Mid-Semester Progress Report

Yiwen Tang Jiayi Zhang Yawei Zhang Rui Zhang

March 6, 2020

1 Comparison of TLSNotary and TLS

1.1 Security problems solved

TLS protocol can be utilized to encrypt HTTP channel. HTTP protocol transfers data in plaintext which can bring risks of eavesdropping, tempering and pretending. TLS protocol is designed to solve these problems. It can encrypt all the transmitted data that won't be acquired by third-party. It has a verification mechanism so that both the sender and the receiver could find out if something has changed. It has an identity certification to prevent identity impersonation. From the following figures Figure 1, after phase 2, the client has all the required values to generate the session key. During phase 3, when the client is sending the client key exchange, the client will generate 48 byte pre-master secret and encrypt it with the server's public key. After the pre-master secret is generated, the master secret can be created from the pre-master secret.

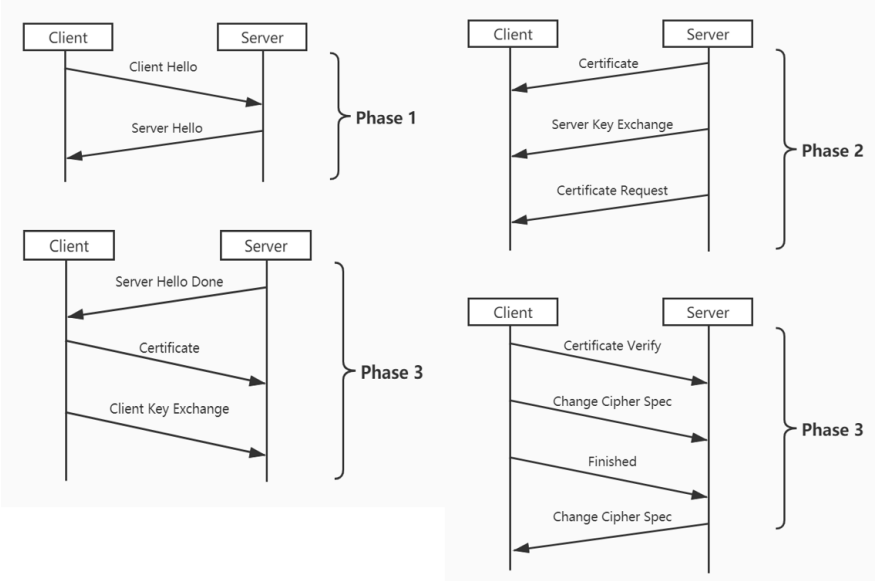

Figure 1: TLS Handshake

TLSNotary (Figure 2) assigns an Auditor to verify some part of the TLS session and withhold some part of the session. The Auditor can prevent the client from faking the traffic since he has the certificate or public key of the server with MAC record generated by the server. In this way, TLSNotoray can solve the security issues that TLS protocol fails to handle. For example, TLS can't handle the man-in-the-middle attack, which the whole session can be controlled by the attacker and the attacker can insert illegal information into the session.

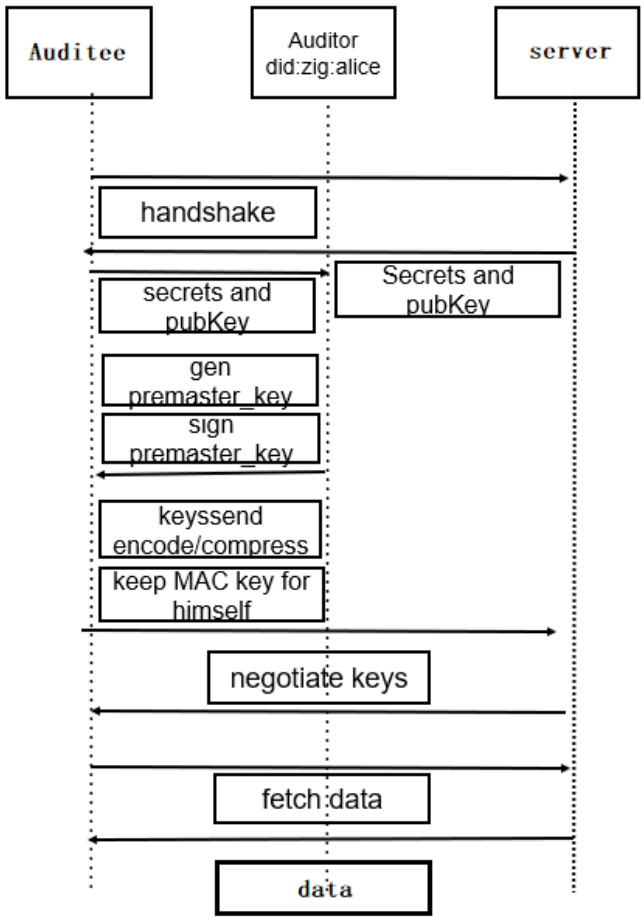

Figure 2: TLSNotary flow chart

1.2 Computation Efficiency comparison

Compared with TLS protocol, TLSNotary requires more computation resources and has lower efficiency. When generating pre-master secret, TLSNotary need an additional step to combine separate parts of Auditor and Auditee to generate the whole session key. When constructing the encryption of the pre-master secret, TLSNotary need to find proper padding because RSA is not secure without padding. Besides, TLSNotary needs additional computation resources to construct two or more multiplicative factors to produce the exact structure. Therefore, TLSNotary gets worse computation efficiency compared with TLS protocol.

2 Algorithm Implementation

We have researched the related algorithms and technology, which are asymmetric crypto, homomorphic encryption and pseudorandom function.

We use the multiplicative homomorphic property of RSA encryption to allow each part of the client

(Auditee and Auditor) to pass the full pre-master secret to the server without sharing the two-part pre-master secret. The equation is shown as 1, where $n = pq$ being the RSA modulus, usually a 2048 bit number.

$$(RSA(x1) \times RSA(x2)) \; mod \; n = RSA(x1 \times x2) \tag{1}$$

With the equation, we want to construct two multiplicative factors that when they are multiplied together, the Auditor and the Auditee can provide each other a factor in a encrypted form. With the consideration of secure, RSA is required to have padding. A random padding string should be prepended to the message to be encrypted. We refer the message to be encrypted as the premaster secret, a string of 46 bytes of random data with 2 bytes of version number prepended. Therefore, the Auditee will have the following format: [39 *bytes* $P1$||00||0301||12 *random bytes*||33 *bytes* 00||01] the Auditor is of this format: [119 *bytes* $P2$||25 *bytes* 00||9 *random bytes*||14 *bytes* 00||01].

After the two sequences are multiplied, we divide the last 48 bytes into two parts, which is the previous multiplicative factor $S1$ and $S2$ with the following format:
[00||02||...205 *bytes of padding*...||00||$S1$||$S2$].
At the same time, because Auditor and Auditee cannot share their own pre-master secret, the Auditor and the Auditee use the pseudo-random function to generate the Server Mac Key and the other three session keys respectively. Pseudorandom function or 'PRF' used in the TLS 1.0 RFC 2246:

$$PRF(secret, label, seed) = P\_MD5(S1, label + seed) \oplus P\_SHA - 1(S2, label + seed) \tag{2}$$

Here, for each hashing algorithm MD5 and SHA-1, P_hash refers to an HMAC construction repeated as many times as necessary to provide sufficient random bytes of data. The main steps of implementing this algorithm are as shown in Figure 3:
In summary, the purpose of this rather complex sequence of steps is: the Auditor withholds some of the secret data from the Auditee (acting as client), so that the Auditee cannot fabricate traffic from the server (since at the time of making his request, he does not have the server mac write secret). Once the Auditee has a made a commitment to the encrypted content of the server's response to his request, the Auditor can provide the Auditee with the required secret data in order to construct the server mac write secret. As a result, this process establishes a session to safely complete the decryption and authentication steps of the TLS protocol where the Auditee and the Auditor do not share the full pre-master secret.

We have implemented this pseudo-random algorithm in order to provide a basis for TLSNotary to perform packet sending and receiving later.

3 Deploy Design

The Auditor is planned to be deployed on AWS EC2. The detail of our plan is as follows:

1. Put the project into newly built EC2 instance;
2. Create a startup script, with the main function of: using two different account to run programs separately; modify the passwords to random string to lock the machine.
3. Choose ubuntu x-86 (version of 18.04) image, create EC2 instance and run script.

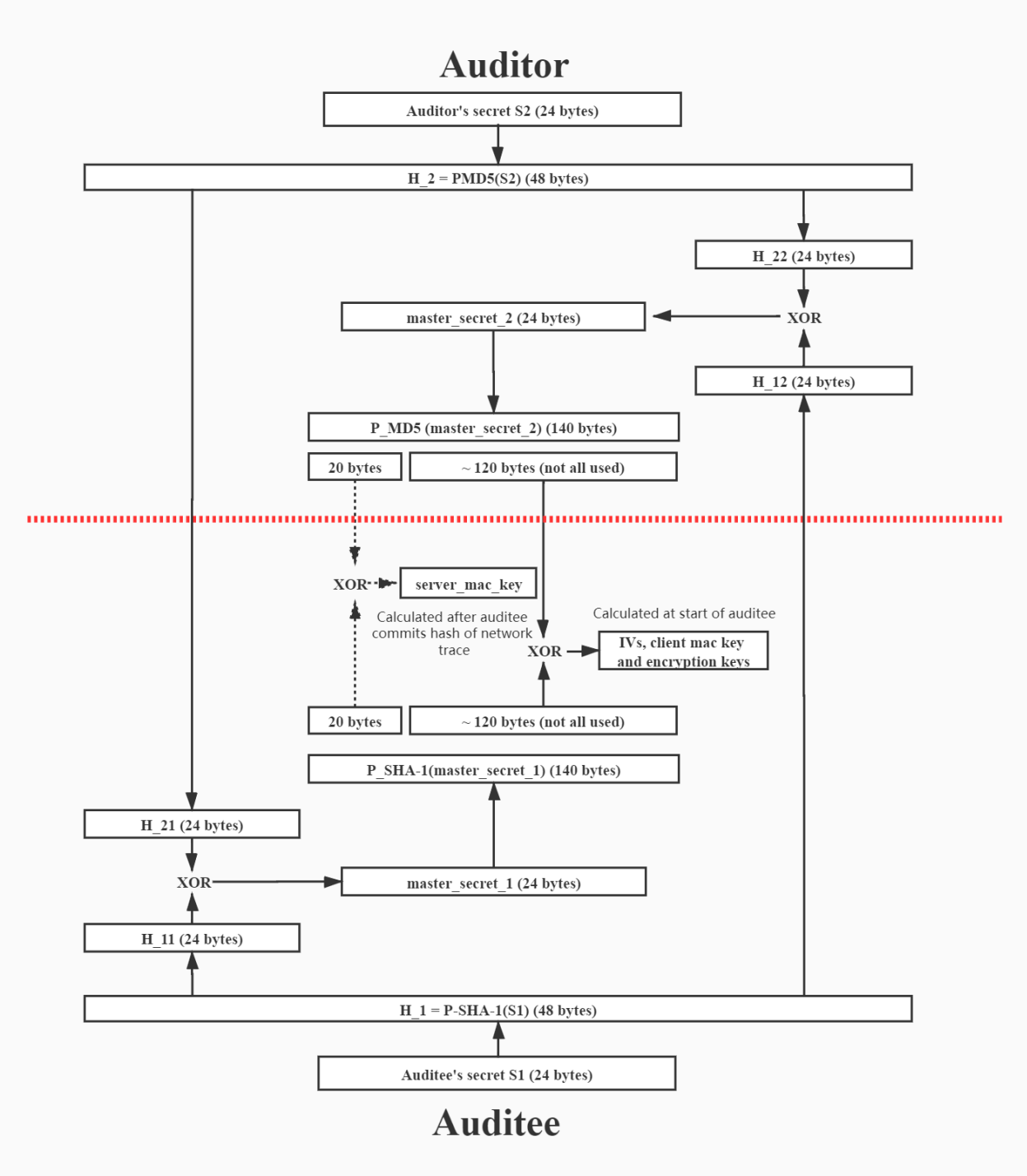

Figure 3: Main steps of implementation

Choose ubuntu x-86 (version of 18.04) image, create EC2 instance and run script. The reason for doing that is to enforce the Auditor to be a public-trusted server. The software and OS of the server are visible to public users. After the server, which runs the Auditor, starts up, the password then is re-modified, no one could control this server any longer after that.

4 Indicator Design

4.1 Functional Indicator

We mainly decide to do two functional tests: starting the TLSNotary to generate proof files, percentage of compatible websites.

Firstly, write a script to make 1000 tests of starting the project and generating proof files. Secondly, because TLSnotary is only compatible with TLS 1.0 and 1.1, the TLS 1.2 version and above changed the entire protocol architecture, so our project does not apply to it. Therefore, in order to prove that our project still can be applied to most websites, we design another functional experiment: Test if our project works on websites based on the top 100 sites on Alexa.

4.2 Latency indicator

When the user enters the URL to start TLSNotary, the system will obtain the data from the IP address indicated by the URL and finally generate a proof file after being authenticated by the Auditor. So there is a time latency between a starting point and a finishing point.

Therefore, we decide to design an experiment to test this latency: use the time package to set the start time when users start the TLSNotary, and then after the proof file is generated we set an end time, at last use the end time subtract the start time to compute the latency.

4.3 Security indicator

We know that the core of TLSNotay lies in the asymmetric encryption of RSA between the client and server to obtain the master secret, thereby generating a session key for data packet exchange. The purpose of the multiplicative homomorphic algorithm is to make the server receive complete pre-master secret when both Auditee and Auditor do not know each other's pre-master secret. Adding random factors in the process of encryption and decryption is to improve security and prevent various attacks. To be more detailed, in order to make the ciphertext generated by each encryption of the same plaintext different, a part of the random number is filled before encryption. So, we decide to use the number of random bytes to measure the security level.

However, it is not to say we should make random bytes as much as possible, because the more random bytes, the more likely it is to generate zero bytes. We use The RSA Encryption RFC 2313 encryption standard here. In order to satisfy the 2048-bit encryption specification of RSA, a padding sequence needs to be generated, and the padding sequence cannot have zero bytes. Therefore, there is a tradeoff between the number of random bytes and the success rate of TLSNotary. At last, we try to find the balance and test the security level.