Implementing PriFi’s Equivocation Protection Protocol Using the SDA Framework

Advisor: Joan Feigenbaum
Jayshree Sarathy

Abstract. Dealing with equivocation attacks, which are attempts by a malicious relay to de-anonymize clients, is crucial for preserving privacy in anonymous communication systems. Previous approaches to protect against such attacks have added a nontrivial latency overhead. PriFi, a recently-developed anonymous communication protocol, utilizes a novel protocol for equivocation protection that adds minimal latency to the system. For this research project, I contributed to the ongoing migration of the PriFi implementation onto SDA, a framework for building secure distributed systems. In particular, I implemented the equivocation protection feature of PriFi in the SDA model, which required manipulating the protocol to conform to SDA’s message passing structure. Adding this key component to the PriFi-SDA implementation ensures provable security against equivocation attacks without sacrificing practicality.

PriFi

PriFi is an anonymous communication protocol developed by researchers at Yale and EPFL. PriFi combines the practicality of Tor [3] and the security of its predecessor, Dissent, to provide a low-latency protocol for anonymous communication in local area networks (LANs) that is provably secure against traffic-analysis attacks. PriFi builds on Dissent’s use of DC-nets (Dining Cryptographer’s Networks)[2], a scheme for perfect anonymity, by significantly improving performance costs. PriFi uses a client-relay-server architecture in which off-path servers compute ciphertexts in parallel with clients to protect client anonymity and minimize communication latency.
A key feature of PriFi is its technique for dealing with equivocation attacks. An equivocation attack occurs when a malicious relay sends inconsistent downstream messages to the clients. Since these messages may affect the requests that clients send back to the relay in subsequent rounds, the relay will be able to identify the source of each request and de-anonymize the clients.

Traditional methods for dealing with equivocation attacks impose a latency overhead on each round of requests. PriFi, however, uses a method that preserves low latency by encrypting upstream messages based on the clients’ agreement on the messages they received in all previous downstream rounds. If there are any discrepancies, the encryption of upstream messages in the next round will produce meaningless data, and the relay will be unable to decrypt the messages. Thus, the continuation of communication is dependent on the trustworthiness of downstream messages sent from the relay to the clients [4].

**SDA**

SDA (Secure Distributed API) is a framework for building distributed systems created by Professor Bryan Ford’s team at EPFL. It allows users to deploy cryptography protocols over a collection of servers that form a collective authority [1]. SDA uses an overlay to map the nodes of a tree structure defined by a protocol to the underlying cothority nodes, or collective authority servers. It provides an app-service-protocol architecture that allows user-space applications to interact with cothority nodes through the public API of services and to pass messages across nodes using protocols. To help our team and other developers navigate the framework, I created a manual describing the different components of SDA and showing how they interact. Below are a diagram and some descriptions from the manual:
**Application Layer.** A *simulation* can instantiate a Protocol or start a Service. *Applications* use the public API of Services to communicate with cothority nodes, or *conodes*.

**Service Layer.** *Services* implement the SDA Service interface and are used to instantiate *protocols*. Unlike protocols, they have a persistent state that is kept over the reboot of the cothorityd binary. Protocols define procedures with concrete entry and exit points for passing messages among conodes. They implement the SDA TreeNodeInstance interface. An instantiation of a protocol’s message passing structure is called a *Protocol Instance*.

**Conode Layer.** A *conode* is a server that is running the cothorityd app.

**Coothority Layer.** A *cothority* is a collective authority formed by any group of two or more conodes.

**Server Layer.** A *server* can run one or more cothorities at the same time.
with different configurations.

Several crypto protocols, such as CosiSign, ByzCoin, and RandHound, have already been implemented using SDA. We are now in the process of migrating PriFi from a standalone implementation to an SDA protocol. This is expected to simplify the code significantly by abstracting out the networking and message passing segments; however, it involves significant manipulation to make the structure of PriFi compatible with the SDA model. Ludovic Barman at EPFL created PriFi-Lib and a PriFi-SDA-Wraper to interface between the SDA Protocol structure and PriFi’s implementation. Matthieu Girod, a student at EPFL, is also working to refactor the wrapper to work better with the Application-Protocol-Service structure of SDA. My contribution to this migration so far has been to implement PriFi’s equivocation protocol in the SDA version of PriFi.

Equivocation Protection

Equivocation protection is achieved by encrypting upstream messages using a key distributed amongst the clients. The relay is able to obtain the key only when all of the client histories match; if there are discrepancies, the relay will be unable to decrypt the messages from that round onward. Communication is thus continually dependent on the trustworthiness of the relay.

Here, we outline PriFi’s low-latency protocol for protecting against relay equivocation attacks [4]:

**Clients.** Each client has an initially empty downstream history. When it receives a downstream message from the relay, it sets its downstream history as the hash of the message appended to the previous history. The client sends to the relay a pair consisting of its ciphertext and a decryption key. If the client is not sending a message in this round, the ciphertext is the XOR of its shared secrets with servers, and the key is the downstream history raised to the sum of its shared secrets. If the client is sending a message in this round, it first selects a pseudorandom key as a one-time pad to encrypt its upstream message. The ciphertext is the XOR of its shared secrets and the encrypted message, and the decryption key includes the chosen pseudorandom key.

**Servers.** Each server sends a pair consisting of its ciphertext and its composite verifiable shared secrets to the relay. The ciphertext is the XOR of all of its shared secrets with clients and the composite verifiable shared secrets is the negative sum of all of its shared secrets.
Relay. The relay starts with an empty downstream history. When it receives a downstream message from the internet, it updates its downstream history to be the hash of the message appended to the previous history. Next, it receives a pair from each client containing the client’s ciphertext and decryption key and a pair from each server containing the server’s ciphertext and composite verifiable shared secrets. Using the inputs from the clients and servers, the relay attempts to decrypt the upstream message sent in this round. First, it obtains the key by multiplying the product of the clients’ decryption keys with the relay’s downstream history raised to the composite verifiable shared secrets from the servers. Only if the downstream histories of the relay and all the clients are equivalent (making the base of the exponents equal) will the exponents containing the shared secrets be able to cancel out, revealing the decryption key of the message sender. Then, the relay can XOR the ciphertexts received from all the clients and servers to reveal the encrypted message. Finally, the relay can use the decrypted key it found earlier to obtain the actual message.

Security. The security of this protocol can be reduced to the hardness of the Discrete Log Problem and the Decisional Diffie-Hellman assumption.

To add this equivocation protocol to PriFi-SDA, I made the following changes. First, I added the UpdateMessageHistory function to hash each new downstream message with the downstream history and set the result as the new downstream history. For the client, I changed the cell coder to OwnedCoder, called the UpdateMessageHistory function whenever it received a downstream message, initialized the client’s cell coder, and set the default MessageHistory. Then, I had the servers collect and compute their verifiable composite shared secrets. To transfer these secrets to the relay at the appropriate time, I created a new message struct and receiver (TRU_REL_TELL_SHARED_SECRETS) for the relay to obtain these shared secrets. Finally, I changed the relay’s cell coder to OwnedCoder, updated its downstream message history by hashing it with the new downstream message, initialized the cell coder and the default message history, and had the relay obtain the servers’ composite verifiable shared secrets using the message struct receiver.

Deliverables

The deliverables from the semester include the following:

- Manual: SDA Components
  https://github.com/lbarman/prifi_dev/wiki/SDA-Components
• Tutorial: Creating a new protocol in SDA

• Code: PriFi equivocation in SDA
   https://github.com/lbarman/prifi_dev/commits/jksarathy-equivocation

Future Work

I plan to continue working on this project. The next steps include completing the migration and having a working demo of PriFi on SDA with equivocation protection capabilities, simplifying the structs for message passing in the PriFi-SDA wrapper, and testing the latency of the system under SDA against that of the standalone implementation.

Acknowledgements

I am grateful to Professor Feigenbaum for supervising this project and providing valuable guidance, and to Mahdi Zamani who taught me the basics of cryptography, helped me understand the PriFi protocol and the equivocation protection algorithm in detail, and worked with me closely throughout the semester.

References


