Introducing the gMix Open Source Framework for Mix Implementations

Karl-Peter Fuchs, Dominik Herrmann, and Hannes Federrath

University of Hamburg, Computer Science Department, Germany

Abstract. In this paper we introduce the open source software framework gMix which aims to simplify the implementation and evaluation of mix-based systems. gMix is targeted at researchers who want to evaluate new ideas and developers interested in building practical mix systems. The framework consists of a generic architecture structured in logical layers with a clear separation of concerns. Implementations of mix variants and supportive components are organized as plug-ins that can easily be exchanged and extended. We provide reference implementations for several well-known mix concepts.

1 Introduction

Mix networks are well-known privacy-enhancing technologies that provide anonymous communication. The basic principle of *mixes* was suggested by David Chaum in 1981 [5]. Since then, a large number of concepts and strategies has been proposed. Application areas include e-mail [5,6], voting [5,28,31], location-based services [19] as well as low-latency communication (e. g., for TCP, HTTP [2,16], DNS [18] and ISDN [29]). So far, the only practically deployed systems are Mixmaster [6] and Mixminion [9] (anonymous transport of electronic mails) and the general-purpose anonymization services Tor [16], JAP (JonDonym) [2] and I2P.[1] The source code of these systems has reached a rather high complexity due to continuous security and performance optimizations, though: for instance, Tor consists of more than 63,000 lines of ANSI-C code. Therefore, it becomes increasingly difficult to understand these systems or to extend them with novel proposals from the research community. Moreover, there is a large body of scientific work without a publicly available or practically usable implementation, e. g., [8,11,13,14,22,23,29,32,33,37].

This situation has three undesirable consequences. First of all, there are considerable efforts involved in implementing a newly proposed scheme for evaluation or production purposes, because most of the time researchers will have to re-invent the wheel, i. e., find solutions for common challenges typically encountered in mix-based systems. Secondly, without an easily accessible implementation it is impossible to repeat and reproduce previous experiments. Thirdly, even if implementations *are* available, it is still difficult to compare the results from

[1] Downloads at `sourceforge.net/projects/mixmaster`, `mixminion.net`, `www.torproject.org`, `anon.inf.tu-dresden.de` and `www.i2p2.de`.

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 487–504, 2012.

one's own experiments with previous work, because of different implementations, runtime environments or missing details regarding the experimental setup. Repeatability, reproducibility and rigor in experimental research are critical for quality research, though.

With the *gMix* project we want to improve the current situation. We believe that the availability of a software framework can serve as an *enabler* here. In fields like cryptography or machine learning, frameworks such as *BouncyCastle* and *Weka* have greatly simplified access to a wide selection of implementations and led to widespread adoption.[2] To the best of our knowledge, in the domain of privacy-enhancing technologies such a software framework does not exist so far. The goals of the *gMix* project are as follows:

1. to provide a repository with compatible, adaptable mix implementations,
2. to simplify development of novel, practically usable mix-based systems and
3. to simplify evaluation of mix systems in a controlled and realistic setting.

Our Contribution. To address the aforementioned objectives we have designed an open and generic architecture for existing and future mixing schemes and built a Java framework with a plug-in mechanism. Our solution embraces the *separation of concerns* paradigm and allows a developer to build a concrete mix from the existing implementations of individual software components with no or only little changes to the code. We have already built reference implementations for various existing mixing schemes to provide a working foundation. At the time of this writing there are implementations for over ten output strategies, four recoding schemes and several auxiliary components (cf. Sect. 5). Furthermore, *gMix* includes a load generator and tools for recording results to simplify experimental evaluation. All components use configuration files, which may improve repeatability of experiments, if these files are published together with a paper. The *gMix* project is hosted at `https://www.informatik.uni-hamburg.de/SVS/gmix/`. Source code is released under GPLv3.

This paper is structured as follows: In Sect. 2 we start out with a high-level overview of the framework. We proceed by discussing its layered architecture (Sect. 3), the overall communication model within the architecture (Sect. 4), and by providing an overview of the currently available implementations (Sect. 5). Notes regarding plug-in compatibility are provided in Sect. 6. In Sect. 7 we present experimental tools which are used in Sect. 8 for a performance evaluation. We conclude the paper in Sect. 9.

2 Framework Overview

The fundamental design of the gMix Framework *(generic mix framework)* is inspired by the layer architecture of the *TCP/IP Model*. The main idea is to extend the *TCP/IP Model* with mix-specific layers, while preserving simple, *standard*

[2] Downloads at `www.bouncycastle.org` and `www.cs.waikato.ac.nz/ml/weka`

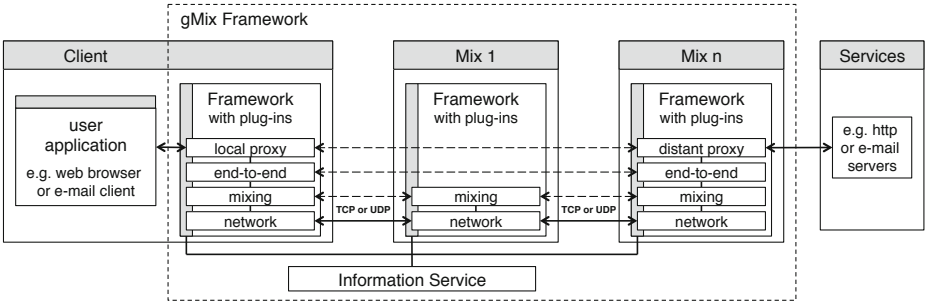

Fig. 1. Overview of the gMix architecture

access to anonymous channels through interfaces almost equal to those of *normal* TCP or UDP sockets. The concrete realization of the anonymous channels (e. g., the *output strategy* and *recoding scheme*) is highly customizable. Layers are implemented as plug-ins and can thus be easily exchanged or composed to a specific mix via configuration files. Additional *low-level* components generally needed for realizing a distributed system are provided as well.

The framework consists of components for clients, mix nodes and an *Information Service*. The *Information Service* can be used for discovery, grouping mixes into cascades and as a public board for information exchange. Clients and mixes form an *overlay network* via *normal* TCP or UDP sockets (cf. Fig. 1). Client and mix plug-ins can be run on a single host to allow peer-to-peer networks.

Anonymous channels can be established between clients and exit nodes (*Mix n* in Fig. 1) and are routed via intermediate mixes through the *overlay network* (*Mix 1* in Fig. 1). For compatibility with standard software and *normal* Internet services, proxy servers can be used as end points of the anonymous channels on both clients and exit nodes. For end-to-end anonymity, public services (e. g., a web server) may be run directly on mix nodes and privacy-preserving user applications, such as a web browser that suppresses identifying pieces of information [20], can be implemented.

The framework can be configured to provide the socket types shown in Table 1. We distinguish between *Stream* and *Datagram Sockets*. *Stream Sockets* are almost equal to *normal* TCP sockets. They feature *connect* and *disconnect* methods as well as *Input* and *Output Streams* with standard *read*, *write* and *flush* methods. *Datagram Sockets* are more flexible. A developer can configure several options. Choosing *duplex = true*, *connection-based = false*, *reliable = false* and *order-preserving = false* would result in a UDP/IP-like socket. Choosing *reliable = true* might be a good choice for e-mail mixes. Setting *connection-based = true* and *reliable = false* would be favorable for anonymizing VoIP traffic. *Connection-based* means that all messages sent through the socket will be tagged with the same random identifier (*Channel ID*). The exit node will use this identifier to map messages to the respective socket end point.

Table 1. Socket types available in the gMix architecture. The sockets are used to access the *mixing layers* from the *end-to-end layers*, i. e., they hide the anonymization process from user applications.

	StreamSocket	DatagramSocket
options	duplex	duplex, connection-based, reliable, order-preserving
implicit properties	connection-based, reliable, order-preserving	–

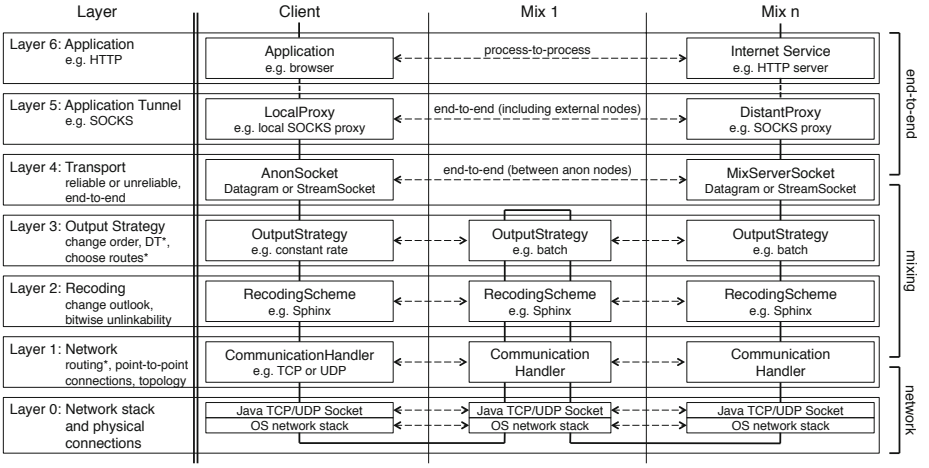

Fig. 2. Abstraction layers of the gMix framework

3 Communication Model and Layered Architecture

The abstraction layers of the gMix architecture are shown in Fig. 2 for the case of communication between a client and a server via two mixes.

Layer 0 represents the physical or logical connections between the nodes of the anonymization network (i. e., hosts running the client and mix components of the framework, *anon nodes*). In most cases, communication will be realized via TCP or UDP sockets opened by the Java Virtual Machine the framework is executed in. As the higher layers of our architecture do not require direct interaction with Layer 0 sockets, various transport protocols (e. g., streaming protocols like SCTP), Internet layer protocols (e. g., IPv6 or IPsec) and application layer protocols (e. g., TLS or DTLS) qualify for implementation.

Instead of using protocol-specific addresses directly, every mix chooses a random number (*Global Identifier*) during its initialization and publishes it via the *Information Service* along with its actual address information that may vary among plug-ins (e. g., IPv4 or IPv6 addresses and port numbers). Translating between *Global Identifiers* and actual addresses is a *Layer 1* task. Higher-order layers use the *Global Identifiers* only.

Layer 1 provides point-to-point connections between *anon nodes*. Its purpose is to hide the details of the underlying (*Layer 0*) communication channels by providing primitives to exchange messages between *anon nodes* (*hop-to-hop*). Its functionality is closely related to the *Internet Layer* of the *TCP/IP Model*, except that (end-to-end) source and destination addresses must be excluded for anonymity reasons, of course. As a result, each mix will get to know only the *next hop* of a message (addresses of further hops are hidden due to encryption).

Choosing the actual message routes is a *Layer 3* task. *Layer 1* will just forward messages *to the next hop* and is thus the *lowest* layer of our *overlay network*. We distinguish between two well-known [4,7,17] types of **routing** for mix messages: *free routes* and *fixed routes*. With *free routes*, the client chooses a series of mixes (e. g., from a list obtained from the *Information Service*) and adds their addresses to the respective header fields of the layered encryption for each mix. With *fixed routes*, mixes can be organized as cascades: All messages belonging to a *fixed route* will travel along the same path. No address information is stored in the mix messages. *Layer 3* plug-ins may choose one of the fixed routes, but not define their own. The *Information Service* can be used to establish and organize cascades.

The general purpose of **Layer 2** is to make it cryptographically difficult to link messages entering and leaving mixes, i. e., to provide bitwise unlinkability of mix messages and to pad them to equal length. This is typically realized by *recoding* (i. e., encrypting or decrypting) messages. As some recoding schemes are deterministic and are thus prone to replay attacks, while others are not, we chose to make *Layer 2* responsible for detecting replays of messages as well. If a deterministic scheme is employed, a replay detection as used in JonDonym [24] can be implemented here. The plug-ins are also responsible for publishing and retrieving information needed for recoding messages (e. g., public keys or initialization vectors) via the *Information Service*.

Layer 3 realizes another core function of a mix, the *output strategy* or *flushing algorithm*. Its purpose is to hide the true sender of a message among other senders, thus building an anonymity set. This is achieved by delaying and reordering messages. In his initial work [5], Chaum suggested to collect (or *batch*) messages until a certain threshold is reached, then putting out all messages together in lexicographic order. Since then, numerous output strategies have been proposed (e. g., [3,5,6,11,13,14,23,29,32,33,36,37]). While these output strategies are highly different in terms of delay and anonymity characteristics, from an architectural point of view, they are fairly equal as already shown in [14]. As some output strategies require clients to send in a specific fashion, not necessarily dependent on the flow characteristics of the user traffic (e. g., at constant rate), we chose to make Layer 3 responsible for initiating the creation and dropping of **dummy messages**: dummy messages contain random data instead of *normal* payload. They are put into the stream of *normal* messages to hamper traffic analysis. While dummy messages may reduce the available bandwidth, they can also reduce latency, e. g., if the output strategy requires a certain number of messages to flush. If all users send messages constantly, the sending of real messages

becomes *unobservable* [8,29]. We discuss arising dependencies between Layer 2 and 3 in Sect. 6.1.

Layer 4 is the interface between the *mixing layers* (1–3) and the *end-to-end layers* 5 and 6. It can be used to establish end-to-end anonymous channels between anon nodes with the socket primitives of Table 1. For *free routes*, a *Global Identifier* can be specified as destination address. The aforementioned *Channel IDs* are used to map different messages of a *connection-based* anonymous channel to the respective sockets. Different *Layer 5* services running on a single node are distinguished by *service port numbers* that work like *normal* port numbers in the *TCP/IP Model*.

In **Layer 5**, application-level proxies (e. g., SOCKS, HTTP, DNS, FTP, SMTP, or VoIP proxies) can be implemented to enable end-to-end communication with hosts not part of the overlay network. *Layer 5* client plug-ins will open a local proxy on the client (i. e., in the area of protection of the client) and tunnel the application level connections (e. g., SOCKS connections established by a web browser) through *Layers 4 to 0* to an exit node running a *distant proxy* (the corresponding *Layer 5* mix plug-in). The *distant proxy* will forward the data of the tunnelled connections to their respective destinations (e. g., *normal* web servers). A plug-in developer can choose whether an anonymous tunnel shall be established for each application level connection or all connections shall be multiplexed through a single anonymous tunnel.

With the socket interfaces of *Layer 4* being very equal to *normal* sockets, we expect easy adaptation of existing proxy software. Since *Layer 5* is the first layer not required to be written completely in Java, proxy services written in different languages should be integrable by implementing an adapter class in Java.

Layer 6 is the layer closest to the user and equivalent to the *application layer* of the *TCP/IP Model*. It refers to higher-level application protocols and end-to-end connections between application programs. The concrete realization of these programs and protocols is out of the scope of our model, though.

4 Inter-layer Communication

To support different implementations (i. e., plug-ins) of the abstraction layers, standard interfaces between the framework and plug-ins are needed. As a result, we had to choose a common model for communication between layers and define a uniform internal message format. We considered two common architectural designs: (1) using a single loop to iterate over all layers or (2) to use a **thread-based approach with asynchronous I/O between layers**. We chose the latter as this in our view

- simplifies the implementation of multiple threads on one layer to speed up operation (e. g., for the recoding scheme),
- allows parallel operation on different layers (e. g., decrypting messages on *Layer 2* while new messages are received via *Layer 1*),
- allows to run several distinct instances on a single layer concurrently (e. g., communication handlers for client and mix connections on *Layer 1*) and

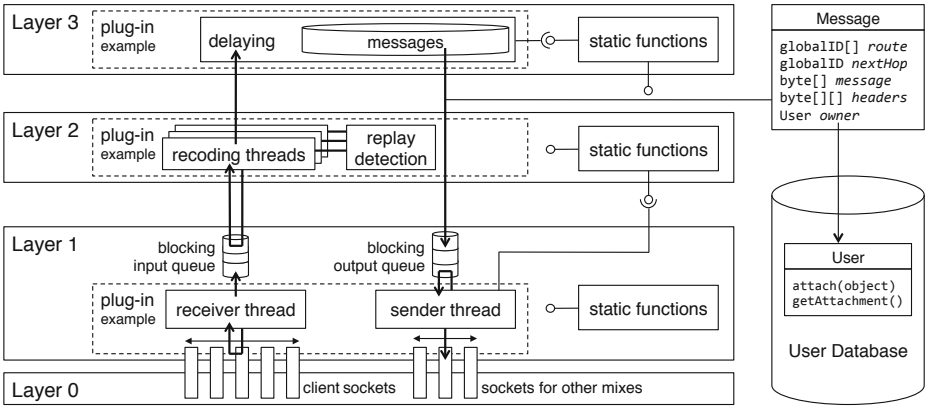

Fig. 3. Inter-layer communication patterns and separation of plug-in and framework concerns. Example of a simplex mix handling client connections.

– leads to more understandable implementations because of loose coupling and a clear separation of concerns.

For process synchronization, we use blocking queues with a wait-notify mechanism (cf. Fig. 3) between *Layer 1* and *Layer 2* (blocking conditions are *empty* and *full* for *getMessage* and *addMessage* operations, respectively). The queues are part of the framework, not the plug-ins. As a result, plug-in developers will not have to implement synchronization mechanisms themselves, unless they decide to have different threads (requiring shared resources) within their plug-ins.

For the **internal message format**, i. e., the Java objects exchanged between layers, we chose a generic solution, defining only the general purpose of a field rather than its actual content. As a result, individual contents and headers may be defined by plug-in developers for each layer. While this allows for tailored solutions, it also requires that plug-ins are developed pair-wise for clients and mixes, unless another plug-in speaking the same protocol is already implemented (we discuss dependencies between plug-ins in Sect. 6.1). In the remainder of this section we will focus on the details of the internal message format and further mechanisms included with the framework to keep state across multiple layers and assure compatibility between (plug-ins of) different layers.

As illustrated in Fig. 3, the Java objects exchanged between layers contain a byte array with the actual bit representation of the message to be eventually transmitted via *Layer 0*. Each layer is allowed to add additional headers. End-to-end headers (*Layer 4* and above) can be added directly to the *message* field. Headers that are required at each hop (i. e., in *Layer 3*) must be stored in the data structure *headers*. *Layer 2* implementations (recoding schemes) must indicate, whether they are capable of adding additional header fields for each mix (cf. Sect. 6.1). Currently, the only implementation requiring additional headers is the stop-and-go output strategy [23].

End-to-end destination addresses of *messages* are chosen on *Layer 4* and stored in the data structure *route*. In *free route* setups, *Layer 3* may add the addresses of further hops. The *route* information is stored in the layered encryption by the recoding scheme (*Layer 2*). Due to the encryption of the *message* on *Layer 2*, the cleartext field *nextHop* is needed for routing purposes on *Layer 1*. This field can also be re-set at each hop to allow for adaptive or random routing.

For keeping state, the component *User Database* is available. It stores a data structure (*user*) for each connected *client*. For each message, a reference to the respective user object is set on *Layer 1*, which allows for immediate access without look-up delays on all layers. Plug-ins can *attach* individual objects to each *user*. The *Java generics* mechanism is used to assure compile-time type safety.

Layers may offer *static functions* to adjacent layers. For example *recoding scheme* plug-ins may offer an interface to create dummy messages for *output strategy* plug-ins. Classes of general use for different plug-ins of the same layer can be offered as *static functions* as well, e. g., a class for *replay detection*.

5 Status of Development and Available Implementations

Started in 2011, the gMix project is still under heavy development. While individual implementations are quite basic, others have already reached practicable quality. The framework can load individual plug-in combinations specified in a config file. The *Information Service* can be used to organize mixes in cascades via network (for real deployment) or on a single workstation (for testing, measurement and teaching) without having to deal with individual IP addresses or port numbers. A load generator can be used to evaluate components and test implementations (cf. Sect. 7). A PKI is not included yet, but will be added soon. Framework and plug-ins currently consist of more than 16,000 SLOC in total. At present, the mix plug-ins listed in Table 2 are available.

On *Layer 3*, we have implemented the output strategies described in [5,6,14,23,33] and [37]. On the client side (not included in Table 2), we offer implementations to send at *constant rate*, send requests and receive replies *alternately* and to send data *immediately* on request of *Layer 4* (e. g., for datagram services). Another implementation mimics the general behavior of TCP/IP sockets by waiting a configurable amount of time for packets to be filled before forwarding them.

Currently we offer four *Layer 2* implementations: Two plug-ins (*RSA_AES_Channel* and *RSA_AES_LossTolerantChannel*) are supposed to be used for low-latency mix systems and streaming data. Both use RSA (in OAEP mode with configurable key size) to establish anonymous channels. Data is sent in cells of configurable size, each layer encrypted with AES. The *RSA_AES_Channel* scheme uses OFB and is *order-preserving* (cf. [38]). The *RSA_AES_LossTolerantChannel* employs AES in CBC mode with explicit initialization vectors (IV). With this mode and IVs prepended to each encryption layer, each cell can be decrypted separately, i. e., lost cells can be tolerated (the same mechanism is used in DTLS). We use HMAC-SHA256 for

Table 2. Currently available plug-ins and their *general* capabilities (**D**uplex, **R**eliable**, C**onnection-**B**ased and **O**rder-**P**reserving, cf. Table 1 and Sect. 6.1). *True*, *false* and *any* values should be seen as properties of our implementations, not as general properties of the concepts as some may be implemented differently.

MIX PLUG-INS	D	R	CB	OP
Layer 3				
BinomialPool [14,32], BasicBatch [5], BasicPool [6], ThresholdPool [33], TimedBatch [33], CottrellPool [6], ThresholdAndTimedBatch [33], ThresholdOrTimedBatch [33], BatchWithTimeout [33], TimedDynamicPool [6], CottrellRandomDelay [6], CottrellTimedPool [6]	any	true	false	false
StopAndGo [23]	any	false	false	false
SynchronousBatch (simplified version of [29]), DLPA [37]	any	true	true	true
NoDelay (will forward data immediately as in [2,16])	any	true	any	true
Layer 2				
Sphinx [10] (*SPHINX* in Sect. 8)	false	true	false	false
RSA_OAEP_AES_OFB (*RSA-OFB* in Sect. 8)	false	true	false	false
RSA_AES_Channel (*SYM-CH* in Sect. 8)	any	true	true	true
RSA_AES_LossTolerantChannel (*LT-CH* in Sect. 8)	false	true	true	false
Layer 1				
Mix-Client TCP FCFS Sync. I/O, Mix-Client TCP Round-robin Sync. I/O, Mix-Client TCP FCFS Async. I/O, Mix-Mix TCP Multiplexed Sync. I/O	any	true	any	true
Mix-Client UDP FCFS Async. I/O, Mix-Mix UDP Async. I/O	false	false	any	false

message integrity in both cases. The two remaining plug-ins are *Sphinx* and *RSA_OAEP_AES_OFB*. The very compact *Sphinx* scheme [10] is optimized for services with typically short messages (e. g., electronic mail or micro-blogging).[3] The *RSA_OAEP_AES_OFB* plug-in is pretty close to the original suggestion of David Chaum [5], except that we use a hybrid scheme with RSA in OAEP and AES in OFB mode.

On *Layer 1*, we have implemented several mix plug-ins to handle client connections via different protocols (TCP, UDP), with varying scheduling mechanism (first-come first-served, round-robin) and with diverse I/O models (asynchronous and synchronous I/O). For connections between mixes, a plug-in capable of multiplexing messages of different clients through a single TCP connection is available (*Mix-Mix TCP Multiplexed Sync. I/O*). Using UDP between mixes is possible as well (*Mix-Mix UDP Async. I/O*).

6 Mix Composition and Compatibility

The layer concept of the *gMix* architecture provides a highly structured and in our opinion easily comprehensible view of a mix. Nevertheless, it also introduces

[3] Our implementation is a Java port of the Python implementation provided at `crysp.uwaterloo.ca/software/` using Curve25519 ECDH.

additional complexity, because the developer is faced with the decision to select adequate implementations on each layer.

Currently, we require the developer to choose a reasonable plug-in composition, or use predefined configurations included with the framework. As we expect the target audience of *gMix* to consist of researchers and developers familiar with mix systems, we do not consider this to be an issue for now. Nevertheless, simplifying the composition of plug-ins and investigating and documenting the dependencies between different mix concepts is certainly a desirable goal.

In Sect. 6.1 we discuss some important dependencies and their implications for plug-in development. Afterwards we present a rather basic, but extendable matching mechanism for capabilities and requirements that we plan to include in a future version of the framework in Sect. 6.2.

6.1 Dependencies and Implications for Plug-in Development

During plug-in development we found that most dependencies between implementations are closely related to the socket options of Layer 4 (cf. Table 1). Given the strict interfaces between the framework layers and taking those dependencies into account, many plug-ins of different layers are compatible without further efforts. We will therefore illustrate these dependencies along with the basic **capabilities** of our current plug-ins first, before we discuss **dummy traffic** and highlight **design choices for plug-in development**.

The **capabilities** of the current plug-ins for Layer 4 socket options are displayed in Table 2 (***D**uplex*, ***C**onnection-**B**ased*, ***R**eliable* and ***O**rder-**P**reserving*). While **true** and **false** indicate whether a plug-in has a certain capability or not, **any** means that a plug-in is *adaptive*, i. e., it can be configured to offer the respective capability (for example, a Layer 1 plug-in using UDP can introduce sequence numbers for packets in order to support *order-preserving* transfer).

The **duplex** capability specifies whether or not a plug-in distinguishes between request and reply messages. On Layer 1, *duplex* simply means that plug-ins can receive as well as send messages. The Layer 2 plug-in *Sphinx* does not make a difference between requests and replies, i. e., this protocol does not exhibit the duplex property according to our definition. On Layer 3, plug-ins that collect both requests and replies within a joint message pool (i. e., they are part of a common anonymity set) are defined to be simplex. To support duplex sockets on Layer 4, all lower-layer plug-ins must support *duplex* as well. If that is not the case, two simplex sockets can be established on Layer 4 to offer end-to-end duplex connections (and more secure simplex plug-ins like *Sphinx* can be used).

As stated before, the **connection-based** attribute is used to describe plug-ins that allow the linking of packets that belong to the same anonymous channel. For instance, the Layer 3 plug-in *SynchronousBatch* will collect a message for each channel before output and must therefore know which messages belong to which channel. The same applies for the Layer 2 plug-in *RSA_AES_Channel*, as it is required to use the same cipher instance for each message of a channel. To support *connection-based* Layer 2 and Layer 3 plug-ins, Layer 1 plug-ins are required to *tag* individual packets of one connection with the same random

identifier. The identifier must of course be changed from mix to mix and be deactivated for non-connection-based sockets for security reasons.

The attribute **order-preserving** is only of relevance for *connection-based* sockets. For instance, using the *RSA_AES_Channel* plug-in will require that Layer 3 and Layer 1 plug-ins do not change the order of messages belonging to one channel as ciphers on client and mixes must stay in sync.

Dummy Traffic. In addition to the dependencies discussed above and described in Sect. 4 (concerning the routing mechanisms *free* and *fixed routes*), another dependency arises when combining *connection-based* sockets with non-end-to-end **dummy traffic**, i. e., when mixes are supposed to generate dummy messages for a certain channel (as for example with the *DLPA* plug-in [37]). In case of *stateful* recoding schemes (the decryption of subsequent messages depends on the decryption results of previous messages), the recoding component of a mix will not be able to generate an indistinguishable dummy for successive mixes due to the state of the client cipher being secret.

End-to-end dummy traffic (Layer 4) does not introduce additional dependencies, as Layer 2 plug-ins will have to add and remove padding for payloads smaller than designated anyway. An end-to-end dummy message can be considered a normal mix message with a payload field containing padding only. Dummy traffic introduced by mixes in case of *stateless* recoding schemes is straightforward as well since mixes can use client-side plug-ins to generate messages, too.

Design Choices for Plug-in Development. Whether plug-ins are compatible or not is not always an inherent property of the mix concept implemented, but often a design choice of the developer. It is an important decision, whether to strive for adaptivity (plug-ins will be more complex and difficult to understand and modify) or simplicity (individual plug-ins will be less complex, but the number of plug-ins will increase and code redundancy across plug-ins may become an issue). While the final choice will always be made by the implementer of a plug-in, we suggest to strive for simple plug-ins. So far we chose to implement adaptive plug-ins in two cases only: When a plug-in is expected to need a certain requirement for most use cases and *making it adaptive* basically means that some part of its functionality can be *turned off* (e. g., a Layer 1 TCP connection handler that can be configured to only read but not write data), or when making a plug-in adaptive can be achieved by adding only a few lines of code.

6.2 Matching Mechanism for Capabilities and Requirements

To simplify the composition of compatible plug-ins to individual mixes and to document the capabilities, requirements and dependencies of plug-ins, we plan to include a rather simple but extendable *matching mechanism* in the future. The basic idea of our proposal is inspired by the capability mechanism used in *Weka*, a framework for machine learning algorithms. In our case, requirements arising from rather *general* design choices (e. g., which topology shall be used or which

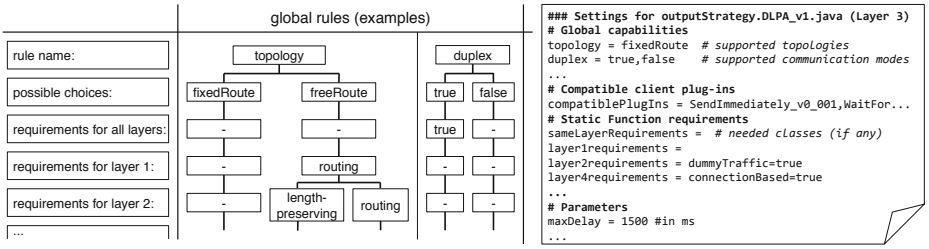

Fig. 4. Modeling dependencies as *global rules*, *requirements* and *capabilities*

socket type shall be available on Layer 4) could be stored as a set of *global rules*, i. e., *requirements* could be specified for each layer (cf. left side of Fig. 4: choosing the *free route topology* would for example require a *length-preserving* recoding scheme on Layer 2 that is able to include *routing* information within the message headers). If developers specify the *global capabilities* (cf. right side of Fig. 4) of their plug-ins, i. e., which of the *requirements* defined in the *global rules* they fulfill, an automated matching is possible. As a result, invalid compositions can be detected, or suitable plug-ins suggested. Adding further security or quality of service attributes to plug-in descriptions might be an option as well.

As described before, plug-ins must be implemented pair-wise for clients and mixes, as compatibility cannot be assumed. Nevertheless, some plug-ins may *be* compatible. For those, we suggest a *white list* (parameter *compatiblePlugIns* in Fig. 4): If a compatible plug-in is specified, the requirement to implement mix or client counterparts can be relaxed. Requirements for classes of general use for different plug-ins (the *static functions* described in Sect. 4) should be specified by plug-in developers as well (parameters *sameLayerRequirements* and *layerXrequirements* for *static functions* required on the same or on another layer).

We believe that the *matching mechanism* outlined above can serve as a useful tool for modeling and verifying dependencies and will help developers to get a better understanding of design options for individual plug-ins.

7 Experimentation Tools

Evaluating mix systems in terms of performance is a challenging task. Common methods include *manual mathematical analysis* (e. g., based on queueing theory), *discrete-event network simulation* (a simulation program is used to model the behavior of network nodes and communication lines on a single workstation), *network emulation* (a *real* local area network is used; traffic is routed through an emulation workstation that alters packet flow according to the characteristics of the network situation of interest) and *evaluation in real world settings* (like the Internet) or within *global research networks* (e. g., PlanetLab).

While each evaluation method has advantages and drawbacks, in our view *network emulation* fits best with our goal of *evaluating existing and new mix techniques in a controlled and realistic setting* (cf. Sect. 1). It allows to use

physical network nodes running a *full* mix implementation instead of relying on simplified models of mix node behavior. An experimenter can specify various network attributes (e. g., bandwidth, round-trip-time, jitter, packet loss, packet duplication or packet reordering) and evaluate their influence on the overall performance without the need to distribute network nodes across the Internet.

The need for empirical evaluation tools is increasingly recognized in the privacy community. Recently, two promising approaches have been suggested for Tor: ExperimenTor [1] and Shadow [21]. Both of them address deployment and automated testing. On the other hand, they are tightly integrated with Tor and therefore difficult to extend or adapt to other applications and mixing concepts. The main advantage of our framework in this respect is the high number of different plug-ins available for comparison and the possibility of extension with new proposals. Reproducibility of results is simplified as the source code of *gMix* has been published and configuration files of individual experiments (containing plug-in names, version numbers and parameters) can be released together with a scientific publication (e. g., in the appendix or on a website).

To simplify testing we have included a *load generator* that automatically instantiates several clients on a single workstation and makes them send messages according to commonly used statistical models (e. g., according to a poisson process or at a uniform rate). To support more realistic evaluations we plan to add more advanced statistical traffic models as well as extend the *load generator* to replay traffic according to log files recorded in real-world settings.

8 Performance Evaluation

Given the limited space and the high number of possible plug-in-combinations, parameters and test scenarios, we have to focus on a small subset of evaluations for this publication. The basic goal of this section is to assess the performance of the framework and to demonstrate that Java and our architectural design offer adequate performance to build practical mix systems rather than evaluating the effects of different output or dummy policies. To this end, we focus on an evaluation of the recoding scheme plug-ins (which introduce the highest computational cost of all mix components) and evaluate the influence of parallelization, as the framework is optimized to take advantage of multi core systems. To determine the throughput limits we performed several tests in a controlled lab environment with 1 Gbit/s Ethernet (*Setup 1: Lab Environment*). In a second test scenario (*Setup 2: Emulated Environment*), we add a network emulator to reproduce one of the findings of [30], i. e., the negative influence of packet loss when messages of different users are multiplexed over a single TCP connection. Configuration files used for the experiments can be downloaded from the project website (`https://www.informatik.uni-hamburg.de/SVS/gmix/`).

8.1 Test Parameters

All experiments have been carried out using multiple, identically configured off-the-shelf desktop machines (Intel i5-2400 3.1GHz quad core CPUs, 8 GB RAM).

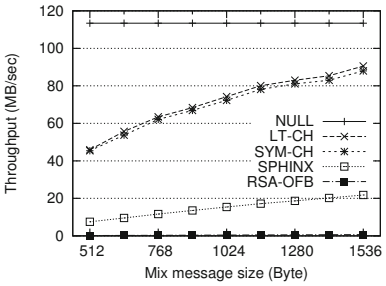

Fig. 5. Throughput for various message sizes and recoding schemes

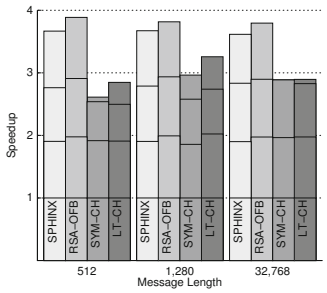

Fig. 6. Speedup gained by using up to 4 recoding threads

The software environment consisted of CentOS 6 Linux running the OpenJDK v1.6.0_22 64 bit ServerVM and Linux Kernel v2.6.32. The MTU value (maximum transmission unit) within the network was 1500 bytes. Motivated by practical anonymity systems, we use 128 bit AES keys (Tor and JonDonym) and 2048 bit RSA keys. For *Sphinx* we use Dan Bernstein's Curve25519 as suggested in [10].

8.2 Lab Environment

We start out by comparing the achievable throughput for all recoding schemes of Table 2. The throughput refers to the payload only (excluding the overhead for mix message headers). We deploy load generators on 4 workstations to simulate a total of 32 clients that send mix packets at maximum rate to a single mix via the Layer 1 plug-in *Mix-Client TCP Round-robin Sync. I/O* in simplex mode.

Figure 5 shows the throughput for various mix message lengths below the MTU (1500 byte). Without cryptographic operations the mix achieves a throughput of 116.1 MB/s (*NULL cipher*). The two *channel schemes SYM-CH* and *LT-CH* (cells sent through anonymous channels are encrypted symmetrically only) allow for up to 93 MB/s, the two schemes using a hybrid cryptosystem for each message (*SPHINX* and *RSA-OFB*) for up to 22.3 MB/s and 0.7 MB/s, respectively. Sphinx seems to be fast enough to saturate a 100 Mbit/s communication line despite its hybrid cryptosystem. As expected, the aggregated throughput increases with the message size for all plug-ins, as the constant overhead per message for headers and switching between ciphers becomes less relevant.

Message dwell times, i. e., the time messages are delayed in the mix (measured on Layer 1 with 512 byte message size) are below 1.2 ms (*SYM-CH*), 0.7 ms (*LT-CH*), 1.2 ms (*SPHINX*) and 44.7 ms (*RSA-OFB*) for 95 % of messages.

Figure 6 shows the *speedup*, i. e., to what extent the recoding scheme plug-ins benefit from the availability of multiple CPU cores. For instance, for a message length of 512 bytes the throughput for Sphinx increases from 2.1 MB/s (which is equivalent to a speedup of 1 for this scheme) to 4 MB/s (which is 1.9×2.1 MB/s) if two threads are used. The two schemes using a hybrid cryptosystem benefit most from multi-threading and scale almost linearly. On the other hand the

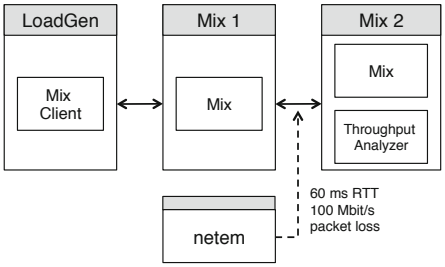

Fig. 7. Setup for emulated environment (cf. Sect. 8.3)

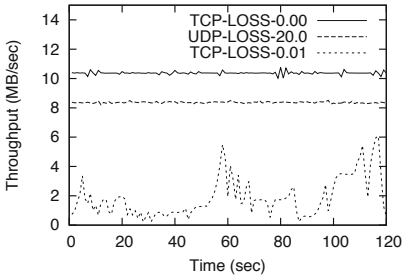

Fig. 8. Effect of packet loss on multiplexed TCP connections

channel schemes profit less, as (depending on message size) two threads can be enough to saturate the network.

We repeated our experiments using a single CPU core (with the Linux kernel directive *maxcpus=1*) and found that the resulting throughput is still and constantly well above 14 MB/s (*SYM-CH*), i. e., switching between Layer 1 and Layer 2 threads (which is under control of the JVM) does not lead to fluctuating throughput for communication links slower than 100 Mbit/s in our setup.

As a reference point for the perceived results, we measured throughput for a Tor node (v0.2.2.35) in our test setting as well (using *netio* via *tsocks*). With mix packets sized 512 byte (equal to Tor's cell size), a maximum throughput of 46 MB/s is possible with the *SYM-CH* plug-in (quad core), while we measured 37 MB/s for the Tor node. The comparable performance is interesting as on the one hand Tor is written in C, but on the other hand it does not support multithreading for message recoding. In this experiment our framework manages to compensate for the slower performance of Java by a better utilization of the available hardware. In the end it achieves a similar throughput as an implementation in C, at the cost of a higher number of instruction cycles (cf. the throughput of only 14 MB/s in case of *maxcpus=1*). We conclude that – despite the use of Java and its generic architecture – our framework offers adequate crypto performance for practical scenarios with up to 100 Mbit/s links. Real-world performance, i. e., when mixes are distributed over the Internet, cannot be deduced from these results, though, due to network congestion, differing bandwidth of anon nodes and predefined routes once packets have been sent (source routing) [12].

8.3 Emulated Environment

In the following experiment, we use a cascade of two mixes and shape traffic using the freely available network link emulator *Netem* [25][4] to show one of the findings of [30]: the negative influence of packet loss when messages of different

[4] A comparative study of network link emulators can be found in [27]. For even more sophisticated evaluations, virtual network emulators (e. g., Emulab [39] or Modelnet [35], like in [1]) can be employed.

users are multiplexed over a single TCP connection. RTT between mixes was set to 60 ms. Bandwidth was limited to 100 Mbit/s to assure crypto overhead is not the limiting factor. The experimental setup is shown in Fig. 7.

We use the *Mix-Mix TCP Multiplexed Sync. I/O* plug-in between mixes and configure the emulator to drop messages between the two mixes to show the effect. For comparison, we provide results for the UDP Layer 1 plug-ins as well. Figure 8 shows that even for *low* packet loss of 0.01 %, TCP throughput is highly unstable as the messages of all users are blocked by a dropped message of a single user, i. e., the operating system TCP buffer will not forward any data (possibly packets of other users) until the lost packet is retransmitted. While this effect was already shown in [30], we want to stress that we were able to reproduce this finding simply by combining existing plug-ins and without writing a single line of code.

9 Conclusion

In this paper we proposed a generic architecture for mixes and our open source implementation, the *gMix framework*. First and foremost, the gMix framework aims to provide easy access to the central components of a mix, which are structured into distinct logical layers. Secondly, we strive for easily understandable implementations to allow developers to build a fully functional mix from a set of rather simple components (plug-ins). Thirdly, the *gMix* framework aims to improve the process of the evaluation of mixes. At the moment, this objective is addressed with a load generator. Tools for test automation will be included in the future. Moreover, the consequent use of configuration files ensures repeatability of experiments and reproducibility of results.

We see our work as a first step towards the standardization of mix systems which can help lower the bar for research of mixes as well as their deployment in practice. In the long run we hope that the availability of a comprehensive software framework will serve as an enabler for mixes and will lead to the increased dissemination of privacy-enhancing technologies in existing and new application areas as well as to new proposals that can be integrated into deployed systems like Tor or JonDonym.

References

1. Bauer, K., Sherr, M., McCoy, D., Grunwald, D.: ExperimenTor: A Testbed for Safe Realistic Tor Experimentation. In: Workshop on Cyber Security Experimentation and Test (2011)
2. Berthold, O., Federrath, H., Köpsell, S.: Web MIXes: A System for Anonymous and Unobservable Internet Access. In: Federrath, H. (ed.) Anonymity 2000. LNCS, vol. 2009, pp. 115–129. Springer, Heidelberg (2001)
3. Berthold, O., Langos, H.: Dummy Traffic against Long Term Intersection Attacks. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 110–128. Springer, Heidelberg (2003)

4. Böhme, R., Danezis, G., Díaz, C., Köpsell, S., Pfitzmann, A.: On the PET Workshop Panel "Mix Cascades Versus Peer-to-Peer: Is One Concept Superior?". In: Martin, Serjantov [26], pp. 243–255
5. Chaum, D.: Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Communications of the ACM 24(2), 84–90 (1981)
6. Cottrell, L.: Mixmaster and Remailer Attacks (1995), http://www.obscura.com/~loki/remailer-essay.html
7. Danezis, G.: Mix-Networks with Restricted Routes. In: Dingledine [15], pp. 1–17
8. Danezis, G., Diaz, C., Troncoso, C., Laurie, B.: Drac: An Architecture for Anonymous Low-Volume Communications. In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 202–219. Springer, Heidelberg (2010)
9. Danezis, G., Dingledine, R., Mathewson, N.: Mixminion: Design of a Type III Anonymous Remailer Protocol. In: IEEE Symposium on Security and Privacy, pp. 2–15. IEEE Computer Society (2003)
10. Danezis, G., Goldberg, I.: Sphinx: A Compact and Provably Secure Mix Format. In: IEEE Symposium on Security and Privacy, pp. 269–282. IEEE Computer Society (2009)
11. Danezis, G., Sassaman, L.: Heartbeat Traffic to Counter (n-1) Attacks: Red-Green-Black Mixes. In: Jajodia, S., Samarati, P., Syverson, P.F. (eds.) WPES, pp. 89–93. ACM (2003)
12. Dhungel, P., Steiner, M., Rimac, I., Hilt, V., Ross, K.W.: Waiting for Anonymity: Understanding Delays in the Tor Overlay. In: Peer-to-Peer Computing, pp. 1–4. IEEE (2010)
13. Díaz, C., Preneel, B.: Taxonomy of Mixes and Dummy Traffic. In: Deswarte, Y., Cuppens, F., Jajodia, S., Wang, L. (eds.) International Information Security Workshops, pp. 215–230. Kluwer (2004)
14. Díaz, C., Serjantov, A.: Generalising Mixes. In: Dingledine [15], pp. 18–31
15. Dingledine, R. (ed.): PET 2003. LNCS, vol. 2760. Springer, Heidelberg (2003)
16. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The Second-Generation Onion Router. In: 13th USENIX Security Symposium, pp. 303–320 (2004)
17. Dingledine, R., Shmatikov, V., Syverson, P.F.: Synchronous Batching: From Cascades to Free Routes. In: Martin, Serjantov [26], pp. 186–206
18. Federrath, H., Fuchs, K.P., Herrmann, D., Piosecny, C.: Privacy-Preserving DNS: Analysis of Broadcast, Range Queries and Mix-Based Protection Methods. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 665–683. Springer, Heidelberg (2011)
19. Federrath, H., Jerichow, A., Pfitzmann, A.: MIXes in Mobile Communication Systems: Location Management with Privacy. In: Anderson, R.J. (ed.) IH 1996. LNCS, vol. 1174, pp. 121–135. Springer, Heidelberg (1996)
20. Huber, M., Mulazzani, M., Weippl, E.: Tor HTTP Usage and Information Leakage. In: De Decker, B., Schaumüller-Bichl, I. (eds.) CMS 2010. LNCS, vol. 6109, pp. 245–255. Springer, Heidelberg (2010)
21. Jansen, R., Hopper, N.: Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In: Proceedings of the Network and Distributed System Security Symposium. Internet Society (2012)
22. Kate, A., Goldberg, I.: Using Sphinx to Improve Onion Routing Circuit Construction. In: Sion [34], pp. 359–366.
23. Kesdogan, D., Egner, J., Büschkes, R.: Stop-and-Go-MIXes Providing Probabilistic Anonymity in an Open System. In: Aucsmith, D. (ed.) IH 1998. LNCS, vol. 1525, pp. 83–98. Springer, Heidelberg (1998)

24. Köpsell, S.: Vergleich der Verfahren zur Verhinderung von Replay-Angriffen der Anonymisierungsdienste AN.ON und Tor. In: Dittmann, J. (ed.) Sicherheit 2006. LNI, vol. 77, pp. 183–187. GI (2006)
25. Linux Foundation: Netem (2009), http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
26. Martin, D., Serjantov, A. (eds.): PET 2004. LNCS, vol. 3424. Springer, Heidelberg (2005)
27. Nussbaum, L., Richard, O.: A Comparative Study of Network Link Emulators. In: Wainer, G.A., Shaffer, C.A., McGraw, R.M., Chinni, M.J. (eds.) SpringSim. SCS/ACM (2009)
28. Park, C., Itoh, K., Kurosawa, K.: Efficient Anonymous Channel and All/Nothing Election Scheme. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 248–259. Springer, Heidelberg (1994)
29. Pfitzmann, A., Pfitzmann, B., Waidner, M.: ISDN-MIXes: Untraceable Communication with Small Bandwidth Overhead. In: Effelsberg, W., Meuer, H.W., Müller, G. (eds.) Kommunikation in Verteilten Systemen. Informatik-Fachberichte, vol. 267, pp. 451–463. Springer, Heidelberg (1991)
30. Reardon, J., Goldberg, I.: Improving Tor using a TCP-over-DTLS Tunnel. In: USENIX Security Symposium, pp. 119–134. USENIX Association (2009)
31. Sako, K., Kilian, J.: Receipt-Free Mix-Type Voting Scheme. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 393–403. Springer, Heidelberg (1995)
32. Serjantov, A.: A Fresh Look at the Generalised Mix Framework. In: Borisov, N., Golle, P. (eds.) PET 2007. LNCS, vol. 4776, pp. 17–29. Springer, Heidelberg (2007)
33. Serjantov, A., Dingledine, R., Syverson, P.F.: From a Trickle to a Flood: Active Attacks on Several Mix Types. In: Petitcolas, F.A.P. (ed.) IH 2002. LNCS, vol. 2578, pp. 36–52. Springer, Heidelberg (2003)
34. Sion, R. (ed.): FC 2010. LNCS, vol. 6052. Springer, Heidelberg (2010)
35. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J.S., Becker, D.: Scalability and Accuracy in a Large-Scale Network Emulator. In: OSDI (2002)
36. Venkitasubramaniam, P., Tong, L.: Anonymous Networking with Minimum Latency in Multihop Networks. In: IEEE Symposium on Security and Privacy, pp. 18–32. IEEE Computer Society (2008)
37. Wang, W., Motani, M., Srinivasan, V.: Dependent Link Padding Algorithms for Low Latency Anonymity Systems. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conference on Computer and Communications Security, pp. 323–332. ACM (2008)
38. Westermann, B., Wendolsky, R., Pimenidis, L., Kesdogan, D.: Cryptographic Protocol Analysis of AN.ON. In: Sion [34], pp. 114–128.
39. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: OSDI (2002)