Bulletproofs
Term Project Report
CS6500 Jan-May 2022

CS18B032 Girinath P
CS18B050 Aniswar Srivatsa Krishnan

May 16, 2022

Declaration

We declare that the work done in this project has not been and will be not used as-is for any other registered course at IITM; if this project code is extended further for any subsequent BTP/MTP project or MS/PhD thesis, this project work will be duly referenced in the reports/theses documents.

Abstract

We present the implementation of Bulletproofs, a new non-interactive zero-knowledge proof protocol with very short proofs and without a trusted setup and the proof size is only logarithmic in the witness size. Bulletproofs are especially well suited for efficient range proofs on committed values. They are useful for implementing confidential transactions in Bitcoin and other cryptocurrencies. We also implement the aggregation of range proofs, so that one can prove that m commitments lie in a given range by providing only an additive $O(log(m))$ group elements over the length of a single proof.

Introduction

Confidential transactions are used to obfuscate transaction amounts. A transaction contains some inputs and some outputs. For e.g. if A has \$10 and B has \$0 initially, and A wants to send \$4 to B. The input of this transaction is \$10. The outputs of this transaction are \$4 (the one sent to B) and \$6 (the amount remaining in A's wallet). We observe that two key properties of a transaction are:

1. The sum of inputs minus outputs in the transaction is equal to 0.
2. All the outputs are positive numbers (this is a property which prevents a person from spending an amount which is greater than the amount that they have).

When a transaction amount is encrypted, the first property above is proved using homomorphic encryptions (commonly Pedersen commitments). The second property is proven with the help of range proofs.

Range proofs do not leak any information about the secret value, other than the fact that they lie in a certain interval. Due to this fact, they are also referred to as zero-knowledge proofs. Bulletproofs are well suited for giving efficient range proofs. For this reason, it has been adopted in the backend of the Monero cryptocurrency. Bulletproofs rely on a variation of the discrete logarithm assumption to secure the proofs against a PPT (Probabilistic Polynomial Time) adversary. Our major goal in this project is to implement the bulletproof protocol in the Python language. We also implement certain extensions to the core bulletproof technique, namely the Fiat-Shamir heuristic and Aggregate Range Proofs.

Proof Protocols

Pedersen Commitments

A commitment algorithm is defined as follows:

Definition (Commitment)**.** A non-interactive commitment scheme consists of a pair of probabilistic polynomial time algorithms (Setup, Com). The setup algorithm $pp \leftarrow \mathsf{Setup}(1^\lambda)$ generates public parameter pp for the scheme, for security parameter λ. Given a message space M_{pp}, randomness space R_{pp} and commitment space C_{pp}, commitment algorithm Comm defines a function $M_{pp} \times R_{pp} \to C_{pp}$. Given a message $\mathbf{x}$, the algorithm draws a random element $\mathbf{r}$ from R_{pp} and computes $\mathbf{comm} = \mathsf{Comm}(\mathbf{x};\mathbf{r})$

Definition (Homomorphic Commitment)**.** A commitment scheme is a homomorphic commitment if $\forall x_1, x_2 \in M_{pp}$and $\forall r_1, r_2 \in R_{pp}$ we have that,

$$\mathsf{Com}(x_1 + x_2; r_1 + r_2) = \mathsf{Com}(x_1; r_1) + \mathsf{Com}(x_2; r_2)$$

Homomorphic Commitment property is required for us because we need to verify that sum of inputs is same as sum of outputs, and homomorphic property ensures that we can do addition of encrypted values to get encrypted value of sum.

Definition (Binding Commitment)**.** A commitment scheme is said to be computationally binding if it is computationally hard to find $(x_2; r_2) \neq (x_1; r_1)$ such that $\mathsf{Comm}(x_1; r_1) = \mathsf{Comm}(x_2; r_2)$

Definition (Hiding commitment)**.** A commitment scheme is said to be perfectly Hiding if it is impossible to distinguish 2 commitment values based on the original x values. It means that the commitment values leak no information about x

Now let us see the how Pedersen commitment is defined.

Definition (Pedersen Commitment)**.** The message space and random space is $\mathbb{Z}_p$ and commitment space is a group $\mathbb{G}$ of order p, p being a prime.
$\mathsf{Setup} : g, h \xleftarrow{\$} \mathbb{G}$
$\mathsf{Com}(x; r) : g^x h^r$

Under discrete log difficulty assumption, Pedersen commitment is computationally binding and perfectly hiding.

Inner Product Proofs

Let $\mathbb{G}$ denote a cyclic group of prime order p , and let $\mathbb{Z}_p$ denote the ring of integers modulo p. Let $\mathbb{G}^n$ and $\mathbb{Z}_p^n$ be vector spaces of dimension n over $\mathbb{G}$ and $\mathbb{Z}_p$ respectively. Let $\mathbb{Z}_p^*$ denote $\mathbb{Z}_p \backslash \{0\}$. All bold variables denote vectors. Let $\langle \mathbf{a}, \mathbf{b} \rangle = \Sigma a_i.b_i$ denote the inner product between two vectors $\mathbf{a}$, $\mathbf{b} \in \mathbb{G}^n$. And $\mathbf{a} \circ \mathbf{b} = (a_1.b_1, ...a_n.b_n)$ be the Hadamard product or entry wise multiplication of two vectors. For a vector $\mathbf{g} = (g_1, ...g_n) \in \mathbb{G}^n$ and $\mathbf{a} = (a_1...a_n) \in \mathbb{Z}_p^n$, $\mathbf{g}^\mathbf{a}$ is defined as $\prod g_i^{a_i} \in \mathbb{G}$. For a $k \in \mathbb{Z}_p$, $\mathbf{k}^n = (1, k, k^2..k^{n-1})$.

Requirement. Given generators $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u \in \mathbb{G}$, a scalar $c \in \mathbb{Z}_p$, and $P \in \mathbb{G}$. The argument lets the prover convince a verifier that the prover knows two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{G}^n$ such that

$$P = \mathbf{g}^\mathbf{a}.\mathbf{h}^\mathbf{b}.u^c \text{ and } c = \langle \mathbf{a}, \mathbf{b} \rangle$$

For proving this, the prover and verifier engage in the following protocol:

```
input: $(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u, P \in \mathbb{G} \;;\; \mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n)$
        $\mathcal{P}_{\mathsf{IP}}$'s input: $(\mathbf{g}, \mathbf{h}, u, P, \mathbf{a}, \mathbf{b})$
        $\mathcal{V}_{\mathsf{IP}}$'s input: $(\mathbf{g}, \mathbf{h}, u, P)$
output: $\{\mathcal{V}_{\mathsf{IP}}$ accepts or $\mathcal{V}_{\mathsf{IP}}$ rejects$\}$
        if $n = 1$ :
            $\mathcal{P}_{\mathsf{IP}} \rightarrow \mathcal{V}_{\mathsf{IP}} : a, b \in \mathbb{Z}_p$
            $\mathcal{V}_{\mathsf{IP}}$ computes $c = a \cdot b$ and checks if $P = g^a h^b u^c$:
                if yes, $\mathcal{V}_{\mathsf{IP}}$ accepts; otherwise it rejects

        else: $(n > 1)$
        $\mathcal{P}_{\mathsf{IP}}$ computes:
            $n' = \frac{n}{2}$
            $c_L = \langle \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]} \rangle \in \mathbb{Z}_p$
            $c_R = \langle \mathbf{a}_{[n':]}, \mathbf{b}_{[:n']} \rangle \in \mathbb{Z}_p$
            $L = \mathbf{g}_{[n':]}^{\mathbf{a}_{[:n']}} \mathbf{h}_{[:n']}^{\mathbf{b}_{[n':]}} u^{c_L} \in \mathbb{G}$
            $R = \mathbf{g}_{[:n']}^{\mathbf{a}_{[n':]}} \mathbf{h}_{[n':]}^{\mathbf{b}_{[:n']}} u^{c_R} \in \mathbb{G}$

        $\mathcal{P}_{\mathsf{IP}} \rightarrow \mathcal{V}_{\mathsf{IP}} : L, R$

        $\mathcal{V}_{\mathsf{IP}} : x \xleftarrow{\$} \mathbb{Z}_p^\star$
        $\mathcal{V}_{\mathsf{IP}} \rightarrow \mathcal{P}_{\mathsf{IP}} : x$

        $\mathcal{P}_{\mathsf{IP}}$ and $\mathcal{V}_{\mathsf{IP}}$ compute:
            $\mathbf{g}' = \mathbf{g}_{[:n']}^{x^{-1}} \circ \mathbf{g}_{[n':]}^{x} \in \mathbb{G}^{n'}$
            $\mathbf{h}' = \mathbf{h}_{[:n']}^{x} \circ \mathbf{h}_{[n':]}^{x^{-1}} \in \mathbb{G}^{n'}$
            $P' = L^{x^2} P R^{x^{-2}} \in \mathbb{G}$
        $\mathcal{P}_{\mathsf{IP}}$ computes:
            $\mathbf{a}' = \mathbf{a}_{[:n']} \cdot x + \mathbf{a}_{[n':]} \cdot x^{-1} \in \mathbb{Z}_p^{n'}$
            $\mathbf{b}' = \mathbf{b}_{[:n']} \cdot x^{-1} + \mathbf{b}_{[n':]} \cdot x \in \mathbb{Z}_p^{n'}$
        recursively run Protocol 2 on input $(\mathbf{g}', \mathbf{h}', u, P'; \mathbf{a}', \mathbf{b}')$
```

Range Proofs

Requirement. Let $v \in \mathbb{Z}_p$ and let $V \in \mathbb{G}$ be a Pedersen commitment to v using randomness γ. The prover will convince the verifier that $v \in [0, 2^n - 1]$.

To prove that $v \in [0, 2^n - 1]$, we introduce 2 vectors $\mathbf{a}_L$ and $\mathbf{a}_R$ and require that

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \text{ and } \mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}^n \text{ and } \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n$$

The three conditions above make sure that $\mathbf{a}_L$ is the bitwise representation of v. This is because if any non 1 or zero value was present in $\mathbf{a}_L$, then second condition won't be satisfied.

The above is proved by using 2 random challenge values y, z got from verifier,

$$z^2.\langle \mathbf{a}_L, \mathbf{2}^n \rangle + z.\langle \mathbf{a}_L - \mathbf{a}_R - \mathbf{1}^n, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = z^2.v$$

which is same as

$$\langle \mathbf{a}_L - z.\mathbf{1}^n, \mathbf{y}^n \circ (\mathbf{a}_R + z.\mathbf{1}^n) + z^2.\mathbf{2}^n \rangle = z^2.v + \delta(y,z) \tag{1}$$

$$\delta(y,z) = (z - z^2).\langle \mathbf{1}^n, \mathbf{y}^n \rangle - z^3.\langle \mathbf{1}^n, \mathbf{2}^n \rangle$$

Note that $\delta(y,z)$ can be calculated by the verifier too.

Since prover can't send $\mathbf{a}_L$ and $\mathbf{a}_R$ to verifier directly(as it would leak v), we introduce 2 blinding vectors $\mathbf{s}_L, \mathbf{s}_R$. The protocol starts as follows :

```
$\mathcal{P}_{\mathsf{IP}}$ on input $v, \gamma$ computes:
    $\mathbf{a}_L \in \{0,1\}^n$ s.t.$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$
    $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n \in \mathbb{Z}_p^n$
    $\alpha \xleftarrow{\$} \mathbb{Z}_p$
    $A = h^\alpha \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R} \in \mathbb{G}$          // commitment to $\mathbf{a}_L$ and $\mathbf{a}_R$
    $\mathbf{s}_L, \mathbf{s}_R \xleftarrow{\$} \mathbb{Z}_p^n$          // choose blinding vectors $\mathbf{s}_L, \mathbf{s}_R$
    $\rho \xleftarrow{\$} \mathbb{Z}_p$
    $S = h^\rho \mathbf{g}^{\mathbf{s}_L} \mathbf{h}^{\mathbf{s}_R} \in \mathbb{G}$          // commitment to $\mathbf{s}_L$ and $\mathbf{s}_R$
$\mathcal{P} \rightarrow \mathcal{V} : A, S$
$\mathcal{V} : y, z \xleftarrow{\$} \mathbb{Z}_p^\star$          // challenge points
$\mathcal{V} \rightarrow \mathcal{P} : y, z$
```

Now we introduce 3 polynomials $\mathbf{l}, \mathbf{r}, \mathbf{t}$.

$$\begin{aligned} l(X) &= (\mathbf{a}_L - z \cdot \mathbf{1}^n) + \mathbf{s}_L \cdot X && \in \mathbb{Z}_p^n[X] \\ r(X) &= \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot \mathbf{1}^n + \mathbf{s}_R \cdot X) + z^2 \cdot \mathbf{2}^n && \in \mathbb{Z}_p^n[X] \\ t(X) &= \langle l(X), r(X) \rangle = t_0 + t_1 \cdot X + t_2 \cdot X^2 && \in \mathbb{Z}_p[X] \end{aligned}$$

Now the prover can send the value of $\mathbf{l}(x), \mathbf{r}(x)$ for a random x to verifier because of blinding vectors which can hide the value of $\mathbf{a}_L$ and $\mathbf{a}_R$. t_0 value is the LHS in equation 1, so prover needs to show that it satisfies

$$t_0 = z^2.v + \delta(y,z)$$

To do this, prover shows that it has commitments to t_1, t_2 by checking the value of $\mathbf{t}(x)$ at a random x.

$\mathcal{P}_{\mathsf{IP}}$ computes:

$$\begin{aligned}
&\tau_1, \tau_2 \xleftarrow{\$} \mathbb{Z}_p \\
&T_i = g^{t_i} h^{\tau_i} \in \mathbb{G}, \quad i = \{1, 2\} && // \quad \textit{commit to } t_1, t_2
\end{aligned}$$

$\mathcal{P} \rightarrow \mathcal{V} : T_1, T_2$

$\mathcal{V} : x \xleftarrow{\$} \mathbb{Z}_p^\star$

$\mathcal{V} \rightarrow \mathcal{P} : x$ $\quad$ // *a random challenge*

$\mathcal{P}_{\mathsf{IP}}$ computes:

$$\begin{aligned}
&\mathbf{l} = l(x) = \mathbf{a}_L - z \cdot \mathbf{1}^n + \mathbf{s}_L \cdot x \in \mathbb{Z}_p^n \\
&\mathbf{r} = r(x) = \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot \mathbf{1}^n + \mathbf{s}_R \cdot x) + z^2 \cdot \mathbf{2}^n \in \mathbb{Z}_p^n \\
&\hat{t} = \langle \mathbf{l}, \mathbf{r} \rangle \in \mathbb{Z}_p && // \quad \hat{t} = t(x) \\
&\tau_x = \tau_2 \cdot x^2 + \tau_1 \cdot x + z^2 \cdot \gamma \in \mathbb{Z}_p && // \quad \textit{blinding value for } \hat{t} \\
&\mu = \alpha + \rho \cdot x \in \mathbb{Z}_p && // \quad \alpha, \rho \textit{ blind } A, S
\end{aligned}$$

$\mathcal{P} \rightarrow \mathcal{V} : \tau_x, \mu, \hat{t}, \mathbf{l}, \mathbf{r}$

Now verifier does the following to check if $\langle \mathbf{l}(x), \mathbf{r}(x) \rangle = \mathbf{t}(x)$

$$\begin{aligned}
h'_i &= h_i^{(y^{-i+1})} \in \mathbb{G}, \quad \forall i \in [1, n] && // \quad \mathbf{h}' = \left(h_1, h_2^{(y^{-1})}, h_3^{(y^{-2})}, \ldots, h_n^{(y^{-n+1})}\right) \\
g^{\hat{t}} h^{\tau_x} &\overset{?}{=} V^{z^2} \cdot g^{\delta(y,z)} \cdot T_1^x \cdot T_2^{x^2} && // \quad \textit{check that } \hat{t} = t(x) = t_0 + t_1 x + t_2 x^2 \\
P &= A \cdot S^x \cdot \mathbf{g}^{-z} \cdot (\mathbf{h}')^{z \cdot \mathbf{y}^n + z^2 \cdot \mathbf{2}^n} \in \mathbb{G} && // \quad \textit{compute a commitment to } l(x), r(x) \\
P &\overset{?}{=} h^\mu \cdot \mathbf{g}^{\mathbf{l}} \cdot (\mathbf{h}')^{\mathbf{r}} && // \quad \textit{check that } \mathbf{l}, \mathbf{r} \textit{ are correct} \\
\hat{t} &\overset{?}{=} \langle \mathbf{l}, \mathbf{r} \rangle \in \mathbb{Z}_p && // \quad \textit{check that } \hat{t} \textit{ is correct}
\end{aligned}$$

The last 2 checks can be combined as, for a random generator $u \in \mathbb{G}$,

$$P.h^{-\mu} \overset{?}{=} \mathbf{g}^{\mathbf{l}}.(\mathbf{h'})^r.u^{\langle \mathbf{l}, \mathbf{r} \rangle}$$

The above can be checked by using inner product proof, which concludes the range proof protocol.

Fiat-Shamir Heuristic

The protocol discussed above is an interactive protocol. It can be made into a non-interactive one by using fiat shamir hueristic. Note that the messages from verifier are all random challenge points. We now calculate these random values by using hashing technique. We hash the transcript upto the current point as the random challenge value. Transcript contains all the messages that would have been exchanged in case it were interactive along with some details of the proof itself, like for example the commitment value, n etc.

For example, in range proof protocol, y would be equal to $\mathsf{Hash}(V, n, A, S)$ and z would be equal to $\mathsf{Hash}(V, n, A, S, y)$.

Aggregating Range Proofs

A single transaction will involve more than one output. To provide range proofs for all of them can be done more efficiently than separately generating proofs for each output.

Let there be **m** transactions. Then, we can have $n * m$ sized vectors instead of n size in range proof, so that final number of messages would be $O(\log n * m)$.

Requirement. Let $\mathbf{v}, \boldsymbol{\gamma} \in \mathbb{Z}_p^m$ and $\mathbf{V} \in \mathbb{G}^m$, we need to show that $\forall j \in [1..m], V_j = g^{v_j}.h^{\gamma_j}$ and $v_j \in [0, 2^n - 1]$

The proof is quite similar to range proof with $n * m$ bits. $\mathbf{a}_L$ will now be $n * m$ sized, and will have the bits of all $v_j s$ concatenated. $L(X), R(X)$ will change as follows:

$$l(X) = (\mathbf{a}_L - z \cdot \mathbf{1}^{n \cdot m}) + \mathbf{s}_L \cdot X \in \mathbb{Z}_p^{n \cdot m}[X]$$

$$r(X) = \mathbf{y}^{n \cdot m} \circ (\mathbf{a}_R + z \cdot \mathbf{1}^{n \cdot m} + \mathbf{s}_R \cdot X) + \sum_{j=1}^{m} z^{1+j} \cdot \left(\mathbf{0}^{(j-1) \cdot n} \parallel \mathbf{2}^n \parallel \mathbf{0}^{(m-j) \cdot n}\right) \in \mathbb{Z}_p^{n \cdot m}$$

In the computation of τ_x, we need to adjust for the randomness of each commitment V_j, so that $\tau_x = \tau_1 \cdot x + \tau_2 \cdot x^2 + \sum_{j=1}^{m} z^{1+j} \cdot \gamma_j$. Further, $\delta(y, z)$ is updated to incorporate more cross terms.

$$\delta(y, z) = (z - z^2) \cdot \langle \mathbf{1}^{n \cdot m}, \mathbf{y}^{n \cdot m} \rangle - \sum_{j=1}^{m} z^{j+2} \cdot \langle \mathbf{1}^n, \mathbf{2}^n \rangle$$

In the last calculations done by verifier, the following are the changes in verification step and calculation of P

$$g^{\hat{t}} h^{\tau_x} \stackrel{?}{=} g^{\delta(y,z)} \cdot \mathbf{V}^{z^2 \cdot \mathbf{z}^m} \cdot T_1^x \cdot T_2^{x^2}$$

$$P = AS^x \cdot \mathbf{g}^{-z} \cdot \mathbf{h}'^{z \cdot \mathbf{y}^{n \cdot m}} \prod_{j=1}^{m} \mathbf{h}'^{z^{j+1} \cdot \mathbf{2}^n}_{[(j-1) \cdot n : j \cdot n - 1]}$$

Implementation

Modules

The following are the modules that have been implemented in our code:

Group

This module provides functions that can do various operations on elements of a certain group. An abstract class has been written with the different operations that can be performed with the group elements. The list of functions are:

- `exp(self, a)`: Raises the group element to the power a.
- `mult(self, a)`: Multiplies the group element with another group element a.
- `inverse(self, a)`: Returns the inverse of the group element.
- `serialize(self, a)`: Gives the string representation of the group element. Used for transporting the group element over the network.
- `deserialize(p, val)`: Constructs the group element from the string representation.
- `get_generators(n)`: Returns n generators of the group generated at random based on a seed which is fixed in the prover or the verifier.

A concrete implementation for this abstract class has been done for the group of integers modulo p and the `NIST P-256` elliptic curve group. The proof requires that the group order be a prime number. Therefore since the `NIST P-256` group satisfies this property, it is the group which has been chosen for enabling an implementation of our proof protocols.

The module also contains functions to do operations on vectors of group elements. The list of operations that can be performed on the vector are:

- `inner_prod(self,V)`: Returns the inner product of the vector with another vector V.
- `exp(self, V)`: If V is a vector it calculates the product of the element-wise exponentiations of the vector elements with corresponding elements of V. If V is an integer it returns a vector which contains elements of the vector raised to the power V.
- `mult(self, V)`: if V is a vector it calculates the elementwise product (Hadamard product) of the vector with V and returns a vector. If V is an integer it returns a vector where each element is the corresponding element of the original vector multiplied by V.
- `add(self, V)`: Adds the vector with the vector V.
- `serialize(self, a)`: Gives the string representation of the vector element. Used for transporting the vector element over the network.
- `deserialize(p, val, Grp=None)`: Constructs the vector element from the string representation.

Inner Product Proof

This module implements the inner product argument explained the previous section, in logarithmic time. A point to note is that due to the nature of this algorithm, we require that the size of the vectors passed to this module be a power of 2.

Range Proof

This module forms the core of the bulletproof technique and implements the range proof in logarithmic time taking help of the above inner product proof.

Fiat-Shamir Heuristic

This heuristic converts the interactive proofs above to a non-interactive proof using a hashing technique.

Aggregating Range Proof

A modification of the the range proof system enables us to conduct multiple range proofs with the proof size growing by an additive factor of $log_2(m)$ as opposed to a multiplicative factor of m when creating m independent range proofs.

Development Environment

The language used for the implementation of all the modules is Python 3. The following libraries were used:

- `argparse:` Used for parsing command line operations.
- `pycryptodome:` Used for cryptographic operations like elliptic curve group operations and inverse calculation.
- `random:` Used for generating a random value based on a certain seed.
- `secrets:` Used for generating cryptographically secure random numbers.
- `socket:` Used for communication between the prover and verifier.
- `sys:` Used for exit function.

The virtual machine in which the programs were tested is an Ubuntu 18.04 (64-bit) machine.

Conclusions

We have successfully implemented Bulletproofs in Python. This is a new non-interactive zero-knowledge proof protocol with very short proofs. We have also implemented two extensions to the core bulletproof technique, namely the Fiat-Shamir heuristic and Aggregate Range Proofs. The method to run our code was shown in the demo, and different cases (both successful verification and verification failure) were demonstrated.

Learning-Experience

1. Understanding of different cryptographic principles such as zero-knowledge proofs, range proofs, elliptic curve cryptographic, homomorphic encoding.
2. Increased familiarity with different types of mathematical proofs and their applications for practical security systems.
3. Basic understanding of the working of a secure cryptocurrency.
4. Experience in programming mathematical systems and algorithms.
5. Usage of abstract classes for defining certain prototypes and application programming interfaces (APIs) which can be instantiated concretely later on.

Suggestions

1. We gained a good understanding of different network security protocols and cryptographic systems through the course.

2. We have gained skills which will help us analyze any system from a network security perspective.

3. The assignments exposed us to the details of the protocols like Transport Layer Security (TLS) and also familiarized us with common cryptographic libraries such as OpenSSL.

4. The course project was an interesting aspect of the course which led us to read and understand some state-of-the-art systems and ongoing research projects. We also gained a good understanding of different current systems by watching the interim presentations given by the different students.

5. In the future versions of the course, a competition similar to capture-the-flag can be conducted which helps the students apply the various skills learned in the course in a fun and real-time manner.

References

[1] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, “Bulletproofs: Short proofs for confidential transactions and more,” in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 315–334, 2018.