A Portable Implementation of the TESLA Secure Multicast Algorithm

Tom Burns

Dr. Michel Barbeau

COMP 4905

April 4th, 2006

Table of Contents

1. Introduction

1.1 Context

In multicast environments, it is possible for a malicious entity to forge packets claiming to be from the source node. An example environment is with satellites – satellite communication requires secure channels. Satellite environments also deal with a multitude of constraints including bandwidth and computational/storage limitations.

The CanX satellite program is a novel approach to satellite development: as explained in [3], the program combines state-of-the-art technology into a highly optimized nanosatellite of size 1 to 5 kilograms. The CanX-2 satellite, launching during 2006, provides an ideal testbed for a lightweight secure communications protocol. Our application will be tested during a portion of the satellite uptime, following it's launch.

There are a number of constraints due to the CanX-2 environment:

- unidirectionality, as it is not expected that we will have access to the radio uplink via the Internet connection to the ground station.
- computationally inexpensive, since the target platform for the application is the Philips LPC2214 60 mhz ARM-based microcontroller.
- portability, as the exact details of the target environment may change.
- low bandwidth overhead, since the maximum downlink baud rate is 4000 bit/s [3].

1.2 Problem

The problem addressed by the project is to implement a secure multicast protocol that is suitable to run in the target environment as described above. The implementation should attempt to provide as much cryptographic strength to the communication channel as possible, including authentication and confidentiality. As well, the implementation must be portable, with the specific target in mind being the embedded computer board based on the Philips LPC2214 chipset as used in the CanX-2 nanosatellite.

1.3 Result

An implementation of the TESLA protocol (as described in [1]) has been developed with respect to the given constraints. The TESLA protocol provides both source and message integrity to the multicast messages from the server. As described in Section 2.1, the TESLA protocol is lightweight with respect to both processing and bandwidth resources. As well, with minor modifications, the protocol can operate unidirectionally to provide server authentication.

In conjunction with the TESLA protocol, the public-key RSA algorithm is used to provide absolute authentication of the server by the clients. While the RSA operations are expensive, they are only used periodically to synchronize clients with the current TESLA chain sequence.

The application has been developed in a modular fashion, entirely in C, which aids portability. As well, it does not depend on any non-portable libraries. Therefore the task of porting it to a different architecture should be simple.

1.4 Outline

The paper is divided into 3 sections. Section 2 provides required background concepts and an overview of the protocol that has been implemented. Section 3 gives an analysis of the resultant application, including a breakdown of structure and performance. Section 4 concludes the work done to date, and outlines the work to be completed in the future.

2. Background

The security of the TESLA protocol is derived from the security of cryptographic hash functions, also known as 'one-way' functions. A cryptographic hash function, as described in [2], is a function chosen "such that it is computationally infeasible to find two distinct inputs which hash to a common value" [2]. As well, cryptographic hash functions have "pre-image resistance" - that is, given a hash output h(x) it is computationally infeasible to compute the input value x.

As described in [1], a hash function *h* is used recursively on a supplied input seed to create a one-way chain. The interesting feature of the chain is that, given an intermediate element x of the chain, we can compute all of the following elements of the chain by computing x' = h(x), x'' = h(x'), and so forth. However, it is computationally infeasible to compute any value in the chain that appears before x.

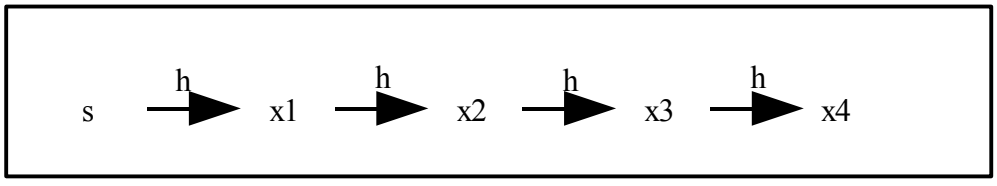

Figure 1. An example one-way chain is shown.

Figure 1 illustrates an example one-way chain. An initial value s is fed into the cryptographic hash function h, which returns the value x1. By feeding x1 into h again we obtain x2, and so forth until the final value x4. Since we use the one-way function h to travel from x(i) to x(i+1), we cannot compute x(i) given x(i+1), due to the pre-image resistance of cryptographic hash function h. However, if we were given the value x1, we could compute all following values x2, x3, x4, and so on, since we know x1 and we know h.

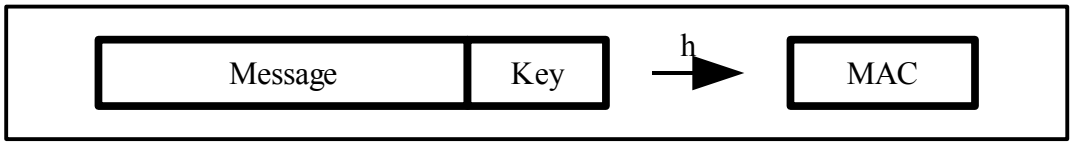

Figure 2. Use of hash function h to compute a MAC using a symmetric key.

Cryptographic hash functions are also used in TESLA for message signatures. Every message sent by the server has a Message Authentication Code, or MAC, attached to it, that acts as a signature. As shown in Figure 2, the MAC is calculated by appending a given symmetric key to the message, and hashing the result. Due to the properties of any cryptographic hash function, only a user who has knowledge of both the key and message data can compute the MAC value for the message.

In the TESLA protocol, the server divides time into intervals. Every interval has assigned to it a key for signing as well as a key to publicly disclose during that interval. Both keys exist as part of the same one-way chain, in a manner such that the signing key is earlier in the one-way chain than the disclosed key. That is, there is no way to determine the signing key from the disclosed key, but if one had the signing key they could calculate the disclosed key by computing the hash of the signing key a determinate number of times.

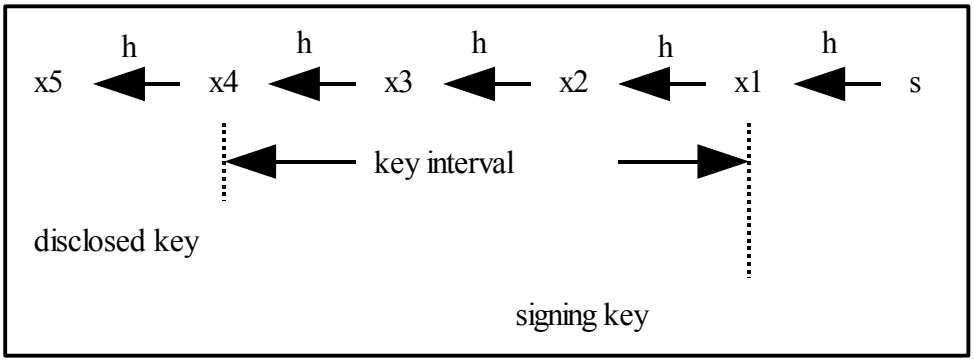

Figure 3. Position of disclosed and signing key, the distance between the two equal to the key interval, within the current one-way chain

As shown in Figure 3, the disclosed key occurs a set distance from the signing key in the one-way chain – this distance is called the key interval. When time has passed and the server is in the next time interval, the messages in that interval would be disclosing key x3 and signed with key s.

As the client receives messages, it queues them until an interval has passed that is equal to the key interval. Messages being sent during this interval will disclose a key that was used to sign a subset of the messages in the queue. We can prove that this message has been sent from the same entity that sent the earlier messages by showing that the disclosed keys are all part of the same chain.

The security of TESLA relies on loose time synchronization between the server and clients. By synchronizing the clients and server, clients can determine an upper bound on the interval of the server. When clients receive a message that is signed by a key x, they can determine if the server could potentially be in the interval where it has disclosed key x. If the server could be in that interval, then the message is discarded since another client could have received a valid message from the interval disclosing key x, and spoofed a message apparently from the earlier interval.

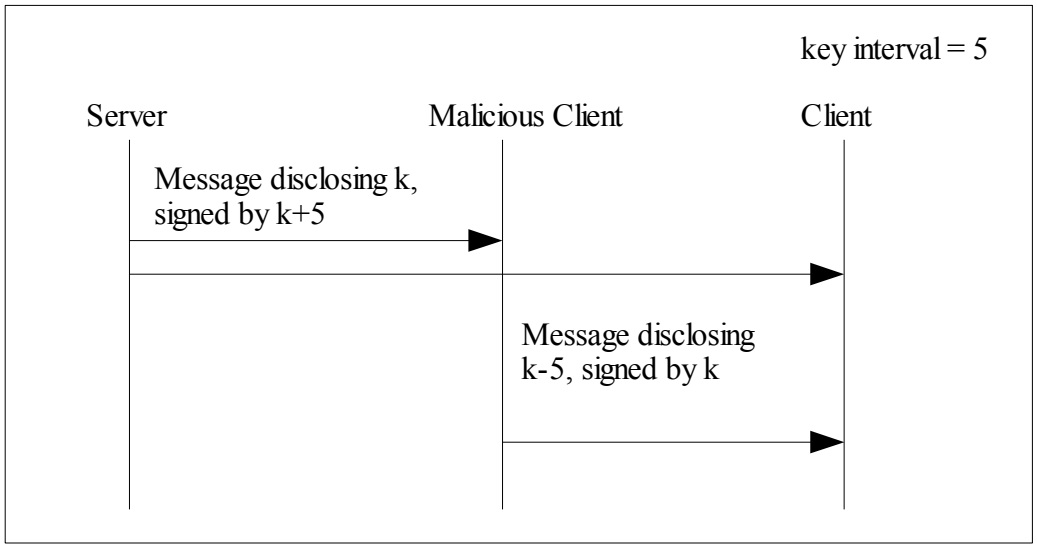

Figure 4. A malicious client attempts to fool a valid client into accepting a message that is signed using an already disclosed key.

An example of this attack is depicted in Figure 4. A malicious client attempts to fool a valid client into accepting a message that has been signed by a previously disclosed key. The target client can detect the attack since it can calculate that the server could be in a time interval where it has already disclosed the key that the malicious client is using for signing the malicious message. Therefore, it rejects the message.

In essence, the TESLA protocol provides relative authenticity: That is, any message signed using the TESLA mechanism proves that it is sent by the same user as the other messages. However, it does not provide absolute entity authenticity: it does not provide proof that any of the messages were sent by the server itself. Therefore, to provide absolute authenticity, we use the public-key RSA algorithm to periodically send messages that are public-key signed by the server that disclose a given key in the current TESLA one-way chain. A description of RSA can be found in [2].

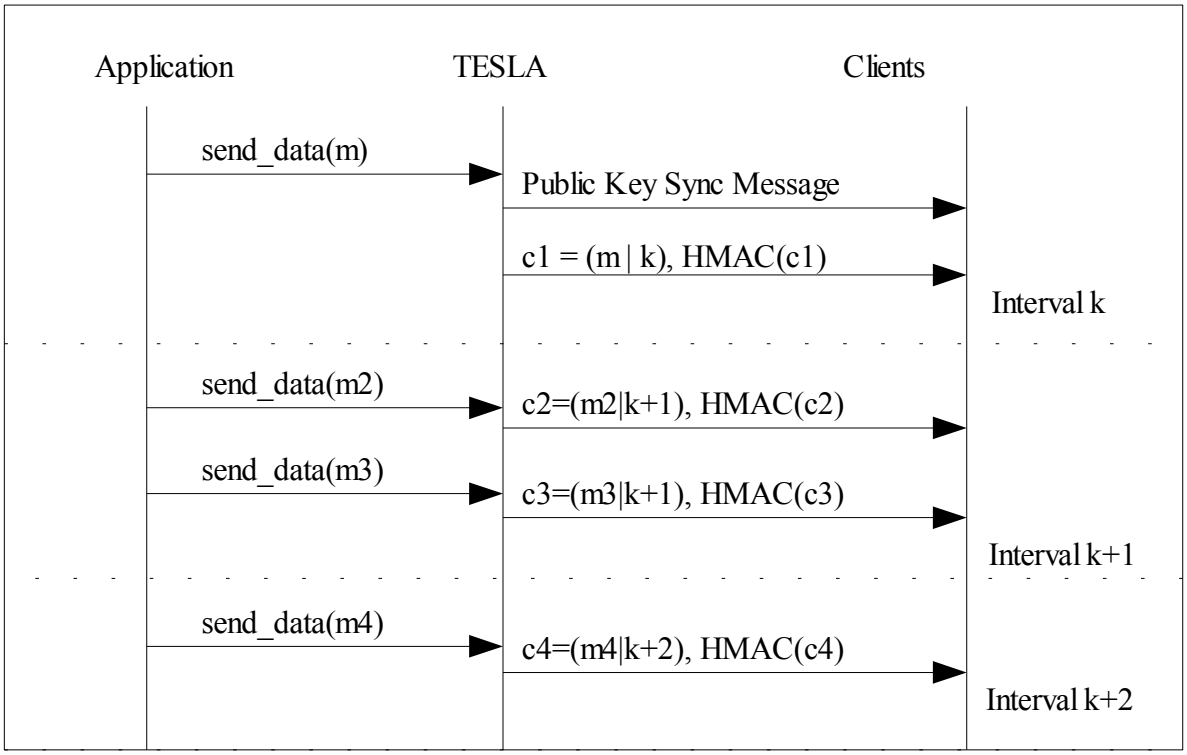

Figure 5: an example Message Sequence Diagram for a series of messages being sent.

As shown in Figure 5, when the application first attempts to send data, the TESLA layer precedes the data message with a public-key signed message that discloses the key to be disclosed for the current interval (k). Receiving clients can then synchronize themselves with the server, and messages sent during the following intervals can be authenticated to having been sent from the server through the TESLA protocol. For example, messages sent during interval k+1, disclosing key k+1, can be shown to be part of the same chain since h(k+1) = k. Once checking that the message is part of the same chain as the previously authenticated key, the client buffers the message until the server discloses the key (k+key_interval), which was used to sign the messages in interval k. If key_interval=2, for example, then in Figure 5, the message c4 discloses the key that was used to sign messages during interval k – message m. Receiving clients can now validate HMAC(c1) and trust that message m is authentic.

3. Implementation

3.1 Notable Features

Some notable features of our implementation of the TESLA algorithm are reviewed hereafter.

Firstly, the cost of generating (all but the initial) one-way chains on the server-side is amortized over the duration of using the predecessor chain. This is implemented by having the next chain be generated at the same rate that the current chain is being used up. The outcome is a mostly static latency between the start and end of the TESLA data sending calls on the server side no matter if the protocol requires a new chain for signing or not.

As well, the data structure for one-way chains has been optimized to reduce storage and computational resources needed to compute a specific key within the key chain. The data structure stores O(log n) keys, spaced O(log n) entries apart in the chain. When a specific key is requested, the function only generates the remaining keys from the nearest stored key.

The ease of use of the implementation should be reiterated. Aside from a half-dozen initialization calls, the internal layers of the application handle all protocol issues, making the security of the protocol a seamless integration into already created applications.

3.2 Requirements

Written entirely in C, there are not many hard-coded requirements. Due to the modularity of the program, most requirements can easily be changed to something a target platform provides. In it's current state, the program requires:

- GCC, and the Standard C Library
- BSD sockets interface (but it is trivial to write a new I/O driver)
- libgcrypt, for the public key authentication functionality. Libgcrypt does not depend on any other libraries aside from those already required by the application.

3.3 Architecture

A layered approach is taken in the implementation of the protocol. Implementing the application in this manner makes maintenance and porting simple, since upper layers mostly only depend on the lower layers. The job of maintenance is also simple since each layer has defined functions and dependencies. The job of porting is simple since a porter can port each distinct layer incrementally and not worry about porting the entire application at one time.

While the layered approach has been taken on a logical level, during implementation it became clear that in some instances it made sense for a lower layer to call an upper layer function. Therefore the layering depicted below does not globally represent the actual implementation of the system.

A flaw in the current implementation is that there is a lack of meaningful error messages between function calls. Most functions return SUCCESS upon success, and FAILURE otherwise, but in an embedded system that is to operate autonomously, a more robust set of error messages would lead to better error handling and better system stability.

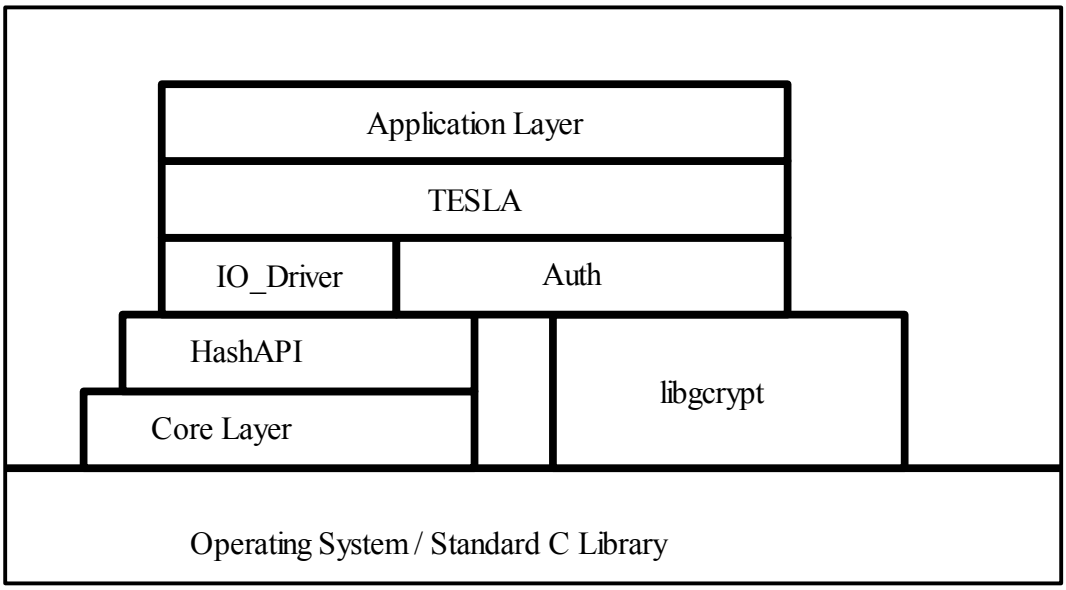

Figure 6. The Layered approach to implementation. In the diagram, any layer has the ability to call all layers that sit below it. To port the code, the simplest approach would be to port the underlying layers first, followed by each layer in order.

3.3.1 Core Layer

The core layer provides a wrapper around the Standard C library calls. It is provided to make porting as simple as possible, as the majority of later system calls depend only on the core layer and not the operating system. In the current state of the application this is not globally implemented; some upper modules do in fact directly call the C standard library functions instead of a wrapper function. However, the framework is provided to easily port the remaining calls to the Core layer if a target platform does not natively provide the required function.

3.3.2 HashAPI Layer

The HashAPI layer provides a large portion of the protocol, namely:

- standardized interface to one-way hash functions (see Features for further details)
- interface and data structures to easily facilitate the generation and using

of one-way chains

- creation and deletion of message structures, including MAC calculation and validation.

3.3.3 IO_Driver Layer

The IO_Driver layer implements a small API to send and receive messages over a target communications channel. All communication details are abstracted from the application layer: programs only have to tell the IO_Driver layer to initialize itself (and provide some required parameters, such as target host and TCP port number for a TCP/IP based driver), and then use the provided message sending and receiving functions to transfer data. Provided at this time are drivers for communication over simple POSIX data pipes, and over a BSD socket layer.

3.3.4 Authentication Layer

The authentication layer is a wrapper around the libgcrypt library that provides a simple API for dealing with public-key cryptography. Functionality provided includes:

- generating keys of arbitrary size
- loading embedded keys from the application
- signing arbitrary sets of data
- validating signatures on data sets.
- conversion utilities to/from internal libgcrypt data representations

3.3.5 TESLA Layer

The goal of the protocol implementation is to have using the implementation of the protocol no harder than using standard read/write system calls. Therefore, the TESLA layer was created to encapsulate all protocol details and hide them under

a simple API that consists solely of initialization, send data, and receive data. All actual protocol details, including signing and verifying data, key generation, and the like, is handled behind this layer, allowing application developers to focus solely on their application and not the communications protocol underneath it.

3.4 Performance

Whereas a message signed with a MAC using a TESLA key is computationally inexpensive to compute (and has low bandwidth overhead: 2 times the size of the hash output, usually 16-20 bytes), a public-key signed message is expensive and is completely overhead (since the public key message only synchronizes clients with TESLA keys in the key chain, not send payload data). Therefore we can consider performance as a ratio between number of public key messages to number of TESLA messages sent.

The minimum number of public key messages sent would be one per TESLA chain used, to initially prove ownership of the first element in the keychain. This would be sufficient to provide proof of authenticity over all messages sent using the key chain. The problem with this approach is that a new client joining the session cannot trust any messages until it receives a public key signed message proving the server is using a specific key chain – until then it has to buffer messages. With a large keychain this is obviously an unsuitable solution since the client would be buffering potentially thousands of messages.

Therefore the server must balance between optimum efficiency (sending few public key messages) and allowing for quick client synchronization (sending public key messages often enough to allow for quick client synchronization with the server). Since different environments have different requirements for these parameters, the implementation allows the developer to determine how often (in

terms of number of TESLA periods) the server should send a public key signed message.

4. Conclusion

4.1 Summary

An easy to use, functioning TESLA protocol has been implemented. As well, the protocol has been augmented in order to meet further real-world constraints such as full authenticity proof of the server by the clients through the use of the RSA algorithm. The implementation is written in portable C, with few dependencies. Modularity has been used to abstract environment-specific routines such as sending and receiving of messages, making it simple for developers to target the application to a new platform. Thus, it should be easy to port to any modern computer architecture.

4.2 Future Work

As of the writing of this paper, the protocol implementation is fully functioning on a desktop linux PC. The next step will be the task of porting the application to an embedded board based on the Philips LPC2214, following which a real-world test of the implementation will take place onboard the UTIAS/SFL CanX-2 nanosatellite.

4.3 Unsolved Issues

During development a choice was made to ignore memory allocation issues with respect to the libgcrypt library. The documentation of the library does not make it clear precisely where and when memory freeing is suitable. Therefore some work must be completed in researching where the library is allocating memory and expects

the calling function to free it.

Some research should be devoted to ensuring that the implementation is not susceptible to attacks. For example, in the current implementation the server periodically releases the current value of the system clock in order to aid clients in synchronizing their clock. However, the system clock is used initially to seed a pseudo-random algorithm which is used to create the random seeds for the initial values of the one-way chains. Therefore, it is not suggested the server leak this information as it may result in a way for a client to predetermine undisclosed keys in a chain.

4.4 Confidentiality

While not implemented, an addition to the application is planned to support confidentiality for payload messages between server and client. There are a few technical issues surrounding the topic – namely a method for secure key distribution. The problem is that the server must actively switch keys so that the effect of a key leak is limited to only a small portion of messages.

The first step to the solution is to provide a means for the server to be able to inform only the authorized clients of the new key, while outside entities cannot determine the new key. A suitable method of achieving this is to give all authorized clients a mutual private key. The server encrypts the new key with the matching public key, and only clients that have the mutual private key are able to decrypt the message to obtain the new encryption key.

There is an obvious problem with this solution: If a client leaks the shared private key, or a client is to be removed from the authenticated client pool, he will still be able to decrypt further key distribution messages from the server and obtain symmetric keys. Therefore, the server must be actively generating and using

different public/private key pairs for encrypting the new symmetric key, or it must rely on a trusted third party for key distribution to authenticated nodes.

The solution proposed is as follows: There exists a trusted third party who owns a public-key certificate. The server knows the public key of this party. When the server chooses a new encryption key, it encrypts it using the trusted third party (TTP)'s public key. The TTP receives this message, decrypts it to obtain the new symmetric key, and forwards it to only the authenticated clients.

5. References

[1] A. Perrig, R. Canetti, J. Tygar, D. Song. The TESLA Broadcast Authentication Protocol, 2002. http://citeseer.ist.psu.edu/perrig02tesla.html

[2] A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography, 1996.

[3] D. Rankin, D. D. Kekez, R. E. Zee, F. M. Pranajaya, D. G. Foisy and A. M. Beattie, "The CanX-2 Nanosatellite: Expanding The Science Abilities of Nanosatellites," Proc. 55th International Astronautical Congress, Vancouver, Canada, Oct. 2004.

Appendix A: Building & Running

Each layer is implemented as a separate library with it's own subdirectory under the project directory, so the directory structure is as follows. Each library and driver has it's own makefile in it's subdirectory. There is a file named "makefile.common" in the root project directory that defines some common variables, such as which core and io_driver module to use. All layer makefiles refer to that makefile.common for those variables.

TESLA/

- the top directory for the project

TESLA/makefile.common

- common variables for all modules

TESLA/core/

- A subdirectory containing all of the core libraries provided (currently only 'linux'). To change which core library is linked to by the application, change the variable "SYSTYPE" in makefile.common.

TESLA/hlib/

- The HashAPI layer.

TESLA/auth/

- The Authentication layer.

TESLA/io_driver/

- A subdirectory containing all of the implemented I/O drivers. Currently implemented are linux_stdio and linux_tcp. Linux_serial is a planned addition but is not yet coded. Set which io_driver to use by modifying the variable in TESLA/makefile.common.

TESLA/tesla/

- The TESLA layer

TESLA/client/

- The client application

TESLA/server/

- The server application.

For the most part, each module has it's own test application to test the layer's functionality. See each makefile for the makefile target name for the test applications. To have layers compile correctly, compile in the following order: core, io_driver, hlib, auth, tesla, then server and client. So, an example build script (see TESLA/build.sh):

```
#!/bin/bash
SYSTYPE=linux
IO_DRIVER=linux_tcp
for i in core/${SYSTYPE} ../../io_driver/${IO_DRIVER} ../../hlib ../auth ../tesla
../server ../client; do
   echo "Building $i"
   cd $i
   make clean all
done
```

The resulting client and server binaries will result in their respective directories.

Executing

Details on executing the application vary slightly based on which io_driver is in use. If the linux_stdio driver is in use, then one must pipe the output of the server to the input of the client, with a command such as (from the TESLA/ project top directory)

```
./server/server | ./client/client
```

Using the linux_tcp application, the target server and port are hard-coded into the client and server source codes. After modifying them appropriately and rebuilding, simply start up the server, followed by the client. Assuming their parameters match, they will connect and begin to communicate.