ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ & ΑΛΓΟΡΙΘΜΩΝ

Designing Secure and Fair Protocols with Bitcoin

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αιμίλιου Δ. Τσουβελεκάκη

Επιβλέπων: Δημήτριος Φωτάκης, Επ. Καθηγητής Ε.Μ.Π.
Συνεπιβλέπων: Άγγελος Κιαγιάς, Επ Καθηγητής ΕΚΠΑ

Αθήνα, Οκτώβριος 2016

Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης & Υπολογιστων

Designing Secure and Fair Protocols with Bitcoin

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αιμίλιου Δ. Τσουβελεκάκη

Επιβλέπων: Δημήτριος Φωτάκης, Επ. Καθηγητής Ε.Μ.Π.
Συνεπιβλέπων: Άγγελος Κιαγιάς, Επ Καθηγητής ΕΚΠΑ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Οκτωβρίου 2016

.....	.....	.....
Δ. Φωτάκης	Α. Κιαγιάς	Α. Παγουρτζής
Επ. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής ΕΚΠΑ	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

.....

Αιμίλιος Δ. Τσουβελεκάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το ενδιαφέρον για το Bitcoin και τα υπολοιπα ψηφιακά κρυπτό-νομίσματα έχει παρουσιάσει αύξηση τα τελευταία χρόνια. Το Bitcoin είναι το πρώτο αποκεντρωμένο ψηφιακό κρύπτο-νόμισμα και το πιο δημοφιλές σε χρήση. Το συντακτικό του συστήματος συναλλαγών του Bitcoin μας επιτρέπει να δημιουργήσουμε έξυπνα συμβόλαια στα οποία η μεταφορά χρημάτων μπορεί να επιτευχθεί αυτόματα, αφού επιτευχθούν συγκεκριμένες προϋποθέσεις.

Σε αυτή τη διπλωματική εργασία παρουσιάζεται το P2P δίκτυο του Bitcoin μελετώντας τους κόμβους, τις ανταλλαγές μηνυμάτων, τις συναλλαγές, τα block και τα script. Στο υπόλοιπο μέρος της εργασίας, επικεντρωνόμαστε σε πρωτόκολλα ασφαλών υπολογισμών και βρίσκουμε τρόπους να εξαναγκάσουμε τη δικαιοσύνη σε περιβάλλον 2 παικτών ή N παικτών. Παρουσιάζονται πρωτόκολλα τα οποία έχουν εφαρμογή σε λοτταρίες, ασφαλείς υπολογισμούς, επαληθεύσιμους υπολογισμούς και σε ηλεκτρονικές ψηφοφορίες. Επεκτείνουμε κάποιες από τις προαναφερθείσες λειτουργίες που υπάρχουν στη βιβλιογραφία ώστε να πετύχουμε καλύτερη πολυπλοκότητα στο πρωτόκολλο ηλεκτρονικής ψηφοφορίας που έχει ήδη προταθεί και προτείνουμε ένα πρωτόκολλο το οποίο είναι αποκεντρωμένο και δε βασίζεται σε επικοινωνία των ψηφοφόρων μέσω ιδιωτικών καναλιών.

Το πρώτο μέρος είναι ένας πρόλογος σε αυτή την εργασία. Το δεύτερο μέρος επικεντρώνεται στο δίκτυο Bitcoin. Αρχικά, παρουσίαζουμε βασικές αρχές κρυπτογραφίας που χρησιμοποιούνται στο Bitcoin και μελετάμε την ανταλλαγή μηνυμάτων μεταξύ των κόμβων. Στη συνέχεια, επικετρωνόμαστε στις βασικές λειτουργίες οι οποίες είναι οι συναλλαγές και τα block. Επιπροσθέτως, μελετούμε τα Bitcoin Scripts τα οποία είναι χρήσιμα για την κατασκευή έξυπνων συμβολαίων. Το τρίτο μέρος είναι αφιερωμένο στους ασφαλείς υπολογισμούς και στην κατασκευή τέτοιων πρωτόκολλων στο μοντέλο τιμωρίας. Αναλύουμε πρωτόκολλα κρυπτογραφίας τα οποία θα χρησιμοποιηθούν αργότερα στην εργασία για την κατασκευή πρωτόκολλων ασφαλών υπολογισμών. Στα επόμενα 2 κεφάλαια εστιάζουμε σε εφαρμογές ασφαλών υπολογισμών μέσω Bitcoin. Στο τελευταίο κεφάλαιο μελετάμε πρωτόκολλα ηλεκτρονικής ψηφοφορίας και προτείνουμε το δικό μας πρωτόκολλο. Το τέταρτο μέρος αποτελεί τον επίλογο με τα συμπεράσματα και προεκτάσεις για μελλοντική εργασία.

Λέξεις-κλειδιά

Bitcoin, ασφαλείς υπολογισμοί, συμβόλαια, δίκαιη συναλλαγή, ηλεκτρονικές ψηφοφορίες, ιδιωτικότητα, αποκεντρωμένα συστήματα, κρυπτογραφία

Abstract

There has been an increasing interest in Bitcoin and other crypto-currencies the past few years. Bitcoin is the first decentralized crypto-currency that is currently by far the most popular one in use. The bitcoin transaction syntax is expressive enough to setup smart contracts whose fund transfer can be enforced automatically, after specified conditions are met.

This thesis studies the Bitcoin Peer to Peer (P2P) network and focuses on nodes, messages exchange, transactions, blocks and scripts. The rest of the thesis, is focused on secure multiparty computation and ways to enforce fairness in the two-party or multi-party setting. Several different protocols are presented which target specific applications like lottery, secure computation, verifiable computation and e-voting. We extend some of the aforementioned functionalities that exist in bibliography in order to achieve better complexity for the e-voting protocol already proposed and we propose our protocol which is decentralized and does not make use of any private channels among the voters.

The first part is a small introduction to this thesis. The second part focuses on the Bitcoin Network. At first, we present some basic cryptographic primitives that are used in Bitcoin. Then we focus on the network part of Bitcoin and the messages exchanged between peers. Afterwards, we focus on the core functionalities of Bitcoin which are transactions and blocks and occupy with Bitcoin Scripts as which are useful for constructing smart contracts. The third part is devoted to secure computation and how we can build up protocols for secure computation in the penalty model. We analyze cryptographic schemes which will be used later in this thesis for constructing securecomputation protocols. In the next two chapters we focus on applications of secure computation via Bitcoin. In the last chapter of this part, we study electronic voting protocols and we propose our own protocol. The fourth and last part of this thesis, is a conclusion and provides directions for future work.

Keywords

Bitcoin, secure computation, contracts, fair exchange, e-voting, privacy, decentralized systems, cryptography

Ευχαριστίες

Με την εκπόνηση αυτής της διπλωματικής εργασίας ολοκληρώνεται ο κύκλος των προπτυχιακών σπουδών μου στο Εθνικό Μετσόβιο Πολυτεχνείο και θα ήθελα μέσα από τις παρακάτω γραμμές να εκφράσω την ευγνωμοσύνη μου στους ανθρώπους που βρέθηκαν δίπλα μου κατα τη διάρκεια αυτών των ετών. Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντά μου κ. **Κιαγιά** για την εμπιστοσύνη που μου έδειξε και το χρόνο που μου αφιέρωσε ώστε να ολοκληρώσω αυτό το έργο καθώς και τους κ. **Φωτάκη** και κ. **Παγουρτζή** για την αποδοχή αυτής της εργασίας.

Στη συνέχεια, θα ήθελα να εκφράσω τις ευχαριστίες μου για τα ωραία φοιτητικά χρόνια τους συμφοιτητές μου **Ηλία Αλευρά**, **Νίκο Ανδρουλάκη**, **Φώτη Ηλιόπουλο**, **Φαίη Καζάκου**, **Σάκη Καραγιάννη**, **Χάρη Κοντούλη**, **Παναγιώτη Μαρινάκη**, **Μαρία Παπαμιχάλη**, **Νίκο Σταματόπουλο**, **Ελένη Τσακιράκη**, **Γιώργο Φουστούκο**, και τους εύχομαι κάθε επιτυχία για τη μετέπειτα σταδιοδρομία τους.

Σημαντικό ρόλο όλα αυτά τα χρόνια έπαιξαν οι παιδικοί μου φίλοι **Παναγιώτης Δήμας**, **Παναγιώτης Δωροβίνης**, **Γιώργος Νικολιάς**, που με ανέχτηκαν, με στήριξαν στις χαρές και στις λύπες και ήταν πάντα δίπλα μου.

Δε θα πρέπει να παραλείψω να ευχαριστήσω τη **Σοφία Παπαδακάκη** και το **Γιάννη Τουρνάκη** για την αμέριστη βοήθειά τους τόσο κατα τη διάρκεια της παραμονής μου στο CERN όσο και μετέπειτα, για τις πολύτιμες συμβουλές τους και για το χρόνο που μου διέθεσαν όποτε και αν τους χρειάστηκα.

Τέλος, οφείλω να ευχαριστήσω την οικογένεια μου για όλα όσα μου προσέφερε ώστε να φτάσω ως εδώ και να επιτύχω όσα έχω καταφέρει.

Αθήνα, Ιούνιος 2016

Εκτεταμένη Περίληψη

Εισαγωγή

Το Bitcoin είναι ένα ψηφιακό αποκεντρωμένο νόμισμα. Η πρώτη αναφορά σε αυτό έγινε από έναν ανώνυμο δημιουργό, δίχως να γνωρίζουμε αν είναι ένα άτομο ή μία ομάδα ατόμων, με όνομα Satoshi Nakamoto ο οποίος δημοσίευσε το paper με τίτλο Bitcoin: A peer-to-peer electronic cash system και ήταν ο δημιουργός του πρώτου Bitcoin client. Το Bitcoin έχει ως κύριο χαρακτηριστικό του τις συναλλαγές οι οποίες είναι δημόσιες και εισάγονται σε block ώστε να δημιουργήσουν το ιστορικό των συναλλαγών. Σε αντίθεση με το κανονικό χρήμα το Bitcoin δε βασίζεται σε κάποια κεντρική αρχή αλλά στο ίδιο το δίκτυο ώστε να επιβεβαιώσει τις συναλλαγές. Επιπροσθέτως, έχει προκαθορισμένο τρόπο εισαγωγής νομισμάτων σε κυκλοφορία μέχρι τον τελικό αριθμό των 21000000.

Σκοπός της διπλωματικής είναι να μελετήσουμε τη δημιουργία πρωτοκόλλων ασφαλών υπολογισμών με το Bitcoin να έχει το ρόλο της κεντρικής αρχής. Επειδή το Bitcoin είναι ένα αποκεντρωμένο σύστημα πρέπει να εξασφαλίσουμε ότι στο τέλος του πρωτοκόλλου όλοι οι συμμετέχοντες θα λάβουν το αποτέλεσμα. Για να το πετύχουμε αυτό, εφαρμόζουμε τη δικαιοσύνη μέσω του μοντέλου της τιμωρίας στο οποίο αν κάποιος κακοήθης χρήστης μάθει το αποτέλεσμα χωρίς να το διαμοιράσει στους υπόλοιπους χρήστες ή διακόψει το πρωτόκολλο τότε είναι αναγκασμένος να πληρώσει τους υπόλοιπους συμμετέχοντες. Συγκεκριμένα μελετάμε μία εφαρμογή των ασφαλών υπολογισμών, τις ηλεκτρονικές ψηφοφορίες.

Bitcoin

Σε αυτή την ενότητα θα παρουσίασουμε τα βασικά χαρακτηριστικά του Bitcoin. Το σύστημα αποτελείται απο διευθύνσεις και συναλλαγές. Κάθε χρήστης έχει τη δυνατότητα να δημιουργεί ένα ζεύγος κλειδιών, το δημόσιο και το ιδιωτικό κλειδί. Το δημόσιο κλειδί κωδικοποιείται σε μία διεύθυνση και χρησιμοποιείται για να λάβουμε χρήματα ενώ το ιδιωτικό κλειδί χρησιμοποιείται για να ξοδέψουμε χρήματα αποδεικνύοντας ότι ο χρήστης είναι ο πραγματικός κάτοχος των χρημάτων. Κάθε χρήστης έχει τη δυνατότητα να δημιουργήσει άπειρα ζεύγη κλειδιών τα οποία αντιστοιχούν και σε άπειρο αριθμό διευθύνσεων.

Η βασική δομή του δικτύου είναι οι συναλλαγές. Μία συναλλαγή μεταφέρει χρήματα από ένα κάτοχο σε έναν άλλο. Αναπαριστώντας τις συναλλαγές ως γράφο η κάθε συναλλαγή αποτελεί ένα κόμβο και έχει τουλάχιστον μία εισερχόμενη και τουλάχιστον μία εξερχόμενη ακμή. Η εισερχόμενη ακμή αντιπροσωπεύει ποιος ξοδεύει τα χρήματα και η εξερχόμενη ακμή ποιος λαμβάνει τα χρήματα. Κάθε ακμή περιλαμβάνει 2 επιπλέον στοιχεία τον ιδιοκτήτη των χρημάτων και το ποσό των χρημάτων που

μεταφέρεται. Όλες οι συναλλαγές δημοσιεύονται και έχουν τη δυνατότητα να τις δουν χρήστες οι οποίοι δε συμμετέχουν στο δίκτυο. Κάθε συναλλαγή έχει τη δική της μοναδική ταυτότητα η οποία είναι το SHA256(SHA256) των δεδομένων της συναλλαγής. Για να μεταφέρουμε χρήματα συνδέουμε μία ή περισσότερες εξερχόμενες ακμές σε μία ή περισσότερες εισερχόμενες ακμές. Τα χρήματα τα οποία ανήκουν σε κάθε χρήστη είναι αυτά τα οποία βρίσκονται σε αξόδευτες εξερχόμενες ακμές. Τα βήματα που ακολουθεί κάποιος χρήστης για να ξοδέψει τα χρήματά του είναι τα εξής:

1. Ο χρήστης που θέλει να ξοδέψει βρίσκει μία συναλλαγή με αξόδευτες εξερχόμενες ακμές και βεβαιώνεται ότι είναι ο ιδιοκτήτης αυτής της ακμής.
2. Δημιουργεί μία καινούρια συναλλαγή συνδέοντας την αξόδευτη εξερχόμενη ακμή, στην εισερχόμενη ακμή της καινούριας συναλλαγής.
3. Αφήνει την εξερχόμενη ακμή της νέας συναλλαγής ασύνδετη.
4. Καθορίζει την αξία και τον ιδιοκτήτη της νέας εξερχόμενης ακμής.

Μία συναλλαγή μπορεί να έχει πολλές ακμές ως εισόδους και πολλές ακμές ως εξόδους. Το ερώτημα που προκύπτει είναι τι θα συμβεί αν ένας κακοήθης χρήστης προσπαθήσει να ξοδέψει την ίδια εξερχόμενη ακμή 2 φορές. Η πρώτη συναλλαγή θα είναι έγκυρη ενώ η δεύτερη συναλλαγή θα είναι άκυρη. Αν είχαμε μία κεντρική αρχή θα ήταν εύκολο να διαχωρίσουμε τη συναλλαγή που προηγήθηκε στο χρόνο επειδή όμως το σύστημα είναι αποκεντρωμένο δεν υπάρχει αυτή η δυνατότητα. Δύο συναλλαγές οι οποίες ξοδεύουν την ίδια εξερχόμενη ακμή ονομάζονται διπλό ξόδεμα. Για να αποφύγουμε αυτό το πρόβλημα πρέπει να βάλουμε τις συναλλαγές σε μία σειρά. Συνεπώς το δίκτυο θα πρέπει να φτάσει σε μία συμφωνία όσον αφορά την ακολουθία των συναλλαγών.

Για να πετύχουμε αυτή την κοινή συμφωνία μαζεύουμε τις συναλλαγές σε κουτιά τα οποία ονομάζουμε block. Κάθε συναλλαγή μπορεί να περιλαμβάνεται μόνο μία φορά σε ένα block. Το δίκτυο φροντίζει να δημιουργείται ένα block κάθε 10 λεπτά το οποίο περιλαμβάνει τις πιο πρόσφατες συναλλαγές που δεν υπήρχαν σε προηγούμενα block. Η ταυτότητα του block είναι το SHA256 των δεδομένων που περιλαμβάνει. Κάθε συαλλαγή που περιλαμβάνεται σε έγκυρο block έχει επιβεβαιωθεί.

Για να υπάρξει συμφωνία στο δίκτυο, δημιουργούμε μία αλυσίδα απο block όπου το κάθε block αναφέρεται στο προηγούμενο μέσω ενός δείκτη που δείχνει στον πατέρα του. Συνεπώς κάθε block δεν μπορεί να περιέχει κάποια συναλλαγή η οποία κάνει διπλό ξόδεμα. Με αυτό τον τρόπο, επιτυγχάνουμε την κοινή συμφωνία στο δίκτυο συνεπώς η συναλλαγή Α προηγείται της συναλλαγής Β αν η Α περιλαμβάνεται σε προηγούμενο block από την Β. Η διαδικασία παραγωγής block ονομάζεται εξόρυξη και όλοι οι χρήστες έχουν τη δυνατότητα να συμμετέχουν. Τα βήματα της εξόρυξης είναι τα εξής:

1. Κάθε χρήστης παρακολουθεί το δίκτυο για συναλλαγές και block.
2. Περιλαμβάνουμε στο υποψήφιο block όλες τις συναλλαγές που δεν έχουν εμφανιστεί σε προηγούμενο block που γνωρίζουμε και μία αναφορά στο πιο πρόσφατο block που γνωρίζουμε ως πατέρα.
3. Αναζητούμε απόδειξη εργασίας.
4. Αν βρούμε απόδειξη εργασίας αναμεταδίδουμε το block στο δίκτυο.
5. Αν μάθουμε ότι κάποιος άλλος χρήστης βρήκε block, πετάμε την προηγούμενη δουλειά μας και

συνεχίζουμε να κάνουμε εξόρυξη πάνω στο πιο πρόσφατο block.

Το δίκτυο δίνει κίνητρο στους χρήστες ώστε να εξορύξουν τα block που διατηρούν την κοινή συμφωνία για την σειρά των συναλλαγών. Τα κίνητρα τα οποία έχει ένας χρήστης που βρίσκει ένα block είναι:

- Μία συναλλαγή η οποία ονομάζεται coinbase είναι η πρώτη συναλλαγή στο block και η αξία της σήμερα είναι 25 Bitcoin. Αυτή η αξία υποδιπλασιάζεται κάθε 210000 block.
- Από τα fees των συναλλαγών που συμπεριλαμβάνονται στο block που είναι η διαφορά στην αξία των εξερχόμενων ακμών από τις εισερχόμενες.

Η απόδειξη εργασίας είναι μία διαδικασία κατά την οποία αποδεικνύει ότι ξόδεψε επεξεργαστική ισχύ για να κάνει έναν υπολογισμό. Στην περίπτωση του Bitcoin η αποδείξη εργασίας που κάθε χρήστης κάνει είναι να υπολογίσει το SHA256 των συναλλαγών, του δείκτη που δείχνει στο block του πατέρα και μίας τυχαίας τιμής nonce. Το αποτέλεσμα αυτής της διαδικασίας πρέπει να είναι μικρότερο από μία συγκεκριμένη τιμή. Συνεπώς, για να βρούμε ένα τέτοιο hash πρέπει να κάνουμε πολλές δοκιμές είτε αλλάζοντας την τιμή nonce είτε διαλέγοντας περισσότερες συναλλαγές.

Ηλεκτρονικές Ψηφοφορίες

Οι ηλεκτρονικές ψηφοφορίες αποτελούν μία ειδική περίπτωση ασφαλών υπολογισμών. Για να υλοποιήσουμε πρωτόκολλα ηλεκτρονικής ψηφοφορίας χρειαζόμαστε μία κεντρική αρχή η οποία θα επιβάλλει τη δικαιοσύνη. Στη συγκεκριμένη περίπτωση η δικαιοσύνη εξασφαλίζεται από το δίκτυο μέσω του μοντέλου τιμωρίας. Έστω ότι έχουμε ν ψηφοφόρους και 2 υποψηφίους, τα πρωτόκολλα που κατασκευάζουμε θα πρέπει να ικανοποιούν τις εξής ιδιότητες:

- Ιδιωτικότητα: Η ψήφος κάθε ψηφοφόρου είναι μυστική ενώ ταυτόχρονα ο ίδιος μπορεί να αποδείξει ότι ακολούθησε το πρωτόκολλο.
- Επαληθευσιμότητα: Το αποτέλεσμα μπορεί να επιβεβαιωθεί από όλους τους ψηφοφόρους και από κάποιον εξωτερικό παρατηρητή.

Κάθε πρωτόκολλο ηλεκτρονικής ψηφοφορίας βασίζεται σε 2 στάδια, στο πρώτο στάδιο οι ψηφοφόροι δεσμεύονται στη ψήφο τους η οποία θα παραμείνει μυστική για τους υπόλοιπους χρήστες και μετά το αποτέλεσμα της ψηφοφορίας. Σε αυτό το στάδιο δεν είναι απαραίτητη η χρήση του Bitcoin. Στο δεύτερο στάδιο, οι ψηφοφόροι χρησιμοποιούν το Bitcoin για να ανακοινώσουν τις μυστικές ψήφους και ο νικητής μπορεί να πάρει το βραβείο του. Υλοποιούμε 2 διαφορετικά στάδια για την ανακοίνωση των ψήφων τα οποία και θα δούμε αργότερα.

Το συγκεκριμένο πρωτόκολλο λειτουργεί για ν ψηφοφόρους και 2 υποψηφίους. Για να διατηρήσουμε την ιδιωτικότητα της ψήφου θα μετατρέψουμε τις ψήφους σε τυχαίους αριθμούς, οι οποιοι περιλαμβάνουν και την ψήφο, όπου το άθροισμα θα μας δώσει το νικητή αναλόγως αν αυτό βρίσκετα πάνω από $\nu/2$ ή κάτω από αυτό. Κάθε ψηφοφόρος έχει μία ψήφο $V_i \in \{0, 1\}$ και δημιουργεί ν τυχαίους αριθμούς το άθροισμα των οποίων θα πρέπει να κάνει 0. Κάθε ψηφοφόρος P_i στέλνει τον αριθμό r_{ij} στον P_j μέσω ενός ιδιωτικού καναλιού. Οι ψηφοφόροι έχουν το δικό τους μοναδικό αριθμό R_i, δεσμεύονται σε αυτό τον αριθμό C_i, δημιουργούν τη μυστική ψήφο $\hat{V}_i = V_i + R_i$ και δεσμεύονται σε αυτή τη ψήφο $\hat{V}_i$ με δέσμευση $\hat{C}_i$. Όλες οι δεσμεύσεις δημοσιεύονται ενώ τα κλειδιά που ανοίγουν τις δεσμεύσεις και οι

ψήφοι παραμένουν μυστικές. Κάθε ψηφοφόρος πείθει τους υπόλοιπους ψηφοφόρους ότι ακολουθεί το πρωτόκολλο δίνοντας αποδείξεις μηδενικής γνώσης. Το στάδιο δέσμευσης της ψήφου δίνεται παρακάτω:

Το πρωτόκολλο τρέχει ανάμεσα σε ν ψηφοφόρους όπου για κάθε $i \in [n]$, ο ψηφοφόρος έχει μία μυστική ψήφο $V_i \in \{0, 1\}$. Θεωρούμε ότι τα κλειδιά για τα zk-SNARKs έχουν δημιουργηθεί και διαμοιρασθεί σε όλους τους ψηφοφόρους. Για κάθε ψηφοφόρο P_i η διαδικασία είναι η εξής:

1. Για κάθε $j \in [n]$ δημιούργησε ν τυχαίους αριθμούς όπου $\sum_j r_{ij} = 0$ και δεσμεύσου $(c_{ij}, k_{ij}) \leftarrow Commit(r_{ij})$
2. Δημιούργησε αποδείξεις μηδενικής γνώσης που δείχνουν ότι $\sum_j r_{ij} = 0$. Το κύκλωμα C δίνει 1 αν οι ανοιχθείσες τιμές έχουν άθροισμα 0. Αναμετάδωσε τις δεσμεύσεις και τις αποδείξεις μηδενικής γνώσης στους υπόλοιπους ψηφοφόρους.
3. Παρέλαβε τις δεσμεύσεις και επαλήθευσε τις αποδείξεις μηδενικής γνώσης από τους υπόλοιπους ψηφοφόρους.
4. Για κάθε $j \in [n]$ i, στείλε στον P_j το κλειδί k_{ij} το οποίο ανοίγει τη δέσμευση. Για $j \in [n]$ i, περίμενε το κλειδί k_{ji} το οποίο ανοίγει τη δέσμευση από τον P_j και έλεγξε αν $r_{ji} = Open(c_{ji}, k_{ji})$.
5. Υπολόγισε $R_i \leftarrow \sum_j r_{ji}$, $\hat{V}_i \leftarrow R_i + V_i$ και δεσμεύσου $(C_i, K_i) \leftarrow Commit(R_i)$, $(\hat{C}_i, \hat{K}_i) \leftarrow Commit(\hat{V}_i)$, όπου $K_i, \hat{K}_i$ είναι τα κλειδιά όπου ανοίγουν τις δεσμεύσεις. Αναμετάδωσε τις δεσμεύσεις C_i και $\hat{C}_i$ δημόσια.
6. Αναμετάδωσε τις αποδείξεις μηδενικής γνώσης για τα εξής:
   - $R_i = \sum_j r_{ji}$
   - Η δεσμευμένη τιμή $\hat{C}_i$ μείον την τιμή C_i είναι μεταξύ 0 και 1.
7. Παρέλαβε και επαλήθευσε τις αποδείξεις από τους υπόλοιπους ψηφοφόρους. Το πρωτόκολλο τερματίζει.

Figure 0.0.1: Στάδιο Δέσμευσης Μυστικής Ψήφου

Το ένα από τα 2 στάδια ανακοίνωσης της ψήφου βασίζεται στην λειτουργία Claim or Refund και στο μηχανισμό σκάλας. Οι ψηφοφόροι αποκαλύπτουν ο ένας μετά τον άλλο τις ψήφους τους με τον τελευταίο ωηφοφόρο να είναι αυτός που έχει το αποτέλεσμα. Η λειτουργία Claim or Refund αποτελείται από 3 στάδια:

- Στάδιο Κατάθεσης: Ο αποστολέας δημιουργεί μία συναλλαγή κατάθεσης όπου στέλνει X Bitcoins σε ένα παραλήπτη. Η συναλλαγή για να αποκτηθεί πρέπει είτε να δωθούν οι υπογραφές των 2 ατόμων είτε ο παραλήπτης να δώσει την υπογραφή του και μία απόδειξη ότι ικανοποιεί την απαίτηση του αποστολέα. Η συναλλαγή παραμένει μυστική. Ο αποστολέας δημιουργεί μία συναλλαγή αποζημίωσης όπου παίρνει πίσω τα X Bitcoins μετά από κάποιο χρονικό διάστημα τ και στέλνει τη συναλλαγή στον παραλήπτη να την υπογράψει. Αφού ο παραλήπτης υπογράψει, υπογράφει και ο αποστολέας τη συναλλαγή αποζημίωσης. Το στάδιο ολοκληρώνεται μόλις ο αποστολέας δημοσιεύσει τη συναλλαγή κατάθεσης.

- Στάδιο Απαίτησης: Ο παραλήπτης δημιουργεί μία συναλλαγή απαίτησης η οποία παίρνει τα X Bitcoins δίνοντας την υπογραφή του και την απόδειξη την οποία είχε απαιτήσει ο αποστολέας πριν το χρονικό διάστημα τ.
- Στάδιο Αποζημίωσης: Αν ο παραλήπτης δεν πάρει τα X Bitcoins μέχρι το καθορισμένο διάστημα τ, τότε ο αποστολέας χρησιμοποιεί τη συναλλαγή αποζημίωσης για να πάρει τα χρήματα του πίσω.

Στάδιο Κατάθεσης

- P_n δημοσιεύει 2 Claim or Refund instances, τα οποία δίνουν ν Bitcoins στο νικήτη της ψηφοφορίας αφού αποδείξει ότι το μεγαλύτερο άθροισμα ψήφων.

$$P_i \xrightarrow[n,\tau_{n+1}]{\hat{V}_1,...,\hat{V}_n} A$$
$$P_i \xrightarrow[n,\tau_{n+1}]{\hat{V}_1,...,\hat{V}_n} B$$

- Ταυτόχρονα για κάθε $i \neq n$, P_i επαληθεύει ότι η συναλλαγή κατάθεσης στο προηγούμενο βήμα βρίσκεται στην αλυσίδα των block και αναμεταδίδει την παρακάτω F^*_{CR} συναλλαγή κατάθεσης στο δίκτυο

$$P_i \xrightarrow[2,\tau_n]{\hat{V}_1,...,\hat{V}_n} P_n$$

- Σειριακά για i από n μέχρι 2: P_i επαληθεύει ότι οι συναλλαγές κατάθεσης στο προηγούμενο βήμα βρίσκεται στην αλυσίδα των block και αναμεταδίδει την παρακάτω F^*_{CR} συναλλαγή κατάθεσης στο δίκτυο

$$P_i \xrightarrow[i-1,\tau_{i-1}]{\hat{V}_1,...,\hat{V}_{i-1}} P_{i-1}$$

Στάδιο Απαίτησης

- Για $i \neq n$, αν πριν το χρονικό διάστημα τ_i, όλα τα προηγούμενα μυστικά $\hat{V}_1, ..., \hat{V}_{i-1}$ έχουν αποκαλυφθεί, τότε ο P_i αποκαλύπτει το μυστικό του $\hat{V}_i$ και χρησιμοποιεί τη συναλλαγή απαίτησης για να λάβει i bitcoins από τον P_{i+1}.
- Αν πριν το χρονικό διάστημα τ_n, όλα τα μυστικά $\hat{V}_i$ για $i \neq n$ έχουν αποκαλυφθεί, ο P_n αποκαλύπτει το μυστικό του $\hat{V}_n$ και χρησιμοποιεί τη συναλλαγή απαίτησης για να λάβει q bitcoins από κάθε P_i με $i \neq n$.
- Αν πριν το χρονικό διάστημα τ_{n+1} όλα τα μυστικά έχουν αποκαλυφθεί, ο νικητής καθορίζεται και χρησιμοποιεί την ανάλογη συναλλαγή απαίτησης για να λάβει ν bitcoins από τον P_n.
- Σε οποιοδήποτε χρονικό διάστημα, αν κάποια F^*_{CR} συναλλαγή δεν έχει πραγματοποιηθεί μέχρι το αντίστοιχο timelock, ο αποστολέας μπορεί να πάρει το ποσό που είχε καταθέσει πίσω μέσω μίας συναλλαγής αποζημίωσης.

Figure 0.0.2: Ανακοίνωση Μυστικής Ψήφου μέσω Μηχανισμού Σκάλας

Η πρώτη μας συνεισφορά γίνεται στην επέκταση της λειτουργίας Claim or Refund ανάμεσα σε ένα αποστολέα και πολλούς παραλήπτες, αντί για 1 παραλήπτη. Ο παραλήπτης μπορεί να απαιτήσει αυτή τη συναλλαγή αφού αποδείξη ότι κέρδισε και δώσει την υπογραφή του, ενώ η συναλλαγή του αποστολέα που του επιστρέφει τα νομίσματα του έχει υπογραφεί από τον ίδιο και όλους τους πιθανούς παραλήπτες. Με αυτό τον τρόπο επιτυγχάνουμε μείωση της πολυπλοκότητας στο πρώτο βήμα του σταδίου της κατάθεσης καθώς ο τελευταίος ψηφοφόρος δε θα πρέπει να δώσει 2N νομίσματα αλλά N(άρα και από 1 οι υπόλοιπο) και δε θα φορτώσουμε το δίκτυο με επιπλέον συναλλαγές.

Ο δεύτερος τρόπος ανακοίνωσης της μυστικής ψήφου, γίνεται ταυτόχρονα από όλους τους παίκτες και χρειάζεται σταθερό αριθμό γύρων bitcoin. Για να συμμετέχουμε στο πρωτόκολλο, κάθε ψηφοφόρος P_i καταθέτει (1+d) Bitcoins, εκ των οποίων το 1 θα δωθεί στο νικητή υποψήφιο εάν ο κάθε ψηφοφόρος αποκαλύψει την μυστική ψήφο του και το υπόλοιπο d Bitcoins θα χρησιμοποιηθούν ως αποζημίωση σε περίπτωση που ο ψηφοφόρος P_i δεν αποκαλύψει τη μυστική ψήφο.

Η ιδέα είναι ότι οι ν ψηφοφόροι θα υπογράψουν μία συναλλαγή COMPUTE απο κοινού όπου θα συνεισφέρουν τις ψήφους τους. Το πρωτόκολλο κλειδώνει τη ψήφο κάθε ψηφοφόρου και η τελική συναλλαγή COMPUTE μπορεί να εξαργυρωθεί αν όλοι οι ψηφοφόροι συμφωνούν ομόφωνα. Αντίθετα, αν το πρωτόκολλο ματαιωθεί και η συναλλαγή COMPUTE δεν έχει δημιουργηθεί με επιτυχία τότε οι ψηφοφόροι θα πάρουν το πόσο της συνεισφοράς πίσω χρησιμοποιώντας τη REFUND συναλλαγή. Λόγω των malleability problems που προκύπτουν θα χρησιμοποιήσουμε (n, n)-threshold signature scheme. Οι ν ψηφοφόροι δημιουργούν από κοινού μια ομαδική διεύθυνση έτσι ώστε κάθε ψηφοφόρος P_i μαθαίνει το δημόσιο κλειδί της ομάδας $\hat{pk}$ και κομμάτι του ιδιωτικού κλειδιού $\hat{sk}_i$. Συνεπώς, το ιδιωτικό κλειδί της ομάδας μπορεί να ανακατασκευαστεί μόνο μόνο εάν οι ν ψηφοφόροι παρέχουν το κομμάτι του ιδιωτικού κλειδιού τους $\hat{sk}_i$.

Στάδιο Κλειδώματος Ψήφου Κάθε ψηφοφόρος βάζει (1+d) Bitcoins στην αρχή του πρωτοκόλλου, όπου το 1 Bitcoin χρηματοδοτεί το νικητή, και τα d Bitcoins χρησιμοποιούνται σε περίπτωση διακοπής του πρωτοκόλλου

- Ο P_i δημιουργεί μία συναλλαγή $Lock_i$, με ακμή εισροής (1+d) Bitcoins τα οποία ανήκουν στον P_i και ως ακμή εκροής τη διεύθυνση του δημόσιου κλειδιού της ομάδας $\hat{pk}$. Ο P_i δημιουργεί μία απλοποιημένη συναλλαγή $Refund_i$ όπου μεταφέρει τα χρήματα απο τη συναλλαγή $Lock_i$ πίσω στον P_i και δεν αναμεταδίδει τη συναλλαγή στο δίκτυο. Ο P_i αναμεταδίδει την $Refund_i$ στους υπόλοιπους ψηφοφόρους.
- Λαμβάνοντας την $Refund_j$ για $j \in [n] \backslash \{i\}$, ο P_i ελέγχει ότι το hash της δεν είναι ίδιο με αυτό της συναλλαγής ($Lock_i$).
- Για κάθε $j \in [n]$, P_i συμμετέχει στο threshold signature scheme για να υπογράψει $Refund_j$ χρησιμοποιώντας το δικό του μυστικό κομμάτι του ιδιωτικού κλειδιού $\hat{sk}_i$.
- Λαμβάοντας τις σωστές υπογραφές για τη συναλλαγή $Refund_i$, ο P_i είναι έτοιμος να στείλει τη συναλλαγή $Lock_i$ στο δίκτυο.

Figure 0.0.3: Ανακοίνωση Μυστικής Ψήφου μέσω κοινής συναλλαγής - Στάδιο Κλειδώματος

Αφού οι ψηφοφόροι κλειδώσουν τις ψήφους τους θα υπογράψουν τη συναλλαγή COMPUTE χρησιμοποιώντας το (n, n)-threshold signature scheme. Η συναλλαγή COMPUTE έχει ν εισόδους (1+d) Bitcoins και ν+1 εξόδους, όπου η μία έξοδος είναι το βραβείο N Bitcoins για τον νικητή και οι υπόλοιπες ν έξοδοι για το deposit των ν ψηφοφόρων. Το deposit κάθε ψηφοφόρου θα επιστραφεί σε αυτόν αν αποκαλύψει τη ψήφο του ή αν χρησιμοποίησει την ομαδική υπογραφή. Επίσης κάθε ψηφοφόρος θα πρέπει να δημιουργήσει μία συναλλαγή PAY η οποία θα χρησιμοποιηθεί μόνο αν δεν ανοίξει τη ψήφο του μέσα σε κάποιο χρονικό διάστημα. Η διατύπωση του σταδίου υπολογισμού της ψήφου είναι:

Στάδιο υπολογισμού ψηφοφορίας Θεωρούμε ότι το προηγούμενο στάδιο έχει ολοκληρωθεί επιτυχώς και κάθε ψηφοφόρος P_i έχει δημιουργήσει μία συναλλαγή $LOCK_i$, όπου το hash είναι γνωστό δημοσίως.

- Οι ψηφοφόροι δημιουργούν από κοινού τη συναλλαγή COMPUTE
  - Έχει ν εισόδους απο τις συναλλαγές $Lock_i$ με ποσό κάθε συναλλαγής (1+d) Bitcoins
  - Έχει ν+1 εξόδους
    1. $deposit_i$, $i \in [n]$: κάθε συναλλαγή έχει αξία d Bitcoins και απαιτεί το κλειδί $\hat{K_i}$ και την υπογραφή του P_i's η οποία επαληθεύει το δημόσιο κλειδί pk_i ή μία έγκυρη υπογραφή η οποία είναι επαληθεύσιμη από το δημόσιο κλειδί $\hat{pk}$ της ομάδας.
    2. prize: έχει αξία N Bitcoins και απαιτεί όλα τα κλειδιά που ανοίγουν τις ψήφους K_i's και την υπογραφή του υποψήφιου νικητή.
- Οι ψηφοφόροι υπογράφουν απο κοινού τη συναλλαγή COMPUTE χρησιμοποιώντας το threshold signature scheme, ο καθένας με το δικό του κομμάτι ιδιωτικού κλειδιού sk_i.
- Κάθε ψηφοφόρος δημιουργεί, για κάθε $i \in [n]$, την ίδια απλοποιημένη συναλλαγή PAY_i με χρονικό όριο τ_2 όπου η εισερχόμενη ακμή αναφέρεται $out\_deposit_i$. Η εξερχόμενη ακμή λειτοργεί ως αποζημίωση d Bitcoins αν ο ψηφοφόρος P_i δεν αποκαλύψει τη ψήφο του μέχρι το χρονικό διάστημα τ_2 . Για παράδειγμα, όταν το $d = 2n$, η αποζημίωση μπορεί να διαμοιραστεί ανάμεσα στους 2 υποψηφίους. Οι ν ψηφοφόροι υπογράφουν από κοινού τη συναλλαγή PAY_i χρησιμοποιώντας το threshold signature scheme.
- Κάθε ψηφοφόρος P_i επαληθέυει ότι τα προηγούμενα βήματα έχουν ολοκληρωθεί και δημοσιεύει τη συναλλαγή $LOCK_i$ στο δίκτυο.
- Όταν όλες οι συναλλαγές $LOCK_i$'s έχουν εμφανιστεί στο blockchain, η συναλλαγή COMPUTE δημοσιεύεται στο blockchain.
- Όσο η συναλλαγή COMPUTE δεν έχει εμφανιστεί στην αλυσίδα συναλλαγών, μέχρι το χρονικό διάστημα έστω τ_1, κάθε ψηφοφόρος P_i μπορεί να τερματίσει το πρωτόκολλο δημοσιόποιόντας τη συναλλαγή $BACK_i$ παίρνοντας πίσω (1 + d) Bitcoins.

Figure 0.0.4: Ανακοίνωση Μυστικής Ψήφου μέσω κοινής συναλλαγής - Στάδιο Υπολογισμού

Όταν εμφανιστεί η συναλλαγή COMPUTE στην αλυσίδα των συναλλαγών κάθε ψηφοφόρος μπορεί να πάρει πίσω d Bitcoins δημοσιεύοντας μία συναλλαγή η οποία δίνει το κλειδί που αποκαλύπτει τη δεσμευμένη ψήφο του. Αν όλοι οι ψηφοφόροι πάρουν πίσω τα d Bitcoins τους τότε ο νικητής μπορεί να καθοριστεί και να παραλάβει το βραβείο του. Αν κάποιος κακοήθης χρήστης διακόψει το πρωτόκολλο

τότε το βραβείο χάνεται όμως οι υποψήφιοι παίρνουν την εγγύηση μετά από χρονικό διάστημα τ.

Παρατηρούμε ότι τα 2 πρωτόκολλα που έχουν προταθεί έχουν περιορισμό ως προς τους υποψηφίους, λόγω του τρόπου με τον οποίο κρύβουμε την ψήφο αλλά και λόγω του μηχανισμού σκάλας, επείδη ανακατασκευάζουμε το αποτέλεσμα από όλους τους υποψηφίους. Ένα ακόμα μειονέκτημα είναι ότι δίνεται η δυνατότητα σε κάποιον κακοήθη χρήστη να διακόψει την ψηφοφορία χωρίς να μπορούμε να εξάγουμε αποτέλεσμα. Για αυτό το λόγο θα χρησιμοποιήσουμε ένα καινούριο τρόπο να κρύψουμε την ψήφο ενώ για την ανακοίνωση της ψήφου θα χρησιμοποιήσουμε τον τρόπο της κοινής συναλλαγής. Θεωρούμε ότι οι ψηφοφόροι έχουν πρόσβαση σε authenticated public channels όπου μπορούν να γράψουν μόνο αυτοί χωρίς να διαγράψουν ή να αλλάξουν αυτά που έχουν ήδη γράψει. Αυτό μπορεί να επιτευχθεί μέσω των ψηφιακών υπογραφών. Σε αυτό το πρωτόκολλο δε χρησιμοποιούνται ούτε ιδιωτικά κανάλια ανάμεσα στους ψηφοφόρους ούτε κάποιος τρίτος ως κεντρική αρχή.

Αρχικό Στάδιο

- Όλοι οι ψηφοφόροι επιλέγουν ένα τυχαίο αριθμό στο $\mathbb{Z}q$. Κάθε ψηφοφόρος P_j κρατά το a_j μυστικό και δημοσιεύει το $h_j = h^{a_j}$ και μία ποδείξη μηδενικής γνώσης ότι το h_j έχει κατασκευαστεί σωστά αποδεικνύοντας γνώση του a_j.
- Κάθε ψηφοφόρος P_i ελέγχει την εγκυρότητα των αποδείξεων μηδενικής απόδειξης και υπολογίζει την τιμή $h_i^{y_i} = \frac{\prod_{j=1}^{i-1} h_j}{\prod_{j=i+1}^{n} h_j}$.

Figure 0.0.5: Στάδιο Δημιουργίας Μυστικής Ψήφου μέσω Decisional Diffie Helmann

Στάδιο Ψηφοφορίας

- Κάθε ψηφοφόρος κατασκευάζει την ψήφο του ως $b_i = h_j^{x_i y_i} g^{v_i}$ όπου $V_i \in \{0, 1\}$ και δίνει μία απόδειξη γνώσης ότι το $V_i \in \{0, 1\}$.
- Κάθε ψηφοφόρος ανακοινώνει την ψήφο του b_i μέσω της COMPUTE συναλλαγής. Κάθε ψήφος είναι μία δέσμευση, μιας εισερχόμενης ακμής στη συναλλαγή $COMPUTE$, άρα ο τελευταίος ψηφοφόρος δε μπορεί να υπολογίσει το αποτέλεσμα πριν ψηφίσει.

Στάδιο Υπολογισμού Αποτελέσματος

- Αν όλοι οι ψηφοφόροι ανοίξουν τις δεσμεύσεις τους, αποκαλύψουν τα κλειδιά με τα οποία ανοίγουν οι δεσμευμένες ψήφοι τότε το αποτέλεσμα της ψηφοφορίας μπορεί να υπολογιστεί ως $u = log_g V$ όπου $V = \prod_{i=1}^{n} b_i = g^{\sum_{i=1}^{n} v_i}$. Ο υπολογισμός του διακριτού λογαρίθμου είναι ένα δύσκολο πρόβλημα, γνωρίζουμε ότι το αποτέλεσμα βρίσκεται $1 < v < n$ και μπορούμε να ψάξουμε για την τιμή V με διάφορους αλγορίθμους όπως τον baby step giant step.
- Αν κάποιοι από τους ψηφοφόρους δεν ανοίξουν τις δεσμεύσεις τους τότε θα χρησιμοποιήσουμε τη μέθοδο της επόμενης παραγράφου για να υπολογίσουμε το αποτέλεσμα μόνο με τους έντιμους ψηφοφόρους.

Figure 0.0.6: Στάδιο Ψηφοφορίας και Υπολογισμού Αποτελέσματος

Ψηφοφορία με τους έντιμους ψηφοφόρους

Σε αυτή την παράγραφο ενισχύουμε το πρωτόκολλο βάζοντας ακόμα ένα γύρο υπολογισμού όπου θα χρησιμοποιήσουμε μόνο τους έντιμους χρήστες για τον υπολογισμό του αποτελέσματος. Έστω L η ομάδα των ψηφοφόρων που δεν άνοιξαν τις δεσμεύσεις τους για τη συναλλαγή COMPUTE. Για να μπορέσουμε να υπολογίσουμε το αποτέλεσμα, κάθε ψηφοφόρος υπολογίζει το εξής $\hat{h_j}^{y_i} = \frac{\prod_{j\in\{i+1,\ldots,n\}} h_j}{\prod_{j\in\{1,..,i-1\}} h_j}$ μαζί με μία υπογραφή γνώσης που $log_g h_j = log_{\hat{h}_j} \hat{h}_j^{y_i}$.

Με αυτό τον υπολογισμό εξαλείφουμε τους χρήστες τους οποίους δεν άνοιξαν τις δεσμεύσεις τους. Εάν θέλουμε να υπολογίσουμε το αποτέλεσμα της ψηφοφορίας αρκεί να υπολογίσουμε το $u = log_g V$, όπου $V = \prod_{j\in\{L\}} \hat{h_j}^{y_i x_i} * h_j^{y_i x_i} * g_i^v = \prod_{j\in\{L\}} \hat{h_j}^{y_i x_i} * b_i$.

Ψηφοφορία με κ υποψηφίους

Θεωρούμε ότι έχουμε ν ψηφοφόρους και κ υποψηφίους. Διαλέγουμε μία τιμή m έτσι ώστε ο μικρότερος ακέραιος να είναι $2^m > n$. Η τροποποίηση που κάνουμε είναι ότι η κάθε ψήφος είναι κωδικοποιήμενη στην παρακάτω μορφή $V_i = \{2^0 c_0, 2^{(k-1)} c_1, 2^{(k-1)2} c_2, 2^{(k-1)2} c_2, ..., 2^{(k-1)m} c_{k-1}\}$ όπου c_j είναι ο αριθμός των ψήφων κάθε ψηφοφόρου. Για να υπολογίσουμε το V εργαζόμαστε όπως και προηγουμένως και τα c_j μπορούν να υπολογιστούν με τη βοήθεια του αλγορίθμου *knapsack*

Προεκτάσεις

Στη συγκεκριμένη διπλωματική μελετήσαμε τη δημιουργια πρωτοκόλλων που βασίζονται στο bitcoin ώστε να διατηρήσουμε τη δικαιοσύνη ανάμεσα στους παίκτες που τρέχουν κάθε φορά το πρωτόκολλο. Εστιάσαμε στη δημιουργία πρωτοκόλλων για τις ηλεκτρονικές ψηφοφορίες. Πιθανές προεκτάσεις σε αυτή την εργασία είναι:

- Περιγραφή περισσότερων εφαρμογών οι οποίες μπορούν να εφαρμόσουν δικαιοσύνη μέσω του μοντέλου τιμωρίας.
- Μείωση πολπλοκότητας στα πρωτόκολλα τα οποία μελετάει αυτή η εργασία.
- Απόδειξη ασφάλειας των πρωτοκόλλων στο universal composable framework.

Contents

1 Introduction **1**

1 Introduction **3**
1.1 Problem Statement . . . 3
1.2 Motivation . . . 3
1.3 Outline . . . 4

2 The Bitcoin Protocol **5**

1 Theoretical Background **7**
1.1 Hash Functions . . . 7
1.2 Proof of Work . . . 8
1.3 Merkle Trees . . . 9
1.4 Public Key Cryptography . . . 11
1.5 Digital Signatures . . . 11
1.6 Elliptic Curve Cryptogrphy . . . 12
1.7 Private Keys . . . 13
1.7.1 Encrypted Private Keys . . . 13
1.7.2 Mini Private Keys . . . 14
1.8 Generating the Public Key . . . 14

2 Bitcoin Network **15**
2.1 Node Types . . . 15
2.1.1 Full Blockchain Nodes . . . 16
2.1.2 Simple Payment Verification (SPV) Nodes . . . 16
2.2 Joining the Network . . . 17
2.3 Discovering Peers . . . 18
2.4 Messages . . . 18
2.4.1 Version Message . . . 19
2.4.1.1 Network Address Structure . . . 20
2.4.2 Version Acknowledge Message . . . 21
2.4.3 GetAddr Message . . . 21
2.4.4 Addr Message . . . 21
2.4.5 GetBlocks Message . . . 22
2.4.6 GetHeaders Message . . . 23
2.4.7 Reject Message . . . 23
2.5 Syncing inside the network . . . 24
2.5.1 Block Propagation . . . 24
2.5.2 Transaction Propagation . . . 25
2.6 Messages Part 2 . . . 25

2.6.1 Inventory Message . . . 25
2.6.2 GetData Message . . . 26

3 Bitcoin Architecture 27
3.1 Transactions . . . 27
3.1.1 Standard Transactions . . . 28
3.1.2 Coinbase Transactions . . . 31
3.1.3 Transaction Fees . . . 32
3.2 Blocks . . . 32
3.2.1 Block Header . . . 33

4 Bitcoin Ownership 35
4.1 Script Language and Script Construction . . . 35
4.2 Standard Transaction Scripts . . . 36
4.2.1 Pay to PubKey . . . 36
4.2.2 Pay to PubKey Hash . . . 37
4.2.3 Pay to ScriptHash . . . 38
4.2.4 MultiSig . . . 39
4.2.5 Nulldata . . . 40
4.3 Non Standard Transactions . . . 41
4.3.1 Anyone Can Spend . . . 41
4.3.2 Not Strict DER Encoding . . . 41
4.4 Signatures . . . 41
4.4.1 SIGHASH_ALL . . . 43
4.4.2 SIGHASH_SINGLE . . . 44
4.4.3 SIGHASH_NONE . . . 45
4.4.4 SIGHASH_ANYONECANPAY . . . 46

3 Secure Multiparty Computations via Bitcoin 47

1 Preliminaries 49
1.1 Standard Primitives . . . 49
1.1.1 Zero Knowledge Proofs . . . 49
1.1.2 Commitment Schemes . . . 50
1.1.3 Secret Sharing Schemes . . . 50
1.1.4 Public Verifiable Computation . . . 51
1.2 The Ideal/Real Paradigm . . . 52
1.3 Special Ideal Functionalities . . . 52
1.3.1 Definition of Ideal Functionality F^*_{CR} . . . 52
1.3.2 Definition of Ideal Functionality F^*_f . . . 53
1.3.3 Definition of Ideal Functionality F^*_{Rec} . . . 55
1.4 Bitcoin Based Timed Commitment Scheme . . . 56
1.5 Simultaneous Bitcoin Based Timed Commitment Scheme . . . 58

2 Designing Fair Protocols 59
2.1 Introduction . . . 59
2.2 Two Party Lottery . . . 60
2.2.1 Realizing a Two Party Lottery Protocol . . . 61
2.3 Multiparty Lottery . . . 62

2.3.1 Realizing a Multi Party Lottery Protocol . . . 62
2.4 Secure Two-Party Computation . . . 64
2.4.1 Realizing a Two Party Secure Computation Protocol . . . 64
2.5 Multi Party Secure Computation . . . 65
2.5.1 Non Malleable Secret Sharing . . . 65
2.5.2 Fair Reconstruction . . . 65
2.5.3 The Ladder Mechanism . . . 65
2.5.4 Framework for The Ladder Mechanism . . . 66
2.5.5 Realizing Multi Party Computation with Penalties Protocol . . . 67

3 Incentivizing Correct Computations 69
3.1 Introduction . . . 69
3.2 Verifiable Computation . . . 69
3.2.1 Definition of Ideal Functionality F^*_{exitCR} . . . 70
3.2.2 Incentivizing Public Verifiable Computation . . . 71
3.3 Fair Computation . . . 72
3.3.1 Definition of Ideal Functionality F^*_{ML} . . . 72
3.3.2 Bitcoin Enhancement Proposal . . . 73

4 Electronic Voting 75
4.1 Ballot Commitment via Randomization of Ballots . . . 75
4.2 Ballot Casting via Ladder Mechanism . . . 77
4.3 Ballot Casting via MultiLock Transactions . . . 78
4.4 Ballot Commitment via Decisional Diffie-Hellman . . . 80
4.5 Ballot Casting via Compute Trasaction . . . 82
4.6 Robustness . . . 82
4.7 Extending to k Candidates . . . 83

4 Conclusion 85

1 Conclusion 87
1.1 Summary . . . 87
1.2 Future Work . . . 88

Bibliography 89

List of Figures

0.0.1 Στάδιο Δέσμευσης Μυστικής Ψήφου . . . xiv
0.0.2 Ανακοίνωση Μυστικής Ψήφου μέσω Μηχανισμού Σκάλας . . . xv
0.0.3 Ανακοίνωση Μυστικής Ψήφου μέσω κοινής συναλλαγής - Στάδιο Κλειδώματος . . . xvi
0.0.4 Ανακοίνωση Μυστικής Ψήφου μέσω κοινής συναλλαγής - Στάδιο Υπολογισμού . . . xvii
0.0.5 Στάδιο Δημιουργίας Μυστικής Ψήφου μέσω Decisional Diffie Helmann xviii
0.0.6 Στάδιο Ψηφοφορίας και Υπολογισμού Αποτελέσματος xviii

1.3.1 Merkle Tree with even number of Leaves . . . 10
1.3.2 Merkle Tree with odd number of Leaves . . . 10

2.3.1 Initial Connection to The Network . . . 18
2.4.1 Generic Message Structure . . . 18
2.4.2 Version Message Payload . . . 19
2.4.3 Network Address Structure . . . 21
2.4.4 Addr Message Payload . . . 22
2.4.5 GetBlocks Message Payload . . . 22
2.4.6 Reject Message Payload . . . 23
2.5.1 Block Propagation . . . 24
2.5.2 Transaction Propagation . . . 25
2.6.1 Inventory Message Payload . . . 25
2.6.2 Inventory Vector Payload . . . 26
2.6.3 GetData Message Payload . . . 26

3.1.1 Transaction Structure . . . 28
3.1.2 Transaction Output Reference Computation . . . 29
3.2.1 Block Structure . . . 32
3.2.2 Block Header Structure . . . 33

4.2.1 Pay-to-Pubkey Structure . . . 36
4.2.2 Pay-to-PublicKey Hash Structure . . . 37
4.2.3 Pay-to-ScriptHash Structure . . . 38
4.2.4 Multisig Structure . . . 39
4.2.5 Nulldata Structure . . . 40
4.3.1 Anyone Can Spend Structure . . . 41
4.4.1 Signature Computation - SIGHASH ALL . . . 43
4.4.2 Signature Computation - SIGHASH SINGLE . . . 44
4.4.3 Signature Computation - SIGHASH NONE . . . 45
4.4.4 Signature Computation - SIGHASH ALL|SIGHASH ANYONECANPAY . . . 46

1.3.1 Claim or Refund Functionality Definitiion . . . 53
1.3.2 Secure Computation with Penalties Functionality Definition . . . 54
1.3.3 Secure Computation with Penalties Functionality Definition 2 . . . 55

1.3.4 Secure Computation with Penalties Functionality - Reconstruct Definition 55
1.4.1 Bitcoin Based Commitment Scheme 56
1.5.1 Simultaneous Bitcoin Based Timed Commitment Scheme 58

2.2.1 Two Party Lottery in Bitcoin 61
2.3.1 MultiParty Lottery in Bitcoin 63
2.4.1 Two Party Secure Computation 64
2.5.1 Framework for the Ladder Mechanism 66
2.5.2 Realizing F^*_{rec} in the F^*_{CR} hybrid model 67

3.2.1 Claim or Refund Functionality Definitiion with Exit Clause 70
3.2.2 Public Verifiable Computation Protocol 71
3.2.3 Public Verifiable Computation Protocol 71
3.3.1 The ideal functionality F^*_{ML} 72

4.1.1 Ballot Commitment Phase 76
4.2.1 Ballot Casting via Ladder Mechanism 77
4.3.1 Ballot Casting via MultiLock Transaction - Lock Phase 79
4.3.2 Ballot Casting via MultiLock Transaction - Compute Phase 80
4.4.1 Ballot Masking via Decisional Diffie Helmann 81
4.5.1 Ballot Casting via Compute Transaction 82

List of Tables

1.1.1 History of Hash Functions 8
1.7.1 Private Key Formats 13

2.6.1 Inventory Object Type Values 26

3.1.1 LockTime Values 28
3.1.2 Coinbase Values 31

4.2.1 Pay-to-Pubkey *scriptSig* Execution 36
4.2.2 Pay-to-Pubkey *scriptPubkey* Execution 37
4.2.3 Pay-to-PubkeyHash *scriptSig* Execution 37
4.2.4 Pay-to-PubkeyHash *scriptPubkey* Execution 38
4.2.5 Pay-to-ScriptHash *scriptSig* Execution 39
4.2.6 Pay-to-ScriptHash *scriptPubkey* Execution 39
4.2.7 Multisig *scriptSig* Execution 40
4.2.8 Multisig *scriptPubkey* Execution 40
4.4.1 Signature Hash Types 41

Part 1

Introduction

Chapter 1

Introduction

1.1 Problem Statement

One fundamental problem of Cryptography is Secure Multiparty Computations which studies whether mutually distrusting parties can jointly compute a function over their inputs while keeping these inputs private. It has been proven in literature that we can compute securely the output of the function [1] but we cannot guarantee fairness [2]. Therefore, fairness can only be achieved through a trusted third party or a central authority.

There are many cryptographic tasks which lie into the field of secure multiparty computations and some of them are lotteries, auctions, secure computation, verifiable computation and electronic voting. The thesis overviews some of these problems, provides their definitions and studies the protocols that exist in literature.

The thesis focuses on Electronic Voting which is a subfield of Secure Multiparty Computation. We study the protocols proposed and then we construct our own protocol based on Decisional Diffie Hellman and Bitcoin.

1.2 Motivation

The motivation for this thesis rises from the advent of decentralized cryptocurrencies, specifically Bitcoin. Bitcoin is the first decentralized electronic crypto currency. The design of Bitcoin was first described in a self published paper by Satoshi Nakamoto [3], which is used as pseudonym and no real name has been linked as of today, in October 2008. The first blockchain which is a public ledger containing all transactions broadcasted was created on January 3rd 2009, as the genesis block which is the first block of the blockchain references the title of an article published in The Times. Unlike paper cash or electronic cash, Bitcoin does not rely on a central authority. Instead, it relies on its network to verify and authenticate transactions, which are also made public for further verification. This new form of currency is also unique in that the number of coins in

circulation will increase in a pre-determined way until the goal of 21 million coins in circulation is reached sometime in the year 2140.

Our purpose is to study whether fairness can be provided without central authorities but with Bitcoin's decentralized nature. Therefore, users of the network guarantee the fair execution of the protocols. Specifically, we study the decentralized electronic voting problem.

1.3 Outline

This thesis consists of 4 parts with several chapters. The *first* part is the introduction which gives the problem statement, motivation for this work and the outline of this thesis.

The *second* part of the thesis covers the core functionality of Bitcoin as a system. It includes:

1. **Theoretical Background**: An introduction to some cryptographic primitives that are used in Bitcoin and they are going to be useful during this thesis.
2. **Bitcoin Network**: We evaluate Bitcoin network and we study how peers connect to the network and the necessary messages that are exchanged for their communication.
3. **Bitcoin Architecture**: We study the two core concepts of Bitcoin, transactions and blocks, giving an overview of their structure and how they work inside the network.
4. **Bitcoin Ownership**: Continuing from the previous chapter we analyze the script functionality of Bitcoin transactions which allows us to build more complex senarios with transactions and are usually referred as complex contracts.

The *third* part of the thesis discusses secure multiparty computation protocols built via Bitcoin. It includes:

1. **Preliminaries**: An introduction to some cryptographic schemes and definitions of some functionalities useful for building protocols in the next chapters.
2. **Designing Fair Protocols**: This chapter studies lottery and secure multiprty computation protocols with the use of Bitcoin.
3. **Incentivizing Correct Computations**: Continuing from the previous chapter we describe certain cryptographic tasks and the way we can build protocols to enforce users to provide the correct result.
4. **Electronic Voting**: We study the protocols already proposed and then we propose our protocol based on Decisional Diffie Hellman for ballot masking and Bitcoin for ballot casting.

The *fourth* part is a summary of the thesis and includes some directions for future work.

Part 2

The Bitcoin Protocol

Chapter 1

Theoretical Background

1.1 Hash Functions

A hash function is a mathematical function which takes an arbitrary finite length data and computes a fixed-length value based on the data, called a hash or a digest. This function has certain properties which make it important:

- Input can be a string of any size.
- It produces a fixed-length output. Bitcoin uses SHA-256 as hash function, the result is always 256 bits and does not depend on the length of the input.
- It is easy to compute and verify the hash for any given input. Computing the hash of an n-bit string should have a running time of O(n).

A cryptographic hash function is considered secure when it has the following three properties:

1. Pre-Image Resistance
2. Second-Image Resistance
3. Collision Resistance

An overview of these properties is provided below:

- **Pre-Image Resistance:** A hash function is pre-image resistant, if given a hash value h in the output space of the hash function, it is hard to find any message m such that $h = hash(m)$. This concept is related to that of one-way functions. A one-way function is easy to compute on every input, but hard to invert given the image of a random input.
- **Second-Image Resistance:** A hash function is second-image resistant, if given a message m_1, it is hard to find a different message m_2 such that $hash(m_1) = hash(m_2)$. The difference between this property and the collision resistance property is that the adversary is given the message m_1 and he should find a message m_2 where $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$.
- **Collision Resistance:** A hash function is collision resistant, if given two messages m_1 and m_2, it is hard to find a hash h such that $h = hash(k, m_1) = hash(k, m_2)$, where k is the hash

key.

We should note that collisions exist but it is hard to find them. A simple proof for the above assumption is that the domain space of the hash function contains all strings of all sizes while the range space contains strings of fixed-length size. As the domain space is infinite, it is bigger than the range space. Consequently, there must be strings which map to the same output string.

A very simple approach to find collisions in a hash function with an output size of 256 bits is to choose $2^{256} + 1$ distinct values, compute the hashes for each of them and then compare them all for duplicates. If we start picking strings at random and computing hashes we are going to find a collision long before trying $2^{256} + 1$ values. We can find a collision by only examining roughly the square root of the number of possible outputs, known as the birthday paradox attack. The algorithm described above for finding collisions is working in every hash function but since it's a bruteforce algorithm it is slow and it depends on the number of tries we have to perform. Below, there is a table containing several hash functions and the size and year that were broken. We should note that in 2005 a collision for SHA-1 was found but the hash is not considered broken.

Hash Scheme	**Year Constructed**	**Number of Bits**	**Year Broken**
MD4	1990	128	1992
MD5	1992	128	1994
SHA-1	1994	160	*
SHA-256	2005	256	?

Table 1.1.1: History of Hash Functions

It is very easy to construct a function that is not collision resistant. One example can be a function that takes a random size length string and it returns always the same result. If we want to specify that result then we can use the function $H(msg) = msg \ mod \ 2^{256}$. This fuction has an efficient method to find a collision since it returns only the last 256 bits of the input. We should note that the hash functions we use we suspect that they are collision resistant but there are no hash functions proven to be collision resistant.

Hash functions are very useful as they can serve as a message digest. This means that we do not need to remember the whole package we have but we can check the hash to see if we have seen it before or if we are provided the same file.

1.2 Proof of Work

A proof of work system is used to ensure that a party has spent a certain amount of work to provide a solution which can be easily verified by anyone. Producing a proof of work can be a random process with low probability. Therefore, a lot of attempts are required on average before a valid one is generated. Specifically, Bitcoin requires each block generated proves that a significant amount of work was invested during its creation. As a result, the cost to modify a particular block increases with every new block added to the blockchain. Consequently, dishonest peers who want to modify past blocks have to spend a bigger computational effort than honest peers who only want to add

new blocks to the blockchain. Bitcoin's mining system incorporates a proof of work system based on Adam Back's hashcash [4]. The advantages of this system are:

- The party providing the proof of work has invested a predefined amount of effort in order to create the proof.
- The proof can be verified efficiently.

The proof of work used in Bitcoin takes advantage of the apparently random nature of cryptographic hashes, which was explained in the previous section. For a party to prove that spent a certain amount of computational work to create a block, it must create a hash of the block header which does not exceed a certain value. In Bitcoin the hashing algorithm is double SHA-256 and the predefined structure is a hash less or equal than a target value T. We call this value threshold target and the goal is to find a hash that is numerically less than the target. Every time we want to change the result of the hash we change a variable which is called nonce usually by incrementing it by 1. The success probability of finding a nonce n for a given message (msg), such that $H = SHA256^2(msg||n)$ is less or equal to the target T is

$$P[H \leq T] = \frac{T}{2^{256}}$$

The expected number of trials performed by a party attempting to find a proof of work is, on average, the following amount of computations

$$T[H \leq T] = \frac{1}{P[H \leq T]} = \frac{2^{256}}{T}$$

Finally, it is easily and efficiently verifiable to check whether the nonce accompanied with the message is indeed a valid proof of work by simply evaluating

$$SHA256^2\left(msg \,||\, n\right) \leq T$$

1.3 Merkle Trees

Merkle Trees [5] are binary hash trees which allow the efficient verification of large sets of data and they were named by their inventor Ralph Merkle. Each block in the Bitcoin blockchain contains a group of transactions and these blocks are leaves in the Merkle tree. The cryptographic hash algorithm used in Bitcoin's merkle trees is SHA256 applied twice, also known as double-SHA256. A Merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the root, or Merkle root. Furthermore, if an adversary tries to change a data block at the bottom of the tree, then the hash pointer, one level up, won't match and even if he continues changing this block, the change will eventually propagate to the top of the tree where he won't be able to manipulate the hash pointer that we have stored. Consequently, any attempt to tamper with any piece of data will be detected by just referencing to the hash pointer at the Merkle root.

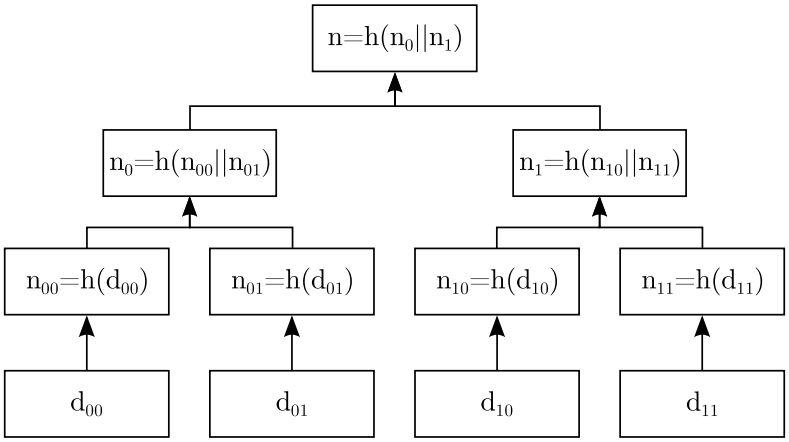

Figure 1.3.1: Merkle Tree with even number of Leaves

An example of a Merkle tree can be seen in the figure above [6]. Leaves are computed directly as hashes over data blocks, whereas parent nodes further up the tree are computed by concatenating their respective children. The process continues until there is only one node at the top, the node known as the Merkle Root. Data blocks in the bitcoin network are the transactions IDs. Merkle tree is a binary tree, thus it needs an even number of data blocks. If there is an odd number of data blocks the last one will be duplicated to create an even number of them, also known as a balanced tree. This is shown in the example below [6]:

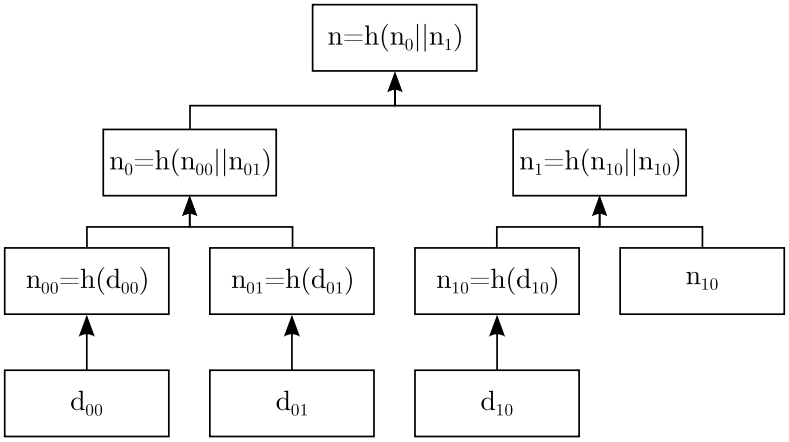

Figure 1.3.2: Merkle Tree with odd number of Leaves

The figures above are used as examples for constructing a tree which consists of four transactions, although the method shown can be generalized to construct trees of any size. In bitcoin it is common to have several hundread to more than a thousand transactions in a single block, which are summarized in exactly the same way producing just 32 bytes of data as the single merkle root.

Another advantage that Merkle Tree offers, unlike the blockchain, it allows a concise proof of membership. If we want to prove that a certain data block is a member of the Merkle Tree while we have only the root of the tree it is enough to show the path from this data block to the root. We can ignore the rest of the tree, as these blocks alone, allow us to verify the hashes all the way up to the root of the tree. Merkle Trees is a very efficient data structure because when N data elements

are hashed and summarized in a Merkle Tree, we can check if any data element is included in the tree with at most $2log_2N$ calculations.

Merkle trees are used extensively by Simplified Payment Verification nodes. SPV nodes don't have all transactions and do not download full blocks, just block headers. In order to verify that a transaction is included in a block, without having to download all transactions inside that block, they use an authentication path, or merkle path. More about Simplified Payment Verification Nodes in chapter 2.

1.4 Public Key Cryptography

Public-key cryptography introduces an absolutely new way of thinking about encryption, decryption and digital signing since it allows encrypted communication without private key exchange. In order to encrypt and decrypt messages, we create two different keys, or a key pair: the public key and the private key. It is computationally hard to deduce the private key from the public key. These mathematical functions are practically irreversible, meaning that they are easy to calculate in one direction and impossible to calculate in the reverse direction. Anyone with the public key can encrypt a message but not decrypt it. Only the person with the private key can decrypt the message. If we publish our public key, anyone is able to send us messages encrypted with it, and those messages cannot be read by anyone else than us. The mathematical formula deriving public key cryptography is the following, let C denote the encrypted message:

$$C = encrypt(M, Kpub)$$

$$M = decrypt(C, Kpri)$$

There are many public-key algorithms, and RSA algorithm is the most widely used. Bitcoin uses elliptic curve multiplication as the basis for its public key cryptography.

1.5 Digital Signatures

A digital signature is the second cryptographic primitive after hash functions that we need to build blocks. It should have two basic properties:

- One person can create his/her own signature and everyone else can verify the validity of the signature.
- We want this signature to be tied to a particular message so this signature can be used to signify the message you signed.

Public-key cryptography is used for digital signing in Bitcoin: we can find the hash of the message and encrypt it with the private key, thus forming a digital signature. If someone who has the public key receives the message with the digital signature, it is possible for him/her to verify both the authenticity and integrity of the message by decrypting the signature with the public key and

comparing the result to the hash of the message. The signed message also has the property of non-repudiation, that is, the sender is not able to falsely deny sending the message. The Bitcoin system must ensure who broadcasts the message so that the system knows that only one party can broadcast his own message. Steps for sending a message with contents:

1. Get the hash of the message: $Hash = SHA256(message)$
2. Encrypt hash with the private key to get signature: $S = encrypt(Hash, Kpri)$
3. Send the signature along with the message.

Steps for verifying a signature for a given message:

1. Take the hash of the message: $Hash = SHA256(msg)$
2. Decrypt signature with public key: $Hash' = decrypt(S, Kpub)$
3. Compare Hash with Hash'. If they are equal the message is valid.

The main Digital Signature Schema used by Bitcoin is the Elliptic Curve Digital Signature Algorithm (ECDSA), a variant of Digital Signature Algorithm (DSA) which uses Elliptic Curve Cryptography (ECC).

1.6 Elliptic Curve Cryptogrphy

Elliptic curve cryptography is a type of asymmetric public key cryptography based on the elliptic curve discrete logarithm problem. Elliptic curve cryptographic schemes were proposed independently in 1985 by Neal Koblitz [7] and Victor Miller [8]. It's a process that uses an elliptic curve and a finite field to sign data in such a way that third parties can verify the authenticity of the signature while the signer retains the exclusive ability to create the signature. Transferring ownership of bitcoins from user A to user B is realized by attaching a digital signature (using user A's private key) of the hash of the previous transaction and information about the public key of user B at the end of a new transaction.

This scheme derives from the following equation according to spec 2.4 [9]

$$y^2 = (x^3 + ax + b) \, mod \, p$$

Bitcoin uses a specific elliptic curve and set of mathematical constants, as defined in a standard called secp256k1, established by the National Institute of Standards and Technology (NIST). The secp256k1 curve is defined by the following function, which produces an elliptic curve:

$$y^2 = (x^3 + 7) \, mod \, p$$

1.7 Private Keys

A private key is a number, generated randomly. The private key provides control over all funds associated with the corresponding bitcoin address. It is used to create signatures that are required to spend bitcoins by proving ownership of funds used in a transaction. The private key must remain secret, as revealing it to a third party is equivalent to giving them control over the bitcoins secured by that key. The private key should be backed up and protected from accidental loss, since if lost it cannot be recovered and the funds secured by it are forever lost too. The private key can be represented in a number of different formats, all of which correspond to the same 256-bit number. These formats include:

Type	**Prefix**	**Description**
Hex	None	64 hex digits
Wallet Import Format	5	Base58Check encoding: Base-58 with version prefix of 128 and 32-bit checksum
Wallet Import Format Compressed	K or L	As above, with added suffix 0x01 before encoding
Mini	S	Less than 30 characters

Table 1.7.1: Private Key Formats

1.7.1 Encrypted Private Keys

BIP0038 [10] proposes a common standard for encrypting private keys with a passphrase and encoding them with Base58Check so that they can be stored securely on backup media, transported securely between wallets, or kept in any other condition where the key might be exposed. The advantage of encrypting your paper wallet's private key with a password is that if your paper wallet is stolen or otherwise exposed, the balance on the wallet is safe unless the passphrase used to encrypt the wallet is guessed. Although this means that the balance in your wallet is as safe as the passphrase you use. A BIP0038 encryption scheme takes as input a bitcoin private key, usually encoded in the Wallet Import Format (WIF), as a Base58Check string with a prefix of "5" [11]. Additionally, the BIP0038 encryption scheme takes a passphrase usually composed of several words or a complex string of alphanumeric characters. The result of the BIP0038 encryption scheme is a Base58Check-encoded encrypted private key that begins with the prefix 6P. If you see a key that starts with 6P, that means it is encrypted and requires a passphrase in order to decrypt it back into a WIF-formatted private key (prefix 5) that can be used in any wallet. Many wallet applications now recognize BIP0038-encrypted private keys and will prompt the user for a passphrase to decrypt and import the key.

1.7.2 Mini Private Keys

Mini private key format [12] is a method for encoding a private key in under 30 characters, enabling keys to be embedded in a small physical space, such as physical bitcoin tokens, and more damage-resistant QR codes. They usually have the prefix 'S'. In order to derive the full private key, the user simply takes a single SHA256 hash of the original mini private key. This process is one-way: it is intractable to compute the mini private key format from the derived key.

1.8 Generating the Public Key

The public key is derived from the private key. We start by creating a random 256-bit private key, k. We multiply it by a predetermined point on the curve called the generator point G to produce another point somewhere else on the curve, which is the corresponding public key Q. The generator point is specified as part of the secp256k1 standard and is always the same for all keys in bitcoin.

$$Q = k * G$$

Since the generator point is always the same for all bitcoin users, a private key k multiplied with G will always result in the same public key Q. The relationship between k and Q is fixed, that means that if the private key is very large it is easy to compute the public key but very difficult for an attacker to bruteforce.

Chapter 2

Bitcoin Network

The term Bitcoin Network refers to the group of nodes running the bitcoin P2P protocol. Every user connected to the network through a client is considered to be a Bitcoin node. Although Bitcoin is the main protocol there are other protocols such as Stratum, which provide functionality for mining, lightweight clients and mobile wallets.

2.1 Node Types

All Bitcoin nodes are considered equal inside the network although they might support a different functionality to it. A Bitcoin node has certain functionalities such as routing, blockchain database, mining and wallet services. All nodes include routing as a basic functionality and they differentiate in which of the rest functionalities include. Below there is a list of the most common node types on the extended bitcoin network [11]:

- **Reference Client:** Contains a wallet, miner, database and network routing node on the bitcoin P2P network.
- **Full Blockchain Node:** Contains a database and network routing node on the bitcoin P2P network.
- **Solo Miner:** Contains a mining function with database and network routing node on the bitcoin P2P network.
- **Light-Weight Wallet (SPV):** Contains a wallet database and network routing node on the bitcoin P2P network.
- **Pool Protocol Servers:** Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.
- **Mining Nodes:** Contain a mining function without a blockchain, with Stratum protocol or another pool protocol.
- **Light-Weight Stratum Wallet (SPV):** Contains a wallet and a network node on the Stratum protocol, without a blockchain.

2.1.1 Full Blockchain Nodes

In the first years of Bitcoin all nodes were full nodes, since the majority of them were using the reference client. Full Blockchain nodes maintain the most up-to-date copy of the blockchain with all the transactions. The construction of the blockchain starts with the genesis block and continues by building up and verifying until the latest known block in the network. This type of node can verify transactions autonomously without relying on any other node as an external reference. A full node relies only on the network to receive updates about new blocks of transactions, verifies them and then migrates them into its local copy of the blockchain. The blockchain nowadays is around 90 Gb and depending on your internet connection it might require a couple of hours or days to sync to the bitcoin network. The most known implementation is Bitcoin Core which was the first client available for Bitcoin originally developed by Satoshi Nakamoto.

2.1.2 Simple Payment Verification (SPV) Nodes

Portability is a huge factor these days. With the uprising of smartphones and tablets and due to restrictions in hardware storage and limitations in size of mobile applications, new clients have been introduced which do not maintain a copy of the blockchain and they are considered to be lightweight clients. This type of clients are called Simplified Payment Verification allow operating in the network without storing the blockchain. It is possible to build a Bitcoin implementation that does not verify everything, but instead relies on either connecting to a trusted node, or puts its faith in high difficulty as a proxy for proof of validity. SPV nodes connect to full nodes and download only the block headers without the transactions included in each block [11]. SPV nodes verify transactions using a slightly different methodology that relies on peers to provide partial views of relevant parts of the blockchain on-demand. Simplified Payment Verification verifies transactions by reference to their depth in the blockchain instead of their height. Whereas a full-blockchain node will construct a fully verified chain of thousands of blocks and transactions reaching down the blockchain all the way to the genesis block, an SPV node will verify the chain of all blocks and link that chain to the transaction of interest. As a further optimization, block headers that are buried sufficiently deep can be thrown away after some time. An SPV node cannot be tricked that a transaction exists when it does not because the SPV node establishes the existence of a transaction in a block by requesting a merkle path proof and by validating the proof-of-work in the chain of blocks. On the other hand, a transaction can be hidden from an SPV node due to the fact that it does not have the ability to verify that a transaction does not exist because it does not have a ledger of all transactions. To defend against attacks an SPV Node should connect randomly to several nodes, in order to increase the probability to have at least one honest node in its network. SPV nodes get only the block headers using a GetHeaders command. The responding peer will send up to 2000 block headers using a single headers message [11].

2.2 Joining the Network

Bitcoin works as a peer-to-peer network in which the participants jointly emulate the central authority that controls the correctness of transactions. In this section, we will explain how a client can join the network, which actions are performed on the network and how information is propagated through it. Before a Bitcoin node exists in the network, it needs to connect to a random peer, but it should find the first peer. In the past they existed 4 ways of finding the first peer, nowadays one of them is deprecated. Note that the method of finding peers applies for Bitcoin Core and not some other bitcoin client implementation [13].

1. The primary mechanism, if the client has ever run on this machine before and its database is intact, is to look at its database. It tracks every node it has seen on the network, how long ago it last saw it, and its IP address. Bitcoin Core will spend up to 11 seconds trying to connect to a peer in its database. If that doesn't work, it will follow the second mechanism; it will query a DNS Server.
2. DNS Seeds: The client issues DNS requests to learn the addresses of other peer nodes. The client includes a list of host names for DNS services that are seeded.
   - bitseed.xf2.org
   - dnsseed.bluematt.me
   - seed.bitcoin.sipa.be
   - dnsseed.bitcoin.dashjr.org
   - seed.bitcoinstats.com

   Addresses discovered via DNS are initially given a zero timestamp; therefore they are not advertised in response to a GetAddr request. This is the default seeding mechanism, as of v0.6.x and later. If that doesn't work within 60 seconds, it will fall back to one of its hardcoded addresses.
3. Hard Coded "Seed" Addresses: Should the DNS Addresses method fail, the client contains hardcoded IP addresses that represent bitcoin nodes. They are only used for finding other peers, as to avoid overload, the connection thread will close seed node connections when the local node has enough addresses. Seed Addresses are initially given a zero timestamp; therefore they are not advertised in response to a GetAddr request.
4. IRC Addresses: In addition to learning and sharing its own address, the node learned about other node addresses via an IRC channel. After learning its own address, a node encoded its own address into a string to be used as a nickname. Then, it randomly joined an IRC channel named between bitcoin00 and bitcoin99. Then it issued a WHO command. The thread read the lines as they appeared in the channel and decoded the IP addresses of other nodes in the channel. It did this in a loop, forever, until the node was shutdown. When the client discovered an address from IRC, it set the timestamp on the address to the current time, but it used a "penalty" of 51 minutes, which means it looked like it was actually seen almost an hour earlier. As of version 0.6.x the Bitcoin client no longer uses IRC bootstrapping by default, and as of version 0.8.2 support for IRC bootstrapping has been removed completely.

2.3 Discovering Peers

Once connected, the joining node learns about other nodes by asking their neighbours for known addresses and by listening spontaneous advertisements of new addresses. Figure below visualizes the connection of a user entering the network and connecting to a random peer.

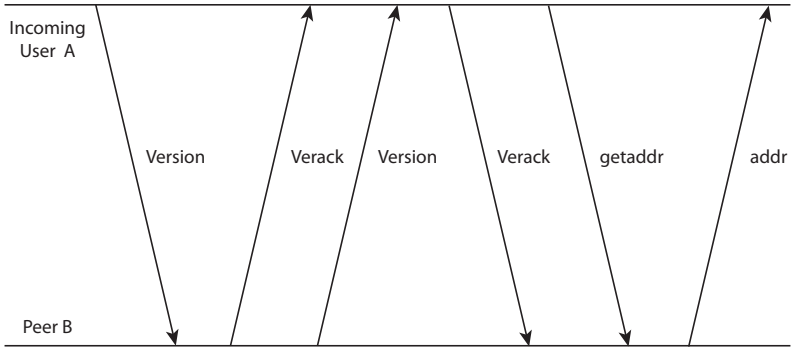

Figure 2.3.1: Initial Connection to The Network

The incoming user sends a version message containing version number, block count current time and several other information. The remote peer will send back a version acknowledge message and its own version message if it accepts connections from this user. User will respond with its own version acknowledge message if it accepts connections from node's version message. Right after connecting to the peer, the incoming user is a new node to the network and needs to learn about more active peers. Therefore, there is an exchange of GetAddr and Addr messages, storing all addresses that the new node does not know about. Addr messages usually contain only one address, but this number can be up to 1000 usually in cases where a new node enters the network. When a node receives a GetAddr request, it checks how many addresses it has, have a timestamp in the last 3 hours and it sends those addresses. We should note that the maximum nuber of nodes that a GetAddr request is 2500.

2.4 Messages

In this section we are going to analyze the stucture of the most used messages for connection. All messages in the network protocol use the same container format, which consists of two parts a required multifield header and an optional payload. The generic message structure is the following:

magic number
command
length
checksum
payload

Figure 2.4.1: Generic Message Structure

Magic Number (Network ID)
Network ID field is indicating the network which broadcasted the message and is used to seek the next message when stream state is unknown. This number consists of four bytes which start every message, for the main Bitcoin network is 0xD9B4BEF9. There are 3 more magic values for testnet, testnet3 and namecoin.

Command
Following the Network ID is a 12 byte field describing the command being sent in the message. This is usually an ASCII string identifying the packet content that is zero padded if needed to fill the length.

Length
Length fields contains the size of the payload for this specific packet in bytes.

Checksum
The next 4 bytes are a checksum of the payload. This is created by SHA256 hash of SHA256 hash of payload and then discarding everything after the first 4 bytes of this 32byte hash. If payload is empty the checksum is always 0x5df6e0e2 which is the result of double SHA256 of empty string.

Payload
Payload field contains the actual data which is defined according to which message is relayed and its structure. In case the packet has no payload, then payload length is zero(0x00) bytes. Payload decoding, depends upon the protocol version and the command.

2.4.1 Version Message

The version message provides information about the transmitting node to the receiving node at the beginning of a connection. Any other messages will be rejected until both clients peers have accepted version messages. Version message follows the general message structure, where in payload field contains the actual data of the version message and has the following structure:

version
services
timestamp
addr_recv
addr_from
nonce
user_agent
start_height
relay

Figure 2.4.2: Version Message Payload

Version
Version field identifies the protocol version being used by the node.

Services
Services field is a list of services supported by the node with this connection. The service which is currently available at this point is NODE_NETWORK. It defines that this node can be asked for full blocks instead of just headers.

Timestamp
Timestamp field contains the standard UNIX timestamp in seconds since epoch according to the transmitting node's clock.

Addr_recv
Addr_recv field is the network address of the node receiving this message. Network address is a data structure which will be analyzed in the next subsection.

Addr_from
Addr_from field is the network address of the node emitting this message. Network address is a data structure which will be analyzed in the next subsection.

Nonce
Nonce is a randomly generated unsigned integer every time a version packet is sent. This nonce is used to detect duplicate payloads (connections to it). If the nonce is 0, the nonce field is ignored. If the nonce is anything else, a node should terminate the connection on receipt of a version message with a nonce it previously sent.

User_agent
User agent field is a variable length string containing the name of the Node implementation.

Start_height
Start Height field is the last block received by the emitting node, also known as the height of the latest block in our database. As Bitcoin node adds more blocks to its database, this height grows as well. When the node finds another node with higher Block start address, it can request newer blocks from that node to update its database.

Relay
Relay field defines whether the remote peer should announce relayed transactions or not.

2.4.1.1 Network Address Structure

As we have seen above Addr_recv and Addr_from need a network address. Protocol defines a network address structure which is the following:

time
services
IPv4/6
port

Figure 2.4.3: Network Address Structure

Services
Services is a field of features which will be enabled with this connection and has the same functionality as in the version structure.

IPv4/v6
IP field has 16 bytes. Till version 0.7.0 the client could only read IPv4 network addresses so it was reading the last 4 bytes only since the first 12 were 00 00 00 00 00 00 00 00 00 00 FF FF.

Port
The next field is port which is the port number in network byte order.

2.4.2 Version Acknowledge Message

The version acknowledge (verack) message is sent in reply to version message, informing the connecting node that it can begin to send other messages. This message consists of only a message header with the command string ”verack” and empty payload data. Verack message belongs to a category of messages in which payload is zero.

2.4.3 GetAddr Message

The GetAddr message requests from the receiving node information about known active peers. The response to this message is to transmit one or more addr messages with one or more peers from a database of known active peers. A node is considered active if it has been sending a message within the last three hours. The structure is the following with payload being zero since no data is transmitted.

2.4.4 Addr Message

Addr message relays connection information for peers inside the network. Each peer who wants to accept incoming connections creates an addr message providing its connection information and then sends that message to its peers unsolicited. Non-advertised nodes should be forgotten after typically 3 hours. Addr message consists of the message header plus the payload which consists of 2 items, a counter of addresses sent and a list of addresses. Payload structure is shown below:

Count
AddrList

Figure 2.4.4: Addr Message Payload

Count
Count is the number of address entries which is maximum 1000 entries.

AddrList
AddrList field gets the addresses of other nodes on the network by calling a structure which is the same as the structure in the network address.

2.4.5 GetBlocks Message

GetBlocks message is used to request a list of blocks starting after the last known hash in the slice of block locator hashes. The list is returned via an inventory message and is limited by a specific hash to stop at or the maximum number of blocks per message, which are currently 500. The locator hashes are processed by a node in the order as they appear in the message. If a block hash is found in the node's main chain, the list of its children is returned back via the inventory message and the remaining locators are ignored, no matter if the requested limit was reached, or not. To receive the next blocks hashes, a peer needs to issue GetBlocks again with a new block locator object [14].

version
hash count
block locator hashes
hash_stop

Figure 2.4.5: GetBlocks Message Payload

Version
Version field identifies protocol version being used by the node.

Hash Count
Hash count is the number of block locator hash entries.

Block Locator Hashes
Block Locator Hashes is an object which locates the last block we have.

Hash_stop
This field has the hash of the last block which a client should have. Hash stop can bring up to 500 hashes. It is allowed to send in fewer known hashes down to a minimum of just one hash.

2.4.6 GetHeaders Message

The GetHeaders message requests a Headers message that provides block headers starting from a particular point in the blockchain. It allows a peer which has been disconnected or started for the first time to get the headers it has not yet sen. The GetHeaders message is nearly identical to the GetBlocks message, with one minor difference, the inventory message reply to the GetBlocks message will include no more than 500 block header hashes; the headers reply to the GetHeaders message will include a maximum of 2000 block headers.

2.4.7 Reject Message

The reject message informs the receiving node that one of its previous messages has been rejected.

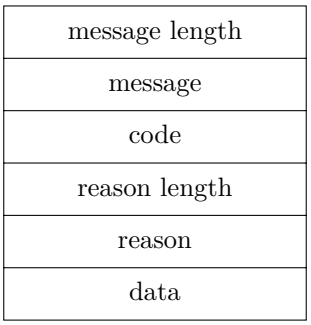

Figure 2.4.6: Reject Message Payload

Message Length
Message Length field indicates the number of bytes in the following message field.

Message
Message field indicates the type of rejected message as ASCII text without zero padding.

Code
Code field indicates the rejected message code.

Reason Length
Reason Length indicates the number of bytes in the following reason field. May be 0x00 if a text reason isn't provided.

Reason
Reason field indicates the reason for the rejection in ASCII text. This should not be displayed to the user; it is only for debugging purposes.

Data
Data field indicates optional additional data provided with the rejection. For example, most rejections of tx messages or block messages include the hash of the rejected transaction or block header.

2.5 Syncing inside the network

Transactions and block messages are important for synchronizing and updating the blockchain. In order to avoid sending transaction and block messages to nodes that have already received them from other nodes, they are not forwarded directly [11]. The availability of transactions and blocks is announced to the neighbors by sending them an inventory (inv) message once the transaction or block has been completely verified. The first thing a full node will do once it connects to peers and before starting validating unconfirmed transactions and recently-mined blocks, it must download and validate all blocks from block 1. If it is a brand-new node and has no blockchain at all, then it only knows one block, genesis block, which is hardcoded in the client software. Starting with block #0, the genesis block, the new node will have to download blocks up to the current tip of the best block chain to synchronize with the network and re-establish the full blockchain. The inventory message contains a set of transaction hashes and block hashes that have been received by the sender and are now available to be requested. A node, receiving an inventory message for a transaction or block that it does not yet have locally, will issue a GetData message to the sender of the inventory message containing the hashes of the information it needs. The actual transfer of the block or transaction is done via individual block or tx messages.

2.5.1 Block Propagation

Figure 2.5.1 visualizes a single block propagation in the network. Node A receives a block, verifies it and announces it to its neighbors. Node B receives the inventory message and since it does not recognise this block, it will issue a GetData message. Upon receiving the GetData message, Node A will deliver the block to Node B.

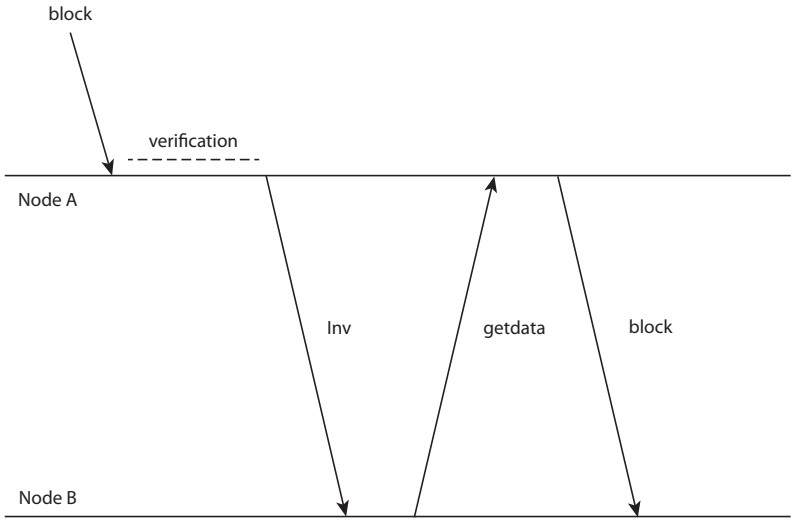

Figure 2.5.1: Block Propagation

2.5.2 Transaction Propagation

Transaction propagation has the same structure although it presents the reception of a transaction instead of a block, as Node A will deliver the transaction to Node B. Each block or transaction which is introduced to the network at one of the nodes, is propagated throughout the network using the below broadcast mechanism.

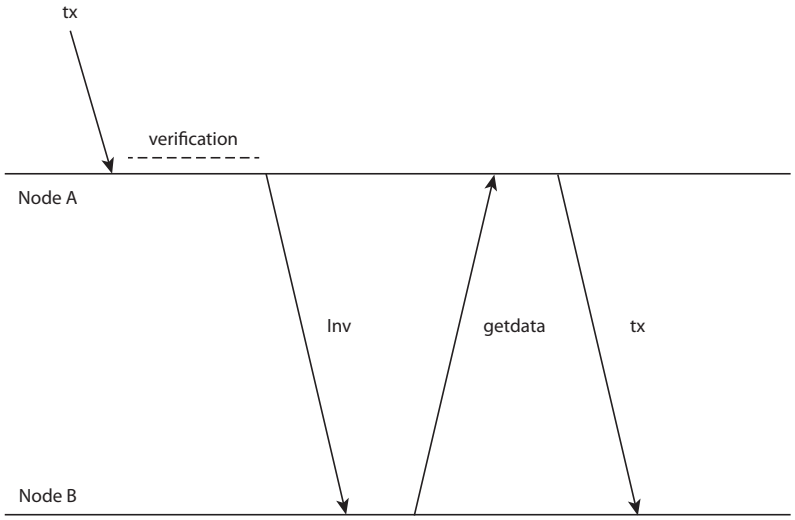

Figure 2.5.2: Transaction Propagation

2.6 Messages Part 2

2.6.1 Inventory Message

The inventory message allows a node to advertise its knowledge of one or more objects, usually blocks or transactions. It can be sent unsolicited to announce new transactions or blocks, or it can be sent in reply to a GetBlocks message or MemPool message. The recipient can compare the inventories from an inv message against the inventories it has already seen, and then use a follow-up message to request unseen objects. If the inventory message we receive is for a piece of data our node does not have, it will respond to that peer with a GetData message request. Once we have the new piece of data, we will then send out inventory messages to the rest of our peers letting them know that we have the data and they are welcome to request it from us.

count
inventory

Figure 2.6.1: Inventory Message Payload

Count
The count field is an integer which has the number of inventory vector entries.

Inventory

The inventory field is of type inventory vector and is used for notifying other nodes about objects they have or data which is being requested. Inventory vectors can have up to 50.000 entries and consist of the following data format:

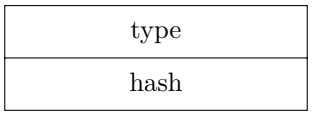

Figure 2.6.2: Inventory Vector Payload

Type

Identifies the type of object hashed, linked to this inventory. Client needs to check type field to recognize which object is receiving. Object type can be one of the 4 values which are error, transaction, block and filtered block according to the table below:

Value	Name
0	ERROR
1	MSG_TX
2	MSG_BLOCK
3	MSG_FILTERED_BLOCK

Table 2.6.1: Inventory Object Type Values

Hash

SHA256(SHA256()) hash of the object is sent.

2.6.2 GetData Message

GetData is used in response to inventory, to retrieve the content of a specific object, and is usually sent after receiving an inventory packet, after filtering known elements.

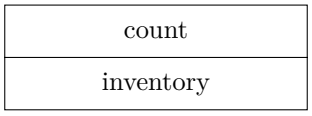

Figure 2.6.3: GetData Message Payload

GetData message follows the same structure as inventory message since it is the actual reply to it and a request for transactions or blocks. The response to a GetData message can be a tx message, block message, merkleblock message, or notfound message. This message cannot be used to request arbitrary data, such as historic transactions no longer in the memory pool or relay set. Full nodes may not even be able to provide older blocks if they've pruned old transactions from their block database. For this reason, the getdata message should usually only be used to request data from a node which previously advertised it had that data by sending an inv message. The format and maximum size limitations of the GetData message are identical to the inv message; only the message header differs.

Chapter 3

Bitcoin Architecture

Bitcoin is a decentralized digital currency based on a P2P network. Within the network each node is responsible for processing transactions and maintaining a public ledger of all transactions. Transactions are processed in a procedure called mining. The ledger, also known as the blockchain, is a record of all transactions that have ever occurred in the Bitcoin system and is used to track ownership of Bitcoins.

3.1 Transactions

Transactions are the core of the bitcoin system and let users spend satoshis. Each transaction consists of several parts which enable both simple direct payments and complex contracts. Bitcoin is designed to ensure that transactions are created, propagated on the network, validated, and finally added to the blockchain. A transaction informs the network that the owner has authorized the transfer of a certain amount of bitcoins to someone else. In principle, there are two types of transactions, coinbase transactions and regular/standard transactions. Coinbase transactions are special transactions in which new Bitcoins are introduced into the system for circulation. They are included in every block as the first transaction and are used as a reward for solving a proof-of-work problem. Standard transactions, on the other hand, are used to transfer existing Bitcoins amongst different accounts which may belong to the same user or not. A transaction is a data structure that encodes a transfer of value from a source of funds, called an input, to a destination, called an output. Transaction inputs and outputs are not related to accounts oridentities. Transactions are messages which contain a number of fields and their structure is presented below:

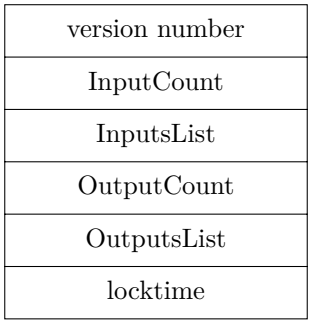

Figure 3.1.1: Transaction Structure

Version Number
The version field, stores the transaction version number and the rules followed by the transaction. The current transaction version number is 1.

InputCount
This field stores the number of elements in the inputs vector.

InputsList
The InputsList field stores a vector of one or more transaction inputs. This field will be explained in section of standard transactions and coinbase transactions.

OutputCount
This field stores the number of elements in the output vector.

OutputsList
The OutputsList field stores a vector of one or more transaction outputs. This field will be explained in section of standard transactions and coinbase transactions.

Locktime
This field stores the locktime which indicates the earliest time a transaction can be added to the blockchain. Once the lock time has been exceeded, the transaction is locked and becomes immune to transaction replacement. The locktime is encoded as either a timestamp in UNIX format or as a block number:

Value	Description
0	Always Locked
$< 5 * 10^8$	Block number at which transaction is locked
$> 5 * 10^8$	UNIX timestamp at which transaction is locked

Table 3.1.1: LockTime Values

3.1.1 Standard Transactions

Every bitcoin transaction creates outputs, which are recorded in the blockchain. All of these outputs, create spendable chunks of bitcoins called unspent transaction outputs, which are then

recognized by the whole network and available for the owner to spend in a future transaction. Sending someone bitcoin is creating an unspent transaction output registered to the sender's address and available for him/her to spend. Every transaction in Bitcoin has one or more inputs and outputs. Each input/output contains a function associated which is called a script.

InputsList

The InputsList field, stores a vector of one or more transaction inputs. Each transaction[6] input is a data structure which is composed of a reference to a previous output (hash, index), the length of the digital signature script in bytes (scriptSigLen), the digital signature script (scriptSig) itself and a transaction sequence number (nSequence).

1. **(hash, index)**
   Because a single transaction may include multiple outputs, the previous output structure is uniquely identified by the tuple (hash, index). Hash, is referred to as the transaction ID (TxId) and is computed as a double-SHA256 hash of the raw transaction:

$$TransactionID = SHA256\,(SHA256)$$

   Transactions are identified uniquely by their hash outputs within specific transaction are identified by their output index.

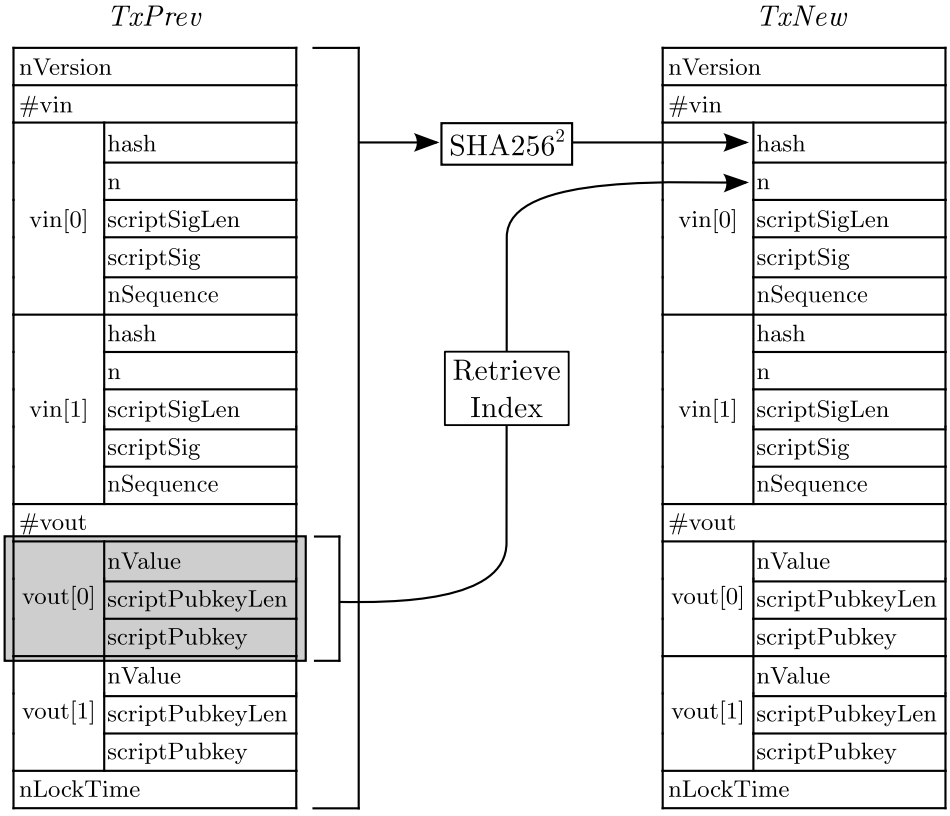

Figure 3.1.2: Transaction Output Reference Computation

2. **ScriptSigLen**
   This field stores the length of the signature script field scriptSig in bytes. Maximum number of bytes is 10000.

3. **ScriptSig**
   The signature script field contains a response script corresponding to the challenge script (ScriptPubKey) of the referenced transaction output (prevout). Whilst the challenge script specifies conditions under which the transaction output can be claimed, the response script is used as a proof that the transaction is allowed to be claimed.

4. **Sequence Number**
   This field stores the transaction sequence number. Sequence numbers were meant to allow multiple signers to agree to update a transaction; when they finished updating the transaction, they could agree to set every input's sequence number to the four-byte unsigned maximum (0xffffffff), allowing the transaction to be added to a block even if its time lock had not expired. Transaction replacement feature is currently unsupported in Bitcoin but it has a huge impact on complex contracts. If the transaction is locked permanently, then the sequence number is set to the highest 4-byte integer 0xFFFFFFFF. That's the default number for Bitcoin Core and almost all other programs.

OutputsList

The OutputsList field stores a vector of one or more transaction outputs. Each transaction output [6] is composed of an amount of BTC being spent (Value), the length of the public key script (scriptPubkeyLen) and the public key script (scriptPubkey) itself.

1. **Value**
   The Value field stores the amount of BTC to be spent by the output. The amount is encoded in Satoshis, that is 10^{-8} BTC, allowing tiny fractions of a Bitcoin to be spent. The sum of all outputs may not exceed the sum of satoshis previously spent to the previous output provided in the input section. However, note that in the reference implementation transactions with outputs less than a certain value, also called "dust", and are considered non-standard. This value is currently by default 546 Satoshi and can be defined by each node manually. Dust transactions are neither relayed nor mined.

2. **ScriptPubKeyLen**
   This field stores the length of the public key script (scriptPubkey) in bytes. Maximum number of bytes is 10000.

3. **ScriptPubKey**
   The public key script field contains a challenge script for transaction verification. More precisely, whilst the challenge script specifies conditions under which the transaction output can be claimed, the response script defines the conditions which must be satisfied to spend it.

3.1.2 Coinbase Transactions

Coinbase transactions are included in the block and are generated as a reward for the miner who will find this block.

InputsList
The InputsList field stores a vector of precisely one transaction input. The input is a data structure which is composed of a reference to a previous output (hash, index), the length of the coinbase field in bytes (coinbaseLen), the coinbase field (coinbase) itself and a transaction sequence number.

1. **(hash, index)**
   In a coinbase transaction new coins are introduced into the system and therefore no previous transaction output is referenced. The (hash, index) tuple stores the following constant values:

$$hash = 0$$

$$index = 2^{32} - 1$$

   The hash does not reference any previous transaction output and is therefore set to zero whereas the output index is set to its maximal value.

2. **Coinbase Length**
   This field stores the length of the coinbase field coinbase in bytes. It is in the range of 2-100 bytes.

3. **Coinbase**
   The coinbase field, also referred to as the coinbase script, stores the block height, the block number within the blockchain, and arbitrary data.

Field Name	Size	Description
BlockHeightLen	1	Length of BlockHeight field
BlockHeight	BlockHeightLen	Block Height Encoding
ArbitraryData	(BlockHeightLen+1)	Arbitrary Data Field

Table 3.1.2: Coinbase Values

OutputsList
The transaction output vector is constrained by the maximal sum of Bitcoins that are allowed to be transacted.

1. **Value**
   In a coinbase transaction the miner is allowed to transfer the current mining subsidy, as well as transaction fees for all included transactions, as a reward for solving the proof-of-work problem. The subsidy for finding a valid block is currently 25 BTC and is halved every 210000 blocks. The transaction fee, on the other hand, is computed for each transaction as

the difference between the sum of input values (referenced output values) and the sum of output values.

3.1.3 Transaction Fees

Most transactions include transaction fees, which compensate the bitcoin miners for securing the network. Transaction fees serve as an incentive to include a transaction into the next block and also as a disincentive against ”spam” transactions or any kind of abuse of the system and network, by imposing a small cost on every transaction. Transaction fees are collected by the miner who mines the block that records the transaction on the blockchain.

Transaction fees are calculated based on the size of the transaction in kilobytes, not the value of the transaction in bitcoin. Most miners and mining pools prioritize transactions by fees and then priority. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next-most mined block, while a transaction with insufficient or no fees may be delayed, on a best-effort basis and processed after a few blocks or not at all. Transaction fees are not mandatory and transactions without fees may be processed eventually; however, including transaction fees encourages priority processing.

3.2 Blocks

A block is a record of some or all of the most recent Bitcoin transactions that have not yet been added in any previous blocks. Each block contains all the transactions that have been verified and added to the blockchain. Each block within the blockchain is identified by a hash, generated using the SHA256 cryptographic hash algorithm on the header of the block and references a previous block, known as the parent block through the ”previous block hash” field in the block header. This sequence of hashes links back to the genesis block. General Block Structure is shown in the figure below:

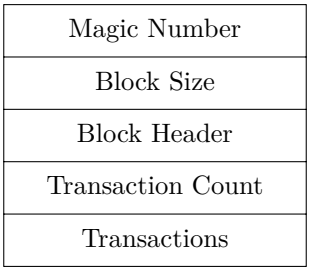

Figure 3.2.1: Block Structure

Magic Number (Network ID)
This field is indicating message origin network, and is used to seek the next message when stream state is unknown. Four defined bytes which start every message, for the main Bitcoin network is 0xD9B4BEF9.

Block Size
This field is indicating the number of bytes following up to end of block.

Block Header
The header stores the current block header version, a reference to the previous block, the root of the Merkle tree, a timestamp, a target value and a nonce. These items will be explained in the block header section.

Transaction Count
A counter for how many transactions have been included in the block.

Transactions
A vector which has the transactions recorded in this block.

3.2.1 Block Header

The block header consists of three sets of block metadata. First, there is a reference to a previous block hash, which connects this block to the previous block in the blockchain. The second set of metadata, namely the difficulty, timestamp and nonce, relate to the mining competition. The third piece of metadata is the Merkle Tree root, a data structure used to efficiently summarize all the transactions in the block.

Version
Previous Block Hash
HashMerkleRoot
Time
Bits
Nonce

Figure 3.2.2: Block Header Structure

Version
The version field stores the version number of the block format. The block version number indicates which set of block validation rules to follow. Ever since BIP34 [15] is in place, the block format version is 2, until sufficient number of miners move to Bitcoin Core 0.10.0 and higher. Furthermore, by now the 95% rule is in place, which states that all version 1 blocks should be rejected once 950 of the last 1000 blocks are version 2 or greater.

Previous Block Hash
This field stores a reference to the previous block, computed as a hash over the block header. It ensures no previous block can be changed without also changing the current block header. A double-SHA256 hash is calculated over the concatenation of all elements in the previous block

header:

$$SHA256(SHA256(Version||PreviousBlockHash||MerkleRoot||Timestamp||Bits||Nonce))$$

HashMerkleRoot
This field stores the root of the Merkle hash tree. The merkle root is derived from the hashes of all transactions included in this block, is used to provide integrity of all transactions included in the block and is computed according to the scheme described in Merkle Trees section. The parameters used for computing the tree are double-SHA256 as the hashing algorithm and raw transactions as data blocks.

Time
The time field stores the timestamp in UNIX format denoting the approximate block creation time, according to the miner. As the timestamp is a parameter included in the block mining process, it is recorded at the beginning of it.

Difficulty Target / Bits
The Bits field stores a compact representation of the target value T. The target value is a 256 hex-digit long number, whereas its corresponding compact representation is only 8 hex-digits long and thus encoded with only 4 bytes [6]. The upper bound for the target is defined as 0x1D00FFFF whereas there is no lower bound. The very first block, the genesis block, has been mined using the maximum target. In order to ensure that blocks are mined at a constant rate of one block per 10 minutes throughout the growing network, the target T is recalculated every 2016 blocks based on the average time it took to mine, due to an off-by-one error, the last 2015 blocks [14].

Nonce
The nonce field contains arbitrary data and is used as a source of randomness for solving the proof-of-work problem, as it can vary the output of a cryptographic function. However, since it is fairly small in size with 4 bytes, it does not necessarily provide sufficient variation for finding a solution. If all 32-bit values are evaluated then the coinbase field changes which leads to changes in the merkle root.

Chapter 4

Bitcoin Ownership

Bitcoin clients validate transactions by executing a script, written in a Forth-like scripting language. Both the locking script placed on a UTXO and the unlocking script that usually contains a signature are written in this scripting language. When a transaction is validated, the unlocking script in each input is executed alongside the corresponding locking script to see if it satisfies the spending condition.

4.1 Script Language and Script Construction

Scripts can contain signatures over simplified forms of the transaction itself. Input scripts specify who the money is from and generally claim previous outputs (unless the input is a coinbase transaction), thus using coins received from previous transactions. Output scripts specify who the money is going to and the conditions that must be met to claim it. A new transaction is valid if the transaction scripts of its input field and the transaction script of its predeceasing transaction validates to true. The bitcoin script language, also named Script, is a Forth-like stack-based execution language designed stateless and not Turing complete. Script is called stack based because it uses a stack as a basic data structure. A stack allows two operations: push and pop. Push adds elements to the stack while pop removes elements from the stack. A script is essentially a set of instructions that are processed left to right. Script is used to encode two components - a challenge script and a response script [6]:

- A challenge script (scriptPubKey) accompanies a transaction output and specifies the conditions which must be met to claim the output in the future.
- A response script (scriptSig) accompanies a transaction input; usually contains a digital signature and is used to prove that the referenced output can be rightfully claimed.

Bitcoin clients verify transactions by executing both scripts simultaneously. For each input in the transaction, the validation process will first retrieve the unspent transaction outputs referenced by the input. Then scriptSig is evaluated, by copying the resulting stack and finally evaluating

scriptPubKey of the referenced transaction output for all transaction inputs in it. If during the evaluation no failure is triggered and the final top stack element yields true, then the ownership has been successfully verified. The scriptSig is executed using the stack execution engine. If scriptSig gets executed succesfully, the main stack is copied and the scriptPubKey is executed. If the result of executing the scriptPubKey with the stack data copied from the scriptSig has evaluated to true the scriptSig has successfully resolved the conditions imposed by the scriptPubKey. Consequently, the input is a valid authorization to spend the unspent transaction output. If the result which remains false after execution of the combined script, the input is invalid as it has failed to satisfy the spending conditions placed on the unspent transaction output.

4.2 Standard Transaction Scripts

Script allows a party to construct complex conditions under which coins can be redeemed, although much of its functionality is currently disabled in the reference implementation and only a restricted set of standard scripts is accepted. In particular, calculating hashes, verifying signatures and simple arithmetic operations are the only supported operations. These limitations are encoded in a function called isStandard() which defines five types of ”standard” transactions in particular, calculating hashes, verifying signatures and simple arithmetic operations are the only supported operations. These are Pay-to-Pubkey (P2PK), Pay-to-PubkeyHash (P2PKH), Pay-to-ScriptHash (P2SH), Multisig and Nulldata.

4.2.1 Pay to PubKey

One of the two more common ScriptPubKey types is Pay to Public Key. In a Pay-to-Pubkey transaction the sender transfers Bitcoins directly to the owner of a public key. He specifies in the challenge script the public key and the only requirement that the recipient has to prove:

- Knowledge of the private key corresponding to the public key.

The recipient creates a response script with a signature that will be used to verify the public key [6].

```
scriptPubkey: <pubkey> OP_CHECKSIG
scriptSig:    <signature>
```

Figure 4.2.1: Pay-to-Pubkey Structure

Stack	Remaining Script	Description
`Empty`	`<signature>`	The signature is pushed on the stack.
`<signature>`	`Empty`	Final state after evaluating *scriptSig*.

Table 4.2.1: Pay-to-Pubkey *scriptSig* Execution

Stack	Remaining Script	Description
`<signature>`	`<pubkey> OP_CHECKSIG`	State after copying the stack of the signature script evaluation. The public key is pushed on the stack.
`<pubkey>` `<signature>`	`OP_CHECKSIG`	The signature is verified for the top two stack elements and the result is pushed on the stack.
`True`	`Empty`	Final state after evaluating *scriptPubkey*.

Table 4.2.2: Pay-to-Pubkey *scriptPubkey* Execution

4.2.2 Pay to PubKey Hash

The structure of the challenge and response scripts of a Pay-to-PubkeyHash transaction can be seen below [6]:

```
scriptPubkey: OP_DUP OP_HASH160 <pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig:    <signatures> <pubkey>
```

Figure 4.2.2: Pay-to-PublicKey Hash Structure

In a Pay-to-PubkeyHash transaction the sender transfers Bitcoins to the owner of a P2PKH address. He specifies in the challenge script the public key hash (pubkeyHash) of the Bitcoin address and two requirements that the redeemer has to prove:

- Knowledge of the public key corresponding to pubkeyHash.
- Knowledge of the private key corresponding to the public key.

To do so, the redeemer creates a response script with a signature, which is created from the private key and a public key. The scripts are then executed as shown in the figures below. First, it is verified if the public key (pubkey) provided by the claimant corresponds to the public key hash (pubkeyHash) provided by the sender and then whether the signature is valid.

Stack	Remaining Script	Description
`Empty`	`<signature> <pubkey>`	The signature and the public key are pushed on the stack.
`<pubkey>` `<signature>`	`Empty`	Final state after evaluating *scriptSig*.

Table 4.2.3: Pay-to-PubkeyHash *scriptSig* Execution

Stack	Remaining Script	Description
<pubkey> <signature>	OP_DUP OP_HASH160 <pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	State after copying the stack of the signature script evaluation. The top stack element is duplicated.
<pubkey> <pubkey> <signature>	OP_HASH160 <pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	The top stack element is first hashed with SHA256 and then with RIPEMD160.
<pubkeyHashNew> <pubkey> <signature>	<pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	The public key hash is pushed on the stack.
<pubkeyHash> <pubkeyHashNew> <pubkey> <signature>	OP_EQUALVERIFY OP_CHECKSIG	Equality of the top two stack elements is checked. If it evaluates to true then execution is continued. Otherwise it fails.
<pubkey> <signature>	OP_CHECKSIG	The signature is verified for the top two stack elements.
True	Empty	Final state after evaluating *scriptPubkey*.

Table 4.2.4: Pay-to-PubkeyHash *scriptPubkey* Execution

4.2.3 Pay to ScriptHash

The structure of the challenge and response scripts of a Pay-to-ScriptHash transaction is shown below [6]:

```
scriptPubkey: OP_HASH160 <scriptHash> OP_EQUAL
scriptSig:    <signatures> {serializedScript}
```

Figure 4.2.3: Pay-to-ScriptHash Structure

In a Pay-to-ScriptHash transaction the sender transfers Bitcoins to the owner of a P2SH Bitcoin address. He specifies in the challenge script the serialized script hash scriptHash of the Bitcoin and one requirement that the redeemer has to prove:

- Knowledge of the redemption script serializedScript corresponding to scriptHash.

To do so, the redeemer creates a response script with one or more signatures and the serialized redemption script serializedScript. Note that unlike in any other standard transaction type the responsibility of supplying the conditions for redeeming the transaction is shifted from the sender to the redeemer. The redeemer may specify any conditions in the redemption script serializedScript conforming to standard transaction types.

Stack	Remaining Script	Description
Empty	<signature> {<pubkey> OP_CHECKSIG}	The signature and the redemption script are pushed on the stack.
{<pubkey> OP_CHECKSIG} <signature>	Empty	Final state after evaluating *scriptSig*.

Table 4.2.5: Pay-to-ScriptHash *scriptSig* Execution

Stack	Remaining Script	Description
{<pubkey> OP_CHECKSIG} <signature>	OP_HASH160 <scriptHash> OP_EQUAL	State after copying the stack of the signature script evaluation. The top stack element is first hashed with SHA256 and then with RIPEMD160.
<scriptHashNew> <signature>	<scriptHash> OP_EQUAL	The redemption script hash is pushed on the stack.
<scriptHash> <scriptHashNew> <signature>	OP_EQUAL	Equality of the top two stack elements is checked. The result of the evaluation is pushed on the stack.
True <signature>	Empty	Final state after evaluating *scriptPubkey*.

Table 4.2.6: Pay-to-ScriptHash *scriptPubkey* Execution

4.2.4 MultiSig

The structure of the challenge and response scripts of a Multi-sig transaction is depicted below [6]:

```
scriptPubkey: m <pubkey 1> ... <pubkey n> n OP_CHECKMULTISIG
scriptSig:    OP_0 <signature 1> ... <signature m>
```

Figure 4.2.4: Multisig Structure

In a Multisig transaction the sender transfers Bitcoins to the owner of m-of-n public keys. He specifies in the challenge script n public keys (pubkey 1...n) and a requirement that the redeemer has to prove:

- Knowledge of at least m private keys corresponding to the public keys.

To do so, the redeemer creates a response script with at least m signatures in the same order of appearance as the public keys. Note that due to an off-by-one error OP CHECKMULTISIG pops one too many elements off the stack and it is therefore required to prepend the response script with a zero data push OP 0. Note that only 3-of-3 transactions are allowed to the current version 0.9. Any multi-sig transaction with more than 3 public keys is considered non-standard.

Stack	Remaining Script	Description
Empty	OP_0 <signature 1> ... <signature m>	The signatures are pushed on the stack.
<signature m> ... <signature 1> OP_0	Empty	Final state after evaluating *scriptSig*.

Table 4.2.7: Multisig *scriptSig* Execution

Stack	Remaining Script	Description
<signature m> ... <signature 1> OP_0	m <pubkey 1> ... <pubkey n> n OP_CHECKMULTISIG	State after copying the stack of the signature script evaluation. The public keys are pushed on the stack.
n <pubkey n> ... <pubkey 1> m <signature m> ... <signature 1> OP_0	OP_CHECKMULTISIG	The signatures are verified in order of appearance and the result is pushed on the stack.
True	Empty	Final state after evaluating *scriptPubkey*.

Table 4.2.8: Multisig *scriptPubkey* Execution

4.2.5 Nulldata

The structure of Nulldata transaction is presented below [6]:

```
scriptPubkey: OP_RETURN [ARBITRARY DATA]
scriptSig:
```

Figure 4.2.5: Nulldata Structure

This transaction cannot be spent and that is the reason why it does not specify a reciepient and does not have a signature. will always be invalid. Its purpose is let you add a small amount of arbitrary data to the block chain in exchange for paying a transaction fee. After the OP_RETURN opcode you can insert arbitrary data. OP_RETURN outputs will be added to the blockchain while they will not be added to the UTXO database of full nodes. Bitcoin core allows currently up to 40 bytes of arbitrary data. The size will increase to 80 bytes in Bitcoin core 0.11.0.

4.3 Non Standard Transactions

When using a standard pubkey script in an output, nodes that use the reference client will not accept, broadcast and mine your transaction. Creating a response script hashing it and using the hash in a P2SH output will be accepted as the network can only see the hash. Although, spending that output will not be possible unless you find a miner who disables the default settings of the client because these nodes are going to check the response script to see whether it contains a standard transaction output script or not.

4.3.1 Anyone Can Spend

The structure of Anyone can spend transaction is presented below:

```
scriptPubkey: <is_empty>
scriptSig:    OP_TRUE
```

Figure 4.3.1: Anyone Can Spend Structure

As the title of the transaction suggests this transaction can be spent by anyone. The output of the scriptpubkey is empty, so the scriptSig can simply push TRUE on the stack making it valid and claiming it. Anyone-Can-Spend are currently non-standard, and not broadcasted in the network.

4.3.2 Not Strict DER Encoding

Transactions that do not use strict DER encoding have been nonstandard since Bitcoin Core 0.8.0

4.4 Signatures

Signatures are used to prove ownership of the private key corresponding to a public key or a Bitcoin address. For a transaction, a signature is included in each of the transaction input scripts (scriptSig) to prove that the referenced transaction outputs are indeed owned by the redeemer.

Whenever the redeemer computes a signature for a transaction input, he specifies one of four signature types. The various signature types are listed in the table below. Depending on the redeemer's choice different parts of the transaction will be covered by the signature.

Name	Value
SIGHASH_ALL	0x00000001
SIGHASH_NONE	0x00000002
SIGHASH_SINGLE	0x00000003
SIGHASH_ANYONECANPAY	0x00000080

Table 4.4.1: Signature Hash Types

1. SIGHASH_ALL, the default, signs all the inputs and outputs, protecting everything except

the scriptSigs against modification.

2. SIGHASH_NONE signs all of the inputs but none of the outputs, allowing anyone to change where the satoshis are going unless other signatures using other signature hash flags protect the outputs.
3. SIGHASH_SINGLE signs only this input and only one corresponding output (the output with the same output index number as the input), ensuring nobody can change your part of the transaction but allowing other signers to change their part of the transaction. The corresponding output must exist or the value "1" will be signed, breaking the security scheme.

In addition to the above base types there is one more hash type
SIGHASH_ANYONECANPAY which can be used in combination with all of them:

1. SIGHASH_ALL|SIGHASH_ANYONECANPAY signs all outputs but only this specific input. Everyone is allowed to add or remove other inputs thus anyone can contribute but they cannot modify the number of satoshis being sent or their destination.
2. SIGHASH_NONE|SIGHASH_ANYONECANPAY signs only this specific input. Everyone is allowed to add or remove other inputs or outputs thus anyone can spend this the way they want.
3. SIGHASH_SINGLE|SIGHASH_ANYONECANPAY signs only this specific input and only one corresponding output. Everyone is allowed to add or remove other inputs.

4.4.1 SIGHASH_ALL

The default signature hash type SIGHASH. It signs the entire transaction with the exception of the signature scripts.

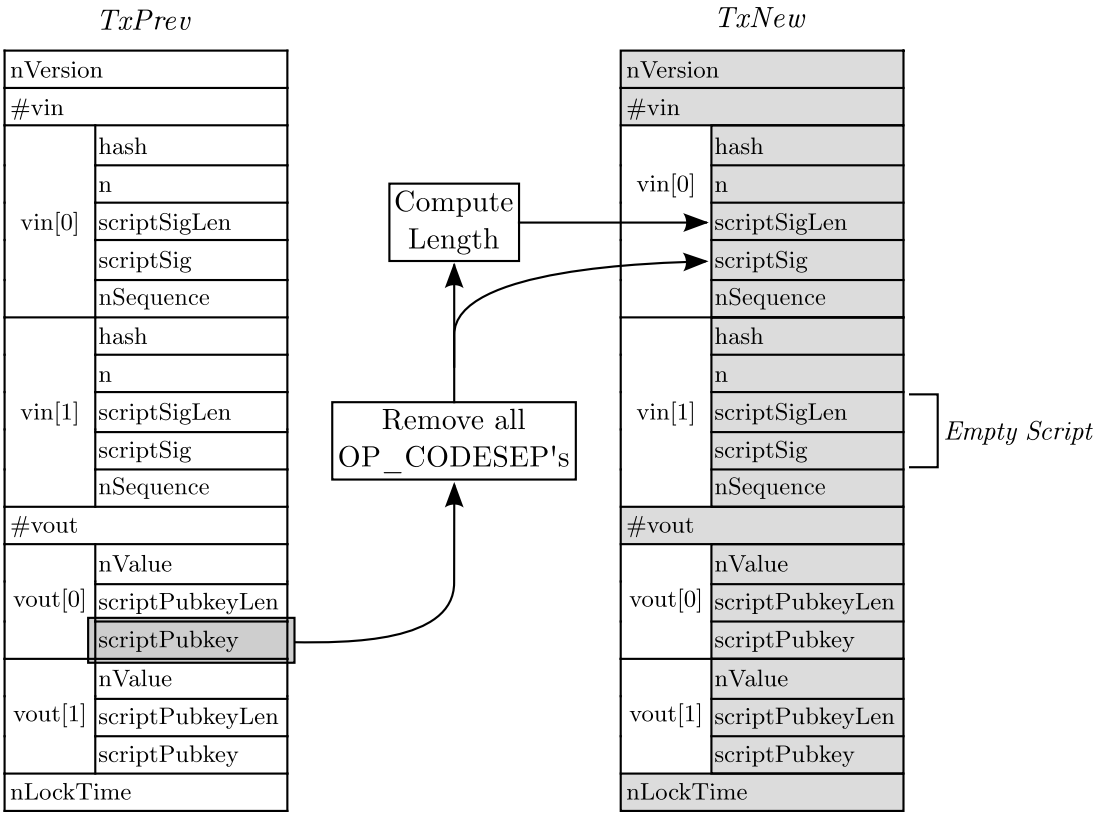

Figure 4.4.1: Signature Computation - SIGHASH ALL

Before the signature is computed, several temporary changes are made to the transaction [6]:

- The signature script of the currently signed input is replaced with the public key script, excluding all occurrences of OP_CODESEPARATOR in it, of the referenced transaction output.
- The signature scripts of all other inputs are replaced with empty scripts.

4.4.2 SIGHASH_SINGLE

In the second signature hash type SIGHASH_SINGLE all transaction inputs and the transaction output corresponding to the currently signed input is signed.

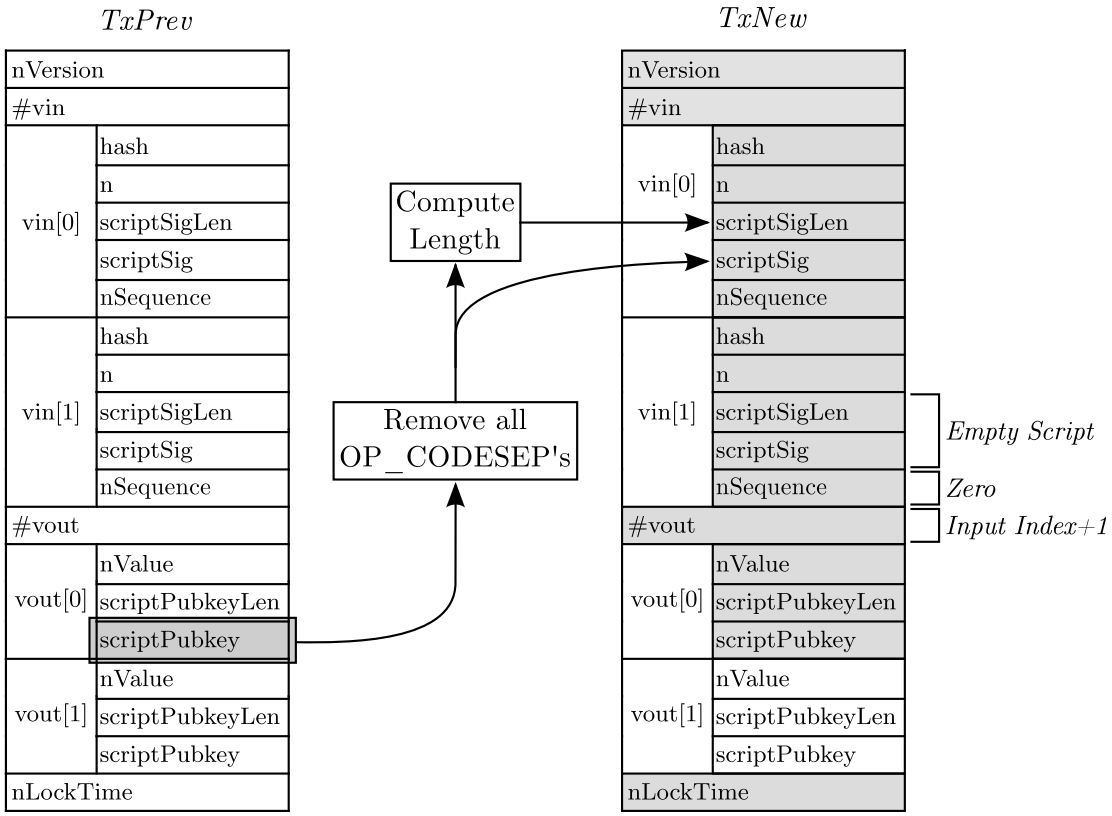

Figure 4.4.2: Signature Computation - SIGHASH SINGLE

Before the signature is computed, several temporary changes are made to the transaction [6]:

- The signature script of the currently signed input is replaced with the public key script, excluding all occurences of OP_CODESEPARATOR in it, of the referenced transaction output.
- For all the remaining transaction inputs:
  1. The signature scripts are replaced with empty scripts.
  2. The sequence number is set to zero.
- The number of transaction outputs is set to the currently signed transaction input index plus one.
- All transaction outputs up to the currently signed one are emptied.

4.4.3 SIGHASH_NONE

In the third signature hash type SIGHASH NONE all transaction inputs and none of the transaction outputs are signed.

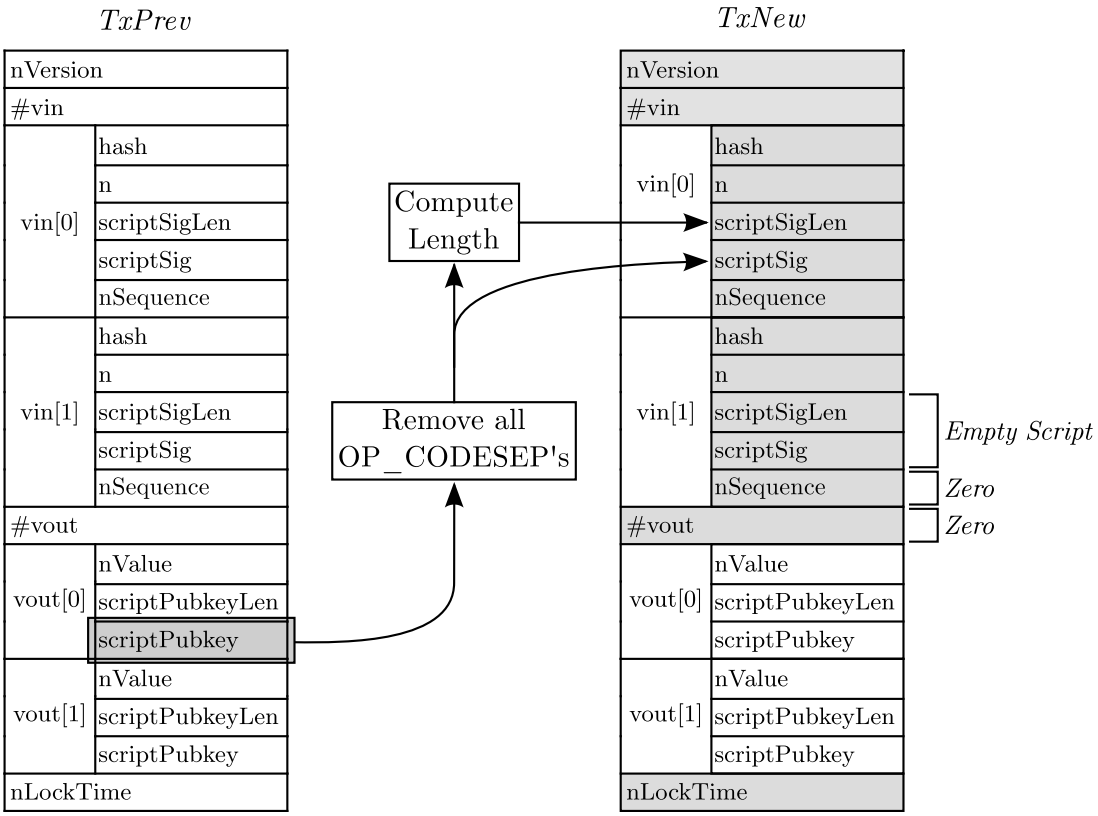

Figure 4.4.3: Signature Computation - SIGHASH NONE

Before the signature is computed, several temporary changes are made to the transaction:

- The signature script of the currently signed input is replaced with the public key script, excluding all occurrences of OP CODESEPARATOR in it, of the referenced transaction output.
- For all the remaining transaction inputs [6]:
  1. The signature scripts are replaced with empty scripts.
  2. The sequence number is set to zero.
- The number of transaction outputs is set to zero.
- All transaction outputs are removed.

4.4.4 SIGHASH_ANYONECANPAY

The SIGHASH ANYONECANPAY modifier is used in conjunction with a base type and affects the signaturecoverage of transaction inputs. It is used to only cover the currently signed input by the signature.

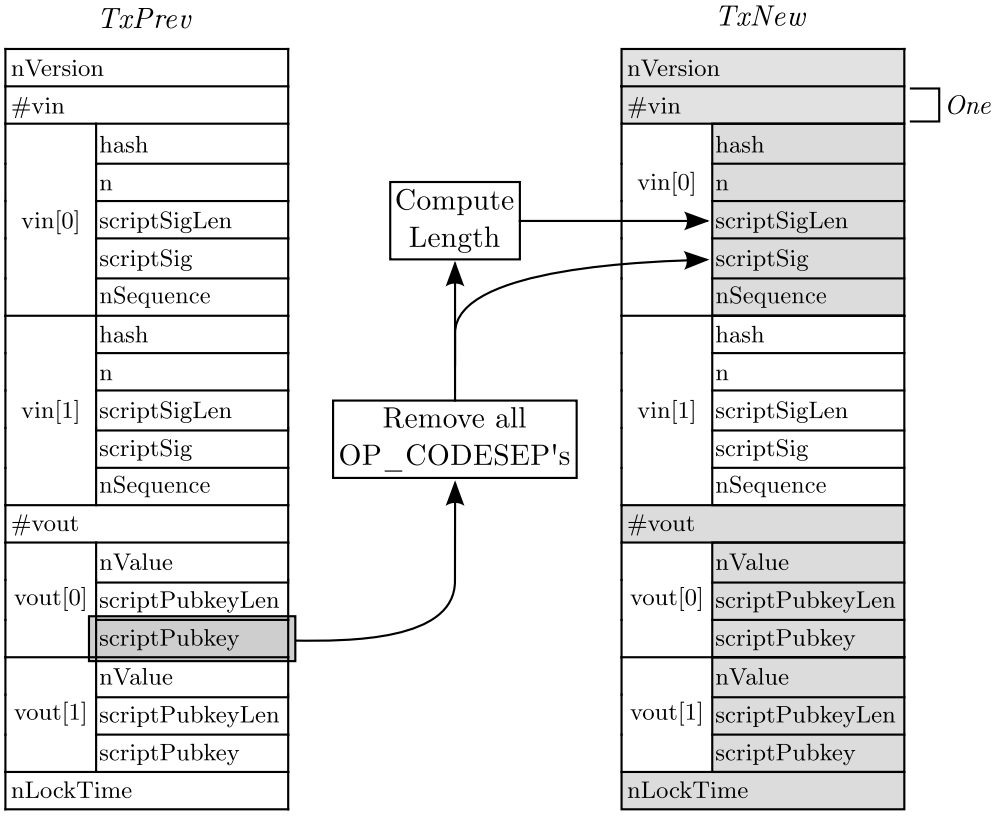

Figure 4.4.4: Signature Computation - SIGHASH ALL|SIGHASH ANYONECANPAY

Before the signature is computed, in addition to the changes performed by the base hash type, the following temporary changes are made [6]:

- The number of transaction inputs is set to one
- All transaction inputs, except for the currently signed one, are removed.

Part 3

Secure Multiparty Computations via Bitcoin

Chapter 1

Preliminaries

In this chapter we provide some basic cryptographc schemes along with their definitions and properties that are used for building protocols throughout the second part of this thesis. We define also ideal functionalities with the most important being the Claim or Refund Functionality that will be utilized for building our protocols.

1.1 Standard Primitives

1.1.1 Zero Knowledge Proofs

Zero Knowledge Proof is a protocol by which the prover can prove to the verifier that a statement is true. Non Interactive Zero knowledge proofs are a variant in which no interaction is necessary between prover and verifier. ZK-Snarks [16] are used to prove a statement without revealing the corresponding witness while the proofs are short and easy to verify. ZK-Snakrs proof system consists of three algorithms Gen (key generation), P (proving), and V (verifying):

- Generator G: It takes as input 1^k (where k is the security parameter) a circuit C and ouputs a proving key pk and a verification key vk.
- Prover P: Takes as input the proving key pk, a word $x \in L$ and a NP-witness w for x, and outputs the proof π.
- Verifier V: Takes as input verification key, x and π, and returns 0 or 1 depending on whether the verifier accepts the proof that $x \in L$.

The system must satisfy the three following properties:

- Completeness: If $x \in L$ and w is a NP-witness for x, then the proof π produced by the prover on input (x, w) will be accepted by the verifier, with probability 1.
- Soundness: for any polynomial-time adversary running on input (1^k, pk) and producing a pair (x, π), the probability that x is not in L and that (x, π) is accepted by V is negligible in k.

- Succinctness: An honestly-generated proof π has O(1) bits and Verify runs in polynomial time.
- Zero-knowledge. There exists a polynomial simulator S, who first generates key pairs (pk, vk), such that for any $x \in L$ chosen by a polynomial adversary, S generates a proof for x. The proof generated by S is computationally indistinguishable from honestly generated ones.
- Proof of knowledge. There exists a polynomial-time extractor E such that if a polynomial-time prover P convinces the verifier V to accept some $x \in L$ with non-negligible probability, then given oracle access to P, the extractor can produce a witness w such that $(x, w) \in R$ with non-negligible probability.

1.1.2 Commitment Schemes

Commitment schemes [17] emerged out of the need for parties to commit to a chosen value with the ability to reveal the committed value later to the other parties involved in a way that is fair to all the parties. Consequently, we do not want the recipient to know the commitment before it is revealed to him by sender and we do not want sender to be able to change the chosen value that he committed to. The protocol consists of two phases: the first one, called the commitment phase, sender commits to some secret message m by interacting with recipient and the second one, called the opening phase in which sender opens the commitment by interacting again with recipient, which results in recipient learning the message m.

- *Commitment Phase*: Sender commits to a secret message m by generating $(c, k) \leftarrow Commit(m)$ where c is the commitment and k is the opening key. The commitment c is sent to the recipient, while m and k remain secret.
- *Opening Phase*: Sender reveals the opening key k to the recipient, who gets the secret message $m \leftarrow Open(c, k)$. If the pair (c, k) is invalid then $Open(c, k)$ returns nothing.

The above scheme needs to have 2 security properties:

- Hiding: Receiving a commitment to a secret message m should give no information to the recipient about m
- Binding: The sender cannot cheat in the opening phase and send a different key K' that causes the commitment to open a different message m'

1.1.3 Secret Sharing Schemes

A secret sharing scheme allows a dealer to split some secret $s \in F$, where F is some publicly known field, into shares, such that reconstruction of s is possible only if enough shares are known.

Definition A m-out-of-n secret sharing scheme is a pair of polynomial time algorithms (Share, Rec). On input m, $n \in N$ and $s \in F$, Share(m,n,s) outputs shares $s_1, s_2, ..., s_n \in F$ with the following properties:

- For any $S \subset (s_1, ..., s_n)$ such that $|S| < m$, S reveals nothing abut s information theoritically

- For any $S \subset (s_1, ..., s_n)$ such that $|S| \geq m$, Rec(S) = s

Non-malleable secret sharing schemes. A non-malleable secret sharing scheme is a secret sharing scheme with an additional property, guaranteeing that if any party manipulates their share in any way, the reconstruction protocol outputs a special failure symbol. For our purposes, a m-out-of-n scheme will suffice, so we define that.

Definition A m-out-of-n non-malleable secret sharing scheme (NMSS scheme) is defined by a pair of polynomial-time algorithms (Share,Rec) with the following properties:

- Share(s,r) returns n shares, (s_1,...,s_n) (where s_i is the share of the i-th party) such that a group of shares reveals no information about s.
- Rec(Share(s,r)) = (s,0) for every s,r. The second output of Rec serves as a flag which is set to 0 if the secret has been successfully reconstructed.
- Any attempt by a player to modify their share (independently of the remaining share) is detected with overwhelming probability. Formally, we say that (Share,Rec) is ϵ-non-malleable if for every secret s, every (computationally unbounded) adversary A can win the following game with probability at most ϵ:
  - A corrupts one of the parties.
  - Random shares (s_1, s_n) from Share(s,r) are given to the n parties.

1.1.4 Public Verifiable Computation

A public verifiable computation scheme [18] pubVC consists of a set of three polynomial-time algorithms (KeyGen, Compute,Verify) defined as follows:

- $(ek_f, vk_f) \leftarrow KeyGen(f, 1^k)$: The randomized key generation algorithm takes the function f to be outsourced and security parameter k; it outputs a public evaluation key ek_f, and a public verification key vk_f.
- $(y, \psi_y) \leftarrow Compute(ek_f, u)$: The deterministic worker algorithm uses the public evaluation key ek_f and input u. It outputs $y \leftarrow f(u)$ and a proof ψ_y of y's correctness.
- $0, 1 \leftarrow Verify(vk_f, u, (y, \psi_y))$: Given the verification key vk_f, the deterministic verification algorithm outputs 1 if f(u) = y, and 0 otherwise

The scheme pubVC should satisfy the following conditions:

- **Correctness**: For any function f, it holds that
  $\Pr\left[(ek_f, vk_f) \leftarrow KeyGen(f, 1^k),\ (y, \psi_y) \leftarrow Compute(ek_f, u) : 1 \leftarrow Verify(vk_f, u, (y, \psi_y))\right] = 1$
- **Soundness**: For any function f and any PPT A the following is negligible in :
  $\Pr\left[(u', y', \psi'_y) \leftarrow Compute(ek_f, vk_f) : f(u') \neq y' \wedge Verify(vk_f, u', (y', \psi'_y))\right]$
- **Efficiency**: KeyGen is assumed to be a one-time operation whose cost is amortized over many calculations, but we require that Verify is cheaper than evaluating f.

1.2 The Ideal/Real Paradigm

A common method in defining protocol security in multiparty computation is the ideal versus real world model [19]. We consider two separate worlds the real world where the protocol is implemented, executed and attacked, and the ideal world that contains the specification of the protocol's behaviour. When the protocol is properly described and set up in both worlds we can say that a protocol is secure, if its output in the real world cannot be distinguished from its output in the ideal world. The ideal world needs an incorruptable party which is modelled through an ideal functionality F.
For understanding better the paradigm we are going to use a Zero Knowledge protocol where the verifier has an input and the prover wants to prove to verifier that there exists some witness for that input, without reveling anything else. In this ideal case, prover could give input and witness to the trusted third party and then the third party returns true or false to the verifier. Although, in the real world we do not have such third parties and we have to subsitute them with a cryptographic protocol. The real/ideal paradigm requires that whatever information an adversary A could retrieve in the real world, there is a way to retrive the same information in the ideal world.

1.3 Special Ideal Functionalities

Kuramesan and Bentov [20] used the following ideal functionalities to build their protocols. Below we can find a high level overview of them and ther definitions. We should note that the authors provide Universally Composable definitions [21] but they work in the standard model of secure computation. Their model is composed of wallets, safes and coins and is defined as Secure Computation with coins [20].

- **Claim-or-Refund Functionality** F^*_{CR}. It is a two party protocol that accepts deposits from a sender and transfers the deposit to a recipient upon meeting certain conditions set by sender. If the recipient defaults, then the deposit is returned to the sender after a prespecified time.
- **Secure Computation with Penalties** F^*_f. In a n-party setting, a protocol for secure computation with penalties guarantees that if an adversary aborts after learning the output but before delivering output to honest parties, then each honest party is compensated by a prespecified amount.
- **Secure Lottery with Penalties** F^*_{lot}. In a multiparty setting, a protocol for secure lottery with penalties guarantees that if an adversary aborts after learning the outcome of the lottery but before revealing the outcome to honest parties, then each honest party is compensated by a prespecified amount equal to the lottery prize.

1.3.1 Definition of Ideal Functionality F^*_{CR}

This is the main and most important functionality as it will be used for realizing more complex functionalities and protocols. We give a high level overview of F^*_{CR} functionality which allows a

sender P_s to conditionally send coins(q) to a recipient P_r.

- It accepts a deposit of coins(q), a boolean circuit ϕ and a time limit τ from the designated sender P_s.
- Waits until time τ to receive a proof/witness w from a designater recipient P_r which satisfies the following $\phi(w) = 1$.
- If a witness was received before time τ exceeds then the deposit of coins(q) is transferred to the recipient P_r.
- If time τ expires then coins(q) are returned to P_s

We need to give some remarks about the functionallity. The time limit is formalized as a round number like interacting in a game with rounds. When a witness is provided that satisfies the condition set the witness is made public. The definition of the functionality is provided verbatim from the author paper [20]:

F^*_{CR} with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S proceeds as follows:

- *Deposit Phase.* Upon receiving the tuple (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) from P_s, record the message (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) and send it to all parties. Ignore any future deposit messages with the same ssid from P_s to P_r.
- *Claim Phase.* In round τ, upon receiving (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, x, w) from P_r check if
  1. a tuple (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) was recorded
  2. if $\phi_{s,r} = 1$

  If both checks pass, send (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, x, w) to all parties, send (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) to P_r and delete the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x)
- *Refund Phase.* In round $\tau + 1$, if the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) was not deleted then send (refund, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) to P_s and delete the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x)

Figure 1.3.1: Claim or Refund Functionality Definitiion

We should note that there are simpler ways to express F^*_{CR} through CHECKLOCKTIMEVERIFY [22] but this script functionality has not been enabled in Bitcoin yet and it is beyond the scope of this thesis.

1.3.2 Definition of Ideal Functionality F^*_f

The definition of the functionality of secure computation with penalties guarantees the 2 following properties:

- An honest party never has to pay any penalty.
- If a party aborts after learning the output and does not deliver output to honest parties, then every honest party is compensated.

We give a high level overview of the functionality before its definition. All parties send their input to the functionality F_f^*. The functionality allows the ideal world adverasry S to deposit some coins that will be used as a compensation for honest parties in case S aborts the protocol after learning the output and without delivering the result to the honest parties. Honest parties make a deposit during their input phase which is returned to them by the functionality during the execution of the output phase. The adversary S gets the chance to look the output if he deposited sufficient amount of coins such as $x = h * q$ where h are the honest parties and q is the penalty amount. In that case S has two options, if he continues the protocol he sends the output to each honest party and he gets back the coins he deposited or if he aborts the protocol he compensates every honest party using the penalty deposited during his input phase. Below are presented two versions of this functionality the second with a slight modification.

F_f^* with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S that corrupts parties $(P_s)_{s\in C}$ proceeds as follows: Let $H = [n]\backslash C$ and $h = |H|$. Let d be a parameter representing the safety deposit and let q denote the penalty amount.

- Input Phase: Wait to receive a message(input, sid, ssid, r, y_r, coins(d)) from P_r for all $r \in H$. Then wait to receive a message (input, sid, ssid, $(y_s)_{s\in C}$, H', coins($h'q$)) from S where $h' = |H'|$.
- Output Phase:
  - Send (return, sid, ssid, coins(d)) to P_r for all $r \in H$.
  - Compute $(z_1, ..., z_n) \leftarrow f(y_1, ..., y_n)$.
    - If $h' = 0$ then send message (output, sid, ssid, z_r) to P_r, for $r \in [n]$ and terminate.
    - If $0 < h' < h$, then send (extra, sid, ssid, coins(q)) to P_r for each $r \in H'$ and terminate.
    - If $h' = h$, then send (output, sid, ssid, $(z_s)_{s\in C}$) to S.
  - If S returns (continue, sid, ssid, H''), then send (output, sid, ssid z_r) to P_r for all $r \in H$, and send (return, sid, ssid, coins($(h - h'')q$)) to S, where $h'' = H''$ and send (extrapay, sid, ssid, coins(q)) to P_r for all $r \in H''$.
  - Else if S returns (abort, sid, ssid), send (penalty, sid, ssid, coins(q)) to P_r for all $r \in H$.

Figure 1.3.2: Secure Computation with Penalties Functionality Definition

F_f^* with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S that corrupts parties $(P_s)_{s\in C}$ proceeds as follows: Let $H = [n]\backslash C$ and $h = |H|$. Let d be a parameter representing the safety deposit and let q denote the penalty amount.

- Input Phase: Wait to receive a message(input, sid, ssid, r, y_r, coins(d)) from P_r for all $r \in H$. Then wait to receive a message (input, sid, ssid, $(y_s)_{s\in C}$, coins(hq)) from S.
- Output Phase:
  - Send (return, sid, ssid, coins(d)) to P_r for all $r \in H$.
  - Compute $(z_1, ..., z_n) \leftarrow f(y_1, ..., y_n)$ and send (output, sid, ssid, $(z_s)_{s\in C}$) to S.
  - If S returns (continue, sid, ssid), then send (output, sid, ssid z_r) to P_r for all $r \in H$, and send (return, sid, ssid, coins($h * q$)) to S.
  - Else if S returns (abort, sid, ssid, coins(t'hq)), send (extra, sid, ssid, coins($q + t'q$)) to P_r for all $r \in H$.

Figure 1.3.3: Secure Computation with Penalties Functionality Definition 2

1.3.3 Definition of Ideal Functionality F_{Rec}^*

F_{Rec}^* with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S that corrupts parties $(P_s)_{s\in C}$ proceeds as follows: Let $H = [n]\backslash C$ and $h = |H|$. Let d be a parameter representing the safety deposit and let q denote the penalty amount. We assume that F_{Rec}^* is parameterized by the pubNMSS scheme.

- Input Phase: Wait to receive a message(input, sid, ssid, r, AllTags, $Token_r$, coins(d)) from P_r for all $r \in H$. Then wait to receive a message (input, sid, ssid, $(Token_s)_{s\in C}$, H', coins(h'q)) from S where $h' = |H'|$.
- Output Phase:
  - Send (return, sid, ssid, coins(d)) to P_r for all $r \in H$.
  - Compute $(z) \leftarrow Rec(AllTags, Token_1, ..., Token_n)$.
    * If $h' = 0$ then send message (output, sid, ssid, z_r) to P_r, for $r \in [n]$ and terminate.
    * If $0 < h' < h$, then send (extra, sid, ssid, coins(q)) to P_r for each $r \in H'$ and terminate.
    * If $h' = h$, then send (output, sid, ssid, $(z_s)_{s\in C}$) to S.
  - If S returns (continue, sid, ssid, H''), then send (output, sid, ssid z_r) to P_r for all $r \in H$, and send (return, sid, ssid, coins($(h - h'')q$)) to S, where $h'' = H''$ and send (extrapay, sid, ssid, coins(q)) to P_r for all $r \in H''$.
  - Else if S returns (abort, sid, ssid), send (penalty, sid, ssid, coins(q)) to P_r for all $r \in H$.

Figure 1.3.4: Secure Computation with Penalties Functionality - Reconstruct Definition

1.4 Bitcoin Based Timed Commitment Scheme

The commitment scheme is executed between 2 parties the committer and the recipient but it can be generalized to be used amongst N parties. The committer starts the protocol with some secret value s. During the commitment phase the Committer commits himself to some string s by revealing its hash $h = H(s)$. Moreover the parties agree on a moment of time τ until which the Committer should open the commitment. The secret s will become known to every recipient after the opening phase is executed. Informally, we require that, if the committer is honest, then before the opening phase started, the adversary has no information about s. On the other hand, every honest recipient can be sure that, no matter how a malicious sender behaves, the commitment can be open in exactly one way. The standard commitment schemes suffer from the following problem: there is no way to force the committer to reveal his secret s, and, in particular, if he aborts before the Open phase starts then s remains secret. Bitcoin offers a solution to this problem with timelock transactions, since the committer can create transactions which will pay a deposit to every receipient if he does not open his commitment by a specified time. For this to work the pay deposit transactions need to have a bigger value than the commit transaction. The Bitcoin Based timed commitment scheme [23] and consists of 3 phases:

Setup Phase

- The keypair of the Committer is C and each Recipient's is R_i.
- The Committer chooses the secret string s.
- The ledger contains N unredeemed transactions $U_1^C, ..., U_N^C$ which can be redeemed by the committer's key and each of them has value of d bitcoins.

Commit Phase

- The committer computes $h = H(s)$ and sends to the Ledger transactions $Commit_1, ..., Commit_N$ where N is the number of recipients. He reveals h, as it is a part of each $Commit_i$.
- If within time max_{ledger} the $Commit_i$ transactions do not appear on the Ledger or they are incorrect then the parties abort.
- The committer creates the bodies $PayDeposit_i$ transactions for each recipient, signs them and sends the signed body to each recipient. If a transaction does not reach a recipient then he halts.

Open Phase

- Committer sends to the ledger $Open_i$ transactions, which reveal the secret s.
- If within time τ the transaction $Open_i$ does not reach the ledger then the recipient signs and sends to the Ledger the transaction $PayDeposit_i$ and earns N Bitcoins.

Figure 1.4.1: Bitcoin Based Commitment Scheme

The committer will talk independently to each recipient R_i. For each of them he will create in the commitment phase a transaction $Commit_i$ with value d that normally will be redeemed by him in the opening phase with a transaction $Open_i$. The transaction $Commit_i$ will be constructed in such a way that the $Open_i$ transaction has to automatically open the commitment. Technically it will be done by constructing the output script of $Commit_i$ in such a way that the redeeming transaction has to provide s. Of course, this means that the money of the committer is locked until he reveals s. However, to set a limit on the waiting time of the recipient, we also require the committer to send to each receipient a transaction $PayDeposit_i$ that can redeeem $Commit_i$ if time τ passes.

The properties of the CS protocol are as follows:

1. The Recipients have no information about the secret s before the Committer broadcasts the transaction $PayDeposit_i$.
2. The Committer cannot open his commitment in a different way than revealing his secret s.
3. The honest Committer will never lose his deposit, he will receive it back not later than at the time τ.
4. If the Committer does not reveal his secret s before the time $(\tau + \Delta\tau)$ then the Recipients will receive d Bitcoins of compensation.

1.5 Simultaneous Bitcoin Based Timed Commitment Scheme

The diffrence between the BBCS and the Simultaneous one is that if the protocol is not aborted during the commitment phase then both parties are committed. Transaction Commit is constructed by the existing transctions T^A and T^B. The transaction Commit has two outputs, one is used to commit A to s_A and the other one to commit B to s_B. The first output can be claimed by A with revealing her secret or after time τ by B. The latter option is technically achieved by signing at the very beginning of the protocol a transaction $Fuse_A$, which redeems Commit, can be claimed only by B and has a timelock τ. The second output of Commit is analogous.

Setup Phase

- The keypair of the committer is C and recipient is R
- Commiter knows the secret s_c and receipent knows the secret s_r, both players know the hashes $h_c = H(s'_c)$ and $h_r = H(s'_r)$ where $s'_c = (s_c||r_c)$ and $s'_r = (s_r||r_r)$.
- There are two unredeemed transactions T_C, T_R of value d Bitcoins, which can be redeemed with the keys C and R respectively.

Commit Phase

- Both players compute the body of the transaction Commit using T_C, T_R as inputs.
- Both players compute the bodies of the transactions $Fuse_C$ and $Fuse_R$ using appropriate outputs of the Commit transaction. Then, they sign $Fuse_C$ and $Fuse_R$ and exchange the signatures.
- A signs the transaction Commit and sends the signature to B.
- B signs the transaction Commit and broadcasts it.
- Both parties wait until the transaction Commit is included in the block chain.
- If the transaction Commit does not appear on the block chain until time $\tau - 2\tau'$ where τ' is the maximum delay we can have for broadcasting and including it in the Ledger, then A immediately redeemsthe transaction T_C and quits the protocol. Analogously, if A did not send her signature to B until time $\tau - 2\tau'$, then B redeems the transaction T_R and quits the protocol.

Open Phase

- Committer and Recipient broadcast the transactions $Open_C$ and $Open_R$ respectively, what reveals the secrets s_C, s_R.
- If within time τ the transaction $Open_C$ does not appear on the block chain, then Receipient broadcaststhe transaction $Fuse_C$ and gets d Bitcoins. Similarly, if within time τ the transaction $Open_R$ does not appear on the block chain, then Committer broadcasts the transaction $Fuse_R$ and gets d Bitcoins.

Figure 1.5.1: Simultaneous Bitcoin Based Timed Commitment Scheme

Chapter 2

Designing Fair Protocols

In this chapter we focus on designing fair protocols with the help of Bitcoin instead of a trusted third party or a central authority. The protocols presented are lotteries and secure computations. We examine both protocols in the two party and in the multiparty setting.

2.1 Introduction

Secure multiparty computation (MPC) protocols [24][1] allow a group of parties to compute a joint function on inputs they privately contribute to the protocol execution. Beyond privacy, a secure MPC protocol is highly desirable to be fair and robust. By fair, we mean that either all parties learn the result or none and by robust we mean that the delivery of the output is guaranteed and the adversary cannot mount a denial of service against the protocol.

With the advent of Bitcoin [3] and other decentralized cryptocurrencies, the works of [25][23][20] pointed to a new direction considering the fairness property. Fairness could be achieved through the penalty model. In this model a breach of fairness by the adversary is still possible but in case of protocol abortion from the adversary then honest parties are getting compensated. At the same time, in case fairness is not breached, it is guaranteed that no party loses any money (despite the fact that currency transfers may have taken place between the parties). The rationale here is that we are inserting the notion of fairness from the penalty model to force the adversary to operate in the protocol fairly.

While the main idea of fairness with penalties sounds simple enough, its implementation proves to be quite challenging. The main reason is that due to the decentralized nature of cryptocurrencies, they do not rely on trusted third party that will collect money from all parties and then either return it or redistribute it according to the pre-agreed penalty structure. The mechanism used to solve the problem of third party is the capability of the Bitcoin network to issue transactions that are timelocked, become valid only after a specific time and prior to that time may be superseded by other transactions that are posted in the public ledger. Superseded timelocked transactions

become invalid and remain in the ledger without ever being redeemed.

2.2 Two Party Lottery

Adam Back and Iddo Bentov in 2013 [26] have proposed a protocol for coin flipping based on Bitcoin that allow us to have a two party lottery which achieves fairness in the penalty model. The idea has been described in the Bitcoin wiki by G. Maxwell [27] and has been given an informal Bitcoin script structure by [28]. The reason why it provides fairness between malicious adversaries who decide to abort after discovering they lost the bet is that Bitcoin scripting language allows us to have a primitive with which a party locks a certain amount of coins until a specified time in the future and the other party can claim these coins to another address at any time before the specified one upon meeting arbitrary conditions which are specified in advance via a Bitcoin script. If the receiver does not claim the coins then they are returned to the sender.

The primitive is going to be described below: Sender creates a transaction that takes inputs that he/she owns and can be spent with the following condition, '(Sender's signature AND Recipient's signature) OR (arbitrary conditions)'. Arbitrary conditions can be a proof required by sender plus Recipient's signature. Sender keeps the transaction secret and creates a refund transaction that sends the money the transaction created to an address he/she owns but has a time lock set in the future. Sender signs the Refund transaction and sends Recipient a message through a secure channel with the Refund transaction asking him/her to sign it. Recipient can only see the hash of the secret transactio and can protect himself by generating a new public key and can ask sender create the secret transaction with the corresponding public address of this key. Recipient computes his signature for the Refund transaction and sends it to Sender through a secure channel. Now Sender can broadcast his/her transaction to the network. Any node coming with a proof that meets Sender's conditions will claim the coins otherwise the coins will be returned to the Sender after the time lock expires.

We see that this primitive is used later inside this thesis as functionality F^*_{CR} that allows us to construct more complex protocols and describe other functionalities. The functionality has been given a definition in the previous chapter.

2.2.1 Realizing a Two Party Lottery Protocol

Suppose that Alice and Bob wish to do a fair coin toss where each of them inputs X coins and the winner gets the 2X coins. This can be done by selecting the winner according to the least significant bit of two committed secrets, with the following protocol [26]:

1. Alice picks up a random secret s_A and sends a private message to Bob with the value $A = SHA256(s_A)$.
2. Bob picks up a random secret s_B and sends a private message to Alice with the value $B = SHA256(s_B)$.
3. Bob creates a bet transaction that takes 2N of his own coins and can be spent by the following conditions:
   - [Alice's signature AND Bob's signature] OR
   - [SHA256(s_A) == A AND SHA256(s_B) == B AND (((($s_A \oplus s_B$) mod 2 == 0 AND Alice's signature)) OR ((($s_A \oplus s_B$) mod 2 == 1 AND Bob's signature))]
4. Bob asks Alice to sign a $Refund_{bet}$ transaction that which spends his 2N coins to an address that he controls and has locktime of 20 blocks in the future.
5. Bob broadcasts the bet transaction into the network.
6. Alice creates a reveal transaction that takes as input N of her own coins and can be spent by the following condition:
   - [Alice's signature AND Bob's signature] OR [SHA256(s_B) == B AND Bob's signature]
7. Alice asks Bob to sign a $Refund_{reveal}$ transaction which spends her N coins to an address that she controls and has locktime of 10 blocks in the future.
8. Alice broadcasts the reveal transaction to the Bitcoin network.
9. Bob redeems the reveal transaction by revealing s_B.
10. Alice redeems the bet transaction if she won, otherwise she sends s_A to Bob so that he could redeem the "bet" transaction without waiting for the locktime to expire.

Figure 2.2.1: Two Party Lottery in Bitcoin

2.3 Multiparty Lottery

A multiparty lottery protocol [23] is executed among a group of parties. All parties $P_1, ..., P_n$ deposit their money to one common place and the winner will claim them. The deposit is set at N bitcoins so in case of protocol abort we fully compensate each party.

2.3.1 Realizing a Multi Party Lottery Protocol

The general idea behind the secure Multiparty Lottery protocol is that each party first commits to its inputs using the Bitcoin based commitment scheme.

- Each party executes the commitment phase. Once the phases are executed succesfully the parties proceed to the next steps
- Each party posts his transaction PutMoney on the Ledger.
- Once all these transactions appear on the Ledger they create the $Compute_N$ transaction. All parties except P_1 compute their signatures and send them to P_1. P_1 puts all received signatures into inputs of transaction Compute and posts it to the Ledger.
- Once the transaction appears on the Ledger they open the commitments.

Since they used the Bitcoin based commitment scheme, they can now punish the other party that did not open its commitment by executing PayDeposit after the time τ passes, and claim its deposit. Each honest party is always guaranteed to get its deposit back, hence it does not risk anything investing this money at the beginning of the protocol.

We need to define the time τ for the execution of the protocol. Supposing that we need $2 * max_{ledger}$ time until all the parties put their PutMoney transaction to the Ledger, $1 * max_{ledger}$ for P_1 to receive all signatures and post the transaction $Compute_N$ to the Ledger and $1 * max_{ledger}$ for every party to open its commitment we need totally a time $\tau = \tau' + 4 * max_{ledger}$ where τ' is the time the protocol started.

The parameter d should be chosen in such a way that it will fully compensate to each party the fact that a player aborted. The deposit each player has to submit needs to be equal to N(N -1) Bitcoins which makes the protocol efficient for $N \leq 5$. A realization of the protocol is presented in the figure below.

Setup Phase

- For each i, player P_i holds a pair of keys ($P_i.sk$, $P_i.pk$).
- For each i, the Ledger contains a standard transaction T^i that has value 1B and whose recipient is P_i. The Ledger contains also the transactions U^i_j for $i, j \in (1, ..., N)$ and $i \neq j$, such that each U^i_j can be redeemed by P^i and has value $d = N$ Bitcoins.

Init Phase

- Each player P_i generates a pair of keys ($\bar{P}_i.sk$, $\bar{P}_i.pk$) and sends his public key $\bar{P}_i.pk$ to all other players.
- Each player P_i chooses chooses his secret s_i from S^N_k.

Deposits Phase

- Let τ be the current time. For each i, the commitment phase CS.Commit(P_i, d, $\tau + 4 * max_{ledger}$, s_i) is executed using the transactions U^i_j as inputs.
- If any two commitments of different players are equal then the players abort the protocol.

Execution Phase

- For each i, player P_i puts the transaction $PutMoney^i$ to the Ledger. The players halt if any of those transactions did not appear on the Ledger before time $\tau + 2 * max_{ledger}$.
- For each i, $i \geq 2$ player P_i computes his signature on the transaction $Compute^N$ and sends it to the player P_1.
- Player P_1 puts all signatures and his own as transaction inputs in $Compute^N$ and puts it to the Ledger. If n transaction $Compute^N$ does not appear on the Ledger in time $\tau + 3 * max_{ledger}$ then the players halt.
- During each i, the player P_i puts his Open transaction on the Ledger what reveals his secret and sends back to him the deposits he made during the executions of CS protocol. If some player did not reveal his secret in time $\tau + 4 * max_{ledger}$ then all the other players send the appropriate PayDeposit transactions from that player CS protocols to the Ledger to get N Bitcoins.
- The player who wins the lottery gets the pot by sending a transaction $ClaimMoney^{(f_1,...,f_N)}$ to the Ledger.

Figure 2.3.1: MultiParty Lottery in Bitcoin

2.4 Secure Two-Party Computation

A secure two party computation protocol allows two parties to jointly compute an arbitrary function on their inputs without sharing the value of their inputs with the opposing party.

2.4.1 Realizing a Two Party Secure Computation Protocol

Suppose that we have 2 parties, Alice and Bob and they agree on a deposit of d Bitcoins. If the protocol terminates successfully, then both parties get their deposits back. However, if one of the parties interrupts the protocol after she learned the output, the other party takes both deposits [25].

Setup Phase

- A holds a key pair A and B holds the key pair B.
- The parties agree on a function they want to jointly compute and on a value of deposits equal d Bitcoins each.

Computation Phase
The parties execute the two-party protocol of Goldreich and Vainish [29][30] additionally secured against an active adversary with ZK proofs, without reconstructing the secret. At the end of the execution A holds s_A and B holds s_B, such that the result of the computation in equal $s_A \oplus s_B$.
Commit Phase

- A computes her secret s'_A as a concatenation of her share s_A and some random string ρ_A of length α, where α is a security parameter.
- A sends $h_A := H(s'_A)$ to B and makes a zero-knowledge proof to B that this value is indeed equal to $H(s_A||\rho_A)$ for some string ρ_A.
- Similarly, B computes $s'_B := s_B||\rho_B$ for some random string ρ_B of length α, sends $h_B := H(s'_B)$ to A and makes an analogous proof.
- The parties execute SCS.Commit(A, B, d, τ) for some moment of time τ in the future.

Open Phase

- The parties execute SCS.Open(A, B, d, τ) protocol.
- If A reveals s_A before time τ, then B computes s_A as a prefix of s'_A of an appropriate length and computes the result of the computation $s := s_A \oplus s_B$. Otherwise, B earns d Bitcoins from $Fuse^A$ transaction.
- If B reveals s_B before time τ, then A computes s_B as a prefix of s'_B of an appropriate length and computes the result of the computation $s := s_A \oplus s_B$. Otherwise, B earns d Bitcoins from $Fuse^B$ transaction.

Figure 2.4.1: Two Party Secure Computation

2.5 Multi Party Secure Computation

Secure Multi Party computation allows a set of parties to compute an arbitrary function of their private inputs. The protocol described turns MPC into Fair Recontruction to achieve that [20].

2.5.1 Non Malleable Secret Sharing

Given a secret s, we generate tag-token pairs in the following way:

- Perform a n-out of n sharing on s to obtain $s_1, ..., s_n$
- To generate the i-th "tag-token" pair, compute com_i using randomness w_i to secret share s_i by applying the sender algorithm of honest-binding commitment and set $Tag_i = com_i$ and $Token_i = (s_i, w_i)$.

2.5.2 Fair Reconstruction

The reconstruct algorithm guarantees the following properties:

- An honest party never has to pay a penalty.
- If the adversary reconstructs the secret but an honest party cannot then the honest party is compensated.

Then the reconstruction algorith takes as input (AllTags, $Token_i$) and as long as a party collects all the tokes then it can reconstruct the secret. On the other hand, if one token is not revealed then the secret cannot be reconstructed and remains hidden. A sender P_s may use (a set of) tags to specify a F^*_{CR} transaction with the guarantee that its deposit can be claimed by a receiver P_r only if he provides the corresponding (set of) tokens.

2.5.3 The Ladder Mechanism

The mechanism is composed of a sequence of F^*_{CR} deposits made into two phases. In the first phase parties $P_1, ..., P_{n-1}$ make a F^*_{CR} deposit of coins(q) to P_n that can be claimed only if P_n provides the corresponding set of tokens $T_1, ..., T_n$. This phase is called roof deposits. If all roof deposits are completed, then parties proceed to the second phase, known as the ladder deposits. Then, in the second phase, for i = n to 2, each P_n makes a deposit of coins((n-1)*q) to P_{n-1} that can be claimed only if tokens $T_1, ..., T_{n-1}$ are produced by P_{n-1}.

We handle aborts in the deposit phase in the following way. If a corrupt party aborts the protocol(does not make his roof deposit), then all parties terminte the protocol immediately and get their roof deposits back. On the other hand, if a corrupt party P_r fails to make the ladder deposit it is supposed to make, then for all $s \leq r$, party P_s does not make its ladder deposit at all, while for all $s \geq r$, party P_s continues to wait until a designated round to see whether its ladder deposit is claimed.

The ladder deposits are claimed in reverse order. The tokens required to claim the i-th ladder deposit consist of tokens possessed by the recipient of the i-th ladder deposit plus the tokens required to claim the (i + 1)-th ladder deposit. Therefore, if the (i + 1)-th ladder deposit is claimed, then the i-th ladder deposit can always be claimed.

2.5.4 Framework for The Ladder Mechanism

The ladder mechanism is parameterized by a protocol Init, predicates $\phi_1, ..., \phi_n$, and procedures Extend and Recon.

Initialization: Parties $P_1, ..., P_n$ run Init with their respective inputs $x_1, ..., x_n$ to obtain respective outputs $y_1, ..., y_n$. If there is an abort in this step such that some parties did not obtain their outputs, then all parties terminate and output

Roof Deposits: For each $j \in [n-1]$ simultaneously:

$$P_j \xrightarrow[q,\tau_n]{\phi_n} P_n$$

If a party P_j does not make an F^*_{CR} deposit to P_n as above, then each party P_i terminates the protocol and wait to collect refund from $Tx_{n,i}$

Ladder Deposits: For i = n − 1 down to 1:

$$P_{i+1} \xrightarrow[i*q,\tau_i]{\phi_i} P_i$$

Handling aborts. If a party P_{i+1} does not make an F^*_{CR} deposit to P_i as above, then

- each party P_j for $j < i$ does not make its ladder deposit and waits to collect refund from $Tx_{n,j}$
- each party P_j for $j > i$ continues on to the ladder claim phase.

Claims: P_1 claims Tx_1 using witness $\alpha_1 = Extend(1, \perp; y_1)$. For i = 1 to n − 1 (one-by-one), at time τ_i:

- If P_i claimed Tx_i, then let $_i$ be the witness satisfying ϕ_i. P_{i+1} computes $_{i+1} \leftarrow Extend(i+1, \alpha_i; y_{i+1})$. If $i + 1 \neq n$, then P_{i+1} claims Tx_{i+1} using witness $_{i+1}$. If $i + 1 = n$, then P_n claims $Tx_{n,j}$ for all j using witness $_n$.
- If P_i did not claim Tx_i, then P_{i+1} terminates the protocol and waits to collect refund from $Tx_{n,i+1}$ if $i + 1 = n$.

Output: If $Tx_{n,j}$ was claimed by P_n for some j, then let α_n be the witness satisfying ϕ_n. Each party P_i outputs $z_i = Recon(\alpha_n; y_i)$ and terminates the protocol. If no $Tx_{n,j}$ was claimed, then each party outputs $\perp$

Figure 2.5.1: Framework for the Ladder Mechanism

2.5.5 Realizing Multi Party Computation with Penalties Protocol

By applying the previous steps we are reducing the Secure Multi Party Computation problem to a Fair Reconstruction problem. If a party learns the output and aborts the protocol then the honest parties are compensated. The formal description of the protocol is presented below [20]:

1. For $s \in [1, ..., n-1]$, party P_s does the following:
   - Send (deposit, sid, ssid, s, n, ϕ_{rf}, τ_n, coins(q)) to F^*_{CR}
   - If there exists $r \in [n-1]$ such that the message (deposit, sid, ssid, r, n, ϕ_{rf}, τ_n, q) was not received from F^*_{CR}, then output NULL, and if $s \neq n$ then wait to receive message (refund, sid, ssid, s, n, ϕ_{rf}, τ_n, Q^s_{rf}. Terminate the protocol.
2. For s = n−1 to 1, each P_{s+1} sends (deposit, sid, ssid, s+1, s, ϕ^s_{lad}, τ_s, $Q^{s+1}_{lad} = coins(s \cdot q)$) to F^*_{CR} only if for each r = n − 1 to s + 1, the message (deposit, sid, ssid, r + 1, r, ϕ^r_{lad}, τ_r, $r \cdot q$) was received from F^*_{CR}.
3. For each $s \in 0, ..., n-1$, party P_{s+1} does the following:
   - Wait until round tau_s to receive a message (claim, sid, ssid, s + 1, s, ϕ^{lad}_s, τ_s, $s \cdot q$, W_s) from F^*_{CR}. If such a message was received by round tau_s, create $W_{s+1} \leftarrow W_s \cup Token_{s+1}$, and if $s + 1 \neq n$, send message (claim, sid, ssid, s + 2, s + 1, ϕ^{lad}_{s+1}, τ_{s+1}, $(s+1) \cdot q$, W_{s+1}) to F^*_{CR}, and receive back (claim, sid, ssid, s + 2, s + 1, ϕ^{lad}_{s+1}, τ_{s+1}, Q^{lad}_{s+2}) from F^*_{CR}.
   - If no such message was received output NULL, and do:
     - Wait until round τ_{s+1} to receive message (refund, sid, ssid, s + 1, s, ϕ^{lad}_s, τ_s, Q^{lad}_{s+1}) from F^*_{CR}.
     - If $s + 1 \neq n$, wait until round τ_{n+1} to receive message (refund, sid, ssid, s+1, n, ϕ^{lad}_{s+1}, τ_{s+1}, Q^{s+1}_{rf}) from F^*_{CR}, and terminate the protocol.
4. After round τ_{n-1}, for each $s \in [n-1]$, party P_n sends (claim, sid, ssid, s, n, ϕ_{rf}, τ_n, q, W_n) to F^*_{CR}, following which it waits to receive message (claim, sid, ssid, s, n, ϕ_{rf}, τ_n, Q^s_{rf}. Finally, P_n outputs $W_n = Token_1, ..., Token_n$, reconstructs the secret, and terminates the protocol.
5. For each $s \in 1, ..., n-1$, party P_s waits until round τ_n to receive a message (claim, sid, ssid, s', n, ϕ_{rf}, τ_n, q, W_n) from F^*_{CR} for some $s' \in [n-1]$. If such a message is received for some $s' \in [n-1]$, then use $W_n = Token_1, ..., Token_n$ to reconstruct the secret. If no such message is received for $s' = s$, then P_s waits until time τ_{n+1} to receive message (refund, sid, ssid, s, n, ϕ_{rf}, Q^s_{rf}, τ_n) from F^*_{CR}.

Figure 2.5.2: Realizing F^*_{rec} in the F^*_{CR} hybrid model

Chapter 3

Incentivizing Correct Computations

3.1 Introduction

In this chapter we are going to describe a model of incentivizing correct computations in certain cryptographic tasks which are based on the formal definitions of the ideal functionalities described in Preliminairies chapter. We setup protocols for every cryptographic task described and we analyze efficiency improvements in the F_{CR}^{*} hybrid model.

3.2 Verifiable Computation

Verifiable Computation is enabling a delegator to outsource the computation of a function to a worker who expects to get paid in return for delivering the correct output. An incentivizable protocol for verifiable computation between D and W must provide the following guarantee:

- Fast verification: The amortized work performed by delegator for verification is less than the work required to compute f.
- Pay to learn output: Worker obtains coins(q) from delegator if and only if delegator received the correct output of the computation from worker.

Designing an incentivized computation protocol in the F_{CR}^{*} hybrid model can be described as:

- Delegator sends (f, u) to worker.
- Delegator creates F_{CR}^{*} transaction with circuit $\phi_{f,u}$ that lets worker redeem coins(q) if worker reveals a witness w such that $\phi_{f,u}(w) = 1$, where the circuit/script $\phi_{f,u}(w)$ is satisfied if and only if f(u) = w.

The above protocol is sufficient but has several drawbacks when implemented in the Bitcoin network. To validate the claim transaction each miner has to verify whether the witness provided was indeed valid. This means that each miner has to compute f in order to confirm the validity of the transaction. This is clearly undesirable for the following reasons:

1. It puts a heavy load on the Bitcoin network and corresponds to heavy loss of resources.
2. All nodes in the Bitcoin network need to compute f(u) when validating this script, while only worker gets paid. Essentially all the other miners are working for free.

3.2.1 Definition of Ideal Functionality F^*_{exitCR}

F^*_{exitCR} [31] is a modification of F^*_{CR} which allows parties to mutually agree to discard checking the condition to release payment. We give a high level overview of F^*_{exitCR} functionality:

- The sender P_s deposits his/her coins(q) while specifying a timebound τ and a circuit ϕ.
- The receiver P_r can claim the coins(q) by publicly revealing a proof/witness w that satisfies $\phi(w) = 1$.
- At any point τ' before time τ ($\tau' < \tau$), P_s and P_r can agree to release the coins(q) to P_r without revealing w.
- If P_r didn't claim within time τ, coins(q) are refunded to P_s.

The definition of the F^*_{exitCR} is presented below:

F^*_{exitCR} with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S proceeds as follows:

- *Deposit Phase.* Upon receiving the tuple (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) from P_s, record the message (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) and send it to all parties. Ignore any future deposit messages with the same ssid from P_s to P_r.
- *Claim Phase.* In round τ, upon receiving (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, x, w) from P_r check if
  1. a tuple (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) was recorded
  2. if $\phi_{s,r} = 1$

  If both checks pass, send (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, x, w) to all parties, send (claim, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) to P_r and delete the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x)
- *Exit Phase.* In round τ', where $\tau' < \tau$, send (release, sid, ssid, s, r, $\phi_{s,r}$, τ', x) to all parties, send (release, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) to P_r and delete the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x)
- *Refund Phase.* In round $\tau + 1$, if the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x) was not deleted then send (refund, sid, ssid, s, r, $\phi_{s,r}$, τ, coins(x)) to P_s and delete the record (deposit, sid, ssid, s, r, $\phi_{s,r}$, τ, x)

Figure 3.2.1: Claim or Refund Functionality Definitiion with Exit Clause

3.2.2 Incentivizing Public Verifiable Computation

We present a high level overview of the Public Verifiable Computation below, informally described in [31]

1. Delegator and worker engage in secure computation to obtain $(ek_f, vk_f) \leftarrow KeyGen(f, 1^\lambda)$.
2. Delegator sends input u to worker.
3. Delegator and worker invoke F^*_{exitCR} with circuit $\phi = Verify(vk_f, u, \cdot)$ and timebound τ that lets worker earn delegator's coins(q) if worker reveals $w = (y, \psi_y)$ such that $Verify(vk_f, u, (y, \psi_y)) = 1$.
4. Worker executes $(y, \psi_y) \leftarrow Compute(ek_f, u)$ and sends $w = (y, \psi_y)$ to delegator within time $\tau' < \tau$.
5. Delegator verifies (y, ψ_y), then delegator and worker release the coins(q) to worker.
6. If delegator doesn't release the coins(q) until time τ' then worker will redeem the F^*_{exitCR} transaction between time τ' and τ and claim the coins(q).

Figure 3.2.2: Public Verifiable Computation Protocol

The above protocol minimizes the validation complexity. Current public verification schemes [18] still require 288 bytes storage and 9ms to verify. Therefore, each miner has to spend 9ms to execute the verification algorithm to validate the F^*_{CR} transaction. In case, we have delegator and worker who are interested in reducing further the validation complexity we can follow the steps below [31]:

1. Execute the steps of the previous protocol until Step 4.
2. Worker executes $(y, \psi_y) \leftarrow Compute(ek_f, u)$ and sends $w = (y, \psi_y)$ to delegator.
3. Delegator runs $Verify(vk_f, u, (y, \psi_y))$ to check if answer is 1.
4. Delegator creates another transaction which pays the worker. If he does not pay worker, then worker can always claim the $F^*_{exit,CR}$ transaction.
5. If delegator pays worker then they use the release condition of $F^*_{exit,CR}$ transaction. Therefore, a malicious worker cannot get paid twice by also claiming coins(q) of $F^*_{exit,CR}$ transaction.

Figure 3.2.3: Public Verifiable Computation Protocol

3.3 Fair Computation

Secure Computation with Penalties has been discussed in the previous chapter. In this section we describe how to design fair protocols that are more round-efficient than the previous constructed protocol. The efficiency is gained by a new ideal functionality called F^*_{ML} [31].

3.3.1 Definition of Ideal Functionality F^*_{ML}

The multilock functionality allows n parties to lock their coins where each party P_i commits to the following statement: Before round τ, I need to reveal a witness w_i that satisfies $\phi_i(w_i) = 1$, or else I will forfeit my security deposit of x coins.

- The design of F^*_{ML} guarantees that either all the n parties agreed on the circuits ϕ_i, the timebound τ, and the security deposit amount x, or else no coins become locked.
- Each corrupt party who aborts after the coins become locked is forced to pay $coins(\frac{x}{n-1})$ to each honest party.
- If P_i reveals the w_i which satisfies ϕ_i then the witness w_i is made public.
- The limit τ prevents the possibility that a corrupt party learns the witness of an honest party, and then waits for an indefinite amount of time before recovering her own coins amount.

Below we can see the definition of the ideal functionality [31]:

F^*_{ML} with session identifier sid, running with parties P_1,...,P_n, a parameter 1^k and an ideal adversary S proceeds as follows:

- Lock phase. Wait to receive (lock, sid, ssid, i, $D_i = (x, \phi_1, ..., \phi_n, \tau)$, coins(x)) from each P_i and record (locked, sid, ssid, i, D_i). Then if $\forall$ i,j : $D_i = D_j$ send message (locked, sid, ssid) to all parties and proceed to the Redeem phase. Otherwise, for all i, if the message (locked, sid, ssid, i, D_i) was recorded then delete it and send message (abort, sid, ssid, i, coins(x)) to P_i and terminate.
- Redeem phase. In round τ: upon receiving a message (redeem, sid, ssid, i, w_i) from P_i, if $\phi_i(w_i) = 1$ then delete (locked, sid, ssid, i, D_i), send (redeem, sid, ssid, coins(x)) to P_i and (redeem, sid, ssid, i, w_i) to all parties.
- Payout phase. In round $\tau + 1$: For all i $\in$ [n]: if (locked, sid, ssid, i, D_i) was recorded but not yet deleted, then delete it and send the message (payout, sid, ssid, i, j, $coins(\frac{x}{n-1})$) to every party $P_j \neq P_i$.

Figure 3.3.1: The ideal functionality F^*_{ML}

The protocol proceeds in the following way:

- Parties run an MPC protocol that accepts inputs y_i and computes $z \leftarrow f(y_1, ..., y_n)$
- The output z is splitted into shares $sh_1, ..., sh_n$ with pubNMSS primitive and then for every $j \in [n]$, computes honest binding commitments Tag_j on share with the corresponding

decommitment $Token_j$.

- At the end of the protocol each party P_i has $(Tag_i, ..., Tag_n, Token_i)$
- For the Fair Reconstruction part instead of using the ladder mechanism with F^*_{CR} transactions we use F^*_{ML}.

3.3.2 Bitcoin Enhancement Proposal

F^*_{ML} functionality cannot be deployed under the current Bitcoin Protocol. That is the reason why the authors suggested two modifications in the Bitcoin protocol.

- Reference to tx_{old} in the transaction tx_{new} by using SHA256d(tx^{simp}_{old})
- Using SHA256d(tx_{old}) as the id of tx_{old}.

The second modification was proposed as a solution to transaction malleability which computes the transaction hash over the body of the transaction without its input scripts. The advantage that it provides is that it allows a user to commit coins on condition that another transaction would become valid. The first modification allows us to know which proof a party revealed since the proof of work computations on the root of the Merkle tree to which SHA256d(tx_{old}) belongs will commit to the witness that redeemed tx_{old}. In a later section we will present the threshold signature scheme which allows a valid signature to be produced by all n voters together. In particular, a different signature cannot be efficiently produced for a previously signed transaction, unless all n voters agree to re-sign it.

Chapter 4

Electronic Voting

Electronic voting is a special case of secure multiparty computation where it allows a group of people to jointly make a decision while keeping individual decision private. While MPC can secure the outcome is received by all parties they cannot guarantee that the outcome is respected. The e-voting problem can be distinguished in decentralized e-voting systems [32][33][34], where voters run a multi-party computational protocol without any additional parties and centralized e-voting systems, where election administrators run the election [35][36]. We study the decentralized e-voting problem via bitcoin. Suppose that we have N voters and 2 candidates. The winning candidate who is the one who gets the majority of the votes will claim N Bitcoins. The decentralized e-voting protocols proposed by Zhao and Chan [37] satisfy the following properties:

- *Privacy*: Each voter can keep his vote private and each voter can prove that he followed the protocol.
- *Irrevocability*: Once the outcome has been revealed nobody can withdraw his funds.

The participants need to participate to the voting protocol as the winner is going to claim the funds. The voting protocols consists of 2 phases, the first phase is the ballot commitment phase where the voters commit themselves to their vote and the ballot casting phase where voters cast their vote. For the ballot commitment phase we propose a protocol based on randomzation of votes to integers. For the ballot casting we will use again 2 different protools, one is the ladder mechanism and the other is to lock the votes into one compute transaction.

4.1 Ballot Commitment via Randomization of Ballots

In this stage, each voter P_i has a vote $V_i \in \{0, 1\}$. The idea is that we are going to convert each vote to a random number and then check if the sum of the random numbers is bigger than n/2. If it is bigger then candidate B has won otherwise candidate A has won. Therefore, there is a limitation on ballot masking method and we cannot extend the protocol to m candidates. For this reason we propose later another ballot masking protocol with Decisional Diffie Hellman.

Each voter P_i generates n numbers whose sum is 0. For each i and j, voter P_i sends the number r_{ij} to P_j via a secret channel. Every voter has a number R_i, makes a commitment C_i, creates his masked vote by $\hat{V}_i = V_i + R_i$ and commits also to $\hat{V}_i$ with commitment $\hat{C}_i$. The commitments are broadcasted publicly while the opening keys and the votes remain secret. Every voter persuades the rest of the voters that he follows the protocol by providing zero knowledge proofs. A formal desciption of this protocol is given below [37]:

This protocol runs among n voters, where for $i \in [n]$, party P_i has secret vote $V_i \in \{0, 1\}$. We assume the proving and verification keys for zk-SNARKs are already generated and distributed to all voters. For each $i \in [n]$ voter P_i the procedure is:

1. For each $j \in [n]$ generate n random numbers whose sum $\sum_j r_{ij} = 0$. For each $j \in [n]$, commit $(c_{ij}, k_{ij}) \leftarrow Commit(r_{ij})$
2. Generate zero knowledge proofs that $\sum_j r_{ij} = 0$. The circuit C evaluates to 1 if the opened values sum to 0. Broadcast the commitments and zero-knowledge proofs to all other voters.
3. Receive commitments and verifies the zero-knowledge proofs from all other parties.
4. For all $j \in [n] \backslash \{i\}$, send to P_j the opening key k_{ij}. For $j \in [n]$ i, wait for the opening key k_{ji} from P_j, and check that $r_{ji} = Open(c_{ji}, k_{ji})$.
5. Compute $R_i \leftarrow \sum_j r_{ji}$ and $\hat{V}_i \leftarrow R_i + V_i$ and commit $(C_i, K_i) \leftarrow Commit(R_i)$ and $(\hat{C}_i, \hat{K}_i) \leftarrow Commit(\hat{V}_i)$, where $K_i, \hat{K}_i$ are the opening keys. Broadcast the commitment C_i and $\hat{C}_i$ publicly.
6. Generate and broadcast publicly the zero-knowledge proofs for the following:
   (a) $R_i = \sum_j r_{ji}$
   (b) The committed value in $\hat{C}_i$ minus that in C_i is either 0 or 1.
7. Receive and verify all proofs from other parties generated in previous step. The protocol terminates.

Figure 4.1.1: Ballot Commitment Phase

The ballot masking protocol does not require the use of the Bitcoin. When the execution has ended, all parties hold a random number, they will open during the ballot casting phase which makes use of Bitcoin and Claim or Refund functionality to enforce fairness.

4.2 Ballot Casting via Ladder Mechanism

Below we give a formal description of the Voting Protocol based on Vote Commitment and Vote Casting.

Deposit phase

- P_n submits 2 instances of the Claim or Refund, which delivers the amount of n bitcoins to the winner as long as he provides a proof that he has the biggest sum of the votes between the candidates.

$$P_i \xrightarrow[n,\tau_{n+1}]{\hat{V}_1,...,\hat{V}_n} A$$
$$P_i \xrightarrow[n,\tau_{n+1}]{\hat{V}_1,...,\hat{V}_n} B$$

- Simultaneously for each $i \neq n$, P_i verifies that the deposit transaction broadcasted in the previous step is on the block chain, and broadcasts the deposit transaction of the following F^*_{CR} instance to the bitcoin system:

$$P_i \xrightarrow[2,\tau_n]{\hat{V}_1,...,\hat{V}_n} P_n$$

- Sequentially for i from n down to 2: P_i verifies that all deposit transactions broadcast previously have appeared in the blockchain, and broadcasts the deposit transaction of the following F^*_{CR} instance to the bitcoin system:

$$P_i \xrightarrow[(i-1),\tau_{i-1}]{\hat{V}_1,...,\hat{V}_{i-1}} P_{i-1}$$

Claim phase

- For $i \neq n$, if before time τ_i, all previous secrets $\hat{V}_1, ..., \hat{V}_{i-1}$ are revealed, then P_i reveals his secret $\hat{V}_i$ and use the claim transaction to receive i Bitcoins from P_{i+1}.
- If before time τ_n, all secrets $\hat{V}_i$ for $i \neq n$ are revealed, P_n reveals his secret $\hat{V}_n$ and use the claim transactions to receive q Bitcoins from each P_i for $i \neq n$.
- If before time τ_{n+1} all secrets are revealed, the winner is determined and he can use the corresponding claim transaction to receive n Bitcoins from P_n.
- At any time when the locktime of a COR instance has passed, the sender can immediately use the corresponding refund transaction to get his amount back.

Figure 4.2.1: Ballot Casting via Ladder Mechanism

Our first contribution comes at this point. As we have seen before a F^*_{CR} transaction is a 2 party protocol and is based on the primitive (Sender's signature AND Receiver's signature) OR (arbitrary conditions). Before we describe our Ballot Casting Protocol via the Ladder Mechanism we will give a definition of Claim or Refund transaction between one sender and N possiblw winning candidates.

The functionality suffices in a way that a party can commit a transaction where a group of parties may be possible candidates to claim. In our case, the winner of the voting will be able to provide proof that he won and he will claim the N bitcoins.
We give a high level overview of the functionality which allows a sender P_s to conditionally send N bitcoins to a receiver P_r who comes from a group of possible winners.

- The sender P_s deposits his/her N Bitcoins into a BET transaction while specifying a time-bound τ and a circuit ϕ. P_s does not brodcast this transaction.
- The sender P_s asks all the possible candidates to sign the $REFUND_{BET}$ transaction and then brodcasts the BET transaction to the blockchain.
- Waits until time τ to receive a proof/witness w from one of the possible winners P_r which satisfies the following $\phi(w) = 1$.
- If a witness was received before time τ exceeds then the deposit of N Bitcoins is transferred to the winner P_r.
- If time τ expires then P_s can claim N Bitcoins from the $REFUND_{BET}$ transaction.

Consequently we can reduce the complexity of the first step of the deposit phase by making an instance of the new claim or refund functionality. The last party P_n does not have to deposit two times the money for the winner and every voter(besides the last one) deposits 1 Bitcoin instead of 2 during the roof deposits phase. Moreover, it exists less overload to the network as we have reduced the number of transactions that need to be brodcasted to the blockchain.

4.3 Ballot Casting via MultiLock Transactions

To participate each voter P_i needs (1 + d)Bitcoins, of which 1 Bitcoin is to be paid to the winning candidate if everyone reveals his masked vote and the d Bitcoins is for deposit that will be used for compensation if P_i does not reveal his masked vote. The protocol guarantees the following [37]:

- If a voter reveals his masked vote, he can get back the deposit.
- If every voter reveals his masked vote, the sum of the votes determines the winner who receives N Bitcoins.
- If at least one voter does not reveal his masked vote, the n Bitcoins originally intended for the winner will be locked. For each voter that does not reveal his masked vote, his deposit will be used for compensation.

The idea is that the n voters will sign some transaction COMPUTE together by contributing their inputs. The protocol locks each voter's contribution which can be redeemed if all voters agree universally. On the other hand, if the protocol is aborted and the COMPUTE transaction is not created successfully then the voters get their contribution back using REFUND transaction. Due to malleability problems we are going to use (n,n)-threshold signature scheme [16]. The n voters jointly generate a group address such that voter P_i learns the group public key $\hat{pk}$ and his share $\hat{sk}_i$ of the private key. Therefore the group private key can only be reconstructed only if the n voters provide their $\hat{sk}_i$.

At the start of the protocol all voters will sign a COMPUTE transaction where they contribute their inputs. Their contribution is locked until it gets permission from every voter to be claimed. If the protocol is aborted then we want the voters that contributed until the corrupted voter who aborted to get their inputs back.

Lock Phase Each voter locks (1+d) Bitcoins into the system, where 1 Bitcoin is to fund the winner, and d Bitcoins is for deposit in case of protocol abort

- P_i creates a transaction $Lock_i$. Its input is (1 + d) Bitcoins owned by P_i, and its output is the address of the group public key $\hat{pk}$. P_i also creates a simplified transaction $Refund_i$ that transfers the money from $Lock_i$ back to P_i. It does not broadcast yet the transaction. P_i broadcasts (simplified) $Refund_i$ to all other voters.
- On receiving $Refund_j$ for $j \in [n] \backslash \{i\}$, P_i checks that the hash value referred to by its input is not hash($Lock_i$). At this point, P_i has only contributed coins to $\hat{pk}$ through the transaction $Lock_i$, and hence, he can sign anything else using $\hat{sk}_i$ without losing money.
- For each $j \in [n]$, P_i participates in the threshold signature scheme to sign $Refund_j$ using his secret key share $\hat{sk}_i$.
- On receiving the correct signature for $Refund_i$, P_i is ready to submit $Lock_i$ to the bitcoin network later.

Figure 4.3.1: Ballot Casting via MultiLock Transaction - Lock Phase

After the lock phase is completed the n voters will sign the JOIN transaction using the threshold signature scheme. The JOIN transaction has n inputs of (1+d) bitcoins and n+1 outputs where one is the prize of N Bitcoins for the winner and the rest n handle the deposit d of each voter. The deposit of each voter can be claimed if he reveals his masked vote with a key associated to P_i or is signed by the group signature. In any case before JOIN appears on the blockchain every party has to create a PAY transaction which redeems his deposit after some time if he does not reveal his vote.

Compute Phase Assume that the MultiLock Protocol has been run, and each P_i has created the transaction $LOCK_i$, whose hash is publicly known.

- Voters generate the simplified transaction JOIN
  - JOIN has n inputs coming from $Lock_i$ hat contribute 1+d Bitcoins
  - JOIN has n+1 outputs
    1. $deposit_i$, $i \in [n]$: each has value d Bitcoins, and requires either the opening key $\hat{K}_i$ and a signature verifiable with P_i's public key pk_i or a valid signature verifiable with the group's public key $\hat{pk}$
    2. prize: has value n Bitcoins, and requires all opening keys K_i's and a signature from the winning candidate.
- The voters jointly sign JOIN using the threshold signature scheme, each with his private key share sk_i.
- Each voter generates, for each $i \in [n]$, the same simplified transaction PAY_i with timelock t_2 whose input refers to $out - deposit_i$. The output handles the compensation dB if voter P_i does not reveal his masked vote by time t_2 . For instance, with $d = m * n$, the compensation can be shared between the m candidates. The n voters jointly sign PAY_i using the threshold signature scheme. We should note tht this realization is not very efficient for $m \geq 5$
- Each voter P_i verifies that the above steps have been completed, and submit $LOCK_i$ to the bitcoin system.
- After all $LOCK_i$'s have appeared on the blockchain, JOIN is submitted to the blockchain.
- As long as JOIN has not appeared on the blockchain, say by time t_1, any voter P_i can terminate the whole protocol by submitting $BACK_i$ to get back (1 + d) Bitcoins.

Figure 4.3.2: Ballot Casting via MultiLock Transaction - Compute Phase

After JOIN appears on the blockchain each voter can get his deposit back by submitting a transaction which provides the opening key of the commitment for his masked vote. If all voters claim their deposits the winner is determined and he gets N Bitcoins. If a voter does not submit the opening key for his masked vote then the N bitcoins are locked and his deposit is used as a compensation.

4.4 Ballot Commitment via Decisional Diffie-Hellman

In this section we are going to present a protocol which is fair, robust and can be extended to multiple candidates. The above protocols described they two major drawbacks:

- They cannot support m candidates due to the nature of Ballot masking.
- The protocols are not fault tolerant due to the way the ladder protocol is constructed and due to the nature of Ballot masking.

We are going to present a protocol which is based on Decisional Diffie-Hellman [32][33][34] for ballot masking and on joint transaction for Ballot Casting. Therefore, we can achieve elections for multiple candidates and avoid denial of service attacks as we are going to treat malicious voters as voters who did not cast a ballot. In bibliography [32][33][34] we have reached a consensus that the following properties are essential for decentralized voting schemes:

- *Perfect Ballot Secrecy*: A voter's vote is private, besides what can be computed from the published tally.
- *Self-tallying*: At the end of the protocol, voters and observers can tally the election result from public information.
- *Fairness*: Nobody has access to partial results before before casting their vote.
- *Dispute-freeness*: A scheme is dispute free if anyone can verify that the protocol was run correctly and that each voter acted according to the rules of the protocol.

In addition, we will add robustness which was proposed by Khader et al [38]

- *Robustness*: A malicious voter cannot prevent the election result from being announced.

We assume the use only of an authenticated brodcast channel to every participant. This assumption is made also in [32][33][34][38]. There are several ways to realize such a public channel: by physical means or digital signatures [32][33]. There is no need for private channels or trusted third parties.

Let G denote a finite cyclic group of prime order q in which the Decision Diffie-Hellman (DDH) problem is computationally hard. Let h be a generator in G. There are n participants, and they all agree on (G,h).

Setup Phase

- All voters now select at random an element in $\mathbb{Z}q$. Each voter P_j keeps x_j secret and publishes to $h_j = h^{x_j}$ and a zero-knowledge proof that g_j has been constructed correctly by proving knowledge of x_j.
- Each voter P_i checks the validity of the zero knowledge proofs and computes $h_i^{y_i} = \frac{\prod_{j=1}^{i-1} h_j}{\prod_{j=i+1}^{n} h_j}$.

Figure 4.4.1: Ballot Masking via Decisional Diffie Helmann

During setup round, we use Zero Knowledge Proofs to ensure participants follow the protocol faithfully. The same technique is also used in [32][33][34]. During setup round, each voter needs to demonstrate his knowledge of the exponent without revealing it. We can use Schnorr's signature [39], which is a well-established technique. Let H be a publicly agreed, secure hash function. To prove the knowledge of the exponent for h^{x_i} one sends $(h^u, r = u - x_i z)$ where $u \in \mathbb{Z}q$ and $z = H(h, h^u, h^{x_i}, i)$. This signature can be verified by anyone through checking whether h^u and $h^r h^{x_i z}$ are equal.

During vote round, each participant needs to demonstrate that the encrypted vote is one of {1,0} without revealing which one. We adapt an efficient technique proposed by Cramer, Damguard and Schoenmakers in [40] and more about it can be found in [32][34].

We keep the round complexity in 3 rounds while we avoid the drawback which exists in [34] where the last voter is able to calculate the result before he/she votes. To avoid this problem we use commitment schemes applied on Bitcoin [23]. This happens during the COMPUTE phase where voters commit to their masked votes and lock them as an input to the COMPUTE transaction. After, all voters have locked their votes, every voter has to reveal his/her commitment otherwise he will pay a fee.

4.5 Ballot Casting via Compute Trasaction

As long as the voters lock their votes into COMPUTE transaction, they have to open their commitments as to calculate the final result of the voting. Since, every transaction is made public, observers and voters can solve the discrete logarithm problem and get the election result.

Voting Phase

- The voter constructs $b_i = h_j^{x_i y_i} g^{v_i}$ where $V_i \in \{0, 1\}$. A disjunctive proof of equality between discrete logarithms $\log_g a_i = \log_{h_i} b_i/g$ is computed, to prove that $V_i \in \{0, 1\}$.
- Each voter casts his ballot b_i via the COMPUTE transaction as in section 3.4.3. Since his ballot b_i is a commitment locked as an input to the COMPUTE transaction the last candidate cannot exploit the result of the election.

Self-Tallying Phase

- If all voters open their commitments the self-tallying property allows the election result to be derived by observers and voters. The result can be calculated $u = log_g V$ where $V = \prod_{i=1}^{n} b_i = g^{\sum_{i=1}^{n} v_i}$. Although the computation of the discrete logarithm is hard in general, we know that the election result is such that $1 < v < n$ and, therefore, the search for the value v is feasible using baby step giant step algorithm.
- If some of the voters are not honest then we use the method described in the next section to get a subset of honest voters and calculate the result of the election.

Figure 4.5.1: Ballot Casting via Compute Transaction

4.6 Robustness

In the protocol proposed in sections 3.4.4 and 3.4.5 a voter can prevent the election result by aborting. In this section, we add a recovery round which was proposed in [38] where the election result will be calculated with honest majority. Let's assume that L is the group of voters that did not open their commitment to the COMPUTE transaction. A recovery round can be executed as

follows to allow the election result to be announced. Each voter computes $\hat{h_j^{y_i}} = \frac{\prod_{j\in\{i+1,\ldots,n\}} h_j}{\prod_{j\in\{1,..,i-1\}} h_j}$ together with a signature of knowledge asserting $log_g h_j = log_{\hat{h_j}} \hat{h_j}^{y_i}$.

The outputs of this computation help us to eliminate the need for private keys of voters who did not open their committed vote. If we want to calculate the result of the election for all voters L, such that all the signatures of knowledge hold, the result can be calculated as $u = log_g V$ where $V = \prod_{j\in\{L\}} \hat{h_j}^{y_i x_i} * h_j^{y_i x_i} * g_i^u = \prod_{j\in\{L\}} \hat{h_j}^{y_i x_i} * b_i$.

4.7 Extending to k Candidates

Assuming we have n voters and k candidates [34]. A value m is chosen such that it is the smallest integer where $2^m > n$. The modification we do is tht each vote is encoded in $V_i = \{2^0 c_0, 2^{(k-1)} c_1, 2^{(k-1)2} c_2, 2^{(k-1)2} c_2, ..., 2^{(k-1)m} c_{k-1}\}$ where c_j is the number of votes for every candidate. The value v can be efficiently computed by using baby-step giant-step algorithm and values $c_1, ..., c_k$ can be recovered using the super-increasing nature of the encoding and with the help of knapsack algorithm.

Part 4

Conclusion

Chapter 1

Conclusion

1.1 Summary

In this thesis we have studied the Bitcoin system, which is a decentralized cryptocurrency based on a P2P network. In the first chapter we tried to described the inner workings of the network, starting with some small introduction on cryptographic primitives, seeing how peers communicate inside the network. We proceeded further, exploring the core concepts of Bitcoin which are transactions and blocks while we are giving the basic Script functionality.

In the second part we are focusing on secure multiparty computation protocols. In bibliography, it is proven that fairness cannot exist without a trusted third party. Our purpose is, to elaborate on that problem by studying several problems proposed by other authors and we focus on decentralized e-voting.

Our contribution is twofold, at first we extended Claim or Refund functionality from 2 parties (a sender and a recipient) to N paarties (a sender and N possible recipients). Therefore, we manage to reduce the amount of money the parties have to deposit to the protocol and we avoid overloading the protocol with non-needed transactions. The e-voting protocols proposed consider have two drawbacks:

1. They consider only 2 possible candidates.
2. They are not designed to avoid a denial of service attack, allowing a malicious party to abort the protocol which results in aborting the election.

We propose a protocol based on Decisional Diffie Hellman for ballot masking and on Multilock transactions for ballot casting based on Bitcoin Based Timed Commitments. We present this protocol in the 2-candidate setting (yes/no voting), then we extend it to m candidates. Furthermore, we avoid abortion of the protocol, by adding an extra recovery round which calculates the result of the election only with the honest majority of the voters. The protocol has a complexity of 2 rounds for voting but needs an additional Bitcoin round, thus the parties can open their commitments and

everyone can compute the result of the election.

1.2 Future Work

Our work and the work already proposed in literature can be extended in several ways:

- Define more complex functionalities than F^*_{CR} or F^*_{ML} which can be used to create new and more complex protocols.
- Adapt more cryptographic tasks which belong to the secure multiparty computation theory to the model proposed by this thesis.
- Create protocols that their security belongs to the Universal Composable Framework.
- Reduce on-chain and off-chain complexity of the Bitcoin proposed protocols.

Bibliography

[1] A. C.-C. Yao. "How to generate and exchange secrets". In: *FOCS* (1986).

[2] Richard Cleve. "Limits on the Security of Coin Flips When Half the Processors are Faulty". In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC)*. 1986, pp. 364–369.

[3] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: `http://bitcoin.org/bitcoin.pdf` (visited on 06/09/2015).

[4] Adam Back. *Hashcash - a denial of service counter-measure*. 2002. URL: `www.hashcash.org/papers/hashcash.pdf` (visited on 06/09/2015).

[5] Unknown. *Merkle Trees*. `http://en.wikipedia.org/wiki/Merkle_tree`. Accessed: 2015-03-30.

[6] K. Okupski. *Bitcoin Developer Reference*. 2014. URL: `enetium.com/resources/Bitcoin.pdf` (visited on 10/22/2015).

[7] N. Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation* 48.177 (1987), pp. 203–209.

[8] V. Miller. "Uses of elliptic curves in cryptography". In: *Advances in Cryptology - CRYPTO '85, Lecture Notes in Computer Science* (218 1986), pp. 417–426.

[9] D. Brown. *Standards for Efficient Cryptography Group*. `http://www.secg.org/sec2-v2.pdf`. Accessed: 2014-10-12.

[10] Mike Caldwell, Aaron Voisine. *Passphrase-protected private key*. `https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki`. Accessed: 2015-09-12.

[11] Andreas Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. 1st ed. O'Reilly, 2014.

[12] Unknown. *Mini Private Keys*. `https://en.bitcoin.it/wiki/Mini_private_key_format`. Accessed: 2015-02-03.

[13] N. Odell. *Finding other Bitcoin Clients*. `https://bitcoin.stackexchange.com/questions/3536/how-do-bitcoin-clients-find-each-other`. Accessed: 2014-07-16.

[14] Unknown. *Bitcoin Developer Guide*. `https://bitcoin.org/en/developer-guide`. Accessed: 2015-04-21.

[15] Gavin Andresen. *Block v2, Height in Coinbase*. `https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki`. Accessed: 2014-09-23.

[16] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. "Snarks for C: verifying program executions succinctly and in zero knowledge". In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*. Santa Barbara, CA, USA, 2013, pp. 1676–1683.

[17] Oded Goldreich. *Foundations of Cryptography - vol 1*. 1st ed. Cambridge University Press, 2001.

[18] B. Parno, J. Howell, C. Gentry and M. Raykova. Pinocchio. "Nearly practical verifiable computation". In: *Security and Privacy*. 2013.

[19] Oded Goldreich. *Foundations of Cryptography - vol 2*. 1st ed. Cambridge University Press, 2004.

[20] Iddo Bentov and Ranjit Kumaresan. "How to use bitcoin to design fair protocols". In: *ePrint 2014/129* (2014).

[21] Ran Canetti. "Security and Composition of Multiparty Cryptographic Protocols". In: *Journal of Cryptology* 13.1 (2000), pp. 143–202.

[22] Peter Todd. *OP CHECKLOCKTIMEVERIFY*. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`. Accessed: 2016-06-23.

[23] M. Andrychowicz, S. Dziembowski, D. Malinowski and L.Mazurek. "Secure multiparty computations on bitcoin". In: *IEEE Security and Privacy* (2014).

[24] O. Goldreich, S. Micali and A. Wigderson. "How to play any mental game". In: *STOC* (1987).

[25] M. Andrychowicz, S. Dziembowski, D. Malinowski and L.Mazurek. "Fair two-party computations via the bitcoin deposits". In: *First Workshop on Bitcoin Research, FC* (2014).

[26] A. Back and I. Bentov. *Fair coin toss with no extortion and no need to trust a third party*. 2013. URL: `https://bitcointalk.org/index.php?topic=277048.0` (visited on 04/12/2016).

[27] G. Maxwell. *Zero Knowledge Contingent Payment*. 2011. URL: `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment` (visited on 05/01/2016).

[28] S. Barber, X. Boyen, E. Shi and E. Uzum. "Bitter to better - how to make bitcoin a better currency". In: *FC* (2012).

[29] Oded Goldreich and Ronen Vainish. "How to solve any protocol problem - an efficiency improvement". In: *CRYPTO* (1987).

[30] Ronald Cramer. "Introduction to secure computation". In: *Lectures on Data Security* (1998).

[31] R. Kumaresan and I. Bentov. "How to use bitcoin to incentivize correct computations". In: *CCS*. 2010, pp. 30–41.

[32] A. Kiayias and M. Yung. "Self-tallying elections and perfect ballot secrecy". In: *Public Key Cryptography '02. LNCS, vol. 2274*. 2002, pp. 141–158.

[33] Jens Groth. “Efficient maximal privacy in boardroom votisng and anonymous broadcast”. In: *Financial Cryptography ’04. LNCS, vol. 3310*. 2004, pp. 90–104.

[34] Fao Hao, Peter Y. A. Ryan and Piotr Zielinski. “Anonymous voting by two-round public discussion”. In: *Journal of Information Security*. 2010, 4(2):62–67.

[35] A. Juels, D. Catalano and M. Jakobsson. “Coercion-Resistant Electronic Elections”. In: *Proc. of Workshop on Privacy in the Electronic Society (WPES05)*. Alexandria, VA, USA, 2005, pp. 61–70.

[36] Peter Y. A. Ryan and Vanessa Teague. “Pretty Good Democracy”. In: *Proc. of the 17th Security Protocols Workshop*. 2009.

[37] Z. Zhao and T.-H. H. Chan. “How to vote privately using bitcoin”. In: *ePrint 2015/1007* (2014).

[38] D. Khader, B. Smyth, P. Y. Ryan and F. Hao. “A fair and robust voting system by broadcast”. In: *in EVOTE’12: 5th International Conference on Electronic Voting*. 2012.

[39] C.P. Schnor. “Efficient signature generation by smart cards”. In: *Journal of Cryptology. vol. 4, no .3*. 1991, pp. 161–174.

[40] R. Cramer, I. Damguard and B. Schoenmakers. “Proofs of partial knowledge and simplified design of witness hiding protocols”. In: *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology. LNCS, vol.839*. 1994, pp. 174–187.