ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ

Διπλωματική Εργασία

Συγκριτική μελέτη και αξιολόγηση συστημάτων κατανεμημένης ομοιοτυπίας με χρήση μηχανισμών ασφαλείας υλικού

Γιώργος Μύταρος

Ιωάννινα, Σεπτέμβριος 2018

Εξεταστική Επιτροπή:

- **Κωνσταντίνος Μαγκούτης**, Επίκουρος Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Συνεπιβλέπων)
- **Ευαγγελία Πιτουρά**, Καθηγήτρια, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Συνεπιβλέπων)
- **Όνομα Επώνυμο**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Στην εργασία αυτή ξεκινήσαμε στοχεύοντας να ερευνήσουμε τα οφέλη που μπορούν να προσφέρουν οι μηχανισμοί ασφαλείας υλικού στα συστήματα blockchain. Ένα blockchain είναι μια ανοιχτή βάση δεδομένων που διατηρεί ένα κατανεμημένο λογιστικό κατάλογο που συνήθως αναπτύσσεται μέσα σε ένα δίκτυο peer-to-peer. Αποτελείτε από μια συνεχώς αυξανόμενη λίστα εγγραφών που ονομάζονται block, τα οποία περιέχουν συναλλαγές. Τα block προστατεύονται από παραβιάσεις με την χρήση κρυπτογραφικών hashes και με μηχανισμό συμφωνίας. Η τεχνολογία bockchain μπορεί να μειώσει δραστικά το κόστος της εμπιστοσύνης μέσω μιας αποκεντρικοποιημένης προσέγγισης της λογιστικής και, κατ ’επέκταση, να δημιουργήσει έναν νέο τρόπο δομής των οικονομικών οργανισμών.

Αυτό που κάνει ένα blockchain ένα ιδιαίτερο είδος καταλόγου είναι ότι αντί να διαχειρίζεται ένα ενιαίο κεντρικό ίδρυμα, όπως μια τράπεζα ή κυβερνητική υπηρεσία, αποθηκεύεται σε πολλαπλά αντίγραφα σε πολλούς ανεξάρτητους υπολογιστές μέσα σε ένα αποκεντρικοποιημένο δίκτυο. Καμία μεμονωμένη οντότητα δεν ελέγχει το κατάλογο. Οποιοσδήποτε από τους υπολογιστές του δικτύου μπορεί να κάνει μια αλλαγή στο κατάλογο, αλλά μόνο ακολουθώντας κανόνες που υπαγορεύονται από ένα ”πρωτόκολλο συμφωνίας”, έναν μαθηματικό αλγόριθμο που απαιτεί από την πλειοψηφία των άλλων υπολογιστών του δικτύου να συμφωνούν με την αλλαγή.

Τα blockchains χωρίζονται σε permissionless και permissioned. Τα permissionless blockchains επιτρέπουν στο κατάλογο να συντηρείται εντελώς ανώνυμα. Ο κάθε peer μπορεί να κρατήσει ένα αντίγραφο του καταλόγου και να δημιουργήσει νέα blocks στον κατάλογο. Για παράδειγμα το Bitcoin για την επιλογή του επόμενου block που θα προστεθεί στον κατάλογο χρησιμοποιεί τον αλγόριθμο Proof-of-Work (PoW). Η βασική ιδέα πίσω από τον αλγόριθμο του PoW είναι να περιοριστεί ο ρυθμός των νέων block με την επίλυση ενός κρυπτογραφικού παζλ, δηλ. να εκτελεστεί ένας δύσκολος υπολογισμός για τον επεξεργαστή που χρειάζεται χρόνο για την επίλυση, αλλά μπορεί να επαληθευτεί γρήγορα.Το αντίκτυπο σε αυτό είναι ότι η υπολογιστική προσπάθεια που συνδέεται με τον αλγόριθμο συμφωνίας (π.χ Proof of Work στο Bitcoin) στα permissionless blockchains είναι τόσο ενεργοβόρα όσο και χρονοβόρα. Από την άλλη τα permissioned blockchains έχουν εξελιχθεί ως μια εναλλακτική των permissionless blockchains (όπου οποιοσδήποτε μπορεί να συμμετάσχει, π.χ Bitcoin, Ethereum), για να αντιμετωπίσουν την ανάγκη της λειτουργίας blockchain τεχνολογιών μεταξύ ενός συνόλου από γνωστούς και αναγνωρίσιμους κόμβους που πρέπει να υπάρχουν μέσα στο blockchain δίκτυο. Αυτό έχει εφαρμογή κυρίως σε εφαρμογές για επιχειρήσεις , όπου οι συμμετέχοντες απαιτούν κάποια μέσα αναγνώρισης μεταξύ τους, χωρίς να υποχρεώνονται να εμπιστεύονται πλήρως ο ένας τον άλλο. Τα permissioned blockchains βασίζονται κυρίως σε ασύγχρονα Byzantine fault tolerance

(BFT) πρωτόκολλα συμφωνίας. Αυτά τα πρωτόκολλα είναι γνωστά για την παραδοχής τους, που ορίζει ότι για την επίτευξη συμφωνίας χρειάζονται τουλάχιστον $3f + 1$ κόμβοι για μέχρι και f ελαττωματικούς κόμβους.

Οι μηχανισμοί ασφαλείας υλικού παρέχουν ένα ασφαλές και απομονωμένο περιβάλλον επεξεργασίας. Στον χώρο αυτό υπάρχουν πολλές προτάσεις από διάφορες εταιρείες, όπως το AMD Secure Technology, το ARM TrustZone και το Intel Software Guard Extensions (SGX). Ο μηχανισμός ασφαλείας υλικού με τον οποίο ασχοληθήκαμε σε αυτήν την εργασία είναι το Intel Software Guard Extensions (SGX). Το Intel SGX είναι η πιο εξέχουσα τεχνολογία Προστατευόμενου Περιβάλλοντος Εκτέλεσης σήμερα και υπάρχει σε κάθε βασικό επεξεργαστή της Intel από τη γενιά Skylake και μετά. Δημιουργεί περιβάλλοντα προστατευόμενης εκτέλεσης στη CPU που ονομάζονται enclaves. Τα enclaves απομονώνουν δεδομένα και προγράμματα από το λειτουργικό σύστημα που τρέχει και εξασφαλίζουν ότι τα εξαγόμενα δεδομένα είναι σωστά. Ένα enclave μπορεί να εκτελέσει ένα μικρό κομμάτι κώδικα μιας εφαρμογής ή ακόμη και μια ολόκληρη εφαρμογή. Χρησιμοποιώντας ένα μηχανισμό ασφαλείας υλικού όπως το Intel SGX αρκεί για να πετύχουμε τις ιδιότητες που θέλουμε στον κόμβο blockchain και να συμμετέχει στα πρωτόκολλα. Τα συστήματα που βασίζονται στο κώδικα που τρέχει στο enclave κινδυνεύουν μόνο από την κατάρρευση του κόμβου ολόκληρου και όχι από παραβίαση του κώδικα που τρέχει στο enclave.

Η εργασία αυτή εστίασε στα BFT πρωτόκολλα που όπως αναφέρθηκε παραπάνω τα περισσότερα permissioned blockchains χρησιμοποιούν για μηχανισμό συμφωνίας. Για παράδειγμα, το Hyperledger της εταιρείας IBM [1] χρησιμοποιεί το Practical Byzantine Fault Tolerance (PBFT) [2] για μηχανισμό συμφωνίας. Παρόλο που το PBFT μπορεί να επιτύχει μεγαλύτερη απόδοση από το μηχανισμό του Bitcoin ακόμη δεν μπορεί να φτάσει τις αποδόσεις των υπαρχόντων μεθόδων συναλλαγής (π.χ η Visa διαχειρίζεται δεκάδες χιλιάδες συναλλαγές το δευτερόλεπτο). Επιπλέον το PBFT μπορεί να επεκταθεί μόνο σε μερικές δεκάδες κόμβων αφού ανταλλάσει $O(n^2)$ μηνύματα για την επίτευξη συμφωνίας για μία λειτουργία μεταξύ n διακομιστών. Έτσι, η βελτίωση της επεκτασιμότητας και της απόδοσης των πρωτοκόλλων BFT είναι απαραίτητη για την εξασφάλιση της καθιέρωσης τους σε υπάρχουσες λύσεις blockchain.

Σε αυτή την εργασία, βασιστήκαμε σε μία υπάρχουσα υλοποίηση του BFT, στον αλγόριθμο MinBFT [3]. Είναι μεταγενέστερο του PBFT και βελτιώνει στο ότι χρειάζεται λιγότερους διακομιστές καθώς και ότι ανταλλάσει λιγότερα μηνύματα. Το MinBFT είναι ένας αλγόριθμος βυζαντινής συμφωνίας ο οποίος απαιτεί $2f + 1$ διακομιστές για f ελαττωματικούς (Βυζαντινούς) διακομιστές. Οι BFT αλγόριθμοι τυπικά χρειάζονται $3f + 1$ διακομιστές όμως με την βοήθεια ενός μηχανισμού ασφαλείας που υπάρχει σε κάθε υπολογιστή που εκτελεί το MinBFT ο αριθμός των διακομιστών μπορεί να πέσει απο $3f + 1$ σε $2f + 1$. Αυτός ο μηχανισμός ασφαλείας πρέπει να λειτουργεί σωστά ακόμα και αν όλο το σύστημα έχει παραβιαστεί. Το βασικό πρόβλημα είναι ότι όλοι οι διακομιστές θα πρέπει να εκτελέσουν την ίδια ακολουθία λειτουργιών. Ο μηχανισμός ασφαλείας θα πρέπει να παρέχει μια υπηρεσία η οποία θα ορίζει την ακολουθία των λειτουργιών που εκτελούν, με τέτοιο τρόπο όπου ένας κακόβουλος διακομιστής δεν θα μπορεί να κάνει τους άλλους σωστούς διακομιστές να εκτελέσουν μια άλλη λειτουργία αντί για την κ-οστη λειτουργία. Αυτό μπορεί να υλοποιηθεί με την βοήθεια ενός έμπιστου μονοτονικού μετρητή που θα αντιστοιχίζει ακολουθιακούς αριθμούς σε κάθε λειτουργία. Στην αρχική έκδοση του MinBFT [3] είχε χρησιμοποιηθεί ο μηχανισμός ασφαλείας Atmel TPM 1.2 για την υλοποίηση της υπηρεσίας USIG (βλ. ενότητα 2.7) η οποία είναι υπεύθυνη για την δημιουργία του μονοτονικού μετρητή.

Η ενασχόληση με τα blockchains μας οδήγησε στην μελέτη του BFT και η εργασία στόχευσε για λόγους χρονικούς στην χρήση των μηχανισμών ασφαλείας στα πρωτόκολλα BFT. Σε αυτήν την εργασία αλλάξαμε τον μηχανισμό ασφαλείας που είχε χρησιμοποιηθεί στην αρχική έκδοση του MinBFT και τον υλοποιήσαμε με την χρήση του Intel SGX. Τα αποτελέσματα αυτής της αλλαγής τα παρουσιάζουμε στην ενότητα 4 όπου μέσα από κάποια benchmarks εξάγαμε συμπεράσματα και όπου μπορούσαμε τα συγκρίναμε με την αρχική έκδοση του MinBFT. Παρακάτω στην ενότητα 1.2 περιγράφεται η δομή της εργασίας εξηγώντας τι θα βρει ο αναγνώστης σε κάθε ένα από τα παρακάτω κεφάλαια.

1.1 Στόχοι

Η παρούσα διπλωματική εργασία είναι σχετική με το Προστατευόμενο Περιβάλλον Εκτέλεσης Intel SGX και έχει ως στόχο να αναδείξει τα οφέλη που μπορεί να παράσχει στους αλγορίθμους Byzantine Fault Tolerance καθώς και στις τεχνολογίες Blockchain που πρόσφατα υπάρχει μεγάλο ενδιαφέρον γύρω από αυτές.

1.2 Δομή της Διπλωματικής Εργασίας

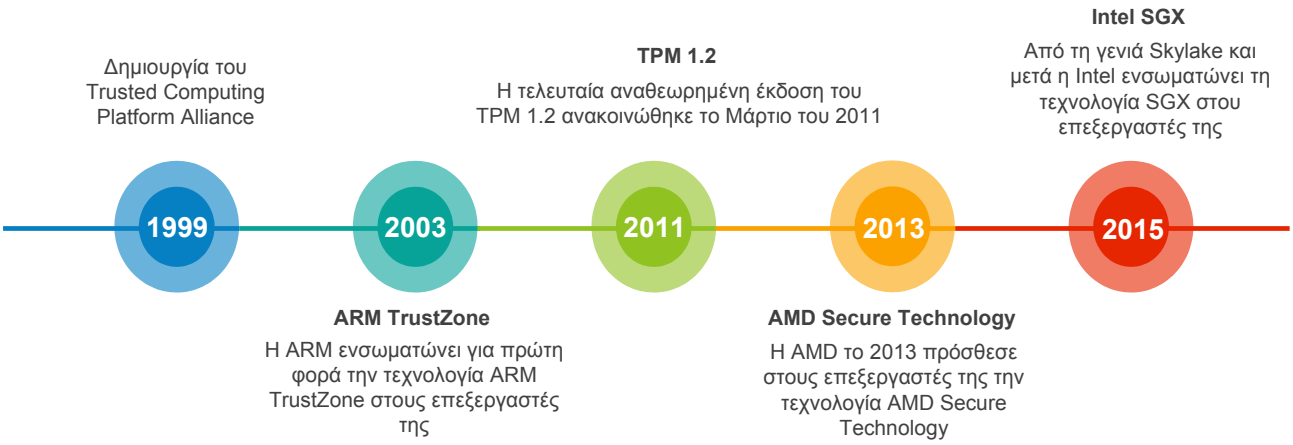
Η διπλωματική περιέχει 5 κεφάλαια. Στο πρώτο κεφάλαιο γίνεται εισάγουμε τον στόχο της διπλωματικής εργασίας. Στο δεύτερο κεφάλαιο αναφέρεται ο,τι είναι απαραίτητο να γνωρίζει ο αναγνώστης προτού εμβαθύνει στην ανάγνωση της εν λόγω διπλωματικής. Η διαδικασία της υλοποίησης του USIG με χρήση του Intel SGX καθώς και των προγραμμάτων για τον έλεγχο αυτής της υπηρεσίας περιλαμβάνεται στο τρίτο κεφάλαιο. Στο τέταρτο κεφάλαιο γίνεται αναφορά στην αξιολόγηση της υπηρεσίας, ενώ τέλος στο πέμπτο κεφάλαιο περιλαμβάνονται τα συμπεράσματα που προέκυψαν από το σύνολο της διπλωματικής εργασίας όπως και κάποιες σκέψεις σχετικά με μελλοντική δουλεία.

Κεφάλαιο 2

Υπόβαθρο

2.1 Προστατευόμενο Περιβάλλον Εκτέλεσης

Ιστορικά υπήρχε η ανάγκη να οριστεί το Trusted Computing Base, ένα ασφαλές περιβάλλον εκτέλεσης στους προσωπικούς μας υπολογιστές. Από το 1999 είχε δημιουργηθεί το Trusted Computing Platform Alliance μία ομάδα αποτελούμενη από την AMD, Hewlett-Packard, IBM, Intel και την Microsoft με στόχο να υλοποιήσουν Trusted Computing Πλατφόρμες σε όλους τους προσωπικούς υπολογιστές. Η βασική ιδέα του Trusted Computing είναι η προσθήκη ενός ασφαλούς hardware για την αντιστάθμιση ανασφαλούς λογισμικού. Να επιτρέπει στις εξωτερικές οντότητες να έχουν αυξημένο επίπεδο εμπιστοσύνης ότι το σύστημα θα λειτουργεί όπως αναμένεται/προσδιορίζεται.



Σχήμα 2.1: Ιστορική εξέλιξη των μηχανισμών ασφαλείας υλικού

2.2 Intel Software Guard Extensions - SGX

Ένα Προστατευόμενο Περιβάλλον Εκτέλεσης (Trusted Execution Environment, TEE) είναι ένα ασφαλές και απομονωμένο περιβάλλον επεξεργασίας. Μπορεί να γίνει με διάφορους τρόπους συμπεριλαμβανομένων ειδικές επεκτάσεις του επεξεργαστή που εγγυώνται απομόνωση. Τα κύρια χαρακτηριστικά των TEEs είναι η απομονωμένη εκτέλεση, απομακρυσμένη βεβαίωση, ασφαλή αποθήκευση και αξιόπιστη ροή εκτέλεσης. Οι εφαρμογές που τρέχουν στα TEEs είναι εμπιστευτικές με προστατευμένη την ακεραιότητα και είναι απροσπέλαστη από εξωτερικά μέρη. Η Intel το 2015 ανακοίνωσε την τεχνολογία Software Guard Extensions την οποία πρόσθεσε στην έκτη γενιά επεξεργαστών που βασίζεται στην μικροαρχιτεκτονική Intel Skylake. Το Intel Software Guard Extensions (SGX) είναι ένα σύνολο από αρχιτεκτονικές προσθήκες που εισήχθησαν για να λύσουν το πρόβλημα της ασφαλής απομακρυσμένης εκτέλεσης κώδικα. Διατυπώνει ένα νέο σύνολο από οδηγίες οι οποίες παρέχουν ένα προστατευόμενο περιβάλλον εκτέλεσης στον κώδικα του χρήστη [4].

2.2.1 Enclave

Αυτό που επιτρέπει στους επεξεργαστές που είναι εξοπλισμένοι με το SGX να παρέχουν αυτές τις ισχυρές εγγυήσεις βασίζεται στα δύο ακόλουθα hardware: Το Processor Reserved Memory (PRM) που είναι ένα συνεχές μπλοκ μνήμης που προορίζεται για το SGX, απρόσιτο από το μη αξιόπιστο λογισμικό και ακόμη και από το υλικό. Η λειτουργία Enclave είναι μια λειτουργία υπό την οποία ένας επεξεργαστής αποκτά πρόσβαση στο PRM.

Χρησιμοποιώντας αυτό το hardware, το SGX ορίζει την έννοια ενός enclave. Ένα enclave είναι μια ενότητα λογισμικού απομονωμένη από το υπόλοιπο σύστημα. Η μνήμη της βρίσκεται αποκλειστικά στην PRM, εμποδίζοντας την πρόσβαση από άλλες διαδικασίες και enclaves που λειτουργούν στο σύστημα. Το PRM προστατεύεται από τις μη εγκλωβισμένες διεργασίες με τη λειτουργία εγκλωβισμού (enclave).Όταν μια διεργασία δεν βρίσκεται σε enclave mode και προσπαθεί να έχει πρόσβαση στην PRM, η πρόσβαση αυτή δεν επιτρέπεται. Το PRM μιας διεργασίας enclave προστατεύεται από τις προσβάσεις από άλλες διεργασίες σε λειτουργία enclave από τη δομή ελέγχου SGX Enclave Control Structure (SECS).

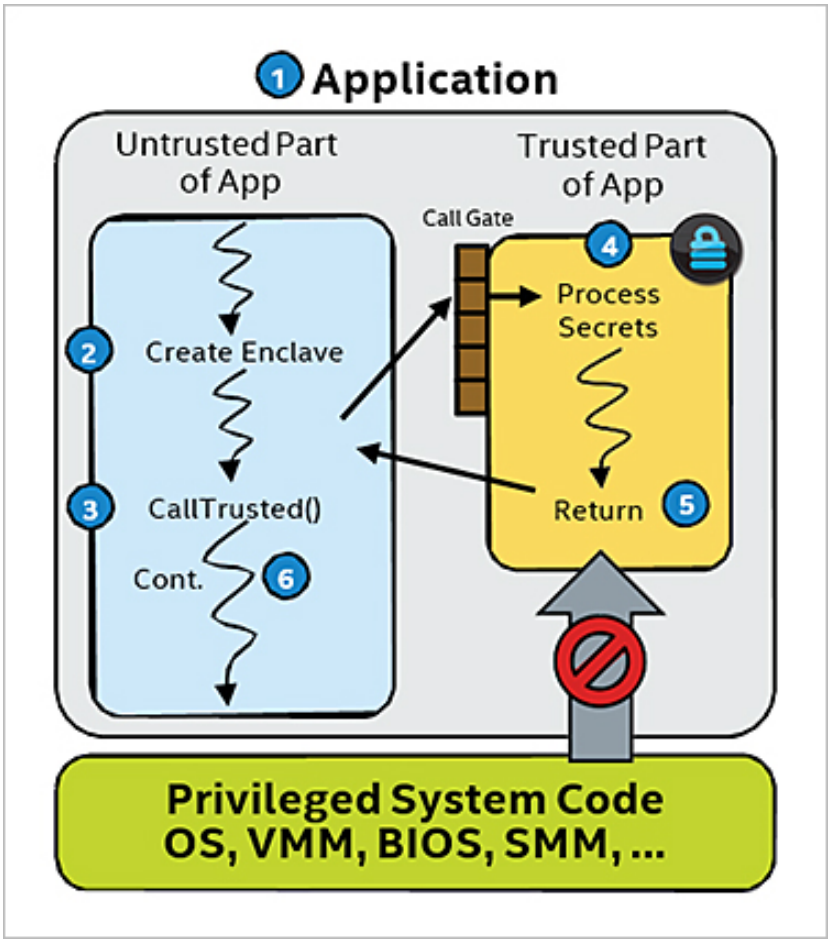
Παρόλο που το SGX παρέχει ισχυρές εγγυήσεις μέσω της έννοιας του enclave, δεν εγγυάται την ορθότητα του λογισμικού και δεν προστατεύει από εσφαλμένο λογισμικό. Αντίθετα, το SGX ενθαρρύνει τους προγραμματιστές να απομονώσουν ένα μικρό κομμάτι του λογισμικού τους, το Trusted Computing Base (TCB), σε ένα περιβάλλον αξιόπιστου enclave και να διατηρήσουν τα υπόλοιπα ως παραδοσιακές διαδικασίες συστήματος. Με την ελαχιστοποίηση του μεγέθους του TCB και επομένως το μέγεθος του κώδικα που πρέπει να εμπιστευτεί κανείς, οι κοινές αρχές ασφάλειας υποδηλώνουν ότι η πιθανότητα ελαττωμάτων ασφαλείας μειώνεται.

Προκειμένου να είναι χρήσιμη η απομόνωση που παρέχει ένα enclave, η επικοινωνία μεταξύ του αξιόπιστου και του μη αξιόπιστου λογισμικού ενεργοποιείται μέσω των enclave calls (ECALL) και των out calls (OCALL). Αυτή η διεπαφή πρέπει να οριστεί κατά την μεταγλώττιση, καθορίζοντας ένα API των ECALL για το enclave καθώς και όλες τις μη αξιόπιστες υπηρεσίες που χρειάζονται ως OCALLs σε ένα αρχείο EDL.

Όταν κατασκευαστεί, το enclave είναι ένα απλό δυαδικό αρχείο στο μη αξιόπιστο σύστημα αρχείων. Καθώς ένα enclave υπό τέτοιες συνθήκες θα ήταν ευάλωτο σε παραβίαση

πριν από την αρχικοποίηση, το SGX επιβάλλει μια αυστηρή πολιτική υπογραφής. Ένα enclave πρέπει να περιλαμβάνει μια υπογραφή Enclave (Enclave Signature) που περιέχει (α) ένα hash του κώδικα και τα δεδομένα αρχικοποίησης του enclave, (β) το δημόσιο κλειδί του δημιουργού και (c) μια έκδοση enclave. Κατά την αρχικοποίηση του enclave πραγματοποιείται έλεγχος υλικού, διασφαλίζοντας ότι το Enclave Signature ταιριάζει με το δυαδικό enclave που έχει φορτωθεί από το σύστημα αρχείων.

Ένα άλλο ισχυρό εργαλείο ενός SGX enclave είναι η δυνατότητα ανάγνωσης και εγγραφής δεδομένων σε ένα μη αξιόπιστο μέσο αποθήκευσης, ενώ παράλληλα διασφαλίζεται η εμπιστευτικότητα των περιεχομένων του. Τέτοιες δυνατότητες χρειάζονται καθώς ο PRM είναι ευμετάβλητος και δεν θα παραμείνει μετά το κλείσιμο. Στο SGX αυτή η διαδικασία είναι γνωστή ως σφράγιση. Η σφράγιση επιτρέπει κρυπτογράφηση και αποκρυπτογράφηση χρησιμοποιώντας κλειδί σφραγίδας (Seal Key), το οποίο είναι εμπιστευτικό για την υπογραφή Enclave Signature.



Σχήμα 2.2: Τα βήματα που ακολουθούνται στο enclave κατά τη διάρκεια εκτέλεσης. Πηγή: software.intel.com

1. Η εφαρμογή δημιουργείται με trusted και untrusted μέρη
2. Η εφαρμογή εκτελείται και δημιουργεί το enclave, το οποίο είναι σε ασφαλή απομονωμένη μνήμη
3. Καλείται η trusted μέθοδος και η εκτέλεση μεταφέρεται στο enclave

4. Το enclave βλέπει όλα τα δεδομένα τη διεργασίας, η πρόσβαση από έξω στα δεδομένα του enclave απαγορεύεται.
5. Η trusted function επιστρέφει τα δεδομένα του enclave
6. Η εφαρμογή συνεχίζει την κανονική της εκτέλεση

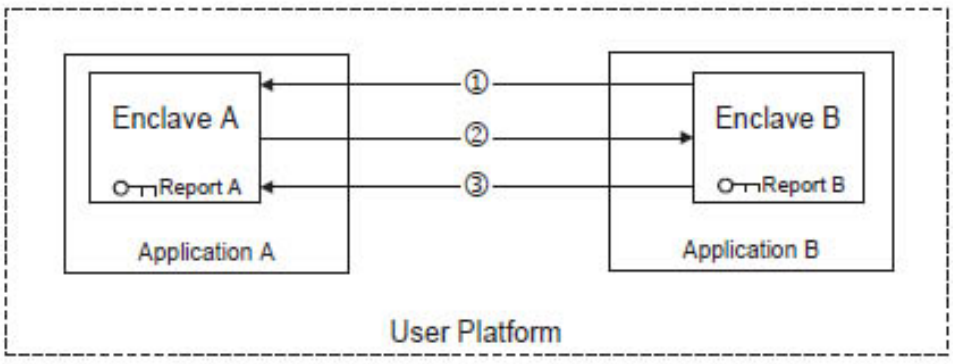
2.2.2 Attestation

Η βεβαίωση (Attestation), μέσα στην πλατφόρμα Intel SGX, είναι η δράση επαλήθευσης της ύπαρξης συγκεκριμένων enclave. Η εφαρμογή του Attestation είναι διπλή, στην περίπτωση που υπάρχουν πολλαπλά enclaves που εκτελούνται τοπικά στην ίδια CPU και στην περίπτωση που τα enclave πρέπει να επικοινωνούν με enclaves που εκτελούνται σε εξωτερικούς επεξεργαστές.

Local attestation

Η τοπική βεβαίωση (Local attestation) βασίζεται σε ένα μυστικό αναγνωριστικό της CPU. Ένα enclave μπορεί να χρησιμοποιήσει μια εντολή SGX για να δημιουργήσει μια αναφορά που προσδιορίζει μοναδικά το enclave. Αυτή η αναφορά είναι ένα MAC tag που περιέχει ένα κλειδί που προέρχεται από το συγχωνευμένο μυστικό αναγνωριστικό της CPU και το αναγνωριστικό του enclave. Δεν είναι δυνατό να πλαστογραφηθεί μια τέτοια αναφορά MACed, καθώς το MACing συμβαίνει σε στοιχεία του υλικού που εξασφαλίζουν ότι η αναφορά αντιστοιχεί στο καλούντος enclave. Το συγκεκαλυμμένο μυστικό δεν είναι άμεσα προσβάσιμο από το software, αλλά μπορεί να χρησιμοποιηθεί μόνο από συγκεκριμένες κλήσεις hardware που προστατεύουν από κακόβουλη χρήση. Ένας ελεγκτής της βεβαίωσης θα ζητήσει μια αναφορά MACed από τον enclave-πελάτη και μετά την παραλαβή του θα πάρει το ίδιο κλειδί για να επαληθεύσει το MAC. Επειδή μια αναφορά μπορεί να δημιουργηθεί μόνο από το enclave που περιγράφει, ο ελεγκτής μπορεί να είναι σίγουρος για ποιο enclave εκτελεί ο πελάτης αν είναι έγκυρο το MAC.

Η τοπική βεβαίωση είναι χρήσιμη όταν οι εφαρμογές έχουν πάνω από ένα enclave με το οποίο πρέπει να δουλέψουν για να ολοκληρώσουν μια εργασία ή όταν δύο διαφορετικές εφαρμογές πρέπει να ανταλλάξουν δεδομένα μεταξύ των enclaves. Κάθε enclave πρέπει να πιστοποιήσει το άλλο για να μπορέσει να επιβεβαιώσει ότι είναι αξιόπιστο.



Σχήμα 2.3: Διαδικασία τοπικής βεβαίωσης (Local attestation). Πηγή: software.intel.com

Το σχήμα 2.3 δείχνει ένα παράδειγμα ροής για το πως δύο enclave στην ίδια πλατφόρμα πιστοποιούν το ένα το άλλο.

1. Η εφαρμογή A έχει το enclave A και η εφαρμογή B έχει το enclave B. Αφού πρώτα η κάθε εφαρμογή δημιούργησε μια διαδρομή επικοινωνίας με το enclave της, το enclave B στέλνει την ταυτότητα (MRENCLAVE) του στο enclave A.

2. Το enclave A ζητάει από το hardware να δημιουργήσει μία αναφορά προορισμένη για το enclave B χρησιμοποιώντας το MRENCLAVE που έλαβε από το enclave B. Το enclave A στέλνει την αναφορά του στο enclave B μέσω της untrusted εφαρμογής. Σαν μέρος της αναφοράς, το enclave A μπορεί επίσης να στείλει δεδομένα στον B.

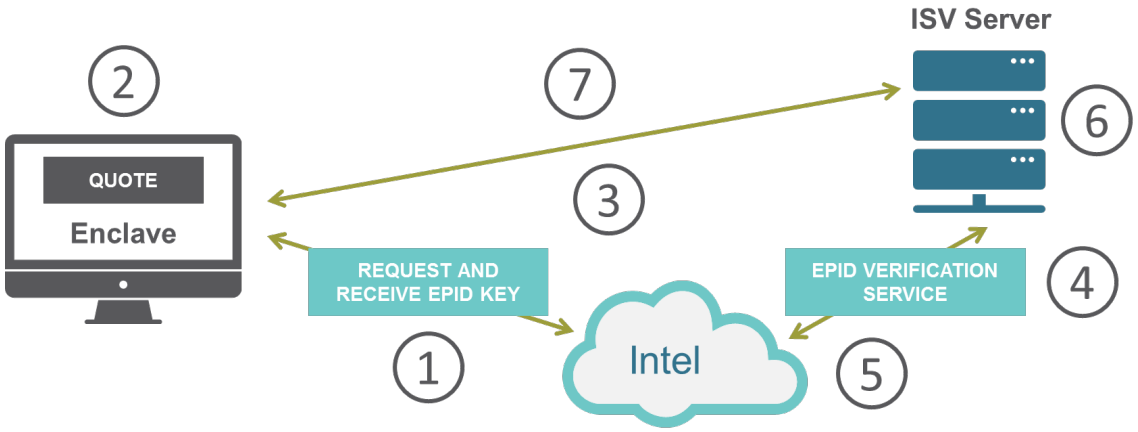
3. Μόλις λάβει το enclave B την αναφορά απο το enclave A, το enclave B ζητάει από το hardware να πιστοποιήσει την αναφορά για να επιβεβαιώσει το enclave B ότι το enclave A είναι στην ίδια πλατφόρμα με αυτό. Το enclave B μπορεί τώρα να απαντήσει χρησιμοποιώντας την αναφορά του enclave A, χρησιμοποιώντας την MRENCLAVE τιμή από την αναφορά που μόλις έλαβε. Το enclave B στέλνει την αναφορά του στο enclave A.

4. Το enclave A τότε πιστοποιεί την αναφορά για να επιβεβαιωθεί ότι το enclave B ανήκει στην ίδια πλατφόρμα με το enclave A.

Remote attestation

Οι απομακρυσμένες βεβαιώσεις (Remote attestation) βασίζονται στην ίδια έννοια μιας αναφοράς, αλλά ο ελεγκτής και ο πελάτης βρίσκονται τώρα σε διαφορετικές CPU, δεν έχουν πλέον το ίδιο μυστικό αναγνωριστικό της CPU. Αντ' αυτού, η απομακρυσμένη βεβαίωση βασίζεται στο Enhanced Privacy ID (EPID). Κάθε CPU που είναι εξοπλισμένη με SGX έχει λάβει ένα ιδιωτικό κλειδί μέλους EPID. Όπως το μυστικό αναγνωριστικό της CPU, έτσι και αυτό το ιδιωτικό κλειδί δεν είναι άμεσα προσβάσιμο σε ένα enclave. Στο πλαίσιο του καθεστώτος EPID, όλοι οι εκδότες που δημιούργησαν ιδιωτικά κλειδιά μοιράζονται το ίδιο δημόσιο κλειδί (δημόσιο κλειδί ομάδας EPID). Όταν πραγματοποιείται απομακρυσμένη βεβαίωση, το enclave-πελάτης δημιουργεί μια αναφορά και βεβαιώνει τοπικά με το Quoting Enclave (QE). Το QE, πλέον πεπεισμένο για τον εντοπισμό του enclave πελάτη, απαλείφει το MAC από την αναφορά και αντ' αυτού υπογράφει με το ιδιωτικό κλειδί μέλους EPID, το οποίο το QE έχει ειδικά προνόμια πρόσβασης. Ο απομακρυσμένος ελεγκτής μπορεί να επαληθεύσει την υπογεγραμμένη αναφορά με το δημόσιο κλειδί της ομάδας EPID και να βεβαιωθεί ότι (α) η αναφορά έχει υπογραφεί από ένα EPID Member Private Key, (β) τα ιδιωτικά κλειδιά των EPID Member παρέχονται μυστικά στις QE και (γ) δεν θα υπογράψουν ψευδείς αναφορές.

Η απομακρυσμένη βεβαίωση είναι ο τρόπος με τον οποίο οι εφαρμογές Intel SGX παρουσιάζουν επαληθεύσιμες αποδείξεις για το μηχάνημα στο οποίο εκτελούνται σε απομακρυσμένο μέρος (όχι απαραίτητα της Intel). Επιτρέπει την ασφαλή μεταφορά μυστικών μεταξύ των υπολογιστών, όταν οι χρήστες από τις δύο πλευρές ενδέχεται να μην έχουν τον έλεγχο του άλλου.



Σχήμα 2.4: Διαδικασία απομακρυσμένης βεβαίωσης (Remote attestation)

1. Μετά την εγκατάσταση του λογισμικού Intel SGX Platform (PSW) στον υπολογιστή-πελάτη, το PSW ζητά από την υπηρεσία παροχής υπηρεσιών SGX της Intel να παράσχει το κλειδί EPID (Enhanced Privacy ID) του υπολογιστή. Αν η ανταλλαγή πρωτοκόλλων μεταξύ της υπηρεσίας Intel SGX PSW και της υπηρεσίας παροχής υπηρεσιών Intel είναι επιτυχής - κάτι που απαιτεί γνήσια CPU εξοπλισμένη με Intel SGX, τότε η συσκευή ή ο διακομιστής θα έχει το κλειδί EPID.

2. Το enclave παράγει ένα quote. Ένα quote είναι απλώς μια υπογεγραμμένη αναφορά των τρεχουσών τιμών του Platform Configuration Registers (PCRs) στον υπολογιστή. Ο επεξεργαστής του υπολογιστή που τρέχει το enclave δημιουργεί αυτομάτως αυτό το quote.

3. Ο υπολογιστής που τρέχει το enclave στέλνει τότε το quote στον ανεξάρτητο προμηθευτή λογισμικού (Independent software vendor, ISV) που έχει ένα μυστικό που χρειάζεται η εφαρμογή του enclave.

4. Ο ISV με τη σειρά του στέλνει το quote στην Intel για επαλήθευση.

5. Η Intel επαληθεύει το quote για τον ISV. Ωστόσο, η Intel δεν παρέχει άλλες πληροφορίες (όπως ποιος δημιούργησε το quote). Στην περίπτωση αυτή, η Intel είναι σαν συμβολαιογράφος: το μόνο που μπορεί να πει είναι ότι το κρυπτογραφικά υπογεγραμμένο quote προέρχεται από έγκυρο enclave, αλλά τίποτα περισσότερο.

6. Στη συνέχεια, ο ISV είναι υπεύθυνος για την επιθεώρηση του quote. Για παράδειγμα, το ISV μπορεί να επιβεβαιώσει ότι το κλειδί είναι δικό του.

7. Αφού ο ISV ικανοποιηθεί με το quote, ο διακομιστής του ISV και το enclave μπορούν τώρα να παράξουν ένα κλειδί κρυπτογράφησης από το quote. Αυτό είναι ένα συμμετρικό κλειδί μεταξύ του διακομιστή και του enclave που χρησιμοποιεί τον αλγόριθμο Diffie-Hellman.

Ένα σημαντικό πράγμα που πρέπει να σημειωθεί είναι ότι το συμμετρικό κλειδί που τελικά δημιουργήθηκε από τον εξυπηρετητή ISV και το εν λόγω enclave είναι μοναδικό για εκείνο το enclave. Έτσι, ακόμη και μια επίθεση man-in-the-middle θα μπορούσε να δει μόνο την κρυπτογραφημένη ανταλλαγή 128-bit (από τα 256-bit) μεταξύ του διακομιστή και του enclave.

2.2.3 Monotonic counters

Μία από τις λειτουργίες που παρέχει η Intel SGX, η οποία είναι ιδιαίτερα σημαντική για αυτήν την εργασία, είναι η πρόσβαση σε αξιόπιστους μονοτονικούς μετρητές. Όπως υποδεικνύεται από το όνομα, οι μονοτονικοί μετρητές είναι ακέραιοι μετρητές, οι οποίοι μπορούν μόνο να αυξηθούν.

Η δημιουργία ενός μονοτονικού μετρητή (MC) συνεπάγεται στην εγγραφή στην μη πτητική μνήμη (Non-volatile memory) που είναι διαθέσιμη στην πλατφόρμα από το Intel Management Engine (ME) και είναι προσβάσιμο μόνο μέσω των κλήσεων SGX. Οι επαναλαμβανόμενες λειτουργίες εγγραφής θα μπορούσαν να προκαλέσουν υπερφόρτωση στη μνήμη κατά τη διάρκεια του κανονικής λειτουργίας της πλατφόρμας. Το Intel SGX το αποτρέπει αυτό περιορίζοντας το ρυθμό με τον οποίο μπορούν να εκτελεστούν οι λειτουργίες MC. Αν γίνει υπέρβαση του ορίου, η λειτουργία MC μπορεί να επιστρέψει μήνυμα μη διαθεσιμότητας της υπηρεσίας. Ο μέγιστος επιτρεπόμενος αριθμός από μονοτονικούς μετρητές σε ένα enclave είναι 256.

Το Intel SGX παρέχει τέσσερις κλήσεις σχετικές με τον μονοτονικό μετρητή:

- Η κλήση $sgx\_create\_monotonic\_counter$ δημιουργεί έναν μονοτονικό μετρητή με την προεπιλεγμένη πολιτική κατόχου 0x1, που σημαίνει ότι τα enclaves με το ίδιο κλειδί υπογραφής έχουν πρόσβαση στον μονοτονικό μετρητή.
- Η κλήση $sgx\_increment\_monotonic\_counter$ αυξάνει την τιμή ενός μετρητή κατά 1.
- Η κλήση $sgx\_read\_monotonic\_counter$ επιστρέφει την τιμή του μετρητή.
- Η κλήση $sgx\_destroy\_monotonic\_counter$ καταστρέφει τον μετρητή.

2.3 Blockchain

Το κίνητρο για αυτή την διπλωματική εργασία ήταν οι τεχνολογίες blockchain που πρόσφατα υπάρχει μεγάλο ενδιαφέρον γύρω απο αυτές. Τα blockchains κατέχουν το κλειδί για απεριόριστες τεχνολογικές δυνατότητες. Ο καθένας από τους επιχειρηματίες στις μεγάλες εταιρείες και ακόμη και οι κυβερνήσεις αναπτύσσουν χρήσιμες εφαρμογές που βασίζονται σε blockchain. Με τα blockchains να βρίσκουν περισσότερες εφαρμογές σε ποικίλους τομείς, οι προσδοκίες είναι υψηλές. Από το 2014, έχει καταστεί φανερό ότι τα blockchains μπορούν να εφαρμοστούν σε κάτι περισσότερο από τα οικονομικά και τις εφαρμογές fintech. Το blockchain Ethereum και η εισαγωγή των smart contracts άνοιξαν νέες βλέψεις δυνατοτήτων. Πιο συγκεκριμένα στα πρώτα στάδια της διπλωματικής μου εργασίας ανέλυσα το blockchain Hyperledger Sawtooth το οποίο χρησιμοποιεί έναν μοναδικό μηχανισμό για την επίτευξη συμφωνίας σχετικά με την εγκυρότητα του καταλόγου που βασίζεται σε έμπιστο κώδικα που τρέχει μέσα σε ένα θωρακισμένο λογισμικό Intel Guard Guard Extensions (SGX).

2.3.1 Τι είναι Blockchain

Ένα blockchain είναι μια ανοιχτή βάση δεδομένων που διατηρεί ένα κατανεμημένο λογιστικό κατάλογο που συνήθως αναπτύσσεται μέσα σε ένα δίκτυο peer-to-peer. Αποτελείτε από μια συνεχώς αυξανόμενη λίστα εγγραφών που ονομάζονται block, τα οποία περιέχουν συναλλαγές. Τα block προστατεύονται από παραβιάσεις με την χρήση κρυπτογραφικών hashes και με μηχανισμό συμφωνίας.

Η δομή ενός blockchain αποτελείται από μια σειρά από blocks όπου το κάθε ένα περιέχει το κρυπτογραφικό hash του προηγούμενου του block στην αλυσίδα. Επιπλέον, ο μηχανισμός συμφωνίας χρησιμοποιείται για (1) την πρόληψη της τροποποίησης ολόκληρου του blockchain και (2) να αποφασίσει ποιο block θα προστεθεί στο λογιστικό κατάλογο.

Στο blockchain υπάρχουν δύο ειδών μοντέλα τα permissionless ή permissioned. Οι permissionless κατάλογοι διατηρούνται σε δίκτυα peer-to-peer με αποκεντρωμένο και ανώνυμο τρόπο. Για παράδειγμα στο blockchain που χρησιμοποιεί το Bitcoin προκειμένου να προσδιοριστεί ποιο block θα προστεθεί στη συνέχεια στον κατάλογο, οι peers θα πρέπει να εκτελέσουν τον αλγόριθμο συμφωνίας Proof-of-Work (PoW). Η βασική ιδέα πίσω από τη συμφωνία του PoW είναι να περιοριστεί ο ρυθμός των νέων block με την επίλυση ενός κρυπτογραφικού παζλ, δηλ. να εκτελεστεί ένας δύσκολος υπολογισμός για τον επεξεργαστή που χρειάζεται χρόνο για την επίλυση, αλλά μπορεί να επαληθευτεί γρήγορα. Αυτό επιτυγχάνεται υποχρεώνοντας τους peers να βρουν ένα nonce N τέτοιο ώστε με δεδομένο ένα Block B και ένα όριο L, ο κρυπτογραφικός κατακερματισμός του B||N να είναι χαμηλότερος από L. Ο πρώτος peer που βρει μια τέτοια λύση προσθέτει το block στον κατάλογο. Σε γενικές γραμμές, όσο ο αντίπαλος ελέγχει λιγότερο από το ήμισυ της συνολικής υπολογιστικής ισχύος που υπάρχει στο δίκτυο, ο αλγόριθμος συμφωνίας PoW εμποδίζει τον αντίπαλο να δημιουργήσει νέα block ταχύτερα από τους έντιμους συμμετέχοντες.

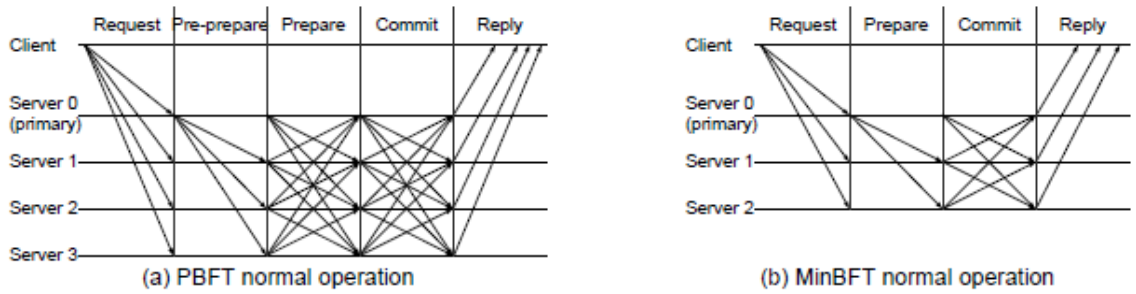
Τα permissionless blockchains έχουν το πλεονέκτημα να επιτρέπουν στο κατάλογο να συντηρείται εντελώς ανώνυμα. Ο κάθε peer μπορεί να κρατήσει ένα αντίγραφο του καταλόγου και να δημιουργήσει νέα blocks στον κατάλογο. Από την άλλη πλευρά, η υπολογιστική προσπάθεια που συνδέεται με τον αλγόριθμο συμφωνίας του PoW είναι τόσο ενεργοβόρα όσο και χρονοβόρα. Ακόμη και αν χρησιμοποιείται εξειδικευμένο υλικό για να βρεθεί μια PoW, αυτή η διαδικασία εξακολουθεί να βάζει ένα όριο στην καθυστέρηση της συναλλαγής.

Αντίθετα, τα permissioned blockchains απαιτούν ένα σύνολο αξιόπιστων κόμβων που θα συμβάλουν με τη δημιουργία νέων block και θα εκτελούν ένα πρωτόκολλο βυζαντινής συμφωνίας (BFT) για να αποφασίσουν τη σειρά με την οποία τα μπλοκ εισάγονται στο κατάλογο. Ως εκ τούτου, τα permissioned blockchains δεν καταναλώνουν το ποσό των πόρων που τα permissionless blockchains καταναλώνουν και είναι σε θέση να επιτύχουν καλύτερη καθυστέρηση συναλλαγής και απόδοσης. Επιπλέον, γίνεται δυνατός ο έλεγχος όλων των συμμετεχόντων που είναι υπεύθυνοι για τη συντήρηση του καταλόγου - καθιστώντας αυτό το είδος blockchain ελκυστικότερη λύση για μεγάλες εταιρείες, αφού μπορεί να διαχωριστεί από το dark web ή από παράνομες δραστηριότητες [5].

2.3.2 Το πρωτόκολλο BFT στην τεχνολογία Blockchain

Η αναπαραγωγή του blockchain πάσχει από βραχυπρόθεσμη ασυνέπεια. Ακόμα κι αν υποθέσουμε άπειρη εκτέλεση και είμαστε πρόθυμοι να υποθέσουμε ότι ένα μπλοκ που είναι θαμμένο από κρυπτογραφικά παζλ αξίας μιας ώρας είναι απολύτως ασφαλές, αυτό σημαίνει ότι πρέπει να περιμένουμε μια ώρα! Στην πραγματικότητα ένα από τα μεγαλύτερα πλεονεκτήματα των πρωτοκόλλων BFT είναι ότι παρέχουν ”άμεση οριστικότητα”.

Η αναπαραγωγή με BFT πρωτόκολλο φέρνει πολλά πλεονεκτήματα. Δεν υποφέρει από έλλειψη οριστικότητας. Αντίθετα, μόλις δεσμευτεί ένα μπλοκ, δεν μπορεί ποτέ να ανακληθεί. Επιπλέον, τα σύγχρονα συστήματα BFT SMR έχουν πολύ χαμηλή λανθάνουσα κατάσταση και υψηλή απόδοση. Συγκεκριμένα, αυτά τα συστήματα συχνά συντονίζονται για να παρέχουν υψηλές επιδόσεις σε κανονικές περιπτώσεις. Ακόμη και σε μια



Σχήμα 2.5: Ακολουθία μηνυμάτων των PBFT και MinBFT (Figure 1 από [3]).

εγκατάσταση WAN με δεκάδες servers, αυτά τα συστήματα κατά κανόνα καταφέρνουν να διαπράττουν εκατοντάδες (αν όχι χιλιάδες) λειτουργίες ανά δευτερόλεπτο σε καλές συνθήκες δικτύου και εκτελέσεις χωρίς αστοχία.

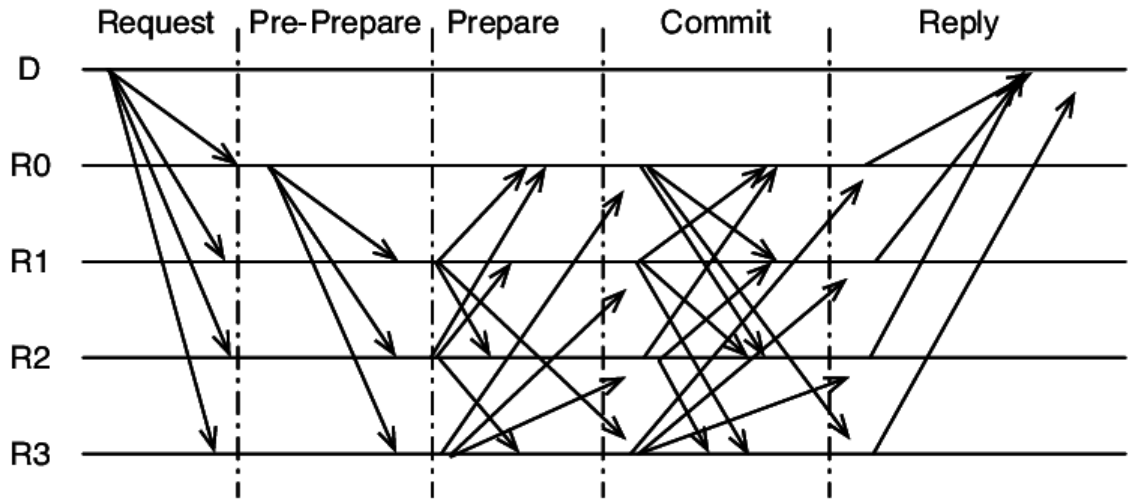
2.4 Πρωτόκολλο BFT - Byzantine fault tolerance

Υπενθυμίζουμε ότι ένα blockchain απαιτεί ένα μηχανισμό για την επίτευξη κατανεμημένης συμφωνίας ή την επικύρωση και την κανονικοποίηση ενός μόνο καταλόγου. Το πρωτόκολλο BFT αναφέρεται στο χαρακτηριστικό των κατανεμημένων συστημάτων που τους επιτρέπει να φτάσουν σε συμφωνία ενάντια στα βυζαντινά σφάλματα, δηλαδή σε καταστάσεις όπου τα συστατικά του συστήματος θα αποτύχουν - αλλά όχι μόνο να αποτύχουν - οι εσφαλμένοι βυζαντινοί κόμβοι θα ενεργήσουν αυθαίρετα και συχνά παρουσιάζουν αντικρουόμενες πληροφορίες σε διαφορετικούς κόμβους του συστήματος.

Οι αλγόριθμοι BFT τυπικά απαιτούν $3f + 1$ servers (ή replicas) για να ανεχθούν f βυζαντινούς (ή ελαττωματικούς) διακομιστές. Βασίζονται στην ιδέα ότι τα σωστά αντίγραφα μπορούν να ξεπεράσουν τα ελαττωματικά αντίγραφα με μια σειρά ψήφων και, για να συμβεί αυτό, θα χρειαστούν τουλάχιστον $2f + 1$ σωστά αντίγραφα, σε σύνολο εξυπηρετητών $3f + 1$. Ωστόσο, αυτοί οι αλγόριθμοι απαιτούν περισσότερους διακομιστές από αυτό το ελάχιστο.

2.5 Practical BFT - PBFT

Το Practical Byzantine Fault Tolerance (PBFT) είναι ένας αλγόριθμος BFT που δημοσιεύθηκε το 1999 στην εργασία ”Practical Byzantine Fault Tolerance” [2]. Στοχεύει να βελτιώσει τον αυθεντικό BFT μηχανισμό συμφωνίας και υλοποιήθηκε σε πολλά μοντέρνα κατανεμημένα υπολογιστικά συστήματα, συμπεριλαμβανομένων και μερικών γνωστών blockchain. Το PBFT απαιτεί $3f + 1$ servers (ή replicas) για να ανεχθεί f βυζαντινούς (ή ελαττωματικούς) διακομιστές. Ουσιαστικά, όλοι οι κόμβοι του μοντέλου PBFT οργανώνονται σε μια ακολουθία με έναν κόμβο να είναι ο πρωτεύον κόμβος (leader) και οι άλλοι να είναι εφεδρικοί κόμβοι. Όλοι οι κόμβοι του συστήματος επικοινωνούν μεταξύ τους και ο στόχος είναι η επίτευξη συμφωνίας για την κατάσταση του συστήματος μέσω πλειοψηφίας.



Σχήμα 2.6: Κανονική λειτουργία του PBFT

Κάθε γύρος στο PBFT για την επίτευξη συμφωνίας αποτελείτε από 4 φάσεις. Αυτό το μοντέλο είναι περισσότερο «Διοικητή και υποδιοικητή» από ένα καθαρό πρόβλημα βυζαντινών στρατηγών (BFT), όπου όλοι οι στρατηγοί είναι ίσοι, λόγω της παρουσίας ενός κόμβου ηγέτη. Οι φάσεις είναι οι εξής:

1. Ένας client στέλνει ένα αίτημα στον πρωτεύον διακομιστή.
2. Ο πρωτεύον κόμβος διανέμει το αίτημα στους εφεδρικούς κόμβους.
3. Οι κόμβοι εκτελούν το αίτημα και στη συνέχεια στέλνουν μια απάντηση στον client.
4. Ο client αναμένει $f + 1$ (το f αντιπροσωπεύει το μέγιστο αριθμό κόμβων που ενδέχεται να είναι ελαττωματικοί) από διαφορετικούς κόμβους.

Οι τρεις φάσεις είναι οι pre-prepare, prepare, και commit. Οι φάσεις pre-prepare και prepare χρησιμοποιούνται για να διατάξουν αιτήματα που στάλθηκαν στον ίδιο γύρο ακόμα και αν ο πρωτεύων διακομιστής είναι ελαττωματικός, που σημαίνει η διάταξη των αιτημάτων είναι λανθασμένη. Οι φάσεις prepare και commit χρησιμοποιούνται για να βεβαιωθεί ότι τα αιτήματα που γίνονται commit είναι πλήρως διατεταγμένα σε κάθε γύρο.

Είναι απαραίτητο οι κόμβοι να είναι ντετερμινιστικοί και να ξεκινούν από τη ίδια κατάσταση. Το τελικό αποτέλεσμα είναι όλοι οι ειλικρινής κόμβοι να έρθουν σε συμφωνία, είτε αποδέχονται είτε απορρίπτουν ένα αίτημα.

2.6 MinBFT

Το MinBFT, είναι ένας $2f + 1$ αλγόριθμος συμφωνίας με βυζαντινά σφάλματα (BFT). Ο αλγόριθμος MinBFT ακολουθεί ένα πρότυπο ανταλλαγής μηνυμάτων παρόμοιο με το PBFT (βλ. Σχήμα 2.5). Οι διακομιστές οργανώνονται σε ομάδες που ονομάζονται $views$. Κάθε $view$ έχει έναν πρωτεύον διακομιστή και όλοι οι υπόλοιποι είναι εφεδρικά αντίγραφα (backups). Οι clients είναι αυτοί που δημιουργούν τα αιτήματα που ενσωματώνουν κάποιες λειτουργίες για να εκτελεστούν από τους διακομιστές.

Στην κανονική λειτουργία, η σειρά των γεγονότων είναι η ακόλουθη: (1) ένας client στέλνει ένα αίτημα σε όλους τους διακομιστές (2) ο πρωτεύον διακομιστής ενσωματώνει

έναν αριθμό ακολουθίας στο αίτημα και το στέλνει σε όλους τους διακομιστές σε ένα μήνυμα τύπου PREPARE (3) ο κάθε διακομιστής αφού λάβει ένα μήνυμα τύπου PREPARE από το πρωτεύον κάνει πολυεκπομπή (multicast) ένα μήνυμα τύπου COMMIT στους άλλους διακομιστές (4) όταν ένας διακομιστής εγκρίνει ένα αίτημα, εκτελεί την αντίστοιχη λειτουργία του αιτήματος και στέλνει μια απάντηση στον client (5) ο client περιμένει για ίδιες απαντήσεις για ένα συγκεκριμένο αίτημα και ολοκληρώνει τη λειτουργία.

Όταν ένα σύνολο απο $f + 1$ αντίγραφα ασφαλείας υποπτευθούν ότι ο πρωτεύον διακομιστής είναι ελαττωματικός, εκτελείται μια λειτουργία αλλαγής προβολής(view change operation) και ένας νέος διακομιστής γίνεται ο πρωτεύων. Αυτός ο μηχανισμός παρέχει πρόοδο στο σύστημα αφού του επιτρέπει να προχωρήσει όταν ο πρωτεύον διακομιστής είναι ελαττωματικός.

Clients. Ένας client c για να ζητήσει την εκτέλεση μιας λειτουργίας op στέλνει ένα μήνυμα της μορφής <REQUEST, c, seq, op> σ_c σε όλους τους διακομιστές. Ο αριθμός seq είναι ο αναγνωριστικός κωδικός της αίτησης που χρησιμοποιείται για να εξασφαλιστεί η μοναδικότητα. Οι διακομιστές αποθηκεύουν σε ένα πίνακα V_{req} τον αριθμό seq του τελευταίου αιτήματος που έχουν εκτελέσει για κάθε πελάτη, απορρίπτουν αιτήματα με αναγνωριστικό κωδικό seq μικρότερο από το τελευταίο εκτελεσμένο (για να αποφευχθεί η εκτέλεση του ίδιου αιτήματος δύο φορές) και τέλος τυχόν αιτήματα που λαμβάνονται όσο το προηγούμενο είναι υπό επεξεργασία. Τα αιτήματα υπογράφονται με το ιδιωτικό κλειδί του client και αιτήματα με μη έγκυρη υπογραφή c απλά απορρίπτονται. Μετά από την αποστολή ενός αιτήματος, ο client περιμένει για $f + 1$ απαντήσεις της μορφής <REPLY, s, seq, res> από διαφορετικούς διακομιστές s με αντίστοιχα αποτελέσματα res, γεγονός που εξασφαλίζει ότι τουλάχιστον μία απάντηση θα προέρχεται από σωστό διακομιστή. Στην περίπτωση που ο client δεν λάβει αρκετές απαντήσεις κατά τη διάρκεια ενός χρονικού διαστήματος που διαβάζεται από το τοπικό ρολόι του, ξαναστέλνει το αίτημα. Σε περίπτωση που το αίτημα αυτό έχει ήδη υποβληθεί σε επεξεργασία, οι διακομιστές θα ξαναστείλουν την αποθηκευμένη απάντηση τους.

Διακομιστές. Ο πυρήνας του αλγορίθμου MinBFT εκτελείται από τους διακομιστές κατα την επεξεργασία των μηνυμάτων $PREPARE$ και $COMMIT$ (βλ. Σχήμα 2.5). Το MinBFT έχει μόνο δύο βήματα επικοινωνίας, όχι τρία όπως το PBFT. Όταν ο πρωτεύων διακομιστής (primary server) λαμβάνει ένα αίτημα από έναν client, χρησιμοποιεί ένα μήνυμα τύπου PREPARE για να διανείμει το αίτημα σε όλους τους διακομιστές. Ο κύριος ρόλος του πρωτεύοντος διακομιστή είναι να αντιστοιχίσει έναν σε κάθε αίτημα. Αυτός ο αριθμός είναι η τιμή μετρητή που επέστρεψε η υπηρεσία USIG. Αυτοί οι αριθμοί είναι διαδοχικοί, όσο ο κύριος διακομιστής δεν αλλάζει.

Η ιδέα είναι ότι ένα αίτημα m στέλνεται από το πρωτεύον διακομιστή s_i σε όλους τους διακομιστές σε ένα μήνυμα τύπου $< PREPARE, v, s_i, m, UI_i >$, και κάθε server s_j το ξαναστέλνει σε όλους τους άλλους σε ένα μήνυμα τύπου $< COMMIT, v, s_j, s_i, m, UI_i, UI_j >$, όπου το UI_j λαμβάνεται καλώντας το $createUI$. Κάθε μήνυμα που αποστέλλεται, είτε είναι $PREPARE$ είτε $COMMIT$, έχει ως εκ τούτου ένα μοναδικό αναγνωστικό UI που αποκτάται καλώντας τη συνάρτηση $createUI$, έτσι ώστε να μην υπάρχουν δύο μηνύματα με το ίδιο αναγνωριστικό. Οι διακομιστές ελέγχουν εάν τα αναγνωριστικά των μηνυμάτων που λαμβάνουν ισχύουν για αυτά τα μηνύματα, χρησιμοποιώντας μια λειτουργία επαλήθευσης.

Στην περίπτωση που ένας διακομιστής s_k δεν έλαβε κάποιο μήνυμα PREPARE αλλά έλαβε ένα μήνυμα COMMIT με έγκυρο αναγνωριστικό, τότε στέλνει το δικό του COMMIT. Αυτό μπορεί να συμβεί αν ο αποστολέας είναι ελαττωματικός και δεν στέλνει το μήνυμα PREPARE στον server s_k (αλλά το στέλνει σε άλλους διακομιστές) ή εάν το μήνυμα

PREPARE απλώς καθυστερεί και λαμβάνεται μετά τα μηνύματα COMMIT. Ένα αίτημα m γίνεται αποδεκτό από έναν διακομιστή που εκτελεί τον αλγόριθμο εάν ο διακομιστής λάβει $f+1$ έγκυρα μηνύματα $COMMIT$ από διαφορετικούς διακομιστές για το αίτημα m.

Αυτός ο βασικός αλγόριθμος πρέπει να ενισχυθεί για να αντιμετωπίσει ορισμένες περιπτώσεις. Ένας σωστός εξυπηρετητής s_j πολυεκπέμπει ένα μήνυμα COMMIT ως απάντηση σε ένα μήνυμα $< PREPARE, v, s_i, m, UI_i >$ μόνο εάν πληρούνται οι τρεις παρακάτω προϋποθέσεις: (1) το v είναι ο τρέχων αριθμός προβολής στο s_j και ο αποστολέας του μηνύματος $PREPARE$ είναι ο κύριος διακομιστής του v (μόνο ο πρωτεύον διακομιστής μπορεί να στείλει μηνύματα $PREPARE$). (2) το αίτημα m περιέχει έγκυρη υπογραφή που παράγεται από τον αιτούντα πελάτη (για να αποφευχθεί η περίπτωση όπου ένας ελαττωματικός κύριος διακομιστής δημιουργεί δικά του αιτήματα). και (3) s_j έχει ήδη αποδεχθεί ένα αίτημα $m^{'}$ με τιμή μετρητή $cv^{'} = cv - 1$, όπου cv είναι η τιμή του μετρητή στο UI_i (για να αποτρέψει ένα ελαττωματικό πρωτεύον διακομιστή από τη δημιουργία "κενών" στην ακολουθία των μηνυμάτων). Αυτή η τελευταία προϋπόθεση εξασφαλίζει ότι όχι μόνο οι αιτήσεις εκτελούνται με τη σειρά που ορίζεται από τον μετρητή του πρωτεύοντος διακομιστή, αλλά και ότι γίνονται αποδεκτές με την ίδια σειρά. Επομένως, όταν γίνει δεκτή μια αίτηση, μπορεί να εκτελεστεί αμέσως (δεν χρειάζεται να περιμένει για αιτήσεις με μικρότερο αριθμό ακολουθίας). Η μόνη εξαίρεση είναι ότι εάν ο διακομιστής είναι ελαττωματικός, μπορεί να δρομολογήσει το ίδιο αίτημα δύο φορές. Επομένως, όταν ένας διακομιστής δέχεται ένα αίτημα, πρώτα ελέγχει στο V_{req} αν το αίτημα είχε ήδη εκτελεστεί και το εκτελεί μόνο αν όχι.

Αυτός ο μηχανισμός διάταξης μηνυμάτων χρειάζεται μια διάταξη FIFO που χρησιμοποιείται και σε άλλα μηνύματα (όπως στη λειτουργία αλλαγής προβολής) που επίσης λαμβάνουν ένα μοναδικό αναγνωριστικό UI. Ένας ελαττωματικός διακομιστής επεξεργάζεται ένα μήνυμα $< ..., s_i, ..., UI_i, ... >$ απεσταλμένο από οποιονδήποτε διακομιστή s_i με τιμή μετρητή cv στο UI_i πριν επεξεργαστεί το μήνυμα $< ..., s_i, ..., UI_i^{'}, ... >$ απεσταλμένο από s_i με τιμή μετρητή $cv-1$. Για να εφαρμοστεί αυτή η ιδιότητα, κάθε διακομιστής διατηρεί ένα πίνακα V_{acc} με την υψηλότερη τιμή μετρητή cv που έλαβε από κάθε έναν από τους άλλους διακομιστές στα μηνύματα $PREPARE$, $COMMIT$, $CHECKPOINT$ ή $VIEW\text{-}CHANGE$. Η διάταξη FIFO δεν εγγυάται ότι ο αλγόριθμος λειτουργεί σε lockstep, δηλ. ο πρωτεύον διακομιστής μπορεί να στείλει πολλά μηνύματα $PREPARE$ αλλά όλοι οι διακομιστές θα δεχθούν τα αντίστοιχα αιτήματα ακολουθώντας την σειρά ακολουθίας που έχει οριστεί από το πρωτεύον διακομιστή.

Η διάταξη FIFO χρειάζεται για να συμπληρώσει τη χρήση των μοναδικών αναγνωριστικών για την αποτροπή των διπλότυπων μηνυμάτων. Αυτά τα αναγνωριστικά από μόνα τους δεν εμποδίζουν την αναπαραγωγή μηνυμάτων διότι ένας ελαττωματικός διακομιστής μπορεί να στείλει ασυνεπή μηνύματα σε διαφορετικούς διακομιστές, ακόμη και με διαφορετικά αναγνωριστικά. Με την επεξεργασία των μηνυμάτων με διάταξη FIFO που υποδεικνύουν οι μετρητές, όλοι οι σωστοί διακομιστές επεξεργάζονται τα μηνύματα που στέλνονται από τον ελαττωματικό διακομιστή με την ίδια σειρά ή σταματούν την επεξεργασία τους εάν δεν λάβουν ένα από αυτά, αποτρέποντας έτσι την αναπαραγωγή.

2.7 Η υπηρεσία USIG

Κάθε διακομιστής που εκτελεί τον αλγόριθμο συμφωνίας με βυζαντινά σφάλματα MinBFT πρέπει να ενσωματώνει την υπηρεσία Unique Sequential Identifier Generator (USIG). Η υπηρεσία αυτή αναθέτει στα μηνύματα (π.χ πίνακες με bytes) την τιμή ενός

μετρητή και την υπογράφει. Τα αναγνωριστικά είναι μοναδικά, μονοτονικά και διαδοχικά για τον συγκεκριμένο διακομιστή. Αυτές οι τρεις ιδιότητες υποδηλώνουν ότι το USIG (1) δεν θα αποδώσει ποτέ το ίδιο αναγνωριστικό σε δύο διαφορετικά μηνύματα (μοναδικότητα), (2) δεν θα αποδώσει ποτέ ένα αναγνωριστικό που είναι μικρότερο από ένα προηγούμενο (μονοτονικότητα) και (3) ποτέ δεν θα δώσει ένα αναγνωριστικό που δεν είναι ο διάδοχος του προηγούμενου (sequentiality). Αυτές οι ιδιότητες πρέπει να είναι εγγυημένες, ακόμη και αν ο διακομιστής έχει παραβιαστεί, οπότε η υπηρεσία θα πρέπει να εφαρμοστεί σε μια αξιόπιστη και απαραβίαστη μονάδα. Το interface της υπηρεσίας έχει δύο λειτουργίες:

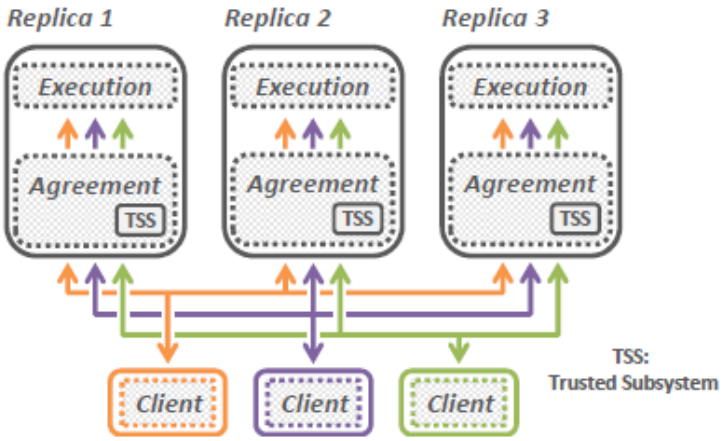
- createUI (m) - επιστρέφει ένα πιστοποιητικό USIG που περιέχει ένα μοναδικό αναγνωριστικό UI και πιστοποιεί ότι αυτό το UI δημιουργήθηκε από την αξιόπιστη και απαραβίαστη μονάδα για το μήνυμα m. Το μοναδικό αναγνωριστικό είναι ουσιαστικά μια ανάγνωση του μονοτονικού μετρητή, ο οποίος αυξάνεται κάθε φορά που καλείται η $createUI$.

- verifyUI (PK, UI, m) - επαληθεύει εάν το μοναδικό αναγνωριστικό UI είναι έγκυρο για το μήνυμα m, δηλαδή εάν το πιστοποιητικό USIG ταιριάζει με το μήνυμα και τα υπόλοιπα δεδομένα στο UI.

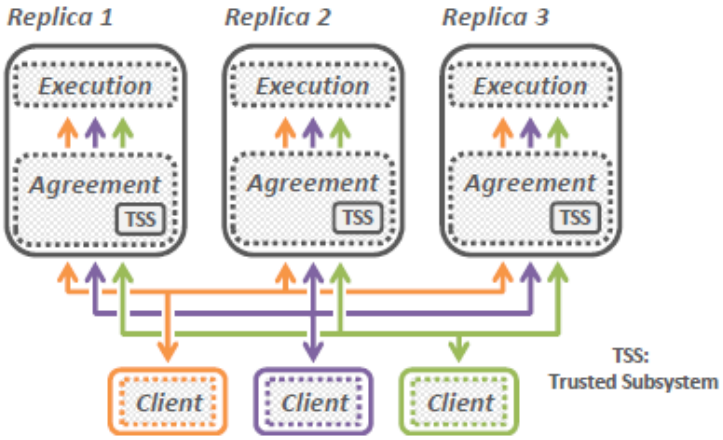
Η αξιόπιστη υπηρεσία που παρουσιάζεται σε αυτήν την εργασία παρέχει μια διεπαφή με λειτουργίες μόνο για την αύξηση ενός μετρητή και για την επαλήθευση εάν οι άλλες τιμές μετρητή (που αυξήθηκαν από άλλα αντίγραφα) έχουν πιστοποιηθεί σωστά. Σε προηγούμενες προσεγγίσεις του αλγορίθμου MinBFT είχε χρησιμοποιηθεί το ασφαλές hardware Trusted Platform Module (TPM) [3]. Περισσότερες πληροφορίες για την υλοποίηση αυτής της υπηρεσίες μπορούν να βρεθούν στο Κεφάλαιο 3.

2.8 Σύγκριση με άλλες ερευνητικές εργασίες

Οι μηχανισμοί ασφαλείας με χρήση hardware έχουν γίνει ευρέως διαδεδομένοι στους προσωπικούς υπολογιστές. Τα περιβάλλοντα αξιόπιστης εκτέλεσης (Trusted Execution Environment, TEE) είναι ήδη διαδεδομένα και σε κινητές συσκευές. Το Trusted Platform Modules (TPM) είναι διαθέσιμο στους προσωπικούς υπολογιστές εδώ και πολλά χρόνια. Πιο σύγχρονα TEEs, όπως το Intel SGX, είναι ήδη διαθέσιμα σε προσωπικούς υπολογιστές και υπολογιστές που προορίζονται για διακομιστές. Τα TEEs, όπως αναφέραμε και παραπάνω, παρέχουν προστατευόμενη μνήμη και απομονωμένη εκτέλεση από το λειτουργικό σύστημα ή τις εφαρμογές που τρέχουν στο σύστημα μας. Τα TEE προσφέρουν επίσης την απομακρυσμένη επαλήθευση της τρέχουσας διαμόρφωσης (configuration) και συμπεριφοράς μιας συσκευής μέσω απομακρυσμένης βεβαίωσης (remote attestation). Με άλλα λόγια, ένα TEE μπορεί να καταρρεύσει, αλλά δεν μπορεί να είναι Βυζαντινό.

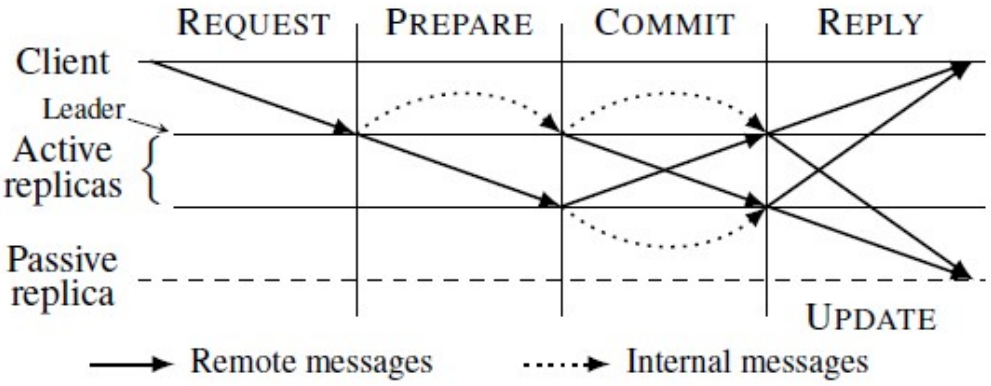


Σχήμα 2.7: Hybrid BFT state-machine replication (Figure 1 από [6])



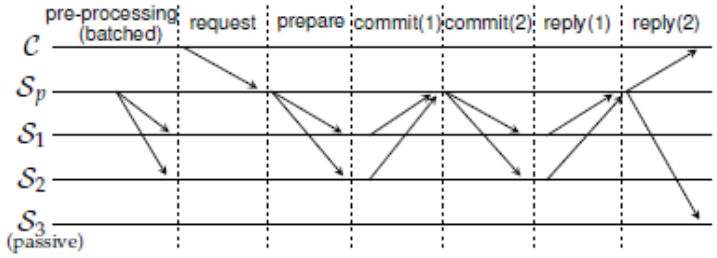
Σχήμα 2.8: Η ιδέα της παραλληλοποίησης του υβριδικού πρωτοκόλλου (Figure 2 από [6])

Παρόμοιες ερευνητικές εργασίες δείχνουν πως να χρησιμοποιούμε μηχανισμούς ασφαλείας με χρήση hardware για να μειώσουμε τους διακομιστές ή και τα βήματα επικοινωνίας στα πρωτόκολλα BFT. Στο EuroSys' 17 παρουσίαν το πρωτόκολλο Hybster [6]. Το Hybster παρουσιάζει ένα μοναδικό στοιχείο που είναι η παραλληλοποίηση του πρωτοκόλλου αναπαραγωγής, σε αντίθεση με όλα τα προηγούμενα υβριδικά συστήματα που απαιτούν κάποιο είδος διαδοχικής επεξεργασίας. Στο Hybster κάθε αντίγραφο (replica) έχει πολλαπλά περιβάλλοντα προστατευόμενης εκτέλεσης (TSS), που σημαίνει ότι μπορεί να υποστηρίξει n εικονικές χρονοσειρές σε κάθε αντίγραφο. Αντί για μόνο μία εικονική χρονοσειρά, κάθε αντίγραφο χρησιμοποιεί πολλαπλές ανεξάρτητες χρονοσειρές και αντί για μόνο ένα ασφαλές υποσύστημα, κάθε αντίγραφο έχει τόσα ασφαλή υποσυστήματα όσα και ο βαθμός παραλληλοποίησης απαιτεί. Αυτό υποστηρίζεται από την ιδέα της παραλληλοποίησης των πρωτοκόλλων συμφωνίας [7], όπου ίσες μονάδες επεξεργασίας είναι υπεύθυνες για ένα στατικά καθορισμένο υποσύνολο μονάδων συμφωνίας, εμποδίζοντας έτσι την ευκαιρία για αμφισβήτηση. Τα πολλαπλά περιβάλλοντα προστατευόμενης εκτέλεσης (TSS) ανά αντίγραφο (replica), υποστηρίζονται εύκολα για την υλοποίηση τους από την τεχνολογία Intel SGX. Η παραπάνω εργασία σχετίζεται με την δική μας δουλειά αφού και αυτό είναι ένα πρωτόκολλο BFT. Τα δύο αυτά πρωτόκολλα έχουν διαφορετικό σχεδιασμό αλλά είναι πολύ κοντά μιας και τα δυο χρησιμοποιούν την τεχνολογία Intel SGX.



Σχήμα 2.9: Ακολουθία μηνυμάτων του CheapBFT (Figure 4 από [8]).

Μια άλλη σχετική δουλειά είναι το CheapBFT [8] που χρησιμοποιεί TEE σε ένα οπτιμιστικό BFT πρωτόκολλο. Το οπτιμιστικό BFT πρωτόκολλο στοχεύει στην επίτευξη υψηλότερης απόδοσης και χαμηλότερης καθυστέρησης από άκρο σε άκρο, χρησιμοποιώντας ένα πιο αδύναμο μοντέλο συνέπειας στους διακομιστές αντίγραφα. Σε περιπτώσεις που δεν υπάρχουν σφάλματα το CheapBFT χρειάζεται μόνο $f + 1$ ενεργούς διακομιστές αντίγραφα για να επιτευχθεί συμφωνία και να εκτελεστούν τα αιτήματα των clients. Οι υπόλοιποι f ”παθητικοί” διακομιστές απλά τροποποιούν την κατάσταση τους από τις ενημερώσεις που λαμβάνουν από τους άλλους διακομιστές. Σε περίπτωση υποψίας για λανθάνουσα συμπεριφορά, το CheapBFT ενεργοποιεί ένα πρωτόκολλο μετάβασης για να ενεργοποιηθούν οι ”παθητικοί” διακομιστές.



Σχήμα 2.10: Ακολουθία μηνυμάτων του FastBFT (Figure 2 από [9]).

Ομοίως, το FastBFT [9] χρησιμοποιεί το οπτιμιστικό παράδειγμα όπως το CheapBFT που περιέγραψα παραπάνω. Χρησιμοποιείτε ομαδοποίηση (aggregation) των μηνυμάτων και κατά την διάρκεια του commit κάθε ενεργός διακομιστής αντίγραφο στέλνει το δικό του μήνυμα commit κατευθείαν στον κύριο διακομιστή αντί να το κάνει πολυεκπομπή σε όλους τους διακομιστές. Για να αποφευχθεί το επιπλέον κόστος που σχετίζεται με τη συσσωμάτωση των μηνυμάτων χρησιμοποιώντας πρωτόκολλα όπως οι πολλαπλές υπογραφές, χρησιμοποιείτε ο διαμοιρασμός μυστικών για συσσωμάτωση. Κατα την φάση του pre-processing, ο κύριος διακομιστής δημιουργεί ένα σύνολο από τυχαία μυστικά και μοιράζει τα κρυπτογραφικά hash του κάθε μυστικού. Τότε, ο κύριος διακομιστής χωρίζει κάθε μυστικό σε κομμάτια και μοιράζει το κάθε κομμάτι σε κάθε ενεργό διακομιστή αντίγραφο. Αργότερα, κατά την φάση του prepare ο κύριος διακομιστής δεσμεύει για κάθε αίτημα από τον client ένα μυστικό που προηγουμένως μοιράστηκε. Κατα την διάρκεια του commit, κάθε ενεργός διακομιστής αντίγραφο σηματοδοτεί το commit του αποκαλύπτοντας το μερίδιό του από το μυστικό. Έπειτα ο κυριος διακομιστής συλλέγει

όλα τα κομμάτια και κατασκευάζει το αρχικό μυστικό που είχε μοιράσει, το οποίο αντιπροσωπεύει τα συνολικά commit από όλους τους ενεργούς διακομιστές. Μετά ο κύριος διακομιστής στέλνει το ανασυγκροτημένο μυστικό σε όλους τους ενεργούς διακομιστές που μπορούν να το ελέγξουν με το αντίστοιχο hash. Κάθε διακομιστής χρησιμοποιεί ένα TEE. Το TEE στον κύριο διακομιστή δημιουργεί τα μυστικά, τα χωρίζει και μοιράζει με ασφάλεια το κάθε κομμάτι από το μυστικό στους άλλους διακομιστές. Κατά την φάση του commit, το TEE από κάθε ενεργό διακομιστή θα ελευθερώσει το κομμάτι του μόνο αν το μήνυμα από την φάση του prepare ήταν σωστό. Ο κύριος διακομιστής δεν θα μπορέσει να ανασυνθέσει το αρχικό μυστικό εάν δεν λάβει αρκετά κομμάτια από τους ενεργούς διακομιστές. Το TEE του κύριου διακομιστή με ασφάλεια αντιστοιχίζει στο μυστικό την τιμή ενός μονοτονικού μετρητή κατά την φάση του pre-processing. Η βασική απαίτηση είναι ότι το TEE δεν θα χρησιμοποιήσει ποτέ το ίδιο μυστικό για μία τιμή μετρητή αλλά ούτε η ίδια τιμή μετρητή θα χρησιμοποιηθεί για διαφορετικά μυστικά. Εκτός από τη διατήρηση και επαλήθευση των μονοτονικών μετρητών όπως τα υπάρχοντα πρωτόκολλα BFT που χρησιμοποιούν υλικό hardware (για να λειτουργούν με $n = 2f + 1$ αντίγραφα για να είναι ανεκτικά σε f (βυζαντινά) σφάλματα), το FastBFT χρησιμοποιεί επίσης τα TEEs για τη δημιουργία και διανομή μυστικών.

Κεφάλαιο 3

Υλοποίηση

Στην εργασία αυτή βασίστηκα σε κώδικα ο οποίος είχε αναπτυχθεί για την εργασία Efficient Byzantine Fault Tolerance [10]. Στην εργασία αυτή είχε αναπτυχθεί ο κώδικας για τον αλγόριθμο MinBFT, που αναφέραμε στην ενότητα 2.6. Το MinBFT για την λειτουργία του χρησιμοποιεί το interface USIG (βλ. ενότητα 2.7). Στην δουλεία εκείνη για την ανάπτυξη του USIG χρησιμοποιήθηκε η τεχνολογία Trusted Platform Module (TPM).

Ένα TPM chip είναι ένα ασφαλής κρυπτο-επεξεργαστής που σχεδιάστηκε να αναλαμβάνει κρυπτογραφικές λειτουργίες. Ο βασικός σκοπός του TPM είναι να διασφαλίσει την σωστή συμπεριφορά ενός ηλεκτρονικού υπολογιστή ανεξάρτητα από το λειτουργικό του σύστημα. Διασφαλίζει ότι η διαδικασία εκκίνησης ξεκινά από έναν αξιόπιστο συνδυασμό υλικού hardware και software. Η αξιόπιστη εκκίνηση βασίζεται στην μέτρηση ολόκληρου του software stack απο τον bootloader μέχρι το λειτουργικό σύστημα. Αυτό μας επιτρέπει να γνωρίζουμε ακριβώς τι λογισμικό τρέχει στο σύστημα μας και να το συγκρίνουμε με ένα άλλο software stack αναφοράς. Ένας διαχειριστής εξακολουθεί να έχει τα ίδια δικαιώματα όπως πριν και οι εφαρμογές δεν είναι απομονωμένες από το υπόλοιπο σύστημα. Το TPM δεν μπορεί να ενεργήσει σε περίπτωση παραβίασης της ακεραιότητας, μπορεί όμως να αντισταθεί σε προσπάθειες αλλοίωσης του συστήματος όπως επιθέσεις glitching (Μια επίθεση glitching είναι σκόπιμη βλάβη που γίνεται για να υπονομεύσει την ασφάλεια της συσκευής). Σε αντίθεση το Intel SGX μπορεί να παρέχει εμπιστευτικότητα και ακεραιότητα στις διεργασίες που εκτελούνται και μπορεί να επαληθεύει τον εκτελέσιμο κώδικα πριν τον εκτελέσει. Οι διεργασίες εκτελούνται μέσα σε ασφαλή enclave. Τα enclave δεν μπορούν να διαβαστούν ούτε από το kernel ούτε από οποιαδήποτε άλλη διεργασία. Εξαιτίας αυτού, μια διεργασία που τρέχει μέσα σε ένα enclave μπορεί να καταρρεύσει ή να σταματήσει αλλά δεν μπορεί να παραβιαστεί.

Σε αυτή την εργασία βασίστηκα στον κώδικα από το USIG interface και οι βασικές αλλαγές ήταν να αφαιρέσω το κομμάτι του TPM και να το αντικαταστήσω με το Intel SGX. Για να γίνει αυτό αρχικά έπρεπε να γίνει η ενεργοποίηση του Intel SGX στο σύστημα μας. Μια διαδικασία που απαιτεί την εγκατάσταση των κατάλληλων drivers και ενεργοποίηση μέσα από το bios. Στην συνέχεια εκτελώντας διάφορα παραδείγματα που δίνονται από το SDK του Intel SGX αλλά και γράφοντας δικά μου προγράμματα που κάνουν χρήση του Intel SGX κατάφερα να κατανοήσω πως λειτουργεί. Για την κατανόηση του MinBFT με την χρήση του TPM, χρειάστηκε να αναλύσω τον κώδικα και να κατανοήσω την λειτουργία του έτσι ώστε να εντοπίσω που ακριβώς επεμβαίνει το TPM και να μπορέσω να το αντικαταστήσω.

Αφού κατανόησα τον κώδικα του MinBFT άλλα και πως λειτουργεί το Intel SGX συνέχισα με την αντικατάσταση του TPM με το SGX. Αρχικά ανέπτυξα έναν ασφαλή μο-

νοτονικό μετρητή με την χρήση του Intel SGX SDK. Επειδή το MinBFT είχε αναπτυχθεί στην γλώσσα προγραμματισμού Java και τα προγράμματα για το Intel SGX γράφονται σε C++, χρησιμοποίησα το Java Native Interface (JNI) για να μπορέσουν να συνεργαστούν μεταξύ τους.

3.1 Ενεργοποιώντας το Intel SGX

Η τεχνολογία Intel SGX στους υπολογιστές από προεπιλογή είναι απενεργοποιημένη. Για να χρησιμοποιήσει κανείς το Intel SGX πρέπει να το ενεργοποιήσει αρχικά μέσω του BIOS. Αυτό απαιτεί ένα BIOS όπου από τον κατασκευαστή του υποστηρίζει ρητά το Intel SGX. Η υποστήριξη που παρέχεται από το BIOS μπορεί να ποικίλει μεταξύ κατασκευαστών. Για να λειτουργήσει το Intel SGX, αφού ενεργοποιήσουμε στο BIOS, πρέπει να εγκατασταθεί στο σύστημα το πακέτο λογισμικού Intel SGX Platform (ή αλλιώς PSW)[11].

Στην εργασία μας η εγκατάσταση του Intel SGX έγινε σε υπολογιστή με λειτουργικό σύστημα Ubuntu 16.04 LTS Server 64bits. Για αυτό χρησιμοποιήσαμε το πακέτο λογισμικού Linux Intel(R) SGX που αποτελείται από το πρόγραμμα οδήγησης Intel SGX, το Intel SGX SDK και το λογισμικό πλατφόρμας SGX Platform (PSW) της Intel[11]. Το πακέτο λογισμικού Linux Intel(R) SGX έρχεται μαζί με κάποια δείγματα κώδικα τα οποία εκτελέσαμε σε λειτουργία Hardware Mode για να ελέγξουμε το Intel SGX στον υπολογιστή μας. Εκτελέσαμε το δείγμα SampleEnclave το οποίο δημιουργεί και καταστρέφει ένα enclave, κάνει κλήσεις ECALLS και OCALLS και χρησιμοποιεί trusted βιβλιοθήκες μέσα στο enclave.

3.2 Trusted Monotonic Counter

Το USIG αναπτύχθηκε κυρίως σε Java. Το αξιόπιστο υποσύστημα USIG πραγματοποιείται στη βάση του Intel SGX, γραμμένο σε C/C++ και είναι συνδεδεμένο με την Java μέσω του Java Native Interface (JNI). Η υλοποίηση του μονοτονικού μετρητή χωρίζεται σε δύο μέρη: Στο **Enclave** που είναι ο κώδικας του enclave και είναι σε θέση να σφραγίζει και να αποσφραγίζει δεδομένα, έτσι ώστε να προστατεύονται από παραβίαση έξω από το enclave. Μπορεί να δημιουργήσει, να αυξήσει και να διαβάσει μονοτονικούς μετρητές. Λόγω της φύσης του enclave, δεν είναι σε θέση να πραγματοποιήσει κλήσεις εισόδου/εξόδου.

Στο **LibSgxJni** βρίσκεται το αναξιόπιστο κομμάτι του κώδικα. Λειτουργεί ουσιαστικά ως περιτύλιγμα για το enclave, επειδή είναι υπεύθυνο για όλες τις κλήσεις I/O. Είναι σε θέση να λαμβάνει και να αναλύει τις εντολές από την υπηρεσία USIG και να καλεί τις κατάλληλες λειτουργίες enclave. Είναι επίσης σε θέση να διαβάζει σφραγισμένους μονοτονικούς μετρητές. Η διεπαφή μεταξύ **Enclave** και **LibSgxJni** καθορίζεται από το αρχείο $Enclave.edl$. Το EDL καθορίζει τις μεθόδους που μπορεί να καλέσει ο μη αξιόπιστος κώδικας στο enclave (ECALLs).

```
enclave {
    from "sgx_tae_service.edl" import *;

    /* enum definition */
    enum TEE_ERROR {
        TEE_ERROR_INVALID_SIGNATURE = 0,
        TEE_ERROR_INVALID_COUNTER = 1,
        TEE_ERROR_INVALID_SECRET = 2
    };

    trusted {
        /* define ECALLs here. */
        public uint32_t create_counter(void);
        public uint32_t read_counter([out] uint32_t* ctr);
        public uint32_t increment_counter(void);
        public uint32_t destroy_counter(void);
    };

};
```

Snippet 3.1: $Enclave.edl$, η διεπαφή μεταξύ του enclave και του αναξιόπιστου κώδικα

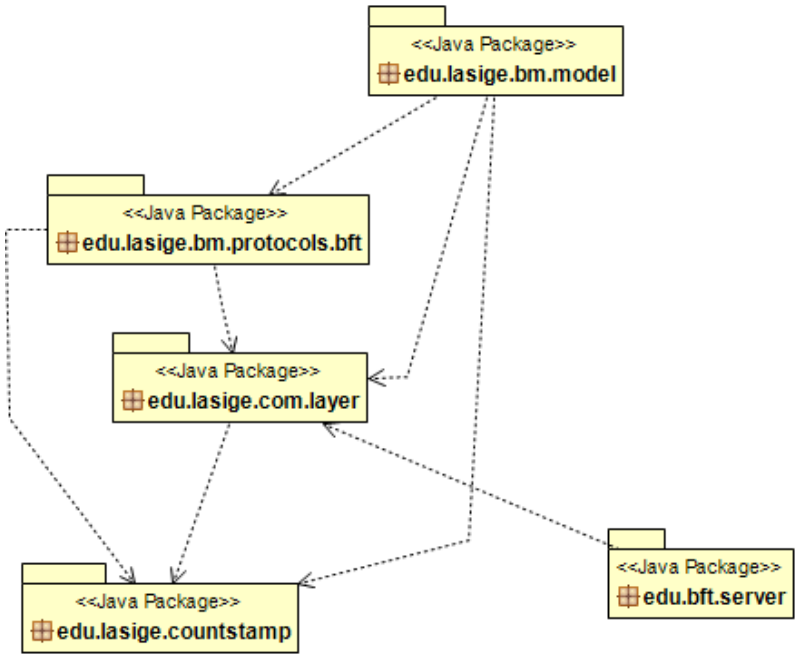
Η δομή που δημιουργήθηκε για τα δεδομένα μονοτονικού μετρητή ονομάζεται monotonic counter (Snippet 3.1) και είναι προσβάσιμη μόνο από το $enclave$. Αποτελείται από την τρέχουσα τιμή του μετρητή, καθώς και το μοναδικό αναγνωριστικό του μονοτονικού μετρητή, τα οποία χρησιμοποιεί το SGX για την πρόσβαση στην κατάλληλη μνήμη.

```
typedef struct _monotonic_counter
{
sgx_mc_uuid_t mc;
uint32_t mc_value;
} monotonic_counter;
```

Snippet 3.2: Η δομή monotonic_counter στο Enclave.cpp

Κατά τη δημιουργία και την πρόσβαση στους μονοτονικούς μετρητές, το SGX πρέπει να δημιουργήσει μια σύνδεση με το Platform Services Enclave (PSE), το οποίο είναι ένα προστατευόμενο enclave που είναι προσβάσιμο μόνο από άλλα τοπικά enclaves. Το PSE παρέχει πρόσβαση σε προστατευμένες λειτουργίες, όπως μονότονοι μετρητές, σφράγιση και βεβαίωση. Μετά την καθιέρωση αυτής της σύνδεσης, η δημιουργία ενός μονοτονικού μετρητή είναι απλά μια κλήση στο $sgx\_create\_monotonic\_counter$. Όταν έχει δημιουργηθεί ο μονοτονικός μετρητής, το struct είναι σφραγισμένο και διαβιβάζεται στον μη αξιόπιστο κώδικα. Εδώ είναι σημαντικό να σημειώσουμε ότι η μνήμη που περιέχει τα μη σφραγισμένα δεδομένα πρέπει να καθαριστεί πριν επιστρέψουν τα σφραγισμένα δεδομένα στον μη αξιόπιστο κώδικα, για να αποφευχθεί η διαρροή δεδομένων.

Η αύξηση ενός μονοτονικού μετρητή είναι πολύ παρόμοια με τη δημιουργία του. Λόγω του γεγονότος ότι ο μη αξιόπιστος κώδικας έχει μόνο πρόσβαση σε σφραγισμένες



Σχήμα 3.1: Σχεδιάγραμμα των βασικών πακέτων του MinBFT.

δομές, τα δεδομένα πρέπει πρώτα να αποσφραγιστούν. Αυτό επιτυγχάνεται με μια κλήση στο sgx_unseal_data. Επίσης, τα δεδομένα επαληθεύονται (δηλ. Ότι η τιμή του μονοτονικού μετρητή είναι όπως αναμενόταν), για να διασφαλιστεί ότι δεν έχουν παραποιηθεί τα δεδομένα. Η επαλήθευση δεν εγγυάται ότι μια άλλη οντότητα του enclave δεν έχει αποσφραγίσει τα δεδομένα και έχει αυξήσει τον μετρητή. Ωστόσο, πρέπει να είναι μια άλλη οντότητα του ακριβώς ίδιου enclave, καθώς η διαδικασία σφράγισης είναι κρυπτογραφημένη με ένα Seal Key, το οποίο είναι μοναδικό για το enclave και την πλατφόρμα[12].

3.3 MinBFT-SGX

Για την υλοποίηση του συστήματος MinBFT-SGX χρησιμοποίησα σαν βάση το σύστημα MinBFT από την εργασία Efficient Byzantine Fault Tolerance[3]. Ξεκίνησα από τον κώδικα που βρίσκεται στην ιστοσελίδα της εργασίας[10]. Ανέλυσα και κατανόησα την δομή του κώδικα ο οποίος αποτελείται από 4 βασικά μέρη(Βλ Σχήμα 3.1).

Το package **edu.lasige.com.layer** είναι υπεύθυνο για την επικοινωνία των διακομιστών και την διαχείρηση των μυνημάτων που λαμβάνει ο κάθε διακομιστής. Οι δύο βασικές κλάσεις μεσα στο πακέτο ειναι η $CommunicationSystem$ και η $BatchControl$. Η κλάση $CommunicationSystem$ αναλαμβάνει την λήψη και την μετάδοση των μυνημάτων, από και προς το $BatchControl$. Στην κλάση $BatchControl$ προωθείται καθε μήνυμα που λαμβάνει ο διακομιστής και ανάλογα με τον τύπο του μηνύματος κάνει τις αντίστοιχες λειτουργίες.

Το package **edu.lasige.com.layer** είναι υπεύθυνο για την επικοινωνία των διακομιστών και την διαχείριση των μηνυμάτων που λαμβάνει ο κάθε διακομιστής. Οι δύο βασικές κλάσεις μεσα στο πακέτο είναι η $CommunicationSystem$ και η $BatchControl$. Η κλάση $CommunicationSystem$ αναλαμβάνει την λήψη και την μετάδοση των μηνυμάτων, από και προς το $BatchControl$. Στην κλάση $BatchControl$ προωθείται κάθε μήνυμα που λαμβάνει ο διακομιστής και ανάλογα με τον τύπο του μηνύματος κάνει τις

αντίστοιχες λειτουργίες.

Τέλος, το package **edu.lasige.countstamp** είναι ο κορμός του $USIG$. Σε αυτό το πακέτο υπάρχουν οι κλάσεις οι οποίες είναι υπεύθυνες για την δημιουργία του μονοτονικού μετρητή. Η κλάση $TPMCountStampManager$ είναι αυτή η οποία επικοινωνεί με το TPM και δημιουργεί, αυξάνει και διαβάζει τον μονοτονικό μετρητή. Στην εργασία αυτή έχω επέμβει σε αυτό το κομμάτι του κώδικα και ανέπτυξα μία νέα κλάση, την $SGXCountStampManager$ η οποία επικοινωνεί μέσω JNI με το $Enclave$ και έτσι εκτελεί τις κλήσεις που υλοποίησα στο κομμάτι του Enclave για να δημιουργεί, αυξάνει και να διαβάζει τον μονοτονικό μετρητή.

3.4 Πώς τρέχει το MinBFT-SGX

Το MinBFT-SGX είναι ένας $2f + 1$ BFT αλγόριθμος. Για τρέξει λοιπόν σωστά χρειαζόμαστε τουλάχιστον τρείς διακομιστές για $f = 1$ ελλαττωματικούς διακομιστές. Αρχικά ο κάθε διακομιστής πρέπει να υποστηρίζει και να έχει εγκατεστημένη την πλατφόρμα Intel SGX. Επίσης σε κάθε διακομιστή πρέπει να έχουμε εγκατεστημένη την έκδοση Java 8 ή μεγαλύτερη. Τέλος ο κάθε διακομιστής πρέπει να γνωρίζει τις διευθύνσεις των άλλων διακομιστών καθώς και των clients. Οι διευθύνσεις αυτές είναι καταχωρημένες στο αρχείο $hosts.config$ στο directory $config$. Αφού έχουμε κάνει όλα τα παραπάνω, εκτελούμαι την παρακάτω εντολή σε κάθε διακομιστή:

```
java -cp dist/LASIGE-BFT-Protocols.jar:lib/* edu.bft.server.
   ServerCS [server id] [protocol] [size] [delay attack] [
   batchSize]
```

Το όρισμα $[serverid]$ δηλώνει το αναγνωριστικό του διακομιστή, το $[protocol]$ είναι κάθε φορά MinBFT, το $[size]$ δηλώνει το μέγεθος του μηνύματος που στέλνει για απάντηση ο διακομιστής, στο $[delayattack]$ αν δώσουμε τιμή μεγαλύτερη από το 0 δηλώνει σε πόσο χρόνο αυτός ο διακομιστής θα καταρρεύσει και τέλος το $[batchSize]$ το οποίο δηλώνει πόσα μηνύματα θα επιβεβαιώνει ταυτόχρονα.

3.5 Προγράμματα Αξιολόγησης (Benchmarks)

Για την αξιολόγηση του monotonic counter σε συνδυασμό με την κλήση από κάποια Java εφαρμογή μέσω του JNI ανέπτυξα την εφαρμογή $SgxTimeTest$. Το $SgxTimeTest$ παίρνει σαν όρισμα έναν αριθμό που αφορά πόσες αυξήσεις του μονοτονικού μετρητή θα κάνει. Δημιουργεί ένα $enclave$ και δημιουργεί ένα for-loop όσο η τιμή του ορίσματος που δώσαμε και σε κάθε επανάληψη παίρνει την τιμή του τοπικού ρολογιού αμέσως πριν την αύξηση του μετρητή και παίρνει ακόμη μια τιμή αμέσως μετά από την αύξηση του μετρητή και αποθηκεύει αυτές τις μετρήσεις. Μετά το τέλος του for-loop η εφαρμογή υπολογίζει τον μέσο όρο της διάρκειας μιας αύξησης του μετρητή και υπολογίζει και την τυπική απόκλιση του μέσου όρου.

Για την αξιολόγηση του $MinBFT - SGX$ ανέπτυξα 3 είδη client, τα οποία βρίσκονται στο package $edu.bft.client$. Στην κλάση $ThroughputClient$ φτιάχνω έναν client ο οποίος παίρνει σαν ορίσματα τις εξής μεταβλητές:$[clientid][request][concurrent][interval]$. Το $[clientid]$ είναι το αναγνωριστικό του client, το $[request]$ είναι πόσα μηνύματα ο client

θα στείλει, το $[concurrent]$ είναι πόσα μηνύματα θα στείλει ταυτόχρονα ο client και τέλος το $[interval]$ είναι πόσο θα περιμένει ανάμεσα σε κάθε ταυτόχρονη αποστολή μηνυμάτων που θα κάνει. Αυτό το είδος client είναι για την μέτρηση του throughput των διακομιστών αφού το μόνο που κάνει είναι να στέλνει μαζικά μηνύματα έτσι ώστε να δημιουργήσει υψηλό φόρτο στους διακομιστές. Στην κλάση $ClientThread$ φτιάχνω έναν client ο οποίος παίρνει σαν ορίσματα τις εξής μεταβλητές:$[client][clients][request][interval]$. Το $[clientid]$ είναι το αναγνωριστικό του client, το $[clients]$ είναι πόσα threads από clients θα φτιαχτούν, το $[request]$ είναι πόσα μηνύματα θα στείλει ο κάθε client και το $[interval]$ είναι το πόσο θα περιμένει ανάμεσα σε κάθε αποστολή μηνύματος. Για κάθε μήνυμα που στέλνει κρατάει την τιμή του τοπικού ρολογιού και μόλις λάβει 3 επιβεβαιώσεις από τους διακομιστές κρατάει πάλι την τιμή του τοπικού ρολογιού. Μετά το τέλος της λήψης όλων των επιβεβαιώσεων από τους διακομιστές για όλα τα μηνύματα τυπώνει τον μέσο όρο επιβεβαίωσης μιας αίτησης και την τυπική απόκλιση του μέσου όρου. Τέλος, στην κλάση $LatencyClient$ φτιάχνω έναν client ο οποίος παίρνει σαν ορίσματα τις εξής μεταβλητές: $[clientid][request][ignore][size]$. Το $[clientid]$ είναι το αναγνωριστικό του client, το $[request]$ είναι είναι πόσα μηνύματα θα στείλει ο client, το $[ignore]$ είναι πόσα μηνύματα θα αγνοήσει ο client απο τις μετρήσεις του και το $[size]$ είναι το μέγεθος του κάθε μηνύματος.

Σύνοψη προγραμμάτων για benchmark	
Πρόγραμμα	Περιγραφή
SgxTimeTest	Υπολογισμός μέσης διάρκειας για την αύξηση ενός μονοτονικού μετρητή με το Intel SGX
ThroughputClient	Μέτρηση του throughput των διακομιστών
ClientThread	Μέτρηση της μέσης καθυστέρησης για ένα σύνολο μηνυμάτων
LatencyClient	Μέτρηση της μέσης καθυστέρησης για ένα μήνυμα

Πίνακας 3.1: Σύνοψη και περιγραφή των προγραμμάτων αξιολόγησης που χρησιμοποιήθηκαν

3.5.1 Πώς τρέχουν οι clients

Για την εκτέλεση ενός client θα χρειαστούμε ένα ξεχωριστό μηχάνημα το οποίο πρέπει να έχει εγκατεστημένη την έκδοση Java 8 ή μεγαλύτερη. Επίσης ο client πρέπει να γνωρίζει τις διευθύνσεις των άλλων διακομιστών. Η διευθύνσεις αυτές είναι καταχωρημένες στο αρχείο $hosts.config$ στο directory $config$. Αφού έχουμε κάνει όλα τα παραπάνω, εκτελούμε την παρακάτω εντολή στον client:

```
java -cp dist/LASIGE-BFT-Protocols.jar:lib/* edu.bft.client.
   ThroughputClient [client id] [request] [concurrent] [interval
   ]
```

Κεφάλαιο 4

Αξιολόγηση

Αυτή η ενότητα παρουσιάζει τα αποτελέσματα απόδοσης του αλγορίθμου MinBFT-SGX με τη χρήση micro-benchmarks. Μετρήσαμε την καθυστέρηση και την απόδοση των εφαρμογών MinBFT με μηδενικές λειτουργίες. Το PBFT θεωρείται συχνά ως η γραμμή βάσης για τους αλγόριθμους BFT, επομένως μας ενδιαφέρει η σύγκριση της δικής μας υλοποίησης του αλγορίθμου με την εφαρμογή που είναι διαθέσιμη στο web. Για να συγκρίνουμε αυτή την εφαρμογή με τον αλγόριθμο MinBFT-SGX, χρησιμοποίησα την υλοποίηση της κανονικής λειτουργίας του PBFT στην Java (JPBFT). Θα υπάρξουν κάποια micro-benchmarks τα οποία θα εστιάσουν σε μικρές μετρήσεις, όπως για παράδειγμα τον χρόνο που απαιτείται για την αύξηση του μετρητή μέσα στο enclave. Υπάρχουν επίσης και κάποια End-to-End bechmarks, τα οποία μετράνε την καθυστέρηση και το throughput του αλγορίθμου. Θεωρήσαμε μια εγκατάσταση που μπορεί να ανεχτεί έναν ελαττωματικό διακομιστή ($f = 1$), απαιτώντας $n = 4$ servers για διακομιστές JPBFT και $n = 3$ για MinBFT. Δεν ερευνήσαμε τις υψηλότερες τιμές του f επειδή η αύξηση του αριθμού των αντιτύπων είναι δαπανηρή, στην πράξη το $f = 1$ είναι μια χρήσιμη επιλογή. Εκτελέσαμε από 1 έως 4 λογικούς πελάτες σε 1 μηχάνημα. Το μηχάνημα που χρησιμοποιήσαμε ήταν ένα OptiPlex 3050 της Dell με επεξεργαστή Intel Core i5-7500 CPU @ 3.40GHz × 4 και μνήμη 4GB. Λόγω έλλειψης πόρων για τις μηχανές διακομιστών και πελατών όπου αναφέρετε χρησιμοποιήσαμε την υπηρεσία Google Cloud Platform. Οι διακομιστές ήταν εξοπλισμένοι με επεξεργαστή Intel Skylake @ 2,8 GHz και με RAM 7.5 GB. Όλα τα μηχανήματα εκτελούσαν Open JDK 1.8 πάνω στο λειτουργικό σύστημα Ubuntu 16.04.4 LTS. Οι υπολογιστές του Google Cloud Platform δεν είχαν υποστήριξη του Intel SGX οπότε όλα τα πειράματα έγιναν σε Simulation Mode. Όλα τα πειράματα λειτουργούν μόνο σε κανονική λειτουργία, χωρίς ελαττωματικούς διακομιστές και λήξεις χρονικού ορίου, τα οποία θεωρείται συνήθως η συνήθης περίπτωση.

4.1 Micro-Benchmarks

Προτού ερευνήσουμε την απόδοση του πρωτοκόλλου MinBFT-SGX, αξιολογούμε το υποσύστημα USIG με Intel SGX. Για αυτό το σκοπό υλοποιήσαμε το πρόγραμμα $SgxTimeTest$ που περιγράφω στην ενότητα 3.5. Το μηχάνημα που εκτελέσαμε τα πειράματα μας είναι εξοπλισμένο με Intel Sgx που μπορεί να χρησιμοποιηθεί είτε απευθείας είτε από την Java με την βοήθεια του JNI. Τρέξαμε το πρόγραμμα $SgxTimeTest$ για 10000 επαναλήψεις. Το πρόγραμμα $SgxTimeTest$ μετράει τη καθυστέρηση κάθε φορά που κάνουμε μια αύξηση του μετρητή.

Ο μέσος χρόνος που απαιτείται από την υπηρεσία USIG που βασίζεται σε Intel SGX

για την εκτέλεση του createUI, αύξηση και ανάγνωση μονοτονικού μετρητή, είναι κατά μέσο όρο 340 ms με τυπική απόκλιση 21.11, ενώ του TPM είναι 797 ms (ενότητα 7.2 στην εργασία [3]). Παρατηρούμε μια σημαντική μείωση στο χρόνο που είναι της τάξεως του 57.34% συγκριτικά με το Intel SGX του σήμερα και του TPM (2011). Βλέπουμε λοιπόν ότι οι μηχανισμοί αυτοί έχουν βελτιωθεί τα τελευταία χρόνια. Μια εξήγηση για τον χρόνο αυτό είναι ότι σε κάθε κλήση για αύξηση του μετρητή σημαίνει πως γίνεται εγγραφή στην μη πτητική μνήμη (non-volatile) που είναι διαθέσιμη στην πλατφόρμα. Οι επανειλημμένες όμως λειτουργίες εγγραφής μπορεί να προκαλέσουν στην μνήμη φθορά. Για αυτό το Intel SGX το αποτρέπει αυτό περιορίζοντας το ρυθμό που οι λειτουργίες μονοτονικού μετρητή μπορούν να εκτελεστούν.

4.2 End-to-End Benchmarks (Latency)

Για το πρώτο μέρος των End-to-End benchmarks συγκρίναμε το MinBFT-SGX με το PBFT. Αξιολογήσαμε την απόδοση του αλγορίθμου, JPBFT με του MinBFT-SGX, σε ένα τοπικό δίκτυο. Χρησιμοποιώντας το πρόγραμμα $LatencyClient$ (βλ ενότητα 3.5) μετρήσαμε την καθυστέρηση των αλγορίθμων χρησιμοποιώντας μια απλή υπηρεσία που εκτελεί null λειτουργίες, με το μέγεθος των αιτήσεων και των απαντήσεων να κυμαίνονται μεταξύ 0 και 4K byte. Η καθυστέρηση μετρήθηκε στον client, διαβάζοντας το ρολόι του αμέσως πριν από την αποστολή του αιτήματος, και αμέσως μετά την ενοποίηση μιας απάντησης (δηλ. Η ίδια απάντηση λήφθηκε από τρεις διακομιστές) και αφαιρώντας την πρώτη από την τελευταία. Κάθε αίτηση εκτελέστηκε συγχρονισμένα, δηλαδή περίμενε μια απάντηση προτού προβεί σε νέα ενέργεια. Τα αποτελέσματα προέκυψαν χρονομετρώντας 100.000 αιτήματα σε δύο εκτελέσεις. Οι παρακάτω χρόνοι καθυστέρησης (Latency) είναι οι μέσοι όροι αυτών των δύο εκτελέσεων. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 4.1.

Μετρήσεις Latency			
Req/Res	JPBFT	MinBFT-SGX Hardware Mode	MinBFT-SGX Simulation Mode
0/0	0.58 ms	1339.46 ms	1.55 ms
4K/0	0.59 ms	1432.68 ms	1.56 ms
0/4k	0.60 ms	1546.67 ms	1.58 ms
4k/4k	0.60 ms	————- ms	1.60 ms

Πίνακας 4.1: Καθυστέρηση για διάφορα μεγέθη Αίτησης και Απάντησης για το JPBFT και το MinBFT-SGX

Σε αυτό το πείραμα, το JPBFT έχει δείξει καλύτερη απόδοση από το MinBFT είτε σε simulation είτε σε hardware mode. Αυτό εξηγείται γιατί σε σύγκριση με την JPBFT, η MinBFT-SGX έχει πολύ μεγαλύτερη καθυστέρηση εξαιτίας του επιπλέον κόστους για να έχει πρόσβαση στην υπηρεσία USIG για να δημιουργήσει το UI. Εξετάσαμε την περίπτωση που το USIG υλοποιείτε με έναν τοπικό μετρητή και οι χρόνοι βγήκαν σχεδόν ίδιοι με αυτούς του SGX σε simulation mode. Μια πιθανή εξήγηση για αυτό το επιπλέον κόστος, σε σχέση με το PBFT, είναι επειδή το MinBFT χρησιμοποιεί αρκετές ουρές και event listeners για την ομαλή διεκπεραίωση όταν έχουμε μεγαλύτερο αριθμό συνδέσεων.

Στο παρακάτω πίνακα βλέπουμε την καθυστέρηση από το MinBFT με τη χρήση του TPM USIG όπως αναφέρετε στην εργασία Efficient Byzantine Fault Tolerance [3]. Βλέπουμε ότι αυτοί οι δύο χρόνοι να είναι πολύ κοντά με λίγο γρηγορότερη την έκδοση με

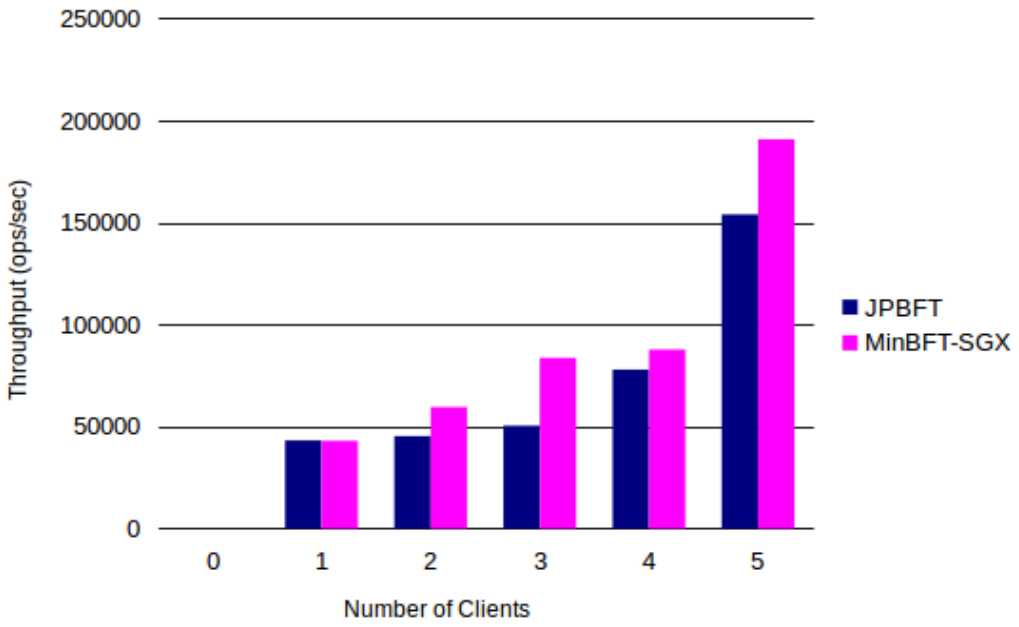
την χρήση του Intel SGX. Παρόλο που η πρώτη έκδοση έχει γίνει περίπου δέκα χρόνια πριν και από τότε έχει υπάρξει τεχνολογική εξέλιξή, βλέπουμε πως δεν υπάρχει μεγάλη βελτίωση στο χρόνο αυτό. Η διαφορά στους χρόνους οφείλετε κυρίως στην γρηγορότερη εκτέλεση της αύξησης και της ανάγνωσης του μονοτονικού μετρητή που παρουσίασα παραπάνω.

Αλγόριθμος	Latency	Peak Throughput
MinBFT-TPM	1617 ms	23404 ops/s

Πίνακας 4.2: Καθυστέρηση και μέγιστο throughput του MinBFT με χρήση του TPM USIG (Table 4 σελίδα 12 από [3])

4.3 Αξιολόγηση του MinBFT-SGX (Throughput)

Σε αυτό το πείραμα έχουμε ως στόχο τη μέτρηση της μέγιστης απόδοσης του αλγορίθμου με διαφορετικά φορτία. Χρησιμοποιώντας το πρόγραμμα $ThroughputClient$ (βλ ενότητα 3.5) πραγματοποιήσαμε πειράματα με αιτήματα και απαντήσεις μεγέθους 0 bytes. Έχουμε διαφοροποιήσει τον αριθμό των λογικών clients μεταξύ 1 έως 5 σε κάθε πείραμα, όπου κάθε πελάτης έστειλε αιτήσεις με το συγκεκριμένο ρυθμό που του ορίζω εγώ (χωρίς να περιμένει απαντήσεις), προκειμένου να επιτευχθεί η μέγιστη δυνατή απόδοση. Η παράμετρος στο πρόγραμμα $ThroughputClient$ που ορίζει τον ρυθμό με τον οποίο στέλνει τις αιτήσεις είναι το $[concurrent]$ το οποίο ορίζει πόσα μηνύματα θα στείλει ταυτόχρονα ο client ταυτόχρονα και τέλος το $[interval]$ που ορίζει πόσο θα περιμένει ο client ανάμεσα σε κάθε ταυτόχρονη αποστολή μηνυμάτων. Στα πειράματα μου όρισα το $[concurrent]$ ίσο με 10000 μηνύματα και το $[interval]$ ίσο με 3ms. Κάθε πείραμα έτρεξε για 100.000 αιτήσεις ανά client για να επιτρέψει τη σταθεροποίηση της απόδοσης πριν από την καταγραφή δεδομένων για τις ακόλουθες 100.000 λειτουργίες. Η μέτρηση γίνεται στο πρωτεύον διακομιστή όταν στέλνει την απάντηση στον client. Όλοι οι διακομιστές και ο client τρέχουν σε ένα μηχάνημα με το SGX σε hardware mode.



Σχήμα 4.1: Απόδοση για 0/0 λειτουργίες για το MinBFT και το JPBFT

Το Σχήμα 4.1 δείχνει ότι τα λιγότερα βήματα επικοινωνίας και ο αριθμός των αντιγράφων στο MinBFT-SGX αντικατοπτρίζεται σε υψηλότερη απόδοση όταν το φόρτο ανεβαίνει , επιτυγχάνοντας περίπου 200.000 λειτουργίες ανά δευτερόλεπτο. Είναι ενδιαφέρον να παρατηρήσουμε ότι ο μειωμένος αριθμός σταδίων επικοινωνίας και αντιγράφων στο MinBFT κάνει τα αντίγραφα (διακομιστές) να επεξεργάζονται λιγότερα μηνύματα (λιγότερα I/O), πράγμα που αυξάνει την απόδοση.

Για αναφορά στην εργασία Efficient Byzantine Fault Tolerance [3] η μέγιστη απόδοση όταν το MinBFT χρησιμοποιεί το TPM USIG που έτρεξε σε σύνολο δέκα μηχανημάτων (4 servers, 6 clients), ήταν 23404 ops/sec (Table 4, Εργασία [3]). Στην δική μας περίπτωση η μέγιστη απόδοση ήταν καλύτερη με τον περιορισμό όμως πως οι διακομιστές και οι clients έτρεξαν σε ένα μηχάνημα. Δεν μπορέσαμε να κάνουμε δικές μας μετρήσεις του MinBFT με TPM USIG διότι δεν διαθέταμε το κατάλληλο hardware. Ωστόσο, για να ληφθούν αυτές οι τιμές με το TPM USIG, χρειαζόταν να συγκεντρωθεί μεγάλος αριθμός αιτημάτων στα μηνύματα $PREPARE$ επειδή το TPM έχει τον περιορισμό μιας αύξησης ανά 3,5 δευτερόλεπτα. Στην περίπτωση μας το Intel SGX δεν έχει αυτόν τον περιορισμό οπότε πετυχαίνουμε καλύτερη απόδοση με πολύ μικρότερο batch size. Στο MinBFT με το TPM USIG η μέγιστη απόδοση επιτεύχθηκε με batch size μεγαλύτερο από 20000 μηνύματα, ενώ στο MinBFT με SGX χρησιμοποιήσαμε batch size ίσο με 5000 μηνύματα.

Στο δεύτερο πείραμα για την αξιολόγηση της συνολικής απόδοσης του αλγορίθμου χρησιμοποιήσαμε το πρόγραμμα $ClientThread$ (βλ ενότητα 3.5) για να δούμε την επίδραση του μεγέθους του response. Με αυτό το πείραμα μετράμε το μέσο χρόνο της συνολικής εκτέλεσης των requests ενός client. Δοκιμάσαμε σε clients από 1 έως 4 με τον κάθε client να στέλνει συνολικά 100000 requests και οι διακομιστές να χρησιμοποιούν batchsize ίσο με 5000. Το μέγεθος των requests και των απαντήσεων ήταν 0/0 και 0/4 και ο client περιμένει τουλάχιστον 2 απαντήσεις για κάθε αίτημα.

Σχήμα 4.2: Μέσος χρόνος ολοκλήρωσης για 0/0 και 0/4 λειτουργίες για το MinBFT

Παρατηρούμε ότι το διαφορετικό πλήθος από clients το οποίο συνεπάγεται και σε μεγαλύτερο φόρτο στους διακομιστές δεν επηρεάζει κατά πολύ τους χρόνους διεκπεραίωσης των αιτημάτων. Αυτοί οι χρόνοι εξηγούνται επειδή το σύστημα μας μπορεί και επιβεβαιώνει τα αιτήματα ομαδοποιώντας τα και στέλνει μαζικές απαντήσεις στους clients.

Κεφάλαιο 5

Συμπεράσματα

5.1 Συμπεράσματα

Όπως αναφέρθηκε και στην εισαγωγή απώτερος σκοπός της διπλωματικής είναι η ανάδειξη των ωφελειών του Προστατευόμενου Περιβάλλοντος Εκτέλεσης Intel SGX όπως επίσης και η εκλογή χρήσιμων συμπερασμάτων, τα οποία συμπεράσματα αντλούνται καθ' όλη τη διάρκεια της ενασχόλησης με τη διπλωματική εργασία. Οι αλγόριθμοι BFT συνήθως απαιτούν διακομιστές $3f + 1$ να ανεχτούν τους βυζαντινούς διακομιστές, γεγονός που συνεπάγεται σημαντικό κόστος στο υλικό, το λογισμικό και τη διαχείριση. Τα αποτελέσματα που πήραμε από την εργασία αυτή συγκρινόμενα με αυτά από την εργασία Efficient Byzantine Fault Tolerance [3] στην οποία υλοποιήθηκε η αρχική έκδοση του MinBFT δείχνουν ότι η έκδοση με τη χρήση του Intel SGX είναι ανταγωνιστική. Ωστόσο, το TPM είναι μια τεχνολογία δέκα χρόνων παλαιότερη σε σχέση με το Intel SGX και επίσης δεν καταφέραμε να τρέξουμε την αρχική έκδοση του MinBFT στο δικό μας hardware ώστε να μπορούμε να συγκρίνουμε τα αποτελέσματα με ασφάλεια. Ένα από τα οφέλη που προέκυψαν είναι ότι το MinBFT με την χρήση του Intel SGX πέτυχε καλύτερη απόδοση από την προηγούμενη προσέγγιση με την χρήση του TPM. Σε αυτό συνέβαλε ότι το Intel SGX δεν έχει τον περιορισμό της αύξησης του μονοτονικού μετρητή ανά 3.5 δευτερόλεπτα όπως έχει το TPM. Επίσης η Intel από το 2015 που ανακοίνωσε την τεχνολογία Software Guard Extensions, από την έκτη γενιά (Skylake) επεξεργαστών και μετέπειτα την ενσωματώνει δωρεάν σε κάθε νέα γενιά επεξεργαστών. Αυτό σημαίνει πως η τεχνολογία αυτή βρίσκεται σε κάθε προσωπικό υπολογιστή και τα οφέλη της υλοποίησης μας είναι προσβάσιμα από όλους.

5.2 Μελλοντική Δουλειά

Σαν μελλοντική δουλεία το σύστημα αυτό μπορεί να επεκταθεί σε πραγματικά μηχανήματα όταν αυτό γίνει διαθέσιμο. Επιπλέον, το σύστημα αυτό μπορεί να αποτελέσει την καρδιά του μηχανισμού συμφωνίας σε ένα σύστημα blockchain ή να αντικαταστήσει το BFT SMaRT [13] στο Hyperledger Fabric [1] της εταιρείας IBM. Το Hyperledger Fabric είναι ένα permissioned blockchain ,στο οποίο συνεισέφερε αρχικά η IBM και η Digital Asset, παρέχοντας μια αρθρωτή αρχιτεκτονική που περιγράφει τους ρόλους μεταξύ των κόμβων, την εκτέλεση των έξυπνων συμβολαίων (smart contracts) και του μηχανισμού συμφωνίας. Η αρχιτεκτονική Hyperledger Fabric επιτρέπει στους χρήστες να επιλέγουν για μηχανισμό συμφωνίας το πρωτόκολλο BFT. Είναι σημαντικό να σημειωθεί ότι το

BFT, εφαρμόζεται μόνο για την εισαγωγή συναλλαγών στο Hyperledger Fabric. Η δουλειά του είναι να εξασφαλίσει ότι κάθε διακομιστής έχει την ίδια λίστα συναλλαγών στο κατάλογο του. Με την χρήση του MinBFT-SGX στο Hyperledger θα μπορούσε να έχει αύξηση στην απόδοση του blockchain καθώς η συμφωνία θα επιτυγχάνεται με μεγαλύτερο ρυθμό και η συντήρηση του δικτύου θα είναι ευκολότερη και λιγότερο οικονομικά δαπανηρή αφού το MinBFT-SGX μειώνει τους διακομιστές σε $2f + 1$. Βέβαια στα πλαίσια της μελλοντικής δουλειάς εντάσσεται και η σύγκριση των δύο αυτών μηχανισμών συμφωνίας μέσα στο Hyperledger.

Κεφάλαιο 6

Παράρτημα

6.1 Monotonic Counter code

```
1  #include "Enclave_t.h"
2  #include "sgx_trts.h"
3  #include "sgx_tseal.h"
4  #include "sgx_tae_service.h"
5  #include "string.h"
6
7  #define REPLAY_PROTECTED_SECRET_SIZE  32
8
9  typedef struct _monotonic_counter
10 {
11         sgx_mc_uuid_t mc;
12         uint32_t mc_value;
13 }monotonic_counter;
14
15
16 static uint32_t verify_mc(monotonic_counter* mc2verify)
17 {
18         uint32_t ret = 0;
19         uint32_t mc_value;
20         ret = sgx_read_monotonic_counter(&mc2verify->mc, &
              mc_value);
21         if (ret != SGX_SUCCESS)
22         {
23                 switch (ret)
24                 {
25                 case SGX_ERROR_SERVICE_UNAVAILABLE:
26                         /* Architecture Enclave Service Manager
                               is not installed or not
27                             working properly.*/
28                         break;
29                 case SGX_ERROR_SERVICE_TIMEOUT:
30                         /* retry the operation later*/
31                         break;
32                 case SGX_ERROR_BUSY:
```

```
33                      /* retry the operation later*/
34                      break;
35              case SGX_ERROR_MC_NOT_FOUND:
36                      /* the the Monotonic Counter ID is
                           invalid.*/
37                      break;
38              default:
39                      /*other errors*/
40                      break;
41              }
42      }
43      else if (mc_value != mc2verify->mc_value)
44      {
45              ret = 0x002;
46      }
47      return ret;
48 }
49
50 static uint32_t read_and_verify_monotonic_counter(
51      const sgx_sealed_data_t* mc2unseal,
52      monotonic_counter* mc_unsealed)
53 {
54      uint32_t ret = 0;
55      monotonic_counter temp_unseal;
56      uint32_t unseal_length = sizeof(monotonic_counter);
57
58      ret = sgx_unseal_data(mc2unseal, NULL, 0,
59                            (uint8_t*)&temp_unseal, &
                                 unseal_length);
60      if (ret != SGX_SUCCESS)
61      {
62              switch (ret)
63              {
64              case SGX_ERROR_MAC_MISMATCH:
65                      /* MAC of the sealed data is incorrect.
66                         The sealed data has been tampered.*/
67                      break;
68              case SGX_ERROR_INVALID_ATTRIBUTE:
69                      /*Indicates attribute field of the
                           sealed data is incorrect.*/
70                      break;
71              case SGX_ERROR_INVALID_ISVSVN:
72                      /* Indicates isv_svn field of the sealed
                            data is greater than
73                          the enclave's ISVSVN. This is a
                              downgraded enclave.*/
74                      break;
75              case SGX_ERROR_INVALID_CPUSVN:
76                      /* Indicates cpu_svn field of the sealed
                            data is greater than
```

```
77                              the platform's cpu_svn. enclave is
                                   on a downgraded platform.*/
78                      break;
79              case SGX_ERROR_INVALID_KEYNAME:
80                      /*Indicates key_name field of the sealed
                            data is incorrect.*/
81                      break;
82              default:
83                      /*other errors*/
84                      break;
85              }
86              return ret;
87      }
88      ret = verify_mc(&temp_unseal);
89      if (ret == SGX_SUCCESS)
90              memcpy(mc_unsealed, &temp_unseal, sizeof(
                   monotonic_counter));
91      /* remember to clear secret data after been used by
           memset_s */
92      memset_s(&temp_unseal, sizeof(monotonic_counter), 0,
93               sizeof(monotonic_counter));
94      return ret;
95 }
96
97 uint32_t get_size() {
98      return sgx_calc_sealed_data_size(0, sizeof(
           monotonic_counter));
99 }
100
101 uint32_t create_sealed_monotonic_counter(uint8_t*
       sealed_mc_result, uint32_t sealed_mc_size)
102 {
103      uint32_t ret = 0;
104      int busy_retry_times = 2;
105      monotonic_counter mc2seal;
106      memset(&mc2seal, 0, sizeof(mc2seal));
107      uint32_t size = sgx_calc_sealed_data_size(0,
108                                                sizeof(
                                                     monotonic_counter
                                                     ));
109      if (sealed_mc_size != size)
110              return SGX_ERROR_INVALID_PARAMETER;
111      do {
112              ret = sgx_create_pse_session();
113      } while (ret == SGX_ERROR_BUSY && busy_retry_times--);
114      if (ret != SGX_SUCCESS)
115              return ret;
116      do
117      {
118              ret = sgx_create_monotonic_counter(&mc2seal.mc,
                       &mc2seal.mc_value);
```

```
119                 if (ret != SGX_SUCCESS)
120                 {
121                         switch (ret)
122                         {
123                         case SGX_ERROR_SERVICE_UNAVAILABLE:
124                                 /* Architecture Enclave Service
                                       Manager is not installed or
                                       not
125                                    working properly.*/
126                                 break;
127                         case SGX_ERROR_SERVICE_TIMEOUT:
128                                 /* retry the operation later*/
129                                 break;
130                         case SGX_ERROR_BUSY:
131                                 /* retry the operation later*/
132                                 break;
133                         case SGX_ERROR_MC_OVER_QUOTA:
134                                 /* SGX Platform Service enforces
                                        a quota scheme on the
                                       Monotonic
135                                    Counters a SGX app can
                                          maintain. the enclave has
                                          reached the
136                                    quota.*/
137                                 break;
138                         case SGX_ERROR_MC_USED_UP:
139                                 /* the Monotonic Counter has
                                       been used up and cannot
                                       create
140                                    Monotonic Counter anymore.*/
141                                 break;
142                         default:
143                                 /*other errors*/
144                                 break;
145                         }
146                         break;
147                 }
148
149                 /*sealing the plaintext to ciphertext. The
                       ciphertext can be delivered
150                    outside of enclave.*/
151                 ret = sgx_seal_data(0, NULL, sizeof(mc2seal), (
                       uint8_t*)&mc2seal,
152                                     sealed_mc_size, (
                                        sgx_sealed_data_t*)
                                        sealed_mc_result);
153         } while (0);
154
155         /* remember to clear secret data after been used by
                memset_s */
156         memset_s(&mc2seal, sizeof(monotonic_counter), 0,
```

```
157                 sizeof(monotonic_counter));
158         sgx_close_pse_session();
159         return ret;
160 }
161
162 uint32_t increment_monotonic_counter(uint8_t* sealed_mc_result,
      uint32_t sealed_mc_size)
163 {
164         uint32_t ret = 0;
165         int busy_retry_times = 2;
166         monotonic_counter mc_unsealed;
167         monotonic_counter mc2seal;
168         do {
169                 ret = sgx_create_pse_session();
170         } while (ret == SGX_ERROR_BUSY && busy_retry_times--);
171         if (ret != SGX_SUCCESS)
172                 return ret;
173         do
174         {
175                 ret = read_and_verify_monotonic_counter((
                      sgx_sealed_data_t*)sealed_mc_result,
176                                                         &
                                                            mc_unsealed
                                                            );
177                 if (ret != SGX_SUCCESS)
178                         break;
179
180                 memcpy(&mc2seal, &mc_unsealed, sizeof(
                      monotonic_counter));
181
182                 ret = sgx_increment_monotonic_counter(&mc2seal.
                      mc,
183                                                       &mc2seal.
                                                          mc_value
                                                          );
184                 if (ret != SGX_SUCCESS)
185                 {
186                         switch (ret)
187                         {
188                         case SGX_ERROR_SERVICE_UNAVAILABLE:
189                                 /* Architecture Enclave Service
                                    Manager is not installed or
                                    not
190                                 working properly.*/
191                                 break;
192                         case SGX_ERROR_SERVICE_TIMEOUT:
193                                 /* retry the operation*/
194                                 break;
195                         case SGX_ERROR_BUSY:
196                                 /* retry the operation later*/
197                                 break;
```

```
198                         case SGX_ERROR_MC_NOT_FOUND:
199                                 /* The Monotonic Counter was
                                       deleted or invalidated.
200                                    This might happen under
                                       certain conditions.
201                                    For example, the Monotonic
                                       Counter has been deleted,
                                       the SGX
202                                    Platform Service lost its
                                       data or the system is
                                       under attack. */
203                                 break;
204                         case SGX_ERROR_MC_NO_ACCESS_RIGHT:
205                                 /* The Monotonic Counter is not
                                       accessible by this enclave.
206                                    This might happen under
                                       certain conditions.
207                                    For example, the SGX Platform
                                        Service lost its data or
                                       the
208                                    system is under attack. */
209                                 break;
210                         default:
211                                 /*other errors*/
212                                 break;
213                         }
214                         break;
215                 }
216
217                 /* If the counter value returns doesn't match
                       the expected value,
218                    some other entity has updated the counter,
                          for example, another instance
219                    of this enclave. The system might be under
                       attack */
220                 if (mc2seal.mc_value != mc_unsealed.mc_value +
                       1)
221                 {
222                         ret = 0x002; //error
223                         break;
224                 }
225
226                 /* seal the incremented mc */
227                 ret = sgx_seal_data(0, NULL, sizeof(mc2seal), (
                       uint8_t*)&mc2seal,
228                                     sealed_mc_size, (
                                        sgx_sealed_data_t*)
                                        sealed_mc_result);
229         } while (0);
230
```

```
231         /* remember to clear secret data after been used by
               memset_s */
232         memset_s(&mc_unsealed, sizeof(monotonic_counter), 0,
233                  sizeof(monotonic_counter));
234
235         /* remember to clear secret data after been used by
               memset_s */
236         memset_s(&mc2seal, sizeof(monotonic_counter), 0,
237                  sizeof(monotonic_counter));
238         sgx_close_pse_session();
239         return ret;
240 }
241
242 uint32_t read_sealed_monotonic_counter(uint8_t* sealed_mc_result
       , uint32_t sealed_mc_size, uint32_t* mc_value)
243 {
244         uint32_t ret = 0;
245         int busy_retry_times = 2;
246         monotonic_counter mc_unsealed;
247         monotonic_counter mc2seal;
248         do {
249                 ret = sgx_create_pse_session();
250         } while (ret == SGX_ERROR_BUSY && busy_retry_times--);
251         if (ret != SGX_SUCCESS) {
252                 return ret; // todo: Clear mem
253         }
254
255         ret = read_and_verify_monotonic_counter((
               sgx_sealed_data_t*)sealed_mc_result,
256                                                 &mc_unsealed);
257
258         *mc_value = mc_unsealed.mc_value;
259
260         /* remember to clear secret data after been used by
               memset_s */
261         memset_s(&mc_unsealed, sizeof(monotonic_counter), 0,
262                  sizeof(monotonic_counter));
263
264         /* remember to clear secret data after been used by
               memset_s */
265         memset_s(&mc2seal, sizeof(monotonic_counter), 0,
266                  sizeof(monotonic_counter));
267         sgx_close_pse_session();
268         return ret;
269 }
```

Snippet 6.1: Δημιουργία monotonic counter στο Enclave.cpp

Βιβλιογραφία

[1] *IBM blockchain*. http://www.ibm.com/blockchain/. Accessed: 2018-09-15.

[2] Miguel Castro and Barbara Liskov. *Practical Byzantine Fault Tolerance*. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[3] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. *Efficient Byzantine Fault Tolerance*. URL: https://ieeexplore.ieee.org/document/6081855/.

[4] Gopinath Nirmala and Rakesh. *Improving the Security and Efficiency of Blockchain-based Cryptocurrencies*. URL: https://aaltodoc.aalto.fi/handle/123456789/27919.

[5] João Sousa, Alysson Bessani, and Marko Vukolić. *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform*. URL: https://arxiv.org/abs/1709.06921.

[6] Johannes Behl, Tobias Distler, and Rudiger Kapitza. *Hybrids on Steroids: SGX-Based High Performance BFT*. URL: https://www4.cs.fau.de/Publications/2017/behl_17_eurosys.pdf.

[7] J. Behl, T. Distler, and R. Kapitza. *Consensus-oriented parallelization: How to earn your first million.* URL: https://www4.cs.fau.de/Publications/2015/behl_15_mw.pdf.

[8] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schroder-Preikschat, and K. Stenge. *CheapBFT: resource-efficient byzantine fault tolerance*. URL: https://dl.acm.org/citation.cfm?id=2168866.

[9] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. *Scalable Byzantine Consensus via Hardware-assisted Secret Sharing*. URL: https://arxiv.org/pdf/1612.04997v4.pdf.

[10] *The source code used in Efficient Byzantine Fault Tolerance paper*. http://www.gsd.inesc-id.pt/~mpc/software/minimal/index.html. Accessed: 2018-06-21.

[11] *Intel(R) Software Guard Extensions for Linux* OS*. https://github.com/intel/linux-sgx. Accessed: 2018-06-21.

[12] *Intel(R) Software Guard Extensions Developer Guide*. https://software.intel.com/en-us/documentation/sgx-developer-guide. Accessed: 2018-06-21.

[13] Joao Sousa Alysson Bessani and Eduardo Alchieri. *State Machine Replication for the Masses with BFT-SMaRt*. URL: http://hdl.handle.net/10451/14170.