D7.4 Final Version of the Truly Anonymous Data Collection Prototype

Document Identification			
Status	Final	**Due Date**	30/9/2020
Version	1.0	**Submission Date**	30/9/2020

Related WP	WP7	**Document Reference**	D7.4
Related Deliverable(s)	D7.1, D7.2, D7.3, D7.5, D7.6, D7.7, D7.8, D7.9, D7.10, D7.11, D7.12	**Dissemination Level(*)**	PU
Lead Participant	WALLIX	**Lead Author**	Norman Scaife (WALLIX)
Contributors	WALLIX	**Reviewers**	Miguel Angel Mateo Montero (ATOS) Azam Soleimanian (ENS)

Keywords:
Final Version Demonstrator Report, technical specification, use-case, web analytics, crypto API

This document is issued within the frame and for the purpose of the FENTEC project. This project has received funding from the European Union's Horizon2020 under Grant Agreement No. 780108. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level.-PU: Public, fully open, e.g. web; CO: Confidential, restricted under conditions set out in Model Grant Agreement; CI: Classified, Int = Internal Working Document, information as referred to in Commission Decision 2001/844/EC.

Document Information

List of Contributors	
Name	Partner
Norman Scaife	WALLIX

Document History			
Version	Date	Change editors	Changes
0.1	19/05/2020	WALLIX	First content
0.2	27/07/2020	WALLIX	Mention DSum protocol
0.3	07/09/2020	WALLIX	Describe partitioning and key server
0.4	21/09/2020	WALLIX	Final readthrough and completion of front matter
0.5	22/09/2020	WALLIX	Version for internal review
1.0	28/09/2020	WALLIX	Incorporate reviewers comments, final version

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Norman Scaife (WALLIX)	30/9/2020
Technical Manager	Michel Abdalla (ENS)	30/9/2020
Quality Manager	Diego Esteban (ATOS)	30/9/2020
Project Coordinator	Francisco Gala (ATOS)	30/9/2020

Table of Contents

List of Figures

List of Acronyms

Acronym	Description
API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
DMCFE	Decentralized Multi-Client Functional Encryption
FE	Functional Encryption
REST	REpresentational State Transfer

Executive Summary

This deliverable describes the WALLIX Web Analytics use case prototype at the end of the project. The cryptographic protocol is described and the platform upon which the prototype is based is presented. Some notes about the software implementation are provided and a demonstration system for proving the viability of the technique is shown. We present a method for improving the reliability of online polls conducted using this system and discuss issues of trust with our current implementation. We conclude that the technique of Functional Encryption can be deployed in a useful manner for real-world applications in the field of Web Analytics.

1 Introduction

Following the Requirements Analysis D3.1[6], the Initial Technical Specification D7.1[8] and the Preliminary Use Case Report D7.3[13], we now present the final work into the development of the Wallix truly anonymous web analytics use-case prototype.
The goals of this document are to describe, for the Wallix use-case:

- how the prerequisites for prototype development were met, and
- the final state of development of the use-case in terms of functionality.

We defer analysis of performance and security of the prototype to the two final test reports D7.10 [16] and D7.12 [17].
The technical specification documents D7.1[8] and D7.2[12] defined the required functionality for the prototype and how to achieve that using existing tools plus cryptographic protocols defined by the academic partners and the FENTEC library implemented by XLAB.
The overall goal of this phase of the project (Task 7.2) is to produce a working prototype in order to assess the suitability and efficacy of the cryptographic protocol and its ability to meet the requirements of the application. To this end, in this document, we concentrate on describing how our work meets these requirements. In this final phase of the project we concentrate more upon optimising the performance and security aspects of the use-case while maintaining the functional requirements.
This deliverable is produced concurrently with the development of the prototypes for the other two use-cases and leads into the reports on performance and security:

- D7.1 Preliminary specification of FENTEC prototypes (M9)
- D7.2 Final specification of FENTEC prototypes (M25)
- D7.3 First version of the truly anonymous data collection prototype (M22)
- D7.5 First version of the privacy enhanced digital currency prototype (M22)
- D7.6 Final version of the privacy enhanced digital currency prototype (M33)
- D7.7 First version of the IoT key distribution prototype (M22)
- D7.8 Final version of the IoT key distribution prototype (M33)
- D7.9 First test report of the FENTEC prototypes (M23)
- D7.10 Final test report of the FENTEC prototypes (M34)
- D7.11 Performance report for FENTEC prototypes after first cycle (M24)
- D7.12 Final performance report for FENTEC prototypes after second cycle (M36)

1.1 Structure of the Document

section 2.2 describes the cryptographic protocol used for the prototype.

section 2.4 specifies the physical phases of the polling process.

section 2.5 gives a brief description of the platform which has been developed.

section 2.6 outlines the structure of the software and the method of operation.

section 2.7 summarises what the demonstrator version of the use case application attempts to prove.

section 3 concludes with a brief statement about conformance with project goals.

section 4 gives some brief indications for the future of the technology for after the project.

2 Data Collection Prototype Demonstrator

2.1 Introduction

The WALLIX Web Analytics use case is intended to allow the gathering of statistical information in such a way that the participants' data remain anonymous to the entity conducting the analytics and also from each other.

This is a scenario in which Functional Encryption (FE) is ideal since it allows the application of simple functions to encrypted data. With the *proviso* that the participants in the statistical analysis (which we call a poll) are prepared to send their data in encrypted form for this type of analysis we are able to provide simple statistical analysis of the participants' data.

For the FENTEC use case the participants are WALLIX Awless clients. Awless is a command line tool for managing Amazon AWS accounts. Statistical analysis was originally implemented using cleartext transmissions to a central server hosted at WALLIX. However, the Awless user community refused to countenance the sending of any part of their Amazon configuration data to a third party, even a trusted one such as WALLIX. As a result, the statistical analysis was removed from public versions of the tool.

Further analysis of the Awless code combined with a review of the existing cryptographic protocols available, explained in FENTEC deliverables D4.1 [7] and D4.2 [9], led to the requirements analysis in D3.1 [6] and the technical specification for the use case prototype in D7.1 [8] and the subsequent initial version of the prototype reported in D7.3 [13] and analysed in D7.9 [14] and D7.11 [11]. To summarize this work, we can state that the requirements of the system are mainly that all encryption is carried out on the clients which means a distributed key system. At the time of writing the specification, there existed one cryptographic protocol which suited these requirements, an inner product distributed multi-client functional encryption protocol (DMCFE [3]). Note that other protocols have become available subsequently, for example the DSum [4] protocol and the schemes not using pairings presented in D4.3 [15]. For this final phase of the project, we retain the original two implemented protocols and concentrate upon imrpoving performance and security although we comment upon the relevance of the newer protocols.

The requirement in this protocol for redistribution of encrypted data and the generation of shared keys plus the fact that we are fixed with an internet-based client/server model means that we cannot eliminate the need for a server altogether. Hence we propose a cryptographically neutral "Proxy" server whose only purpose is to enable the redistribution of data between clients. Subsequently, it has become host to other necessary elements in the prototype. We also spend some effort upon mitigating the unavoidable trust required in the proxy server by the clients.

Given this design we have constructed a prototype of a platform for conducting web analytics over a set of Awless clients. The additional requirements for management of the entire polling process have been worked out in a preliminary manner by using a mixture of automatic and manual control of the process. The proxy server has been implemented and is based upon the core of the WALLIX DataPeps end-to-end encryption service. This was chosen partly because it has all of the required support to run an API-based service on the internet and partly because the ultimate intention at WALLIX for this work is to incorporate the web analytics into DataPeps itself.

Demonstrating the prototype presents some problems in that the clients need a working version of Awless linked to an Amazon AWS account. This is difficult to arrange in practice but we have circumvented the problem by implementing a simple web server which allows actions to be performed upon a preset Awless installation. This allows participants in the poll to access AWS through the website and contribute to a predefined poll on a proxy server also built into the website host.

For the most part, the prototype demonstrator presented here conforms almost exactly to the specification presented in the technical specification document D7.2 [12]. The architecture is fixed for our application in any case and so has not changed at all since the specification was written and updated. For the software

components, we have made some simplifications for the purposes of demonstration and added some minor new components to enable live demonstration of the application in a timely manner. For clarity, we here assume familiarity with the specification.

2.2 Cryptographic Protocol

The DMCFE protocol [3] has four functions, which we summarize briefly here:

- **Setup:** Starting from a type 3 pairing $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ provided with hash functions $\mathcal{H}$ and $\widetilde{\mathcal{H}}$ onto $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, each client randomly generates s_i and $\widetilde{s_i}$, also respectively. Then after redistribution of the $\widetilde{s_i}$ values we compute $\widetilde{dk_1} = \sum_i \widetilde{s_i}$ which with the pairing definition and hash functions becomes the master public key mpk. The encryption key is $ek_i = s_i$ and the secret key is $sk_i = (s_i, \widetilde{s_i})$.
- **Encryption:** $\texttt{Encrypt}(ek_i, x_i, l)$, for encryption key ek_i $(= s_i)$, data value x_i and label l, returns encrypted data $C_{l,i} = \mathcal{H}(l)^{s_i} \cdot g^{x_i}$
- **DKeyGen:** $\texttt{DKeyGen}((sk_i)_i, \mathbf{y})$, for secret keys $(sk_i)_i$ and weighting vector $\mathbf{y}$, we compute distributed key $\widetilde{C_{\mathbf{y},i}} = \widetilde{\mathcal{H}}(\mathbf{y})^{\widetilde{s_i}} \cdot \widetilde{g}^{s_i \cdot y_i}$. We then have to redistribute $\widetilde{C_{\mathbf{y},i}}$.
- **Decryption:** For $\texttt{Decrypt}(dk, l, C)$, for distributed key dk, label l and encrypted data C, we first have to compute $dk_{\mathbf{y}} = \prod_i \widetilde{C_{\mathbf{y},i}} \times \widetilde{\mathcal{H}}(\mathbf{y})^{-\widetilde{dk_1}}$. From $\mathbf{C} = (C_i)_i$ we can compute $g_T^{\alpha} = e(\prod_i C_i^{\mathbf{y}_i}, \widetilde{g}) / e(\mathcal{H}(l), dk)$ where $e(\cdot, \cdot)$ is the pairing relation. Finally, we have to compute a discrete logarithm to access the result $\alpha = \langle \mathbf{x}, \mathbf{y} \rangle$.

This description omits some of the mathematical precision required to accurately describe the protocol but it contains the necessary steps and computations to explain how we have mapped this protocol to our architecture.
Any realistic implementation has to take into account the architecture of our platform but we can state a couple of simple principles here. Firstly, redistribution must occur in two phases, a collection phase where the clients send individual data to the proxy server and a distribution phase where the combined data are returned to the clients. Secondly, the data flow in the protocol is sufficiently flexible that we have some leeway to map individual computations to different phases of the poll to allow maximum efficiency. For example, we can combine the distribution phase of a previous operation with the collection phase of the following one. The mapping from protocol computations to poll phases is not necessarily one cryptographic function per poll phase.
More recent advances in FE include the DSum protocol [4]. This protocol does not require pairings and also does not need to perform a discrete logarithm to generate the decrypted result. It also does not compute an inner product but generates a simple unweighted sum although this is sufficient for our purposes for the data analytics use case. We have implemented this protocol as an option in the WALLIX prototype. This method gives very good computational performance but requires the redistribution of all of the keys to each client. This limits the scalability of the method given the difficulty of performing such a redistribution in an internet-based client-server model. For the remainder of this deliverable we discuss only the DMCFE protocol.

2.3 Support Mechanisms

2.3.1 Partitioning

There is an inherent weakness in this protocol such that all of the keys are required to generate the distributed key and if one key is missing then decryption is not possible. By this we mean that once a client has registered in a poll its key becomes an essential element to enable the decryption. We have some leeway here since we can start the cryptographic protocol as late in the polling cycle as possible. Any missing keys during this initial phase of the poll can simply be ignored without affecting the cryptographic computations. Once we start the protocol, however, we are committed to requiring all of the keys from that point on.

In order that the technique can be viable in practice, where keys may very well go missing, we need a degree of robustness built into the process. The ideal case would be to incorporate notions of handling missing keys into the cryptographic protocol. This is an extremely difficult thing to do and so, for the moment, we are obliged to consider only non-cryptographic solutions to this problem. In fact, the most obvious and simple solution is to partition the clients into disjoint sets and run multiple polls, in other words run a full cryptographic cycle from setup to decryption upon each partition. The idea here is that a missing key will only prevent one partition from decrypting, the remaining partitions will have all the required keys and can proceed to generate their results.

This says nothing about how we might allocate clients to partitions or how we might manage the partitions within the polls in practice. Within the constraints of the FENTEC web analytics application, we have elected to implement this idea as virtual partitions on the same server. This allows us fine control over the allocation of clients and eliminates the need for a coordination layer on top of the partitioning since all of the data is in the same database.

There are also several possible methods we might use to populate the partitions. For example, we could assume that we have a set number of clients and we simply divide them up into a fixed number of partitions. Alternatively, we could fix the partition size and populate as many partitions as required as clients are added to the poll. Currently, we adopt the first policy where the number of partitions and the number of clients are included in the poll configuration data.

2.3.2 Outlier Detection

Another standard technique for dealing with erroneous data is statistical outlier detection. By this we mean data in the result set which do not fit a pattern in the majority of the data. There are, however, numerous different techniques for detecting such anomalies based upon a wide variety of different assumptions. Selecting a method also depends upon a number of factors, for example, whether a model for correct data exists or not. For the purposes of our prototype, we do not have such a model so we use methods with few *a priori* assumptions upon the characteristics of the data.

The ZScore method is a simple technique which involves eliminating normalised values outside of an arbitrary number of standard devations from the mean. It is simple to implement, requires very little *a priori* information and can work on quite small sample sizes. It does, however, assume a normal distribution for the data. The DBScan [5] method is a spatial clustering method which also applies equally on one-dimensional data such as our lists of integers. It is one of the most popular clustering algorithms in the literature and again, there are no underlying assumptions about the data model. Thresholds can be estimated by trial and error.

We have integrated both of these techniques into our prototype implementation at the decryption stage. This works by treating the result of each partition as a single data point so there is a compromise here between the effectiveness of the outlier detection and the number of partitions. Too small a partition size reduces both the security and reliability of the partition result but too few partitions reduce the effectiveness of the outlier detection.

2.3.3 Key Server

The proxy server is a potential weakness in the design of the prototype but is essential for efficient implementation of the data redistribution required by the protocol. In essence, it has to behave as an active key server which does not just store and make available the keys but also performs operations on those keys in order to generate the distributed keys needed by the decryption phase of the protocol. This is necessary for efficiency and security reasons. We could cause each client to download all of the keys and perform the distributed key generation operation themselves but this is neither secure nor efficient and so we are forced to perform this computation on the proxy server.
The security and trust aspects of such a key server are well known and there is significant technology already available in commercial key servers which puts great effort into the identification, distribution and timely update of the stored keys. There are several standards for the operation of key servers including the PKCS format, the X.509 certificate format and the OpenPGP public key format. There are also numerous implementations of key servers including SKS which conforms to OpenPGP and Amazon's own KMS Key Management Service.
The problem with all of these bespoke key servers is that they only deal with keys as individual entities. There is no concept of aggregation of keys either by collecting keys into related groups or performing operations upon groups of keys. The basic function of these key servers is to provide guarantees about the ownership of the keys and to arrange timely update of keys throughout the network (including for example timely revocation of keys). For our FE protocols, however, we do need to perform such operations upon the keys.
While we would like to incorporate the security and trust provided by these key servers into our FE-based systems there is a fundamental performance problem with using them as they are currently implemented. Each client, for instance, in our DMCFE protocol would have to access the key server for each key required by the distributed key generation, that is all of the other clients. This is infeasible for both computations and communications loads. We therefore have to persuade the proprietors of the key servers to provide a service which includes the functionality of the proxy server in our demonstration. This is, unfortunately, outwith the scope of the FENTEC project but could possibly be considered for future exploitation of the technique. Note that there are some cryptographic protocols in the Multi Party Computation field where clients communicate with each other to generate a distributed key in a secure manner. Although such protocols do not need a trusted key server, they are not usually very efficient.

2.4 Polling Process

The necessary, physical phases of the type of internet poll we wish to conduct are as follows:

Configuration Required to tell the Awless clients that they are participating in a poll. The parameters include a description of which statistical data to collect, the cryptographic parameters and the method of controlling the poll, for example the timing parameters such as the start and end times for the phases. We have proposed a configuration server for the poll in order to keep the proxy server as clean as possible (without any cryptographic data on it), but for the current prototype we simply combine this function with the proxy server.

Statistics gathering During the statistics collection phase the clients operate independently of any interference with the poll server other than to periodically check whether the gathering phase has ended. This is to minimize the impact of the poll upon Awless users. Since the values we are collecting are very simple there should be almost no effect upon the performance of Awless.

Selection One problem with the cryptographic protocol is the requirement for having all of the keys in order to perform the decryption. The larger the poll, therefore, the greater the probability of a failure

due to missing keys. Some form of mitigation is therefore necessary. We thus move the cryptography to as late a stage of the polling process as possible and check the availability of keys before starting the cryptographic protocol. This allows us to eliminate a small number of potential scenarios where, for example, the problem occurs during the statistics gathering phase. We would also like to introduce outlier detection into our polling, however, and for this purpose we now partition the clients into disjoint sets.

Data aggregation Once we have decided upon which clients will contribute to the final analysis we have to perform the encrypted data and shared key redistribution. Depending upon the protocol this can entail a significant communications overhead. Once this phase is complete we then have the ability to decrypt the result.

Decryption and dissemination of results The decryption can occur anywhere that the combined data and keys are stored which is currently the proxy server. It is then up to that server to decide who has access to the results although since the clients have all participated in the `Setup` phase they only need the combined values to do the decryption themselves. Due to the potential sensitivity of the results (particularly in the case of degenerate result sets, for example all of the clients return zero values except one which means that the results are actually the data provided by that client) we need to be careful how we distribute access to the results. This may mean additional symmetric encryption.

2.5 Platform

As already mentioned, our architecture is simply the proxy server to control the analytics process and the Awless clients as the system we wish to analyze. In order to implement the full functionality of the web analytics, however, there are several components built into both the proxy server and the clients. These were explained in D7.1 and are reproduced here in Figure 1. Since this design has not been altered for the purposes of the demonstrator we can use it as the basis for our description in this deliverable (with the exception that the proxy server (WP2) has been implemented in the same component as the configuration host (W1a)).

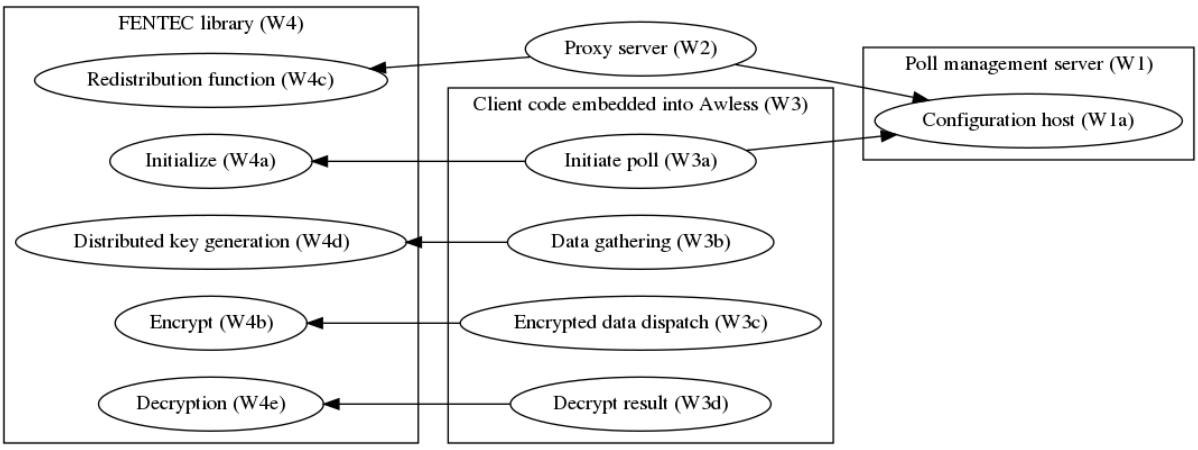

Figure 1: Web Analytics software dependencies between components

The FENTEC library components (W4a, W4b, W4d and W4e) have already been reported in FENTEC deliverable D6.2 [10] (Section 6.1). They correspond exactly to the functions in the DMCFE protocol we

use with an additional component (W4c) to act as a placeholder for the data redistribution function. Note that the library functions provided in the FENTEC library have been refactored slightly from the original definition of the protocol for efficiency reasons but still correspond to the functions we have described in this document.

2.5.1 Proxy server

The proxy server (W2) hosts an API originally intended to be one method for controlling the DataPeps platform. The API endpoints have been rewritten to manage the web analytics platform instead. These endpoints are divided into two types; those to create and delete the poll data itself and those needed to enable the Awless clients to perform data redistribution. There are also some auxiliary endpoints which implement monitoring of the poll progress.
The configuration server (W1a) is currently implemented in the same code base as the proxy server and is a simple two-endpoint section of the API, one to create a poll and another to delete a poll.
Note that the proxy server depends upon an SQL server for persistent data. We do not describe the SQL setup here but for the size of data plus the number of accesses per second the SQL server is required to handle it can be implemented with a small single-instance server and is provided as standard on most cloud-based services, for example Amazon RDS. Future deployments of more large-scale web analytics systems may require a larger capacity and speed of persistent data.

2.5.2 Awless clients

The Awless platform has a complex structure but our required functionality, in terms of statistics gathering, is quite simple. The main problem here is integrating the functions in the cryptographic protocol into the phases of the polling already described. In architectural terms the entire client code is a single unit called from the main dispatch routine for executing Awless commands. This has been done to keep the FENTEC code as distinct as possible from the main Awless code base. In order to distribute a version of Awless we need to provide a custom version of the tool and by keeping the web analytics code separate it makes it easier to track updates to the main branch of the Awless repository.
For persistent data we use the existing database based on BoltDB[1] which was already used to store the internal configuration for Awless. Note that we have not yet implemented secure access to the web analytics poll management server so poll initiation (W3a) consists simply of the Awless user issuing a command to download the poll configuration data and setting up the web analytics database entries.
The data gathering phase (W3b) is similarly hooked into the command loop and has been designed to be as efficient as possible. We use a very simple indexing system based upon the command line interface which is very quick to execute but somewhat limited in scope. For the purposes of demonstration we restrict the statistics gathering to counting accesses to Amazon AWS endpoints although almost for free we can also count errors generated by these accesses. Configuring error counting and also the variance calculation is managed by appending a suffix (`_err` or `_var`) to the label name associated with each command. This allowed us to avoid propagating database access throughout the code.
The encrypted data dispatch phase (W3c) actually requires multiple accesses to the proxy server and so are spread across multiple command line requests and we lose the one-to-one relationship between Awless functionality and web analytics calls.
Finally, the decryption phase (W3d) is implemented in a completely open manner and involves a single access to the proxy server which is linked into an Awless command.
One potential problem is that the clients could be a very heterogeneous collection of computing platforms with widely varying performance. This may mean having a flexible method of managing the polls as they

[1] `https://github.com/boltdb/bolt`

proceed. For the FENTEC demonstrator, however, the time frame for running a poll is extremely short and so we use a simple, manual method for advancing the poll between the phases.

2.6 Software

Both platforms (the DataPeps core of the proxy server and the Awless clients) are written in Golang which at least gives some uniformity to the software.

PHASE	Awless	Request	Response	Endpoint	Action
<ANY>	`create <pollConfig>`	`pollConfig` $\longrightarrow$	$\longleftarrow$	`/define`	Create poll definition in DB and set poll status to `CREATED`.
CREATED	<none>			<none>	No poll actions are possible, system must wait until status is switched to `CONTRIBUTION` either manually or by timer.
CONTRIBUTION	`contribute`	$\longrightarrow$	`pollConfig` $\longleftarrow$	`/contribute`	Add `clientNo` to list of contributors in DB, return poll configuration.
STATISTICS	`status`	$\longrightarrow$	`pollStatus` $\longleftarrow$	`/status`	Return the current status for the poll, phase and numbers of contributors, selected clients and participants
SELECTION	`select`	`hasData` $\longrightarrow$	$\longleftarrow$	`/select`	Client sends a flag indicating whether data is present or not.
COMPLETED	`activate`	$\longrightarrow$	`selectNo, T` $\longleftarrow$	`/activate`	Server returns `selectNo` which is the index into the final selected clients and $\widetilde{dk_1}$ ($= T$ which is in matrix form).
COMPLETED	`send`	$C_{l,i}, \widetilde{C_{y,i}}$ $\longrightarrow$	$\longleftarrow$	`/keygen`	Client executes `Encrypt` and `DKeyGen` and sends results to server which stores them in the DB.
COMPLETED	`decrypt`	$\longrightarrow$	`pollResults` $\longleftarrow$	`/decrypt`	Server checks cache, if results present then they are returned. Otherwise the shared data are extracted from the DB and the `Decrypt` function is executed, results are then stored in cache and returned.

Table 1: Poll phases and actions

Table 1 shows the sequence of events occurring during a normal poll (that is, one which runs normally without errors). Note that the cryptography is not actually deployed until after selection despite the fact that the `Setup` function can be run by all clients (even those not selected) to improve the efficiency. For clients which are not selected the key data is ignored and those clients are returned dummy information during the cryptographic operations. This is to prevent clients from deducing that they have not been selected.
The poll is managed from the Awless command line so a poll is currently created by issuing the command:

```
awless tawa create <url> <name> <poll params>
```

We use the acronym `tawa` to designate the set of sub-commands which relate to the FENTEC polling system in Awless. This stands for "truly anonymous web analytics". In fact, we use this acronym throughout all the FENTEC web analytics code. Note that there is a large list of parameters for this command which is a bit unwieldy for a command line interface.
Awless polling commands are divided into two types, one which requires the address of the proxy server as a command line argument, as in the `create` command here, the others only use the poll name and the address of the server is looked up in the the internal database. The Awless commands typically relate to one endpoint on the proxy server, for example the `create` command is simply a direct call to the `/api/v0/tawa/define` endpoint on the server. The poll name is a mandatory argument to all of these commands but have been omitted from Table 1. Likewise, there are other universal values such as the

response status which are left out. Some phases, such as `COMPLETED` require a more complicated structure and multiple calls to different proxy server endpoints are required.
The `CREATED` status is really just a pending state allowing the poll to be defined before contacting potential participants. The polling actually starts when the phase is switched to `CONTRIBUTION`. Currently, the whole process is open in that any client connecting to the server will be added to the poll until the preset number of contributors is reached. For the demonstrator, however, we have implemented a system whereby the demonstration client is sent an access key in order to participate in a poll. While switching to the next phase at a specific time can be arranged in the poll parameters, there are deadlines for each active phase of a poll, we prefer for demonstration purposes to do this manually with another Awless command:

```
awless tawa force $url $name contribution
```

This forces the phase to move to the next poll and there is logic at the server end to complete housekeeping tasks, such as computing any cryptographic functions, before the switch is made and to prepare properly for the new phase.
The `status` command returns information needed to monitor the progress of the poll:

```
awless tawa status $url $name
```

This returns the current phase name plus the counts of clients which have replied to the calls for contributions, selection and the final upload of encrypted data plus shared keys. This allows the Awless user to tell when the required numbers have been reached and to switch to the next phase enabling complete manual control of the poll.
Apart from these auxiliary request/response functions only the `select` command does not relate directly to the cryptography. This is a phase inserted between the statistics gathering phase and the completion phase and consists of the clients sending their "data present" flag `hasData` which is used to allocate the clients to their index in the actual cryptography itself. The actual selection is carried out when the poll switches to `COMPLETED` mode.
The remaining commands are:

- `activate` which returns the selection index plus the `mpk`,
- `send` which runs the `Encrypt` and `DKeyGen` functions on the client and sends the values for storage on the server and
- `decrypt` which combines the shared data and runs the `Decrypt` function to generate the results.

2.6.1 Server API

The full API for the proxy server is as follows:

```
func initTawaHandlers(c *config.Config, router *restflect.Router, h *handlers.Sessions) {
  // Create, delete and monitor polls
  router.MustHandleChain("/api/v0/tawa/define",     "POST", h.ChainLogTx, h.TawaDefine)
  router.MustHandleChain("/api/v0/tawa/delete",     "POST", h.ChainLogTx, h.TawaDelete)
  router.MustHandleChain("/api/v0/tawa/status",     "POST", h.ChainLogTx, h.TawaStatus)
  // Main poll and crypto phases
  router.MustHandleChain("/api/v0/tawa/contribute", "POST", h.ChainLogTx, h.TawaContribute)
  router.MustHandleChain("/api/v0/tawa/select",     "POST", h.ChainLogTx, h.TawaSelect)
  router.MustHandleChain("/api/v0/tawa/activate",   "POST", h.ChainLogTx, h.TawaActivate)
  router.MustHandleChain("/api/v0/tawa/keygen",     "POST", h.ChainLogTx, h.TawaKeygen)
  router.MustHandleChain("/api/v0/tawa/decrypt",    "POST", h.ChainLogTx, h.TawaDecrypt)
  // Auxiliary endpoints for debug
  router.MustHandleChain("/api/v0/tawa/posterror",  "POST", h.ChainLogTx, h.TawaPostError)
  router.MustHandleChain("/api/v0/tawa/force",      "POST", h.ChainLogTx, h.TawaForce)
}
```

The individual endpoints have the following functionality:

2.6.1.1 Create poll The type below is the create poll call parameter structure in `protobuf` format. There are quite a lot of parameters required to define a poll, these are explained in Table 2. Note that for `create` the only response is an `OK` flag which indicates success plus an error message upon failure. The action taken at the server upon this call is to simply create a database entry with the given poll parameters and set the phase for that poll to the initial poll state. Some verification is applied to the parameters.

```
enum TawaCryptoType {
     DMCFE_BN256 = 0;
     DMCFE_WR_E1 = 1;
     DMCFE_WR_E2 = 2;
}

enum TawaStatus {
    NULL = 0;
    CREATED = 1;
    CONTRIBUTION = 2;
    STATISTICS = 3;
    SELECTION = 4;
    COMPLETED = 5;
    FAILED = 6;
}

enum TawaSelectionType {
    MOSTSIGNIFICANT = 0;
    RANDOMSAMPLE = 1;
}

message TawaDefineRequest {
    string Name = 1;
    int32  Status = 2;
    repeated string Commands = 3;
    string Start = 4;
    string Contribution = 5;
    string Statistics = 6;
    string Selection = 7;
    uint32 MaxClients = 8;
    uint32 NumClients = 9;
    uint32 Bound = 10;
    bool AllowNullData = 11;
    bool Manual = 12;
    TawaSelectionType SelectionType = 13;
    int32 Partition = 14;
    TawaCryptoType CryptoType = 15;
    bytes P = 16;
    bytes Gen = 17;
}

message TawaDefineResponse {
    bool OK = 1;
}
```

Parameter	Description
`Name`	The name of the poll to create.
`Status`	The starting phase for the poll (usually `CREATED` or `CONTRIBUTION`. The `NULL` phase simply means that the poll has not been defined and is only used for status query responses.
`Commands`	List of AWS endpoints for which to gather statistical information using the standard naming scheme.
`Start`	For automatic poll timing, this is the date and time at which to end the `CREATED` phase and switch to `CONTRIBUTION`.
`Contribution`	Length of time to remain in the `CONTRIBUTION` state.
`Statistics`	Length of time to remain in the `STATISTICS` state.
`Selection`	Length of time to remain in the `SELECTION` state.
`MaxClients`	Total number of clients allowed to contribute to the poll, ie. before selection.
`NumClients`	Number of clients to select for the poll.
`Bound`	Crypto parameter, bound for the integers for the `DMCFE_BN256` protocol.
`AllowNullData`	Boolean flag, when set allows clients with no data to contribute to the poll.
`Manual`	Boolean flag, override the timing parameters above and assume that the `force` command will be used to switch between polling phases.
`SelectionType`	Indicates the method of selection, either according to statistical significance or a purely random sample.
`Partition`	The number of partitions to divide the clients up into.
`CryptoType`	Protocol to use for the functional encryption. Several protocols are possible but we only use the `DMCFE_BN256` protocol for now.
`P`	Crypto parameter, the base prime for the `DMCFE_WR_E1` and `DMCFE_WR_E2` protocols.
`Gen`	Crypto parameter, the group generator for the `DMCFE_WR_E1` and `DMCFE_WR_E2` protocols.

Table 2: Create request parameters

2.6.1.2 Delete poll For the delete poll endpoint the only request parameter is the name of the poll and the only result is the `OK` status. The poll database entry is deleted along with *any other associated data*. This includes the results of the poll.

```
message TawaDeleteRequest {
    string Name = 1;
}

message TawaDeleteResponse {
    bool OK = 1;
}
```

2.6.1.3 Status of poll The `status` call takes the name of a poll and returns the current poll phase plus the numbers of responses from each of the data collection phases, these include contributions, selections, participants (in the crypto itself) and `Ciphers` which is the number of clients which have returned their encrypted data.

```
message TawaStatusRequest {
    string Name = 1;
}

message TawaStatusResponse {
    int32  Status = 1;
    uint32 Contributors = 2;
    uint32 Selected = 3;
    uint32 Participants = 4;
    uint32 Ciphers = 5;
}
```

2.6.1.4 Contribute The request for signaling intention to contribute is simply the poll name. The server responds, however, with a sufficient number of parameters for the polling to proceed on the client including the cryptographic parameters. For the most part these values correspond to the poll creation parameters listed in Table 2 except that the `contribute` response also includes the current poll phase plus a client number to be used for subsequent communications. The server functionality for this call is to increment the current count for contributors.

```
message TawaContributeRequest {
    string Name = 1;
}

message TawaContributeResponse {
    string Name = 1;
    int32  Status = 2;
    repeated string Commands = 3;
    uint32 ClientNumber = 4;
    uint32 MaxClients = 5;
    uint32 NumClients = 6;
    int32  CryptoType = 7;
    int32  Partition = 8;
    uint32 Bound = 9;
    bytes P = 10;
    bytes Gen = 11;
}
```

2.6.1.5 Selection For selection, the client checks if data is present for each endpoint count and sets `HasData` to be `true` if any of them are non-zero. The significance value is computed from the endpoint counts in order to penalize zero values (`total` $*$ `total` $*$ (`nocmds` $-$ `zeros`))/`nocmds`, where `total` is the total count for all endpoints, `nocmds` is the number of endpoints that have been monitored and `zeros` is the number of endpoint counts which equal zero). These values are sent in cleartext so we need to confirm with the clients that it is acceptable to provide this information unencrypted.
At this point, the server only records the number of clients selected plus their statistical summary. The selection itself is actually performed when the poll phase is switched from `SELECTION` to `COMPLETED`. Note that for the `DMCFE_WR_E1` protocol we also need to return the cryptographic parameters computed during the setup.

```
message TawaSelectEncryptedCount {
    string Command = 1;
    bytes U = 2;
    bytes U2 = 3;
    bytes C = 4;
    bytes C2 = 5;
}

message TawaSelectPubKeys {
    string Command = 1;
    bytes PubKey = 2;
    bytes PubKey2 = 3;
}

message TawaSelectRequest {
    string Name = 1;
    uint32 ClientNumber = 2;
    bool HasData = 3;
    int32 Significance = 4;
    repeated TawaSelectEncryptedCount Counts = 5;
    repeated TawaSelectPubKeys PubKeys = 6;
}
```

2.6.1.6 Activation The `activate` call is the first true cryptographic operation for the FE. Activation consists of the clients again signaling willingness to continue with the poll and in return receive the necessary parameters to perform encryption. Note that clients which have not been selected receive dummy information. This is so that non-selected clients do not actually know that they have not been selected. However, this varies between protocols since some protocols require the selection number to perform encryption. Apart from this, the `activate` call returns the $\widetilde{dk_1}$ value (`PubKeys`) for the `DMCFE_BN256` protocol or the combined `U` values for the `DMCFE_WR_E1` protocol. The `Status` and `Commands` values are included for convenience. The functionality of the `activate` call is to compute the master public key and keep track of which clients have responded.

```
message TawaActivateRequest {
    string Name = 1;
    uint32 ClientNumber = 2;
}

message TawaUs {
  string Command = 1;
  repeated bytes Us = 2;
  repeated bytes U2s = 3;
}

```

```
message TawaPubKeys {
  string Command = 1;
  repeated bytes PubKeys = 2;
  repeated bytes PubKey2s = 3;
}

message TawaActivateResponse {
    string Name = 1;
    int32 Status = 2;
    repeated string Commands = 3;
    uint32 ClientNumber = 4;
    uint32 SelectNumber = 5;
    uint32 NumClients = 6;
    uint32 PartitionSize = 7;
    repeated TawaUs us = 8;
    repeated TawaPubKeys PubKeys = 9;
}
```

2.6.1.7 Key generation The `keygen` call allows the clients to upload their encrypted counts (`EncryptedCount`) and generated distributed keys (`KeyShare` or `M`) to the server. The encrypted squared counts are wrapped together with the original values for efficiency. The server does not perform any computation at this point, the shared values are merely stored in the database.

```
message TawaKeygenRequest {
    string Name = 1;
    uint32 ClientNumber = 2;
    string Label = 3;
    bytes EncryptedCount = 4;
    repeated bytes KeyShare = 5;
    bytes EncryptedCountSq = 6;
    repeated bytes KeyShareSq = 7;
    bytes M = 8;
    bytes MSq = 9;
}

message TawaKeygenResponse {
    bool OK = 1;
}
```

2.6.1.8 Decryption The `decrypt` call is open to all clients. The first client to request decryption triggers the `Decrypt` function, the results of which are then stored into the database for subsequent calls. The response is a list of poll results, one for each partition for each endpoint monitored during the statistics gathering phase. A result consists of the partition number, partition size, command name, the date and time that the poll was completed, the sum of access counts plus the sum of the squares of these values and also the type of cryptography used to perform the FE. The caching is essential because this call is computationally heavier than the other cryptographic-related calls. All of the shared keys and encrypted data have to be collected from the database and combined into the requisite values for the `Decrypt` call. If a poll has failed, because of missing or erroneous values then this is the call where the error shows up. This is greatly mitigated by the presence of partitions but currently, it us up to the client to perform any bad partition checking on the raw partition data. This explains the presence of the partition size in result since this is needed by the outlier detection routines.
The average values can be computed from:

$$\texttt{SumAccesses}/\texttt{NumClients}$$

and the variance can be computed from:

$$(\texttt{SumSqAccesses} - (\texttt{SumAccesses} * \texttt{SumAccesses})/\texttt{NumClients})/(\texttt{NumClients} - 1.0)$$

```
message TawaDecryptRequest {
    string Name = 1;
}

message PollResult {
    uint32 Partition = 1;
    uint32 PartitionSize = 2;
    string Command = 3;
    string PollDate = 4;
    string SumAccesses = 5;
    string SumSqAccesses = 6;
    int32 CryptoType = 7;
    string Msg = 8;
}

message TawaDecryptResponse {
    bool OK = 1;
    repeated PollResult Results = 2;
}
```

2.6.1.9 Post error into error log The `posterror` call allows clients to log errors into the servers' error logs. This feature has limited use since many potential errors results in failure of the client/server connection. It has been included for the rare circumstance in which an error on a client during a poll needs to be recorded on the server for subsequent analysis. It will be more useful for actual live polls than for the demonstrations given so far.

```
enum TawaSeverity {
    Debug = 0;
    Info = 1;
    Notice = 2;
    Warning = 3;
    Error = 4;
    Critical = 5;
}

enum TawaErrorKind {
    PollInternalError = 0;
    InsufficientParticipants = 1;
    PollInformation = 2;
    PollManagerError = 3;
    PollProtocolError = 4;
    PollComment = 5;
    PollClientError = 6;
}

message TawaPostErrorRequest {
    string stamp = 1;
    string name = 2;
    uint32 source = 3;
```

```
    TawaSeverity severity = 4;
    TawaErrorKind kind = 5;
    string errortext = 6;
}

message TawaPostErrorResponse {
    bool OK = 1;
}
```

2.6.1.10 Force poll status The force poll call is not intended to be part of the normal operation of the polling process. It is present to enable manual control of the process by allowing the phases to be switched externally. For this to work, the poll must be put in manual mode (see Table 2). When the phases are switched the normal polling operations associated with that change are called. The parameters for this call include the desired poll `Status` value and, if successful, the response will include the actual new state.

```
message TawaForceRequest {
    string Name = 1;
    int32 Status = 2;
}

message TawaForceResponse {
    int32 Status = 2;
}
```

2.6.2 Awless code

2.6.2.1 Internal database Awless has an internal database for configuration data but we keep the web analytics data separate. The internal DB structure is almost exactly the `pollConfig` data provided by the server but with a few exceptions:

```
type PollDefinition struct {
  // Poll data
  Name           string
  Status         api.TawaStatus
  ServerUrl      string
  Commands       []string
  Start          time.Time
  Contribution   time.Duration
  Statistics     time.Duration
  Selection      time.Duration
  RemoteLogging  bool
  AllowNullData  bool
  Manual         bool
  SelectionType  api.TawaSelectionType
  Partition      int
  PartitionSize  int

  // Crypto data
  ClientNumber int
  SelectNumber int
  MaxClients   int
  NumClients   int
  CryptoType   api.TawaCryptoType

  // Needed by BN256
  Bound    int
  CPK      []CPubKeys
  PubKeys  []api.TawaPubKeys

  // Needed by WR_E1
```

```
  P    big.Int
  Gen  big.Int
  CRS  []CRS
  Us   []api.TawaUs
}
```

We store the address of the server along with the poll configuration data. General commands (such as `create`) always take the server URL from the command line but any polling-related commands use the address stored in the DB. The `RemoteLogging` flag allows an Awless user to disable sending error messages to the server. Finally, we also store the `ClientNumber` and `SelectNumber` values as well. These are updated during polling rather than being provided during configuration.

2.6.2.2 Statistics gathering

Currently, we implement gathering of mean and variance values for counts of accesses to endpoints on the Amazon AWS service. The recording process is simple and is integrated into the main function dispatch routine in Awless:

```
/** RecordCommand
 * Increment the access count for a given command.
 */
func RecordCommand(db *database.DB, cmd []string) error {
  isStatistics, err := CheckStatusRequiredCPD(db, api.TawaStatus_STATISTICS)
  if err != nil {
    return err
  }
  if isStatistics {
    var index string
    var cnt int
    if index, err = indexFromCommand(cmd); err != nil {
      return err
    }
    err = db.SetStringValue(RecordCommandIndexKey, index)
    if err != nil {
      fmt.Printf("RecordCommand: can't store index %s\n", index)
    }
    if !indexIsMonitored(index) {
      return nil
    }
    if cnt, err = db.GetIntValue(strconv.Itoa(clientIndex) + index); err != nil {
      cnt = 0
    }
    cnt = cnt + 1
    return db.SetIntValue(strconv.Itoa(clientIndex)+index, cnt)
  }
  return nil
}
```

The counts are zeroed when Awless connects to the proxy server and are incremented each time a command is issued (`cmd` in the above code) which has an index which matches one of the `Command` values in the poll configuration. Note that we prefix the index with the client number in order to partition the counts for multiple clients in the same Awless instance. This is a debug feature which is also used to enable the demonstrator.

At the point of performing the functional encryption `Encrypt` function, we have the total number of accesses to each monitored endpoint. We can then encrypt these values according to the shared key provided by the proxy server:

```
// Currently we simply set the y values to 1 (average only).
y := data.NewConstantVector(cPD.NumClients, big.NewInt(1))

// Encrypt the values and turn into byte arrays
encCntBytes, keyShareBytes, err = common.EncryptValue(cPD.SelectNumber, cnt, idx, T, y)
if err != nil {
  return err
}
```

```
if isVar {
  encCntSqBytes, keyShareSqBytes, err = common.EncryptValue(cPD.SelectNumber, cnt*cnt, idx, T, y)
  if err != nil {
    return err
  }
} else {
  encCntSqBytes = []byte{}
  keyShareSqBytes = [][]byte{}
}
```

The `isVar` flag is computed from the index and indicates that we wish to encode the square of the count at the same time as the count. Computing the mean and variance from the counts and squared counts is very simple:

```
mean := sum / n
var variance float64
if n == 1 {
  variance = 0.0
} else {
  variance = (sumsq - (sum*sum)/n) / (n - 1.0)
}
```

where `n` is the number of clients, `sum` is the total count for one endpoint and `sumsq` is the sum of squares for the same endpoint. In terms of the statistics we perform for FENTEC this simple result is adequate and could be extended to include other values, such as counts of errors or even system parameters such as maximum memory occupancy. Higher statistical moments should also be possible but are not particularly useful.

2.6.2.3 Awless commands

The relationship between the user-visible commands and the functioning of the poll is as follows:

- The `create` command has the following arguments:

  ```
  awless tawa create URL NAME MAXCLIENTS NUMCLIENTS COMMANDS START_TIME CONTRIBUTION_DURATION \
                     STATISTICS_DURATION SELECTION_DURATION ALLOW_NULL_DATA MANUAL \
                     SELECTION_TYPE PARTITION CRYPTO_TYPE (<BOUND> | <PRIME> <GENERATOR>)
  ```

  These parameters relate exactly to the parameters described in Table 2. The only action awless takes upon this command is to call the `create` endpoint at the server at the given URL. The client creating the poll is not added to the poll, it must perform the normal `contribute` operation. The format of the arguments depends upon which type of cryptography is used.

- The `delete` command is very simple:

  ```
  awless tawa delete URL NAME
  ```

  The `delete` endpoint is called at the given URL and the poll with the given name is deleted. Note that *all* of the poll-related data is deleted, including the poll result, if present.

- The `status` command displays the current poll status.

  ```
  awless tawa status [--phaseCount int] URL NAME
  ```

  If the `-phaseCount` flag is missing then just the phase name is printed, otherwise the output is an integer value for one of the count values. This is used to implement scripts for running a poll under manual mode.

- The `force` command forces the status of the named poll to the supplied value and is intended for manual control of polling.

```
awless tawa force URL NAME STATUS
```

Again, this command is not treated as poll-specific so the URL address is required. Any poll phase can be given since this command is really for debug purposes but it only makes sense for the normal sequence of phases.

- The `contribute` command tells the server that the client wishes to contribute to a poll and returns the poll configuration data. This is the first point of entry into the poll so the URL address is required.

  ```
  awless tawa contribute URL NAME
  ```

  This command is only valid if the poll is currently in the `CONTRIBUTE` phase. The Awless client therefore issues a `status` command to update the local copy of the phase status. Apart from updating the configuration data the counts for the statistical gathering code are all reset.

- The `select` command is only valid during `SELECTION` mode.

  ```
  awless tawa select URL NAME
  ```

  Yet again the action carried out by Awless is to compute the necessary statistical summary and call the `select` endpoint on the server.

- The `send` command is more complicated for the Awless clients. Since the results of the redistributed `Setup` data are required for encryption the `send` command performs the `activate` and `keygen` calls in a single operation.

  ```
  awless tawa send
  ```

  The URL provided by the `contribute` command is used. The `activate` call returns the combined encryption key ($\widetilde{dk_1}$) which is then used to encrypt the statistical data. The distributed key ($\widetilde{C_{y,i}}$) is then generated and the two are combined in a single `keygen` call which stores these values on the server.

- The `decrypt` command triggers the `Decrypt` function on the server which is assumed to have gathered the necessary data to perform it. This command only works in `COMPLETED` mode and after all the encrypted data and shared keys have been uploaded.

  ```
  awless tawa decrypt URL NAME
  ```

  The server `decrypt` call only returns the counts for each endpoint monitored, the Awless client converts these into average and variance and generates a suitable display.

- The `comment` command allows the user to insert a message into the server's logs.

  ```
  awless tawa comment URL NAME COMMENT
  ```

- The `set` command allows control over the local behaviour of the Awless client's web analytics functions. Currently only the `RemoteLogging` facility can be switched on and off.

  ```
  awless tawa set PARAMETER VALUE
  ```

In general, the functionality in the client code has been kept as simple as possible. However, the process could be made more automatic. The current design was mainly to enable experimentation with the various options for deploying the cryptographic protocol. Future versions of the client code will mostly just involve enabling the polling process and then letting it run automatically in the background.

2.7 Demonstrator

To present the techniques proposed by the WALLIX web analytics use case in short period of time presents some difficulties. Firstly, the time scales for significant statistics gathering are far too long for demonstration purposes. Secondly, our application requires access to both Amazon AWS services plus a configured and operational version of the WALLIX Awless tool. Finally, very little is visible during the polling process apart from the presentation of the final result.
To overcome these problems and allow some visible presentation of how the web analytics is supposed to work, we have devised a simple demonstration framework in which a small-scale poll is conducted locally. The proxy server has been deployed in an open manner to allow connection via a locally-installed Awless client. The Awless client is then front-ended by a simple web page allowing AWS operations to be conducted by proxy through the single Awless installation. The individual clients are identified by the web server and the AWS name-spaces are partitioned by prefixing the name-space with a value unique to each connected participant.

2.7.1 Demonstration architecture

The demonstration core is simply a web server implemented alongside the proxy server which provides two pages, a page for emulating the Awless commands to be counted and a page used to manage the polling process. The Awless emulation page is a simple form-driven page with no client-side code and implements a simple protocol for translating requests to create, delete and list users into actual Awless commands on the server. The management page operates in a similar manner and allows the creation, deletion and management of polls on the proxy server. This situation is illustrated in Figure 2.
To conduct a demonstration, a small number of volunteers are asked to log into the Awless emulation web page and execute random commands using the demonstration web page. Security is provided by a simple token system which has to be generated on the server and sent to the participant. Four commands are possible:

- Create a user on the AWS account.
- Delete a user from the AWS account.
- List the users currently defined.
- Display the results of the FE decryption.

The last option is only available when all of the participating clients have returned their encrypted data. The poll management system displays a count of the current number of connections being counted (for example, the number of participants during the CONTRIBUTION phase). The poll management system also allows four commands:

- Update the current information (including the displayed count).
- Create a poll at the proxy server.
- Delete a poll from the proxy server (including previous FE results).
- Advance the poll to then next phase.

The final command only appears during a phase which requires manual intervention. The poll is run according to the following procedure:

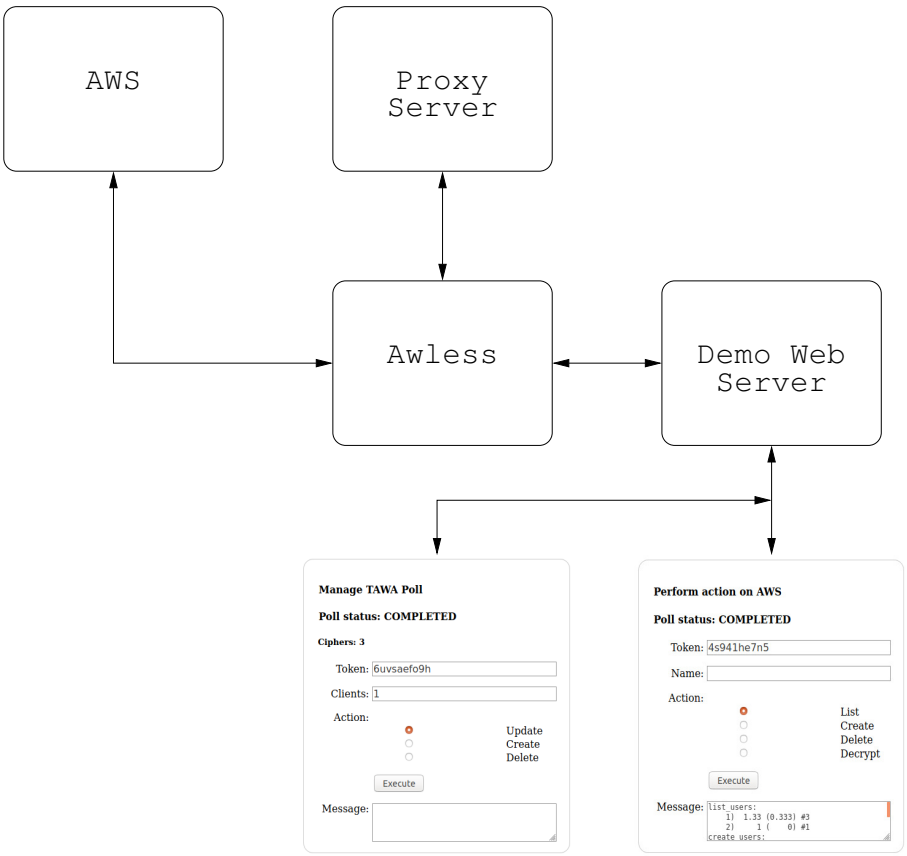

Figure 2: Architecture of the FENTEC Web Analytics Demo

- A poll is created on the proxy server.
- The poll is manually switched from `CREATED` to `CONTRIBUTION`.
- Each client performs any command and the `contribute` Awless call is performed automatically.
- Once all of the designated clients have called `contribute` the proxy server automatically switches to `STATISTICS`.
- For a short period of time each of the users runs a few random commands to generate some statistical data.
- The poll is manually switched from `STATISTICS` to `SELECTION`.
- Each user then causes the client to execute another random command and triggers the `select` function from Awless.
- Once all of the clients have executed `select` the proxy server automatically switches the poll to `COMPLETED`.
- Each user then executes one more command which causes the Awless client to execute the `send` command to trigger sending of the encrypted data.
- The `Decrypt` command becomes available on the demonstration web page and can be used to trigger the `decrypt` function and display the results.

2.7.2 Demonstration security

For legal reasons we are not allowed to use real data for this demonstration. What we are using here, however, is an expired free-tier AWS account which does not allow any paying actions on the AWS servers. For example, it does not allow the user to deploy a server online which, depending upon the type and size of server deployment can be a very expensive operation. In fact, the free account allows the users to create and delete user names but they are completely blank. They have no policies associated with them, cannot be used to log into AWS services and are deleted after the demonstration ends.
Note that we cannot prevent, for example, a user from accidentally typing their bank account data in as a user name and creating a user with this name on AWS servers but even this situation is relatively harmless since the only action which can be carried out is to display the name at the original user's web page. No other clients will see this data.
Similarly, the SQL server only stores the poll data (for example, the cryptographic parameters) plus the temporary data needed to run the FE. None of this data is user identifiable and in any case is deleted as soon as the poll is deleted after the demonstration has ended.
We thus contend that this demonstration meets the legal restrictions placed upon the FENTEC project by its contract with the European Union.

2.7.3 What does the demonstration prove?

In this demonstration we have shown that Functional Encryption can be used to perform statistical analysis in a useful manner. We have investigated the infrastructure necessary to deploy the cryptographic protocol and manage the intermediate data in a secure manner. Our design proved to be quite accurate in this respect since the client/server model imposed upon us matches the proxy server/Awless client combination.
We have proved that for Awless we can compute the mean and variance of integer values gathered from a set of distinct clients. We were able to perform some optimization of the process by merging request/response

pairs and also reduced the communications burden by colocating multiple values in the same messages, for example the sum values and the squared sum values can be transmitted together.
Our demonstrator proxy server is actually deployed on AWS. This gives some evidence that the system can be easily deployed on cloud platforms. In fact, if we were able to use bespoke key servers for this function we would not need to deploy our proxy server at all and the code would then reside exclusively within the Awless platform.
A degree of robustness in the method has been demonstrated by the partitioning technique which we also use to implement simple forms of outlier detection. From practical experience with the demonstrator we have found that missing keys can actually occur quite frequently, even with small sample sizes. It is rare, however, for multiple keys to be missing so we only require small numbers of partitions to gain a considerable degree of robustness in the polling process.
In summary, our demonstration meets the key security requirements from WALLIX's point of view. The users' data are not visible to the poll manager (in this case, the proxy server) and the users are not able to access each others' data unless by deliberate collusion.

2.7.4 What does the demonstration not prove?

Although the cryptographic protocol is sound and backed up by a proof of semantic security we still have to verify the security of the implementation. Since our implementation is not quite how we would design it for use in a real-life poll with potentially sensitive data, this validation would more usefully be carried out on a version more intended for a large-scale poll on the internet. We have, however, performed a partial audit of the security of the system, this will be reported in deliverables D7.10 [16] and D7.12 [17].
We can compute very accurately the communications sizes for a real deployment but the actual transmission times may vary widely depending upon the amount of resources used to implement the servers. We are also hampered somewhat by the fact that our clients will be from a widely-varying set of communications performances. Without a full-scale deployment we cannot make accurate estimates of communications times.
Similarly, we can measure the performance of our prototype and perform extrapolation onto a potential large-scale poll but there are errors inherent in this process. In this case, our main problem is that we cannot easily scale our prototype into any realistic size. Each client is supposed to be an Awless instance linked to a real AWS account performing a wide variety of tasks in order to spot potential improvements which is our fundamental motivation for using FE technology.

2.7.5 Difference between demonstrator and actual working poll system

In order to build a full-size working polling system we would have to modify our demonstrator accordingly:

- The poll management technique could be improved. Specifically, the client code should be automated as much as possible. Setting hard deadlines for each phase has turned out to be quite inflexible so we require more intelligent decisions as to when the phases can be switched, for example we could delay deadlines until the required counts are arrived at. Ultimately, we will have to retain some element of manual override for the system in the event of unforseen circumstances. It may be possible, for example, to regress the phase in order to redo a failed operation but then cryptographic integration becomes more difficult.

- The prototype does not implement full authentication for the proxy server. DataPeps has an authentication mechanism for registered users. This was not included in the prototype server code but could be used in a full deployment to secure access to the poll and its results. Unfortunately, the mechanism is rather complicated and integrated into DataPeps security.

- Currently, any client can access the poll results. It is possible to envisage circumstances in which the poll results themselves can be considered sensitive information. With proper authentication built into the poll it would be simple to implement a system whereby only selected clients were able to access the results of the poll. It is also possible to provide access to poll results for external entities. Due to the nature of the cryptographic protocol the proxy server will always have access to the results an so can decide who gains access to them.

3 Conclusion

In this deliverable, we presented the final version of the WALLIX web analytics use case prototype. For this round of development we concentrated upon improving the reliability and performance of the prototype. The functionality has already been shown to match the functional requirements by the interim report D7.3 [13].

The first major improvement addressed the fragility caused by the requirement to have all of the distributed keys in order to perform decryption. Our experiments with the prototype plus online demonstrations showed this to be a major weakness in the method. There are no efficient cryptographic solutions to this problem so we adopted a simple partitioning method to improve reliability. There were several possible ways of deploying this method but for an initial investigation we simply built it into the existing server code. This gave an immediate benefit in terms of the reliability since if a shared key is missing it only prevents the affected partition from being decrypted leaving the other partitions as valid data. With the right choice of parameters for the partitions, our use case has a reasonable security level, mainly due to the lack of any dependency between the partitions.

A fringe benefit is that we were also able to address problems of spurious or malicious data by implementing outlier detection across partitions. Given a sufficient number of partitions we are able to detect partition values which are outwith the range of values present in the other partitions. This is a very narrow and simple implementation of anomaly detection but has proved effective for our application.

Finally, we can propose what will be a significant performance improvement by implementing partition processing in a concurrent manner. We do not have the resources to test this hypothesis, however, and the technique will also require the implementation of a coordination layer above the partitions in order to collate the data from partitions whose results will reside upon different servers. This method is possible because of the lack of dependencies between partitions and so we can deploy as many servers as we require provided we have the resources to do so.

Note that we have integrated the partitioning scheme into our demonstrator for the prototype although the presentation of the outlier detection results is somewhat primitive. If we plan to do further demonstrations we will have to update the presentation but we mainly wanted it for the improvement in reliability.

We discussed the possibility of using bespoke, commercial key servers to implement the proxy server in a more trustworthy manner. The idea is to access the technology which already exists for the widespread distribution of (usually asymmetric public keys) throughout a network in a trustworthy, identifiable and timely manner. Although this has desirable characteristics for FE implementations, the current technology does not have the functionality to implement the distributed key generation required by, for example, our DMCFE protocol. We carried out some initial implementation work but the inefficiencies introduced by having to access the key server for each individual key were prohibitive. Perhaps with some advocacy in the key server community they could be persuaded to adopt this technology but this is outwith the scope of the FENTEC project.

Although new protocols have become available during the lifetime of the project, for example those described in deliverable D4.3[15], they mostly have the same weaknesses as the preliminary version, that being the weakness caused by the requirement to have all of the keys in order to perform decryption. Despite that there are benefits from these protocols such as improved security assumptions (even going as far as quantum-safe assumptions [1]) and performance improvements (such as not requiring the discrete logarithm computation to access the results [2]) as in the DMCFE protocol presented in this deliverable. For any future integration of FE protocols into other application domains, however, we would probably recommend one of these improved schemes. Our comments upon the existing WALLIX use case implementation will remain valid.

Although we defer performance testing to the remaining deliverables, D7.10 and D7.12, we can say with confidence that it already meets the performance requirements for the specific application which we

described in Section 2.1.
In summary, the functionality of the web analytics prototype is satisfactory and potentially useful in real-world applications. We were able to make the process robust enough for situations where keys and data may be missing and we have been able to demonstrate this with our prototype demonstrator. There may be some questions about the scalability of the method to larger polls than that required by the WALLIX use case but we have some circumstantial evidence that we would be able to meet much larger performance requirements given sufficient resources. Overall, we can recommend this type of FE for gathering statistical information on the internet where a high degree of security is required to protect users' data.

4 Next steps

Until the end of the FENTEC project, our priorities are as follows:

- Finalize performance and security analysis.
- Build a business plan and commercial strategy for exploitation of the results.

After the FENTEC project ends we can consider the following activities:

- Integrate the web analytics into the WALLIX DataPeps product.
- Perform an actual poll with WALLIX clients in order to fully evaluate the performance of the protocols.

References

[1] Michel Abdalla, Fabrice Benhamouda, and Romain Gay. From single-input to multi-client inner-product functional encryption. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 552—-582, Kobe, Japan, December 2019. Springer, Heidelberg, Germany.

[2] Michel Abdalla, Fabrice Benhamouda, Markulf Kohlweiss, and Hendrik Waldner. Decentralizing inner-product functional encryption. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 128—157, Beijing, China, April 2019. Springer, Heidelberg, Germany.

[3] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. In Steven Galbraith and Thomas Peyrin, editors, *Advances in Cryptology — ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 703–732. Springer, 2018.

[4] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Multi-client functional encryption with repetition for inner product. Cryptology ePrint Archive, Report 2018/1021, 2018. `https://eprint.iacr.org/2018/1021`.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD-96 Proceedings*, pages 226–231. AAAI Press, 1996.

[6] FENTEC. D3.1 technical requirement report analysis. Technical report, European Commission, 2018.

[7] FENTEC. D4.1 annual report on functional encryption schemes for prototypes y1. Technical report, European Commission, 2018.

[8] FENTEC. D7.1 preliminary specification of fentec prototypes. Technical report, European Commission, 2018.

[9] FENTEC. D4.2 annual report on functional encryption schemes for prototypes y2. Technical report, European Commission, 2019.

[10] FENTEC. D6.2 preliminary functional encryption toolset api. Technical report, European Commission, 2019.

[11] FENTEC. D7.11 performance report for fentec prototypes after first cycle. Technical report, European Commission, 2019.

[12] FENTEC. D7.2 final specification of fentec prototypes. Technical report, European Commission, 2019.

[13] FENTEC. D7.3 first version of the truly anonymous data collection prototype. Technical report, European Commission, 2019.

[14] FENTEC. D7.9 first test report of the fentec prototypes. Technical report, European Commission, 2019.

[15] FENTEC. D4.3 annual report on functional encryption schemes for prototypes y3. Technical report, European Commission, 2020.

[16] FENTEC. D7.10 final test report of the fentec prototypes. Technical report, European Commission, 2020.

[17] FENTEC. D7.12 final performance report for fentec prototypes after second cycle. Technical report, European Commission, 2020.