Life Insurance Protocol for Next Generation Dissent

William Edward Bailey, III

April 29th, 2015

Faculty Advisor: Bryan Ford
Dept. of Computer Science

I. Background

Dissent is an "accountable anonymous group communication" system that attempts to preserve user privacy in the face of increased tracking and profiling by governments and businesses. Using dining cryptographers and verifiable shuffle algorithms, servers running the Dissent protocol obscure the link between the source and destination of messages to produce greater anonymity. The next generation of Dissent attempts to increase scalability and robustness by implementing a collective signing protocol (Coco). The Coco protocol allows Dissent servers to produce digital signatures that are composites of the private keys of the individual servers. These verifiable signatures will be used to certify various files and messages produced by the system, such as the list of current Dissent nodes.

The next version of Dissent will also improve upon reliability and fault tolerance. In the previous version, the protocol would be unable to continue if a server were to crash or become unresponsive. To provide for greater fault tolerance, I have developed the Life Insurance Protocol (LIP). LIP uses Shamir Secret Sharing to shard a private key into shares that can be recombined to form the original secret. The protocol distributes the shares to servers who will “insure” the policy. The server can then inform the system that it has taken out a policy and is
ready to participate in the system. If the server ever becomes unresponsive, other servers can contact the insurers to retrieve the shares and reconstruct the secret. Hence, the system can make progress in the midst of failures.

II. Key Terms

- **Promise** = the cryptographic object that stores the Shamir shares and can be used to reconstruct a “promised” private key.
- **Promiser** = a server who creates a Promise of its private key.
- **Insurers** = servers who receive and guard a share of a Promise.
- **Clients** = other servers who are clients of the promiser. They will reconstruct the promised secret if the server dies.
- **Certified** = enough insurers have endorsed a Promise for it to be considered trustworthy.
- **Long Term Key** = a public/private key pair used to identify a server across the network.

III. Code Overview

The code for LIP is divided into two main portions: the cryptographic code (promise.go) responsible for producing the Shamir shares and implementing other crypto logic, and the networking code (lifePolicy.go) responsible for sending protocol messages over the network.
A. promise.go

1. Promise

Building upon crypto/poly/sharing.go within Dissent's Github repository, Promise objects provide a mechanism to reconstruct private secrets. New promises can be created as follows:

```go
func (p *Promise) ConstructPromise(secretPair *config.KeyPair, longPair *config.KeyPair,
                                  t, r int, insurers []abstract.Point) *Promise
```

secretPair is the public/private key pair containing the private key to be promised. longPair is the key pair that is used to identify the server over the network. The values t and r are configuration parameters. A total of t shares are needed to reconstruct the secret; LIP will only recognize a Promise as certified once r insurers have endorsed it. Lastly, insurers is an array of the long term public keys of the servers who will act as insurers. The function takes the secret private key of the server and then shards it into n shares where n is the length of the insurers array. It then uses Diffie-Hellman encryption to encode each shard and stores them into an array. Lastly, it produces a Shamir public polynomial that insurers can use to verify that their shares can be used to reconstruct the promised private key.

Promise provides other functions such as ProduceResponse, which insurers can use to verify that a share it received was properly constructed and to produce a response either expressing approval or rejection of a Promise.

2. State

A Promise behaves like a static object. Once it is created, it can not be changed. This inhibits the ability of malicious servers to alter promises and send different versions to different
servers. In order to track dynamic information about a Promise, the State struct is used. When servers receive shares revealed by insurers or responses expressing an insurer's approval or rejection of a Promise, they store these objects within a State. The servers can call PromiseCertified to check whether a Promise is certified or State.PriShares.Secret to reconstruct the promised secret from the shares.

3. Response

```go
type Response struct {
    // The type of response
type responseType

    // For unmarshalling purposes, the suite of the signature or blameProof
    suite abstract.Suite

    // A signature proving that the insurer approves of a Promise
    signature *signature

    // blameProof showing that the Promise has been badly constructed.
    blameProof *blameProof
}
```

Insurers use the Response struct to express approval or disapproval of a Promise. If an insurer checks a share it receives and the share was properly constructed, the insurer will produce a signature response. A signature struct (created by Promise.sign) is a digital signature of the message “Promise Signature” that is signed by the insurer's long term public key. When other servers receive the response, they can verify that the signature was produced by the server that the Promise records as the insurer for that share. This prevents malicious servers from forging approval messages for others.

If the share the insurer receives fails to be verified (either because it fails the public
 polynomial check or fails to be properly decoded by Diffie-Hellman), the promiser has produced a malicious Promise. The insurer will reject the Promise and create a blameProof response.

```go

type blameProof struct {
    // The suite used throughout the blameProof
    suite abstract.Suite

    // The Diffie-Hellman shared secret between the insurer and promiser
    diffieKey abstract.Point

    // A HashProve proof that the insurer properly constructed the Diffie-Hellman shared secret
    proof []byte

    // The signature denoting that the insurer approves of the blame
    signature signature
}
```

A blameProof uses several cryptographic objects to prove that the share was improperly constructed. The insurer reveals the Diffie-Hellman shared secret. It provides a proof using crypto/proof code to demonstrate that the insurer has knowledge of its private key and properly multiplied it by the promiser's public key to produce the Diffie-Hellman key. Once the Diffie-Hellman key is proven to be correct, other servers can then decrypt the insurer's share and see for themselves whether the decoded share passes the Promise's public polynomial check. If it passes, the insurer's blame was unjustified. If it fails, the insurer has proven that the Promise (and in turn the promiser) is malicious. Once a valid blameProof is received, a Promise will forever be considered uncertified.
B. lifePolicy.go

The networking code of LIP is stored within lifePolicy.go. Building upon the Promise cryptographic object, the file defines the LifePolicyModule struct, which servers can use to create promises and send them across the network. The interface defines the following methods:

1. Init

   As its name suggests, Init initializes a LifePolicyModule defining the default configuration for Promise objects created. Users specify the Promise values for t, r, and n (where n is the number of insurers a Promise will have). All promises created by the system will use these values; all promises received are expected to have these values. This prevents malicious servers from sending a Promise with only one insurer (who just so conveniently happens to be malicious as well) that needs only one signature to be considered certified. Init also allows the caller to define other parameters such as the connection manager and the timeout to use for sending requests.

2. TakeOutPolicy

   TakeOutPolicy is the most important function a promiser will use. Given a private key, a list of insurers, and a function for selecting insurers, TakeOutPolicy produces a Promise object for the private key. It then sends requests to the insurers asking them to certify the Promise. It waits for responses until the timeout expires and reports back to the caller whether the Promise is certified. Accepting the method for selecting insurers as a parameter will be useful for future iterations of LIP. The Dissent team is currently considering methods for randomly yet verifiably
choosing insurers so that malicious peers can not handpick co-conspirators. When this function is developed, it can simply be passed to this function.

3. SendPromiseToClient

Once a Promise has been certified, a promiser can send the Promise to clients using this function. In the event that the promiser goes down, clients can use the Promise to reconstruct the promiser's secret and replace the promiser in the system.

4. CertifyPromise

Clients will not blindly accept a Promise it receives. They will call CertifyPromise to contact the insurers and check for themselves whether the Promise is certified. While this \( O(n^2) \) communication cost is not ideal, it is necessary for security. If the promiser were to send clients all the responses it received in addition to the Promise, the promiser could simply leave out blameProofs and pretend the Promise is certified even though it is not. Hence, each client must contact the insurers themselves. The Dissent team is currently considering ways of using cryptographic objects to verify that the Diffie-Hellman encrypted shares were constructed properly without needing to know the Diffie-Hellman shared key. With such methods, a client could completely verify that a Promise was correct without needing to contact insurers. Indeed, a promiser would not even need to send the Promise to the insurers but could simply send it to the clients. If a client ever needed to reconstruct the secret, it could simply include the Promise to the insurers in a request. Despite this, \( O(n^2) \) communications can not be completely escaped since clients and promisers still need to check whether the insurers exist and are alive. Nevertheless,
such a change would dramatically reduce bandwidth costs as certification would simply consist of sending pings to see if the insurers are alive.

5. ReconstructSecret

Once a Promise is certified, clients can trust a promiser to perform work in the system. If the promiser becomes unresponsive while performing some work, clients can contact the insurers via ReconstructSecret to reconstruct the promised secret. When the insurers receive the request, they will call LifePolicyModule.verifyServerAlive to determine if the server is still alive.

\[
\text{verifyServerAlive func(reason string, serverKey, clientKey abstract.Point, timeout int) error}
\]

verifyServerAlive is provided to the LifePolicyModule as a parameter in Init by other protocols using LIP. The function takes as arguments a string stating what work the client was doing when the server became unresponsive (reason), the long term public key of the promiser (serverKey), the long term public key of the client (clientKey), and the timeout. The function will contact the client for any additional information needed regarding the work being done. It will then contact the server to retrieve the results. If the server responds in time, the insurer will simply send the results back to the client. If the timeout expires, the insurer will reveal its share and send it to the client.

By enabling this function to be passed within Init, LifePolicyModule can work with a variety of protocols that have unique requirements for the type of work done. For protocols that simply wish to check if the server can respond, LifePolicyModule provides a default version of this function that simply pings the server.
6. HandlePolicyMessage

Lastly, LifePolicyModule provides a means for users to process LIP messages via HandlePolicyMessage. It is vitally important that users of LIP check for messages frequently. Any server can select any other server as its insurer; hence, servers must be listening for messages and be ready to respond. This is particularly important when insurers are attempting to verify if a server is dead. A server that is alive does not want to appear to be dead simply because it rarely checks LIP messages.

C. policyMessage.go

All LIP messages are defined within this file. Along with the cryptographic objects of promise.go, every policy message implements the Marshaling interface defined by crypto/abstract. In addition to implementing the golang interfaces encoding.BinaryMarshaller and encoding.BinaryUnmarshaller, the code also defines:

- MarshalSize = Determines the number of bytes used in the encoding
- MarshalTo(w io.Writer) = Uses an io.Writer to perform the marshaling
- UnmarshalFrom(r io.Reader) = Uses an io.Reader to perform the unmarshalling
Several marshaling codes handle variable length structs like PromiseResponseMessage, which insurers use to inform clients and promisers whether they approve or reject a promise. The fields PromiserId, Id, and Response are all variable length. For each field, the marshaling code determines its size and then converts the field into binary. The sizes of the fields are stored first in the marshaled binary array followed by the actual data. When performing the unmarshalling, the sizes of the fields are retrieved first and used to compute the total expected size of the buffer. The rest of the buffer is then read to decode the message.

For more detailed documentation about marshaling or the policy messages, please see the code within insure.tar.gz.

IV. Testing and Documentation

One of my main goals for this semester was to leave the Dissent team with a robust and well-documented protocol that they could easily maintain and build upon. To do so, I wrote unit tests for each of the files above:

```go
type PromiseResponseMessage struct {
    // The index of the share being approved or rejected
    ShareIndex int

    // The id of the server who created the promise
    // aka. promise.PromiserId()
    PromiserId string

    // The id of the promise itself
    // aka. promise.Id()
    Id string

    // The insurer's response
    Response *promise.Response
}
```
• promise_test.go = 90.7% coverage with 35 tests
• policyMessage_test.go = 91.9% coverage with 15 tests
• lifePolicy_test.go = 96.5% coverage with 19 tests

For the life policy tests, I also created a functional test. Using go channels to simulate a network, I tested the entire stack from creating promises and certifying them to reconstructing the secret. Since the code sends messages via a connection manager that is passed to LifePolicyModule via Init, configuring the code to work over the network will simply be a matter of passing in the appropriate connection manager. The LIP specific logic has been demonstrated to work via these tests.

Concerning documentation, I commented the code extensively highlighting subtle nuances of the protocol. Sample comments for LifePolicyModule's Init function follow:

```go
/* Initializes a new LifePolicyModule object. */

/* Arguments:
   * kp = the long term public/private key of the server
   * t, r, n = configuration parameters for promises. See crypto's
             promise.Promise for more details
   * cmann = the connection manager for sending/receiving messages.
   * defaultTimeout = the default timeout for waiting for messages.
   * verifyServerAlive = the function used to verify that a server is dead. Enter
                       nil to use the default method. See the documentation for
                       theLifePolicyModule for more information on how to write
                       such a function.

   * Integration Note: cman should provide some way for the server to communicate with
   itself. Insurers will sometimes need to check that the promise they
   are insuring is certified. Hence, they will send messages to
   themselves when trying to get promiseResponses.
*/
```
For each function, I generally provided a few sentences specifying the purpose of the function, listed the arguments defining what they were along with important notes on how they are used, and provided additional guidelines critical for integrating or maintaining the code. In the case of Init, I noted that the connection manager must provide a way for the server to communicate with itself. This is needed to cover the edge case where an insurer seeks to certify a Promise and hence sends a message to itself since it is one of the insurers. In short, the documentation was written to serve as a guide for understanding and maintaining the code.

V. Conclusion

All code has been submitted to Professor Ford as a pull request into Dissent's Github repositories. I am considering working on the project beyond CPSC 490 by making additional changes/ refactoring as suggested on the pull request. I am also considering integrating LIP with Dylan and Iulia's code and perhaps looking into some of the different improvements to the protocol mentioned above. I have enjoyed working with the Dissent team this semester and building a Life Insurance Protocol for a more scalable Dissent.