Rafael Galvez—Ed. (KUL)
Dimitris Mitropoulos (GRNET)
George Tsoukalas (GRNET)
Harry Halpin (Greenhost/LEAP)
Kali Kaneko (Greenhost/LEAP)

Minimum Viable Product (MVP)

Deliverable D4.2

February 28, 2017
PANORAMIX Project, # 653497, Horizon 2020
http://www.panoramix-project.eu

Horizon 2020
European Union funding
for Research & Innovation

Revision History

Revision	Date	Author(s)	Description
0.1	2016-12-09	RG (KUL)	Proposed table of contents
0.2	2016-02-07	HH (GH)	Messaging use case
0.3	2016-02-10	DM, GT (GRNET)	E-voting use-case sketch
0.4	2016-02-09	DM (GRNET)	Architecture overview, e-voting requirements
0.5	2016-02-17	DM (GRNET)	Documentation chapters
0.6	2016-02-23	BW (SAP)	WP6 requirements and sketch
0.7	2016-02-23	DM, GT (GRNET)	Address first review comments
0.8	2016-02-24	HH, KK (GH)	Update on messaging requirements and sketch
0.9	2016-02-24	MZ (UT)	Review
0.91	2016-02-27	BW (SAP)	Review
0.92	2016-02-28	AK (UEDIN)	Review
1.0	2016-02-28	MW,TZ (UEDIN)	Revised final version and submission to the EC

Executive Summary

This deliverable describes the software package ready to be used by use case partners to start integrating PANORAMIX into their own systems.

The Minimum Viable Product (MVP) addresses the requirements detailed in D4.1 (based on the use cases) that enable the setup of a mix network (WP4) for exchanging ballots (WP5), statistical data frames (WP6), and messages (WP7) through it.

Sections 3 and 4 showcase the intent of the MVP developers in regards to how the system will be used and further extended to provide new and enhanced functionality. The documentation provided in Sections 5 and 6 is intended to assist System Administrators and Software Developers to either setup a mix network or to make use of it from a software application.

Contents

1. Introduction

The Minimum Viable Product (MVP) is the first complete system of the PANORAMIX project. It enables the use case partners and other interested third parties to start integrating the functionality it provides through an easy to use and fully documented Application Programming Interface (API) that showcases the basic functionalities of the final system. It also serves as a common ground to reach agreement on the implementation of the requirements left for the final version. The MVP is available as open source software and can be accessed through the following link: `https://github.com/grnet/panoramix`.

To fully demonstrate the applicability of the PANORAMIX MVP, we developed a private anonymized chat room based on it.[1] This prototype demonstrates how messages can be broadcast anonymously, without allowing an eavesdropper or even legitimate users to figure out which user sent which message.

Figure 1.1 presents a sequence diagram with all the participants including the users (low left) who vote, the contributors who help to set up the mix network, and the coordinator. First, the coordinator (top left) sends the initial parameters to initialize the mix-net. Then, he or she sends invitations to the potential contributors (top right). Subsequently, contributors register their parameters and the mix-net is created.

Figure 1.2 illustrates a user connected to a private chat room called "panoramix". The user initially submitted the message "Hello world", but their message did not become visible until four more messages were submitted by other users. After the submission of all five messages, a Sphinx [1] decryption mix-net was employed to shuffle the messages, which were then printed in the chat room in a random, untraceable order. An attacker that performs traffic analysis on the network cannot link a message to its sender due to the shuffling.

In the following, we discuss how the MVP requirements are fulfilled through our MVP (Section 2), describe the architecture of the MVP (Section 3), and finally provide to System Administrators and Software Developers the information they need to set up our prototype (Sections 5 and 6).

[1] Notably, we presented it as a demo at the 10th International Conference on Computers, Privacy & Data Protection (CPDP 2016), January 2017 in Brussels.

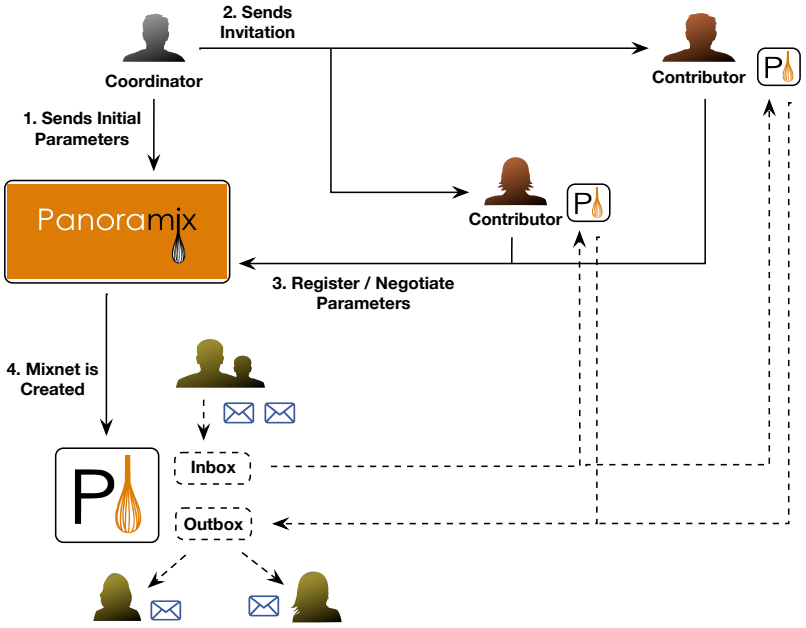

Figure 1.1: Roles and actions.

Figure 1.2: Screenshot of a user connecting to an MVP-based private chat room, called "panoramix".

2. MVP requirements

The PANORAMIX MVP fulfills all the basic requirements set in D4.1 from WP3, WP5, WP6 and WP7. Specifically, it offers an API and libraries for both server and web browser environments. In addition, it hides raw cryptographic material from both the developer and users and encapsulates the cryptographic workflow and the management of the network (e.g. establishing connections between network nodes). Finally, a mix-net coordinator can easily instantiate a mix-net in a secure manner. We further discuss how this is done in Section 3.1. In the following sections we discuss how the requirements of each use case are fulfilled through the MVP.

Based on the three use case scenarios, Sections 2.1, 2.2 and 2.3 show how the implemented requirements have been prioritized based on the needs of the use cases.

2.1 E-voting Requirements

E-voting requirements can be categorized into three categories: security, performance, and engineering [3].

Security. E-voting is a procedure where voters send messages (i.e. ballots) to the committee of trustees. First, these messages must be *verifiable* which means that senders must be convinced that their message will be delivered. In addition, the mix-net must provide means for verification of the *correctness* of the messages in the output batch. *Anonymity* is also very important because an adversary should not be able to link senders with the message they submitted. Messages should also be *private* meaning that only the recipient can read and publish the messages. Finally, the integrity of the message is very important, i.e. the message should come out of the mix-net without any modifications. Typically, a re-encryption mix-net will be employed which will provide proofs of correctness. The MVP already can use the original Zeus mix-net and expand its available choices with PANORAMIX, a new and faster re-encryption mix-net based on the Sphinx mix-net from partner UCL. *Distribution of Trust* is a major issue in e-voting. Typically, the e-voting committee should include an independent institution, the computer system administration, and all major competing interests in the election. In the context of our current MVP, a mix-net can be easily composable by the contributing servers of parties that do not share infrastructure. The distributed parties are also able to verify the security of the mix-net.

Performance. The processing at each stage and the verifiability in the mix-net take a finite amount of time thus, adding to delay in the communications. Therefore, low *latency* is crucial for real-time applications requiring anonymity. Note though that most of the security properties contradict low latency, especially under low-traffic conditions. *Throughput* can be also considered as a measure of the number of sender messages that a mix-net can output per unit of time.

It provides an estimate of the overhead due to the mix-net. The cryptographic functions necessary for producing and verifying re-encryption mix-net proofs are by far the most expensive part of the computation. The MVP has to rely on a fast mix-net algorithm and implementation.

Usability. An e-voting application should be easy to learn how to use and easy to remember how to use. Note that, in the context of PANORAMIX, usability involves voters and contributors who wish to deploy a mix server and contribute to the process. All users should be able to comfortably and effectively use an interface to accomplish the goals that it has been designed to support. In our MVP, goals are easy to accomplish quickly and with no errors and interfaces are easy to learn and navigate as we explain in the upcoming sections.

2.2 Messaging Requirements

In general, messaging requires the same properties as e-voting (i.e. security, performance, and usability), yet the relative importance of the various security properties is ranked completely differently from the e-voting use-case.

In terms of **security** properties, the most important property is *anonymity against third-party global passive adversaries*. The more generic notion of anonymity including sender and recipient anonymity is not as important. *Privacy* therefore is emphasized differently. Although we want the content to be private via the use of end-to-end encryption on the message content, privacy in terms of metadata (headers of the message with routing information, such as sender and recipient) against malicious messaging servers is also an important design goal. Unlike e-voting, cover traffic is of importance when dealing with the global passive adversary threat model (see D7.1), and this kind of adversary is just as, if not more important, than the malicious server (active or passive) threat detailed in D7.1. As the malicious server model is much more difficult, implementation will first tackle third-party anonymity on messages between servers and then we will tackle, if possible, the problem of hiding metadata from a client to the server. However, the MVP should enable both the use of mix networking on the client and the server. *Verifiability* is not needed as much as in voting, as in messaging the system can simply send the message again if it is dropped. Therefore, some kind of notification is needed for the delivery of messages, but not the zero-knowledge proofs of verified shuffling as emphasized in the e-voting use-cases.

In terms of **performance**, lower latency is needed even more than e-voting, as users expect messages should be delivered within minutes (at most) while tallying the results of e-voting can take much longer. *Throughput* is also important, and is should be noted that messaging servers participating in the mix network may have vastly different amounts of throughput due to their geographic disperson. In order to gain these properties, messaging will use a fast decryption mix-net based on Sphinx, rather than a de-cryption mix network such as Zeus. Therefore, the MVP should be able to specify both de-cryption and re-encryption options. In terms of **usability** we are first expecting to handle a static number of messaging servers participating in the mix, and so usability will focus mainly on the server-side. Later, we want to expand to both a dynamic number of servers and support of the mix by the client itself, so developer-facing documentation is necessary. More detail on the messaging requirements, in particular the problems brought up by spam in open-ended network, have already been given in detail on the use-case level in D4.1 and in email-specific technical detail in D7.1. Therefore, we will focus on the relevance of the MVP to the messaging use-case.

The MVP produced by GRNET so far is based on Python using the Django web-application framework. This is well-suited for server-to-server integration of PANORAMIX, and we plan to integrate the software in D7.2. It will likely be more difficult, but still possible, on the LEAP client and the Pixelated MUA, due to the underlying shared Python codebase for desktop

applications, although it will take more effort to do mobile integration due to the fact that Python does not run on the Android platform (so all code would have to be ported to Java). Also, we are not sure if it fulfills latency and other requirements of messaging as given in D4.1 on the abstract level and as detailed in D4.2 on a low level. We plan to experiment with Sphinx and decryption networks in more detail to determine the results, and this will be re-incorporated in D7.2. If there are severe latency or other problems with the underlying Python code, we can use the unified PANORAMIX API of the MVP on top of other custom-built code for the messaging use-case that take into account the needs for messaging for low latency use of a mix network with less of a focus on verifiability and more of a focus on cover traffic. There continue to be concerns about abuse prevention and spam, but these are outside the scope of the MVP and we will deal with these in D7.2. Overall, GRNET's progress has been impressive, and the integration of their MVP by GH and CCT for the messaging use-case will be presented in D7.2.

2.3 Surveys and Statistics

SAP intents to use the PANORAMIX mix network in a "Big Data" scenario where data is collected from several clients and then aggregated in a central database for analysis, sharing, or further processing. The purpose of using the PANORAMIX framework is to provide anonymity to the data subject by covering the origin of the data.

In the data collection process, clients send messages containing their individual data to a central server for aggregation. This resembles the basic messaging functionality of WP7 restricted to unidirectional messages, and explains why the requirements are a subset of the messaging requirements which are covered as described in 2.2. On the other hand, further requirements regarding security, performance, and usability are covered by the E-voting use case.

In addition to protecting the data through the PANORAMIX mix-net, SAP will apply *differential privacy* mechanisms to the data in order to prevent re-identification of the data subject from the data itself. Differential privacy works by perturbing the data using randomness, which requires a certain robustness for the desired analysis and statistics that are to be evaluated. Therefore, some requirements such as latency and strong verifiability can be relaxed since robust statistical data collection can tolerate delayed messages and minor amounts of undelivered messages. Overall, the requirements of WP6 towards the PANORAMIX mix network are a proper subset of the messaging and E-voting requirements.

Beyond the usability requirements from E-voting, SAP's requirements regarding ease of use and flexibility are covered as described in section 3: The Local Agent and Server components allow easy deployment and provide a flexible interface supporting the integration of PANORAMIX on a wide range of systems.

3. Software architecture

3.1 Overview

In this subsection we provide an overview of the MVP architecture. Figure 3.1 illustrates the architecture of our system. In the upcoming subsections, we will describe each component in detail.

There are three main entities involved in the architecture, namely: the *controller*, the *contributor computer*, and the *user computer*. The controller provides several *endpoints* to the other two entities. Specifically, there are two endpoints utilized by a user computer to send and receive encrypted messages and an endpoint that provides specific information regarding the kind of the mix-net and the various parameters that have to be used to either encrypt or decrypt messages. The endpoints used by the contributor computers may vary based on the protocol that is being used (e.g. decryption mix-net).

Every contributor computer contains a *wizard* component. This component can be used by the administrator to set up the mix server which in turn, will act as a mix-net peer. The mix server contains two basic components, the *crypto module* and the *PANORAMIX client*. The latter initializes the former after interacting with the controller through the corresponding endpoints. Each user computer contains a local agent with the same components.

3.2 Local Agent

Applications send or receive messages through a mix-net via a software component called the PANORAMIX Local Agent. This is provided by PANORAMIX as a standalone software service that takes care of all the cryptography and PANORAMIX internals. Specifically, the local agent encrypts and decrypts messages and manages any cryptographic keys needed. Because of the security implications it is always assumed that the local agent runs in the same machine as the application software using the mix-net so that no unencrypted communication is seen by any third party. The local agent offers the following API calls:

`init(mix-net_url, settings): status init` Initializes the local agent to work with a specific mix-net. The agent contacts the mix-net at the given URL (`mix-net_url`) and retrieves detailed information about the nature of the mix-net including its cryptographic parameters. Using this information it instantiates the appropriate software modules to be able to send and receive messages.

`settings` offers finer control over the initialization. For more details, refer to the local agent software documentation.

`status` indicates either success or failure.

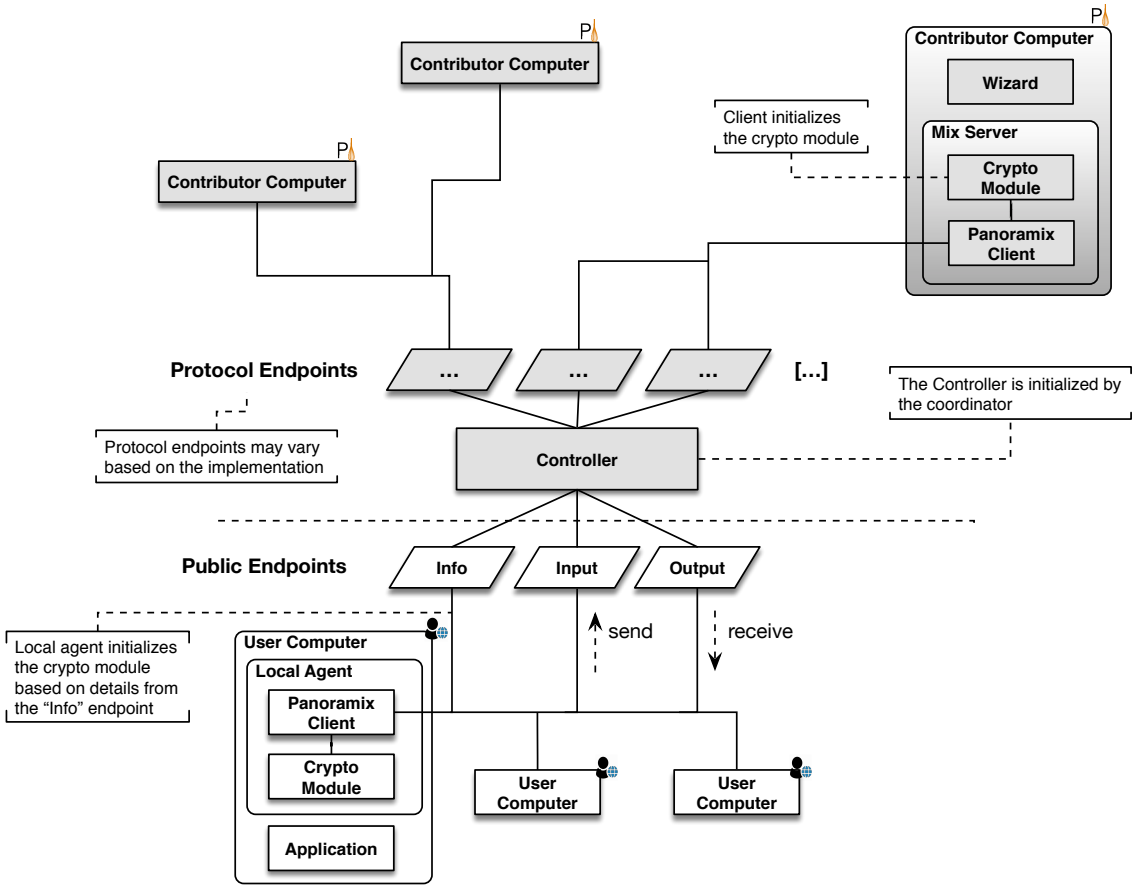

Figure 3.1: MVP Architecture.

`send(mix-net_url, recipient, message, settings): status` Sends a message to a recipient through a mix-net. The recipient is an address that is supported by the mix-net. The message is plaintext but will be encrypted by the local agent running in the same computer. The `mix-net_url` must have been initialized first with `init()`.

`settings` offers finer control over sending messages. For more, see the local agent software documentation.

`status` indicates either success or failure.

`receive(): status, mix-net_url, message` Receives an incoming message. `mix-net_url` indicates the source mix-net, `message` is the actual message contents. `status` indicates success or failure with an error message. Notice the absence of a sender.

3.3 Mix server

The mix server is employed by contributors to the mix and the premise is that they are controlled by independent parties because the mix-net relies on each of them separately to safeguard the privacy of the messages even in the presence of dishonest others.

The person or the organization that wants to contribute to a mix-net must configure the mix-net server to work with the mix-net initiated by the controller. This configuration involves setting cryptographic and other parameters for the mix-net, including who the other mixers might be, and where should the messages go after the mix-net.

Some of these parameters may have defaults or the controller might have proposed their value upon initiating the mix-net. However, at all times the contributors to the mix-net must all confirm that they agree on the same parameters. Otherwise, one might run multiple mix-nets concurrently and have different parties believe they participate in the same mix-net when they are not.

To aid with the process of setting up a mix-net, we have developed an interactive wizard that guides the contributor through the parameters allowing them to confirm or counter-propose different values. This wizard can be tuned case-by-case to process only a subset of the actual parameters interactively. The wizard itself will automatically agree on the other parameters as long as they match its defaults.

After initialization, the mix server enters a loop awaiting messages to process according to the parameters agreed by all. The actual details of the operation differ according to the mix-net type and specific implementation and is encapsulated within the cryptographic module.

3.4 Controller

In this section we provide a simple mix-net setup scenario through a sequence of GET and POST methods that take place between the different entities that we described earlier.

To set up a mix-net, the coordinator must use the controller. Specifically, the coordinator first sets up the controller database and service on a specified URL, using a dedicated configuration wizard. Then the coordinator can employ the contributor wizard to jointly set up the mix-net along with any other mix-net contributor. The coordinator first selects the cryptographic settings, and registers (by providing a specific identifier — below the identifier is 4bdc9bf518b667f36f2d13a) their personal cryptographic key to the controller.

```
"POST /panoramix/peers/ HTTP/1.1" 201 490
"GET /panoramix/peers/4bdc9bf518b667f36f2d13a/ HTTP/1.1" 200 490
```

Next, the coordinator selects the mix-net attributes and initiates a negotiation in order to create the mix-net in agreement with other mix-net contributors. The coordinator invites them to join the process, by sharing with them the controller URL and a cryptographically secure invitation. Using the same wizard, the invited contributors can choose to approve the coordinator's proposal and finalize the creation of the mix-net. This is done via rounds of negotiations until all involved contributors agree upon the same proposal.

```
"POST /panoramix/contributions/ HTTP/1.1" 201 878
"GET /panoramix/negotiations/N7IjtRZic-RqBE-taxeXT0/ HTTP/1.1" 200 1059
"POST /panoramix/peers/ HTTP/1.1" 201 490
"GET /panoramix/peers/03bf1c4b6130ca1/ HTTP/1.1" 200 490
```

Since the mix-net is now set up, we can give the end-users the mix-net URL, which can be used to access the mix-net.

```
https://<controller_url>/panoramix/peers/03bf1c4b6130ca1e16ed/
```

However, the mix-net is not ready to accept incoming messages, before all mix-net contributors agree to create and open an inbox (endpoint - see Figure 3.1). This is also facilitated by the wizard. The coordinator selects the inbox attributes and then it is up to the other contributors to accept them through a round of negotiations.

```
"GET /panoramix/contributions/?negotiation=2neqolRkF20im6L HTTP/1.1" 200 1661
"POST /panoramix/contributions/ HTTP/1.1" 201 879
```

```
"GET /panoramix/negotiations/2neqolRkF20im6LYz8l/ HTTP/1.1" 200 1376
"POST /panoramix/endpoints/ HTTP/1.1" 201 447
"GET /panoramix/endpoints/ep_1/ HTTP/1.1" 200 447
```

Now the mix-net is ready to accept messages on the created inbox. The mix-net contributors can use the wizard to automate the processing of the inbox. The wizard polls the inbox to check whether it is ready to be processed, and facilitates the contributors to agree on its closure.

```
"POST /panoramix/contributions/ HTTP/1.1" 201 864
"GET /panoramix/negotiations/iaaHFXyq25WyBwmvzn7Em/ HTTP/1.1" 200 1361
"PATCH /panoramix/endpoints/ep_1/ HTTP/1.1" 200 654
```

Next, each contributor retrieves the inbox messages, processes them locally using the cryptographic operation specified by the endpoint and uploads the processed messages.

```
"GET /panoramix/messages/?box=ACCEPTED&endpoint_id=ep_1 HTTP/1.1" 200 2037
"POST /panoramix/messages/ HTTP/1.1" 201 2034
```

Depending on the application, messages may be forwarded to the contributors' own endpoints for further processing, until the final results reach the outbox.

3.5 Future development

We are planning to extend our MVP to expand the functionalities of our MVP and fulfill some critical requirements presented in Section 2 in a better manner.

For instance, we mentioned earlier that the MVP already uses a mix-net based on Sphinx from UCL, which involves static routing. This may not be ideal in settings where we would like to empower the sender with the choice of different routes. Hence, in our future work we plan to provide a way where the coordinator has the ability to set up dynamic routes for message delivery. In addition, in the context of WP5, we have developed the re-encryption mix-net developed by PANORAMIX consortium members, Zajac et al. [2]. Our initial tests showed that this mix-net is faster than the Sako-Kilian mix-net that is used by Zeus [4]. We are planning to integrate this mix-net into the PANORAMIX platform in the near future. In this way we will attempt to address the main limiting factor in the wider adoption of Zeus and e-voting in general which is efficiency.

4. Use case sketches

4.1 E-voting

In the Zeus e-voting use case, each of the trustees must guarantee the correctness of the process. Also, they must all agree with each other. Each has a part of the secret key for the decryption of the votes and each one of them can prevent the others from breaching protocol.

Vote secrecy is also guaranteed by the different mix-net contributors. An ideal scenario is when each of the trustees is also a mix-net contributor. Therefore, when setting up an election, a trustee also sets up their mix-server. This can be done easily through the PANORAMIX wizard as we will see in the upcoming Sections. Hence, the trustees are not put off and security is enhanced, especially if a fast mix-net can be deployed so that the trustees can also contribute to the mixing process using their own computers.

First, clients encrypt votes using the mix-net's public key. Then, they submit the encrypted votes to the e-voting server for processing. This processing is based on the Zeus e-voting protocol. The following pseudo-code is executed at the voter's computer:

```
from panoramix import MixnetClient
from zeus import VotingClient

voting = VotingClient("https://panoramix.dev.grnet.gr/zeus/elections/123/")
mixnet = MixnetClient("https://panoramix.dev.grnet.gr/zeus/elections/123/mixnet/")

ballot_choices = voting.get_ballot_choices()
filled_ballot = get_user_preferences(ballot_choices)
encrypted_ballot = mixnet.encrypt(filled_ballot)
voting.submit_ballot(encrypted_ballot)
```

When the voting procedure is finished, the application has all the encrypted ballots ready for mixing and submits them to the mix-net. When the mixing process is done, the application receives the shuffled and decrypted ballots. The decrypted ballots are then ready for counting and the final results are produced. The above are illustrated in the following pseudocode:

```
from panoramix import mixnetclient
from zeus import votingclient

voting = votingclient("https://panoramix.dev.grnet.gr/zeus/elections/123/")
mixnet = mixnetclient("https://panoramix.dev.grnet.gr/zeus/elections/123/mixnet/")

nr_encrypted_ballots = 0
for encrypted_ballot in voting.get_votes_for_mixing():
    mixnet.send(encrypted_ballot)
    nr_encrypted_ballots += 1

nr_votes = 0
```

```
decrypted_votes = []
while nr_votes < nr_encrypted_ballots:
    decrypted_vote = mixnet.receive()
    decrypted_votes.append(decrypted_vote)

proofs = mixnet.get_proofs()
voting.put_results(decrypted_votes)
voting.put_mixing_proofs(proofs)
results = voting.count_results()
```

4.2 Messaging

For messaging, first GH and CCT will tackle the problem of third-party anonymity between servers who also serve as nodes in the mix network, and then we will attempt to integrate the clients. The reason for this is that the server infrastructure (Linux-based) is uniform, while the client side features a number of heterogenous clients (Thunderbird, K-9 mail, Pixelated, as well as proprietary clients such as Apple Mail and Outlook). Therefore, the integration of the PANORAMIX mix-net with the LEAP encrypted email messaging infrastructure will happen in two phases. The server-to-server use of PANORAMIX we will call "Phase 0" and the client-to-server use of PANORAMIX we will call "Phase 1."

In Phase 0, the receiving SMTP endpoint in the provider will inject incoming messages into the mix-net according to the presence of an optional header in the incoming message. In detail, the straightforward methodology for the implementation of this Phase 0 server code is to have it done as a postfix filter. In this stage, the mix-net client is only deployed in the MX (mail) providers, via the LEAP Platform, and mix servers may be run by providers or other third-party infrastructure. The MX server authenticates the SMTP clients based on traditional certificate-methods methods, and after authentication composes the Sphinx packet and delivers the wrapped mail to its own mix-net incoming queue ("inbox" in API's terminology).

To avoid abuse, in the initial phase only messages that are addressed to providers participating of the mix-net will be injected into the mix-net. This means also that all the internal traffic between PANORAMIX-enabled providers (such as traffic between Riseup.net and Bitmask.net) will be routed through the mix-net, but traffic to other servers (such as GMail) will not go through the mix-net.

```
from panoramix import MixnetClient, endpointFactory
mixnet = MixnetClient("https://panoramix.bitmask.net/nodes.json")
entrypoint = endpointFactory(mixnet, 'localhost')

wants_mix = lambda msg: msg.getHeader('X-PANORAMIX-Mixable')
can_mix = lambda msg: get_domain(msg.envelope.getHeader('RCPT TO')) in mixnet

while True:
  incoming = process_incoming_queue()
  for msg in incoming:
    if wants_mix(msg) and can_mix(msg)
        mixable = mixnet.prepare(payload=msg.asString())
        mixable.assign_random_route(mixnet.nodes)
        mixable.deliver_to(entrypoint)
    else:
      passthrough(msg)
```

The delivery of the message at the exit of the mix-net doesn't present major problems, because the outgoing queue of the mix-net outputs a regular SMTP envelope the message is just delivered

to the user mailbox by the traditional pathways, in the same way that GPG-encrypted email delivers an encrypted message in the content.

By contrast, the mix-net client is moved to the client in Phase 1. The Python mix-net package will be shipped as part of the Bitmask client distribution, and the client will learn the needed routing info from the bootstrapped configuration files that the provider exposes. This removes the amount of private information that the Provider receives in the SMTP entrypoint (which still has to sit in front of the mix-net incoming queue, to be able to enforce authentication and abuse control).

In the client (Phase 1), the switch to deliver through the mix-net is done based on some opt-in flag that user has to toggle in their user interface, and both the route choice and the preparation of the mix-net wrapped message encrypted to the key material of the entrypoint happen before sending it to the local SMTP proxy:

```
from panoramix import MixnetClient, endpointFactory
mixnet = MixnetClient("https://panoramix.bitmask.net/nodes.json")
entrypoint = endpointFactory(mixnet, myProvider)

if getConf('mixnet_enabled'):
  mixable = mixnet.prepare(payload=msg.asString())
  mixable.assign_random_route(
    mixnet.nodes, entrypoint=myProvider,
    outBox=myProvider)
  mixable.addPadding(mixnet.size_distribution)
  msg.setPayload(mixable)
  msg.addHeader('X-PANORAMIX-Mixable', 'yes')
  smtpProxy.enable_cover_traffic()
smtpProxy.send_message(msg)
```

The SMTP server in the mx node of the provider will just decide upon the mix-net switch in the header, as before, but now the envelope is just opaque from its perspective:

```
while True:
  incoming = process_incoming_queue()
  for msg in incoming:
    if wants_mix(msg):
        mixable = msg.getPayload()
        mixable.deliver_to(entrypoint)
    else:
      passthrough(msg)
```

4.3 Surveys and Statistics

In the surveys and statistics use case, only basic mix network functionality is used: Data from several clients (e.g. users participating in a survey, or smart devices) will be sent to a central server over the mix network, where the data will be aggregated in a database. In this way, the setup configuration is very similar to the one given in Section 4.1.

We assume a pre-deployed mix-net, where each client and the database server run their own instance of the *Local Agent* (see Section 3). Once data is available at a client, it sends a message containing the data to the central server, using the `send` function of its client-side Local Agent. On the server side, the database server periodically checks for new messages using the `receive` function of the server-side Local Agent. The server then stores the received data in a database for further processing (for instance, differentially private data sanitization or statistics), without learning the source of the data.

5. Documentation for System Administrators

This section provides all the information that a coordinator and a contributor will need to set up our MVP. The scenario follows the steps described in Section 3.4.

5.1 Coordinator

To set up a mix-net, the coordinator must use the controller. Specifically, the coordinator first sets up the controller database and service on a specified URL, using a dedicated server configuration wizard.

```
% panoramix-server-wizard
welcome to panoramix server wizard!
configuration file is: /tmp/panserver
set panoramix_config environment variable to override
set catalog_url: (enter for default 'http://127.0.0.1:8000/')
catalog_url: http://127.0.0.1:8000/
```

After setting the controller URL (CATALOG_URL), the coordinator must specify the cryptographic backend and settings. observe that in our current set up, the Sphinx [1] decryption mix-net is the default option.

```
Select backend, one of SPHINXMIX, ZEUS (default: 'SPHINXMIX')
backend: SPHINXMIX
Set BODY_LEN (default: '1024')
BODY_LEN: 1024
Set GROUP (default: '713')
GROUP: 713
Set HEADER_LEN (default: '192')
HEADER_LEN: 192
```

`BODY_LEN, GROUP, HEADER_LEN` are settings related to Sphinx (e.g. `BODY_LEN` is the maximum length that a message may have). The coordinator is then instructed how to initialize the database and the server.

```
You need to setup your database once with
 panoramix-manage migrate
Start server with
 PANORAMIX_CONFIG=/tmp/panserver panoramix-manage runserver 127.0.0.1:8000
```

Then the coordinator can employ the contributor wizard to jointly set up the mix-net along with any other mix-net contributor. The coordinator first specifies the controller URL and selects the

cryptographic settings, which should match those of the controller.

```
% panoramix-wizard
Welcome to PANORAMIX wizard!
Configuration file is: /tmp/pancfg1
Set PANORAMIX_CONFIG environment variable to override
Set CATALOG_URL (default: 'http://127.0.0.1:8000/')
CATALOG_URL: http://127.0.0.1:8000/
Choose 'create' or 'join' mix-net
role: create
Select backend, one of SPHINXMIX, ZEUS (default: 'SPHINXMIX')
backend: SPHINXMIX
Set BODY_LEN (default: '1024')
BODY_LEN: 1024
Set GROUP (default: '713')
GROUP: 713
Set HEADER_LEN (default: '192')
HEADER_LEN: 192
```

Next, the coordinator creates and registers their personal cryptographic key to the controller.

```
No key available. Choose 'set' or 'create' (default: 'create')
action: create
Created key with values: {'SECRET':
  u'A808BCBBCC3BC141EA7B2FAFB5DD3BDAF6EF5301DB4839296519EBE7', 'PUBLIC':
  '039073ae91caf2eed8971f26cfcbc95a419ea1731a460607ea1aae2023'}
Specify name to register as peer
PEER_NAME: peer1
Registered peer with PEER_ID: 039073ae91caf2eed8971f26cfcbc95a419ea1731a460607ea1aae2023
```

Then, the coordinator selects the mix-net attributes and initiates a negotiation in order to create the mix-net in agreement with other mix-net contributors. The coordinator invites them to join the process, by sharing with them the controller URL and a cryptographically secure invitation.

```
Choose mix-net peer name
name: our_mix-net
Give number of invitations to create
invitations: 1
Send invitations to peers:
  TLCWFVt8Y3gnvvtYNVRM0uM4paQZaUwTRA-HtOnBVM4|xHqbpCEmGirxpSsrByx9zQhmQBFLyrBJXcoxSLJJhUI
Your initial proposal is contribution: 21
```

5.2 Contributor

Using the same wizard, the invited contributors can choose to approve the coordinator's proposal and finalize the creation of the mix-net. This is done via rounds of negotiations until all involved contributors agree upon the same proposal. The contributor will see the same messages but instead he or she will choose to *join* the mix-net.

```
% panoramix-wizard
Welcome to PANORAMIX wizard!
Configuration file is: /tmp/pancfg2
```

```
Set PANORAMIX_CONFIG environment variable to override
Set CATALOG_URL (default: 'http://127.0.0.1:8000/')
CATALOG_URL:
Choose 'create' or 'join' mix-net
role: join
Give invitation to create mix peer
JOIN_INVITATION:
TLCWFVt8Y3gnvvtYNVRM0uM4paQZaUwTRA-HtOnBVM4|xHqbpCEmGirxpSsrByx9zQhmQBFLyrBJXcoxSLJJhUI
Negotiation initialized by peer
039073ae91caf2eed8971f26cfcbc95a419ea1731a460607ea1aae2023 with contribution
21.
Proposed crypto backend: 'SPHINXMIX'; 'accept' or 'abort'? (default:
'accept')
response: accept
Proposed crypto params: '{u'BODY_LEN': 1024, u'GROUP': 713, u'HEADER_LEN':
  192}'; 'accept' or 'abort'? (default: 'accept')
response: accept
```

In the meantime, this message will be displayed to the coordinator.

```
Invitations pending: set(['xHqbpCEmGirxpSsrByx9zQhmQBFLyrBJXcoxSLJJhUI'])
All invited peers have joined. Sending accept contribution.
Your new contribution id is: 24
No consensus yet.
Consensus reached:
f1f3055b0be745c6224e9ad4f9512c98238b4422cd03ea687aee0fc819daba24
Negotiation finished successfully. Applying consensus.
Created combined peer
0255e5a9676ad006a8443acf5fc45d14e49678d1d64f9.
```

Since the mix-net is now set up, we can give the end-users the mix-net URL, which can be used to access the mix-net.

```
http://127.0.0.1:8000/panoramix/peers/0255e5a9676ad006a8443acf5fc45d14e49678d1d64f9/
```

However, the mix-net is not ready to accept incoming messages, before all mix-net contributors agree to create and open an inbox. This is also be done through the wizard. The coordinator selects the inbox attributes and then it is up to the other contributors accept them through a round of negotiations.

```
Specify endpoint name to create on combined peer
ENDPOINT_NAME: gateway
Select endpoint type, one of SPHINXMIX_GATEWAY, SPHINXMIX (default:
SPHINXMIX_GATEWAY)
ENDPOINT_TYPE: SPHINXMIX_GATEWAY
Specify minimum size
MIN_SIZE: 3
Specify maximum size:
MAX_SIZE: 10
Give description:
EP_DESCRIPTION: the mix-net gateway
Contribution pending from: set([u'024f06abba6aa750acb07f084c158395dd4cfc8eaa2e3d13e7bd31ba63

...
```

```
All peer owners have agreed. Sending accept contribution.
Sent contribution 28
No consensus yet.
Consensus reached:
6e2ca8b1198efed79df24b988b6cebfb8b207029fa74b494aff04b3d47285859
Negotiation finished successfully. Applying consensus.
Created endpoint gateway_1.
```

Now the mix-net is ready to accept messages on the created inbox. The mix-net contributors can use the wizard to automate the processing of the inbox. The wizard polls the inbox to check whether it is ready to be processed, and facilitates the contributors to agree on the its closure.

```
Waiting until minimum inbox size is reached.
Contribution pending from:
set([u'024f06abba6aa750acb07f084c158395dd4cfc8eaa2e3d13e7bd31ba63'])
All peer owners have agreed. Sending accept contribution.
Sent contribution 34
Consensus reached:
365e84425bb914bb4342cb2df8eead086e1ca0ef5c968ee4477c25e2e980a50a
Negotiation finished successfully. Applying consensus.
Closed endpoint gateway_1.
```

Next, each contributor retrieves the inbox messages, processes them locally using the cryptographic operation specified by the endpoint and uploads the processed messages. Depending on the application, messages may be forwarded to the contributors' own endpoints for further processing, until the final results reach the outbox.

```
Waiting to collect inbox.
Collected input for inbox of 039073a_for_ep_gateway_1.
Closed endpoint 039073a_for_ep_gateway_1
Processed endpoint 039073a_for_ep_gateway_1
Collected input for processbox of gateway_1.
Contribution pending from:
set([u'024f06abba6aa750acb07f084c158395dd4cfc8eaa2e3d13e7bd31ba63'])
All peer owners have agreed. Sending accept contribution.
Sent contribution 41
Consensus reached:
ddb5dfeb2adf62f682fb1500b289f7430f7d7bc4b4607c32234e7a1956fea035
Negotiation finished successfully. Applying consensus.
Processed endpoint gateway_1.
```

6. Documentation for Software Developers

6.1 Overview

As we discussed in D4.1, the PANORAMIX API is based around `peers` who exchange `messages`. Each peer performs supported operations (e.g. mixing, decrypting) through respective `endpoints`. Each endpoint processes messages in bulk: An endpoint `cycle` opens up in order to accept messages in its `inbox`. When sufficient messages are collected in the inbox, the peer retrieves the messages, processes them and posts them to its `outbox`. An external posting mechanism is responsible to send the outbox messages to the inboxes of their recipients.

6.2 Negotiations and Consensus

Negotiation is a mechanism that allows peers to agree upon a common text after rounds of amendments. The final text is signed by all participating peers. A text can be a prescription for an action that requires consensus of all stakeholders.

When a negotiation completes successfully, a consensus identifier is computed by hashing the negotiation data. This identifier can be provided to any operation that requires a consensus to proceed. For instance, in order to create a new peer with multiple owners there must be a consensus among the owners. The owners of a peer must also agree in order for any peer-related action to take place, for example to create an endpoint or to publish the endpoint's outbox.

6.2.1 Initiate a negotiation

The peer who starts a new negotiation is given a hard-to-guess negotiation id. The peer can then invite other peers to the negotiation by sharing the id with them.

URI	Method	Description
`/negotiations`	POST	Initiate a negotiation

Example request:

```
{
  "data": {},
  "info": {"resource": "negotiation", "operation": "create"},
  "meta": {"signature": "payload signature", "key_data": "public key"}
}
```

6.2.2 Get negotiation details

URI	Method	Description
`/negotiations/<negotiation_id>`	GET	Get negotiation details

Get negotiation details by id or consensus id.

Example response:

```
{
  "data": {
    "id": "long_negotiation_id",
    "text": None,
    "status": "OPEN",
    "timestamp": None,
    "consensus": None,
    "signings": [],
  }
}
```

Example response:

```
{
  "data": {
    "id": "long_negotiation_id",
    "text": "agreed upon text",
    "status": "DONE",
    "timestamp": "consensus timestamp",
    "consensus": "consensus hash",
    "signings": [{"signer_key_id": "peer1",
                  "signature": "peer1 sig"},
                 {"signer_key_id": "peer2",
                  "signature": "peer2 sig"}
                ]
  }
}
```

6.2.3 Contribute to negotiation

Contribute a signed text to a negotiation. The text consists of the text body and a metadata dict. If all peers participating so far sign the same text that include the metadata `"accept": True`, then the negotiation completes successfully and the consensus id is produced. No more contributions are accepted.

Note: If the original contributor submits a text with `"accept": True`, the negotiation will complete successfully, although just one peer has contributed. Such a single-peer "consensus" may be useful in order to record a decision for an action in a uniform way regardless of the number of involved peers.

URI	Method	Description
`/contributions/`	POST	Contribute to negotiation

Example request:

```
{
  "data": {"negotiation_id": "neg_id",
           "text": "a text describing a peer creation",
```

```
            "signature": "text signature"},
  "info": {"resource": "contribution", "operation": "create"},
  "meta": {"signature": "payload signature", "key_data": "public key"}
}
```

Note: The contribution text should be a canonical representation of a dictionary of the following structure:

```
{
  "data": {...},
  "info": {...},
  "meta": {"accept": bool,
           "signers": list,
           ...}
}
```

6.2.4 List contributions to a negotiation

URI	Method	Description
`/contributions/`	GET	List contributions to a negotiation

List contributions. Filtering by negotiation id is required.

Example response:

```
[{
   "data": {
     "id": "contribution_id",
     "negotiation_id": "neg_id",
     "text": "contribution text",
     "latest": True,
     "signer_key_id": "signer's public key",
     "signature": "signature",
   }
}]
```

6.3 Peers

A peer is any participant to the mix-net, either a mix-net contributor, a correspondent, an auditor, or any other stakeholder. A peer must be registered to the mix-net controller using a cryptographic identifier.

6.3.1 Create a Peer

Create a new peer with the specified parameters; see the example below. You must always provide a `consensus_id`, indicating a decision to create a peer agreed upon by all stakeholders through a negotiation. This applies for the simple case of creating a peer with no owners, as well.

URI	Method	Description
`/peers`	POST	Create a peer

Example request:

```
{
  "data": {"key_data": "public key",
           "key_id: "13C18335A029BEC5",
           "status": "READY",
           "owners": [{"owner_key_id": "owner1"},
                      {"owner_key_id": "owner2"}],
           "key_type": 1,
           "name": "peer1"},
  "info": {"operation": "create", "resource": "peer"},
  "by_consensus": {"consensus_id": "<consensus id>",
                   "consensus_type": "structural"}
  "meta": {"signature": "payload signature", "key_data": "public key"},
}
```

6.3.2 Get peer info

Get info for a single peer.

URI	Method	Description
/peers/<peer_id>	GET	Get info for a peer

Example response:

```
{
  "data": {"key_data": "public key",
           "key_id: "13C18335A029BEC5",
           "status": "READY",
           "name": "peer1",
           "key_type": 1,
           "key_type_params": "params",
           "owners": [{"owner_key_id": "owner1"},
                      {"owner_key_id": "owner2"}],
           "consensus_logs": [{"timestamp": "action timestamp",
                               "status": "READY",
                               "consensus_id": "consensus id"}]
         }
}
```

6.3.3 List Peers

Returns a list containing information about the registered peers.

URI	Method	Description
/peers	GET	List peers

Example response:

```
[{
   "data": { ... }
}]
```

6.4 Endpoints

A peer handles messages in its endpoints. An endpoint specifies a type of operation along with relevant endpoint parameters, such as the minimum and maximum number of allowed messages. A correspondent sends messages to an open endpoint. Endpoint owners can agree to close the endpoint when suited and, after processing the inbox, publish the results in the outbox.

6.4.1 Create a peer endpoint

Creating an endpoint requires a consensus id, which proves the agreement of all peer owners on the action.

URI	Method	Description
`/endpoints`	POST	Create a peer endpoint

Example request:

```
{
  "data": {"peer_id": "13C18335A029BEC5",
           "endpoint_id": "identifier",
           "endpoint_type": "ZEUS_SK_MIX",
           "endpoint_params": "",
           "description": "a description",
           "status": "OPEN",
           "size_min": 10,
           "size_max": 1000},
  "info": {"operation": "create", "resource: "endpoint"},
  "by_consensus": {"consensus_id": "<consensus id>",
                   "consensus_type": "structural"}
  "meta": {"signature": "payload signature", "key_data": "public key"},
}
```

6.4.2 Update an endpoint

The status of an endpoint can be updated, given the last consensus id and status-specific required data.

URI	Method	Description
`/endpoint/<endpoint_id>`	PATCH	Partially update an endpoint

Example request:

```
{
  "data": {"endpoint_id": "identifier",
           "status": "PROCESSED",
           "message_hashes": ["a processed message hash"],
           "process_proof": "the processing proof",
          }
  "info": {"operation": "partial_update",
           "resource": "endpoint",
           "on_last_consensus_id": "previous consensus"},
  "meta": {"signature": "payload signature", "key_data": "public key"},
}
```

6.4.3 Get endpoint info

URI	Method	Description
`/endpoint/<endpoint_id>`	GET	Get info for an endpoint

Example response:

```
{
  "data": {"peer_id": "13C18335A029BEC5",
           "endpoint_id": "identifier",
           "endpoint_type": "ZEUS_SK_MIX",
           "endpoint_params": "",
           "description": "a description",
           "status": "CLOSED",
           "size_min": 10,
           "size_max": 1000,
           "inbox_hash": "inbox hash",
           "last_message_id": "message_id",
           "consensus_logs": [{"timestamp": "open action timestamp",
                               "status": "OPEN",
                               "consensus_id": "consensus id1"},
                              {"timestamp": "close action timestamp",
                               "status": "CLOSED",
                               "consensus_id": "consensus id2"}]
         }
}
```

6.4.4 List endpoints

URI	Method	Description
`/endpoints`	GET	List endpoints

Example response:

```
[{
   "data": { ... }
}]
```

6.5 Messages

Messages are posted to an endpoint's `inbox` of a specified peer. Once a sufficient number of messages are collected, the peer retrieves the inbox messages, processes them and uploads the transformed messages to the `processbox`. Once the peer owners agree on the results and mark the endpoint as PROCESSED (see above), the processed messages move to the `outbox`.

6.5.1 Send a message to inbox/processbox

URI	Method	Description
`/messages`	POST	Send a message

No consensus is needed in order to send a message.

Example request:

```
{
  "data": {"endpoint_id": "endpoint name",
           "box": "INBOX",
           "sender": "FC650CA0F7749FF0",
           "recipient": "13C18335A029BEC5",
           "text": "encrypted message"
          },
  "info": {"operation": "create", "resource": "message"},
  "meta": {"signature": "payload signature", "key_data": "public key"}
}
```

6.5.2 List messages

One can list the messages of a specified endpoint and box.

URI	Method	Description
`/messages`	GET	List messages

Example inbox response:

```
[{
   "data": {"endpoint_id": "endpoint name",
            "box": "INBOX",
            "id": 1,
            "sender": "orig_sender1",
            "recipient": "this_peer",
            "text": "encrypted message 1",
            "message_hash": "msg hash 1"}
 },
   "data": {"endpoint_id": "endpoint name",
            "box": "INBOX",
            "id": 2,
            "sender": "orig_sender2",
            "recipient": "this_peer",
            "text": "encrypted message 2",
            "message_hash": "msg hash 2"}
 }
]
```

Example outbox response:

```
[{
   "data": {"endpoint_id": "endpoint name",
            "box": "OUTBOX",
            "id": 3,
            "sender": "this_peer",
            "recipient": "next_peer_a",
            "text": "decrypted message a",
            "message_hash": "msg hash a"}
 },
 {
   "data": {"endpoint_id": "endpoint name",
            "box": "OUTBOX",
            "id": 4,
            "sender": "this_peer",
            "recipient": "next_peer_b",
            "text": "decrypted message b",
```

```
            "message_hash": "msg hash b"}
 }
]
```

In this example, we assume that processing has shuffled the messages in order to hide the connection between encrypted messages (1, 2) and decrypted messages (a and b).

7. Conclusion

The MVP is a solid working system and can act as a foundation for the final system that will provide all the functionality needed in our proposed use cases.

Basic features and essential requirements are already implemented in this released software package. Note that it was successfully demonstrated in a public venue (CPDP 2017) with non expert users, and the use case partners are already working on incorporating the codebase into their products.

The documentation provided guides both the actual deployment of the system and its future development in regards to the complete set of requirements provided in D4.1 and required by WP3, WP5, WP6 and WP7.

Bibliography

[1] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 269–282, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Prastudy Fauzi, Helger Lipmaa, and Michal Zajac. A shuffle argument secure in the generic model. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, pages 841–872, 2016.

[3] Jian Ren and Jie Wu. Survey on anonymous communications in computer networks. *Comput. Commun.*, 33(4):420–431, March 2010.

[4] Georgios Tsoukalas, Kostas Papadimitriou, Panos Louridas, and Panayiotis Tsanakas. From Helios to Zeus. In *2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA, August 12-13*, 2013.