NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Provably Secure, Smart Contract-based Naming Services: Design, Implementation and Evaluation

Christos F. Patsonakis

ATHENS

OCTOBER 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αποδεδειγμένα Ασφαλείς Υπηρεσίες Ονοματοδοσίας Βασισμένες σε Έξυπνα Συμβόλαια: Σχεδίαση, Υλοποιήση και Αξιολόγηση

Χρήστος Φ. Πατσωνάκης

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2019

PhD THESIS

Provably Secure, Smart Contract-based Naming Services: Design, Implementation and Evaluation

Christos F. Patsonakis

SUPERVISOR: Mema Roussopoulos, Associate Professor EKPA

THREE-MEMBER ADVISORY COMMITTEE:

Mema Roussopoulos, Associate Professor EKPA

Aggelos Kiayias, Associate Professor EKPA

Alex Delis, Professor EKPA

SEVEN-MEMBER EXAMINATION COMMITTEE

Mema Roussopoulos,
Associate Professor EKPA

Aggelos Kiayias,
Associate Professor EKPA

Alex Delis,
Professor EKPA

Eystathios Xatzieythimiadis,
Professor EKPA

Alexandros Ntoulas,
Assistant Professor EKPA

Yannis Smaragdakis,
Professor EKPA

Konstantinos Chatzikokolakis,
Associate Professor EKPA

Examination Date: October 31, 2019

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αποδεδειγμένα Ασφαλείς Υπηρεσίες Ονοματοδοσίας Βασισμένες σε Έξυπνα Συμβόλαια: Σχεδίαση, Υλοποιήση και Αξιολόγηση

Χρήστος Φ. Πατσωνάκης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Άγγελος Κιαγιάς, Αναπληρωτής Καθηγητής ΕΚΠΑ

Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Μέμα Ρουσσοπούλου,
Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Άγγελος Κιαγιάς,
Αναπληρωτής Καθηγητής ΕΚΠΑ

Αλέξης Δελής,
Καθηγητής ΕΚΠΑ

Ευστάθιος Χατζηευθυμιάδης,
Καθηγητής ΕΚΠΑ

Αλέξανδρος Ντούλας,
Επίκουρος Καθηγητής ΕΚΠΑ

Ιωάννης Σμαραγδάκης,
Καθηγητής ΕΚΠΑ

Κωνσταντίνος Χατζηκοκολάκης,
Αναπληρωτής Καθηγητής ΕΚΠΑ

Ημερομηνία Εξέτασης: 31 Οκτωβρίου 2019

ABSTRACT

Naming services provide the necessary foundations of developing diverse and important applications, such as e-commerce and e-banking. Currently, these naming services are operated by centralized authorities, which have to be trusted for their correct operation. Unfortunately, centralization (of trust) incurs several downsides in terms of security, availability and fault tolerance, as illustrated by numerous security incidents throughout the years where such authorities have been compromised. Decentralization has been proposed as an alternative to deal with these issues. Nevertheless, decentralization raises other concerns, such as dealing with free-riding and Sybil attacks.

In this thesis, we leverage the scalability, security, as well as, the built-in incentive mechanism of blockchain systems and propose the design of a decentralized, smart contract-based naming service. More specifically, we are the first to fully formalize the naming service design problem in the Universal Composability (UC) framework and formally prove the security of our construction under the strong RSA assumption in the Random Oracle model and the existence of an ideal smart contract functionality. The main barrier in realizing a smart contract-based naming service is the size of the contract's state which, being its most expensive resource to access and modify, should be minimized for a construction to be viable. We resolve this issue by defining and using in our naming service a public-state cryptographic accumulator with constant size, a cryptographic tool which may be of independent interest in the context of blockchain protocols. This accumulator incurs constant-sized storage at the expense of computational complexity. To explore this tradeoff, we propose and implement a second construction, which preserves the security properties of the first and, as illustrated through our evaluation, is the only version with constant-sized state that can be deployed on the live chain of Ethereum, the most notable public smart contract platform at the time of this writing. We compare these two constructions with the simple approach of most prior works, e.g., the Ethereum Name Service, where all identity records are stored on the smart contract's state, to illustrate several shortcomings of Ethereum and its cost model. To address these issues, and others, we introduce an alternative paradigm for developing smart contract-based applications in which their state is of constant size and facilitates the verification of application data that are stored to and queried from an external, potentially unreliable, storage network. This approach is relevant for a wide range of applications, such as any key-value store. We illustrate the efficacy of our approach by presenting a case study where we adapt the most widely deployed standard for fungible tokens, i.e., the ERC20 token standard, to our paradigm. We address Ethereum's monotonically increasing state which, if left unchecked, will have a direct impact on Ethereum's security and, ultimately, its longevity. We introduce recurring fees that are proportional to the state of smart contracts and adjustable by the nodes (miners) that maintain the network. We propose a scheme where the cost of storage-related operations reflects the effort that miners have to expend to execute them. We show that under such a pricing scheme that encourages economy in the state consumed by smart

contracts, the constructions presented in this work reduce the incurred transaction fees by up to an order of magnitude. We argue that these improvements are sensible for any smart contract platform that wishes to support user developed distributed applications.

SUBJECT AREA: Distributed Systems

KEYWORDS: public key infrastructure, smart contract, cryptographic accumulator

ΠΕΡΙΛΗΨΗ

Οι υπηρεσίες ονοματοδοσίας παρέχουν τα απαραίτητα θεμέλια για την ανάπτυξη ποικίλων και σημαντικών εφαρμογών, όπως το ηλεκτρονικό εμπόριο και η ηλεκτρονική τραπεζική. Επί του παρόντος, αυτές οι υπηρεσίες ονοματοδοσίας βρίσκονται υπό τον έλεγχο κεντρικοποιημένων οντοτήτων, τις οποίες πρέπει να εμπιστευόμαστε ότι λειτουργούν σωστά. Δυστυχώς, η κεντρικοποίηση (εμπιστοσύνης) επιφέρει πολλά μειονεκτήματα όσον αφορά την ασφάλεια, τη διαθεσιμότητα και την ανοχή σφαλμάτων, όπως φαίνεται από μία πληθώρα περιστατικών ασφάλειας κατά τη διάρκεια των ετών όπου τέτοιες οντότητες έχουν παραβιαστεί. Η αποκέντρωση έχει προταθεί ως εναλλακτική λύση για την αντιμετώπιση αυτών των ζητημάτων. Παρ ’όλα αυτά, η αποκέντρωση εγείρει άλλα προβλήματα όπως, π.χ., η αντιμετώπιση της μη ανταποδοτικότητας και οι Σιβυλλικές επιθέσεις.

Σε αυτή τη διατριβή, αξιοποιούμε την επεκτασιμότητα, την ασφάλεια, καθώς και τον ενσωματωμένο μηχανισμό παροχής κινήτρων των συστημάτων blockchain και προτείνουμε τον σχεδιασμό μιας αποκεντρωμένης υπηρεσίας ονοματοδοσίας βασισμένη σε έξυπνα συμβόλαια. Πιο συγκεκριμένα, είμαστε οι πρώτοι που παρουσιάζουμε τον πλήρη φορμαλισμό του προβλήματος σχεδιασμού υπηρεσιών ονοματοδοσίας στο πλαίσιο του μοντέλου Γενικής Σύνθεσης και αποδεικνύουμε την ασφάλεια της κατασκευής μας υπό την ισχυρή υπόθεση RSA στο μοντέλο του Τυχαίου Μαντείου και την ύπαρξη μιας ιδεατής λειτουργικότητας έξυπνου συμβολαίου. Το κύριο εμπόδιο στην πραγματοποίηση μιας υπηρεσίας ονοματοδοσίας βασισμένη σε έξυπνα συμβόλαια είναι το μέγεθος της αποθηκευμένης πληροφορίας σε αυτά η οποία, όντας η πιο δαπανηρή πηγή πρόσβασης και τροποποίησης, θα πρέπει να ελαχιστοποιηθεί για να θεωρηθεί μια κατασκευή βιώσιμη. Επιλύουμε αυτό το ζήτημα ορίζοντας και χρησιμοποιώντας στην υπηρεσία ονοματοδοσίας μας έναν προσθετικό, παγκόσμιο κρυπτογραφικό συσσωρευτή δημόσιας κατάστασης σταθερού μεγέθους, ένα κρυπτογραφικό εργαλείο το οποίο μπορεί να είναι ανεξάρτητου ενδιαφέροντος στο πλαίσιο των πρωτοκόλλων blockchain. Αυτός ο συσσωρευτής προκαλεί αποθήκευση σταθερού μεγέθους πληροφορίας εις βάρος υπολογιστικής πολυπλοκότητας. Για να διερευνήσουμε το αντίκτυπο ανάμεσα σε αυτά τα δύο, προτείνουμε και υλοποιούμε μια δεύτερη κατασκευή, η οποία διατηρεί τις ιδιότητες ασφαλείας της πρώτης και, όπως απεικονίζεται μέσα από την αξιολόγησή μας, είναι η μόνη έκδοση με σταθερού μεγέθους αποθηκευμένη πληροφορία που μπορεί να αναπτυχθεί στη βασική αλυσίδα του Ethereum, της πιο αξιοσημείωτης δημόσιας πλατφόρμας έξυπνων συμβολαίων κατα τη στιγμή αυτής της γραφής. Συγκρίνουμε αυτές τις δύο κατασκευές με την απλή προσέγγιση των περισσότερων προηγούμενων υλοποιήσεων, π.χ., του Ethereum Name Service, όπου όλα τα αρχεία ταυτότητας αποθηκεύονται πάνω στο έξυπνο συμβόλαιο, για να καταδείξουμε αρκετές ελλείψεις του Ethereum και του μοντέλου κοστολόγησής του. Για την αντιμετώπιση αυτών των ζητημάτων, καθώς και άλλων, εισαγάγουμε ένα εναλλακτικό παράδειγμα για την ανάπτυξη εφαρμογών βασισμένες σε έξυπνα συμβόλαια στις οποίες το μέθεγος της αποθηκευμένης πληροφορίας σε αυτά είναι σταθερή και διευκολύνει την επαλήθευση των δεδομένων των εφαρμογών, τα οποία αποθηκεύονται σε και αναζητούνται από ένα εξω-

τερικό, δυνητικά αναξιόπιστο, δίκτυο αποθήκευσης. Αυτή η προσέγγιση είναι σχετική για ένα ευρύ φάσμα εφαρμογών, όπως κάθε σύστημα αποθήκευσης κλειδιών και τιμών. Δείχνουμε την αποτελεσματικότητα της προσέγγιση μας με την παρουσίαση μιας μελέτης όπου προσαρμόζουμε το πιο ευρέως αναπτυγμένο πρότυπο για ανταλλάξιμα νομίσματα, δηλ., το πρότυπο νομισμάτων ERC20. Αντιμετωπίζουμε τη μονοτονικά αυξανόμενη αποθηκευμένη πληροφορία του Ethereum η οποία, αν δεν ελεγχθεί, θα έχει άμεσο αντίκτυπο στην ασφάλεια του Ethereum και, τελικά, στη μακροζωία του. Εισαγάγουμε επαναλαμβανόμενα τέλη που είναι ανάλογα με την αποθηκευμένη πληροφορία στα έξυπνα συμβόλαια και ρυθμιζόμενα από τους κόμβους που διατηρούν το δίκτυο. Προτείνουμε ένα μοντέλο όπου το κόστος των λειτουργιών αποθήκευσης αντικατοπτρίζει την προσπάθεια που πρέπει να καταβάλουν οι κόμβοι για να τις εκτελέσουν. Δείχνουμε ότι κάτω από ένα τέτοιο σύστημα τιμολόγησης που ενθαρρύνει οικονομία στην αποθηκευμένη πληροφορία στα έξυπνα συμβόλαια, οι κατασκευές που παρουσιάζονται σε αυτή τη διατριβή μειώνουν τα τέλη συναλλαγών κατά μία τάξη μεγέθους. Υποστηρίζουμε ότι αυτές οι βελτιώσεις είναι λογικές για κάθε πλατφόρμα έξυπνων συμβολαίων που επιθυμεί να υποστηρίζει την ανάπτυξη αυθαίρετων κατανεμημένων εφαρμογών από τους χρήστες της.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: υποδομή δημοσίου κλειδιού, έξυπνο συμβόλαιο, κρυπτογραφικός συσσωρευτής

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

1. Εισαγωγή στη Διατριβή

Σε αντίθεση με τις αρχικές, καθαρές αρχές σχεδιασμού του, το Διαδίκτυο σήμερα δεν είναι απολύτως αποκεντρομένο. Οι υπηρεσίες ονοματοδοσίας του Διαδικτύου, όπως τα συστήματα ονοματοδοσίας τομέα και οι υποδομές δημοσίου κλειδιού, παρέχουν τους πιο κρίσημους δομικούς λίθους που επιτρέπουν και ασφαλίζουν, αντιστοίχως, ψηφιακές επικοινωνίες. Αυτές οι υπηρεσίες διαχειρίζονται αντιστοιχίες ανάμεσα σε ονόματα οντοτήτων και τιμές (π.χ., μια διεύθυνση IP για τα συστήματα ονοματοδοσίας τομέα, ή ένα δημόσιο κλειδί για τις υποδομές δημοσίου κλειδιού). Δυστυχώς, αυτές οι κρίσιμες υπηρεσίες βρίσκονται υπό τον έλεγχο κεντρικοποιημένων, απομακρυσμένων οντοτήτων τις οποίες πρέπει να εμπιστευτούμε για την σωστή τους λειτουργία. Αυτό είναι προβληματικό όπως έχει αποδειχθεί από τα πολλαπλά περιστατικά ασφαλείας σε, π.χ., κεντρικοποιημένες υποδομές δημοσιού κλειδιού, όπου οι αρχές πιστοποίησης έχουν παραβιαστεί (π.χ., [63, 71, 82]).

Η έλευση του Bitcoin έφερε μία επανάσταση στον κόσμο των ψηφιακών πληρωμών αφού ήταν το πρώτο σύστημα που επέτρεπε σε οντότητες που δεν εμπιστεύονται η μία την άλλη να προβούν σε ασφαλείς συναλλαγές χωρίς να βασίζονται σε τρίτες, έμπιστες οντότητες. Η λειτουργία του Bitcoin βασίζετε σε ένα κατανεμημένο δίκτυο ομότιμων οντοτήτων με ανοιχτή πρόσβαση το οποίο διατηρεί ένα υψηλά αναπαράξιμο, ελέγξιμο και αποκλειστικά προσαρτίσιμο αρχείο καταγραφής συναλλαγών, το οποίο αναφέρεται συνήθως ως “αλυσίδα από μπλοκ” (blockchain). Η τεχνολογία αυτή είναι πολλά υποσχόμενη μιας και επιτρέπει την κατασκευή συστημάτων τα οποία είναι απόλυτα κατανεμημένα. Ως αποτέλεσμα αυτών των δυνατοτήτων, έχουν υπάρξει πολλαπλές και επαναλαμβανόμενες κλήσεις από την κοινότητα για την εκ νέου αποκέντρωση του Διαδυκτίου χρησιμοποιώντας τεχνολογίες blockchain για την ανάπτυξη αυτών των κρίσιμων υπηρεσιών ονοματοδοσίας, εξαλίφωντας έτσι την εξάρτηση του Διαδυκτίου σε κεντρικοποιημένες οντότητες.

Το Namecoin ([24]) και το Emercoin ([8]) αποτελούν αξιοσημείωτα παραδείγματα συστημάτων ονοματοδοσίας που βασίζονται σε τεχνολογίες blockchain. Αυτά τα συστήματα χρησιμοποιούν το blockchain για να αποθηκεύσουν, επικυρώσουν και αναζητήσουν εγγραφές που σχετίζονται με ονόματα οντοτήτων. Ωστόσο, αυτή η προσέγγιση δεν είναι αποδοτική για τους ακόλουθους λόγους. Πρώτον, εξαναγκάζει τους πελάτες να κατεβάσουν και να διατηρήσουν ένα ολοκληρωμένο αντίγραφο του blockchain για να έχουν την δυνατότητα να επικυρώσουν εγγραφές. Δεύτερον, η υπολογιστική και η αποθηκευτική πολυπλοκότητα αυξάνεται γραμμικά με το πλήθος των καταχωρημένων εγγραφών. Τρίτον, η εφαρμοσιμότητα του συστήματος περιορίζεται αισθητά μιας και αποκλείονται σημαντικές συσκευές λόγω της περιορισμένης αποθήκευσής τους όπως, π.χ., τα έξυπνα τηλέφωνα (smartphones). Τέλος, αυξάνει το μέγεθος της πληροφορίας που πρέπει να διατηρούν οι κόμβοι του δικτύου, το οποίο ενδεχομένως να αποτρέψει νέους κόμβους από το να συγχρονιστούν και να προσφέρουν στην ασφάλεια του δικτύου.

Σε αυτή τη διατριβή, παρουσιάζουμε τον σχεδιασμό, την υλοποίηση και την αξιολόγηση της πρώτης αποδεδειγμένα ασφαλούς, πλήρως κατανεμημένης υπηρεσίας ονοματοδοσίας. Οι δομικοί λίθοι της υπηρεσίας μας είναι τα έξυπνα συμβόλαια και οι κρυπτογραφικοί συσσωρευτές. Παρουσιάζουμε δύο κατασκευές της υπηρεσίας μας οι οποίες αποθηκεύουν πληροφορία σταθερού μεγέθους στο έξυπνο συμβόλαιο, ανεξάρτητα του πλήθους των εγεγγραμένων οντοτήτων. Με αυτόν τον τρόπο, επιλύουμε πλήρως τα προαναφερθέντα προβλήματα που σχετίζονται με την άμεση αποθήκευση εγγραφών στο blockchain. Η αποθήκευση εγγραφών γίνεται σε ένα εξωτερικό, δυνητικά αναξιόπιστο, κατανεμημένο δίκτυο αποθήκευσης το οποίο, ανάμεσα σε άλλα, επιτρέπει πιο αποδοτική πρόσβαση στις εγγραφές.

Συνοπτικά, οι συνεισφορές της διατριβής αυτής έχουν ως εξής:

- Παρουσιάζουμε τον επίσημο σχεδιασμό μιας υπηρεσίας ονοματοδοσίας βασισμένης σε έξυπνα συμβόλαια.
- Επιλύουμε το βασικό εμπόδιο για την πραγματοποίηση μιας βιώσημης υπηρεσίας ονοματοδοσίας βασισμένης σε έξυπνα συμβόλαια παρουσιάζοντας δύο κατασκευές που αποθηκεύουν σταθερού μεγέθους πληροφορίας στο έξυπνο συμβόλαιο.
- Είμαστε οι πρώτοι που επίσημα αποδεικνύουμε την ασφάλεια μιας υπηρεσίας ονοματοδοσίας βασισμένης σε έξυπνα συμβόλαιο στο πλαίσιο του μοντέλου Γενικής Σύνθεσης, κάτω από θεμελιωμένα κρυπτογραφικά προβλήματα και την ύπαρξη μιας ιδεατής λειτουργικότητας έξυπνων συμβολαίων.
- Υλοποιούμε και πειραματικά αξιολογούμε τις κατασκευές μας, καθώς και τους δομικούς τους λίθους.
- Εισαγάγουμε ένα εναλλακτικό παράδειγμα για την ανάπτυξη εφαρμογών σε πλατφόρμες έξυπνων συμβολαίων.
- Επιλύουμε το πρόβλημα της μονοτονικά αυξανόμενης πληροφορίας που διατηρεί το Ethereum με την εισαγωγή ενός προσαρμοστικού, από τους κόμβους του δικτύου, μοντέλου κοστολόγησης το οποίο ενθαρρύνει οικονομία στον καταναλισκόμενο χώρο αποθήκευσης από τα έξυπνα συμβόλαια.

2. Υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζουμαι συνοπτικά έννοιες που μορφοποιούν το υπόβαθρο για την κατανόηση των υπόλοιπων κεφαλαίων, με στόχο να καταστεί η διατριβή όσο το δυνατόν πιο αυτόνομη. Πιο συγκεκριμένα, παρουσιάζουμε βασικές έννοιες σχετικά με κρυπτοσυστήματα δημοσίου κλειδιού, ψηφιακά πιστοποιητικά, καθώς και τις υποδομές δημοσίου κλειδιού. Εν συνεχεία, περιγράφουμε την λειτουργία των συστημάτων blockchain, που αποτελούν τον δομικό λίθο πάνω στην οποία χτίστηκε η πλατφόρμα έξυπνων συμβολαίων Ethereum πάνω στην οποία υλοποιήσαμε όλες τις κατασκευές που παρουσιάζονται σε αυτή τη διατριβή. Τέλος, παρουσιάζουμε μία εισαγωγή για τους κρυπτογραφικούς συσσωρευτές και τις ιδιότητες τους.

3. Σχετική Βιβλιογραφία

Στο κεφάλαιο αυτό παρουσιάζουμε τα αποτελέσματα της μέχρι τωρα βιβλιογραφίας σχετικά με τις υπηρεσίες ονοματοδοσίας, καθώς και τις υποδομές δημοσίου κλειδιού. Βασισμένοι στο γεγόνος ότι αρκετές προηγούμενες προσεγγίσεις χρησιμοποιούν το ίδιο υπόβαθρο για την ανάπτυξη τέτοιων υπηρεσιών, τις κατηγοριοποιούμε βάση αυτών. Συνοπτικά, εξετάζουμε προηγούμενες κατασκευές που βασίζονται σε: 1) μηχανές αναπαραγόμενης κατάστασης, 2) διαφόρων ειδών δικτύων επικάλυψης και, 3) blockchains. Μέσω της βιβλιογραφικής μας έρευνας παρουσιάζουμε τις αδυναμίες και αστοχίες που έχει τόσο κάθε κατηγορία, όσο και κάθε σύστημα ατομικά. Παρατηρούμε ότι καμία προηγούμενη προσέγγιση δεν αντιμετωπίζει, συνολικά, θέματα όπως η χρονική σήμανση εγγραφών, η ανοχή αυθαίρετων σφαλμάτων, η αντιμετώπιση της μη ανταποδοτικότητας, ανοσία στις Σιβυλλικές επιθέσεις, επίσημες αποδείξεις ασφαλείας και άλλα.

4. Υπηρεσία Ονοματοδοσίας: Δομικοί Λίθοι και Ορισμός

Στο κεφάλαιο αυτό ξεκινάμε εισαγάγωντας θέματα σχετικά με την σημειογραφία που χρησιμοποιούμε σε αυτήν την διατριβή. Επίσης, παραθέτουμε επίσημα βασικούς ορισμούς θεμελιωμένων κρυπτογραφικών προβλημάτων, όπως το ισχυρό RSA και οι ανθεκτικές σε συγκρούσεις κρυπτογραφικές συναρτήσεις κατακερματισμού. Εν συνεχεία, παρουσιάζουμε τον πρώτο επίσημο ορισμό ενός προσθετικού, παγκόσμιου κρυπτογραφικού συσσωρευτή δημόσιας κατάστασης, ο οποίος αποτελεί τον δομικό λίθο για τις κατασκευές της υπηρεσίας ονοματοδοσίας μας. Τέλος, παρουσιάζουμε τον επίσημο ορισμό της υπηρεσίας ονοματοδοσίας μας, στο πλαίσιο του μοντέλου Γενικής Σύνθεσης, την οποία μοντελοποιούμε σαν μία ιδεατή λειτουργικότητα. Επίσης, παρουσιάζουμε τον επίσημο ορισμό δύο ακόμα ιδεατών λειτουργικοτήτων που μοντελοποιούν το (αναξιόπιστο) δίκτυο αποθήκευσης εγγραφών και το έξυπνο συμβόλαιο, αντιστοίχως. Τέλος, παραθέτουμε μία γενικευμένη περιγραφή σχετικά με το πως αλληλεπιδρούν οι πελάτες με όλες τις προαναφερθούσες λειτουργικότητες στα πρωτόκολλα μας, τα οποία τα παρουσιάζουμε στα επόμενα κεφάλαια.

5. Κατασκευή PKI βασισμένη στο RSA

Στο κεφάλαιο αυτό παρουσιάζουμε την πρώτη κατασκευή της υπηρεσίας ονοματοδοσίας μας. Ξεκινάμε παρουσιάζοντας μία κατασκευή ενός προσθετικού, παγκόσμιου κρυπτογραφικού συσσωρευτή δημόσιας κατάστασης του οποίου η ασφάλεια βασίζετε στο ισχυρό RSA στο μοντέλο του Τυχαίου Μαντείου. Το πεδίο ορισμού αυτού του συσσωρευτή είναι περιορισμένο σε πρώτους αριθμούς. Ωστόσο, η υπηρεσία ονοματοδοσία μας απαιτεί την συσσώρευση αυθαίρετων ακολουθιών χαρακτήρων. Επιλύουμε αυτό το πρόβλημα παρουσιάζοντας μία συνάρτηση πολυονημικής πολυπλοκότητας που αντιστοιχίζει αυθαίρετες ακολουθίες χαρακτήρων σε πρώτους αριθμούς. Αυτή η συνάρτηση είναι μία ντετερ-

μινιστική εκδοχή αυτής των Gennaro et al. [68]. Αποδεικνύουμε την ανοχή συγκρούσεων αυτής της συνάρτησης και πως για κάθε είσοδο θα επιστρέψει έναν πρώτο αριθμό με συντρηπτική πιθανότητα επιτυχίας. Εν συνεχεία, παρουσιάζουμε την κατασκευή μίας υποδομής δημοσίου κλειδιού η οποία χρησιμοποιεί τους προαναφερθέντες δομικούς λίθους για να υλοποιήσει τις πράξεις της υπηρεσίας ονοματοδοσίας μας. Πιο συγκεκριμένα, παρουσιάζουμε τον κώδικα του έξυπνου συμβολαίου, καθώς και το πρωτόκολλο της κατασκευής μας. Τέλος, ορίζουμε το βασικό θεώρημα ασφάλειας αυτής της κατασκευής, το οποίο το αποδεικνύουμε στο παράρτημα της διατριβής.

6. Κατασκευή PKI βασισμένη σε Δέντρα Κατακερματισμού

Στο κεφάλαιο αυτό παρουσιάζουμε την δεύτερη κατασκευή της υπηρεσίας ονοματοδοσίας μας. Ξεκινάμε παρουσιάζοντας τον παγκόσμιο κρυπτογραφικό συσσωρευτή των Camacho et al. [53], ο οποίος βασίζεται σε δέντρα κατακερματισμού. Αυτός ο συσσωρευτής είναι *ισχυρός*, δηλαδή, παρέχει περισσότερες δυνατότητες σε σχέση με αυτόν που ορίσαμε στο προηγούμενο κεφάλαιο. Ωστόσο, σε αυτόν τον συσσωρευτή, οι δομές που χρησιμοποιούνται για να αποδείξουμε αν κάποιο στοιχείο είναι συσσωρευμένο ή οχι έχουν λογαριθμική πολυπλοκότητα, σε αντίθεση με τον προηγούμενο συσσωρευτή όπου αυτές οι δομές είναι σταθερού μεγέθους. Εν συνεχεία, παρουσιάζουμε την κατασκευή μίας υποδομής δημοσίου κλειδιού η οποία χρησιμοποιεί τον προαναφερθέντα συσσωρευτή για να υλοποιήσει τις πράξεις της υπηρεσίας ονοματοδοσίας μας. Αντίστοιχα με το προηγούμενο κεφάλαιο, παρουσιάζουμε τον κώδικα του έξυπνου συμβολαίου, το πρωτόκολλο της κατασκευής μας, καθώς και το βασικό θεώρημα ασφάλειας της.

7. Αξιολόγηση

Στο κεφάλαιο αυτό περιγράφουμε την διαδικασία και τα αποτελέσματα της αξιολόγησης της πρωτότυπης υλοποίησης των δύο κατασκευών της υπηρεσίας ονοματοδοσίας μας, καθώς και των δομικών τους λίθων. Επίσης, διασπείρουμε ανάμεσα στα αποτελέσματα μας μία σειρά απο παρατηρήσεις και βελτιώσεις, οι οποίες είναι μικρές και δίκαιές, και οι οποίες θα συνδράμουν αισθητά στη βελτίωση του Ethereum.

Η πρώτη σειρά πειραμάτων μας καταδεικνύει το επιπλέον κόστος που έχει μία υλοποίηση ενός σχήματος ψηφιακών υπογραφών σε ένα έξυπνο συμβόλαιο, σε σχέση με την υλοποίηση που βασίζεται σε προκατασκευασμένα συμβόλαια που διατείθεντε από την πλατφόρμα του Ethereum. Υπολογίζουμε ένα μέσο κόστος επιβάρυνσης δύο τάξεων μεγέθους.

Εν συνεχεία, παρουσιάζουμε τις αξιολογήσεις των κατασκευών δημοσιού κλειδιού. Παρουσιάζουμε την παραμετροποίηση της κάθε κατασκευής (π.χ., τιμές για την παράμετρο ασφάλειας) και το κόστος κάθε πράξης του έξυπνου συμβολαίου. Για κάθε κατασκευή, συζητάμε εκτενώς την σημασιολογία των αποτελεσμάτων και παραθέτουμε την καταλληλότητα εφαρμογής της. Συνοπτικά, για την πρώτη κατασκευή που βασίζετε στο ισχυρό

RSA, τα αποτελέσματα μας δείχνουν ότι δεν μπορεί να εφαρμοστεί στην βασική αλυσίδα του Ethereum, λόγω του υπέρογκου κόστους της διαδικασίας αντιστοιχίας αυθαίρετων ακολουθιών χαρακτήρων σε πρώτους αριθμούς. Ωστόσο, η δεύτερη κατασκευή, η οποία βασίζεται σε δέντρα κατακερματισμού, μπορεί να εφαρμοστεί στην βασική αλυσίδα του Ethereum για μετρίου μεγέθους υποδομές δημοσίου κλειδιού. Συνοδεύουμε την παρουσίαση των αποτελεσμάτων κάθε κατασκευής με μία συζήτηση όπου προτείνουμε μελλοντικές βελτιστοποιήσεις. Τέλος, υλοποιούμε και αξιολογούμε μία υποδομή δημοσίου κλειδιού όπου όλες οι εγγραφές αποθηκεύονται στο έξυπνο συμβόλαιο, η οποία είναι η προσέγγιση που ακολουθείτο από όλες τις προηγούμενες υπηρεσίες ονοματοδοσίας βασισμένες σε blockchain και έξυπνα συμβόλαια. Τα αποτελέσματα μας δείχνουν ότι, προς το παρών, αυτή η προσέγγιση είναι η πιο αποδοτική από άποψη κόστους. Ωστόσο, οι επιπτώσεις αυτής της προσέγγισης στο μέλλον και στην υγεία της πλατφόρμας είναι σοβαρές. Αυτό είναι ένα θερμό θέμα συζήτησης στην κοινότητα του Ethereum, η οποία εδώ και χρόνια αναζητά τρόπους να το αντιμετωπίσει. Ωστόσο, οι δικές μας κατασκευές εναρμονίζονται άψογα με το μέλλον και την υγεία αφού επιφέρουν σταθερό κόστος αποθήκευσης στο έξυπνο συμβόλαιο, ανεξάρτητα με το πλήθος των εγεγγραμένων οντοτήτων.

8. Εναλλακτική Μεθοδολογία για την Ανάπτυξη Εφαρμογών και την Τιμολόγηση Αποθήκευσης σε Πλατφόρμες Έξυπνων Συμβολαίων

Στο κεφάλαιο αυτό, εφορμώμενοι από την έρευνα και τα αποτελέσματα του προηγούμενου κεφαλαίου, παρουσιάζουμε την προσέγγιση μας για να αντιμετωπίσουμε όλα τα θέματα που πηγάζουν από την πρακτική του να χρησιμοποιείτε το blockchain ή το έξυπνο συμβόλαιο σαν μέσο άμεσης αποθήκευσης, επικύρωσης και αναζήτησης πληροφορίας. Εισαγάγουμε μία εναλλακτική μεθοδολογία για την ανάπτυξη εφαρμογών σε αυτές τις πλατφόρμες όπου ο αποθηκευτικός χώρος του έξυπνου συμβολαίου χρησιμοποιείτε μόνο για την επικύρωση των δεδομένων της εφαρμογής. Η αποθήκευση και η αναζήτηση δεδομένων μεταφέρεται σε ένα εξωτερικό, πιθανοτικά αναξιόπιστο, δίκτυο αποθήκευσης. Η μεθοδολογία μας είναι γενική και μπορεί να εφαρμοστεί για την υλοποίηση οποιουδήποτε συστήματος αποθήκευσης κλειδιών και τιμών. Για να υποστηρίξουμε την αποτελεσματικότητα και προσαρμοστικότητα της προσέγγισης μας, παρουσιάζουμε τον σχεδιασμό και την υλοποίηση μίας προσαρμογής του πρότυπου νομισμάτων ERC20. Αξιολογούμε την κατασκευή μας και την συγκρίνουμε με την κλασσική υλοποίηση.

Αντιμετωπίζουμε το πρόβλημα της μονοτονικά αυξανόμενης πληροφορίας που αποθηκεύεται στο Ethereum παρουσιάζοντας ένα προσαρμοστικό, από τους κόμβους του δικτύου, μοντέλο κοστολόγησης των πράξεων αποθήκευσης. Πιο συγκεκριμένα, εισαγάγουμε επαναλαμβανόμενες χρεώσεις αποθήκευσης ανάλογες με το καταναλισκόμενο χώρο αποθήκευσης κάθε έξυπνου συμβολαίου, κοστολογούμε τις πράξεις αποθήκευσης βάση της πολυπλοκότητας εκτέλεσης τους και απελευθερώνουμε αποθηκευτικό χώρο ο οποίος καταλαμβάνεται από συμβόλαια τα οποία δεν χρησιμοποιούνται πλέον. Υπό αυτό το μοντέλο κοστολόγησης, αξιολογούμε ξανά, τόσο την κατασκευή του ERC20 νομίσματος, όσο και την κατασκευή της υποδομής δημοσίου κλειδιού που βασίζεται σε δέντρα κατακερματισμού. Τα αποτελέσματά μας δείχνουν ότι και οι δύο κατασκευές παρέχουν βελτιώσεις που

μειώνουν το κόστος μέχρι μία τάξη μεγέθους.

9. Συμπεράσματα και Μελλοντικά Βήματα

Σε αυτή τη διατριβή παρουσιάσαμε τον σχεδιασμό και τις υλοποιήσεις μίας αποδεδειγμένα ασφαλούς υπηρεσίας ονοματοδοσίας βασισμένη σε έξυπνα συμβόλαια. Η υπηρεσία μας παρέχει ένα συνδιασμό ιδιοτήτων που καμία προηγούμενη προσέγγιση δεν έχει καταφέρει να προσφέρει. Εντούτοις, ενδιαφέροντα τεχνικά ζητήματα εξακολουθούν να παραμένουν προς διερεύνηση. Ενδεικτικά, αναφέρουμε τα παρακάτω:

- Σχεδιασμός, υλοποίηση και αξιολόγηση ενός κατανεμημένου δικτύου αποθήκευσης το οποίο είναι επεκτάσιμο και ανθεκτικό σε Σιβυλλικές επιθέσεις.
- Μείωση ή απαλειφή του υπολογιστικού φόρτου που μπορούν να επιβάλουν χαιρέκακοι κόμβοι του αποθηκευτικού δικτύου στους πελάτες της υπηρεσίας.
- Εφαρμογή και αξιολόγηση τεχνικών επαληθεύσιμου υπολογισμού για την συνάρτηση αντιστοίχησης αυθαίρετων ακολουθιών χαρακτήρων σε πρώτους αριθμούς.
- Επέκταση του μοντέλου ασφαλείας του κρυπτογραφικού συσσωρευτή των Camacho et al. [53] για την μείωση του κόστους της κατασκευής που βασίζεται σε δέντρα κατακερματισμού.

kms...

ACKNOWLEDGEMENTS

I am trying, to no avail however, to find the right words to express the amount of gratitude and respect that I have towards my thesis supervisor, Mema Roussopoulos. She was always present and provided me with firm encouragement and support, even in the (many) dark times that I had during my studies. I feel extremely fortunate for having had the opportunity to work under her guidance. From my experiences talking with many other PhD students from all around the world, I am convinced that she is truly a role model for supervisors. I know that when she reads this, she will probably ask me to tone down my praises towards her, but I will kindly decline!

I would also like to thank deeply the other members of my thesis committee. Professor Aggelos Kiayias, for his involvement and engagement in steering my work and providing the necessary insight in regards to cryptographic aspects that I was not accustomed with in the beginning of my studies. Professor Alex Delis, my mentor I could say, whom I have known and had a close connection with since my undergraduate studies. He is the one that fired up my interest in computer science and research as a whole. He provided me with strong motivation to pursue my graduate studies, while also scolding me for my smoking habbits! His guidance at crucial moments throughout the whole lifecycle of my studies cannot be overstated.

It has been a great pleasure to work at the Distributed Systems Research group. We worked on a variety of challenging and very interesting research problems throughout the years. I especially enjoyed the daily talks that we had, as they helped me grow both as a scientist and, more importantly, as a person. For similar reasons, I would also like to thank the members of the CRYPTO.SEC group, especially Katerina Samari, with whom I worked closely.

I am grateful to my parents for their support over the years and for making an absolutely astounding job at raising me, in spite of all the troubles that I put them through. My mother was always willing to make my life easier and ease my struggles in any possible way, be it by offering kind words of support, or a home cooked meal, the importance of which I cannot even begin to describe. My father, a fellow computer scientist, who is largely responsible for my interest in computers and programming, as he gave me and my brother an Amiga 500 as a present when I was 3 years old!

Last, but not the least, I would like to thank my friends (the ones that lasted). You know who you are.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The findings in this PhD thesis reflect the author's contribution to the goals of the "Protecting and Preserving Human Knowledge for Posterity" (PPP) project, which was funded under Grant Agreement No. 279237. The main goal of this project was the design and development of a secure and decentralized digital preservation system with no single points of failure that could be applied at an international scale. The project's research activities involved work in the following topics:

1. Resilience to adversaries, especially in the face of dynamic subversion and repair of peers.
2. System recovery after intrusions.
3. Peer identity, trust, and reputation.
4. Peer diversity.
5. Peer-to-peer preservation in other problem domains.

The author was involved in the design and development of topics 1 and 2 and had a leading role in the works pertaining to topic 3. The constructions presented in this work have provided a generic, secure and decentralized framework for handling peer identities. This was previously handled in a centralized fashion by manually binding a peer's identity to its (static) IP address. Thus, there was no support for the use of more involved and important constructs, e.g., public key cryptography, issues which are completely addressed by the findings presented of this thesis.

1. INTRODUCTION

The importance of associating human readable names with the addresses of resources that span multiple hosts and administrative domains dates back to the early days of the ARPANET. This association provides the following benefits. First, it diminishes the difficulty of humans, who interface with computer systems, to access resources by relying on memorable names, instead of, e.g., strings of numbers, such as IP addresses. Second, it adds a level of indirection which, among others, allows for transparent updates to the addresses of resources. In the early days of the Internet, the Stanford Research Institute maintained a mapping between domain names and IP addresses in a text file called *hosts.txt* ([62]). However, as the networking environment became more and more diverse, and as the number of resources grew, so did the need to formulate a standardized and automated Naming Service (NS) that will address technical and personnel issues ([84, 85]) in maintaing these associations. At a high level, a NS allows the *registration*, *revocation*, *storage* and *retrieval* of bindings between arbitrary names and addresses.

In today's Internet, Domain Name System (DNSs) and Public Key Infrastructures (PKIs) are NSs that provide the most critical building blocks for facilitating and securing digital communications. Their wide-scale deployment has allowed the development of an important and diverse set of applications, such as e-commerce, e-voting and Internet banking. In DNS, domain names are associated with *zone files* ([86]), which are sequences of *resource records* pertaining to, e.g., IP addresses and textual representations of timing and expiration parameters. In PKIs, identity names are associated with *digital certificates* ([73]), i.e., unforgeable data structures that attest, among others, to the authenticity of an identity's public key. However, contrary to the original, distributed design principles of the Internet, these critical systems remain, to this day, under the control of centralized, remote parties. In the following, we briefly overview the operations of a PKI in a centralized setting, which will allow us to, subsequently, illustrate the issues that centralization of NSs introduces.

In a centralized PKI (CPKI), a Certification Authority (CA), is responsible for issuing, distributing and managing the status of digital certificates. At a high level, the correct operation of CPKIs depends on the following two fundamental assumptions. First, everyone knows the CA's (correct) public key, i.e., all involved parties have a copy of the CA's certificate. Second, statements signed by the CA's private key are considered valid, i.e., everyone trusts the CA. In a CPKI, registering an identity to public-key mapping is a two-phase process (from hereon in, we will use the notation (id, pk) to refer to such mappings). In the first phase, the user proves her claim on an identity, e.g., via a government-issued identity card or some other form of document, to a Registration Authority (RA). Assuming the RA validates the claim, it marks the digital certificate request as valid and forwards it to the CA. In the second phase, the user receives her digital certificate, which is signed by the CA's private key, thus, attesting to its validity. CAs *periodically* publish signed data structures that contain revoked certificates, e.g., a certificate revocation list (CRL, [73]). Distribution of certificate-related information is handled either by the CA (online CA), or, it is delegated to online, publicly accessible directories (offline CA).

While predominant in use, CPKIs have several shortcomings. First, a CA constitutes a single point of failure, both in terms of security and availability. Indeed, there have been several incidents throughout the years where CAs have been compromised. These incidents have had severe repercussions that, for instance, led to the issuance of false certificates for domain names of high-profile corporations, such as Google ([63]). Symantec, which represented more than 30% of the Internet's valid certificates in 2015, was discovered to have mis-issued more than 30,000 certificates ([71]). Furthermore, the Trustwave ([82]) incident reincited the growing concern that governments and private organizations are able to issue false certificates for surveillance, thus, violating the privacy of end-users ([112]). In practice, there exist multiple CAs, which are linked with well-defined, parent-child relationships, based on trust and other policies. The most notable example of this architecture is the SSL/TLS certificate chain. This hierarchical, tree-like, certification model is designed to increase the system's scalability and fault-tolerance. However, root, or even, subordinate CA compromises, such as the ones we mentioned previously, have proven to be catastrophic ([61]). Furthermore, Internet browser vendors reserve and enforce their right to "distrust root certificates present in the operating system's root certificate list" (e.g., [80]). Moreover, users must deal with issues regarding cross-domain certification policies, as well as, the *John Wilson* problem ([99]). The latter is an intractable name distribution problem and refers to how a user can disambiguate if the party she is communicating with is the intended "John Wilson" and not a namesake. Unfortunately, dealing with these issues requires access to information that is generally not publicly available ([61, 72]).

In a decentralized PKI (DPKI), multiple, independent nodes cooperate and deliver the same set of services, without relying on one, or more, trusted third parties (TTPs, e.g., CAs). DPKIs have been proposed because, as distributed systems, they have the potential to offer a number of desirable properties that CPKIs cannot offer, such as scalability, fault-tolerance, load balancing and availability. Researchers have proposed DPKIs based on various distributed primitives, such as distributed hash tables (DHTs) (e.g., [37]). To account for malicious nodes and provide increased security, they employ secret sharing, threshold and byzantine agreement protocols (e.g., [43, 59]). These techniques, while more complex to design and implement correctly, lead to systems that do not exhibit single points of failure. Unfortunately, prior DPKIs do not provide incentives for the participating nodes to ensure that the offered service remains available in the long term, e.g., they fail to address the *free-riding* problem ([76]). To illustrate the importance of incentives and their effect on the perpetuity of systems, note that CPKIs are deployed on large corporations whose operations, and profits, are fueled by real-world currency. Thus, despite multiple, prominent security incidents in which CAs have been involved, they are still in operation. We posit that we can mimic this is a decentralized setting by leveraging modern constructs, which we introduce below.

Bitcoin ([88]), the world's first cryptocurrency, revolutionized the world of digital payments by allowing untrusted entities to transact securely without relying on TTPs. Its operation is based on a distributed network of peers (miners) with open membership that maintains a highly replicated, auditable, append-only log of transactions, which is commonly referred

to as a blockchain. Blockchains solve the well studied problem of distributed consensus ([77]) in an open networking environment. They feature a reward mechanism that has empirically demonstrated it incentivizes miners to participate in the protocol. The rewards come in the form of a digital currency that compensates miners, thus, creating a counter-incentive to free-riding, while still retaining a highly scalable, free-entry system. Since the advent of Bitcoin, blockchains show promise for building systems that are completely distributed. As a result of this potential, there have been calls from the community to "re-decentralize" the Internet by leveraging blockchain technology to build critical NSs and, thus, eliminate the Internet's reliance on centralized entities (e.g., [40, 24]).

Notable examples of blockchain-based NSs are Namecoin ([24]), Emercoin ([8]) and Blockstack's BNS ([40]). These systems employ the blockchain as a medium to store, query and verify the validity of records pertaining to identities. However, "on-blockchain" storage is inefficient for several reasons. First, it forces clients to download and maintain an entire copy of the blockchain to verify the validity of identity records. Second, computational complexity and storage requirements scale linearly with the number of registered records. Third, it limits the system's applicability by excluding important, storage-limited devices, e.g., smartphones. Lastly, it bloats the blockchain, increasing the size of the state that miners have to maintain, which may not incentivize new miners to sync and contribute to the blockchain's security. These inefficiencies are, to a certain extent, attributed to the limited set of operations that the underlying blockchain exposes. For instance, Namecoin's scripting language forbids even simple arithmetic calculations, such as integer multiplication. Thus, one can argue that, until recently, on-blockchain storage was the only available solution of implementing a NS in the face of such a constrained development environment. However, recently, a new generation of programmable blockchains has emerged that allows the development of smart contracts ([101]). These are stateful agents that "live" in the blockchain and can execute arbitrary state transition functions. At the time of this writing, the most notable public smart contract platform is Ethereum ([106]). This ingenious technology provides us with the necessary means to develop distributed NSs that do not suffer from the issues stemming from on-blockchain storage.

In this thesis, we present the design of the first provably secure, smart contract-based NS in the Universal Composability (UC) framework ([55]). The main barrier in realizing a smart contract-based NS is the size of the smart contract's state which, being its most expensive resource to access and update, should be minimized for a construction to be considered viable. To address this issue, we harness the power of cryptographic accumulators, i.e., data structures that provide a succinct representation of a set of elements and allow for verifiable, (non) membership proofs (referred to as *witnesses*). More specifically, our design's main building block is a public-state, additive, universal accumulator, based on the strong-RSA assumption in the Random Oracle model, which we define in this work. This accumulator favors storage overhead at the expense of computational complexity necessary to achieve constant-sized state and witnesses. To explore this tradeoff, we propose a second construction, which is built on top of the Hash tree-based, universal accumulator of Camacho et al. [53]. In this construction, the smart contract's state is still of constant size, however, witness sizes and their verification complexity are logarithmic in the number

of registered identity records. We compare both of our constructions with the simple approach of most prior schemes, e.g., the Ethereum Name Service ([15]), where all identity records are stored on the smart contract's state. We implement and evaluate experimentally on top of Ethereum a PKI variant of all the three aforementioned smart contract-based NSs and illustrate the monetary costs of their operations and their building blocks, as well as, their viability in terms of deployment. Furthermore, via our experimentation, we are able to present several shortcomings of Ethereum's current cost model and propose several modifications, which are minor and fair. Finally, we identify several problem areas we encountered while developing on top of Ethereum and make concrete proposals for improving the platform with the aim of increasing both developer productivity and smart contract reliability. We argue that these improvements are sensible for any smart contract platform that wishes to support user developed distributed applications.

In summary, the contributions of this thesis are as follows:

- We present the formal design of a smart contract-based NS. Due to the interoperability of smart contracts, our design provides a generic naming mechanism that can be used to provide on-blockchain authentication, in the case of a PKI implementation, that, up to this point, was handled in an ad-hoc manner. Furthermore, the programmable nature of smart contract platforms allows us to evolve our implementations with more efficient primitives, when such become available, without the need for a fork in the blockchain, which is **not** the case for all prior specialized blockchains (e.g., [24, 8]). Even though our envisioned application was initially a PKI, we specifically modeled our design as a generic NS. Thus, our design can be ported to implement, efficiently, other services that reside in this paradigm, e.g., a distributed domain name system (DDNS).
- We resolve the main barrier in realizing a viable smart contract based NS by providing two constructions that have the "constant-ness" property, i.e., the smart contract's state, which is expensive to access and even more so to update, is of constant bit-size, regardless of the number of registered identity records.
- We are the first to formally prove the security of a smart contract-based NS ([92]) in the Universal Composability (UC) framework ([55]) under standard cryptographic hardness assumptions and the existence of an ideal smart contract functionality[1]. We believe that formal proofs of security are extremely important for such critical security infrastructures, an issue that was not addressed by **any** prior work.
- We implement and experimentally evaluate two PKI constructions of our NS, as well as, all of their building blocks ([93]), in Ethereum. Our results illustrate that one of our constructions can be deployed on Ethereum's live chain. Our analysis allows us to demonstrate several shortcomings of Ethereum's cost model and the ways in which: i) it affects each PKI construction, ii) it impedes the establishment of a standard library of smart contracts and, iii) it incentivizes smart contract developers to adopt several malpractices. We propose several modifications to Ethereum's cost model, which are minor and fair, to address all the aforementioned issues and others.

[1] Joint work with Katerina Samari ([92]).

- We introduce an alternative paradigm for developing decentralized applications on top of smart contract platforms ([91]) by decoupling the issue of storage from verifying the validity of data, which aligns well with the future of smart contract platforms. Our proposed scheme can be employed by any application that requires a verifiable representation of its application data, i.e., it can be used to implement any key-value store. To illustrate the efficacy of our approach, we present a case study of an ERC20 token construction, the most widely deployed standard for fungible tokens, which numbers over 130,000 compliant contracts on Ethereum's live chain ([9]).
- We address, in a two-fold manner, Ethereum's monotonically increasing state which, if left unchecked, will have a direct impact on Ethereum's security and, utlimately, its longevity. We introduce recurring fees that are proportional to the state of smart contracts and adjustable by the miners that maintain the network. In addition, we propose a scheme where the cost of storage-related operations reflects the effort that miners have to expend to execute them. Lastly, we revisit our ERC20 token and our hash tree-based PKI constructions and show that under such a pricing scheme that encourages economy in the state consumed by smart contracts, both of our constructions reduce the incurred transaction fees by up to an order of magnitude.

2. BACKGROUND

2.1 Outline

In this section, we provide background knowledge required to understand the general framework under which our smart contract-based NS design operates, while also constituting this thesis as self-contained as possible. We begin by providing a brief and informal introduction on basic concepts related to public key (asymmetric) cryptosystems and digital certificates. We introduce a basic overview of blockchains and, subsequently, of Ethereum, the most notable public smart contract platform at the time of this writing and our platform of choice with regards to the implementation of the constructions presented in this work. Lastly, we provide an overview on the topic of cryptographic accumulators, which lie at the heart of the design of our NS.

2.2 Public key Cryptosystems

The core purpose of public key, or assymetric, cryptosystems is to allow parties to communicate securely, over an insecure channel, without having a previously agreed upon secret key. Furthermore, they simplify the process of key management as they alleviate the need to manage distinct cryptographic keys per pair-wise communication channel, which is the case in symmetric cryptosystems. In public-key cryptosystems, each user (or device) has a pair of cryptographic keys, i.e., a public and a private key. The public key (pk) of a user is distributed freely and it is assumed that it can be retrieved reliably. The means under which this takes place is the topic of the next section. The private or secret key (sk), as its name implies, is known only to the user (or device) and should **never** be transmitted or revealed to any other party. Public key cryptosystems are associated with a set of algorithms that employ the aforementioned keys to perform cryptographic operations that allow for secure communication. Typically, the algorithms of a public key cryptosystem are a union of those provided by encryption and digital signature schemes. In the following, we present an informal overview of these algorithms in the asymmetric setting:

- $KeyGen(1^{\lambda}) \rightarrow (pk, sk)$: On input the security parameter λ, output a public (pk) and a private (sk) key pair such that $pk, sk \in \mathcal{K}$, where $\mathcal{K}$ is the set of all keys.
- $Encrypt(m, ek) \rightarrow c$: On input a message $m \in \mathcal{M}$, where $\mathcal{M}$ is the message space, and an encryption key $ek \in \mathcal{K}$, output a ciphertext $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of all ciphertexts.
- $Decrypt(c, dk) \rightarrow m$: On input a ciphertext $c \in \mathcal{C}$ and a decryption key $dk \in \mathcal{K}$, output a message $m \in \mathcal{M}$. This algorithm "undoes" encryption, i.e., the following **correctness** property holds:

$$Decrypt(Encrypt(m, ek), dk) = m \tag{2.1}$$

for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$.

- $Sign(m, sk) \rightarrow \sigma$: On input a message $m \in \mathcal{M}$ and a signing key $sk \in \mathcal{K}$, output a digital signature σ that can be used to verify the authenticity and integrity of message m.
- $VerifySig(m, \sigma, vk) \rightarrow \{0, 1\}$: On input a message $m \in \mathcal{M}$, a digital signature σ and a verification key $vk \in \mathcal{K}$, output 1, if σ is a valid signature for m, otherwise, output 0.

The $KeyGen()$ algorithm of a public key cryptosystem outputs key pairs with the following properties: 1) it is extremely improbable to derive the private key from the public key (the opposite is trivial) and, 2) a message encrypted with one of the keys in the pair can only be decrypted with its corresponding key in the pair. In this setting, the sender encrypts her message with the receiver's public key and transmits the ciphertext. The receiver, via her private key, is the only one that can decrypt the transmitted ciphertext and retrieve the original message. This exchange guarantees *data confidentiality*, i.e., an eavesdropper cannot extract information about the message's plaintext. However, it does not address the following issues. First, if the ciphertext where to be tampered with during transmission, either by some malicious party, or due to, e.g., bit rot, the receiver has no way to verify the integrity of the message. In addition, the receiver of the ciphertext cannot infer the identity of the sender, i.e., she cannot authenticate from where the ciphertext came from. To achieve both of these properties, i.e., *data integrity* and *authentication*, the sender generates with her secret key sk a digital signature σ, via the $Sign()$ algorithm, which is appended, along with her public key, to the transmitted ciphertext. The receiver, as previously, will decrypt the ciphertext with her private key and verify the validity of the signature ($VerifySig()$) by using the public key of the sender as the verification key. An additional and important property of digital signatures is non-repudiation, or, more precisely, *non-repudiation of origin*. This property entails that an entity that has signed a message m cannot, at a later time, deny having signed it.

2.3 Digital Certificate

A digital certificate, also known as a public key or identity certificate, is a digital document that is used to prove the ownership of a public key by some entity. Digital certificates provide information regarding an identity name, its public key and other pertinent details regarding the key owner and the employed cryptographic algorithms. Unlike real world documents that are used for identification, e.g., passports, digital certificates can and are used to identify non-human entities as well, such as software and hardware components. Furthermore, certificates can (and should) be distributed and copied multiple times, which is certainly not the case for passports. In the general case, digital certificates do not contain sensitive information and, thus, their public distribution does not induce privacy or other security risks.

In the previous section, we (conveniently) assumed that the public key of a user can be retrieved reliably via some mechanism. This could be facilitated by a publicly accessible, on-

line directory that maintains an associative array between identity names and certificates. However, this directory now constitutes a single point of failure. Indeed, a malicious party that manages to breach this centralized component will be able to launch impersonation and key modification attacks. To guard against such incidents, a *data integrity* mechanism is required. While digital signatures provide this property, we still have no guarantee that the public key that is contained in a digital certificate belongs to the "claimed" owner. This is a *data authenticity* problem. In the following, we briefly introduce the two most prevalent, but not infallible, approaches in issuing and binding a digital certificate to an entity.

2.3.1 Registration and Certification Authorities

A Registration Authority (RA) is a company or organization that is responsible for receiving and validating client requests for digital certificates. An RA acts as an intermediary between clients and Certification Authorities (CAs), i.e., trusted third parties that are responsible for issuing and managing the status of digital certificates. In this setting, clients generate an asymmetric key pair and, subsequently, construct a certification request ([104]). A certification request contains, among others, the client's public key, an identity name (e.g., a fully qualified domain name) and other credentials (attributes) that can be used to prove the client's claim on the identity. The client signs the certification request with her private key and sends it to the RA. On receipt, the RA verifies the client's request and, if it is deemed valid, it contacts a CA and requests the issuance of a digital certificate. X.509 v3 ([58]) is an Internet standards protocol that defines, among others, the format and encoding of digital certificates issued by CAs. An X.509 v3 certificate is comprised, at minimum, by the following fields:

- **Issuer:** This field identifies the CA that has signed and issued the certificate via a non-empty, hierarchical distinguished name composed of various attributes.
- **Version:** This field describes the version of the encoded certificate and, depending on its value, implies if only basic fields are present or not.
- **Serial Number:** A long integer (of up to 20 octets size) that uniquely identifies each certificate issued by a given issuer.
- **Subject:** This field identifies, via a non-empty, hierarchical distinguished name, the entity associated with the public key stored in the corresponding field.
- **Not Before:** The date on which the certificate validity period begins.
- **Not After:** The date on which the certificate validity period ends.
- **Public Key Info:** This field is used to carry the public key and identifies the algorithm with which the key is used.

- **Signature:** This field contains the algorithm identifier for the algorithm used by the CA to sign the certificate.
- **Signature Value:** This field contains a digital signature computed upon the ASN.1 DER encoding of all the aforementioned fields.
- **Signature Algorithm:** This field contains the identifier for the cryptographic algorithm used by the CA to sign this certificate. This field must contain the same algorithm identifier as the “Signature” field introduced previously.

The CA, i.e., the “Issuer”, generates the client’s digital certificate and signs it with its private key. Consequently, a party that wishes to verify the validity of a client’s digital certificate must have access to the signing CA’s correct public key (or certificate). However, a verifying party cannot verify a client’s digital certificate without, first, having a valid copy of the issuing CA’s certificate. But, how do we escape this infinite loop of delegating validation to some higher tier authority?

To avoid this “chicken” or “egg” paradox, the TLS protocol ([98]) anchors trust in a list of root certificates. Root certificates are self-signed, i.e., the certificate’s subject is also the issuer who signed the certificate. In the World Wide Web (WWW), browsers have a list of several “trustworthy” (quotes reflect the fact that most of them have been compromised) CA root certificates already incorporated when they ship. A web server that uses a certificate that is signed by such a trusted CA is **automatically** trusted by the client’s browser. This is the most widely deployed mechanism for distributing root certificates and, thus, bootstrapping trust.

A root CA can issue digital certificates to subordinate or intermediate CAs, which inherit, to some degree, the trustworthiness of the root CA. This process can be repeated multiple times and results in the so-called TLS certification hierarchy. The distribution of intermediary CA certificates is, typically, offloaded to an online and public directory whose contents can be accessed via the Lightweight Directory Access Protocol (LDAP, [74]).

2.3.2 Web of Trust

Web of Trust (WoT) is a concept that was first introduced by PGP ([114]) to establish the authenticity of bindings between identity names and public keys (or certificates). In WoT, the authenticity of a public key is entirely decentralized, i.e., users are able to designate others as trustworthy by signing their public key. Thus, a user accumulates signatures of her public key from other others that have deemed her trustworthy. A verifier considers a public key as “trustworthy” if she is able to find a valid digital signature from one (or more) users that she trusts. Conceptually, there is no single point of failure regarding trust, as in the previously reviewed case. However, as in the case of CAs, there is nothing that prevents multiple users from creating digital certificates for the same identity. In the following, we provide a high level overview of PGP’s implementation of the WoT model ([57]).

In PGP, keys are always stored at the user's end inside structures that are referred to as key rings. A PGP key ring is a collection of public keys of the entities with whom the owner intends to communicate. Every PGP user maintains two key rings, a private key ring and a public key ring. The private key ring of a user stores all of her public and private key pairs, since users can have multiple pairs associated with them. The private key is encrypted with a passphrase that the user supplies at the time of the key's generation. The public key ring of a user includes information that indicate the validity and the level of trust that can be placed on the public keys of other users. A user's trust in her own key pair is referred to as implicit trust and constitutes the highest level of trust that can be assigned to a key. In PGP, public key ring keys that are signed by one (or more) keys of the user's private key ring, are assumed to be valid. A user can assign the following three levels of trust to the public key of another user:

- Complete Trust: The owner of the key is trusted to sign and validate the public keys of other users.
- Marginal Trust: The owner of the key is trusted to validate the public keys of other users.
- No Trust: The owner of the key is not trusted for anything.

PGP supports X.509 digital certificates, but it also has its own format for certificates, which is referred to as the PGP certificate format. The unique feature of a PGP certificate is that it allows for multiple users to sign it. The signature of the owner in the certificate is called a self−signature and it is contained in every PGP certificate. This means that every user signs her certificate to ensure non-repudiation of her data.

2.4 Public Key Infrastructure

In the real world, it is (still) relatively easy to identify the parties with which we are communicating, either via face recognition based on past knowledge, or various identification documents, such as government issued identity cards, which are (assumed to be) "unforgeable" (quotes reflect the fact that in countless of cases these documents have been forged). However, in the digital world, it is quite an involved task to securely establish the identity of the parties with which we are communicating. Furthermore, since communication networks, such as the Internet, are by design insecure and unreliable, message transmissions are susceptible, among others, to eavesdropping and tampering. The purpose of a public key infrastructure (PKI) is to address these underlying problems of trust, authentication, confidentiality and integrity. PKIs mirror the trust models of the physical world and couple it with standard cryptographic techniques to enable secure electronic communications and transactions.

On a high-level, a PKI can be defined as a real-world system that allows the registration, revocation and distribution (storage and retrieval) of digital certificates. In cryptography, a PKI is defined as an arrangement that *binds* public keys to identities. Notice that there

is a subtle, but very important, difference between the aforementioned definitions. The cryptographic definition is **not** concerned with issues of trust and data authenticity, which are also referred to as *accurate registration* and where reviewed in Sections 2.3.1 and 2.3.2. Instead, its main focus is providing *identity retention*, i.e., the inability of a user to impersonate another registered user. We have already illustrated that these two issues are orthogonal to each other. Since accurate registration is deployment specific and, in some cases, not even required (e.g., in DNSs), the main focus of this thesis is to provide system constructions that offer identity retention.

2.5 Blockchain

Bitcoin ([88]), the world's first cryptocurrency, introduced the term "blockchain" which, nowadays, is used to refer to either a data structure, or a peer to peer (P2P) network. As a data structure, a blockchain is an ordered list of blocks, where each block contains a collision resistant hash of the previous block, i.e., blocks are sequentially "chained" together. Thus, the history of blocks cannot be altered without invalidating the hash chain, a property which is commonly referred to as *immutability*. The first block is of special value as it initializes the data structure and it is referred to as the "genesis block". In the case of Bitcoin, the genesis block was produced by its pseudonymous creator, Satoshi Nakamoto. Each block contains a small (possibly empty) list of transactions. Transactions are data packages that provide the means to clients of the blockchain to encode operations that they want to invoke. For instance, in Bitcoin, transactions can be used to specify (simple) rules for transferring currency value. However, clients can also encode arbitrary data in transactions that are not related to blockchain operations. Thus, the blockchain can be viewed as a general purpose, append-only, secure log of immutable data.

As a P2P network, a blockchain is comprised by nodes, which are referred to as *miners*. Miners aggregate and validate transactions issued by clients of the blockchain and produce blocks that will be appended on the blockchain. The process of creating a new block is referred to as *mining*. Miners broadcast new blocks to the P2P network, thus, each miner holds a replica of the entire blockchain. The P2P network executes a consensus algorithm to decide on the block that will be appended on the chain. Depending on the deployment setting, blockchains employ different consensus algorithms.

In a public setting, i.e., where membership is open and anyone can plug-in a node to the P2P network, or act as a client, it is imperative to employ a consensus algorithm that can defend against Sybil attacks. Douceur et al. [60] has proved that to defend against Sybil attacks, distributed systems must employ either authentication, which requires a centralized authority to assign distinct identifiers to participating entities (e.g., a CA), or, computational power. As the former is not an option for public blockchains, the most commonly employed consensus algorithm is based on computation and is referred to as "Proof-of-Work" (PoW). In a few words, each miner, to produce a new block, must solve a (hard) computational puzzle, which essentially involves inverting a hash function. Clearly, this procedure is not (energy) efficient as illustrated by, e.g., Bitcoin's maximum throughput of 7 transactions

per second. An alternative consensus algorithm that is on the rise for public blockchains is "Proof-of-Stake" (PoS), which we do not cover in this work and refer the interested reader to the work of Badertscher et al. [44]. In all cases, public blockchains feature a reward mechanism, which comes in the form of a digital currency (cryptocurrency), to compensate miners, thus, creating a counter-incentive to free-riding, while still retaining a highly scalable, free-entry system. Put simply, a miner that produces a new block that is appended to the blockchain will be rewarded with some units of the blockchains cryptocurrency.

In a private, or permissioned setting, a set of organizations collectively decide to form a consortium where, typically, each organization hosts one or more peer nodes. In this setting, node membership is assumed to be a solved problem, i.e., each peer can identify every other peer in the network by, e.g., having a copy of its digital certificate. Thus, since Sybil attacks pose no threat, permissioned blockchains can employ far more efficient consensus algorithms. For instance, Hyperledger Fabric ([42]), the most notable permissioned blockchain, via its "pluggable" consensus feature, can employ a variety of consensus algorithms, such as Kafka ([67]) for crash fault tolerance and BFT-Smart ([100]) for byzantine fault tolerance. Furthermore, in permissioned settings, there is no need for an incentive mechanism, as the maintenance of the blockchain itself is assumed to be of value to the consortium as a whole. As an example, a consortium may be formed by several banking institutions to maintain a private, but still verifiable log, of cross bank transactions. In this case, the blockchain can be useful to the banks to provide verifiable financial information to government tax agencies, while still retaining the privacy of their internal operations from other members of the consortium.

2.6 Ethereum

A downside of traditional blockchains, such as Bitcoin, is their limited set of pre-defined operations. This issue is addressed by a new generation of programmable blockchains that allow users to encode operations of arbitrary complexity. This effectively creates a decentralized, smart contract ([101]) platform. The rules of the smart contract are expressed in code and are enforced by the blockchain's consensus protocol. In this section, we introduce Ethereum ([106]), the most notable public smart contract platform, on top of which we implemented our PKI constructions.

Ethereum differentiates itself from Bitcoin, as well as, several other public blockchains that are simple clones of Bitcoin (known as *altcoins*), in several ways. First, to track a client's *ether* balance (this is Ethereum's cryptocurrency), Ethereum employs a simple, account-based, bank-style model, compared to Bitcoin, where a client's balance is spread across multiple unspent transaction outputs (UTXOs). These Ethereum accounts are referred to as *externally owned accounts* (EOA) and are created and controlled by user-generated private keys. Second, Ethereum utilizes ether, which is converted to a unit called *gas* for transaction execution, only as a "fuel" for its operation. Its main focus is to provide a platform that facilitates smart contracts, not another cryptocurrency. In Ethereum, smart contracts are stateful, user-defined programs that specify rules governing transactions, thus,

Table 2.1: Gas price values depending on desired transaction priority ([11]). One unit of Ether corresponds to 10^9 Gwei (Giga wei, also known as Shannon). We refer the interested reader to [12] for more details regarding ether denominations.

Transaction Priority	Gas Price
Low	2 Gwei
Medium	2.5 Gwei
High	4 Gwei

allowing mutually distrustful parties to transact safely with each other. This is facilitated by Ethereum's (Quasi) Turing complete programming language, a far more expressive and potent tool than Bitcoin's *script*, which currently forbids even simple arithmetic operations, such as integer multiplication. Developing a smart contract involves writing its code in a high-level language (e.g., Solidity [28]), which is then compiled to Ethereum Virtual Machine (EVM) initialization code. To avoid reader confusion, we clarify the following: 1) the EVM should not be confused with typical VMs, instead, it should be considered as a sandboxed, network-isolated execution environment, 2) each miner runs its own local EVM instance and, 3) the initialization code is EVM assembly code that, when executed, will create the contract's actual assembly code. Deploying a contract involves wrapping its initialization code in a transaction, signing it, and broadcasting it to the network. A successfully mined block containing a contract creation transaction triggers the following events. First, *all* miners receiving the block will execute the initialization code and produce, as a result, the contract's actual code. Second, a new account is created, which serves as a means of storing the smart contract's code and tracking its state. This is Ethereum's second account type, which is referred to as a *contract account*.

To enhance the reader's understanding of Ethereum and, by extension, of our systems, we elaborate on a few additional key points regarding contracts. First, contracts are accounts, i.e., they have their own, separate ether balance, which is controlled by their code. Second, contracts "live" in the blockchain, meaning that both their state and code is publicly accessible. Thus, they can be trusted for correctness (provided their code was properly audited), but not for privacy. Users "poke" (call) contracts by broadcasting transactions that specify the code (function) to be executed and its input arguments. Moreover, it is possible for a contract to call another contract, which is referred to as a *message call*. Each transaction byte, as well as, all EVM operations, ranging from simple contract instruction fetches, to altering a contract's state variable, cost some amount of gas. Ethereum employs a flat cost model, i.e., each EVM operation is priced in isolation of the rest of the code according to its complexity and input byte size. Both message calls, as well as, transactions that invoke smart contract code, specify an upper bound on the amount of gas that can be expended by their execution, which is referred to as *gas limit*. This mechanism protects miners from, e.g., getting stuck in an infinite loop. This is why Ethereum's computational model is referred to as "Quasi" Turing complete. Miners, apart from their standard block reward, are also compensated with a fee that is proportional to the complexity and

input length of the executed code. The conversion of ether to gas depends on the *gas price* parameter, which is specified in the body of each transaction. Thus, while gas cost is static (for some computation), its correspondence to ether, i.e., its real monetary cost, can vary. Assuming that the execution of a transaction costs g_{cost} units of gas and that the specified gas price in the transaction is g_{price}, the transaction's cost in ether units can be calculated as $Ether_{cost} = g_{cost} \times g_{price}$. Gas price is one of the factors that influences the incentive of miners to include a transaction in their next block. Table 2.1 illustrates standard gas price values, depending on the desired transaction priority (pulled from [11], note that these values are fairly volatile).

2.7 Cryptographic Accumulators

Cryptographic accumulators provide a representation of a set of elements into a single, succinct accumulator value. For every element that has been added to the accumulator, a process which is referred to as *accumulation*, one can efficiently produce a *membership witness*, i.e., a proof that certifies that the element has been accumulated. The party that is responsible for maintaining the accumulator is commonly referred to in the literature as the "accumulator manager". Since their introduction, researchers have proposed several accumulator schemes, where each provides a different combination of features. In the following, we provide a rough overview of the main features and properties of proposed accumulator schemes that are relevant to this work.

A first point of categorization for accumulator schemes pertains to whether the accumulated set of elements can be updated or not. In *static* accumulator schemes, the accumulator's value, as well as, any computed membership witnesses, have to be recomputed from scratch when the accumulated set of elements is updated. Examples of such schemes are the ones proposed by Benaloh et al. [49], who first introduced the concept of cryptographic accumulators, and the subsequent refinement of their construction by Bari et al. [47], who strengthened its security property. Camenisch et al. [54] extended previous works and presented the first *dynamic* accumulator scheme. In this scheme, both the accumulator's value, as well as, membership witnesses can be updated by utilizing only public information (the accumulator's public key), i.e., no trapdoor information is required. Cryptographic accumulators can be further categorized according to the type of updates that they support. As illustrated in the work of Baldimtsi et al. [46], an accumulator scheme is considered as *additive*, if it supports only additions, and *subtractive*, if it supports only deletions. Li et al. [79] introduced the notion of *universal accumulators*, i.e., accumulators that support both membership, as well as, non membership witnesses. Lastly, accumulator schemes can be categorized based on the hardness of their underlying cryptographic assumption. All of the aforementioned accumulator schemes are based on the strong RSA assumption. Other proposed accumulator schemes are based on collision-resistant hash functions and hash trees (e.g., [90]), known order groups (bilinear pairings, e.g., [89]) and the hardness of the short integer solution problem (lattices, e.g., [75]).

Cryptographic accumulators provide a number of benefits. First, their compact (or even

constant) size makes them suitable candidates for storage-limited devices (e.g., smartphones). Second, their security properties are based on standard hardness assumptions, thus, making them suitable for critical security infrastructures. Third, in cases where accumulators are stored on the blockchain, as in our constructions, there is an additional benefit. Indeed, the blockchain is not employed to enforce consensus on the entire set of accumulated elements. Instead, the consensus object is the accumulator's value, which has the following benefits. First, users are not required to perform a complete retrieval and verification of the entire transaction history, i.e., downloading and validating the entire blockchain. Instead, an outdated, or, new client, can download and validate only block headers to update her state, which is far more efficient both in terms of communication and computation. Second, it allows the introduction of an unreliable component that users can query to efficiently obtain, among others, a more compact version of the entire history of operations, compared to the full transaction history. Due to the verifiable nature of cryptographic accumulators, this increased efficiency comes at no cost.

3. RELATED WORK

3.1 Categorization

Several previously proposed systems utilize the same underlying primitive, each in its own unique way, to provide for a decentralized PKI, or naming service. These similarities allow us to perform a rough categorization of all previously proposed schemes and discuss each individual category separately. We focus on full-fledged decentralized PKIs and naming services, i.e., systems that implement registration, revocation, storage and retrieval of records pertaining to identities. Thus, we will not be concerned with certification systems (e.g., [78]), which do not offer revocation, hybrid approaches, e.g., coupling CAs with structured overlays (e.g., [108]), or, even PGP ([114]), whose operation relies on centralized servers for revocation and digital certificate distribution.

3.2 RSM-based PKIs

Researchers have proposed DPKIs based on the replicated state machine (RSM) paradigm ([113, 96]) to enforce a global, consistent view of the system's state. This is achieved by having nodes participate in an authenticated agreement protocol and typically assume: 1) a threshold t of faulty nodes, 2) *join()* and *leave()* protocols for nodes wishing to enter, or leave, a replica group, to adjust the system's threshold parameter and, 3) nodes are able to authenticate any (potential) participant. In RSM-based PKIs, registration requires one to perform an "out-of-band" negotiation with multiple administrative domains, which is cumbersome for the user. In addition, non-determinism, e.g., time-stamping, is a key difficulty of consistent replication since it can lead to replica state-divergence, thus, compromising fault-tolerance. However, time-stamping is essential in a PKI for tracking certificate lifetime. Blockchain-based systems, on the other hand, do not suffer from this issue and they have already been used for the implementation of time-stamping services (e.g., [69]). Furthermore, they employ a different form of agreement which is based on computation. This alternative agreement algorithm has the nice property of being adaptable as nodes freely join and leave the system. Practical experience has illustrated that the blockchain approach has been highly favored by both the research community, as well as, the industry, due to its highly scalable, adaptive and non-restrictive nature ([2, 22, 31]).

3.3 Overlay-based PKIs

Structured overlays have also been proposed to distribute the services of a PKI ([43, 59]). These are, by design, scalable, load-balanced and provide for efficient storage and retrieval of data. Other works (e.g., [87]) employ unstructured overlay P2P networks where data is stored and queried by performing a series of short random walks. Unfortunately,

none of these systems is able to defend against Sybil attacks. Douceur ([60]) has proved that to defend against the Sybil attack, distributed systems must employ either authentication, or, computational power. However, the aforementioned systems do not employ either, thus, they are insecure. Blockchains, on the contrary, are resilient to the Sybil attack since their operation inherently depends on computational power. Furthermore, in all of the aforementioned systems, nodes are expected to participate in resource-intensive protocols without, however, incentivizing node participation, nor, enforcing correct behavior of participating nodes. However, blockchains (e.g., [88, 24]) compensate parties that engage in their consensus protocol via digital currency units. This reward mechanism has empirically demonstrated it incentivizes node participation, while still retaining a highly scalable, free-entry system.

3.4 Blockchain-based Naming Services

Namecoin ([24]) is the first and one of the biggest *altcoins*, i.e., clones of Bitcoin, that provides a distributed DNS as its main function. Other notable blockchain-based naming services are Emercoin ([8]), the Ethereum Name Service (ENS, [15]) and Blockstack's BNS ([40]). These naming services employ the blockchain to store, verify and query identity records. Unfortunately, this approach is inefficient as it forces each user to store an entire copy of the blockchain and traverse its contents every time she needs to validate a mapping. This limits the system's applicability significantly; for example, storing the entire blockchain on a smartphone is prohibitive. Moreover, validating mappings, which is the most frequent operation, requires an increasing amount of computation as more blocks are appended to the blockchain. In contrast, the constructions presented in this work, follow a different design principle that stores in the state of a smart contract a constant-sized and verifiable representation of **all** identity records by employing cryptographic accumulators. Clients of our naming service can verify the validity of identity records via these accumulators and by interacting with a (potentially unreliable) storage network. Thus, in our design, there is no need to linearly search the blockchain. In addition, our design can be implemented on top of any system that allows the development of smart contracts. We chose to implement our constructions on top of Ethereum because it has a more rich and diverse ecosystem of applications. Multisignature wallets (e.g., [3]) and various (non) fungible tokens (e.g., [9]) are just a couple example applications that can benefit from the standard, on-blockchain authentication mechanism that our constructions provide.

Melara et al. [83] introduce CONIKS, a privacy-preserving decentralized PKI where users can monitor the consistency of their own (id, pk) pairs. While privacy is an important property, e.g., for chat applications, it is not a requirement for traditional PKIs. For instance, in the Web-PKI paradigm, identity names and the public keys of participants have to be public. CONIKS's operation is based on "identity providers", i.e., centralized entities that sign authenticated bindings and appropriately transform identity names for privacy purposes. CONIKS assumes the existence of a separate PKI to distribute the public keys of identity providers. Thus, it does not constitute a standalone PKI service, whilst the constructions presented in this work do. More importantly, CONIKS, lacks a formal proof of its

security and privacy guarantees. This is also the case for systems derived from CONIKS, i.e., EthIKS ([50]), Catena ([103]) and Conifer ([110]), which implement CONIKS on top of Ethereum and Bitcoin. Certcoin ([64]) is a blockchain-based PKI proposal that employs cryptographic accumulators but has a number of inefficiencies, e.g., it recomputes, from scratch, accumulator values during **each** revocation. Furthermore, Certcoin has no security model for the PKI it implements nor a proof that it provides the claimed service. Formal proofs of security are imperative for such critical security infrastructures.

4. NAMING SERVICE: BUILDING BLOCKS AND DEFINITION

4.1 Preliminaries

In this section, we introduce basic notation and definitions that will be used throughout this thesis. We use λ to denote the security parameter and negl$(\cdot)$ to denote a function negligible in some parameter.

Definition 4.1.1 (Strong-RSA Assumption [47])**.** *For any p.p.t adversary $\mathcal{A}$,*

$$\mathsf{Pr}[n \leftarrow \mathsf{KeyGen}(1^\lambda); x \leftarrow \mathbb{Z}_n^*; (y, e) \leftarrow \mathcal{A}(n, x) : y^e = x \mod n] = \mathsf{negl}(\lambda),$$

where, $n = pq$, p and q are safe primes.

Definition 4.1.2 (Collision-Resistance [53])**.** *Let $\mathcal{H} : \mathcal{L} \times \mathcal{M} \rightarrow \mathcal{Y}$ be a hash function family. Let λ be a security parameter, where $\lambda = |\mathcal{L}| = |\mathcal{Y}|$. Then, $\mathcal{H}$ is collision-resistant if and only if, for every p.p.t algorithm $\mathcal{A}$, we have:*

$$\mathsf{Pr}[\lambda \xleftarrow{R} \mathcal{L}; (m, m') \leftarrow \mathcal{A}(\lambda) : m \neq m', \mathcal{H}_\lambda(m) = \mathcal{H}_\lambda(m')] = negl(\lambda)$$

where, $\lambda \xleftarrow{R} \mathcal{L}$ means that λ is chosen uniformly at random from the set of keys $\mathcal{L}$.

Definition 4.1.3 (2-Universal Hash Function Family [56])**.** *Let $U = \{f | f : X \rightarrow Y\}$ be a family of functions. We say that U is a 2-Universal Hash Function Family if, for all $x_1, x_2 \in X$ with $x_1 \neq x_2$ and for all $y_1, y_2 \in Y$, $\mathsf{Pr}_{f \in U}[f(x_1) = y_1 \wedge f(x_2) = y_2] = (\frac{1}{|Y|})^2$.*

Definition 4.1.4 (Pseudorandom Generator)**.** *Let $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{p(\lambda)}$ be a deterministic polynomial time algorithm and $p(\cdot)$ a polynomial in some parameter λ. We say that G is a pseudorandom generator if, for any p.p.t. algorithm D,*

$$\big|\mathsf{Pr}[D(r) = 1] - \mathsf{Pr}[D(G(s)) = 1]\big| \leq \mathsf{negl}(\lambda),$$

where, r is a string chosen uniformly at random from $\{0,1\}^{p(\lambda)}$ and s, the seed, is chosen uniformly at random from $\{0,1\}^\lambda$.

4.2 Public State, Additive, Universal Accumulator

At a high level, we consider an accumulator as *public state*, if one can perform all of its operations by only having access to its public key, i.e., no trapdoor knowledge is required. According to the terminology presented in Baldimtsi et al. [46], an accumulator is *additive*, if it only allows for addition of elements, and *universal*, if it allows for both membership and non membership witnesses.

In the following, we present the definition of a public state, additive, universal accumulator. Our definition employs two trusted parties. The first one, T, runs the key generation

algorithm (KeyGen(1^λ)) and publishes the accumulator's public key. The second one, the "accumulator manager" T_{acc}, is responsible for maintaining the accumulator. [1]

Definition 4.2.1 (public-state, additive, universal accumulator)**.** Let D be the domain of the accumulator's elements, and X, the current accumulated set. A public-state, additive, universal accumulator consists of the following algorithms:

- KeyGen(1^λ): On input the security parameter λ, this algorithm generates a key pair (pk, sk) and outputs pk. This algorithm is run by T.
- InitAcc(pk): On input the accumulator's public key pk, this algorithm outputs an initialized accumulator value c_0 for an empty accumulated set, i.e., $X \leftarrow \emptyset$. This algorithm is run by T_{acc}.
- Add$(\mathsf{pk}, \mathsf{x}, \mathsf{c})$: On input the accumulator's public key pk, an element $x \in D$ to be added and an accumulator value c, this algorithm outputs (c', W), where c', is the updated value of the accumulator, and W, is a membership witness for x.
- MemWitGen$(\mathsf{pk}, \mathsf{X}, \mathsf{c}, \mathsf{x})$: On input the accumulator's public key pk, the accumulated set of values X, the accumulator's value c and an element $x \in X$, this algorithm outputs a membership witness W for x.
- NonMemWitGen$(\mathsf{pk}, \mathsf{X}, \mathsf{c}, \mathsf{x})$: On input the accumulator's public key pk, the accumulated set of values X, the accumulator's value c and an element x, such that $x \in D \wedge x \notin X$, this algorithm outputs a non membership witness W for x.
- UpdMemWit$(\mathsf{pk}, \mathsf{x}, \mathsf{y}, \mathsf{W})$: On input the accumulator's public key pk and a membership witness W for x, this algorithm outputs an updated membership witness W' for x. This algorithm is run after $(c', W_y) \leftarrow$ Add(pk, y, c), where W_y is a membership witness for y.
- UpdNonMemWit$(\mathsf{pk}, \mathsf{x}, \mathsf{y}, \mathsf{W})$: On input the accumulator's public key pk and a non membership witness W for x, this algorithm outputs an updated non membership witness W' for x. This algorithm is run after $(c', W_y) \leftarrow$ Add(pk, y, c).
- VerifyMem$(\mathsf{pk}, \mathsf{x}, \mathsf{W}, \mathsf{c})$: On input the accumulator's public key pk, the value of the accumulator c and an element $x \in D$, this algorithm outputs 1, if W is a valid membership witness for x, otherwise, it outputs 0.
- VerifyNonMem$(\mathsf{pk}, \mathsf{x}, \mathsf{W}, \mathsf{c})$: On input the accumulator's public key pk, the value of the accumulator c and an element $x \in D$, this algorithm outputs 1, if W is a valid non membership witness for x, otherwise, it outputs 0.

Informally, an accumulator scheme is *correct* if, for any honestly produced membership witness, the membership verification algorithm (VerifyMem) outputs 1, and if, for any honestly produced non membership witness, the non membership verification algorithm (VerifyNonMem) outputs 1. Furthermore, we consider a universal accumulator as *secure* if, no p.p.t. adversary can produce a valid non membership witness for a member of the

[1] Note that the notion of a *public-state* accumulator is weaker from that of a *strong* accumulator, as defined by Camacho et al. [53]. In a strong accumulator, the KeyGen algorithm, which produces the initial value of the accumulator, is publicly executable and any party can verify the validity of its output. In contrast, in a public-state accumulator, the KeyGen algorithm is run by a trusted party T.

$\mathcal{G}_{\mathcal{A}}^{\text{acc}-\text{sec}}$: On input 1^{λ},

- $\mathcal{C}$ runs KeyGen(1^{λ}), generates (pk, sk) and gives pk to the adversary $\mathcal{A}$.
- $\mathcal{A}$ first makes an InitAcc query to $\mathcal{C}$. $\mathcal{C}$ initializes a set $X \leftarrow \emptyset$, runs InitAcc and returns the value c_0 to $\mathcal{A}$.
- When $\mathcal{A}$ performs an Add query for an element x, $\mathcal{C}$ sets $X \leftarrow X \cup \{x\}$ and computes $(c', W) \leftarrow$ Add(pk, x, c). Then, $\mathcal{C}$ returns the pair (c', W) to $\mathcal{A}$.
- $\mathcal{A}$ outputs (x^*, W^*).

The game returns 1 if at least one of the following conditions holds:

1. $x^* \notin X$ and VerifyMem$(pk, x^*, W^*, c) = 1$,
2. $x^* \in X$ and VerifyNonMem$(pk, x^*, W^*, c) = 1$.

Figure 4.1: The security game between the adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, where $\mathcal{C}$ plays the roles of both T and T_{acc}.

accumulated set, nor, a valid membership witness for an element which is not a member of the accumulated set. The security property of an accumulator can be met as *collision-freeness*, or, *soundness* in the literature. A formal definition of security is given below (Definition 4.2.2), utilizing a game between a Challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, as illustrated in Figure 4.1. For a formal definition of correctness, we refer the interested reader to the work of Mashatan et al. [81].

Definition 4.2.2. *We say that an accumulator is secure if, for any p.p.t. adversary $\mathcal{A}$ interacting with a challenger $\mathcal{C}$, as illustrated in the security game of Figure 4.1, it holds that* $\Pr[\mathcal{G}_{\mathcal{A}}^{\text{acc}-\text{sec}}(1^{\lambda}) = 1] = \text{negl}(\lambda)$.

4.3 Naming Service Definition

We define the security of our naming service in the UC framework ([55]) by modeling it as an ideal functionality $\mathcal{F}_{ns}$ (Figure 4.2). $\mathcal{F}_{ns}$ interacts with n clients, m servers, a party T, which is responsible for setup, and an adversary $\mathcal{S}$, which is called the simulator. It stores (id, pk) pairs and supports a number of operations. The servers are responsible for running the naming service and therefore, before the setup, we require that all servers send a (sid, Init) message. During setup, the party T specifies a relation R, which defines under which condition a public key can be revoked. In practice, this relation might be a verification algorithm for a NIZK proof, or, a signature on a randomly selected message. After the setup phase, a client can register an (id, pk) pair, assuming the identity is available, and, can revoke an (id,pk) pair, assuming her public key satisfies relation R. Furthermore, she is able to retrieve the public key of a registered identity and check, whether an identity, or, an (id, pk) pair, is registered or not. Our model considers only static corruptions, thus, we

$\mathcal{F}_{ns}$:

- On input (sid, Init) by a server S_i, $\mathcal{F}_{ns}$ sends (sid, Init, S_i) to $\mathcal{S}$. If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ sets $Y \leftarrow Y \cup \{S_i\}$ (where Y is initialized as $Y = \emptyset$) and returns success to S_i. When all servers have sent a message (sid, Init), $\mathcal{F}_{ns}$ sets flag $=$ start. Also, $\mathcal{S}$ corrupts a number of clients. We denote as C_{cor} the set of corrupted clients.
- On input (sid, Setup, R) by T, $\mathcal{F}_{ns}$ checks if flag $=$ start and forwards (sid, Setup, R) to $\mathcal{S}$. If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ stores R, initializes a set $X \leftarrow \emptyset$, sets flag $=$ manage and returns success to T.
- On input $(sid, \text{Register}, id, pk)$ by a client C, if flag $=$ manage, $\mathcal{F}_{ns}$ forwards $(sid, \text{Register}, id, pk)$ to $\mathcal{S}$.
  - If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ checks if there is $(id, \cdot) \in X$. If there is no $(id, \cdot) \in X$, it sets $X \leftarrow X \cup (id, pk)$ and returns success to C, otherwise, it returns fail to C.
  - If $\mathcal{S}$ returns fail, $\mathcal{F}_{ns}$ returns fail to C.
  - If $\mathcal{S}$ returns $(sid, \text{Register}, id', pk', C)$, $\mathcal{F}_{ns}$ checks if $C \in C_{cor}$ and if there is $(id', \cdot) \in X$. If there is no $(id', \cdot) \in X$, it sets $X \leftarrow X \cup (id', pk')$ and returns success to C, otherwise, it returns fail to C.
- On input $(sid, \text{Revoke}, id, pk, \text{aux})$ by C, if flag $=$ manage, $\mathcal{F}_{ns}$ forwards $(sid, \text{Revoke}, id, pk, \text{aux})$ to $\mathcal{S}$.
  - If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ checks whether $R(pk, \text{aux}) = 1$ and $(id, pk) \in X$. If both conditions hold, $\mathcal{F}_{ns}$ computes $X \leftarrow X \setminus (id, pk)$ and returns success to C, otherwise, it returns fail to C.
  - If $\mathcal{S}$ returns fail, then $\mathcal{F}_{ns}$ returns fail to C.
  - If $\mathcal{S}$ returns $(sid, \text{Revoke}, id', pk', \text{aux}', C)$, $\mathcal{F}_{ns}$ checks if $C \in C_{cor}$, $R(pk', \text{aux}') = 1$ and $(id', pk') \in X$. If so, $\mathcal{F}_{ns}$ computes $X \leftarrow X \setminus (id', pk')$, and returns success to C, otherwise, it returns fail to C.
- On input $(sid, \text{Retrieve}, id)$ by C, if flag $=$ manage, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow and if $(id, pk) \in X$, $\mathcal{F}_{ns}$ returns pk to C, otherwise, it returns $\perp$. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, $\mathcal{F}_{ns}$ returns fail to C.
- On input $(sid, \text{VerifyID}, id)$ by C, if flag $=$ manage, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow and if $(id, pk) \in X$, $\mathcal{F}_{ns}$ returns 1 to C, otherwise, it returns 0. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, $\mathcal{F}_{ns}$ returns fail to C.
- On input $(sid, \text{VerifyMapping}, id, pk)$ by C, if flag $=$ manage, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow and if $(id, pk) \in X$, $\mathcal{F}_{ns}$ returns 1 to C, otherwise, it returns 0. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, $\mathcal{F}_{ns}$ returns fail to C.

Figure 4.2: The naming service functionality $\mathcal{F}_{ns}$ interacts with a set of n clients, a set of m servers, a trusted party T and the simulator $\mathcal{S}$. It allows clients to register, revoke, retrieve and verify (id, pk) pairs.

assume that the simulator specifies the set of corrupted clients, C_{cor}, before setup. The party T is considered trusted, thus, the simulator, $\mathcal{S}$, is not allowed to corrupt T. Also, note that, in practice, this functionality cannot be realized for any corruption model for the m servers. However, the corruption model for the servers depends on the protocol with which we aim to realize the functionality $\mathcal{F}_{ns}$.

We say that a protocol π securely realizes the ideal functionality $\mathcal{F}_{ns}$ if, for any p.p.t adversary $\mathcal{A}$ interacting with the real-world protocol π, there is a p.p.t. simulator $\mathcal{S}$ interacting with the functionality $\mathcal{F}_{ns}$, such that, no p.p.t. environment $\mathcal{Z}$ can distinguish whether it interacts with π and $\mathcal{A}$ in the hybrid-world, or, with $\mathcal{S}$ and $\mathcal{F}_{ns}$ in the ideal-world, except with negligible probability.

$\underline{\mathcal{F}_{UDB}:}$

- On input $r = (sid, \text{InitUDB})$ by a server S_i, $\mathcal{F}_{UDB}$ sets $S'_{init} \leftarrow S'_{init} \cup \{S_i\}$ (initialized as $S'_{init} \leftarrow \emptyset$) and sends (r, S_i) to $\mathcal{A}$. Then, $\mathcal{F}_{UDB}$ returns success to S_i. If $|S'_{init}| = \ell$, $\mathcal{F}_{UDB}$ sets flag $=$ ready, $DBstate \leftarrow \emptyset$ and $p \leftarrow 0$.
- On input $r = (sid, \text{Post}, x)$ by C, if flag $=$ ready, send (r, C) to $\mathcal{A}$. If $\mathcal{A}$ sends allow, $\mathcal{F}_{UDB}$ sets $p \leftarrow p + 1$, $DBstate[p] \leftarrow x$ and returns success to C.
- On input $r = (sid, \text{RetrieveDB})$ by C, if flag $=$ ready, $\mathcal{F}_{UDB}$ sends (r, C) to $\mathcal{A}$. If $\mathcal{A}$ sends allow, $\mathcal{F}_{UDB}$ returns $DBstate$ to C.
- On input $r = (sid, \text{ChangeDBstate}, DBstate')$ by $\mathcal{A}$, $\mathcal{F}_{UDB}$ sets $DBstate \leftarrow DBstate'$.

Figure 4.3: The functionality $\mathcal{F}_{UDB}$ models an unreliable database that stores information relevant to our protocols. It interacts with a set of n clients, a set of ℓ servers and the adversary $\mathcal{A}$.

In Figure 4.3, we introduce the functionality $\mathcal{F}_{UDB}$, which handles the storage of information that are relevant to our protocols, e.g., (id, pk) pairs. $\mathcal{F}_{UDB}$ interacts with n clients, a set of ℓ servers and the adversary. This functionality models an "unreliable database", i.e., the adversary may tamper with its contents. Its involvement in our protocols is twofold. First, a client queries this functionality to retrieve all the necessary information that will allow her to, subsequently, interact with $\mathcal{F}_{TP}$. Second, following the completion of an interaction with $\mathcal{F}_{TP}$, the client stores in $\mathcal{F}_{UDB}$, among others, information that were output by the smart contract and reflect the new state of the system. We elaborate more on the information that clients query from and store to $\mathcal{F}_{UDB}$ later on in this chapter where we provide a high-level description of each operation.

Our naming service's security depends solely on that of the smart contract platform and the accumulator scheme. This allows us to employ a variety of primitives to realize $\mathcal{F}_{UDB}$, whose concrete specification we leave as future work. For instance, even centralized cloud storage services are a viable option. However, we believe that the best approach is a distributed file storage system, especially one that has "bridges" with the Ethereum network, which is the smart platform that we chose for our implementation. Some notable examples are Swarm ([30]), Storj ([29]) and IPFS ([23]). Another option would be an authenticated DHT network comprised of nodes that have registered in our PKI; this is a

$\mathcal{F}_{TP}$:

- On input $r = (sid, \text{InitTP})$ by a server S_i, $\mathcal{F}_{TP}$ sets $S_{init} \leftarrow S_{init} \cup \{S_i\}$ (initialized as $S_{init} \leftarrow \emptyset$) and sends (r, S_i) to $\mathcal{A}$. Then, $\mathcal{F}_{TP}$ returns success to S_i. If $|S_{init}| = m$, $\mathcal{F}_{TP}$ sets flag $=$ ready.
- On input $r = (sid, \text{Install}, P)$ by T, if flag $=$ ready, $\mathcal{F}_{TP}$ sends (r, T) to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{F}_{TP}$ sets flag $=$ start, $state \leftarrow \varepsilon$, where ε is the empty string, stores P and returns success to T.
- On input $r = (sid, x)$ by C, if flag $=$ ready, $\mathcal{F}_{TP}$ sends (r, C) to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{F}_{TP}$ runs P on input $(x, state)$ which outputs $(y, state')$. $\mathcal{F}_{TP}$ sets $state \leftarrow state'$ and returns $(y, state')$ to C. If x is an invalid input for P, $\mathcal{F}_{TP}$ returns $\perp$ to C.

Figure 4.4: The functionality $\mathcal{F}_{TP}$ captures the role of the smart contract. It interacts with a trusted party T, a set of n clients, a set of m servers and the adversary $\mathcal{A}$.

suitable candidate both in terms of security (i.e., it is Sybil resilient), as well as, efficiency, due to its logarithmic message complexity for querying and storing information.

In Figure 4.4, we define the functionality $\mathcal{F}_{TP}$, which captures the role of the smart contract in our protocols. This functionality interacts with a party T, a set of n clients and a set of m servers, some of which may be corrupted by the adversary prior to the initialization phase. $\mathcal{F}_{TP}$ is initialized by a trusted party T by receiving as input a program P . The state of $\mathcal{F}_{TP}$ is updated after a call to program P and the output is received by the calling party. Note that the implementation of $\mathcal{F}_{TP}$ requires an honest majority of servers, along the lines of [65, 66, 45]. The adversary has always full knowledge of all the computations performed and may interfere by either aborting, or, allowing, an execution of P at will. However, he is restricted from modifying the output. Implementing $\mathcal{F}_{TP}$ using a blockchain protocol has the servers acting as “miners”. Party T and the clients interact with the blockchain by issuing message calls (transactions). The security properties of the underlying blockchain, specifically related to persistence of transactions, cf. [65, 66, 45], imply the security of $\mathcal{F}_{TP}$’s realization.

In our constructions, the smart contract maintains two instances of a public state, additive, universal accumulator to facilitate the verification of the validity of identities, or, (id, pk) pairs. The first accumulator, c_1, accumulates (id, pk) pairs, allowing clients to infer if an (id, pk) pair is currently registered or not. The second accumulator, c_2, accumulates id names, allowing clients to infer if an identity is registered or not. For ease of presentation, in the upcoming chapters, we will describe our protocols using two accumulators. We can achieve the same net result using only one accumulator since both c_1 and c_2 accumulate arbitrary strings. Thus, we are able to accumulate both types of tuples in one accumulator, while still being able to generate the (non) membership witnesses required in our protocols. We provide details on how this can be achieved for each construction.

The smart contract is the core and most expensive component with which to interact. Its active involvement is required only during registration and revocation. To register, a client

queries $\mathcal{F}_{UDB}$ for the history of operations that will allow her to compute a proof that her id is not accumulated in c_2. To revoke her pair, the client, instead, computes a proof that her pair is accumulated in c_1 and proves possession of the secret key (relation R).

5. RSA-BASED PKI CONSTRUCTION

5.1 RSA-based, Public State, Additive, Universal Accumulator

In Figure 5.1, we present a construction of a public-state, additive, universal accumulator. We aim to accumulate identities or (id, pk) pairs, i.e., arbitrary strings. Thus, the accumulator's domain is $D = \{0,1\}^*$. The construction presented here is a combination of the RSA-based universal accumulator of Li et al. [79], accompanied with a procedure Map, which maps arbitrary strings to prime numbers. Namely, for any algorithm run on input an element $z \in \{0,1\}^*$, the party who runs the algorithm, first, executes the procedure Map, which maps z to a prime number, e.g., z_p, and then, proceeds by running the same algorithm as in the accumulator of Li et al. [79] for the prime number z_p. The procedure Map that we utilize is a modified version of a procedure suggested by Gennaro et al. [68]. Thus, we first present the procedure suggested by Gennaro et al. [68] and then we present the modified algorithm Map, which is utilized in the construction of Figure 5.1. Lastly, we prove that the accumulator of Figure 5.1 is secure according to Definition 4.2.2.

5.1.1 Mapping arbitrary strings to primes.

Gennaro et al. [68] describe a procedure that utilizes a universal hash function family U of functions (Definition 4.1.3), which maps strings of $3k$ bits to strings of k bits with the additional property that, for any $y \in \{0,1\}^k$ and given $f \in U$, one can efficiently sample uniformly from the set $\{x \in \{0,1\}^{3k} : f(x) = y\}$. On input $z \in \{0,1\}^*$, it first computes $h(z)$, where $h : \{0,1\}^* \to \{0,1\}^k$ is a collision-resistant hash function. It then samples repeatedly from the set $\{x \in \{0,1\}^{3k} : f(x) = h(z)\}$ to find a prime number $O(k^2)$ times. This procedure is collision-resistant if h is collision resistant and will output a prime number with high probability due to the following Lemma.

Lemma 5.1.1 ([68]). *Let U be a Universal Hash Function Family from $\{0,1\}^{3k}$ to $\{0,1\}^k$. Then, for all but a $(1/2^k)$-fraction of functions $f \in U$ and for any $y \in \{0,1\}^k$, a fraction of at least $1/ck$ elements in the set $\{x \in \{0,1\}^{3k} : f(x) = y\}$ are primes, for a small constant c.*

Therefore, an algorithm which samples ck^2 times from the set $\{x \in \{0,1\}^{3k} : f(x) = h(z)\}$ will fail to find a prime number only with negligible probability. For completeness, we provide a proof of Lemma 5.1.1 in Appendix A. The proof presented in Appendix A is similar to that provided by Sanderand et al. [102].

5.1.2 Map: A modified version of the algorithm of Gennaro et al. [68].

In our RSA-based accumulator construction (Figure 5.1), we employ a deterministic version of the Map procedure, which we introduce below. Specifically, we utilize a pseudorandom generator $G : \{0,1\}^k \to \{0,1\}^{p(k)}$, where $p(k)$ is a polynomial in k, and a labeled

hash function $h : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^k$, which is collision-resistant and is modeled as a Random Oracle.

We start by picking two labels, i.e., $\mathsf{label}_0, \mathsf{label}_1 \in \{0,1\}^*$. Then, Map, on input $z \in \{0,1\}^*$, computes $h(\mathsf{label}_0, z)$ and $G(h(\mathsf{label}_1, z))$. Then, Map samples elements from the set $\{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$ using as randomness $G(h(\mathsf{label}_1, z))$ and stops when a prime number is found. It is easy to see that since $\mathsf{label}_0, \mathsf{label}_1$ are the same in all invocations of the algorithm, the algorithm on input $z \in \{0,1\}^*$ always outputs the same value. Below, in Lemma 5.1.2, we show that Map finds a prime, except with negligible probability, and, in Lemma 5.1.3, we prove that Map is collision-resistant[1].

Lemma 5.1.2. *Let $z \in \{0,1\}^*$. The algorithm* Map*, on input z, outputs a prime number, except with negligible probability, assuming that, G is a pseudorandom generator and, the hash function h, is a random oracle.*

Proof. Assume that Map, on input $z \in \{0,1\}^*$, fails to find a prime number with non-negligible probability α. We will construct a p.p.t distinguisher D, which breaks the property of the pseudorandom generator G, as defined in Definition 4.1.4. Recall that the only difference between the procedure of Gennaro et al. [68] and the algorithm Map, described in the previous paragraph, is the samping of elements from the set $X = \{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$. In the former case, elements are sampled uniformly at random while, in the latter case, elements are sampled by using as randomness the output of the PRG G. Based on that, we consider the following p.p.t. distinguisher D:

- On input a string $x \in \{0,1\}^{p(k)}$, sample from the set $X = \{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$ using as randomness the string x.
- If a prime number p is output, then, return 1, else, return 0.

First, since h is a random oracle, the seed $h(\mathsf{label}_1, z)$ is considered random. Then, if $x = r$, where r is chosen uniformly at random, by Lemma 5.1.1, we have that $\Pr[D(r) = 1] = 1 - \mathsf{negl}(k)$. By the assumption that Map fails to find a prime with non-negligible probability α, we have that

$$\left|\Pr[D(r') = 1] - \Pr[G(h(\mathsf{label}_1, z)) = 1]\right| = 1 - \mathsf{negl}(k) - (1 - \alpha) = \alpha - \mathsf{negl}(k), \quad (5.1)$$

which is a contradiction, according to Definition 4.1.4.

□

Lemma 5.1.3. *The algorithm* Map *is collision-resistant if the hash function h is collision-resistant. Namely, no p.p.t. adversary can find $z_1, z_2 \in \{0,1\}^*$ with $z_1 \neq z_2$, such that* $\mathsf{Map}(z_1) = \mathsf{Map}(z_2) = p$.

Proof. We assume that Map is not collision resistant, i.e., there is a p.p.t. adversary $\mathcal{A}$, which finds $z_1 \neq z_2$, such that $\mathsf{Map}(z_1) = \mathsf{Map}(z_2) = p$. This requires that the algorithm Map samples elements from the same set of solutions $X = \{x \in \{0,1\}^{3k} : f(x) =$

[1]The requirement for a deterministic version of Map was suggested by Aggelos Kiayias. Credits for the proofs of Lemmas 5.1.2 and 5.1.3 go to Katerina Samari.

The domain of the accumulator is $D = \{0,1\}^*$.

- KeyGen(1^λ): Generate a pair of safe primes p, q of equal length, such that $p = 2p' + 1$, $q = 2q' + 1$ and p', q' are also primes. Computes $n = pq$, which has length λ, and choose g randomly from QR_n. Set $\ell = \lfloor \lambda/2 \rfloor - 2$ and choose a deterministic procedure Map, which receives as input an arbitrary string and outputs a prime number less than $2^{\lfloor \lambda/2 \rfloor - 2}$. Sets $sk = (p, q)$ and output $pk = (n, g, \mathsf{Map})$.
- InitAcc(pk): Output $c_0 = g$.
- Add(pk, x, c): Invoke $x_p \leftarrow \mathsf{Map}(x)$, compute $c' = c^{x_p} \mod n$, set $W = c$ and output (c', W).
- MemWitGen(pk, X, c, x): Compute and output the membership witness $W = g^{\prod_{x_i \in X \setminus \{x\}} \mathsf{Map}(x_i)}$ for x.
- NonMemWitGen(pk, X, c, x): Invoke $x_p \leftarrow \mathsf{Map}(x)$ and compute $u = \prod_{x_i \in X} \mathsf{Map}(x_i)$. Since $gcd(x_p, u) = 1$, run the extended Euclidean algorithm and compute $a, b \in \mathbb{Z}$, such that $au + bx_p = 1$. By the Euclidean division, a can be written as $a = a' + qx_p$, where $0 \leq a' < x_p$. Therefore, $a'u + (b + qu)x_p = 1$. Set $b' = b + qu$ and compute $d = g^{-b'}$. Finally, output the non membership witness $W = (a', d) = (a \mod x, g^{-b-qu})$ for x.
- UpdMemWit(pk, x, y, W): Invoke $y_p \leftarrow \mathsf{Map}(y)$, compute and output the updated membership witness $W' = W^{y_p} \mod n$ for $x \neq y$.
- UpdNonMemWit(pk, x, y, W): Invoke $x_p \leftarrow \mathsf{Map}(x)$ and $y_p \leftarrow \mathsf{Map}(y)$. Since $y_p \neq x_p$, execute the extended Euclidean algorithm and compute $a_0, r_0 \in \mathbb{Z}$, such that $a_0 y_p + r_0 x_p = 1$. Then, multiply both sides by a, i.e., $aa_0 y_p + ar_0 x_p = a$, and compute $a' = a_0 a \mod x_p$. Then, find $r \in \mathbb{Z}$, such that $a' y_p = a + r x_p$, compute $d' = dc^r \mod n$ and output the updated non membership witness $W' = (a', d')$ for x.
- VerifyMem(pk, x, W, c): Invoke $x_p \leftarrow \mathsf{Map}(x)$ and output 1, if $W^{x_p} = c \mod n$, otherwise, output 0.
- VerifyNonMem(pk, x, W, c): On input $W = (a, d)$, invoke $x_p \leftarrow \mathsf{Map}(x)$. Output 1, if $c^a = d^{x_p} g \mod n$, otherwise, output 0.

Figure 5.1: Construction of a public-state, additive, universal accumulator based on the strong-RSA assumption in the Random Oracle model.

$h(\mathsf{label}_0, z_1)\}$. Thus, $\mathcal{A}$ should find a collision in the hash function h, i.e., $\mathcal{A}$ finds z_1, z_2 such that $h(\mathsf{label}_0, z_1) = h(\mathsf{label}_0, z_2)$. However, this holds only with negligible probability. □

5.1.3 Security of the accumulator of Figure 5.1.

Before formally proving the security of the accumulator of Figure 5.1, we first illustrate, via a simple example, why the procedure Map has to be deterministic in our construction and, thus, the reason we cannot use the procedure of Gennaro et al. [68]. First, assume that we used the procedure of Gennaro et al.[68] without our suggested modification and that an element $x \in \{0,1\}^*$ was added (Add) in the accumulator. T_{acc} computes $x_p \leftarrow \mathsf{Map}(x)$ and adds x_p to the underlying RSA accumulator. An adversary can produce a

valid non membership witness W for x simply by producing a different prime $x'_p \neq x_p$ for the element x and then running the non membership witness generation algorithm for x'_p. Therefore, the security property of the accumulator, as defined in the security game of Figure 4.1, would not hold, since the adversary can output (x, W), such that $x \in X$ and VerifyNonMem$(pk, x, W, c) = 1$.

The security of our accumulator is derived by the security of the accumulator of Li et al. [79], which is proven secure under the strong-RSA assumption, and the properties of the algorithm Map, as proven in Lemma 5.1.2 and Lemma 5.1.3.

Theorem 5.1.1. *The accumulator of Figure 5.1 is secure according to Definition 4.1 under the strong-RSA assumption (Definition 4.1.1) and the collision-resistance of the algorithm* Map *(Lemma 5.1.3) in the Random Oracle model.*

Proof. Assume there is a p.p.t. adversary $\mathcal{A}$, which breaks the security of the accumulator of Figure 5.1. Then, according to Definition 4.1, $\mathcal{A}$ outputs (x^*, W^*), such that: (1) $x^* \notin X$ and VerifyMem$(pk, x^*, W^*, c) = 1$, or, (2) $x^* \in X$ and VerifyNonMem$(pk, x^*, W^*, c) = 1$. Suppose that (1) holds. Then, there are two possible cases: (a) $\mathcal{A}$ comes up with x, x^*, such that, Map$(x) =$ Map(x^*) and $x \in X$, thus, breaking the collision-resistance of the Map procedure, or, (b) $\mathcal{A}$ computes a valid membership witness W^* for a prime x^*_p, where, Map$(x^*) = x^*_p$ and $x^* \notin X$. In the latter case, we can construct a p.p.t adversary $\mathcal{B}$, which breaks the strong-RSA assumption. We refer for further details on this case to the proof of Li et al. [79]. Next, assume that case (2) holds. This implies two possible scenarios: First, $\mathcal{A}$ comes up with a valid non membership witness W^* for a prime x^*_p, where, Map$(x^*) = x^*_p$ and $x \in X$. This means that we can construct a p.p.t. adversary $\mathcal{B}$, which breaks the strong-RSA assumption (see the proof of Li et al. [79]) and, therefore, we have a contradiction. In the second scenario, the procedure Map, on input x^*, outputs two different primes (with non-negligible probability), e.g., $x^*_{p_1}$ and $x^*_{p_2}$, if we invoke it twice. This means that if $x^*_{p_1}$ is added in the accumulator first, then it would be possible for $\mathcal{A}$ to compute a valid non membership witness W^* for $x^*_{p_2}$. However, this is impossible, since the procedure Map is deterministic . □

5.1.4 Constructing a universal accumulator from an additive, universal accumulator ([46]).

Assume that ACC^{add}_U is an additive, universal accumulator, which accumulates elements of the form (x, i, op), where, x is the element to be added, i, is an index, and op, is either a or d. We construct a universal accumulator ACC_U, from ACC^{add}_U, as follows. When an element x is added to ACC_U for the first time, T_{acc} adds the value $(x, 1, a)$ to ACC^{add}_U. Otherwise, it adds (x, i, a), where, the index i indicates that this is the i-th time that x is added to ACC^{add}_U. When an element x is deleted from ACC_U, T_{acc} adds (x, i, d) to ACC^{add}_U. In order to prove membership of x in ACC_U, one should find an index i, such that, $\big((x, i, a) \in ACC^{add}_U\big) \wedge \big((x, i, d) \notin ACC^{add}_U\big)\big)$. Accordingly, to prove that $x \notin X$, one should either prove that $(x, 1, a) \notin ACC^{add}_U$, or, find an index i, such that, $\big((x, i-1, d) \in ACC^{add}_U\big) \wedge \big((x, i, a) \notin ACC^{add}_U\big)\big)$.

5.2 RSA-based PKI

5.2.1 Construction

In this section, we formally present our RSA-based PKI construction, which is built on top of the RSA accumulator that we previously defined (Section 5.2). Figure 5.2 illustrates program P_{RSA} for this implementation, which T inputs to $\mathcal{F}_{TP}$ in the setup phase. Here, c_1 accumulates (id, pk, i, op) tuples and c_2 accumulates (id, i, op) tuples, where $op = a$ or $op = d$. We denote as $W_{i,j}$ the j-th computed witness for accumulator c_i ($i \in \{1, 2\}$) in the context of an individual PKI operation.

1. On input $(\text{Setup}, params)$, where $params = (pk_1, pk_2, R)$, invoke $c_{0,1} \leftarrow \text{InitAcc}(pk_1)$ and $c_{0,2} \leftarrow \text{InitAcc}(pk_2)$. This procedure initializes two public state, additive, universal accumulators c_1 and c_2 by setting $c_1 \leftarrow c_{0,1}, c_2 \leftarrow c_{0,2}$. Store $state \leftarrow (params, c_1, c_2)$ and return $state$.
2. On input $(\text{Register}, id, pk, i, W_{2,1}, W_{2,2})$,
   (a) Check if $\text{VerifyNonMem}(pk_2, (id, i, a), W_{2,1}, c_2) = 1$. Otherwise, return fail.
   (b) If $i \geq 2$, check if $\text{VerifyMem}(pk_2, (id, i-1, d), W_{2,2}, c_2) = 1$ and return fail if it does not.

   Compute $(c'_1, W'_{1,1}) \leftarrow \text{Add}(pk_1, (id, pk, i, a), c_1)$ and $(c'_2, W'_{2,1}) \leftarrow \text{Add}(pk_2, (id, i, a), c_2)$.
   Update $state$ by setting $c_1 \leftarrow c'_1$ and $c_2 \leftarrow c'_2$, and return $((c'_1, W'_{1,1}), (c'_2, W'_{2,1}))$.
3. On input $(\text{Revoke}, id, pk, i, W_1, W_2, \text{aux})$,
   (a) Check if $R(pk, \text{aux}) = 1$. Otherwise, return fail.
   (b) Check if $\text{VerifyMem}(pk_1, (id, pk, i, a), W_{1,1}, c_1) = 1$. Otherwise, return fail.
   (c) Check if $\text{VerifyNonMem}(pk_1, (id, pk, i, d), W_{1,2}, c_1) = 1$. Otherwise, return fail.

   Compute $(c'_2, W'_{2,1}) \leftarrow \text{Add}(pk_2, (id, i, d), c_2)$ and $(c'_1, W'_{1,1}) \leftarrow \text{Add}(pk_1, (id, pk, i, d), c_1)$.
   Update $state$ by setting $c_1 \leftarrow c'_1$ and $c_2 \leftarrow c'_2$ and return $((c'_1, W'_{1,1}), (c'_2, W'_{2,1}))$.
4. On input (RetrieveState),
   Return $state = (params, c_1, c_2)$.

Figure 5.2: The program P_{RSA}, which is input to $\mathcal{F}_{TP}$ during initialization, in the RSA-based PKI construction.

A client that wishes to register her (id, pk) pair computes the following. First, a non membership witness $W_{2,1}$ for the tuple (id, i, a) in c_2. Second, and only if her identity has been registered at least once in the past ($i \geq 2$), a membership witness $W_{2,2}$ for the tuple $(id, i-1, d)$ in c_2. If both conditions hold, she will be able to convince the smart contract that id is available. To construct these witnesses, the client queries $\mathcal{F}_{UDB}$ for the history of operations and locates records (if any) pertaining to id to find the proper value for i. Then, she invokes the Register function of the smart contract and receives the accumulators' updated values and two witness values, $W'_{1,1}$ and $W'_{2,1}$. These are membership witnesses for (id, pk, i, a) in c_1 and (id, i, a) in c_2, respectively. Next, the client computes a non membership witness $W_{1,2}$ for (id, pk, i, d) in c_1, sets $W_{1,1} \leftarrow W'_{1,1}$ and posts a $(\text{Register}, id, pk, i, W_{1,1}, W_{1,2})$ record to $\mathcal{F}_{UDB}$.

1. On input (sid, Init), a server S_i sends (sid, InitTP) and $(sid, \text{InitUDB})$ to $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$. If S_i receives success by both $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, then S_i returns success.
2. On input (sid, Setup, R), T sends $(sid, \text{Install}, P_{RSA})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns as success, T runs KeyGen(1^λ) twice, sets $params = (pk_1, pk_2, R)$ and sends $(sid, (\text{Setup}, params))$ to $\mathcal{F}_{TP}$, which is input to P_{RSA} (if $\mathcal{A}$ returns allow to $\mathcal{F}_{TP}$). If $\mathcal{F}_{TP}$ returns $state \leftarrow (params, c_1, c_2)$ to T, T returns success.
3. On input $(sid, \text{Register}, id, pk)$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. On receipt of $DBstate$, C checks:
   (a) If the last (i-th) record for id is a Register record, C returns fail. Otherwise, C sets $i = i + 1$ and runs $W_{2,1} \leftarrow$ NonMemWitGen$(pk_2, X_2, c_2, (id, i, a))$, $W_{2,2} \leftarrow$ MemWitGen$(pk_2, X_2, c_2, (id, i-1, d))$.
   (b) If no record for id is found, C sets $i = 1$, $W_{2,2} = \perp$ and runs $W_{2,1} \leftarrow$ NonMemWitGen$(pk_2, X_2, c_2, (id, i, a))$.

   C sends $(sid, \text{Register}, id, pk, i, W_{2,1}, W_{2,2})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $((c'_1, W'_{1,1}), (c'_2, W'_{2,1}), state)$, C sets $W_{1,1} \leftarrow W'_{1,1}, X_1 \leftarrow X_1 \cup \{(id, pk, i, a)\}$, runs $W_{1,2} \leftarrow$ NonMemWitGen$(pk_1, X_1, c'_1, (id, pk, i, d))$, sends $\big(sid, \text{Post}, (\text{Register}(id, pk, i, W_{1,1}, W_{1,2})\big)$ to $\mathcal{F}_{UDB}$ and outputs success, or fail otherwise.
4. On input $(sid, \text{Revoke}, id, pk, \text{aux})$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. On receipt of $DBstate$, C searches for $k \geq 1$ records that follow her Register$(id, pk, i, W_{1,1}, W_{1,2})$ record. For each encountered record:
   (a) C sets $y = (id', pk', j, a)$ or $y = (id', pk', j, d)$ if she encounters a $(\text{Register}, id', pk', j, W'_{1,1}, W'_{1,2})$ or a $(\text{Revoke}, id', pk', j)$ record, respectively.
   (b) C runs $W_{1,1} \leftarrow$ UpdMemWit$(pk_1, (id, pk, i, a), y, W_{1,1})$, $W_{1,2} \leftarrow$ UpdNonMemWit$(pk_1, (id, pk, i, d), y, W_{1,2})$.

   Then, C sends $(sid, \text{Revoke}, id, pk, i, W_{1,1}, W_{1,2}, \text{aux})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $((c'_1, W'_{1,1}), (c'_2, W'_{2,1}), state)$, C sends $\big(sid, \text{Post}, (\text{Revoke}, id, pk, i)\big)$ to $\mathcal{F}_{UDB}$ and returns success, or fail otherwise.
5. On input $(sid, \text{Retrieve}, id)$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. On receipt of $DBstate$:
   (a) If the last record for id is Register$(id, pk, i, W_{1,1}, W_{1,2})$, C runs Steps 4a and 4b. Otherwise, C outputs fail.
   (b) C sends $(sid, \text{RetrieveState})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $state$, C runs VerifyMem$(pk_1, (id, pk, i, a), W_{1,1}, c_1)$ and VerifyNonMem$(pk_1, (id, pk, i, d), W_{1,2}, c_1)$. Otherwise, C outputs fail.

   If all algorithms of Step 5b output 1, C outputs pk as the retrieved public key, otherwise, C outputs fail.
6. On input $(sid, \text{VerifyID}, id)$, C runs Step 5. If Step 5 outputs some pk, C outputs 1, otherwise, C outputs 0.
7. On input $(sid, \text{VerifyMapping}, id, pk)$, C runs Step 5. If Step 5 outputs pk, C outputs 1, otherwise, C outputs 0.

Figure 5.3: Description of the protocol π_{RSA} built upon the program P_{RSA} of Figure 5.2.

To revoke an (id, pk) pair, a client computes the following. First, a proof of ownership of her secret key sk, which is captured by relation R and is implemented as a signature of her

public key (aux $= \sigma_{sk}(pk)$). Second, a membership witness $W_{1,1}$ for the tuple (id, pk, i, a) in c_1. Third, a non membership witness $W_{1,2}$ for the tuple (id, pk, i, d) in c_1. Then, she invokes the Revoke function of the smart contract and, on success, posts a $(\text{Revoke}, id, pk, i)$ record to $\mathcal{F}_{UDB}$.

In Figure 5.3, we present the formal description of protocol π_{RSA}, which realizes the functionality $\mathcal{F}_{ns}$. X_1 and X_2 denote the sets of accumulated elements of c_1 and c_2, respectively, and are constructed as follows. For any record of the form $(\text{Register}, id, pk, i, \cdot)$, a client adds (id, pk, i, a) to X_1 and (id, i, a) to X_2. For any record of the form $(\text{Revoke}, id, pk, i)$, a client adds (id, pk, i, d) to X_1 and (id, i, d) to X_2. We show that this construction is secure by providing a formal proof of Theorem 5.2.1 in Appendix B.

Theorem 5.2.1. *The protocol π_{RSA} of Figure 5.3 securely realizes the functionality $\mathcal{F}_{ns}$ of Figure 4.2 in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world under the strong-RSA assumption in the Random Oracle model.*

5.2.2 Using only one accumulator

We denote as pk_c the public key of the accumulator c that the smart contract manages. A client that wishes to register her (id, pk) pair computes the following. First, a non membership witness W_1 for the tuple (id, i, a) in c. Second, and only if her identity has been registered at least once in the past ($i \geq 2$), a membership witness W_2 for the tuple $(id, i-1, d)$ in c. The Register operation of the smart contract is modified and performs the following verifications: 1) VerifyNonMem$(pk_c, (id, i, a), W_1, c) = 1$ and, 2) If $i \geq 2$, VerifyMem$(pk_c, (id, i-1, d), W_2, c) = 1$. If the aforementioned conditions hold, the registration is valid and the contract will perform, in order, the following computations: 1) $(c, W_1') \leftarrow$ Add$(pk_c, (id, i, a), c)$, 2) $(c, W_2') \leftarrow$ Add$(pk_c, (id, pk, i, a), c)$ and, 3) $W_1' \leftarrow$ UpdMemWit$(pk_c, (id, i, a), (id, pk, i, a), W_1')$. Then, the contract will return (c, W_1', W_2'). Lastly, the client computes a non membership witness W_1 for (id, pk, i, d) in c, sets $W_2 \leftarrow W_2'$ and posts a $(\text{Register}, id, pk, i, W_1, W_2)$ record to $\mathcal{F}_{UDB}$. To revoke an (id, pk) pair, a client, as previously, signs pk ($\sigma_{sk}(pk)$) and computes the following. First, a membership witness W_1 for tuple (id, pk, i, a) in c. Second, a non membership witness W_2 for tuple (id, pk, i, d) in c. The contract's Revoke operation is modified and performs the following verifications: 1) VerifyMem$(pk_c, (id, pk, i, a), W_1, c) = 1$ and, 2) VerifyNonMem$(pk_c, (id, pk, i, d), W_2, c) = 1$. If all the aforementioned verifications succeed and $R(pk, \text{aux}) = 1$, the revocation is valid and the smart contract will perform, in order, the following computations: 1) $(c, W_1') \leftarrow$ Add$(pk_c, (id, i, d), c)$ and, 2) $(c, W_2') \leftarrow$ Add$(pk_c, (id, pk, i, d), c)$. Then, the contract will return (c, W_1', W_2'). Lastly, the client posts a $(\text{Revoke}, id, pk, i)$ record to $\mathcal{F}_{UDB}$. This approach cuts down in half the contract's state, however, it increases the computation of the Register operation by one modulo exponentiation which, as discussed in Chapter 7, incurs a notable increase in the operation's cost.

6. HASH TREE-BASED PKI CONSTRUCTION

6.1 Hash tree-based Universal Accumulator

In the following, we provide a high level description of the hash tree, universal accumulator of Camacho et al. [53], which our second PKI construction employs. The security of this accumulator is based on collision-resistant hash functions (Definition 4.1.2). This accumulator employs a public data structure $m = (X, T)$, which is referred to as *memory*. The accumulated set of elements $X = \{x_1, ..., x_n\}, n \in \mathbb{N}$ is ordered and there are two special elements, $x_0 = -\infty$ and $x_{n+1} = +\infty$, such that $x_0 \prec x_j \prec x_{n+1}, \forall x_j \in X$, where $\prec$ is the order relation on the elements of X (e.g., the lexicographic order of strings). The main underlying data structure of this accumulator is a binary, balanced, hash tree T. Each node N of T holds a pair of strings $(Val_N, Proof_N)$, which are referred to as the node's value and proof, respectively. In the following, we illustrate how these strings are computed. Regarding the former, one might assume that the value of each node of T is an element (or its hash) of the accumulated set X. This could have been the case if the goal was to provide *only* membership witnesses. However, to provide for a universal accumulator, i.e., one that supports non membership witnesses as well, Camacho et al. [53] follow a different approach. The value of each node of T is the hash of two consecutive elements of X (recall that X is ordered), e.g., $Val_N = \mathcal{H}(x_1||x_2)$, where $x_1, x_2 \in X$. In this setting, the following equivalences hold:

- Proving $x \in X$ is equivalent to: $(x_a, x_b) \in \{(x_i, x_{i+1}) : 0 \leq i \leq n)\} \wedge (x = x_a \vee x = x_b)$.
- Proving $x \notin X$ is equivalent to: $x_a \prec x \prec x_b \wedge (x_a, x_b) \in \{(x_i, x_{i+1}) : 0 \leq i \leq n\}$.

The proof of a node N of T is computed as $Proof_N = H(Val_N||Proof_{Left_N}||Proof_{Right_N})$, where $Left_N$ and $Right_N$ are N's left and right children and $Parent_{Left_N} = Parent_{Right_N} = N$. If $Left_N$ or $Right_N$ are not in T, i.e., if $Left_N = Nil$ or $Right_N = Nil$, then $Proof_{Nil} = \epsilon$, where ϵ denotes the empty string. For instance, if N is a leaf node, its proof is computed as $Proof_N = H(Val_N||\epsilon||\epsilon)$, or, if N is an inner node and it has no right child, its proof is computed as $Proof_N = H(Val_N||Proof_{Left_N}||\epsilon)$. The accumulator's value is $Acc = Proof_{Root_T}$, where $Root_T$ is the root node of the hash tree T. Figure 6.1 depicts an example of a hash tree T, which corresponds to the accumulated set $X = \{x_1, ..., x_8\}$, where only node values are presented. Note that the placement of the values in the tree is irrelevant, since it depends on the order in which they were accumulated.

Before we describe the accumulator's construction, it is necessary to, first, introduce some additional notation and, second, describe the notion of a *minimal subtree*. Camacho et al. [53] denote the collection of all nodes in a hash tree T, be it a subtree or not, as $\mathcal{V}(T)$. We denote as GenMinSubTree(T, S) a procedure which, on input a hash tree T and a set of node values $S = \{S_i : Val_N = S_i, \text{for some node } N \in \mathcal{V}(T)\}$, outputs a minimal subtree T' of T. In addition, we denote as VerifyTree(T, Acc) a procedure which, on input a (sub) tree and the value of the accumulator Acc, computes $Proof_{Root_T}$ and ouputs

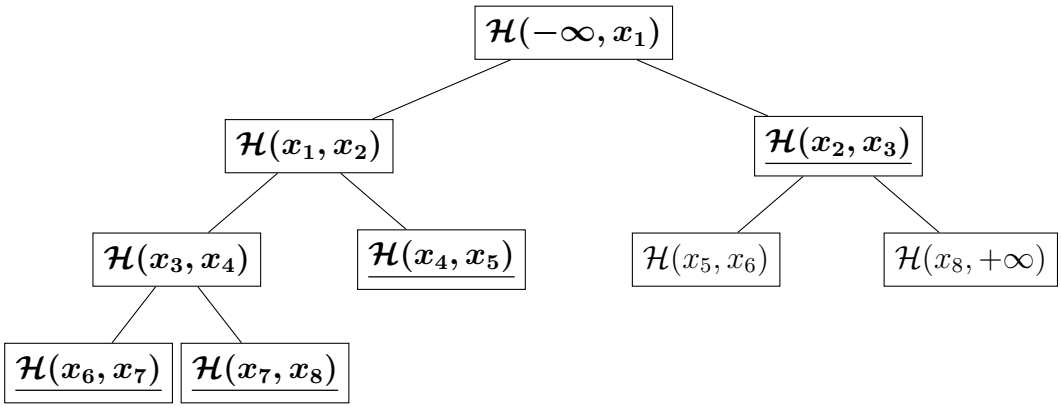

Figure 6.1: Example of a hash tree T that corresponds to the accumulated set $X = \{x_1, ..., x_8\}$. The minimal subtree T', where $\mathcal{H}(x_3, x_4)$ is the starting node's value, is comprised by the nodes in bold font.

1, if $Acc = Proof_{Root_T}$, otherwise, it outputs 0. For clarity, we state that the notation of the aforementioned two procedures is not part of the work of Camacho et al. [53]. However, we decided to include them here since we believe it will enhance the readability of the description of the accumulator's construction. In the following, we illustrate the steps involved in these algorithms.

The algorithm GenMinSubTree(T, S), for each node value $S_i \in S$, finds $N_i \in \mathcal{V}(T)$, such that $Val_{N_i} = S_i$, and adds to T' all nodes of T that belong to all paths from $Root_T$ to N_i, as well as, the children of these nodes (duplicate nodes are discarded). We illustrate how T' is generated via an example where, for simplicity, we have one starting node whose value is $S_1 = \mathcal{H}(x_3, x_4)$ (Figure 6.1). The nodes that belong to all paths from $Root_T$ to the node with value S_i have values $\{\mathcal{H}(-\infty, x_1), \mathcal{H}(x_1, x_2), \mathcal{H}(x_3, x_4)\}$ and are depicted in bold font. Next, we need to add to T' the children of all the aforementioned nodes, which are underlined and in bold font in Figure 6.1. The algorithm VerifyTree(T, Acc) is a simple recursive computation of all node proofs in all of the paths of T, starting from each path's leaf nodes and leading all the way up to $Root_T$, which is required to correctly compute $Proof_{Root_T}$. Clearly, this procedure is very similar to how hash paths are verified in traditional Merkle trees.

In Figure 6.2, we present the construction of the hash tree-based universal accumulator of Camacho et al. [53]. The Setup(1^λ) algorithm is run by the accumulator manager, which initializes a new accumulator instance, based on the input security parameter λ. The Witness(Acc, m, x) algorithm, on input the accumulator's value Acc, its memory m and an element x $\in D$, computes a (non) membership witness W for x, depending on whether $x \notin X$ or not. The corresponding witness verification algorithm, Belongs(Acc, x, W), outputs 1, if W is a valid membership witness for x, 0, if W is a valid non membership witness for x, and $\perp$, in any other case. The algorithm Update$_{\text{op}}$(Acc$_{\text{before}}$, m$_{\text{before}}$, x) updates the accumulator's value by either adding (op = add) or removing (op = del) the element x to/from the accumulated set X_{before}. It outputs the updated values of the accumulator (Acc$_{\text{after}}$) and its memory (m$_{\text{after}}$), as well as, an update witness W$_{\text{op}}$. The latter witness,

The domain of the accumulator is $D = \{0,1\}^{\lambda}$:

- Setup(1^{λ}): Set $X \leftarrow \emptyset$ and initialize T to a single root node N_m, where $Val_{N_m} = \mathcal{H}(-\infty||+\infty)$ and $Proof_{N_m} = \mathcal{H}(Val_{N_m}||\epsilon||\epsilon)$. Output (m_0, Acc_0), where $m_0 = (X, T)$ and $Acc_0 = Proof_{N_m}$.
- Witness(Acc, m, x): If $x \in X$, set $w_1 \leftarrow (x_a, x_b)$ where, $x = x_a$ or $x = x_b$. If $x \notin X$, set $w_1 \leftarrow (x_a, x_b)$ where, $x_a \prec x \prec x_b$. Lastly, set $w_2 \leftarrow$ GenMinSubTree(T, $\{\mathcal{H}(\mathsf{x_a}, \mathsf{x_b})\}$), and output $W = (w_1, w_2)$.
- Belongs(Acc, x, W): Evaluate conditions: 1) VerifyTree($\mathsf{w_2}$, Acc) $= 1$, 2) $Val_N = \mathcal{H}(x_a, x_b) \wedge N \in \mathcal{V}(w_2)$, 3) $x = x_a \vee x = x_b$ and, 4) $x_a \prec x \prec x_b$. Output 1, if (1), (2) and (3) hold, 0, if (1), (2) and (4) hold, and $\perp$ otherwise.
- Update$_{\mathsf{op}}$(Acc$_{\mathsf{before}}$, m$_{\mathsf{before}}$, x):
  - If op $=$ add and $x \notin X_{before}$, find $x_a, x_b \in X_{before}$ and $N \in T_{before}$, s.t. $x_a \prec x \prec x_b$ and $Val_N = \mathcal{H}(x_a, x_b)$. Set $T_{after} \leftarrow T_{before}$, $Val_N \leftarrow \mathcal{H}(x_a, x)$ in T_{after}, add a leaf N' to T_{after}, s.t. $Val_{N'} = \mathcal{H}(x, x_b)$, T_{after} is balanced and update node proofs. Set $\mathsf{w_{add_1}} \leftarrow$ GenMinSubTree($\mathsf{T_{before}}, \{\mathcal{H}(\mathsf{x_a}, \mathsf{x_b}), \mathsf{Val_{Parent_{N'}}}\}$), $\mathsf{w_{add_2}} \leftarrow$ GenMinSubTree($\mathsf{T_{after}}, \{\mathcal{H}(\mathsf{x_a}, \mathsf{x}), \mathsf{Val_{Parent_{N'}}}\}$), $X_{after} \leftarrow X_{before} \cup \{x\}$, $\mathsf{W_{op}} \leftarrow (\mathsf{add}, \mathsf{w_{add_1}}, \mathsf{w_{add_2}})$, $m_{after} \leftarrow (X_{after}, T_{after})$ and $Acc_{after} \leftarrow Proof_{Root_{T_{after}}}$.
  - If op $=$ del and $x \in X_{before}$, find $x_a, x_b \in X_{before}$ and $N_1, N_2, N_3 \in T_{before}$, such that $x_a \prec x \prec x_b, Val_{N_1} = \mathcal{H}(x_a, x), Val_{N_2} = \mathcal{H}(x, x_b)$ and N_3 is the last (rightmost) leaf node of T_{before}. Set $T_{after} \leftarrow T_{before}, Val_{N_1} \leftarrow \mathcal{H}(x_a, x_b)$ and $Val_{N_2} \leftarrow Val_{N_3}$ in T_{after}, delete N_3 from T_{after} so that it remains balanced and node proofs are updated, set $X_{after} \leftarrow X_{before} \setminus \{x\}$, $\mathsf{w_{del_1}} \leftarrow$ GenMinSubTree($\mathsf{T_{before}}, \{\mathsf{Val_{N_1}}, \mathsf{Val_{N_2}}, \mathsf{Val_{N_3}}\}$), $\mathsf{w_{del_2}} \leftarrow (x_a||x_b)$, $\mathsf{w_{del_3}} \leftarrow$ GenMinSubTree($\mathsf{T_{after}}, \{\mathcal{H}(\mathsf{x_a}, \mathsf{x_b})\}$), $\mathsf{W_{op}} \leftarrow (\mathsf{del}, \mathsf{w_{del_1}}, \mathsf{w_{del_2}}, \mathsf{w_{del_3}})$, $\mathsf{m_{after}} \leftarrow (\mathsf{X_{after}}, \mathsf{T_{after}})$, and $\mathsf{Acc_{after}} \leftarrow \mathsf{Proof_{Root_{T_{after}}}}$.

  Output $(\mathsf{Acc_{after}}, \mathsf{m_{after}}, \mathsf{W_{op}})$.
- CheckUpdate($\mathsf{Acc_{before}}, \mathsf{Acc_{after}}, \mathsf{x}, \mathsf{W_{op}}$) : If $\mathsf{W_{op}} = (\mathsf{add}, \mathsf{w_1}, \mathsf{w_2})$, evaluate conditions:
  1. w_2 is a tree produced by adding a leaf node to w_1.
  2. There exist $\mathsf{N_1}, \mathsf{N_2} \in \mathcal{V}(\mathsf{w_2})$, such that $\mathsf{Val_{N_1}} = \mathcal{H}(\mathsf{x_a}, \mathsf{x})$ and $\mathsf{Val_{N_1}} = \mathcal{H}(\mathsf{x}, \mathsf{x_b})$.
  3. Apart from the node with value $\mathcal{H}(x_a, x_b)$ in w_1 and nodes $\mathsf{N_1}, \mathsf{N_2}$ of w_2, all other nodes of w_1 and w_2 have the same values.
  4. VerifyTree($\mathsf{w_1}, \mathsf{Acc_{before}}$) $= 1$.
  5. VerifyTree($\mathsf{w_2}, \mathsf{Acc_{after}}$) $= 1$.

  If all these conditions are true, output 1, otherwise, output 0. We omit the case where $\mathsf{W_{op}} = (\mathsf{del}, \mathsf{w_1}, \mathsf{w_2}, \mathsf{w_3})$, which is similar to when op $=$ add.

Figure 6.2: Construction of the hash-tree based universal accumulator of Camacho et al. [53].

W_{op}, can be input to CheckUpdate$(Acc_{before}, Acc_{after}, x, W_{op})$ to verify that the accumulator was updated correctly.

The accumulator of Camacho et al. [53] is a public-state, additive, universal accumulator with the following differences and additional features. First and foremost, the accumulator is *strong*, i.e., the accumulator manager is not required to be trusted. Informally, a strong accumulator does not require a trusted setup (i.e., there is no KeyGen algorithm as in the RSA case). Second, it supports the deletion of elements without relying on secret information. Third, it allows for updates (additions and deletions) which are publicly verifiable, via the CheckUpdate() algorithm. The accumulator's value is of constant size, however, (non) membership and update witnesses have $\mathcal{O}(\lambda \log(n))$ bit size, where n is the number of accumulated elements.

6.2 Hash tree-based PKI

6.2.1 Construction

In this section, we propose our second PKI construction, which employs the accumulator of Camacho et al. [53] that we presented previously. Since this accumulator supports publicly verifiable updates clients can locally update accumulator values and input to the smart contract witnesses that prove the operations were performed honestly. To generate all involved witnesses, clients query $\mathcal{F}_{UDB}$ for the history of operations. Figure 6.3 illustrates program P_{Hash} for this implementation, which T inputs to F_{TP} during setup.

1. On input $(\text{Setup}, params)$, where $params$ is of the form $(\lambda_1, \lambda_2, R)$, compute $c_1 \leftarrow \text{Setup}(1^{\lambda_1})$ and $c_2 \leftarrow \text{Setup}(1^{\lambda_2})$. Set $state \leftarrow (params, c_1, c_2)$ and return $state$.
2. On input $(\text{Register}, id, pk, W_2, c_{add_1}, W_{add1}, c_{add_2}, W_{add2})$,
   (a) If $\text{CheckUpdate}(c_1, c_{add_1}, (id, pk), W_{add_1}) = 0$, return fail.
   (b) If $\text{CheckUpdate}(c_2, c_{add_2}, id, W_{add_2}) = 0 \vee \text{Belongs}(c_2, id, W_2) \neq 0$, return fail.
   Update $state$ by setting $c_1 \leftarrow c_{add_1}$, $c_2 \leftarrow c_{add_2}$ and return success.
3. On input $(\text{Revoke}, id, pk, W_1, c_{del_1}, W_{del_1}, c_{del_2}, W_{del_2}, \text{aux})$,
   (a) If $R(pk, \text{aux}) = 0$, return fail.
   (b) If $\text{Belongs}(c_1, (id, pk),) \neq 1 \vee \text{CheckUpdate}(c_1, c_{del_1}, (id, pk), W_{del_1}) = 0$, return fail.
   (c) If $\text{CheckUpdate}(c_2, c_{del_2}, id, W_{del_2}) = 0$, return fail.
   Update $state$ by setting $c_1 \leftarrow c_{del_1}$, $c_2 \leftarrow c_{del_2}$ and return success.
4. On input RetrieveState, return $state \leftarrow (params, c_1, c_2)$.

Figure 6.3: The program P_{Hash}, which is input to $\mathcal{F}_{TP}$ during initialization, in our Hash tree-based PKI construction.

To register an (id, pk) pair, a client generates a non membership witness W_2 for id in c_2. She accumulates (id, pk) in c_1, which produces an update witness W_{add_1} and the accumulator's updated value c_{add_1}. She also accumulates id in c_2, which produces an

update witness W_{add_2} and the accumulator's updated value c_{add_2}. Assuming that witness values are computed honestly by the client, the contract will validate them by invoking the appropriate verification functions of the accumulator, i.e., Belongs(c_2, id, W_2), CheckUpdate$(c_2, c_{add_2}, id, W_{add_2})$ and CheckUpdate$(c_1, c_{add_1}, (id, pk), W_{add_1})$, and update its accumulator values. Lastly, the client posts $(\text{Register}, id, pk)$ to $\mathcal{F}_{UDB}$.

To revoke an (id, pk) pair, a client signs pk ($\sigma_{sk}(pk)$) and generates a membership witness W_1 for (id, pk) in c_1. She deletes id from c_2, which produces an update witness W_{del_2} and the accumulator's updated value c_{del_2}. She also deletes (id, pk) from c_1, which produces an update witness W_{del_1} and the accumulator's updated value c_{del_1}. Assuming that witness values are computed honestly by the client, the contract will validate them by invoking the appropriate verification functions of the accumulator, i.e., Belongs$(c_1, (id, pk), W_1)$, CheckUpdate$(c_2, c_{del_2}, id, W_{del_2})$ and CheckUpdate$(c_1, c_{del_1}, (id, pk), W_{del_1})$, and update its accumulator values. Lastly, the client posts a (Revoke, id, pk) record to $\mathcal{F}_{UDB}$.

In Figure 6.4, we present the formal description of protocol π_{Hash}, which realizes the functionality $\mathcal{F}_{ns}$. The respective memories m_1 and m_2 of c_1 and c_2 are constructed as follows. For each record of the form $(\text{Register}, id, pk)$, clients compute $(c_1, m_1, W_1) \leftarrow \text{Update}_{\text{add}}(\text{c}_1, \text{m}_1, (\text{id}, \text{pk}))$ and $(c_2, m_2, W_2) \leftarrow \text{Update}_{\text{add}}(\text{c}_2, \text{m}_2, \text{id})$. For each record of the form (Revoke, id, pk), clients compute $(c_1, m_1, W_1) \leftarrow \text{Update}_{\text{del}}(\text{c}_1, \text{m}_1, (\text{id}, \text{pk}))$ and $(c_2, m_2, W_2) \leftarrow \text{Update}_{\text{del}}(\text{c}_2, \text{m}_2, \text{id})$. This procedure's complexity is linear to the history of operations performed in the system, which can be improved as follows.

First, we extend the interface of $\mathcal{F}_{UDB}$ to allow for queries and updates on an accumulator's memory. Second, instead of storing registration/revocation records on $\mathcal{F}_{UDB}$, we store the memories of the accumulators. In this new setting, prior to their registration/revocation, clients query $\mathcal{F}_{UDB}$ for the hash trees T_1 and T_2 of accumulators c_1 and c_2, respectively. To construct the required witnesses, clients do not need the full sets of accumulated values X_1 and X_2, but just two values x_a and x_b for each witness to be computed. These two values can also be used to update the accumulator for the same input element. Thus, prior to a registration/revocation, clients query $\mathcal{F}_{UDB}$ for the trees of the accumulators and a total of two pairs of values, (x_{a_1}, x_{b_1}) and (x_{a_2}, x_{b_2}), to compute the involved witnesses and update c_1 and c_2. Following a successful registration/revocation, clients directly update the sets X_1 and X_2 by inserting (id, pk) and id, respectively, and **only** the subtrees of T_1 and T_2 that were affected by the updates. The complexity of this procedure, apart from the initial retrieval of the hash trees T_1 and T_2, is logarithmic and we leave it as future work.

The security of both PKI constructions presented in this work is based on the security property of the underlying accumulator. Due to their inherent similarities, we omit the proof for protocol π_{Hash}, whose security is defined by the following theorem.

Theorem 6.2.1. *The protocol π_{Hash} of Figure 6.4 securely realizes the functionality $\mathcal{F}_{ns}$ of Figure 4.2 in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world under the collision-resistance of the hash function family $\mathcal{H}$.*

1. On input (sid, Init), a server S_i sends (sid, InitTP) and $(sid, \text{InitUDB})$ to $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$. If S_i receives success by both $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, S_i returns success.
2. On input (sid, Setup, R), T sends $(sid, \text{Install}, P_{Hash})$ to $\mathcal{F}_{TP}$, where P_{Hash} is the program of Figure 6.3. If $\mathcal{F}_{TP}$ returns success, T sets $params = (\lambda_1, \lambda_2, R)$ and sends $(sid, (\text{Setup}, params))$ to $\mathcal{F}_{TP}$, which executes P_{Hash} on input $(\text{Setup}, params)$ (if $\mathcal{A}$ returns allow to $\mathcal{F}_{TP}$). If $\mathcal{F}_{TP}$ returns $state \leftarrow (params, c_1, c_2)$, T outputs success.
3. On input $(sid, \text{Register}, id, pk)$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate$, if the last record related to id is of the form Register, C outputs fail, otherwise C computes:
   (a) A non membership witness $W_2 \leftarrow \text{Witness}(c_2, m_2, id)$.
   (b) The accumulator's updated value, memory and an update witness $(c_{add_1}, m_{add_1}, W_{add_1}) \leftarrow \text{Update}_{\text{add}}(\text{c}_1, \text{m}_1, (\text{id}, \text{pk}))$.
   (c) The accumulator's updated value, memory and an update witness $(c_{add_2}, m_{add_2}, W_{add_2}) \leftarrow \text{Update}_{\text{add}}(\text{c}_2, \text{m}_2, \text{id})$.

   Then, C sends $(sid, \text{Register}, id, pk, W_2, c_{add_1}, W_{add1}, c_{add_2}, W_{add2})$ to $\mathcal{F}_{TP}$, which runs P_{Hash} on this input. If $\mathcal{F}_{TP}$ returns success, C sends $\big(sid, \text{Post}, (\text{Register}, id, pk)\big)$ to $\mathcal{F}_{UDB}$ and ouputs success, or fail otherwise.
4. On input $(sid, \text{Revoke}, id, pk, \text{aux})$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate$, C computes:
   (a) A membership witness $W_1 \leftarrow \text{Witness}(c_1, m_1, (id, pk))$.
   (b) The accumulator's updated value, memory and an update witness $(c_{del_1}, m_{del_1}, W_{del_1}) \leftarrow \text{Update}_{\text{del}}(\text{c}_1, \text{m}_1, (\text{id}, \text{pk}))$.
   (c) The accumulator's updated value, memory and an update witness $(c_{del_2}, m_{del_2}, W_{del_2}) \leftarrow \text{Update}_{\text{del}}(\text{c}_2, \text{m}_2, \text{id})$.

   Then, C sends $(sid, \text{Revoke}, id, pk, W_1, c_{del_1}, W_{del_1}, c_{del_2}, W_{del_2}, \text{aux})$ to $\mathcal{F}_{TP}$, which runs P_{Hash} on this input. If $\mathcal{F}_{TP}$ returns success, C sends $\big(sid, \text{Post}, (\text{Revoke}, id, pk)\big)$ to $\mathcal{F}_{UDB}$ and ouputs success, or fail otherwise.
5. On input $(sid, \text{Retrieve}, id)$, C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate$, C acts as follows:
   (a) If the last record related to id is of the form Revoke, or, if there is no record related to id, C outputs fail. Otherwise, C extracts (id, pk) from the last $(\text{Register}, id, pk)$ record in $DBstate$.
   (b) C sends $(sid, \text{RetrieveState})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $state$, C computes $W_1 \leftarrow \text{Witness}(c_1, m_1, (id, pk))$. If $\text{Belongs}(c_1, (id, pk), W_1) = 1$, C outputs pk as the retrieved public key, otherwise, C outputs fail.
6. On input $(sid, \text{VerifyID}, id)$, C runs Step 5. If Step 5 outputs some pk, C outputs 1, otherwise, C outputs 0.
7. On input $(sid, \text{VerifyMapping}, id, pk)$, C runs Step 5. If Step 5 outputs pk, C outputs 1, otherwise, C outputs 0.

Figure 6.4: Description of the protocol π_{Hash} built upon the program P_{Hash} of Figure 6.3.

6.2.2 Using only one accumulator

To register an (id, pk) pair, a client generates a non membership witness W for id in c. She accumulates (id, pk) in c, which produces an update witness W_{add_1} and the accumulator's updated value c_{add_1}. She accumulates id in c_{add_1}, which produces an update witness W_{add_2} and the accumulator's updated value c_{add_2}. The Register operation of the smart contract is modified and performs, in order, the following verifications: 1) Belongs$(c, id, W) = 0$, 2) CheckUpdate$(c, c_{add_1}, (id, pk), W_{add_1}) = 1$ and, 3) CheckUpdate$(c_{add_1}, c_{add_2}, id, W_{add_2}) = 1$. If all the aforementioned verifications succeed, the registration is valid and the contract will set $c \leftarrow c_{add_2}$. Lastly, the client posts $(\text{Register}, id, pk)$ to $\mathcal{F}_{UDB}$. To revoke an (id, pk) pair, a client, as previously, signs pk ($\sigma_{sk}(pk)$) and generates a membership witness W for (id, pk) in c. She deletes id from c, which produces an update witness W_{del_1} and the accumulator's updated value c_{del_1}. She deletes (id, pk) from c_{del_1}, which produces an update witness W_{del_2} and the accumulator's updated value c_{del_2}. The Revoke operation of the smart contract is modified and performs, in order, the following verifications: 1) Belongs$(c, (id, pk), W) = 1$, 2) CheckUpdate$(c, c_{del_1}, id, W_{del_1}) = 1$ and, 3) CheckUpdate$(c_{del_1}, c_{del_2}, (id, pk), W_{del_2}) = 1$. If all the aforementioned verifications succeed and $R(pk, \text{aux}) = 1$, the revocation is valid and the contract will set $c \leftarrow c_{del_2}$. Lastly, the client posts a (Revoke, id, pk) record to $\mathcal{F}_{UDB}$. The tradeoff in this case is that the contract's state stores only one accumulator (c) instead of two. The size of the witnesses is slightly bigger but still logarithmic, i.e., $\mathcal{O}(\lambda \log(2n))$.

7. EVALUATION

7.1 Experimental Setup and Preliminary Results

In this section, we present experiments that measure the cost of running on Ethereum the constructions of Sections 5.2 and 6.2, as well as, their building blocks. We intersperse our results with recommendations for modifications and improvements to Ethereum that, we believe, are vital if Ethereum (or any smart contract platform) is to reach its maximum potential of supporting arbitrary, large-scale, distributed applications. We create a private blockchain that is maintained by a single mining node. This eliminates the waiting time that transactions would have in either the live or the test chain to be mined into a block. Thus, we are able to measure accurately transaction gas costs and perform experiments on a larger scale. We use *geth* (v.1.8.17, [20]), the official Ethereum client and conduct our experiments via the *truffle* testing suite (v.4.1.13, [32]). Lastly, we use randomly generated 32-byte identities.

Our implementations[1] employ a variety of primitives, e.g., signatures and accumulators. One option would be to deploy each primitive as a separate library and have the front-end PKI contract issue appropriate message calls. Unfortunately, this option is the most expensive in terms of gas due to the extra cost of message calls (700 gas) and the increased cost of reading the deployment address(es) of the library contract(s) from storage. The more efficient option is to pack all back-end logic into a single library and link it with the front-end PKI contract. This eliminates the aformentioned costs. Thus, Ethereum imposes the following tradeoff. On the one hand, developers will tend to pick the second option, as one of their main incentives is to minimize gas cost. On the other hand, the first option: 1) promotes modular programming, 2) leads to the construction of an on-blockchain "standard library", similar to what common programming languages have and, most importantly, 3) mitigates duplicate logic, i.e., excess, duplicate state and code in the blockchain. Thus, reducing the costs of the first option will aid in the development of future applications and incentivize developers to adopt more modular programming approaches.

Recommendation #1: Significantly reduce the cost of issuing message calls to libraries.

Our first experiment provides insight regarding the overhead of a library implementation, compared to a precompiled contract. Precompiled contracts reside on well-known, static addresses and require less gas because their code does not run in EVM assembly, but in machine language of the physical node hosting the miner. We evaluate the cost of verifying 1,000 Secp256k1 elliptic curve signatures, based on the library contract of [41]. We measure a mean cost of 827,765.53 gas, with a standard deviation of 6,021.64 gas. At the time of this writing, the average *gas limit* of blocks is about 8 million gas ([14]). Thus, signature verification on the library contract consumes 10.3% of the current block gas limit, which is substantial. In contrast, Ethereum's *ecrecover* precompiled contract, which oper-

[1] The code of our implementations can be found at [19].

Table 7.1: Min, max, mean and standard deviation (columns 2-5) of the gas cost of: 1) 10,000 modulo multiplications, exponentiations and primality tests, 2) 10,000 accumulations of primes (Add) and (non) membership witness verifications (VerifyMem, VerifyNonMem) 3) 1,000 mappings (Map) of strings to primes and, 4) registrations (Register, for $(i = 1)$ and $(i \geq 2)$) and revocations (Revoke) of 1,000 (id, pk) pairs in the RSA-based PKI.

Operation	Gas Cost			
	Min	Max	Mean	Std
Mod. Mul.	179,556	182,900	181,470.76	639.16
Mod. Exp.	678,074	745,517	741,846.48	5,001.49
Primality Test	1,481,160	1,502,502	1,490,219.13	5,480.23
Add	810,030	810,158	810,153.32	17.61
VerifyMem	755,130	755,796	755,537.23	126.63
VerifyNonMem	1,473,345	1,525,685	1,519,279.96	5,386.87
Map	1,733,124,331	2,550,435,741	2,141,780,036	577,926,440.35
Register ($i = 1$)	89,801,425	8,620,016,945	1,681,994,990	1,313,539,096.67
Register ($i \geq 2$)	89,160,026	10,676,126,282	2,575,538,734.5	1,715,254,997.91
Revoke	440,467,878	13,805,874,517	3,598,585,618	1,910,918,965.1

ates on the same curve, costs only 3,000 gas. Thus, the cost of the library implementation is two orders of magnitude higher, which illustrates the benefits and importance of having built-in support for a variety of cryptographic operations. In the evaluation of all the constructions that follow, we have modified the library contract of [41] to operate on the Secp256r1 curve. We repeat the same experiment and measure the mean cost of signature verification to be 1,257,103.26 gas, with a standard deviation of 9,178.44 gas.

7.2 RSA-based PKI Evaluation

We evaluate the RSA-based PKI of Section 5.2, which uses the following constructs: signature verification via the Secp256r1 library contract and arbitrary precision integer arithmetic, based on the Big Number library developed by Zerocoin [33]. This library supports operations which are relevant to this construction, such as modulo exponentiation and the Miller-Rabin probabilistic primality test. We modify the implementation of the primality test because: 1) it supports only a range of integers, whilst, our implementation supports all integers, 2) the original algorithm is seeded by externally provided randomness, which we modify to be based on the hash of the last block, thus, limiting the adversary's knowledge and influence on its output and, 3) it does not perform sufficient iterations, which we modify to comply to the NIST standard ([25]). The third construct is the RSA accumulator, which encompasses the Map procedure. The RSA moduli of the accumulators are 3072 bits long, thus, they provide 128-bit security ([26]). We set Map's parameter to $k = 65$, i.e., Map outputs $3k = 195$ bit primes. Thus, except for $1/2^{65}$ fraction of functions $f \in U$, a string will be mapped to a prime number, except with negligible probability ([68]), which

we deem reasonable.

We conduct four sets of experiments where the exponents, moduli and exponentiation bases are 195, 3072 and 3072 bits long, respectively. First, we evaluate the operations of the Big Number library that are relevant to this construction by running 10,000 primality tests, modulo multiplications and exponentiations, respectively. Second, we evaluate the RSA accumulator by running 10,000 iterations of each of the following operations: 1) accumulations (Add) of 195-bit prime numbers, and 2) (non) membership witness verifications (VerifyMem, VerifyNonMem). Third, we measure the cost of mapping 1,000 strings to 195-bit primes (Map). Fourth, we measure the cost of registering (Register) and revoking (Revoke) 1,000 (id, pk) pairs in the RSA-based PKI. Recall that registration differentiates between two cases, i.e., whether an identity is registered for the first time ($i = 1$), or not ($i \geq 2$). Table 7.1 illustrates the results.

Regarding the Big Number library experiments, Table 7.1 shows that modulo exponentiation and primality testing are the more expensive operations. The former is based on one of Ethereum's precompiled contracts ([51]) and its cost is dominated by the exponent's length, especially in cases where it is larger than 32 bytes. In addition, this operation is invoked in the (main) witness loop of the Miller-Rabin test, thus, the data suggest that, on average, the primality test performs two loop iterations. To compute the cost of the RSA accumulator's operations, we have to factor in the cost of reading from storage the accumulator's value, its exponentiation base and its modulus (a total of 36 EVM words). The cost of reading an EVM word (32 bytes) from storage is 200 gas, thus, $36 \times 200 = 7,200$ extra gas. The VerifyMem operation involves one modulo exponentiation. The VerifyNonMem operation involves two modulo exponentiations and one modulo multiplication. Thus, as the data suggest, the gas cost of these two operations follows directly from that of reading the appropriate values from the contract's storage and the invoked operations of the Big Number library. The Add operation involves one modulo exponentiation and modifies the accumulator's value. Thus, in addition to the aforementioned cost of reading from the contract's storage, there is also the cost of storing the accumulator's updated value to the contract's state. Updating an EVM word on storage costs 5,000 gas. The accumulator is 12 EVM words long, thus, $12 \times 5,000 = 60,000$ extra gas, or, a total of $60,000 + 7,200 = 67,200$ gas.

The key result of this section is our demonstration of the practical implications of the RSA-based PKI's design. Recall that in this construction, the smart contract's state and (non) membership witnesses have constant size at the expense of computational overhead. This tradeoff is embodied by the $O(k^2)$ complexity of the Map procedure, which is involved in both the Register and Revoke operations of the RSA-based PKI to map the contract's inputs to prime numbers. These prime numbers are then input to the appropriate witness verification algorithms of the RSA accumulator. For instance, the Revoke operation involves one signature verification, one invocation of VerifyNonMem and VerifyMem and three invocations of Map. We have already illustrated that the cost of signature verification and of the RSA accumulator's operations is deterministic, however, the same cannot be said about Map. While Map is deterministic in terms of its output, the number of iterations it performs to produce its output is not. Thus, we can have cases where one invocation of

Map costs more than a Register or Revoke operation of the RSA-based PKI, as illustrated by the heavily skewed data of Table 7.1. Consequently, Map dominates the cost of the RSA-based PKI's operations. We perform an additional experiment where we measure the cost of running **one iteration** of Map on input of 100,000 strings. We measure the mean cost to be 9,488,542 gas, with standard deviation of 17,794 gas. Thus, even one iteration of Map exceeds Ethereum's block gas limit.

Result #1: The provably secure, RSA-based smart contract PKI is not viable on Ethereum.

Discussion: There are two reasons why Map's gas cost is so high. First, in Ethereum, it is cheaper to access one EVM word (32 bytes) than one byte. This "was chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations", as stated in Ethereum's yellow paper ([107]). It has nothing to do with efficiency as no real-world physical machine, on top of which the EVM runs, supports 32 byte words. Therefore, one potential improvement would be to modify Ethereum's cost model to account for this contradiction. For instance, accessing a single byte could be simply tuned to $\frac{1}{32}$ of the cost of loading an EVM word. This change, apart from being more fair, allows for more packed data encodings, which can reduce transaction size and, as a result, the blockchain's size. Second, Map's computation revolves around bit operations which are, currently, very expensive in Ethereum as they have to be performed via the EVM's integer exponentiation function. For instance, setting one bit of a memory byte array costs 586 gas. However, in the near future, the EVM will support bitwise shifting ([39]), which will only cost 3 gas and, thus, will provide substantial improvements for Map.

Recommendation #2: Ethereum's cost model should be modified to account for the granularity of the data that are accessed.

For the RSA-based PKI, there are alternatives such as verifiable computation that will also benefit from all the aforementioned propositions. In this setting, the smart contract will need only to verify proofs that the client computed Map correctly, instead of invoking Map itself. We leave this as future work.

7.3 Hash tree-based PKI Evaluation

We evaluate the Hash tree-based PKI, which we proposed in Section 6.2. This construction employs the following primitives: 1) the Secp256r1 library contract for signature verification and, 2) the Hash tree-based accumulator, which we presented in Section 6.1. We employ the SHA-256 hash function, which is exposed as a precompiled contract in Ethereum. The accumulators were initialized according to the sizes of their respective input elements. Thus, since c_1 accumulates (id, pk) pairs, we set $\lambda_1 = 768$ bits, and, since c_2 accumulates identities, we set $\lambda_2 = 256$ bits. Recall that in the Hash tree-based PKI, witnesses are a logarithmic function of the number of accumulated elements, in contrast to the RSA-based PKI where they are constant-sized.

In our first experiment, we accumulate a total of 100,000 elements and measure the cost

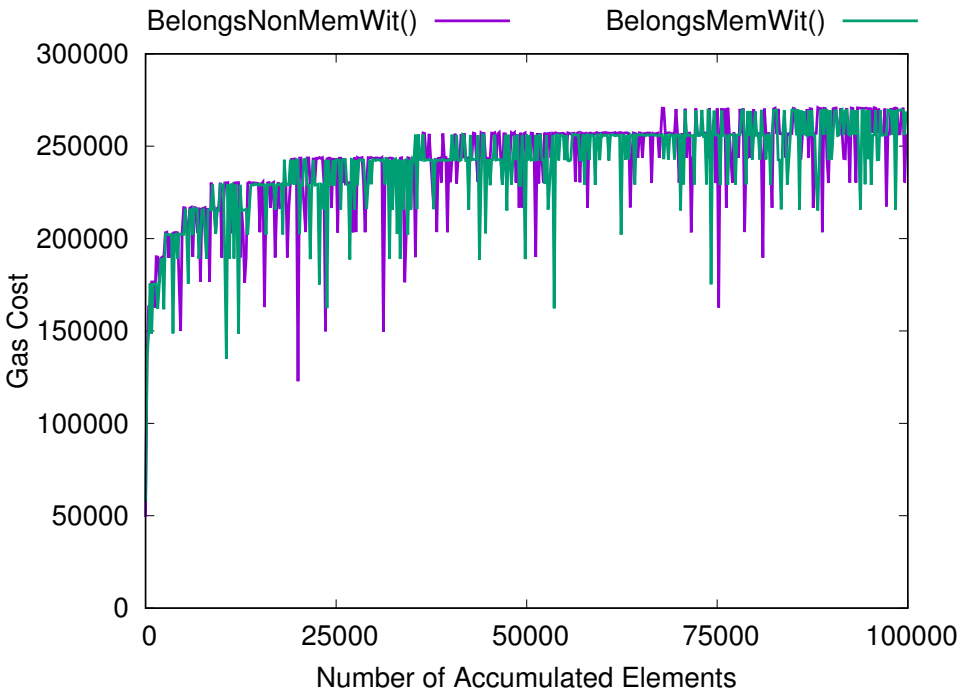

Figure 7.1: Gas cost versus the number of accumulated values of 100,000 membership (green line) and non membership (purple line) witness verifications of the hash tree-based universal accumulator.

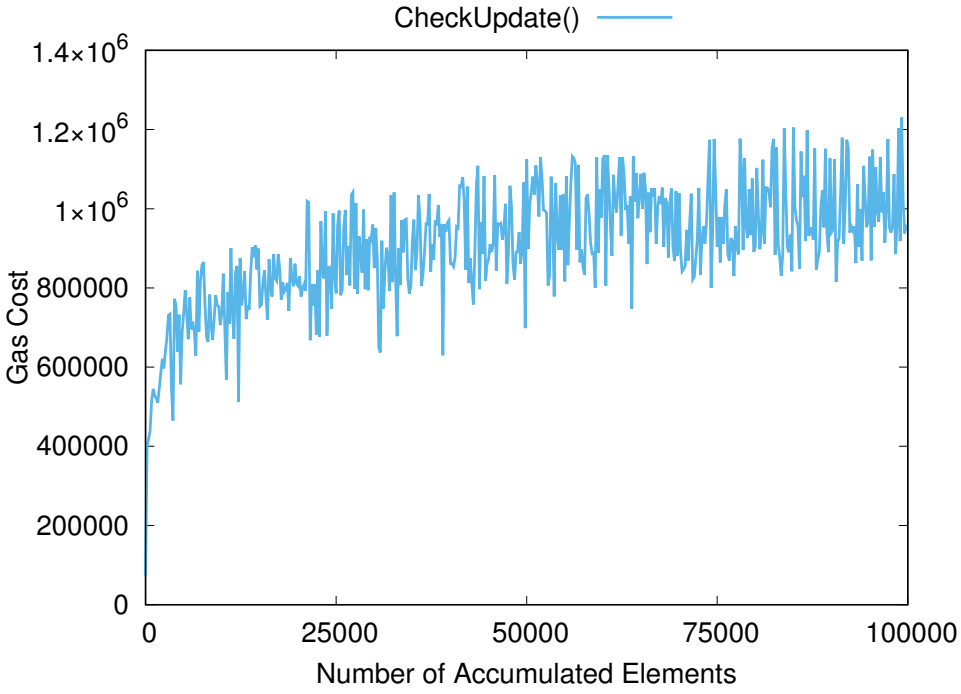

Figure 7.2: Gas cost versus the number of accumulated values of 100,000 update verifications (CheckUpdate()**) of the hash tree-based universal accumulator.**

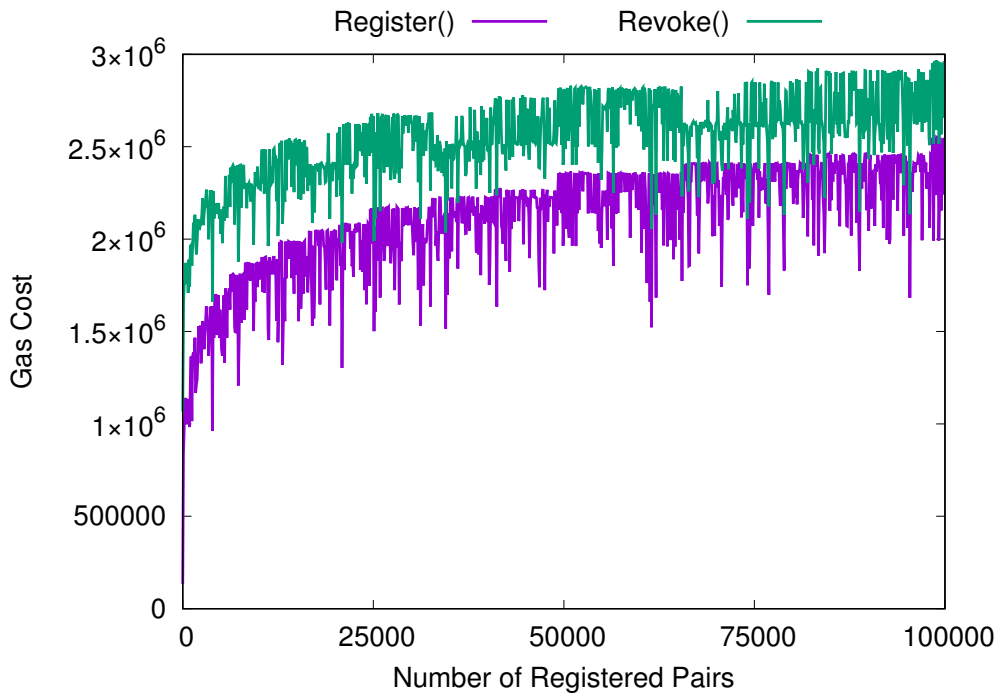

Figure 7.3: Gas cost versus the number of registered (id, pk) pairs of 100,000 registrations (purple line) and revocations (green line) of our hash tree-based PKI.

of the Hash tree-based accumulator's operations. In Figure 7.1, we plot the gas cost versus the number of accumulated elements of verifying membership (green line) and non membership (purple line) witnesses. In Figure 7.2, we plot the gas cost versus the number of accumulated elements of verifying accumulator updates (CheckUpdate$()$). The general trend resembles, as expected, that of a logarithmic function. However, the curves have a large number of dips. This is because proofs are paths of the accumulator's hash tree. The size of each proof depends on the position of the starting node(s) in the hash tree and, thus, its verification cost varies. As illustrated in the graphs, verifying accumulator updates is more expensive than that of (non) membership witnesses. Indeed, the size of the former proofs tends to be two, or even, three times the size of the latter, which is reflected in their respective verification costs.

In our second experiment, we evaluate our Hash tree-based PKI. In Figure 7.3, we plot the gas cost versus the number of registered pairs of registering and revoking 100,000 (id, pk) pairs. Results illustrate that revocation is the more costly procedure as it involves the added cost of verifying signatures. The cost of the most expensive revocation is 2,999,214 gas, i.e., 37.4% of the current block gas limit. Thus, in terms of gas cost, this construction can be deployed on Ethereum's live chain.

Result #2: Our probably secure Hash tree-based PKI construction is viable for deployment on the live chain of Ethereum.

Discussion: The Hash tree-based PKI is best suited for small, to moderately sized PKIs, since the involved proofs are not of constant size. Assuming that the number of registered

pairs monotonically increases, there will come a point where verifying proofs will exceed the block's gas limit. Recall that the cost of a transaction is a function of its computational complexity **and** its byte size. One might argue that this issue can be balanced out by an increase in the block's gas limit, which is certainly the observed trend up to the time of this writing ([14]). However, a miner's main incentive is to produce (hash) blocks as fast and with as low operational costs as possible. Thus, it can be expected that the increase in the block's gas limit will, eventually, plateau. This line of reasoning assumes that the blockchain's consensus mechanism revolves around Proof-of-Work (PoW), as is currently the case. However, Ethereum is planning to replace PoW with Proof-of-Stake (PoS), a consensus protocol that requires a small amount of computation. Discussing PoS is out of the scope of this paper, however, it is reasonable to assume that it will change the incentives of miners. Indeed, in this computationally light paradigm, miners might be willing to expend their computational resources to mine blocks that contain larger transactions to maximize their rewards. This will gradually increase the block's gas limit which, as a result, will favor the scale of our Hash tree-based PKI even more.

We now illustrate a few important points regarding hash functions and precompiled contracts. Ethereum supports three hash functions: 1) RIPEMD160, whose computation costs 600 gas plus 120 gas per input word, 2) SHA-256, whose computation costs 60 gas plus 12 gas per input word, 3) KECCAK-256, whose computation costs 30 gas plus 6 gas per input word. Functions (1) and (2) comply with the NIST standard and are implemented as precompiled contracts, however, function (3), does not comply with the standard and is implemented as an EVM opcode. These distinctions have interesting implications. Because functions (1) and (2) are precompiled contracts, they incur the extra gas cost of a message call (700 gas), while function (3) does not. Consequently, Ethereum's cost model encourages the use of a non-standard-compliant hash function. Thus, application developers are forced to either code the client side (at least in part) in JavaScript, for the sole purpose of having access to Ethereum's non-standard implementation of (3), or, pay the extra gas cost. As one of the main incentives of developers in these platforms is to minimize gas costs, the aforementioned distinctions essentially *encourage* client implementations that are unnecessarily complicated and limit the use of standard, mature and efficient libraries, such as *libgcrypt* ([70]). Recently, a proposal has been submitted ([48]) to address this issue. If accepted, this change will further diminish the gas cost of the Hash tree-based PKI, thus, increasing its deployment scale.

Recommendation #3: Reduce message call costs for precompiled contracts and equalize the costs of all supported hash functions.

Lastly, Ethereum is inconsistent in the way it handles and exposes more complicated instructions. Given that: 1) the size of EVM assembly opcodes is one byte, 2) most values are already in use ([18]) and, 3) the purpose of assembly language is to provide access to low-level instructions, we believe that more sophisticated functionality (e.g., hash functions) should be offloaded to a standard library of precompiled contracts. Furthermore, Ethereum should design and incorporate a more developer-friendly way of addressing these contracts. Currently, developers need to memorize (or look-up) the address of each precompiled contract, e.g., SHA-256 resides on address 0x02, which is cumber-

some. Convenient helper functions in EVM assembly that are translated to the appropriate message calls would be helpful.

Recommendation #4: Sophisticated functionality should be moved to a standard library of precompiled contracts that can be addressed in a developer-friendly manner.

7.4 Linear State PKI Evaluation

We now present experiments that evaluate a simple smart contract PKI which stores all (id, pk) pairs in its state. This is the approach that prior proposals employ (e.g., [15, 38, 111]), including the Ethereum Name Service, and allows us to illustrate the shortcomings of Ethereum's pricing of storage. In this scheme, registration and revocation are straightforward. During registration, the contract checks if there is an entry for the input identity in its state. If there isn't one, it adds it. During revocation, the contract first validates the input signature, as in the prior two constructions, and checks if there is an entry in its state for that identity. If so, the contract simply removes it from its state. We evaluate this PKI's operations by registering and revoking 10,000 (id, pk) pairs. Table 7.2 illustrates the gas cost of registering and revoking 10,000 (id, pk) pairs in this setting. During revocation, part of the contract's storage is freed, and the transaction is refunded gas. As a result, the overall cost of revocation is less than the verification of a Secp256r1 signature.

Table 7.2: Min, max, mean and standard deviation (columns 2-5) of the gas cost of registering and revoking 10,000 randomly generated (identity,public-key) pairs in the Linear State PKI contract.

Operation	Gas Cost			
	Min	Max	Mean	Std
Register	89,469	89,661	89,643.91	33.74
Revoke	904,197	949,505	931,150.28	6,410.29

Discussion: This is, currently, the least costly PKI to deploy on Ethereum and the reasons are simple. First, it is light in terms of computation. Indeed, excluding signature verification, which is revocation's dominant cost, the contract spans 10 lines of Solidity code, consisting only of a few *if* statements. Second, as stated by Buterin ([52]), storage is extremely underpriced and, as of yet, there is no incentive for freeing it. However, the issue of storage and its effect on the size of Ethereum's state is complex. Miners add transactions to their blocks according to their "private cost", i.e., their local resource expenditure. Regardless, their decision affects the entire network, as all miners have to download and validate newly mined blocks. For instance, the Linear State PKI may be preferable to miners with abundant and inexpensive disk space. However, that may not be the case for other nodes participating in the protocol. This suggests a "social cost" of transactions, which may not be completely aligned with an individual miner's private costs.

If this social cost of transactions is not completely accounted for, the increasing size of Ethereum's state may deincentivize new full nodes from entering the system. Furthermore, the size of Ethereum's state can also be an obstacle to nodes merely syncing with the system. This issue affects a variety of topics; from light clients (e.g., smartphones) being able to interface with smart contracts, to blockchain security.

One of the proposed countermeasures is imposing small, **static** rent fees on contracts ([105]) so as to avoid being "deactivated", i.e., no one being able to interact with them. However, it is Ethereum's **static** cost model that has caused this problem in the first place. We believe storage is a special commodity and that its price should be dynamically adjusted. Base storage price should depend on the global size of Ethereum's state, i.e., the bigger the size of its state, the higher the base storage price. In addition, the cost of transactions that increase the size of a contract's state should scale accordingly, thus, providing a counterincentive to over-utilizing contract storage.

Recommendation #5: Storage costs should take into account the size of Ethereum's global state, as well as, the size of the invoked contract's state.

Clearly, the issue of storage costs is complex. In the following chapter, we discuss this issue more thoroughly and make concrete proposals on how to address aspects of it. Nonetheless, the constant state PKI constructions discussed in this work are well aligned with space optimal use of smart contract platforms.

8. AN ALTERNATIVE PARADIGM FOR DEVELOPING APPLICATIONS AND PRICING STORAGE ON SMART CONTRACT PLATFORMS

8.1 Rationale

Smart contract platforms provide the means of developing diverse and important decentralized applications (dApps) in a simple manner which, prior to their introduction, was challenging to implement. Ethereum ([107]) is probably the most notable smart contract platform. Its live chain features dApps that implement naming services ([15]), multisignature wallets ([3]), a large variety of fungible tokens ([9]) and even crypto-collectibles ([4]), all in just a few lines of code. The simplicity of developing dApps on top of these platforms stems from the inherent utility of employing the state of smart contracts to store, query and verify the validity of application data. For instance, all implementations of the most widely deployed standard for fungible tokens, i.e., the ERC20 token standard ([7]), store each account's token balance on the contract's state.

Today, Ethereum's cost model does not adequately take into account the amount of storage consumed by smart contracts. This is problematic for several reasons. First, in Ethereum, storing data on the state of smart contracts requires paying *one, non-recurring fee* at the time the data is stored. Thus, regardless of the amount of state that they consume, contracts have zero maintenance costs and can be part of Ethereum's state **forever**. Second, storage-related operations are underpriced, as stated by Ethereum's creator, Vitalik Buterin, in one of his recent talks ([52]). These two factors facilitate contracts that gain utility from storing small amounts of data per user or transaction and have low computational complexity, such as all variants of the "Linear State PKI" of Section 7.4, as well as, ERC20 tokens. As a result, such contracts have very low transaction fees for their operations. Third and most importantly, Ethereum's state must be maintained by all full nodes, yet there is no incentive mechanism in place for freeing storage. If left unchecked, this can have serious consequences. It will diminish the mining population as proportionally fewer and fewer miners will be able to contribute to the network. This will lead to centralization and may prohibit new nodes from joining and syncing to the network, which will, in turn, have a direct impact on Ethereum's security and, utlimately, its longevity.

To address this issue, we propose an alternative paradigm for developing dApps on top of smart contract platforms, which decouples the issue of data storage from verifying data validity. The former is handled by an external, potentially unreliable, storage network that allows efficient access to the application's data (see Section 4.3 for details regarding the storage network). To verify the validity of data obtained from the storage network, we maintain cryptographic accumulators in the smart contract's state. To evaluate our approach, we present a case study of an accumulator-based implementation of the ERC20 token standard. We chose this standard because it is the most widely deployed token standard for fungible tokens, numbering over 130,000 compliant contracts on Ethereum's live chain ([9]). Via minor modifications, our construction can be modified to fit other, upcoming standards, such as the ERC721 standard ([10]) for non-fungible tokens. However,

we stress that our approach can be adapted to any application that requires a verifiable representation of its application data, e.g., naming services, voting systems or any kind of key-value store.

By requiring only minimal (constant-sized) state to be stored on the contract, our approach promotes diversity, scalability, and security of the Ethereum network. Yet, we show that under Ethereum's current cost model, this accumulator-based approach is penalized for the security properties it provides; it is much more (almost prohibitively) costly than the approach of storing each account's token balance in the contract state. This illustrates one of Ethereum's main incentive misalignments. To address this, we revisit Ethereum's storage cost model and propose modifications that: 1) price storage-related operations based on the effort that miners have to expend to execute them, 2) ensure that contracts pay recurring fees proportionate to the amount of storage they consume and the system's overall capacity and, 3) free space consumed by unused/stale contracts. We show that under such a pricing scheme, our accumulator-based ERC20 token construction reduces the incurred transaction fees by up to an order of magnitude. Furthermore, we compare our Hash tree-based PKI (Section 6.2) with the "Linear State PKI" (Section 7.4) under our newly proposed cost model and show that our construction reduces the incurred transaction fees by up to an order of magnitude. With these modifications, we hope the Ethereum developer community will be encouraged to exercise economy in the state consumed by the smart contracts they develop.

8.2 Accumulator-based ERC20 Token

8.2.1 Construction

The ERC20 token standard ([7]) describes the functions and events that facilitate the exchange of arbitrary crypto-assets. At the time of this writing, it is the most widely deployed token standard, numbering over 130,000 compliant smart contracts ([9]) on the Ethereum live chain. Each token holder's account is associated with an Ethereum **address** data type. The token balance of each account is commonly represented as a **uint** data type, i.e., an unsigned integer. The ERC20 token interface is comprised of the following functions:

1. **totalSupply**$()$: Outputs the total supply of tokens accross all accounts.
2. **balanceOf**$($**address owner**$)$: Outputs the token balance of the input account.
3. **approve**$($**address spender**$,$ **uint tokens**$)$; The account that issues the call (transaction) to this function authorizes the "**spender**" account to transfer the specified number of **tokens** from her account.
4. **allowance**$($**address owner**$,$ **address spender**$)$; Outputs the number of tokens that the spender's account is **approve**'d to transfer from the owner's account.
5. **transfer**$($**address to**$,$ **uint tokens**$)$: The account that issues the call (transaction) to this function transfers the specified number of tokens to the "**to**" account.
6. **transferFrom**$($**address from**$,$ **address to**$,$ **uint tokens**$)$: Transfers from the owner's

account ”**from**“ to the **approve**’d account ”**to**“ the specified number of tokens.

To facilitate the aforementioned functionality, ERC20 compliant smart contracts store two mappings, i.e., associative arrays, in their state: 1) *balances*, which maps account addresses to token balances and, 2) *allowed*, which maps account addresses to another mapping where, the latter, maintains the balance that each **approve**’d account is allowed to transfer from the token owner’s account.

We now illustrate how we employ the hash tree, universal accumulator of Camacho et al. [53] (Section 6.1), to realize an accumulator-based ERC20 token. The core idea is to replace each aforementioned mapping with one accumulator. We replace the *balances* mapping with an accumulator, *balancesAcc*, that accumulates (owner,tokens) tuples and allows clients to infer each account’s token balance. For the *allowed* mapping, which is a “double” mapping, we need two accumulators. The first accumulator, *allowedAddressesAcc*, accumulates (owner,spender) tuples and allows clients to infer the accounts that token owners have **approve**’d. The second accumulator, *allowedBalancesAcc*, accumulates (spender,tokens) tuples and allows clients to infer the token balance that **approve**’d accounts are allowed to transfer from the owner’s account. Thus, we have a constant-sized and verifiable representation of account balances and allowances.

The storage network’s state is assumed to be comprised of the *memory* data structure (see Section 6.1) of each of the aforementioned accumulators. As we show below, the interaction with accumulator-based ERC20 smart contracts requires the construction of (non) membership and update witnesses by the clients which, subsequently, are subject to verification by the smart contract. Clients construct these witnesses by interacting with the storage network. We stress that clients do not need to download the entire *memory* of accumulators to construct these witnesses. The data that needs to be transmitted from storage nodes to clients are hash paths from the appropriate accumulators’ hash trees, i.e., they are of logarithmic complexity. Thus, from hereon in, we assume that clients can efficiently construct the witness values that are required to realize the ERC20 token interface.

Accumulator-based ERC20 token smart contracts cannot implement the **balanceOf** and **allowance** functions since they do not store account balances and allowances in their state. Instead, clients are able to infer the information obtained by these functions by interacting with the storage network. To infer the balance y of account x, clients construct and verify a membership witness that the tuple (x, y) is accumulated in *balancesAcc*. To infer the allowance z of a spender’s account x_2 from an owner’s account x_1, clients construct and verify two membership witnesses. First, a membership witness that the tuple (x_1, x_2) is accumulated in *allowedAddressesAcc*, which proves that the token owner x_1 has allowed the spender account x_2 to transfer some tokens from her account. Second, a membership witness that the tuple (x_2, z) is accumulated in *allowedBalancesAcc*, which proves the number of tokens the spender is allowed to transfer from the token owner’s account.

Assume that an account x_1 wishes to approve z tokens to be transfered from her account by an account x_2. At the time an approval is registered, it is possible that it exceeds

the token owner's account balance. This is handled appropriately in the **transferFrom** function. The involved proofs for the **approve** operation vary based on whether this is the first time that x_1 approves x_2, or not. In the former case, x_1 constructs a non membership witness for the tuple (x_1, x_2) in *allowedAddressesAcc*, an update witness for the addition of the tuple (x_1, x_2) in *allowedAddressesAcc* and, lastly, an update witness for the addition of the tuple (x_2, z) in *allowedBalancesAcc*. In the case that x_1 has approved x_2 in the past for k tokens, the proofs are as follows. First, a membership witness for the tuple (x_1, x_2) in *allowedAddressesAcc*. Second, an update witness for the deletion of the tuple (x_2, k) from *allowedBalancesAcc*. Third, an update witness for the addition of the tuple (x_2, z) in *allowedBalancesAcc*. Note that both cases involve one invocation of Belongs, to verify the appropriate (non) membership witness, and two invocations of CheckUpdate, to verify the appropriate updates of the accumulators. Thus, both cases have the same computational complexity.

An account x_1 with balance y_1 that wishes to **transfer** z tokens ($y_1 \geq z$) to an account x_2 with balance y_2 produces the following proofs. First, a membership witness for the tuple (x_1, y_1) in *balancesAcc*, which proves the owner's account balance. Second, a membership witness for the tuple (x_2, y_2), which proves the balance of the destination account. Third, an update witness for the deletion of the tuple (x_1, y_1) from *balancesAcc*. Fourth, an update witness for the deletion of the tuple (x_2, y_2) from *balancesAcc*. Fifth, an update witness for the addition of the tuple $(x_1, y_1 - z)$ to *balancesAcc*. Sixth, an update witness for the addition of the tuple $(x_2, y_2 + z)$ to *balancesAcc*. Notice that the sequence of the involved updates reflects the transfer of z tokens from x_1 to x_2.

The **transferFrom** operation transfers k tokens from an account x_1 with balance y_1 that has approved the transfer of a total of $z \geq k$ tokens by an account x_2 with balance y_2. This operation involves the following proofs. First, a membership witness for the tuple (x_1, y_1) in *balancesAcc*. Second, a membership witness for the tuple (x_2, y_2) in *balancesAcc*. Third, a membership witness for the tuple (x_1, x_2) in *allowedAddressesAcc*. Fourth, a membership witness for the tuple (x_2, z) in *allowedBalancesAcc*. Fifth, an update witness for the deletion of the tuple (x_1, y_1) from *balancesAcc*. Sixth, an update witness for the deletion of the tuple (x_2, y_2) from *balancesAcc*. Seventh, an update witness for the addition of the tuple $(x_1, y_1 - k)$ to *balancesAcc*. Eighth, an update witness for the addition of the tuple $(x_2, y_2 + k)$ to *balancesAcc*. Ninth, an update witness for the deletion of the tuple (x_2, z) from *allowedBalancesAcc*. Tenth, if $z = k$, an update witness for the deletion of the tuple (x_1, x_2) from *allowedAddressesAcc*, otherwise, if $z > k$, an update witness for the addition of the tuple $(x_2, z - k)$ to *allowedBalancesAcc*. Again, both cases have the same computational complexity.

To summarize, the complexity of the accumulator-based ERC20 token smart contract's operations is as follows. The **approve** operation involves two update witnesses and either one non membership, or, one membership witness, depending on whether the token owner approves the spender's account for the first time or not, respectively. The **transfer** operation involves two membership witnesses and four update witnesses. The **transferFrom** operation involves four membership witnesses and six update witnesses. Thus, the **transferFrom** is the most expensive operation, followed by **transfer** and, lastly,

by **approve**.

8.2.2 Evaluation

In this section, we evaluate our accumulator-based ERC20 token construction. We ran our experiments on a private blockchain that is maintained by a single mining node. We use the latest, stable version of *geth* (v1.8.17, [20]), Ethereum's official client, that was available at the time of this writing. We conducted our experiments via the *truffle* suite (v4.1.13, [32]) that employs *solc-js* (v0.4.24, [27]) to compile smart contracts with optimizations enabled.

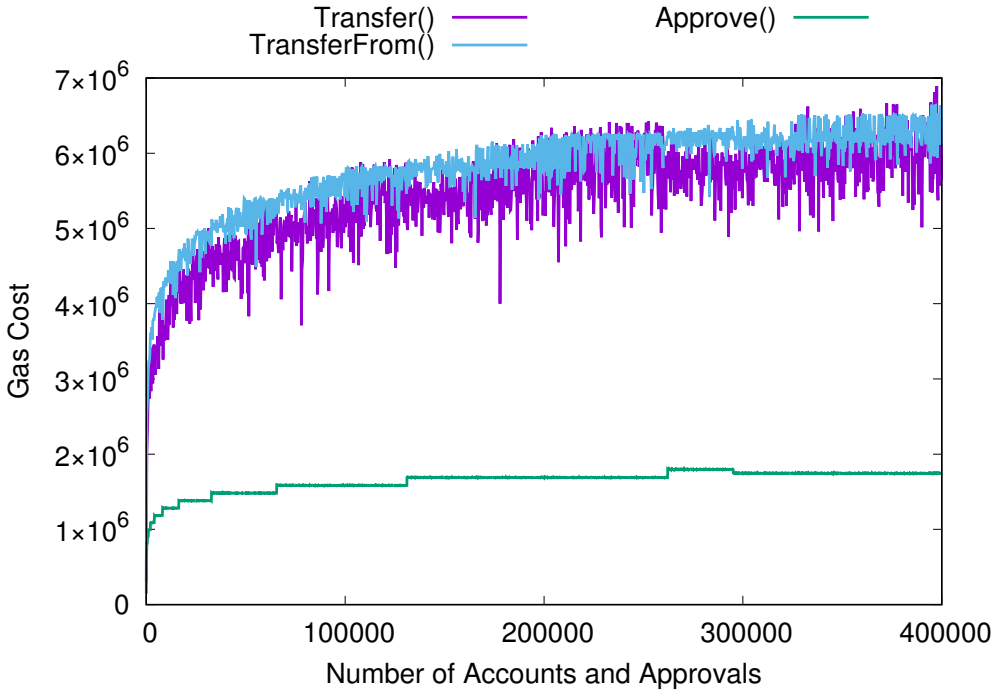

Figure 8.1: Gas cost versus of the transfer, approve and transferFrom operations of our accumulator-based ERC20 token construction for up to a total of 400,000 accounts and 400,000 approvals.

Figure 8.1 illustrates the gas cost of the **transfer**, **approve** and **transferFrom** operations of our accumulator-based ERC20 token for up to a total of 400,000 accounts and 400,000 approvals. Results illustrate that transaction gas costs scale logarithmically, which is expected (same holds for the **approve** operation). Recall that all involved proofs are hash path(s) starting from some node(s) (not necessarily leaf node(s)) in the accumulator's tree. Thus, their size and verification cost varies based on the position of those nodes in the tree. Our construction's operations consume a large portion of the block's limit which is, currently, about 8 million gas ([14]). In the following, we discuss a series of improvements that will diminish the cost of our construction's operations.

The security property of the accumulator of Camacho et al. [53] is based on the presup-

position that, prior to an invocation of CheckUpdate for the deletion or addition of some element x, $x \in X$ or $x \notin X$, respectively. Thus, prior to, e.g., verifying the addition of some element x via the CheckUpdate algorithm, we have to make sure, via a non membership witness verification, that $x \notin X$. Part of an ongoing project is to provide a proof extension that will allow us to lift this assumption from the accumulator's security property. Consequently, we will be able to eliminate one, two and four invocations of the accumulator's Belongs algorithm from the **approve**, **transfer** and **transferFrom** operations of the accumulator-based ERC20 token, respectively. Note that one invocation of Belongs costs, on average, 289,873.23 gas, when $|X| = 400,000$ and will, thus, provide a substantial improvement.

To illustrate the overhead of our accumulator-based ERC20 token construction, we implemented a "bare-bones" ERC20 token (where account balances and allowances are stored in the contract's state [17]) and repeated the same experiment. We measure an average cost of 33,193.12, 42,465.23 and 23,798.35 gas for the **transfer**, **approve** and **transferFrom** operations, respectively. Thus, our accumulator-based construction is much more expensive, despite its constant and minimal space overhead on Ethereum's state. The large discrepancy between the gas cost of the two constructions' operations, as well as, the small and static gas cost of the bare-bones ERC20 token operations, are a by-product of Ethereum's flat cost model. The fact that storage-related operations are underpriced ([52]) and that contracts do not pay a recurring fee proportional to the size of their state is one of Ethereum's main incentive misalignments. This issue, if left unchecked, will have severe consequences to the future of, not only Ethereum, but any smart contract platform that employs a flat cost model. Next, we propose modifications to Ethereum's cost model to deal with this issue.

8.3 Revisiting Ethereum's Storage Cost Model

8.3.1 Adaptive Pricing of Storage Operations

Ethereum employs a flat cost model to price all EVM opcodes ([107]), including storage-related operations. There are two main issues with this approach. First, storing data on the state of smart contracts incurs a *one time fee* which is underpriced ([52]). To our knowledge, there is no other, real world system that provides such high levels of data replication and availability without a recurring fee that is proportional to the volume of the stored data. Furthermore, as there is no incentive for freeing storage, Ethereum is faced with a tragedy of the commons problem with regards to the monotonically increasing size of its state. Second, Ethereum's flat cost model does not account for the complexity of executing storage-related operations, which is a function of the size of the state of smart contracts. We propose the following modifications to Ethereum's pricing of storage to address these issues.

Recurring Storage Fees: The concept of introducing "storage rent", i.e., a recurring fee that smart contracts have to pay based on the amount of storage they consume has been

discussed over the years. Buterin's original proposal ([5]) has spurred a lot of discussion and has led to the publication of several articles (e.g., [34, 36, 6]) which, in their vast majority, stress how important such a mechanism is for the longevity of public blockchains. An additional use of the rent mechanism is to clean up Ethereum's state from accounts (contracts are accounts as well) that are not being used anymore.

Our proposal on the subject of storage rent is based on the following points. First, we believe that rent fees should not be rewarded to anyone as that could introduce new attack vectors. Second, since Ethereum is a global computer, it is rational to assume that it has a predefined storage capacity S_{max} (e.g., Buterin has suggested 500 GB [35]). Naturally, this is a conceptual upper bound on the state's size and will, essentially, reflect an estimate of what is considered reasonable for the average miner. Third, S_{max} should be adjustable by the ones that maintain the network, i.e., the miners, to account for real world, storage trends. This could be achieved via a mechanism similar to the one that is already in use for adjusting block difficulty.

We propose that up to a low utilization percentage of the system's state, e.g., $U_{low} = 25\%$, the rent per storage key of a contract's state should be static to reflect the low burden imposed on miners. When the state's utilization is between U_{low} and, e.g., $U_{high} = 80\%$, the rent per storage key of a contract's state should increase logarithmically with the total number of keys in the system's state. This reflects the fact that Ethereum's state is organized on top of LevelDB whose complexity we elaborate more on the following paragraph. From thereon in, rent fees should be prohibitive, thus, they should scale linearly with the total number of keys in the system's state. To derive a base rent fee per storage key, we consider real world examples of systems that are highly replicated and available and that charge for storage. Cloud storage providers are a prime example. For instance, Amazon's EFS ([1]) charges 0.30 USD per GB per month. At the time of this writing, one unit of Ether corresponds to 202.18 USD ([16]). Based on this analogy, we compute a base rent fee of $R_{base} = 530,657,634.8$ Wei per storage key per year (1 Ether corresponds to 10^{18} Wei). Thus, we have an adaptable scheme for computing rent fees that follows the laws of supply and demand by considering the state's overall utilization and the burden imposed on miners.

Scaling Storage Costs: A contract's state is organized on an on-disk Merkle Patricia (MP) trie ([13]), which is referred to as the *storage trie*. This is a modified version of a typical radix tree with the added property of Merkle trees, i.e., the root hash uniquely identifies the (key,value) pairs in the tree. The nodes of the storage trie and the smart contract's state (storage keys) are stored in a LevelDB ([21]) key-value store, whose underlying data structure is a multi-level Log Structured Merge (LSM) tree. As illustrated in a recent study ([95]), due to Ethereum's authenticated storage (MP trie), one Ethereum read (e.g., reading the root node of a contract's storage trie) can lead to 64 LevelDB get() (read) requests. Each get() may internally involve multiple disk reads due to the large amount of metadata that LevelDB maintains ([109]). Updates to a contract's storage, e.g., adding/updating storage keys, result in updates to its storage trie that have to be committed on disk. In LevelDB, key-value updates are reinserted into a skip list with a monotonically increasing sequence number along with a "tombstone" flag that invalidates the pair's prior version.

To maintain key-value pairs in sorted order, LevelDB uses a compaction method. This process involves multiple merge sorts (one per LSM tree level) and incurs a write amplification factor, which is the ratio of the amount of data written to the amount of data requested for writing by users, of $\times 11$ ([109]).

Ethereum's flat cost model does not reflect the aforementioned complexity of storage-related operations. One might assume that an ideal scheme would scale the cost of these operations based on the number of incurred disk operations. However, this is not possible as Ethereum miners do not have a shared hardware configuration, e.g., their physical hard disks and their caches vary significantly. This would interfere with Ethereum's consensus as the execution of the same transaction would lead to different gas costs across different miners. Instead, we propose a scheme where the cost of storage-related operations is computed on a per transaction basis and scales according to the number of operations to LevelDB's LSM tree, which is the same across all miners. Fetching one key from a LSM tree involves two binary searches ([94]). Accessing the value of a smart contract's storage key involves, at minimum, fetching one node of its storage trie and, subsequently, fetching the storage key itself. Thus, it requires a total of four binary searches, i.e., $4\log_2(n)$ accesses, where n is the number of storage keys. Updating, or, adding a new storage key, involves the same number of accesses to infer the value of the tombstone flag. However, since updates are propagated to all levels of LevelDB's LSM tree during its compaction process, they are subject to LevelDB's write amplification factor, which we discussed above. Thus, updates incur a total of $11 \times 4\log_2(n) = 44\log_2(n)$ operations. Currently, reading, storing and updating storage keys costs 200, 20,000 and 5,000 gas, respectively. Thus, under our proposed scheme, the cost of reading, storing and updating a storage key can be computed via the following functions:

- $readKeyCost(n) = 800\log_2(n)$
- $updateKeyCost(n) = 220,000\log_2(n)$
- $storeKeyCost(n) = 880,000\log_2(n)$

In the following two subsections, we, first, provide details regarding the amount and type (read/write) of storage operations pertaining to the "bare-bones" ERC20 token and "linear state" PKI constructions, respectively. Next, we employ our adaptive scheme for pricing storage-related operations and illustrate the overhead of: 1) the "bare-bones" ERC20 token compared to our accumulator-based ERC20 token and, 2) the "linear state" PKI compared to our hash tree-based PKI.

8.3.2 Adaptive Pricing of Storage: Accumulator-based vs Bare Bones ERC20 Token

We provide a breakdown of the amount and type of storage operations, as well as, the storage cost pertaining to each function of the "bare-bones" ERC20 token, which is as follows:

- The **transfer**() function involves reading and updating the values of two storage keys, i.e., its storage cost can be computed by the function:

$$transferCost(n) = 2 \times readKeyCost(n) + 2 \times updateKeyCost(n) \tag{8.1}$$

- The **transferFrom**() function involves reading four storage keys and updating the values of three storage keys, i.e., its storage cost can be computed by the function:

$$transferFromCost(n) = 4 \times readKeyCost(n) + 3 \times updateKeyCost(n) \tag{8.2}$$

- The **approve**() function involves storing a new key in the contract's storage, i.e., its storage cost can be computed by the function:

$$approveCost(n) = storeKeyCost(n) \tag{8.3}$$

Our accumulator-based ERC20 token construction stores on the smart contract's state a total of four storage keys ($n = 4$) pertaining to the values of its three accumulators (balancesAcc, allowedAddressesAcc, allowedBalancesAcc) and their (common) security parameter (λ). In the following, we provide a breakdown of the amount and type of storage operations, as well as, the storage cost pertaining to each function of our accumulator-based ERC20 token construction:

- The **transfer**() function involves reading and updating the value of the balancesAcc accumulator, i.e., its storage cost is:

$$AccTransferCost = readKeyCost(4) + updateKeyCost(4) \tag{8.4}$$

- The **transferFrom**() function involves reading and updating the values of the balancesAcc, allowedAddressesAcc and allowedBalancesAcc accumulators, i.e., its storage cost is:

$$AccTransferFromCost = 3 \times AccTransferCost \tag{8.5}$$

- The **approve**() function involves reading and updating the values of the allowedAddressesAcc and allowedBalancesAcc accumulators, i.e., its storage cost is:

$$AccApproveCost = 2 \times AccTransferCost \tag{8.6}$$

Figures 8.2, 8.3 and 8.4 illustrate the gas cost of the **transfer**, **transferFrom** and **approve** operations, respectively, of the bare-bones and our accumulator-based ERC20 token under our proposed cost model. Regarding the bare-bones ERC20 token, we only plot the storage-related cost of its operations, which are the dominant factor. The biggest discrepancy is in the **approve** operation (Figure 8.4) where our accumulator-based construction provides an order of magnitude improvement.

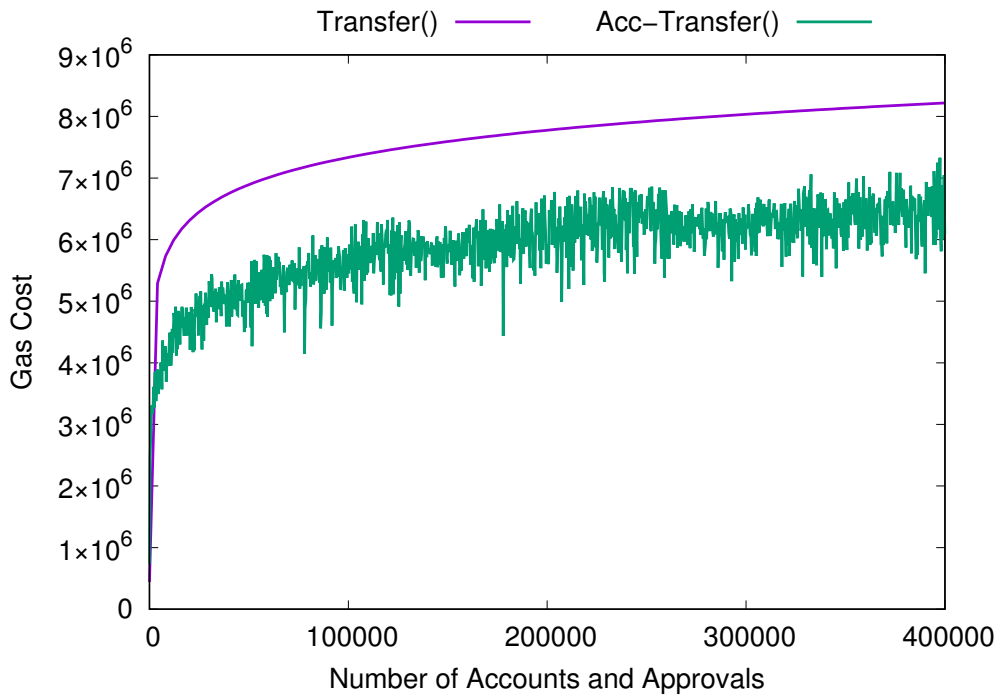

Figure 8.2: Gas cost of the transfer operation of the bare-bones and our accumulator-based ERC20 token for up to a total of 400,000 accounts and 400,000 approvals under our adaptive model for pricing storage operations.

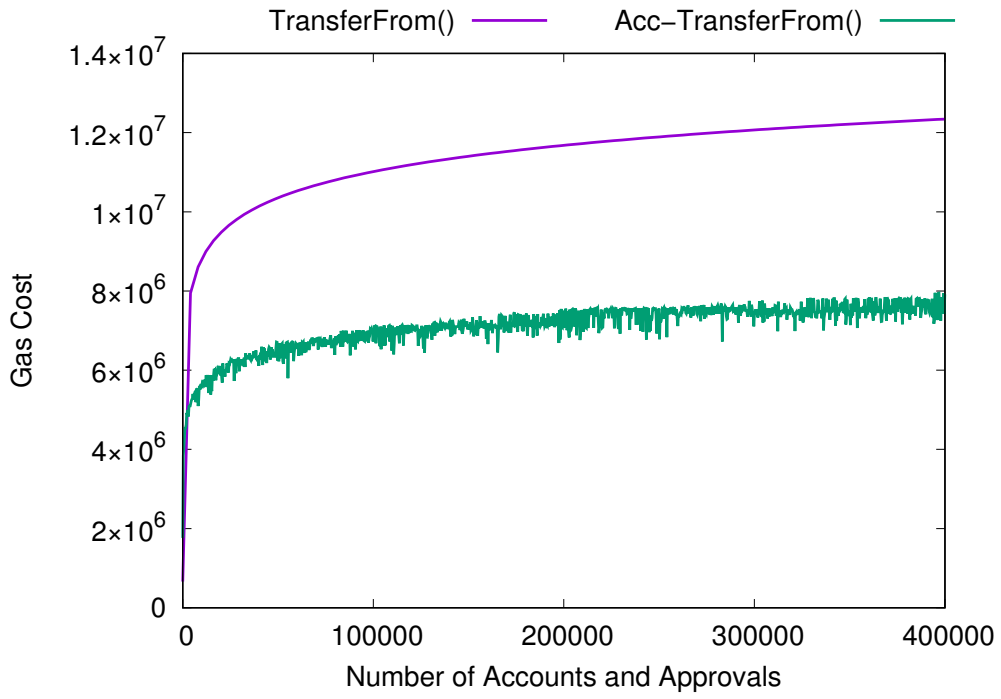

Figure 8.3: Gas cost of the transferFrom operation of the bare-bones and our accumulator-based ERC20 token for up to a total of 400,000 accounts and 400,000 approvals under our adaptive model for pricing storage operations.

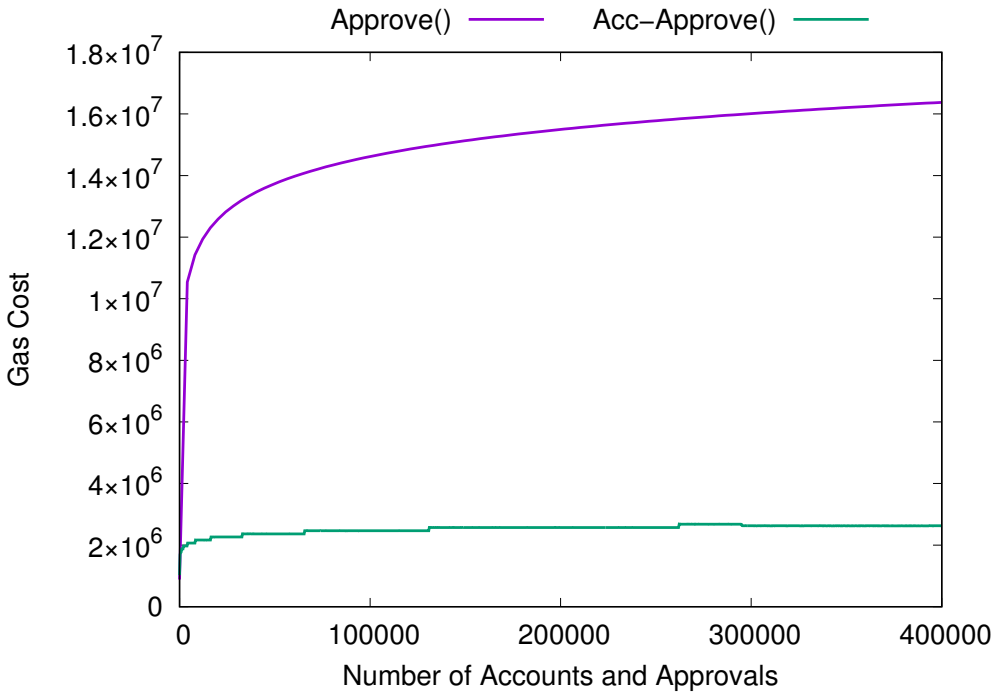

Figure 8.4: Gas cost versus of the approve operation of the bare-bones and our accumulator-based ERC20 token for up to a total of 400,000 accounts and 400,000 approvals under our adaptive model for pricing storage operations.

8.3.3 Adaptive Pricing of Storage: Hash Tree-Based vs Linear State PKI

Recall that we assume 32-byte identities and that ECDSA public keys are two elliptic curve coordinates, where each is 256 bits long. We provide a breakdown of the amount and type of storage operations, as well as, the storage cost pertaining to each function of the "linear state" PKI:

- The Register() function involves reading one EVM word from the contract's storage, to infer if an (id, pk) pair is registered or not, and, the storage of three new keys (assuming that the identity is not taken). Thus, the storage cost of this operation can be computed by the following formula:

$$LinearStatePKIRegisterCost(n) = readKeyCost(n) + 3 \times storeKeyCost(n) \quad (8.7)$$

- The Revoke() function involves reading three EVM words from the contract's storage, i.e., the (id, pk) pair to infer if it is registered or not, and updating the value of one storage key. Thus, the storage cost of this operation can be computed by the following formula:

$$LinearStatePKIRevokeCost(n) = 3 \times readKeyCost(n) + updateKeyCost(n) \quad (8.8)$$

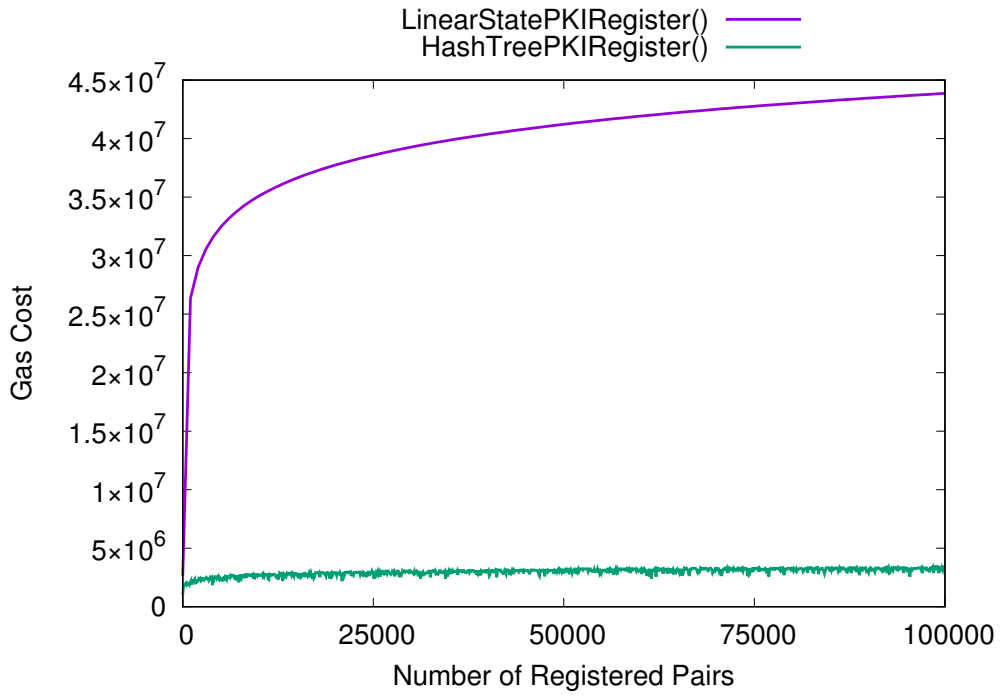

Figure 8.5: Gas cost of registering 100,000 (id, pk) pairs on the linear state versus our hash tree-based PKI under our proposed model for pricing storage operations.

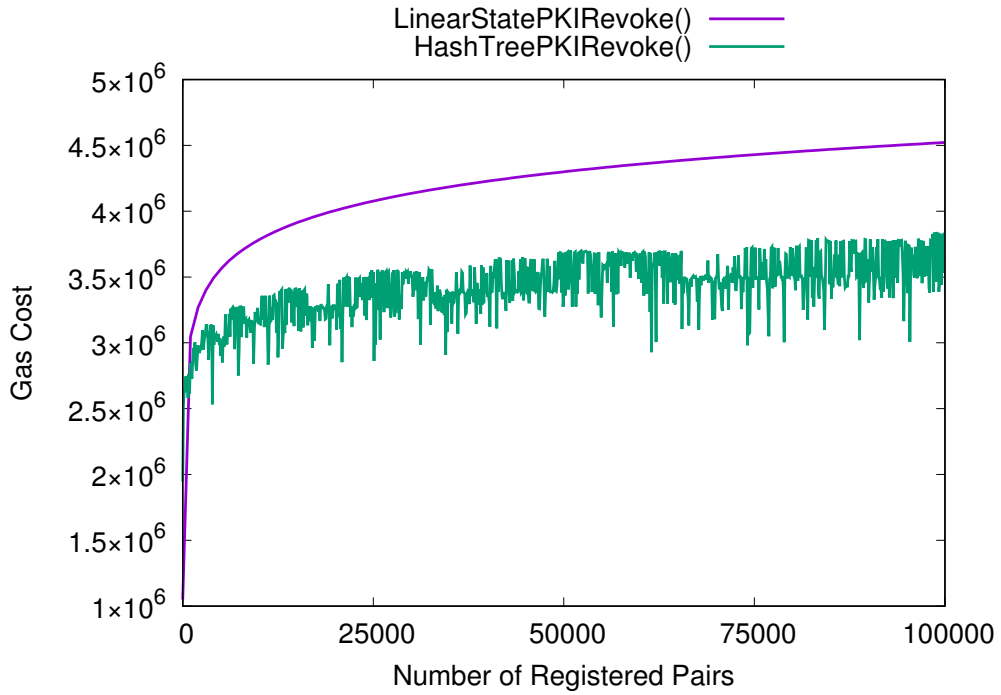

Figure 8.6: Gas cost of revoking 100,000 (id, pk) pairs on the linear state versus our hash tree-based PKI under our proposed model for pricing storage operations.

Our hash tree-based PKI stores on the smart contract's state a total of four storage keys ($n = 4$) pertaining to the values of its two accumulators (c_1, c_2) and their respective security parameters (λ_1, λ_2). The Register() and Revoke() functions involve reading and updating the values of both of its accumulators. Thus, their storage costs can be computed by the following formula:

$$HashTreePKIRegisterStorageCost = HashTreePKIRevokeStorageCost = 2 \times readKeyCost(4) + 2 \times updateKeyCost(4) \tag{8.9}$$

Figures 8.5 and 8.6 illustrate the gas cost of the Register() and Revoke() functions, respectively, of the "linear state" PKI, compared to that of our hash tree-based PKI, under our proposed adaptive cost model. Regarding both PKI operations, our hash tree-based PKI delivers reduced costs. The biggest improvement is in the Register operation, where our hash tree-based PKI provides an order of magnitude improvement.

Overall, results illustrate that, under a pricing scheme that reflects the effort that miners have to expend to execute storage-related operations, the programming paradigm that we propose in this work, as illustrated by our case studies, provides reduced gas costs across all operations. Nevertheless, we believe that the most important property of our approach is that it aligns well with the future of smart contract platforms since it incurs constant storage overhead to miners.

9. CONCLUSIONS AND FUTURE WORK

Naming services provide the necessary foundations to develop important and pervasive applications, such as e-commerce and e-banking. The volume of monetary value that is exchanged in today's online (micro) transactions, coupled with the fact that the underlying networking infrastructure (e.g., the Internet) is, by design, not reliable, necessitates naming services that are secure and fault-tolerant by design. However, it stands as a grand contradiction that these fundamental systems are, to this day, under the control of centralized, trusted, remote parties that have been involved in multiple, prominent security incidents throughout the years. These have had severe consequences on both end-users, as well as, large corporations. Undeniably, centralization provides convenience in terms of systems implementation, as well as, control. Nonetheless, throughout human history, it has been proven, time and time again, that concentration of power and trust has always had severe repercussions. To address the dangers stemming from centralization, it is our firm belief, and the driving force behind this work, that we need to shift to more democratic, or distributed, structures.

This thesis is dedicated to the design, implementation and evaluation of a provably secure, decentralized naming service that, contrary to the norm, has no single point of failure or control and whose security is based on standard cryptographic hardness assumptions. We are the first to provide a complete formalization of the naming service design problem in the Universal Composability framework and we formally prove the security of our construction under the strong RSA assumption in the Random Oracle model. Thus, we have covered one of the literature's main gaps, as no prior work provided a provably secure construction which, we believe, is a necessity for such critical infrastructures.

Our design is built on top of smart contract platforms, a modern, ingenious invention that allows the development of smart contracts, i.e., stateful, programmable agents that can encode arbitrary state transition functions and are executed over a distributed network of peers (e.g., blockchains). This layer provides our design a decentralized, secure and fault tolerant infrastructure that, in addition, encompasses a mechanism that incentivizes its perpetuity. As a result, it counters the free-riding problem and, simultaneously, is resilient to Sybil attacks, while still retaining a scalable, free-entry system. This provides a combination of features that is vital for the deployment of publicly-available naming services, which was not provided by the majority of prior works.

We resolve the issues stemming from on-blockchain, or, on-smart contract storage, by decoupling the issue of data storage from its validity. The former is handled by an external storage network that provides for efficient access to the application's data, compared to downloading and validating the entire blockchain. The adversary is allowed to tamper with its contents in an arbitrary manner which, as we prove in Appendix B, does not affect our design's security. Regarding the storage network, it is important to clarify a few additional points. We stress that in cases where the storage network becomes corrupted, it is possible to establish a clean instance of it. Indeed, the history of operations is still stored on the blockchain, which the adversary cannot corrupt. This procedure, although

costly as one might argue, provides a base level of liveness to our design. However, we believe that a real world deployment mandates greater liveness guarantees, as well as, a few subtle efficiency improvements, which we discuss below. Consider the case where a client wants to register a new (id, pk) pair. As we have already illustrated in the description of our protocols, the client requests the history of operations from the storage network to construct the appropriate witness values. The client can infer the validity of the state she received only after she has computed and verified the witnesses. Consequently, a malicious node of the storage network can impose polynomial computational overhead to the client. In such cases, the client will then need to query an alternative node of the storage network and repeat the same process again. Thus, the protocol between clients and storage network nodes needs to be enhanced in order to limit the computational overhead that malicious nodes can impose on clients. Furthermore, it would be desirable to diminish the amount of data that clients pull from the storage network to construct witness values. This is a challenging and independent research topic that involves exploring and expanding the properties of the underlying constructs that provide data validity in our design, which we discuss below.

To provide for data validity in the face of an unreliable storage network, our design stores, on the smart contract's state, cryptographic accumulators. These are data structures that provide a constant representation of an (updatable) set of elements and allow for (non) membership witnesses. More specifically, we are the first to define the notion of a public-state, universal accumulator, a cryptographic tool of independent interest for applications that employ blockchains, or any type of (public) bulletin board, to verify the validity of their data. We provide an RSA-based construction of such an accumulator, which favors storage overhead at the expense of computational complexity to provide for constant-sized smart contract state and witnesses. To explore this tradeoff, we propose a second construction that is built on top of the hash tree, universal accumulator of Camacho et al. [53], where witnesses are logarithmic in the number of registered identity records. We evaluate both of our constructions, as well as, their building blocks, by implementing PKI variants and by comparing them with the approach of most prior schemes, i.e., on-blockchain storage.

Our evaluation illustrates that our RSA-based PKI construction is not viable for deployment on Ethereum's live chain. This is attributed to the overwhelming cost of the Map procedure. The viability of this construction, however, might be improved when the EVM is extended to support bitwise shifting operations. We leave as future work exploring alternatives, such as verifiable computation, that will diminish the computational complexity of the smart contract and, thus, the cost of its operations. In contrast, our hash tree-based PKI construction can be deployed on Ethereum's live chain and can accommodate small, to moderately sized PKIs. We stress that the deployment scale of this construction might be improved in the future where Ethereum will shift from PoW to PoS consensus. This construction can be further optimized by extending the security property of the accumulator of Camacho et al. [53] which, in short, will allow update witnesses to act as (non) membership witnesses as well. Nevertheless, via our evaluation, we show that on-blockchain storage is the best alternative, in terms of transaction costs. However, we stress that this

approach will have a direct impact on Ethereum's security and, ultimately, its longevity, as its monotonically increasing state will pose as a counterincentive for new nodes to sync and contribute to the network.

We address several issues that stem from Ethereum's pricing of storage related operations. To this end, we propose an adaptive scheme that prices storage-related operations based on the effort that miners have to expend to execute them, introduces recurring storage fees and frees space consumed by unused/stale contracts. We show that under such a pricing scheme, our hash tree-based PKI construction reduces the incurred transaction fees by up to an order of magnitude, compared to the approach of on-blockchain storage.

To conclude, the NS design and constructions presented in this work adhere to the original, distributed design principles of the Internet. Our work has shown that we can build such critical services without blindly trusting centralized entities and suffering the consequences of their security breaches. We believe that our work illustrates the potential that modern, ingenious constructs have and provides the first steps in building a democratic and decentralized digital world that is secure and verifiable by cryptography.

ABBREVIATIONS - ACRONYMS

SSL	Secure Sockets Layer
TLS	Transport Layer Security
WWW	World Wide Web
LDAP	Lightweight Directory Access Protocol
WoT	Web Of Trust
PGP	Pretty Good Privacy
P2P	Peer-to-Peer
PoW	Proof-of-Work
PoS	Proof-of-Stake
VM	Virtual Machine
ENS	Ethereum Name Service
BNS	Blockstack Name Service
EVM	Ethereum Virtual Machine
UTXO	Unspent Transaction Output
EOA	Externally Owned Account
BFT	Byzantine Fault Tolerance
IP	Internet Protocol
CA	Certification Authority
RA	Registration Authority
CRL	Certificate Revocation List
TTP	Trusted Third Party
UC	Universal Composability
NS	Naming Service
DNS	Domain Name System
DDNS	Distributed Domain Name System
PKI	Public Key Infrastructure

CPKI	Centralized Public Key Infrastructure
DPKI	Decentralized Public Key Infrastructure
NIZK	Non Interactive Zero Knowledge
UDB	Unreliable Database
LSM	Log Structured Merge
MP	Merkle Patricia
dApp	decentralized application
sk	secret key
pk	public key
vk	verification key
ek	encryption key
dk	decryption key
id	identity

APPENDIX A. PROOF OF LEMMA 5.1.1

Lemma 5.1.1 ([68]). *Let U be a 2-universal hash function family from $\{0,1\}^{3k}$ to $\{0,1\}^k$. Then, for all but a 2^{-k} fraction of functions $f \in U$, for any $y \in \{0,1\}^k$, a fraction of at least $1/ck$ elements in $f^{-1}(y)$ are primes where c is some small constant.*

Lemma 5.1.1 is proven using a sequence of lemmas. We follow the proof given in the appendix of the paper [102].

Lemma A.0.1. *Let $U = \{f : \{0,1\}^m \to \{0,1\}^k\}$ be a 2-universal hash function family. For any $A \subseteq \{0,1\}^m$, for all but a $O(\frac{2^{2k}}{|A|})$-fraction of $f \in U$, it holds that for any $z \in \{0,1\}^k$, $\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{|A|}{2^{m+1}}$.*

Lemma A.0.1 is proven by first showing Lemma A.0.2 and then Lemma A.0.3.

Lemma A.0.2. *Let $A \subseteq \{0,1\}^m$ and $z \in \{0,1\}^k$. We say that $f \in U$ is (A,z)-balanced if $\frac{2}{3} \cdot \frac{|A|}{2^k} \leq |f^{-1}(z) \cap A| \leq \frac{4}{3} \cdot \frac{|A|}{2^k}$. It holds that* $\mathsf{Pr}_{f \in U}[f$ ***is not*** *(A,z)-balanced$] \leq \frac{9 \cdot 2^k}{|A|}$.*

Proof. Assume that $A = \{a_1, \ldots, a_\ell\}$ and that we choose $f \in U$ uniformly at random. Since U is a 2-universal hash function family (Definition 4.1.3) we have that $\mathsf{Pr}[f(a_i) = z] = \frac{1}{2^k}$.

We define the random variable X_i, which equals 1 if $f(a_i) = z$ and 0 otherwise. Therefore, it holds that $\mathsf{Pr}[X_i = 1] = \frac{1}{2^k}$. If $X = \Sigma_{i=1}^{\ell} X_i$, then it can be easily observed that $X = |f^{-1}(z) \cap A|$. We also have that $\mu = E[X] = \ell \cdot \mathsf{Pr}[f(a_i) = z] = \ell \cdot 2^{-k}$. We will now utilize the Chebychev inequality

$$\mathsf{Pr}[|X - \mu| \geq t] \leq \frac{Var(X)}{t^2}.$$

If we set $t = \mu/3$, we have that

$$\mathsf{Pr}_{f \in U}[|X - \mu| \geq \frac{\mu}{3}] \leq \frac{9Var(X)}{\mu^2}. \tag{A.1}$$

Since $X_i, \ldots, X_\ell$ are pairwise independent, we have that $Var(X) = \Sigma_{i=1}^{\ell} Var(X_i)$. Therefore, $Var(X) = \Sigma_{i=1}^{\ell}\big(E[X_i^2] - E[X_i]^2\big) = \ell \frac{1}{2^k}\Big(1 - \frac{1}{2^k}\Big) \leq \ell 2^{-k}$. Hence, $\frac{9Var(X)}{\mu^2} \leq \frac{9 \cdot 2^k}{\ell}$. Therefore, by (A.1), it holds that

$$\frac{2}{3} \cdot \frac{|A|}{2^k} < |f^{-1}(z) \cap A| < \frac{4}{3} \cdot \frac{|A|}{2^k}, \tag{A.2}$$

except for $\frac{9 \cdot 2^k}{\ell}$ fraction of functions $f \in U$. □

Lemma A.0.3. *Let $A \subseteq \{0,1\}^m, z \in \{0,1\}^k$ and $f \in U$. We say that the pair (f,z) is "bad" for the set A if $\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} \leq \frac{|A|}{2^{m+1}}$. Then, for any $A \subseteq \{0,1\}^m, z \in \{0,1\}^k$,*

$$\mathsf{Pr}_{f \in U}[(f,z) \textit{ is "bad" for } A] \leq \frac{18 \cdot 2^k}{|A|}. \tag{A.3}$$

Proof. We fix $z \in \{0,1\}^k$. Then, by Lemma A.0.2, if we set $A = \{0,1\}^m$, we have that

$$\frac{2}{3} \cdot 2^{m-k} < |f^{-1}(z)| < \frac{4}{3} \cdot 2^{m-k}, \tag{A.4}$$

except for $9 \cdot 2^{k-m}$ fraction of functions $f \in U$. Next, by Lemma A.0.2 for an arbitrary $A \subseteq \{0,1\}^m$, we have that

$$\frac{2}{3}|A| \cdot 2^{-k} < |f^{-1}(z) \cap A| < \frac{4}{3}|A| \cdot 2^{-k}, \tag{A.5}$$

except for $\frac{9 \cdot 2^k}{|A|}$ fraction of functions $f \in U$.

Combining (A.4), (A.5), it holds that

$$\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{|A|}{2^{m+1}}, \tag{A.6}$$

except for $\frac{18 \cdot 2^k}{|A|}$ fraction of functions $f \in U$. □

By Lemma A.0.3, using the union bound, we can get Lemma A.0.1.

Proof of Lemma 5.1.1. Now, let A be the set of prime numbers less than 2^m and $m = 3k$, as Lemma 5.1.1 considers a 2-universal hash function family of functions from $\{0,1\}^{3k}$ to $\{0,1\}^k$. By the prime number theorem, we have that the number of primes which are less or equal to x, denoted as $\pi(x)$, is asymptotically $\frac{x}{\ln x}$. Making use of a non-asymptotic bound ([97]), we have that for any $x \geq 55$, $\pi(x) > \frac{x}{\ln x + 2}$. Therefore, we have that

$$|A| = \pi(2^{3k}) > \frac{2^{3k}}{\ln 2^{3k} + 2}. \tag{A.7}$$

By (A.7) and Lemma A.0.1, it holds that except for $(\frac{18(3k+2\log_2 e)}{2^k \log_2 e})$-fraction of functions $f \in U$,

$$\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{1}{\frac{6k}{\log_2 e} + 2} \approx \frac{1}{4.16k + 2}. \tag{A.8}$$

□

APPENDIX B. PROOF OF THEOREM 5.2.1

The work presented in this Appendix B was partially published in our joint work ([92]) with Katerina Samari, who is credited for drafting the full proof.

Theorem 5.2.1. *Protocol π_{RSA} of Figure 5.3 securely realizes the functionality $\mathcal{F}_{ns}$ in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world under the Strong-RSA assumption in the Random Oracle Model. Namely, for any p.p.t. adversary $\mathcal{A}$ interacting with protocol π_{RSA} in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world, there is a p.p.t. simulator $\mathcal{S}$ interacting with the functionality $\mathcal{F}_{ns}$, such that, for any p.p.t. environment $\mathcal{Z}$, it holds that:*

$$EXEC^{\mathcal{F}_{ns}}_{\mathcal{Z},\mathcal{S}} \overset{c}{\approx} EXEC^{\pi^{\mathcal{F}_{TP},\mathcal{F}_{UDB}}_{RSA}}_{\mathcal{Z},\mathcal{A}}.$$

Proof. We construct a simulator $\mathcal{S}$ (Figure B.1), which emulates an execution of protocol π_{RSA} in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world, in the presence of an adversary $\mathcal{A}$. $\mathcal{S}$ plays the role of T, $\mathcal{F}_{TP}$, $\mathcal{F}_{UDB}$, the role of the servers and acts on behalf of a number of honest clients in the simulation of the hybrid-world protocol π_{RSA}. Based on the construction of our simulator $\mathcal{S}$, we show that an environment can distinguish between the executions in the hybrid and the ideal world *only* by influencing the way the membership or non membership verifications take place in the hybrid world protocol. In other words, the only inconsistency between the two executions can be derived if an adversary manages to convince the functionality $\mathcal{F}_{TP}$ about false statements (whether an element belongs to an accumulated set or not), thus, breaking the security of at least one of the accumulators of protocol π_{RSA}. Note that relation $R(pk, \text{aux})$ does not provide an opportunity for distinguishing since it is the same in both worlds.

We assume that $\mathcal{Z}$ is a p.p.t. environment. For any message sent by $\mathcal{Z}$ to a party (e.g., a client C or party T), we examine the output in both the hybrid and the ideal world. In the UC model, the parties in the ideal world are dummy, i.e., they simply forward any message they receive from the environment to a functionality and vice versa. Below, we show that for any message sent by the environment, the outputs of the parties in the hybrid and ideal world are indistinguishable and, thus, the environment cannot distinguish between the executions in the hybrid and ideal world.

$\mathcal{Z}$ sends (sid, Init) to server S_i: In the hybrid world, server S_i sends (sid, InitTP) to $\mathcal{F}_{TP}$ and $(sid, \text{InitUDB})$ to $\mathcal{F}_{UDB}$. The functionalities $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ add the server S_i to the set S_{init} and S'_{init}, respectively, and inform the adversary $\mathcal{A}$ that S_i is initialized. They both return success to S_i which, in turn, returns success. In the ideal world, the simulator $\mathcal{S}$, upon receiving (sid, Init, S_i) from $\mathcal{F}_{ns}$, according to Step 1 of Figure B.1, sends allow to $\mathcal{F}_{ns}$, which returns success to S_i.

$\mathcal{Z}$ sends (sid, Setup, R) to party T: In the hybrid world, T sends $(sid, \text{Install}, P_{RSA})$ to $\mathcal{F}_{TP}$. If flag $= ready$ and $\mathcal{A}$ returns allow, then $\mathcal{F}_{TP}$ stores P_{RSA}, sets $state \leftarrow \varepsilon$ and returns success to T. Then, T runs KeyGen(1^λ) twice (Step 2, Figure 5.3), sets $params = (pk_1, pk_2, R)$ and sends $(sid, (\text{Setup}, params))$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow, $\mathcal{F}_{TP}$ runs P_{RSA} on input $(\text{Setup}, params)$, returns $state \leftarrow (c_{0,1}, c_{0,2}, params)$ to T and T returns success

Simulator $\mathcal{S}$:

1. Upon receiving (sid, Init, S_i) by $\mathcal{F}_{ns}$, $\mathcal{S}$, on behalf of $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, sends $(sid, \text{InitTP}, S_i)$ and $(sid, \text{InitUDB}, S_i)$ to $\mathcal{A}$ and allow to $\mathcal{F}_{ns}$.
2. Upon receiving (sid, Setup, R) by $\mathcal{F}_{ns}$, $\mathcal{S}$, acting as T in the real-world protocol, runs KeyGen(1^λ) twice. Then, $\mathcal{S}$, by acting as $\mathcal{F}_{TP}$ in the real-world protocol, sends $(sid, \text{Install}, P_{RSA})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ sends $(sid, \text{Setup}, params)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs P_{RSA} on input $(\text{Setup}, params)$ and sends allow to $\mathcal{F}_{ns}$.
3. Upon receiving $(sid, \text{Register}, id, pk)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, acting as an honest client C and $\mathcal{F}_{UDB}$ in the hybrid-world protocol π_{RSA}, sends $(sid, \text{RetrieveDB}, C)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Step 3a. If the last record for id is a Register record, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$, otherwise, it moves to Step 3b. $\mathcal{S}$, acting as $\mathcal{F}_{TP}$, sends $(sid, \text{Register}, id, pk, i, W_{2,1}, W_{2,2})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs P_{RSA} on input $(\text{Register}, id, pk, i, W_{2,1}, W_{2,2})$. If P_{RSA} outputs fail, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$, otherwise, $\mathcal{S}$, acting as C, runs $W_{1,2} \leftarrow \text{NonMemWitGen}(pk_1, X_1, c'_1, (id, pk, i, d))$. $\mathcal{S}$, acting as $\mathcal{F}_{UDB}$, sends $\big(sid, \text{Post}, (\text{Register}(id, pk, i, W_{1,1}, W_{1,2})\big)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ sends allow to $\mathcal{F}_{ns}$ and updates $DBstate$ by storing $(\text{Register}, id, pk, i, W_{1,1}, W_{1,2})$.
4. Upon receiving $(sid, \text{Revoke}, id, pk, \text{aux})$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, acting as an honest client C and $\mathcal{F}_{UDB}$, sends $(sid, \text{RetrieveDB}, C)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Steps 4a, 4b and computes the updated witnesses $W_{1,1}, W_{1,2}$. $\mathcal{S}$, acting as $\mathcal{F}_{TP}$, sends $(sid, \text{Revoke}, id, pk, i, W_{1,1}, W_{1,2}, \text{aux})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs P_{RSA} on input $(\text{Revoke}, id, pk, i, W_{1,1}, W_{1,2}, \text{aux})$ and, if it outputs $(\text{fail}, state)$, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$. Otherwise, $\mathcal{S}$ returns allow to $\mathcal{F}_{ns}$ and updates $DBstate$ by storing $(\text{Revoke}, id, pk, i)$.
5. Upon receiving $(sid, \text{Retrieve}, id)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, on behalf of an honest client C and playing the role of $\mathcal{F}_{UDB}$, sends $(sid, \text{RetrieveDB})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Step 5a of Figure 5.3. If Step 5a returns fail, then, $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, otherwise $\mathcal{S}$ runs Step 5b and simulating $\mathcal{F}_{TP}$, sends $(sid, \text{RetrieveState})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow and all the algorithms at Step 5b return 1, then, $\mathcal{S}$ sends allow to $\mathcal{F}_{ns}$, otherwise it sends fail.
6. Upon receiving $(sid, \text{VerifyID}, id)$ or $(sid, \text{VerifyMapping}, id, pk)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$ runs the simulation as in Step 5.
7. Upon receiving $(sid, \text{ChangeDBstate}, DBstate')$ by $\mathcal{A}$, $\mathcal{S}$ sets $DBstate \leftarrow DBstate'$.
8. Upon receiving $(sid, \text{Register}, id, pk)$ or $(sid, \text{Revoke}, id, pk)$ by $\mathcal{F}_{ns}$ for a ***corrupted*** client C, $\mathcal{S}$ waits for the actions of $\mathcal{A}$ and simulates $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ as in previous cases. If $\mathcal{S}$ receives $(\text{Register}, id', pk', i, W'_{2,1}, W'_{2,2})$ by $\mathcal{A}$, $\mathcal{S}$ checks if $id \neq id'$ or/and $pk \neq pk'$. If the program P_{RSA} does not return $(\text{fail}, state)$ on this input, $\mathcal{S}$ sends $(\text{Register}, id', pk', C)$ to $\mathcal{F}_{ns}$. $\mathcal{S}$ runs similarly when it receives $(\text{Revoke}, id', pk', i, W'_{1,1}, W'_{1,2})$.

Figure B.1: Simulator $\mathcal{S}$.

to $\mathcal{Z}$. In the ideal world, if flag $=$ start, the functionality $\mathcal{F}_{ns}$ sends (sid, Setup, R) to $\mathcal{S}$. $\mathcal{S}$ follows Step 2 of Figure B.1 and simulates T and $\mathcal{F}_{TP}$ in the execution of protocol π_{RSA} in the presence of $\mathcal{A}$. Given that $\mathcal{A}$ returns allow to $\mathcal{S}$ when the latter acts as $\mathcal{F}_{TP}$, $\mathcal{S}$ returns allow to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ returns success to T. Therefore, T returns success both in the hybrid and ideal world.

Below, when we review the cases where $\mathcal{Z}$ sends a Register, Revoke or Retrieve message to a client C, we will distinguish on whether the adversary $\mathcal{A}$ has changed the contents of $DBstate$, by sending a ChangeDBstate message to $\mathcal{F}_{UDB}$, or not. In all cases, we will examine the output of an *honest* and a *corrupted* client (i.e. a client who is controlled by the adversary $\mathcal{A}$) and prove that consistent outputs are returned both in the hybrid and ideal world.

To simplify the description, in the remainder of the proof we will, first, assume that $\mathcal{A}$ returns allow in any interaction with $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$. If $\mathcal{A}$ does not return allow, C returns fail in both the hybrid and ideal world. Second, as illustrated in protocol π_{RSA} (Figure 5.3), both in the cases of a Register or a Revoke message, an *honest* client, first retrieves $DBstate$ by sending a request to $\mathcal{F}_{UDB}$. In the case of a Register message, the client checks if the last record in $DBstate$ is a Revoke record or if there is no record related to the identity (Step 3a). In the case of a Revoke message, a client will check whether there is a register record in $DBstate$ for the corresponding (id, pk) pair (Step 4a). In both cases, if the above checks fail, the client will return fail in the hybrid world. In the ideal world, it is easy to see that fail is also returned. In Steps 3, 4 of Figure B.1, $\mathcal{S}$, simulating an honest client and $\mathcal{F}_{UDB}$ will return in all the aforementioned cases fail to $\mathcal{F}_{ns}$. To avoid repetition, we will not refer to such cases when they arise in our analysis.

$\mathcal{Z}$ **sends a message** $(\textbf{sid}, \textbf{Register}, \textbf{id}, \textbf{pk})$ **to a client C:** Consider the following cases:

Reg1: $\mathcal{A}$ has **not** sent $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \text{Register}, id, pk)$ is sent to client C. Below, we examine two subcases related to whether the identity id is registered or not:

(a) **The identity id is not registered:** In the hybrid world, an *honest* client C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. C, on receipt of $DBstate$, computes witnesses $W_{2,1}, W_{2,2}$, by following the procedure described in Steps 3a, 3b of Figure 5.3. Then, C sends $(sid, (\text{Register}, id, pk, i, W_{2,1}, W_{2,2}))$ to $\mathcal{F}_{TP}$, $\mathcal{F}_{TP}$ returns $((c'_1, W'_{1,1}), (c'_2, W'_{2,1}), state)$, C sets $W_{1,1} \leftarrow W'_{1,1}$ and computes the non membership witness $W_{1,2}$ for (id, pk, i, d). Then, C sends $(sid, \text{Post}, (\text{Register}, id, pk, i, W_{1,1}, W_{1,2}))$ to $\mathcal{F}_{UDB}$ and returns success in the hybrid world. In the ideal world, C returns success as well, since the simulator $\mathcal{S}$, acting as C, $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, returns allow to $\mathcal{F}_{ns}$ (Step 3, Figure B.1). Finally, $\mathcal{F}_{ns}$, verifying that id is not registered, sends success to C.

(b) **The identity id is currently registered:** In the hybrid world, a *corrupted* client C may try to convince $\mathcal{F}_{TP}$ that id is **not** registered. Then, C must either provide a valid non membership witness $W_{2,1}$ for (id, j, a) in c_2, for some $j \geq 2$, such that VerifyNonMem$(pk_2, (id, j, a), W_{2,1}, c_2) = 1$ and a valid membership witness $W_{2,2}$ for $(id, j-1, d)$ in c_2, such that VerifyMem$(pk_2, (id, j-1, d), W_{2,2}, c_2) = 1$, or, a valid non membership witness $W_{2,1}$ for $(id, 1, a)$ in c_2. We show that, by the security of accumulator c_2, such an attack takes place only with negligible probability. Recall that since id

is currently registered, it holds either that (1) there is $\ell \geq 2$ such that $(id, \ell, a) \in X_2$ and $(id, \ell, d) \notin X_2$, where X_2 is set accumulated of c_2, or, (2) $(id, 1, a) \in X_2$. Starting with (1), we consider the cases where $1 < j \leq \ell$, and $j > \ell$. If $1 < j \leq \ell$, then $(id, j, a) \in X_2$ and $(id, j-1, d) \in X_2$. By the security of accumulator c_2, C can produce a valid non membership witness $W_{2,1}$ for (id, j, a) only with negligible probability. If $j > \ell$, then $(id, j, a) \notin X_2$ and $(id, j-1, d) \notin X_2$. By the security of the accumulator c_2, C cannot produce a valid membership witness $W_{2,2}$ for $(id, j-1, d)$. For case (2), similarly, C can produce a valid non membership witness $W_{2,1}$ for $(id, 1, a)$ only with negligible probability. Hence, if a corrupted client C sends $(sid, \text{Register}, id, pk, j, W_{2,1}, W_{2,2})$ to $\mathcal{F}_{TP}$, $\mathcal{F}_{TP}$ will return $(\text{fail}, state)$ and C will return fail in the hybrid world. Correspondingly, in the ideal world, C would also return fail. The simulator $\mathcal{S}$, according to Step 8, waits for the actions of $\mathcal{A}$, i.e., the corrupted client C. Then, $\mathcal{S}$ simulates $\mathcal{F}_{TP}$ in the eyes of the corrupted client C and since $\mathcal{F}_{TP}$ returns $(\text{fail}, state)$, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ sends fail to C, who returns fail as well.

Reg2: $\mathcal{A}$ has sent $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \text{Register}, id, pk)$ is sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$ and the set X_2' derived by $DBstate'$ is different from the set X_2 accumulated in c_2[1]. We consider the following subcases:

(a) **The identity id is not registered, but the last record for id in DBstate′ is of the form** $(\textbf{Register}, \textbf{id}, \textbf{pk}, \textbf{j}, \textbf{W}_{1,1}, \textbf{W}_{1,2})$**:** In the hybrid world, a *corrupted* client C may send $(\text{Register}, id, pk', j', W_{2,1}, W_{2,2})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $(\text{fail}, state)$, then C will return fail. In the ideal world, $\mathcal{S}$, simulating $\mathcal{F}_{TP}$, returns fail to $\mathcal{F}_{ns}$, which returns fail to C. If $\mathcal{F}_{TP}$ returns $((c_1', W_{1,1}'), (c_2', W_{2,1}'), state)$ and $\mathcal{F}_{UDB}$ returns success to C after receiving a $(sid, \text{Post}, \cdot)$ message, C outputs success. Respectively, in the ideal world, $\mathcal{S}$, simulating $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, returns allow to $\mathcal{F}_{ns}$, which first checks that id is not registered, adds the pair (id, pk') and sends success to C. In both cases, C returns consistent outputs in the hybrid and ideal world.

(b) **The identity id is not registered and the last record for id in DBstate′ is of the form** $(\textbf{Revoke}, \textbf{id}, \textbf{pk}, \textbf{j})$**, or there is no record for id:** The reasoning is similar to that of the previous case (Reg2(a)).

(c) **The identity id is registered, but the last record for id in DBstate′ is of the form** $(\textbf{Revoke}, \textbf{id}, \textbf{pk}, \textbf{j})$**, or there is no record for id:** In the hybrid world, an *honest* client C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate'$ from $\mathcal{F}_{UDB}$, she runs Steps 3a, 3b of protocol π_{RSA} (Figure 5.3) and computes $W_{2,1}$, a non membership witness for $(id, j+1, a)$ in c_2 and $W_{2,2}$, a membership witness for (id, j, d) in c_2, or sets $W_{2,2} = \perp$ for the case where there is no record for id in $DBstate'$. Then, C sends $(sid, \text{Register}, id, pk, j+1, W_{2,1}, W_{2,2})$ to $\mathcal{F}_{TP}$. We stress that, using the same arguments as in Reg1(b), if an honest client C, given an accumulated set $X_2' \neq X_2$, could produce a valid non membership witness $W_{2,1}$ for $(id, j+1, a)$ in c_2 and a valid membership witness $W_{2,2}$ for (id, j, d) in c_2 with non negligible probability, by invoking the witness generation algorithms, then, the security of accumulator c_2 would be violated. Therefore, $\mathcal{F}_{TP}$ returns $(\text{fail}, state)$ to C and C returns fail in the hybrid world. C also

[1]As we explained in Section 5.2, X_2' is derived from $DBstate'$ as follows: For any record of the form $(\text{Register}, id, pk, i, W_{1,1}, W_{1,2})$, (id, i, a) is added to X_2, and, for any record of the form $(\text{Revoke}, id, pk, i)$, (id, i, d) is added to X_2.

returns fail in the ideal world, since $\mathcal{S}$ simulates $\mathcal{F}_{TP}$ and C and returns fail to $\mathcal{F}_{ns}$. A *corrupted* client C, in the hybrid world may send $(\text{Register}, id, pk^*, \ell, W^*_{2,1}, W^*_{2,2})$ to convince $\mathcal{F}_{TP}$ that id is not registered. Following the same analysis as in subcase Reg1(b), it is evident that C returns the same output both in the hybrid and the ideal world.

(d) **The identity id is registered, but the last record for id in DBstate′ is of the form $(\textbf{Register}, \textbf{id}, \textbf{pk}, \textbf{i}+\textbf{1}, \textbf{W}_{\textbf{1,1}}, \textbf{W}_{\textbf{1,2}})$:** The reasoning is similar to that of the previous case (Reg2(c)).

Reg3: *$\mathcal{A}$ has sent $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \text{Register}, id, pk)$ is sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$, but the set X_2' derived by $DBstate'$ is the same as X_2 that is accumulated in c_2.* In this case, a similar reasoning with Reg1 can be followed, because the accumulated set remains the same and, thus, an honest client is able to compute correctly the witnesses $W_{2,1}, W_{2,2}$.

$\mathcal{Z}$ sends a message $(\textbf{sid}, \textbf{Revoke}, \textbf{id}, \textbf{pk}, \textbf{aux})$ to a client C: In all the cases below, we assume that $R(pk, \text{aux}) = 1$, otherwise, it can be easily observed that a client C returns fail both in the hybrid and ideal world.

Rev1: $\mathcal{A}$ has **not** sent $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $\mathcal{Z}$ sends a message $(sid, \text{Revoke}, id, pk, \text{aux})$ to C. We consider two different subcases:

(a) $(\textbf{id}, \textbf{pk})$ **is registered:** In the hybrid world, an *honest* client C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and $\mathcal{F}_{UDB}$ responds with $DBstate$. C computes the witnesses $W_{1,1}, W_{1,2}$, by following the procedure described in Steps 4a, 4b, of protocol π_{RSA} (Figure 5.3). Recall that $W_{1,1}$ is a membership witness for (id, pk, i, a) in c_1 and $W_{1,2}$ is a non membership witness for (id, pk, i, d) in c_1. Then, C sends $(sid, \text{Revoke}, id, pk, i, W_{1,1}, W_{1,2}, \text{aux})$ to $\mathcal{F}_{TP}$ and, since $R(pk, \text{aux}) = 1$, $\mathcal{F}_{TP}$ returns $((c_1, W'_{1,1}), (c_2, W'_{2,1}), state)$ to C. Then, C sends $(sid, \text{Post}, (\text{Revoke}, id, pk, i))$ to $\mathcal{F}_{UDB}$ and $\mathcal{F}_{UDB}$ returns success to C. In the ideal world, $\mathcal{S}$, upon receiving $(sid, \text{Revoke}, id, pk, \text{aux})$, simulates $C, \mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, as described in Step 4 of Figure B.1, and sends allow to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ checks that $R(pk, \text{aux}) = 1$, $(id, pk) \in X$, deletes the pair (id, pk) and sends success to C. Thus, an honest client C returns success both in the hybrid and ideal world.

(b) $(\textbf{id}, \textbf{pk})$ **is not registered:** In the hybrid world, a *corrupted* client C may try to convince $\mathcal{F}_{TP}$ that (id, pk) is *currently* registered. Therefore, C must find some i and compute a membership witness $W_{1,1}$ for (id, pk, i, a) in c_1 and a non membership witness $W_{1,2}$ for (id, pk, i, d) in c_1.
Consider the case where (id, pk) has been registered before, and more precisely, assume that (id, pk) has been registered ℓ times in the past. If C chooses $i \in \{1, \ldots, \ell\}$, this means that $(id, pk, i, a) \in X_1$ and $(id, pk, i, d) \in X_1$, since (id, pk) is not currently registered. Although C can compute a membership witness $W_{1,1}$ for $(id, pk, i, a) \in X_1$, due to the security of accumulator c_1, C cannot compute a non membership witness $W_{1,2}$ for $(id, pk, i, d) \in X_1$, except with negligible probability. Therefore, $\mathcal{F}_{TP}$, upon receiving $(\text{Revoke}, id, pk, i, W_{1,1}, W_{1,2})$, will return fail, since program P_{RSA} (Figure 5.2) will fail at Step 3c. If C chooses $i > \ell$, it holds that $(id, pk, i, a) \notin X_1$ and $(id, pk, i, d) \notin X_2$. C is able to compute a non membership witness $W_{1,2}$ for $(id, pk, i, d) \notin X_2$, however, since $(id, pk, i, a) \notin X_1$, C cannot compute a member-

ship witness $W_{1,1}$ for (id, pk, i, a) in c_1, due to the security of accumulator c_1. As a result, $\mathcal{F}_{TP}$, on input (Revoke, $id, pk, i, W_{1,1}, W_{1,2}$), will also return fail in this case. Similarly, for the case where (id, pk) has never been registered before, via similar reasoning, $\mathcal{F}_{TP}$ will again return fail, due to the security of accumulator c_1.
In the ideal world, in all the aforementioned cases, $\mathcal{S}$, simulates $\mathcal{F}_{TP}, \mathcal{F}_{UDB}$ and since $\mathcal{F}_{TP}$ returns $(\text{fail}, state)$, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$. In turn, $\mathcal{F}_{ns}$ sends fail to C.

Rev2: $\mathcal{A}$ has sent $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before a message of the form $(sid, \text{Revoke}, id, pk, \text{aux})$ is sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$ and at least one of the accumulated sets X_i' $(i \in \{1, 2\})$ derived by $DBstate'$ is different from the corresponding one derived by $DBstate$: Considering how X_1', X_2' are computed from $DBstate'$ (Section 5.2), there are two possible scenarios: (1) $X_2' \neq X_2$ and $X_1' \neq X_1$ and, (2) $X_2' = X_2$ and $X_1' \neq X_1$. For both scenarios, we have the following subcases:

(a) $(\textbf{id}, \textbf{pk})$ **is registered**: In the hybrid world, an ***honest*** client C sends $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and then C computes the witnesses $W_{1,1}, W_{1,2}$ according to Steps 4a, 4b of Figure 5.3. C sends (Revoke, $id, pk, j, W_{1,1}, W_{1,2}$) to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns fail, then C returns fail, otherwise, C sends to $\mathcal{F}_{UDB}$ $(sid, \text{Post}, (\text{Revoke}, id, pk, j))$ and returns success in the hybrid world. In the ideal world, $\mathcal{S}$, simulates $C, \mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ and if $\mathcal{F}_{TP}$ returns fail, then $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$ and $\mathcal{F}_{ns}$ returns fail to C. Even in the case where $\mathcal{F}_{TP}$ returns success, it means that $\mathcal{S}$, simulating $\mathcal{F}_{TP}$, will return allow to $\mathcal{F}_{ns}$. Since (id, pk) is registered, $\mathcal{F}_{ns}$ will delete the pair (id, pk) and return success to C. Therefore, C returns consistent outputs both in the hybrid and ideal world.

(b) $(\textbf{id}, \textbf{pk})$ **is not registered**: In the hybrid world, an ***honest*** client C will initially send $(sid, \text{RetrieveDB})$ to $\mathcal{F}_{UDB}$. $\mathcal{F}_{UDB}$ returns $DBstate'$ to C, who then checks whether there is a record of the form $(\text{Register}(id, pk, j, W_{1,1}, W_{1,2}))$ record in $DBstate'$. If not, C returns fail. In the ideal world, $\mathcal{S}$ simulates C and $\mathcal{F}_{TP}$ and returns fail to $\mathcal{F}_{ns}$. Thus, C returns fail in the ideal world as well. If C's registration record exists in $DBstate'$, C, following Steps 4a, 4b of Figure 5.3, computes the witnesses $W_{1,1}, W_{1,2}$ and sends (Revoke, $id, pk, j, W_{1,1}, W_{1,2}$, aux) to $\mathcal{F}_{TP}$. By the security of accumulator c_1, $\mathcal{F}_{TP}$ returns $(\text{fail}, state)$. If an honest client C, given $DBstate'$, could convince $\mathcal{F}_{TP}$ that (id, pk) is currently registered, then, following similar arguments as in case Rev1(b), the accumulator's security would be violated. Therefore, C returns fail in the hybrid world. In the ideal world, since $\mathcal{S}$ simulates C and $\mathcal{F}_{TP}$ sends fail to $\mathcal{F}_{ns}$, which returns fail to C. Consequently, C also returns fail in the ideal world. A ***corrupted*** client C may try to convince $\mathcal{F}_{TP}$ that (id, pk) is currently registered, by sending (Revoke, $id, pk, j^*, W_{1,1}^*, W_{1,2}^*$, aux) to $\mathcal{F}_{TP}$. As in Rev1(b), $\mathcal{F}_{TP}$ will return $(\text{fail}, state)$. Therefore, C returns fail both in the hybrid and ideal world.

Rev3: $\mathcal{A}$ has sent $(sid, \text{ChangeDBstate}, DBstate')$ before $(sid, \text{Revoke}, id, pk, \text{aux})$ is sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$ but the sets X_1', X_2' derived by $DBstate'$ are the same as X_1, X_2: In this case, a similar analysis with Rev1 can be followed, since the witnesses $W_{1,1}, W_{1,2}$ can be computed correctly.

$\mathcal{Z}$ **sends** $(\textbf{sid}, \textbf{Retrieve}, \textbf{id})$ **to a client C**: Proving that a client C in this case returns consistent outputs in the hybrid and ideal world is simpler than previous cases where a Register or a Revoke message is sent to C. This is because the only interaction between C and $\mathcal{F}_{TP}$ in protocol π_{RSA} involves C sending a message $(sid, \text{RetrieveState})$ to

$\mathcal{F}_{TP}$, which replies by sending the values of the two accumulators.

Consider the case where $\mathcal{A}$ has *not* altered the contents of $\mathcal{F}_{UDB}$ and C behaves *honestly*. In the hybrid world, C will output some pk, if a pair (id, pk) is registered, and fail otherwise. In the ideal world, $\mathcal{S}$, simulating C, $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, as in Step 5 of Figure B.1, will return allow to $\mathcal{F}_{ns}$ in the case where a pair (id, pk) is currently registered, and fail otherwise. Therefore, $\mathcal{F}_{ns}$ will return pk in the former case and fail in the latter case, exactly as in the hybrid world.

In the case where $\mathcal{A}$, by sending $(sid, \text{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$, has altered the contents of $\mathcal{F}_{UDB}$, an *honest* client C behaves as follows. First, in any scenario where the last record of $DBstate$ is a Revoke record, C will fail at Step 5a of protocol π_{RSA} and will return fail. In the ideal world, $\mathcal{S}$, simulating C, $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, will return fail to $\mathcal{F}_{ns}$, and, thus, C will return fail. Second, if the last record of $DBstate$ is a register record and id is *not* currently registered, then at least one of the verification algorithms in Step 5b (Figure 5.3) will fail, based on the witnesses computed by C in Step 5a, due to the security of accumulator c_1. Therefore, C will return fail in the hybrid world and it is evident that C will also return fail in the ideal world.

The analysis for the case where C is a *corrupted* client is similar to the case where C is honest, as C can still send $(sid, \text{Retrievestate})$ to $\mathcal{F}_{TP}$, but has no advantage in influencing the output of $\mathcal{F}_{TP}$.

Regarding the cases where $(sid, \text{VerifyID}, id)$ or $(sid, \text{VerifyMapping}, id, pk)$ is sent by the environment $\mathcal{Z}$ to a client C, it can be easily observed, by the description of Steps 6 and 7 of protocol π_{RSA}, that the analysis is similar to the case where $\mathcal{Z}$ sends $(sid, \text{Retrieve}, id)$. This completes our proof. □

REFERENCES

[1] Amazon elastic file system. https://aws.amazon.com/efs/.

[2] Ascap, prs and sacem join forces for blockchain copyright system. https://www.musicbusinessworldwide.com/ascap-prs-sacem-join-forces-blockchain-copyright-system/. Accessed: 2017-07-06.

[3] Consensys: Ethereum multisigwallet. https://github.com/ConsenSys/MultiSigWallet.

[4] Cryptokitties. https://www.cryptokitties.co/.

[5] Eip 103: Blockchain rent. https://tinyurl.com/yc3uc4ak.

[6] Eip 1418 blockchain rent: fixed cost per word-block. https://github.com/ethereum/EIPs/issues/1418.

[7] Eip20 - erc20 token standard. https://tinyurl.com/ycd8mzb3.

[8] Emercoin - distributed blockchain services for business and personal use. http://www.emercoin.com.

[9] Erc20 token market capitalization. https://etherscan.io/tokens.

[10] Erc721 - a class of unique tokens. http://erc721.org/.

[11] Eth gas station. https://ethgasstation.info/.

[12] Ether - ethereum homestead 0.1 documentation. https://tinyurl.com/ybcerf39.

[13] Ethereum - merkle patricia tree. https://tinyurl.com/zl2z4m8.

[14] Ethereum average gas limit chart. https://tinyurl.com/yaokfvl2.

[15] Ethereum name service. https://ens.domains/.

[16] Ethereum price chart us dollar (eth/usd). https://tinyurl.com/jxsjqqd.

[17] Ethereum wiki - erc20 token standard. https://tinyurl.com/yd9fnw9q.

[18] Evm opcodes and instruction reference. https://tinyurl.com/ya7o3t6c.

[19] Github repository of all our implementations. https://github.com/razaden.

[20] Go ethereum. https://tinyurl.com/jkw5ow9.

[21] Google: Leveldb. https://github.com/google/leveldb.

[22] Ibm pushes blockchain into the supply chain. https://www.wsj.com/articles/ibm-pushes-blockchain-into-the-supply-chain-1468528824. Accessed: 2017-07-06.

[23] Ipfs is the distributed web. https://ipfs.io/.

[24] Namecoin. https://namecoin.org/.

[25] Nist: Digital signature standard (dss). https://tinyurl.com/ybjakloz.

[26] Nist: Recommendation for key management. https://tinyurl.com/ybcmxqlv.

[27] solc-js. https://github.com/ethereum/solc-js.

[28] Solidity. https://solidity.readthedocs.io/en/v0.5.3/.

[29] Storj - decentralized cloud object storage that is affordable, easy to use, private, and secure. https://storj.io/.

[30] Swarm. https://tinyurl.com/y7fz8q3u.

[31] Swiss industry consortium to use ethereum's blockchain. https://www.ccn.com/swiss-industry-consortium-use-ethereums-blockchain/. Accessed: 2017-07-06.

[32] Truffle suite. https://truffleframework.com/.

[33] Zerocoin: Solidity big number library. https://tinyurl.com/yaa34saq.

[34] Ethereum?s vitalik buterin wants to create annual ?rent? fees. https://tinyurl.com/yal56med, July 2018.

[35] A simple and principled way to compute rent fees. https://tinyurl.com/y9vv6w59, March 2018.

[36] Vitalik wants you to pay to slow ethereum's growth. https://tinyurl.com/y9gj8zvz, March 2018.

[37] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *Proceedings of the 9th International Conference on Cooperative Information Systems*. Springer-Verlag, 2001.

[38] Mustafa Al-Bassam. Scpki: A smart contract-based pki and identity system. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, 2017.

[39] Pawel Bylica Alex Beregszaszi. Eip 145. https://tinyurl.com/yc4khbj8.

[40] M. Ali, J. Nelson, R. Shea, and M.J. Freedman. Blockstack: A global naming and storage system secured by blockchains. In *USENIX Annual Technical Conferencei (ATC)*, 2016.

[41] androlo. Solidity contracts. https://tinyurl.com/ya9s68dh.

[42] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. Weed Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the 13th EuroSys Conference*. ACM, 2018.

[43] A. Avramidis, P. Kotzanikolaou, C. Douligeris, and M. Burmester. Chord-pki: A distributed trust infrastructure based on p2p networks. *Computer Networks*, 56, January 2012.

[44] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[45] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Proceedings of the 37th Annual International Conference on the Advances in Cryptology (CRYPTO)*, 2017.

[46] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[47] Niko Bari and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, 1997.

[48] Jordi Baylina. Eip 1109. https://tinyurl.com/yckxjogx.

[49] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *Proceedings of Advances in Cryptology - Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 1993.

[50] Joseph Bonneau. Eth iks: Using ethereum to audit a coniks key transparency log. In *Financial Cryptography and Data Security - International Workshops, FC 2016, BITCOIN, VOTING, and WAHC*, 2016.

[51] Vitalik Buterin. Eip 198. https://tinyurl.com/y9mhw6jz.

[52] Vitalik Buterin. Transaction fee economics. https://tinyurl.com/y8ckvboh.

[53] Philippe C., Alejandro H., Marcos A. K., and Roberto O. Strong accumulators from collision-resistant hashing. In *11th International Conference on Information Security*. Springer Berlin Heidelberg, 2008.

[54] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proceedings of the 22ond Annual Internation Conference on Advances in Cryptology (CRYPTO)*, 2002.

[55] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 2001.

[56] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 1979.

[57] Suranjan Choudhury, Kartik Bhatnagar, and Wasim Haque. *Public Key Infrastructure Implementation and Design*. John Wiley & Sons, Inc., 1st edition, 2002.

[58] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008.

[59] A. Datta, M. Hauswirth, and K. Aberer. Beyond ”web of trust”: Enabling p2p e-commerce. In *IEEE International Conference on E-Commerce*, June 2003.

[60] John R. Douceur. The sybil attack. Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS), 2001.

[61] C. Ellison and B. Schneier. Ten risks of pki: What you’re not being told about public key infrastructure. *Computer Security Journal*, 16, December 2000.

[62] E. Feinler, K. Harrenstien, Z. Su, and V. White. Dod internet host table specification. RFC 810, RFC Editor, March 1982.

[63] D. Fisher. Final report on diginotar hack shows total compromise of ca servers. `https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/`. [Online; posted 31-October-2012].

[64] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014.

[65] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, 2015.

[66] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *Proceedings of the 37th Annual International Conference on the Advances in Cryptology (CRYPTO)*, 2017.

[67] Nishant Garg. *Apache Kafka*. Packt Publishing, 2013.

[68] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *EUROCRYPT*, 1999.

[69] Bela Gipp, Norman Meuschke, and André Gernandt. Decentralized trusted timestamping using the crypto currency bitcoin. *iConference*, 2015.

[70] GnuPG. Libgcrypt. `https://tinyurl.com/yaa8m7ao`.

[71] D. Goodin. Google takes symantec to the woodshed for mis-issuing 30,000 https certs. `https://arstechnica.com/information-technology/2017/03/google-takes-symantec-to-the-woodshed-for-mis-issuing-30000-https-certs/`. [Online; posted 24-March-2017].

[72] P. Gutmann. Pki: it’s not dead, just resting. *Computer*, 35, August 2002.

[73] R. Housley, W. Ford, W. Polk, and D. Solo. Internet x.509 public key infrastructure certificate and crl profile. RFC 2459, RFC Editor, January 1999.

[74] Ed. J. Sermersheim. Lightweight directory access protocol (ldap): The protocol. RFC 4511, RFC Editor, June 2006.

[75] Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Compact accumulator using lattices. In *Proceedings of the 5th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, 2015.

[76] M. Karakaya, I. Korpeoglu, and O. Ulusoy. Free riding in peer-to-peer networks. *IEEE Internet Computing*, 13, March 2009.

[77] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[78] F. Lesueur, L. Me, and V. V. T. Tong. An efficient distributed pki for structured p2p networks. In *IEEE P2PC*, 2009.

[79] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS)*, 2007.

[80] Google LLC. `https://www.chromium.org/Home/chromium-security/root-ca-policy`. [Online; posted 04-May-2016].

[81] Atefeh Mashatan and Serge Vaudenay. A fully dynamic universal accumulator. *Proceedings of the Romanian Academy*, 2013.

[82] M. Masnick. Trustwave admits it issued a certificate to allow company to run man-in-the-middle attacks. `https://www.techdirt.com/articles/20120208/03043317695/trustwave-admits-it-issued-certificate-to-allow-company-to-run-man-in-the-middle-attacks.shtml`. [Online; posted 08-February-2012].

[83] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security)*, 2015.

[84] P. V. Mockapetris. Domain names - concepts and facilities. RFC 882, RFC Editor, November 1983.

[85] P. V. Mockapetris. Domain names: Implementation specification. RFC 883, RFC Editor, November 1983.

[86] P. V. Mockapetris. Domain names: Implementation and specification. RFC 1035, RFC Editor, November 1987.

[87] Ruggero Morselli, Bobby Bhattacharjee, Jonathan Katz, and Michael A Marsh. Keychains: A decentralized public-key infrastructure. *Technical Reports from UMIACS*, 2006.

[88] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`. [Online; posted 31-October-2008].

[89] Lan Nguyen. Accumulators from bilinear pairings and applications. In *Proceedings of the Conference on Topics in Cryptology - The Cryptographers' Track at the RSA Conference (CT-RSA)*, 2005.

[90] Kaisa Nyberg. Fast accumulated hashing. In *Proceedings of the 3rd International Workshop on Fast Software Encryption (FSE)*, 1996.

[91] C. Patsonakis and M. Roussopoulos. An alternative paradigm for developing and pricing storage on smart contract platforms. In *IEEE International Conference on Decentralized Applications and Infrastructures*. IEEE, 2019.

[92] C. Patsonakis, K. Samari, M. Roussopoulos, and A. Kiayias. Towards a smart contract-based, decentralized, public-key infrastructure. In *16th International Conference on Cryptology and Network Security*. Springer International Publishing, 2018.

[93] C. Patsonakis, K. Samari, M. Roussopoulos, and A. Kiayias. On the practicality of a smart contract pki. In *IEEE International Conference on Decentralized Applications and Infrastructures*. IEEE, 2019.

[94] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[95] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mlsm: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage*, 2018.

[96] Michael K. Reiter, Matthew K. Franklin, John B. Lacy, and Rebecca N. Wright. The omega key management service. In *ACM Conference on Computer and Communications Security (CCS)*, 1996.

[97] Barkley Rosser. Explicit bounds for some functions of prime numbers. *American Journal of Mathematics*, 1941.

[98] J. Salowey and S. Turner. Iana registry updates for tls and dtls. RFC 8447, RFC Editor, August 2018.

[99] A. Slagell, R. Bonilla, and W. Yurcik. A survey of pki components and scalability issues. In *IEEE International Performance Computing and Communications Conference*, April 2006.

[100] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *Proceeding of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[101] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2, 1997.

[102] Tomas Sanderand Amnon Ta-Shma and Moti Yung. Blind, auditable membership proofs. In *Proceedings of the 4th International Conference on Financial Cryptography (FC)*, 2000.

[103] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[104] S. Turner. The application/pkcs10 media type. RFC 5967, RFC Editor, August 2010.

[105] V. Buterin. A simple and principled way to compute rent fees. `https://tinyurl.com/y9vv6w59`.

[106] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `https://github.com/ethereum/yellowpaper`. [Online; posted April-2014].

[107] Gavin Wood. Ethereum yellow paper. `https://tinyurl.com/yaptyawg`.

[108] Rita H. Wouhaybi and Andrew T. Campbell. Keypeer: A scalable, resilient distributed public-key system using chord, 2008.

[109] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[110] W. Kim Y. Dong and R. Boutaba. Conifer: centrally-managed pki with blockchain-rooted trust. In *IEEE International Conference on Blockchain (Blockchain)*, 2018.

[111] A. Yakubov, W. M. Shbair, A. Wallbom, D. Sanda, and R. State. A blockchain-based pki management framework. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2018.

[112] E. Yüce and A. A. Selçuk. Server notaries: a complementary approach to the web pki trust model. *IET Information Security*, 12, September 2018.

[113] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[114] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.