NEXt generation Techno-social Legal Encryption Access and Privacy nextleap.eu

Grant No. 688722. Project started 2016-01-01. Duration 36 months.

DELIVERABLE D5.4

Privacy-enhanced Analytics for Wisdom of the Crowds Open Source Module

George Danezis and Marios Isaakidis

University College London

Beneficiaries: **UCL** (lead), MER,EPFL

Workpackage: **Open-Source Code and User Validation**

Description: A module producing code for private information retrieval for crowd-sourcing analytics will be produced.

Version: Draft

Nature: Report (R)

Dissemination level: Public (P)

Pages: 5

Date: 2019-1-22

Project co-funded by the European Commission within the Horizon 2020 Programme.

1 Introduction

This deliverable D5.4 is a companion of D2.4 – that described the theory – and D4.5 – that describes the evaluation – and summarizes the open-source software developed as part of NEXTLEAP to support our work on 'Wisdom of the crowds'. The description for D5.4 from the project proposal defines it as: "D 5.4 Privacy-enhanced Analytics for Wisdom of the Crowds Open Source Module (Editor: UCL) [Due: M36] A module producing code for private information retrieval for crowd-sourcing analytics will be produced. Since the module will be used primarily to ask questions of users, a "lean design" round will not be needed proof-proving results."

Since the nature of this deliverable is to describe open source code, we provide within it a short description of the open source code developed, the technology and lines of code produces (LOC) and how it contributed to the project. All our code is open, and has been developed in the open, and can be downloaded or cloned from `github.com`. We chose this service to host our project since it is central to the open-source community, and provides facilities for bug tracking, project planning, as well as advanced revision control and collaboration work-flows.

As part of this deliverable, and in support of D2.4 and D4.5 we describe three categories of code artefacts, connected with Private Information Retrieval, Consensus and Sybil Defences. Our focus on those technologies was a direct result of our previous work on NEXTLEAP key distribution, and in particular our work on the ClaimChain technology and the Autocrypt adoption of parts of it.

In particular it became apparent that clients need some way to request keys from each other, or from services in a privacy preserving manner – as to not reveal who they might be interested in sending an email to. This technology is *Private Information Retrieval*, and we provide a reference implementation of IT-PIR that forms the basis for our ε-PIR work.

Secondly, many clients may wish to collaborate and constitute a collective view about each others' keys or the keys of a larger population of users. Achieving *Consensus*, particularly when some users may be malicious is not trial matter in computing. We experimented with implementing the classic state machines for the DLS and PBFT consensus mechanisms; and then implemented our own proposal for achieving consensus namely Blockmania.

Finally, in an open system the question of admission control, as well as prevention of abuse by multiple 'fake' nodes becomes important. In this context we implemented a prototype of our *SybilQuorum* mechanism, that leverages social network ties, to eliminate malicious (Sybil) nodes from a social graph, and strengthen proof-of-stake Sybil Defence mechanisms.

As part of NEXTLEAP we also contributed to Open Source cryptographic libraries, that we used both for this deliverable as well as previous work on claimchain. We summarize these contributions, and provide references to the repositories, at the end of this deliverable.

Some of this implementation work was done in collaboration with, and partially funded by `chainspace.io`, who have subsequently been using the Blockmania algorithm.

2 Private Information Retrival

Module	qdpir
Repository	`https://github.com/gdanezis/qdpir`
Language	Python (with Numpy in C)
Lines of code	108 LOC
Testing	50 LOC testing both correctness and timing.

This modules provides a reference implementation of an IT-PIR scheme, allowing a client to package a request

to a database; split the request across multiple databases; the databases processing those requests (without learning which record was sought; and providing responses that the client combines to recover the record requested.

Notably the reference code uses the `numpy` library to process multiple requests in parallel on the server side, to increase the computational efficiency of the PIR process. We implemented timing tests to ensure that the majority of the time, from the server side, is indeed spent within those efficient maths libraries rather than Python code.

This module provided the baseline performance (both timing and network overhead) for our work on more efficient ε-PIR schemes as described in D2.4 and evaluated in D4.5.

3 Blockmania Consensus

Module	Blockmania
Repository	`https://github.com/gdanezis/blockmania`
Language	Python (with Simpy for simulation, and TIKZ for illustrations)
Lines of code	461 LOC
Testing	300 LOC testing using Simpy, and testing output using TIKZ.

This module implements the Blockmania core consensus algorithm. The blockmania code is split in two parts: the first allows nodes to exchange a directed acyclic graph of blocks (Block-DAG) containing statement on which all participants need to agree, and also report seeing previous blocks. The second Blockmania component processes this Block-DAG at each node, and emits blocks and statements on which all honest nodes will eventually agree. The full details of the blockmania algorithm are available within Deliverable D2.4.

Testing asyncronous code, such as the one in this module, under real network conditions is notoriously difficult. Real networks can exhibit extreme conditions (high asynchrony, out-of-order delivery, etc), however those conditions cannot be reliably induced or replicated as part of a rigorous testing regime. For this reason, we implemented within this module a discrete event simulator (using Simpy) for an asynchronous network that is able to reproduce deterministically such difficult network conditions, to test both the safety and reliability of the Blockmania code.

We also augmented the debugging output provided by this module with code to automatically produce graphical representations of the consensus algorithm using the TIKZ Latex packages. Samples of those automatically generated figures illustrate the Blockmania algorithm in D2.4 as well as the view-change part of the algorithm in D5.4.

Module	pybft
Repository	`https://github.com/gdanezis/pybft`
Language	Python (with pytest for testing)
Lines of code	699 LOC for consensus
Testing	565 LOC with pytest.

Module	DLSconsensus
Repository	`https://github.com/gdanezis/DLSconsensus`
Language	Python (with pytest for testing)
Lines of code	812 LOC for consensus
Testing	486 LOC testing using pytest.

One does not simply design a partially-synchronous byzantine consensus algorithm. Lamport received a Turing award in 2013 for designing consensus algorithms in a crash-fail setting; and Dwork received the Dijkstra Prize in 2007 for her work on byzantine consensus. Consensus algorithms have to be reliable despite arbitrary failures, any arbitrary delays of messages, asynchronous networks, etc. They are some of the most complex algorithms known, and also must be implemented extremely efficiently since they mediate key network interactions of nodes.

Before embarking in the design of Blockmania, we implemented two consensus algorithms within the literature namely PBFT[1] (1999) and DLS[2] (1988). Both provide consensus given a set of 3f+1 nodes out of which at most f are corrupt, and in the context of network partial synchrony.

Our experience implementing those algorithms informed our choices when designing Blockmania, and focused our minds to design our own algorithm foremost with simplicity in mind. The PBFT algorithm state machine alone spans about 700 LOC and DLS 812 LOCs. Furtermore to ensure they are correct and live we implemented extensive test suites using pytest. Given the complexity and the number of edge cases, those span 565 LOC for PBFT and 486 LOC for DLS.

The lessons we learned from implementing those paid off: in comparison Blockmania only spans 461 LOC both for the state machine and the Block-DAG components; and can be tested using about the same amount of lines of code. Furthermore the testing of the Block-DAG does not involve complex asynchonous code, since it is simply and off-line graph processing algorithm — making testing for correctness much easier than in the PBFT and DLS case.

We have open sourced those modules to provide the community with reference implementations for those classic algorithms. Those are difficult to fully implement from reading the research papers alone, that only describe in narrative style the algorithms; providing only high-level details for the happy paths and most importantly fewer details for the unhappy paths when failures occur. For instance for PBFT we had to reverse engineer the details of the algorithm from the formal proofs of safety (see MIT report) rather than the description in the original paper.

4 SybilQuorum

Module	SybilQuorum
Repository	`https://github.com/gdanezis/SybilQuorum`
Language	Python (with nodex for graph processing)
Lines of code	450 LOC
Testing	200 LOC testing.

SybilQuorum strengthens a proof-of-stake system with nodes placing stake on social network connections. The algorithm analyses this graph of connections, and their value, to detect regions that are poorly connected to the rest of the network and exclude them as Sybils. The remaining 'honest' nodes must then ensure they can constitute quorums to make common consensus decisions. The full algorithm is described in D2.4

This module implements the SybilQuorum algorithm, that takes a weighted social graph; excludes Sybil nodes; and then constitutes quorum sets. A node accepts a decision as authoritative if one of its quorum sets agrees with it (and it itself agrees with it). The module also implements facilities for reading real-world social graphs, from the Stanford Network Repositories[3] and subsampling them to simuate weighted social networks according to strenth of tie. It also implements facilities for simulating ddifferent types of Sybil attacks on those graphs, and attacks of different intencity, and to test whether those attacks are successful or not against SybilQuorum.

[1] See Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." In OSDI, vol. 99, pp. 173-186. 1999. but also Castro, Miguel, and Barbara Liskov. A Correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical Memo MIT-LCS-TM-590, MIT Laboratory for Computer Science, 1999.

[2] See Dwork, C., Lynch, N., & Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. Journal of the ACM (JACM), 35(2), 288-323.

[3] Leskovec, Jure, and Andrej Krevl. "SNAP Datasets:Stanford Large Network Dataset Collection." (2015).

5 Cryptographic Infrastructure

Module	Petlib (Patch for OpenSSL 1.1)
Repository	`https://github.com/gdanezis/petlib/pull/19/files`
Language	Python & C bindings
Lines of code	+1,536 -976
Testing	pytest and Travis CI testing.

As part of our continued work on the above projects and also Claimchain the EPFL team has made a significant contribution to the petlib cryptographic library. They contributed over 1,500 lines of code to modernize the library to use the latest OpenSSL 1.1.x API, over the older OpenSSL 1.0.x API. This allows the NEXTLEAP code to be easy to install in modern systems, and also to ensure that security critical updates can be used by petlib as soon as they become available, and after the end of life of OpenSSL 1.0.x.