	Ipv6Hoplimit	hop limit
	Ipv6Sourceaddr	source address
	Ipv6Destaddr	destination address
UdpPacket	UdpSourceport	the source port
	UdpDestport	the dest port
TcpPacket	TcpSourceport	the source port
	TcpDestport	the dest port
	TcpSeqnum	sequence number
	TcpAcknum	ack number
	TcpFlags	flags
	TcpWindowsize	window size
	TcpUrgentpnt	urgent pointer
	TcpOptions	tcp options
IcmpPacket	IcmpType	type
	IcmpCode	code
	IcmpMisc	misc, depending on type and code
PayloadPacket	PayloadPacketData	the data

Table 3.1: Supported network protocols and protocol attributes in PktAnon

4. Anonymization profiles

PktAnon uses anonymization profiles for configuration. These profiles are based on XML and contain all information necessary for the anonymization process. This includes information about traffic source, mapping of anonymzation primitives to protocol attributes, and other things like runtime measurement.

The structure of the anonymization profile is explained in this section. It consists of two main parts: `Settings` and `AnonMappings`. The `Settings` module contains XML elements for specification of general information. The actual anonymization configuration is defined in the `AnonMappings` module. A detailed example of such an anonymization profile is given in Section 4.3.1.

4.1 Settings

The `Settings` module contains basic information about the anonymization process, e. g. the traffic source, general settings like checksum handling, or which runtime information should be gathered and displayed.

4.1.1 Flow

The `Flow` submodule contains traffic source and sink of the anonymization process. The `Input` and `Output` configuration items can either be file names or the predefined keywords `stdin` or `stdout`, respectively. This allows for several combinations of files and/or standard input/output streams to be handled.

4.1.2 Misc

The `Misc` submodule contains the following configuration items deciding on handling of runtime information:

- `UseMeasure`: Boolean value (0|1) indicating whether runtime information should be gathered. PktAnon supports measurement of runtime information like bytes or packets per second that were processed by the anonymization process. If `UseMeasure` is set, the configuration item `MeasureFile` has to be specified, too.

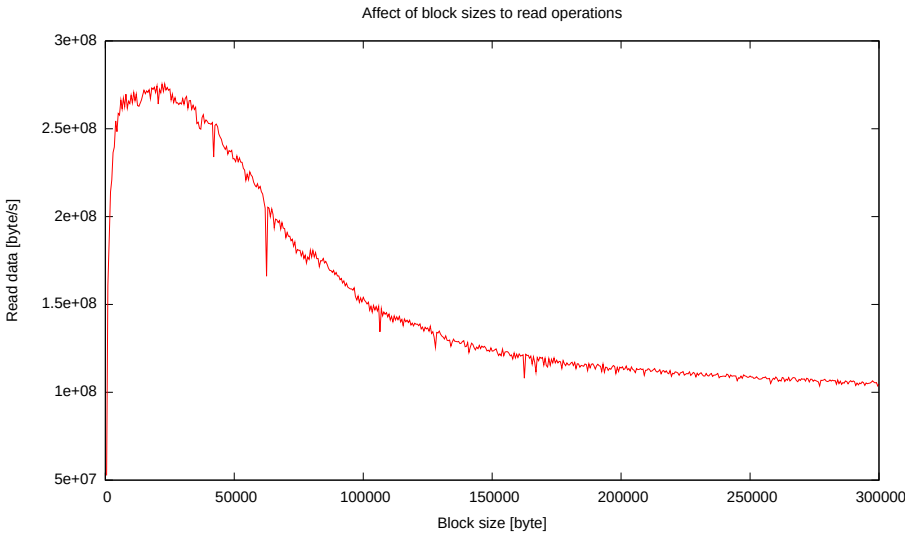

Figure 4.1: Effect of read block size in hard disc system read.

- `MeasureFile`: File name of the file gathered runtime information is stored in. Only considered if `UseMeasure` is set.
- `CreatePlot`: If `UseMeasure` is set and a `MeasureFile` is given, this boolean option (0|1) enables creation of a live plot. Therefore, the gnuplot software is used to plot the runtime information collected.
- `PrintPackets`: Boolean value (0|1) indicating whether each original and anonymous packet should be printed to console in human readable format.
- `SingleStepping`: Boolean value (0|1) indicating that after each anonymous packet the user must press any key to continue anonymization. This option is only sensible in combination with `PrintPackets` and can be used for debugging purposes.
- `BlockSizeRead`: Size of the blocks to read during a single read system call. A default block size of 1024 is used if this value is not set explicitely.
- `BlockSizeWrite`: Same as `BlockSizeRead` for write system calls instead of read.

The `SingleStepping` option is best used in combination with `PrintPackets`. This allows direct visual inspection of original and anonymous packets.

The `BlockSizeRead` and `BlockSizeWrite` sizes can affect performance of anonymization. Often, speed of the hard disc is the limiting factor in anonymization. Figure 4.1 shows the data rate of read calls dependant on the block size used. Thus, we can see that a good choice of the block size for read calls highly effects performance.

4.1.3 Checksumming

The `Checksumming` submodule defines how checksums are handled. The settings of this submodule are applied to all checksums of the complete packet.

- `ReCalculateChecksums`: Boolean value (0|1) which specifies wether a packet's checksums are recalculated after anonymization.
- `SetBadChecksumsToBad`: Boolean value (0|1) that indicates if checksums are validated before anonymization of the original protocol. If a checksum is invalid in the original protocol, the checksum is set to a random value in the anonymous protocol. Only sensible in combination with `ReCalculateChecksums`.

If `ReCalculateChecksums` is not set, anonymous protocol checksums will be set to the original checksum, which will most likely not be correct.

4.2 AnonMappings

The `AnonMappings` module contains the mappings of anonymization primitives to protocol attributes. In this section, we explain the general structure of this module. Section 4.3.1 gives a detailed example of a configuration file containing both sections `Settings` and `AnonMappings`.

One submodule entry exists within the `AnonMappings` module for each supported protocol. The name of the submodule depends on the name of the protocol parser for this protocol. In case of the ethernet protocol, for example, the submodule is named EthernetPacket: `<submodule name="EthernetPacket">`. Each configuration item within a submodule defines a single mapping of an anonymization primitive to a protocol attribute, e. g. `<configitem anon="AnonBytewiseHashSha1" name="MacSource"/>`.

4.3 Predefined profiles

A set of predefined profiles can be found in the folder `profiles` of the PktAnon distribution. In all these profiles the configuration items `Infile` and `Outfile` are filled with a placeholder. Checksums are defined to be recalculated and to be re-set to bad. The predefined profiles contained are the following:

- `settings_identity.xml`: An anonymization profile that uses the anonymization primitive `AnonIdentity` for every protocol attribute. This means that all data of the original traffic is just copied into the anonymous traffic without changes. In this case NO anonymization is achieved. This profile could be used as a template for definition of new anonymization profiles.
- `settings_low.xml`: An anonymization profile with low security that performs anonymization of IPv4 and IPv6 addresses in IPv4, IPv6, and ARP protocols by applying the primitive `AnonBytewiseHashSha1`. In addition, it makes application data anonymous by overwriting each byte with 0x00.
- `settings_medium.xml`: An anonymization profile with medium security that makes the following protocol attributes anonymous:
  - IPv4 and IPv6 addresses in IPv4, IPv6, and ARP protocols using the primitive `AnonBytewiseHashHmacSha1` and the key `KEY`.

  - MAC addresses in Ethernet and ARP protocols using the primitive `AnonBytewiseHashSha1`.
  - IPv4 Type-of-Service attribte (ToS) is overwritten by 0x00.
  - IPv4 Time-to-live attribute (TTL) and IPv6 hop limit attribute by applying `AnonWhitenoise` with a strength of 3.
  - IPv4 and TCP options are completely removed by the primitive `AnonShorten` with a new length of 0.
  - UDP and TCP ports are made anonymous by `AnonBytewiseHashSha1`.
  - TCP urgent pointer attribute is overwritten by 0x00.
  - Application data (`PayloadPacket` is completely removed by `AnonShorten`.

- `settings_high.xml`: An anonymization profile with high security. It differs from the previous medium anonymization profile in the following tasks:
  - HMAC-based (key-based) anonymization is applied to MAC addresses as well as TCP and UDP ports instead of normal hashing.
  - IPv4 and IPv6 addresses are hashed completely by HMAC-SHA1 instead of bytewise.
  - The TCP attributes sequence number, acknowledge number, and window size are made anonymous by `AnonWhitenoise`.

WARNING:
The anonymization profiles distributed with PktAnon are for learning purposes only! Defining an anonymization profile that suits your needs most likely needs input from network operators, security experts, and lawyers!

4.3.1 Exemplary anonymization profile

In this example the anonymization profile `settings_medium.xml` will be explained. An anonymization profile is always enclosed by the `triggerconf` tag.

```
<triggerconf>
  ...
</triggerconf>
```

The first part of the profile is the `Settings` module:

```
  <module name="Settings">
    <submodule name="Flow">
      <configitem name="Input">INFILE</configitem>
      <configitem name="Output">OUTFILE</configitem>
    </submodule>
    <submodule name="Misc">
```

```
      <configitem name="UseMeasure">0</configitem>
      <configitem name="MeasureFile"></configitem>
      <configitem name="CreatePlot">0</configitem>
      <configitem name="PrintPackets">0</configitem>
      <configitem name="SingleStepping">0</configitem>
      <configitem name="BlockSizeRead"></configitem>
      <configitem name="BlockSizeWrite"></configitem>
    </submodule>
    <submodule name="Checksumming">
      <configitem name="ReCalculateChecksums">1</configitem>
      <configitem name="SetBadChecksumsToBad">1</configitem>
    </submodule>
  </module>
```

`INFILE` and `OUTFILE` are just placeholders for your traffic source and sink, respectively. Offline and online traffic can be used for input. Output also can be directed to different destinations. A trace file recorded with tcpdump, for example, can be used as traffic source. Output then can be written into a file in tcpdump format, too:

```
    <submodule name="Flow">
      <configitem name="Input">/home/guest/recordedtrace.pcap</configitem>
      <configitem name="Output">/home/guest/anonymoustrace.pcap</configitem>
    </submodule>
```

PktAnon also supports input and output piping (see Section 5.3 for detailed explanation). The configuration for input piping looks like:

```
    <submodule name="Flow">
      <configitem name="Input">stdin</configitem>
      <configitem name="Output">/home/guest/anonymizedtrace.pcap</configitem>
    </submodule>
```

In case original traffic data should be written from a file and piped to another program after anonymization, the following configuration could be used:

```
    <submodule name="Flow">
      <configitem name="Input">/home/guest/recordedtrace.pcap</configitem>
      <configitem name="Output">stdout</configitem>
    </submodule>
```

For explanation of the other configuration items in the exemplary `Settings` module refer to Section 4.1.2.

The complete mapping of anonymization primitives to protocol attributes is encapsulated in the `AnonMappings` tag:

```
  <module name="AnonMappings">
    ...
  </module>
```

Every supported protocol is defined in a submodule with the `name` item set to the name of the protocol parser, e. g.:

```
<submodule name="EthernetPacket">
  ...
</submodule>
```

Inside of such a `submodule` environment the mapping of protocol attributes to anonymization primitives is specified by configuration items. The supported protocols and their according attributes are listed in Table 3.1. Currently supported anonymization primitives are shown in Table 2.1. Thus, the mapping of the anonymization primitive `AnonBytewiseHashSha1` to the protocol attribute `MacSource` of the Ethernet protocol looks like:

```
  <configitem anon="AnonBytewiseHashSha1" name="MacSource"/>
```

The field `anon` is predefined for the anonymization primitive that should be applied. The field `name` is predefined for the protocol attribute the anonymization primitive is applied to. The complete Ethernet anonymization part in our exemplary profile looks as follows:

```
<submodule name="EthernetPacket">
  <configitem anon="AnonBytewiseHashSha1" name="MacSource"/>
  <configitem anon="AnonBytewiseHashSha1" name="MacDest"/>
  <configitem anon="AnonIdentity" name="MacType"/>
</submodule>
```

Some anonymization primitives need additional parameters, e. g. a key for HMAC-SHA1 (see Table 2.1). Using the primitive `AnonBytewiseHashHmacSha1` for anonymization of the Ethernet attribute `MacSource` could be done using the following configuration:

```
<configitem anon="AnonBytewiseHashHmacSha1" key="K3Y!" name="MacDest"/>
```

The value `K3Y!` is the key used in this HMAC-SHA1 anonymization.

5. Using PktAnon

5.1 Compiling PktAnon

Compiling and (optionally) installing PktAnon is done using the autotools build environment:

1. Open a console
2. Create a new folder
   `mkdir pktanon`
   and download PktAnon into this folder
   `cd pktanon`
   `wget http://www.tm.uka.de/pktanon/download/pktanon-1.2.0-dev.tar.gz`
3. Unzip the tar.gz archive
   `tar xzf ./pktanon-1.2.0-dev-tar.gz`
4. Change into the new folder created by unzipping
   `cd ./pktanon-1.2.0-dev`
5. PktAnon depends on Boost and Xercesc which must be installed on your system. If you are working on a debian-based system, you can install these libraries by typing
   `sudo apt-get install libxerces27-dev libboost-dev`
6. Prepare compiling of PktAnon
   `./configure`
7. Compile PktAnon
   `make`
8. To install PktAnon on your system type
   `sudo make install`

5.2 Running PktAnon

Usage of PktAnon is quite simple. The `pktanon` binary needs a single parameter – the file name of the configuration profile. This file specifies all necessary information like traffic source or anonymization mappings. If the anonymization profile `settings_high.xml`, which was previously described, should be applied, PktAnon is started by

```
./pktanon profiles/settings_high.xml
```

5.3 I/O Piping with PktAnon

PktAnon supports piping of input traffic into PktAnon. This is necessary in order to achieve online anonymization, i. e. ,live traffic is captured with *tcpdump* and directly piped into PktAnon. In addition, piping of input traffic can be used to filter traffic using *tcpdump* parameters. In order to use piping, the configuration item `Input` of the configuration file's `Flow` module must be set to `stdin` (see Section 4.1.1). The following command, then, will capture live traffic from network interface `eth0` and pipe it into PktAnon for anonymization:

```
tcpdump -i eth0 -s 0 -w - | pktanon ./settings.xml
```

The output destination of the data made anonymous is independent of the input. Therefore, this can be either a file or the standard output stream. A use case for standard output is replaying anonymous data by using the tool *tcpreplay*. In this use case traffic is made anonymous by PktAnon and then replayed using *tcpreplay*. Therefore, the configuration item `Output` in the configuration file's `Flow` module must be set to `stdout`.

The following command can be used to read input traffic from a trace file, make it anonymous, and replay it, e. g. into another that is connected to the interface `eth2`:

```
pktanon ./settings.xml | tcpreplay -i eth2
```

6. Extending PktAnon

This chapter will explain how to extend PktAnon by additional protocol parsers (see Section 6.1) and additional anonymization primitives (see Section 6.2). The design of PktAnon allows for easy integration of new primitives and protocols. If you are extending PktAnon please contact the developers in order to integrate these extensions into the PktAnon release.

6.1 Integrating additional protocols

Writing of an additional protocol parser is explained exemplarily by analyzing the UDP protocol parser of PktAnon. The following Section will guide you through the development of an additional protocol parser. Please note that PktAnon currently supports protocol parsers up to transport layer. Writing application layer parsers needs special support that will be integrated into PktAnon in the future.

6.1.1 Interfaces

The UDP protocol parser is defined in `UdpPacket.h` and implemented in `UdpPacket.cpp`. Source files of protocol parsers can be found in the folder `src/packets`. The `UdpPacket` class is derived from the virtual `Packet` base class and implements packet parsing and packet assembling mechanisms for UDP. Each new protocol class derived from the `Packet` class must implement the following methods, which are defined as virtual in the base class:

```
bool          parsePacket        ()
void          assemblePacket     ()
string        toString           ()
unsigned int  getMinProtocolSize ()
```

The `parsePacket` method is called whenever a lower layer protocol indicates UDP as subsequent protocol. In this case a byte buffer that requires parsing is given to this method. The task of the protocol parser is to gather all protocol attributes and provide them using get/set methods.

After completion of the anonymization process assembling—a task contrary to parsing—of protocol attributes is required. in order to put together a well-formed byte buffer. This buffer is written back to the output destination. Assembling must be performed in the `assemblePacket` method by the protocol parser.

PktAnon provides—for debugging and learning purposes—the possibility to print out single networks packets while operating (see Section 4.1). Therefore, each protocol parser must implement the `toString` method for providing a textual representation of the packet contents.

As a last mandatory method each protocol parser must implement the method `getMinProtocolSize`. This method has to return the constant value of the minimal size the implemented protocol requires. The minimal size of UDP is 8 bytes. The IP protocol, e. g. has a minimal size of 20 bytes but actually can be larger due to variable-length IP options.

We will now look into the implementation of this mandatory functionality, starting with the header file `UdpPacket.h` from top to bottom. At first two protocol header structs are defined. The `UDP_HEADER` struct defines the UDP header format. The struct `UDP_PSEUDO_HEADER` is the UDP pseudo-header that is used for calculation of the UDP checksum. The IP address of the lower layer protocol is normally included into the checksumming process. Because we support IPv4 and IPv6 as an underlying protocol, the second part of the pseudo-header is defined in the struct `UDP_IP4_PSEUDO_HEADER` and `UDP_IP6_PSEUDO_HEADER`, respectively. The command `#pragma pack (1)` and `#pragma pack ()` instructs the compiler not to add any alignment between variables in the struct. Usually, it is open to the compiler to perform such optimization, therefore we explicitely disallow this using the `pack` command.

Then, the `UdpPacket` class definition is done including the mandatory methods mentioned above:

```
bool          parsePacket        ()
void          assemblePacket     ()
string        toString           ()
unsigned int getMinProtocolSize ()
```

In addition, several functions for getting and setting the protocol attributes exist. These provide an easy interface for accessing protocol data:

```
unsigned short getSourceport ();
unsigned short getDestport   ();
unsigned short getLen        ();
unsigned short getChecksum   ();
void           setSourceport (unsigned short sp);
void           setDestport   (unsigned short dp);
void           setLen        (unsigned short len);
void           setChecksum   (unsigned short chksum);
```

Furthermore, two methods are declared that are needed especially in case of UDP for calculating checksums. As the UDP pseudo-header used for checksumming needs IP addresses, these two methods allow the IPv4/IPv6 parsers to explicitly set the IP address for the UDP protocol:

```
void          setIpAddresses (IP_ADDR source, IP_ADDR dest);
void          setIpAddresses (IPV6_ADDR source, IPV6_ADDR dest);
```

The following two members of type `AnonPrimitive*` represent the anonymization primitives for source port and destination port in the UDP protocol. They will be set dynamically to the anonymization primitive specified in the configuration file:

```
  static AnonPrimitive*   anonSourceport;
  static AnonPrimitive*   anonDestport;
```

Finally, private members are declared. Here, mainly the `header` member is of interest:

```
UDP_HEADER  header;
```

It represents the UDP protocol header. This member will be used to store information of the binary network data. The rest of the `.h` file contains special checksum handling.

6.1.2 Implementation

Next, we look into the file `UdpPacket.cpp` that contains the implementation of the method interfaces introduced in Section 6.1.1. The implementation will now be detailed from top of the file to the bottom.

At first you can see the initialization of the two anonymization primitives that are bound to the UDP parser in a static way:

```
AnonPrimitive* UdpPacket::anonSourceport = NULL;
AnonPrimitive* UdpPacket::anonDestport   = NULL;
```

Then, the constructor of the UDP parser initializes its header member and sets the `Packet::protocol` member to the protocol number the parser is responsible for:

```
UdpPacket::UdpPacket(void)
{
  memset (&header, 0, sizeof (UDP_HEADER));
  protocol    = Packet::PROTO_UDP;
}
```

The value `Packet::PROTO_UDP` must be defined in the file `Packet.h` to identify the protocol a parser is responsible for.

Following, we look at the `parsePacket` method. It is called by PktAnon when parsing has reached the corresponding protocol parser. The `Packet::buffer` variable in this case holds the data that should be parsed by the protocol parser. Only the data for exactly this protocol header is available within the buffer. Thus, the buffer can be copied directly into the `UDP_HEADER` struct defined in `UdpPacket.h`:

```
memcpy (&header, buffer, sizeof (UDP_HEADER));
```

Each `parsePacket` method must do the following operations:

- Extract the data from `Packet::buffer` member.
- Set the `Packet::nextProtocol` member to the next higher protocol that has been identified. The `IpPacket` parser class, for example, identifies the subsequent protocol by an attribute in its header structure. The UDP parser has no default way of identifying the following protocol. Therefore, the value `nextProtocol` is set to the value `Packet::PROTO_DATA_PAYLOAD`. This indicates that the remaining packet data is seen as pure payload data and parsed using the `PayloadPacket` parser.
- The parser has to set the `Packet::layersize` member to the number of bytes the protocol data uses. The UDP parser sets this to `sizeof(UDP_HEADER)`. IP or TCP parsers may have options that can only be identified during parsing. Therefore the `Packet::layersize` may differ from the minimal protocol header size.

Setting these three members is mandatory for protocol parsers in the `parsePacket` method. A parser should extract all data of the given buffer and provide means to easily get and set these values. This is done in the following methods of the `UdpPacket.cpp` file, namely `getSourceport`, `getDestport`, `setSourceport`, ...

The mandatory `assemblePacket` method is the counterpart to the `parsePacket` method. It is called when the output traffic is assembled and the protocol parser class is requested to provide a binary representation of the current header. Here is a partial listing of the `assemblePacket` method in case of the UDP protocol:

```
void UdpPacket::assemblePacket()
{
  if (nextPacket != NULL)
    nextPacket->assemblePacket ();

  int thissize = sizeof (UDP_HEADER) ;
  int nextsize = nextPacket != NULL ? nextPacket->getSize() : 0;

  setSize (thissize + nextsize);
```

```
  if (nextPacket != NULL)
    memcpy (buffer + thissize, nextPacket->getBuffer(), nextsize);

  setLen (thissize + nextsize);

  // MISSING LISTING: calculate new checksum

  memcpy (buffer, &header, sizeof (UDP_HEADER));

  // ...
}
```

At first, the assembling of the higher layer protocol is requested. As the size of the higher layer protocol may have changed due to anonymization, the size for the UDP length attribute is recalculated and set in the header. Finally, the data of higher layer protocols and the current header are copied into the given buffer.

6.1.3 Integration

6.1.3.1 Packet factory

PktAnon uses a memory object pool for protocol objects to boost performance. Therefore we now integrate the UDP parser into the packet factory. This is done by setting an `#include` in the file `PacketFactory.h` as follows:

```
  #include "UdpPacket.h"
```

Next we add an object pool in the same file:

```
  boost::object_pool <UdpPacket> poolUdpPacket;
```

This pool will be used in the `createPacket` method of the `PacketFactory`:

```
  case Packet::PROTO_UDP:
    ret = poolUdpPacket.construct ();
    break;
```

and in the `freePacket` method:

```
  case Packet::PROTO_UDP:
    poolUdpPacket.destroy((UdpPacket*)packet);
    break;
```

6.1.3.2 Packet transformation

We now integrate the UDP protocol parser into the actual anonymization process that transforms the original packet into an anonymous one. This is implemented in the files `Transformer.cpp` and `Transformer.h`. First, we have a look at the header file. At top of `Transformer.h` we add the include directive for the UDP protocol parser: `#include "packets/UdpPacket"`. Then, the `transformPacket` method is overloaded in regard to the new class `UdpPacket` as follows:

```
void transformPacket (UdpPacket& inpkt, UdpPacket& ret);
```

This method is called in case an anonymization of the UDP protocol is required. In the file `Transformer.cpp` we add the following code to the `getTransformedPacket` method:

```
case Packet::PROTO_UDP:
  transformPacket ((UdpPacket&)inpacket, (UdpPacket&)*newpkt);
  break;
```

This will trigger the aforementioned `transformPacket` method, which is given two parameters: the original packet and an empty packet. The actual transformation method looks like:

```
void Transformer::transformPacket (UdpPacket& inpkt, UdpPacket& ret) {
  ret.setLen (inpkt.getLen());
  ret.setChecksum (inpkt.getChecksum());

  unsigned short destport  = inpkt.getDestport ();
  UdpPacket::anonDestport->anonimizeBuffer (&destport,
          sizeof (unsigned short));
  ret.setDestport (destport);

  unsigned short sourceport = inpkt.getSourceport ();
  UdpPacket::anonSourceport->anonimizeBuffer (&sourceport,
          sizeof (unsigned short));
  ret.setSourceport (sourceport);
}
```

Purpose of this method is to get the values from the original packet `inpkt`, make them anonymous, and set these anonymous values in the new `ret` packet. This method does not explicitly need to know which anonymization primitives are applied to the protocol attributes, e. g. to `UdpPacket::anonDestport`, in order to perform the anonymization. Therefore, the code is simple and transparent for the actual anonymization primitive.

As you can see, no values are deleted from the original packet. PktAnon applies a mechanisms called *defensive transformation*, which first creates a new packet and then fills this empty packet with anonymous values.

6.1.3.3 Anonymization primitive configuration

Anonymization primitives have to be attached to protocol attributes in order to start the anonymization process. This is achieved in the file `Configuration.cpp`. On top of the file the UDP parser definition: `#include "packets/UdpPacket.h"` must be included. In addition, the name of the UDP configuration submodule in the XML-based configuration (see Section 4.2) has to be given. We named it simply "'UdpPacket"':

```
const string Configuration::ANON_SUBMODULE_UDP = "UdpPacket";
```

This constant name must also be defined in the file `Configuration.h`:

```
static const string ANON_SUBMODULE_UDP;
```

Finally, the anonymization primitives are bound to the UDP parser in the method `bindAnonPrimitives`:

```
if (tconf.existsSubmodule (ANON_MODULE, ANON_SUBMODULE_UDP)) {

  UdpPacket::anonSourceport= factory.create   (
                       tconf.getConfigItemAttributes(
                            ANON_MODULE,
                            ANON_SUBMODULE_UDP,
                            "UdpSourceport" ));

  UdpPacket::anonDestport = factory.create (
                       tconf.getConfigItemAttributes(
                            ANON_MODULE,
                            ANON_SUBMODULE_UDP,
                            "UdpDestport" ));

} else {
  cout << "no udp configuration found" << std::endl;
  exit (0);
}
```

Here the names of the anonymization primitives for "'UdpSourceport"' and "'UdpDestport"' are read from the configuration file. The anonymization primitive objects are created by the factory and attached to the UDP protocol parser.

6.2 Integrating additional anonymization primitives

In this section we explain how to write an additional anonymization primitive by exemplarily analyzing the anonymization primitive `AnonShuffle` of PktAnon. This primitive takes the input data and randomly shuffles all the bytes the input contains.

6.2.1 Writing the primitive

The anonymization primitive `AnonShuffle` is defined in `anonprimitives/AnonShuffle.h` and implemented in `anonprimitives/AnonShuffle.cpp`. Each anonymization primitive is derived of the class `AnonPrimitive` and has to implement the following method:

```
ANON_RESULT anonymize (void* buf, unsigned int len);
```

This method is called in order to actually perform anonymization. The result type `ANON_RESULT` contains two values:

- `ANON_RESULT::cont`: Continue with anonymization in case of chained anonymization primitives (see Section 2).
- `ANON_RESULT::newlength`: New length of the data buffer in case the anonymization process changed the original length (like `AnonShorten`, see Section 2).

The implementation of the `AnonShuffle` primitive in the file `AnonShuffle.cpp` looks as follows:

```
AnonPrimitive::ANON_RESULT AnonShuffle::anonymize (void* buf,
  unsigned int len) {
    vector<unsigned char> permvector;

    for (unsigned int i=0; i<len; i++)
        permvector.push_back (*(((unsigned char*) buf) + i));

    random_shuffle (permvector.begin (), permvector.end ());

    for (unsigned int i=0; i<len; i++)
        memset ((unsigned char*) buf + i, permvector.at (i), 1);

    return ANON_RESULT (len);
}
```

The method `anonymize` takes the data buffer that should be made anonymous. Then, random permutations of the single bytes are performed and the result is written back into the buffer. The return value, finally, contains the original length since it was not change during anonymization. In addition, the return value indicates that the anonymization should be continued in case of chained primitives (this constructor of `ANON_RESULT` implicitly sets `cont` to true).

6.2.2 Integrating the primitive

Having implemented the functionality of the anonymization primitive, it must be integrated into PktAnon. Therefore, the include statement `#include "AnonShuffle.h"` must be added to the file `AnonFactory.cpp`. In addition, the `AnonFactory::create` method in the file `AnonFactory.cpp` is extended by the following lines:

```
else if (name.compare ("AnonShuffle") == 0) {
  primitive = new AnonShuffle ();
}
```

These lines are necessary to actually create the anonymization primitive in PktAnon. Afterwards, it can be used in the XML configuration file for anonymization of certain protocol attributes.