ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Comparative Analysis of Stateful Fuzzing Techniques Applied on EDHOC Implementations

---

Συγγραφέας:
Ιωάννης Τσουλούκας

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Αθήνα, Απρίλιος 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Comparative Analysis of Stateful Fuzzing Techniques Applied on EDHOC Implementations

---

Συγγραφέας:
Ιωάννης Τσουλούκας

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Απριλίου 2024.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Κωνσταντίνος Σαγώνας Αναπληρωτής Καθηγητής	Δημήτριος Φωτάκης Καθηγητής	Αριστείδης Παγουρτζής Καθηγητής

Αθήνα, Απρίλιος 2024

Declaration of Authorship

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μή κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθέι ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Υπογραφή: Γιάννης Τσουλούκας

Ημερομηνία: 16/4/2024

Copyright © Ιωάννης Τσουλούκας, 2024.
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Περίληψη

Η παρούσα διπλωματική εργασία διερευνά την έννοια του stateful fuzzing, μιας εξειδικευμένης μορφής fuzz testing που λαμβάνει υπόψη την κατάσταση του συστήματος υπό έλεγχο. Παρουσιάζει μια νέα προσέγγιση όπου το υποκείμενο μοντέλο μηχανής κατάστασης του υπό δοκιμή συστήματος μαθαίνεται πριν από την πραγματική διαδικασία fuzzing, αντί κατά τη διάρκειά της. Η αποτελεσματικότητα αυτής της μεθόδου συγκρίνεται με πρόσφατους fuzzers, τους AFLNet και StateAFL στο πλαίσιο του πρωτοκόλλου EDHOC, ενός πρωτοκόλλου ασφαλούς ανταλλαγής κλειδιών.

Λέξεις κλειδιά: ασφάλεια πρωτοκόλλων, stateful fuzzing, active automata learning, protocol state fuzzing, AFLNet, StateAFL, EDHOC

Abstract

This thesis explores the concept of stateful fuzzing, a specialized form of fuzz testing that considers the state of the system being tested. It presents a novel approach where the underlying state machine model of the system under test is learned before the actual fuzzing process, rather than during it. The effectiveness of this method is compared with state-of-the-art fuzzers, AFLNet and StateAFL in the context of the EDHOC protocol, a secure key exchange protocol.

Keywords: protocol security, stateful fuzzing, active automata learning, protocol state fuzzing, AFLNet, StateAFL, EDHOC

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Kostis Sagonas, for his invaluable guidance, encouragement, and support throughout this project. He introduced me to the fascinating field of fuzzing and inspired me with his enthusiasm and expertise. He also provided me with constructive feedback and useful suggestions that helped me improve the quality of my work.

I am also grateful to Rémi Parrot, the creator of AFLML, for his generous assistance and mentorship. He kindly shared his knowledge and experience with me and helped me overcome many technical challenges. He also gave me access to his code, which enabled me to conduct my experiments and analysis.

Thanasis Typaldos, the mastermind behind EDHOC-Fuzzer, deserves special thanks for his assistance with EDHOC-Fuzzer and EDHOC. More than just granting me permission, he provided me with guidance in modifying his code to suit my requirements. His contribution has been truly invaluable.

Without the help of these people, this work would not have been possible. I appreciate their time and effort and I am honored to have worked with them.

Contents

Περίληψη **6**

Abstract **8**

Acknowledgements **10**

1 Εκτεταμένη Ελληνική Περίληψη **16**
1.1 Εισαγωγή 16
1.1.1 Συνεισφορά 17
1.1.2 Περίγραμμα Ενοτήτων 17
1.2 Θεωρητικό Υπόβαθρο 18
1.2.1 Fuzzing 18
1.2.2 Stateful Fuzzing 18
1.2.3 Active Automata Learning 19
1.2.4 Protocol State Fuzzing 19
1.3 AFL 20
1.3.1 Stateful Fuzzers 20
1.3.2 AFLNet 20
1.3.3 StateAFL 21
1.3.4 AFLML 21
1.3.5 Σχετική έρευνα 22
1.4 Πρωτόκολλα 24
1.4.1 CoAP 24
1.4.2 EDHOC 24
1.4.3 OSCORE 27
1.5 Προετοιμασία Πειραμάτων 28
1.5.1 Συστήματα που χρησιμοποιήθηκαν 28
1.5.2 Ρυθμίσεις 30
1.6 Αποτελέσματα 31
1.6.1 Σύγκριση μηχανών καταστάσεων 31
1.6.2 Σύγκριση κάλυψης κώδικα 33
1.6.3 Σύγκριση των Crashes 33
1.7 Συμπεράσματα 36
1.8 Μελλοντική έρευνα 36

2 Introduction **39**
2.1 Contributions 40
2.2 Outline 40

3 Background **41**
3.1 Fuzzing . . . 41
3.2 Stateful Fuzzing . . . 42
3.3 Active Automata Learning . . . 42
3.4 Protocol State Fuzzing . . . 43
3.5 AFL . . . 43
3.6 Stateful Fuzzers . . . 44
3.6.1 AFLNet . . . 44
3.6.2 StateAFL . . . 44
3.6.3 AFLML . . . 45
3.7 Related Work . . . 46

4 Protocols **47**
4.1 CoAP . . . 47
4.2 EDHOC . . . 48
4.3 OSCORE . . . 49

5 Experimental Setup **51**
5.1 Systems Used . . . 51
5.1.1 uOSCORE-uEDHOC . . . 51
5.1.2 EDHOC-Fuzzer . . . 51
5.1.3 AFLNet . . . 52
5.1.4 StateAFL . . . 52
5.1.5 AFLML . . . 53
5.2 Settings . . . 53

6 Results **56**
6.1 State Machine Comparison . . . 56
6.2 Coverage Comparison . . . 58
6.3 Crashes Comparison . . . 58

7 Conclusion **62**

Bibliography **64**

List of Figures

1.1 Μορφή CoAP μηνυματος 25
1.2 Κωδικοί απόκρισης του CoAP 25
1.3 Ροή μηνυμάτων EDHOC 26
1.4 Ροή μηνυμάτων EDHOC με τον server ως responder 27
1.5 Μοντέλο μηχανής καταστάσεων από τον EDHOC-Fuzzer. 31
1.6 Μοντέλο μηχανής καταστάσεων από τον AFLNet. 32
1.7 Μοντέλο μηχανής καταστάσεων από τον StateAFL (εν μέρει). . . . 33
1.8 Κάλυψη κώδικα των διαφόρων fuzzers με την πάροδο του χρόνου. 34
1.9 AddressSanitizer summary 34
1.10 Μέσος αριθμός unique και distinct crashes που προκλήθηκαν από διαφορετικούς fuzzers με την πάροδο του χρόνου. 35

4.1 CoAP Message Format 48
4.2 CoAP Response Codes 48
4.3 EDHOC message flow 50
4.4 EDHOC message flow with server as responder 50

5.1 Code coverage varying the response waiting time. 54
5.2 Number of crashes (averages from ten runs) varying the response waiting time. 55

6.1 EDHOC-Fuzzer learned model of the SUT. 57
6.2 AFLNet state machine model. 57
6.3 StateAFL state machine model (partial). 58
6.4 Code coverage of the different fuzzers over time (absolute numbers). 59
6.5 Code coverage of the different fuzzers over time (%). 60
6.6 AddressSanitizer summary 60
6.7 Average number of unique and distinct crashes triggered by different fuzzers over time. 61

List of Tables

1.1 Επιλογές που χρησιμοποιήθηκαν για τον AFLNet 29
1.2 Επιλογές που χρησιμοποιήθηκαν για τον StateAFL 30
1.3 Επιλογές που χρησιμοποιήθηκαν για τον AFLML 30

5.1 Options used for AFLNet 52
5.2 Options used for StateAFL 53
5.3 Options used for AFLML 53

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Το software testing είναι σημαντικό για τη διασφάλιση της ποιότητας και της ασφάλειας των συστημάτων λογισμικού. Ωστόσο, η μη αυτόματη διαδικασία ελέγχου μπορεί να είναι χρονοβόρα, κοστοβόρα και επιρρεπής σε σφάλματα. Ως εκ τούτου, έχουν αναπτυχθεί αυτοματοποιημένες τεχνικές ελέγχου με στόχο τη μείωση της ανθρώπινης παρέμβασης και την αύξηση της αποτελεσματικότητας του ελέγχου. Μια από τις πιο δημοφιλείς και επιτυχημένες τεχνικές αυτοματοποιημένου ελέγχου είναι το fuzzing [1], το οποίο περιλαμβάνει την παροχή τυχαίων ή ημι-τυχαίων εισόδων στο υπό δοκιμή σύστημα και την παρατήρηση της συμπεριφοράς του. Το fuzzing μπορεί να αποκαλύψει σφάλματα και κενά ασφαλείας που μπορεί να μην είναι εύκολα ανιχνεύσιμα με άλλες μεθόδους testing. Έχει ιδιαίτερη αξία για τη δοκιμή πολύπλοκων συστημάτων όπου η ασφάλεια είναι κρίσιμη, όπως οι υλοποιήσεις πρωτοκόλλων δικτύου [2, 3].

Ωστόσο, το fuzzing δεν είναι μια γενική λύση ενιαίας εφαρμογής. Οι τεχνικές fuzzing ποικίλλουν ανάλογα με το είδος του υπό δοκιμή συστήματος. Ορισμένα συστήματα έχουν περίπλοκες εσωτερικές καταστάσεις που αλλάζουν με βάση τα μηνύματα που λαμβάνουν και στέλνουν. Σε αυτή τη διατριβή, εστιάζουμε στο stateful fuzzing, το οποίο λαμβάνει υπόψη την κατάσταση του συστήματος και μπορεί να ελέγξει συστήματα με διαφορετικές καταστάσεις και μεταβάσεις πιο αποτελεσματικά από το stateless fuzzing.

Μία από τις προκλήσεις του stateful fuzzing είναι να εξαχθεί το μοντέλο μηχανής καταστάσεων του υπό δοκιμή συστήματος, το οποίο περιγράφει τις πιθανές καταστάσεις και μεταβάσεις του. Το μοντέλο μηχανής καταστάσεων μπορεί να βοηθήσει τον fuzzer να παράγει έγκυρες και αξιόλογες εισόδους που μπορούν να προκαλέσουν διαφορετικές συμπεριφορές στο σύστημα. Οι περισσότεροι από τους υπάρχοντες stateful fuzzers συμπεραίνουν το μοντέλο μηχανής καταστάσεων κατά τη διάρκεια της διαδικασίας fuzzing, παρατηρώντας είτε τα μηνύματα που στέλνονται στο δίκτυο είτε τη μνήμη του συστήματος και ομαδοποιώντας τα σε διαφορετικές καταστάσεις. Ωστόσο, αυτές οι προσεγγίσεις μπορεί να είναι αναποτελεσματικές και ανακριβείς. Οι είσοδοι που χρησιμοποιούνται κατά τη διαδικασία fuzzing δεν είναι ιδανικές για την εκμάθηση της μηχανής καταστάσεων, επειδή δεν αντιπροσωπεύουν αποτελεσματικά τη γλώσσα του πρωτοκόλλου και μπορεί να μην εντοπίσουν ορισμένες καταστάσεις ή μεταβάσεις.

Σε αυτήν την εργασία, παρουσιάζουμε μια νέα προσέγγιση για το stateful fuzzing που μαθαίνει μια καλή προσέγγιση του υποκείμενου μοντέλου μηχανής καταστάσεων της υπό δοκιμή υλοποίησης πριν από την πραγματική διαδικασία fuzzing, αντί να την βρίσκει κατά τη διάρκειά της. Χρησιμοποιούμε μια τεχνική που ονομάζεται active automata learning [4], η οποία υποβάλλει επαναληπτικά queries στο υπό δοκιμή σύστημα με ειδικά επιλεγμένα inputs και μαθαίνει ένα μοντέλο μηχανής καταστάσεων από τις απαντήσεις.

Συγκρίνουμε αυτή την προσέγγιση με δύο πρόσφατους stateful fuzzers, τους AFLNet [5] και StateAFL [6]. Το πρωτόκολλο που χρησιμοποιούμε για αυτή τη σύγκριση είναι το EDHOC (Ephemeral Diffie-Hellman Over COSE) [7], ένα ελαφρύ και ασφαλές πρωτόκολλο ανταλλαγής κλειδιών που έχει σχεδιαστεί για περιορισμένων δυνατοτήτων συσκευές και δίκτυα. Μετράμε την κάλυψη κώδικα και τον αριθμό των ευπαθειών που ανακαλύπτονται από κάθε fuzzer. Συζητάμε επίσης τα δυνατά σημεία, τους περιορισμούς και τις προκλήσεις κάθε προσέγγισης και προτείνουμε κατευθύνσεις για μελλοντική έρευνα.

1.1.1 Συνεισφορά

Η παρούσα εργασία συμβάλλει στον τομέα της ασφάλειας δικτυακών πρωτοκόλλων μέσω μιας εκτεταμένης συγκριτικής εξέτασης των τρεχουσών τεχνικών stateful fuzzing όταν εφαρμόζονται σε υλοποιήσεις δικτυακών πρωτοκόλλων. Με τη συστηματική αξιολόγηση και σύγκριση των stateful fuzzers, στοχεύουμε να παρέχουμε πολύτιμες πληροφορίες σχετικά με τα πλεονεκτήματα και τις αδυναμίες αυτών των τεχνικών και να εντοπίσουμε πεδία για μελλοντική έρευνα. Τα συμπεράσματα που παρουσιάζουμε χρησιμεύουν για την καλύτερη συλλογική κατανόηση στο πεδίο του stateful fuzzing και του fuzzing πρωτοκόλλων. Επιπλέον, αυτή η εργασία θέτει τα θεμέλια για μελλοντικές ερευνητικές προσπάθειες, παρέχοντας παραδείγματα στησίματος και προετοιμασίας του συστήματος για fuzzing.

1.1.2 Περίγραμμα Ενοτήτων

Οι υπόλοιπες ενότητες είναι οργανωμένα ως εξής:

- Η Ενότητα 1.2 καλύπτει το βασικό θεωρητικό υπόβαθρο του stateful fuzzing και των fuzzers που χρησιμοποιούνται.
- Η Ενότητα 1.4 παρέχει τεχνικές λεπτομέρειες σχετικά με τα υπό δοκιμή πρωτόκολλα.
- Η Ενότητα 1.5 περιγράφει την προετοιμασία των πειραμάτων.
- Η Ενότητα 1.6 παρουσιάζει τα αποτελέσματα των πειραμάτων.
- Η Ενότητα 1.7 εξάγει συμπεράσματα και δίνει ιδέες για μελλοντική έρευνα.

1.2 Θεωρητικό Υπόβαθρο

1.2.1 Fuzzing

Το fuzzing [1], επίσης γνωστό ως fuzz testing, είναι μια αυτοματοποιημένη τεχνική testing που χρησιμοποιείται για την ανακάλυψη ευπαθειών και ελαττωμάτων λογισμικού με την αποστολή μεγάλου αριθμού άκυρων, τυχαίων ή απροσδόκητων εισόδων σε ένα πρόγραμμα ή σύστημα. Ο στόχος του fuzzing είναι να ανακαλύψει προβλήματα, όπως κρασαρίσματα, διαρροές μνήμης και υπερχειλίσεις buffer στο λογισμικό-στόχο, το οποίο μπορεί να περιλαμβάνει εφαρμογές, βιβλιοθήκες ή πρωτόκολλα δικτύου.

Συνήθως, οι fuzzers χρησιμοποιούνται για τον έλεγχο προγραμμάτων που δέχονται δομημένες εισόδους. Ένας αποτελεσματικός fuzzer παράγει ημι-εγκυρές εισόδους που είναι ¨αρκετά έγκυρες¨, δεδομένου ότι δεν απορρίπτονται άμεσα από τον parser και είναι ¨αρκετά μη έγκυρες¨ ώστε να αποκαλύπτουν σπάνιες καταστάσεις που δεν έχουν αντιμετωπιστεί σωστά.

Οι αρχικές είσοδοι των fuzzers, όπως αυτοί που αξιολογούμε, αντιπροσωπεύουν μια ακολουθία bytes. Θα αναφερόμαστε σε αυτές ως *concrete seeds*, ενώ θα αναφερόμαστε σε seeds που περιέχουν απλώς τον τύπο ή τους τύπους των δεδομένων (δηλ. τον τύπο των μηνυμάτων όπως ορίζεται από το πρωτόκολλο) ως *abstract seeds*.

1.2.2 Stateful Fuzzing

Το stateful fuzzing επικεντρώνεται στην εύρεση ευπαθειών σε προγράμματα που έχουν εσωτερικές καταστάσεις και αντιδρούν στις εισόδους με βάση την τρέχουσα κατάστασή τους. Αυτό που κάνει το stateful fuzzing ξεχωριστό είναι η ικανότητά του να κατανοεί την τρέχουσα κατάσταση του προγράμματος κατά τη διάρκεια της διαδικασίας ελέγχου. Σε αντίθεση με τις συμβατικές μεθόδους fuzzing, το stateful fuzzing παρακολουθεί τη συμπεριφορά ή τις εσωτερικές μεταβλητές του προγράμματος και χρησιμοποιεί αυτές τις πληροφορίες για να συμπεράνει το υποκείμενο μοντέλο μηχανής καταστάσεων και την τρέχουσα κατάσταση. Αυτό του επιτρέπει να παράγει test προσαρμοσμένα σε συγκεκριμένες καταστάσεις του προγράμματος.

Έχοντας επίγνωση της κατάστασης του προγράμματος, το stateful fuzzing μπορεί να μιμηθεί με μεγαλύτερη ακρίβεια σενάρια και αλληλεπιδράσεις του πραγματικού κόσμου. Αυτή η προσέγγιση είναι ιδιαίτερα πολύτιμη κατά τη δοκιμή πολύπλοκων συστημάτων λογισμικού, όπως τα πρωτόκολλα και οι εφαρμογές διαδικτύου, όπου η συμπεριφορά τους καθορίζεται από την ακολουθία των εισόδων και την εσωτερική κατάσταση του προγράμματος. Ορισμένα σφάλματα μπορεί να αποκαλύπτονται μόνο σε ορισμένες καταστάσεις, οι οποίες απαιτούν μια συγκεκριμένη ακολουθία εισόδων για να βρεθούν.

Η κύρια πρόκληση για το stateful fuzzing είναι η κάλυψη του χώρου καταστάσεων του συστήματος χωρίς να υπάρχει αναλυτική προδιαγραφή του πρωτοκόλλου. Αυτό περιλαμβάνει τη μερική ανακάλυψη του χώρου καταστάσεων του πρωτοκόλλου και την ενσωμάτωση στρατηγικών για τον εντοπισμό καταστάσεων.

1.2.3 Active Automata Learning

Το Active Automata Learning (AAL) [4] είναι μια αυτοματοποιημένη μέθοδος για την κατασκευή ενός μοντέλου μηχανής καταστάσεων που προσεγγίζει καλά τη συμπεριφορά του συστήματος. Αυτό επιτυγχάνεται παρέχοντας εισόδους και παρατηρώντας τις εξόδους του συστήματος. Η διαδικασία αυτή περιλαμβάνει δύο φάσεις:

1. Κατασκευή υποθέσεων: Σε αυτή τη φάση, αποστέλλονται στο σύστημα ακολουθίες συμβόλων εισόδου. Οι απαντήσεις του συστήματος χρησιμοποιούνται για τη δημιουργία ενός απλοποιημένου μοντέλου. Ο στόχος είναι να δημιουργηθεί ένα μοντέλο που παράγει τις ίδιες εξόδους με το σύστημα για όλες τις ακολουθίες εισόδου που αποστέλλονται στο σύστημα. Για τις ακολουθίες εισόδου που δεν χρησιμοποιούνται, το μοντέλο κάνει υποθέσεις με βάση αυτά που έχει μάθει.

2. Επικύρωση υποθέσεων: Σε αυτή τη φάση, το μοντέλο δοκιμάζεται για να διαπιστωθεί αν αναπαριστά με ακρίβεια το σύστημα. Περισσότερες ακολουθίες εισόδου αποστέλλονται τόσο στο σύστημα όσο και στο μοντέλο. Εάν υπάρχουν διαφορές στις εξόδους τους, αυτό υποδηλώνει ότι το μοντέλο χρειάζεται βελτίωση και η διαδικασία επιστρέφει στη φάση κατασκευής υποθέσεων για την τελειοποίηση του μοντέλου. Εάν δεν διαπιστωθούν διαφορές, η διαδικασία εκμάθησης ολοκληρώνεται και το μοντέλο θεωρείται καλή προσέγγιση του συστήματος.

Εάν αυτή η διαδικασία δεν τερματιστεί, τότε είναι πιθανό η συμπεριφορά του συστήματος να μην μπορεί να μοντελοποιηθεί από ένα ντετερμινιστικό αυτόματο πεπερασμένης κατάστασης.

1.2.4 Protocol State Fuzzing

Το Protocol State Fuzzing [8] είναι μια τεχνική που χρησιμοποιεί active automata learning, προκειμένου να βρει τη μηχανή κατάστασης της υλοποίησης ενός πρωτοκόλλου. Τα μοντέλα που μαθαίνονται χρησιμοποιούνται για την αποκάλυψη λογικών σφαλμάτων μέσω μη τυπικών ή απροσδόκητων ακολουθιών μηνυμάτων. Τροποποιώντας ακολουθίες μηνυμάτων και όχι μεμονωμένα μηνύματα, η τεχνική αυτή μπορεί να αποκαλύψει βαθύτερα ευάλωτα σημεία που μπορεί να μην εντοπίζονταν από άλλες μεθόδους.

Για τη διαδικασία εκμάθησης αυτής της τεχνικής απαιτείται ένας Learner, ένας Mapper και ένα System Under Learning (SUL). Ο Learner είναι υπεύθυνος για την αποστολή αιτημάτων στο SUL και τη λήψη των απαντήσεών του, εκτελώντας ουσιαστικά τον αλγόριθμο μάθησης. Ωστόσο, ο Learner δεν γνωρίζει πώς να μετατρέπει abstract σύμβολα εισόδου σε concrete μηνύματα πρωτοκόλλου. Το ζήτημα αυτό επιλύεται από τον Mapper.

Ο Mapper μετατρέπει τα abstract σύμβολα εισόδου από το πεπερασμένο αλφάβητο εισόδου σε concrete μηνύματα πρωτοκόλλου που αποστέλλονται στο SUL, συμπληρώνοντας τις απαραίτητες λεπτομέρειες. Αντιστοιχίζει επίσης τα concrete

μηνύματα από το SUL σε abstract σύμβολα του αλφαβήτου εξόδου, αφαιρώντας τις περιττές λεπτομέρειες που αφορούν το πρωτόκολλο.

1.3 AFL

Ο American Fuzzy Lop (AFL) [9, 10, 11] είναι ένας αποτελεσματικός fuzzer που ξεχωρίζει για την ταχύτητα, την αξιοπιστία και την ευκολία χρήσης του. Χρησιμοποιεί έναν έξυπνο συνδυασμό mutational και coverage-guided τεχνικών. Μεταλλάσσει ένα σύνολο test cases για να φτάσει σε προηγουμένως ανεξερεύνητες περιοχές του προγράμματος. Όταν ένα test case αυξάνει την κάλυψη, αποθηκεύεται στην ουρά test case.

Ο AFL χρησιμοποιεί μια ουρά για τη διαχείριση των test cases που δοκιμάζει. Από προεπιλογή, ακολουθεί μια πολιτική FIFO (First-In-First-Out), εκτελώντας τα με τη σειρά που προστίθενται. Επιπλέον, ο AFL δίνει προτεραιότητα στα μικρότερα και ταχύτερα test cases για να βελτιώσει την απόδοση. Τα επισημαίνει ως ¨favored¨. Κατά την επιλογή του επόμενου test case, ο AFL παραλείπει τα όχι favored cases με μεγάλη πιθανότητα, αν υπάρχει τουλάχιστον μια favored επιλογή διαθέσιμη. Διαφορετικά, τα cases που έχουν δοκιμαστεί προηγουμένως παραλείπονται με μεγαλύτερη πιθανότητα.

Οι μεταλλάξεις στον AFL ταξινομούνται σε δύο κύριες κατηγορίες: ντετερμινιστικές και havoc. Οι ντετερμινιστικές μεταλλάξεις περιλαμβάνουν μεμονωμένες, προκαθορισμένες μεταβολές στο περιεχόμενο των test cases, όπως αναστροφές bit, προσθήκες, αντικαταστάσεις με ακέραιους αριθμούς από ένα προκαθορισμένο σύνολο ενδιαφέρουσων τιμών και άλλα. Οι μεταλλάξεις havoc στοιβάζονται τυχαία και μπορεί να περιλαμβάνουν τη μεταβολή του μεγέθους του test case προσθέτοντας ή διαγράφοντας τμήματα της εισόδου. Επιπλέον, ο AFL μπορεί να συγχωνεύσει δύο test cases σε ένα και στη συνέχεια να εφαρμόσει havoc σε ένα μεταγενέστερο στάδιο γνωστό ως splicing stage.

Αυτός ο γενετικός αλγόριθμος και η favored επιλογή seed είναι ενσωματωμένα στους AFLNet, StateAFL και AFLML.

1.3.1 Stateful Fuzzers

Οι stateful fuzzers είναι μια κατηγορία εργαλείων fuzzing που αξιοποιούν την έννοια των καταστάσεων του συστήματος για να καθοδηγήσουν τη διαδικασία fuzzing. Είναι ειδικά αποτελεσματικοί για τον έλεγχο λογισμικού με πολύπλοκη συμπεριφορά που εξαρτάται από την τρέχουσα κατάσταση.

1.3.2 AFLNet

Ο AFLNet [5] είναι ένας greybox fuzzer σχεδιασμένος για υλοποιήσεις πρωτοκόλλων. Ο AFLNet, που βασίζεται στον AFL [9], υιοθετεί έναν γενετικό αλγόριθμο για να παράγει τις καλύτερες εισόδους. Επιπλέον, χρησιμοποιεί state feedback για να κατευθύνει τη διαδικασία fuzzing.

Ξεκινά με καταγεγραμμένες ανταλλαγές μηνυμάτων μεταξύ ενός server και ενός πραγματικού client, εξαλείφοντας την ανάγκη για προδιαγραφές πρωτοκόλλου ή γραμματικές μηνυμάτων. Ο AFLNet αναλαμβάνει το ρόλο του client, αναπαράγοντας τροποποιημένες εκδόσεις της αρχικής ακολουθίας μηνυμάτων που αποστέλλονται στον server. Διατηρεί εκείνες τις αλλαγές που αποδεικνύονται αποτελεσματικές στην επέκταση της κάλυψης κώδικα ή καταστάσεων.

Για τον εντοπισμό των καταστάσεων από τις οποίες περνά ο server όταν αποστέλλεται μια ακολουθία μηνυμάτων, ο AFLNet βασίζεται στους κωδικούς απόκρισης του server. Χρησιμοποιώντας αυτή την πληροφόρηση, ο AFLNet αποφασίζει σε ποια κατάσταση θα επικεντρωθεί στη συνέχεια, χρησιμοποιώντας διάφορες ευριστικές. Για παράδειγμα, για τον εντοπισμό καταστάσεων που ασκούνται σπάνια, επιλέγει μια κατάσταση με πιθανότητα αντιστρόφως ανάλογη του ποσοστού των μεταλλαγμένων ακολουθιών μηνυμάτων που την έχουν ασκήσει και για να μεγιστοποιήσει την πιθανότητα ανακάλυψης νέων μεταβάσεων σε καταστάσεις, ο AFLNet δίνει προτεραιότητα σε μια κατάσταση που έχει συμβάλει με ιδιαίτερη επιτυχία στην αύξηση της κάλυψης κώδικα ή καταστάσεων όταν είχε επιλεγεί προηγουμένως.

1.3.3 StateAFL

Ο StateAFL [6] είναι ένας greybox fuzzer σχεδιασμένος για servers. Σε αντίθεση με άλλους fuzzers, δεν απαιτεί μη αυτόματη παραμετροποίηση, όπως μοντέλα πρωτοκόλλων, parsers πρωτοκόλλων και learning frameworks. Αντ΄ αυτού, χρησιμοποιεί lightweight ανάλυση του προγράμματος-στόχου.

Λειτουργεί με την παρακολούθηση του server-στόχου, εισάγοντας αισθητήρες στις κατανομές μνήμης και στις λειτουργίες εισόδου/εξόδου δικτύου κατά τη μεταγλώττιση. Κατά τη διάρκεια του χρόνου εκτέλεσης, συμπεραίνει την τρέχουσα κατάσταση πρωτοκόλλου του server-στόχου, λαμβάνοντας στιγμιότυπα από περιοχές μνήμης με μεγάλη διάρκεια ζωής. Στη συνέχεια, χρησιμοποιεί ένα είδος αλγορίθμου κατακερματισμού, ο οποίος επιτρέπει κάποιο επίπεδο αλλαγής, για να μετατρέψει τα περιεχόμενα της μνήμης σε ένα μοναδικό αναγνωριστικό για τη συγκεκριμένη κατάσταση. Αυτό επιτρέπει στον StateAFL να αναγνωρίζει μια κατάσταση ακόμη και αν έχουν συμβεί μικρές αλλαγές στη μνήμη. Ανιχνεύοντας αυτές τις καταστάσεις, ο StateAFL μπορεί να κατασκευάσει σταδιακά μια μηχανή καταστάσεων πρωτοκόλλου για να καθοδηγήσει το fuzzing.

1.3.4 AFLML

Ο AFLML είναι ένας greybox fuzzer που μπορεί να χρησιμοποιηθεί σε συνδυασμό με το Protocol State Fuzzing. Αν και βασίζεται στον AFLNet, ο AFLML δεν εξάγει κωδικούς απόκρισης από τα μηνύματα. Δεν έχει γνώση της γλώσσας πρωτοκόλλου και δεν είναι ικανός να αναγνωρίσει τον τύπο των μηνυμάτων που αποστέλλονται από το υπό δοκιμή σύστημα (SUT).

Η βασική ιδέα πίσω από τον AFLML είναι ότι η απόδοση του fuzzing μπορεί να βελτιωθεί εάν η μηχανή κατάστασης του συστήματος προς fuzzing είναι γνωστή

εκ των προτέρων. Η διαδικασία fuzzing αυτού του fuzzer αφορά τα ακόλουθα βήματα:

1. Model Learning: Το πρώτο βήμα περιλαμβάνει την εκμάθηση της μηχανής καταστάσεων του SUT χρησιμοποιώντας κάποια τεχνική active automata learning.
2. Abstract Seed Generation: Μετά την εκμάθηση της μηχανής καταστάσεων του SUT, παράγονται abstract seeds που αντιστοιχούν σε όλες τις πιθανές μεταβάσεις της μηχανής καταστάσεων.
3. Concretization: Σε αυτό το βήμα, παράγονται concrete seeds από τα abstract seeds. Αυτά τα έγκυρα δεδομένα εισόδου αποστέλλονται στο SUT κατά τη διάρκεια του fuzzing.
4. Fuzzing: Με τα concrete seeds διαθέσιμα, ο fuzzer μπορεί να προχωρήσει στην πραγματική διαδικασία fuzzing. Ο AFLML μπορεί να ξεκινήσει το fuzzing από οποιαδήποτε κατάσταση του SUT χρησιμοποιώντας αυτά τα seeds, επιτρέποντάς του να εξερευνήσει διεξοδικά διαφορετικές καταστάσεις και μεταβάσεις της μηχανής καταστάσεων του SUT. Ο AFLML χρησιμοποιεί μια προσέγγιση round-robin για την επιλογή καταστάσεων χωρίς κάποια ευριστική.

Με την κατανόηση της μηχανής καταστάσεων του SUT, ο AFLML μπορεί να καθοδηγήσει τη διαδικασία fuzzing πιο αποτελεσματικά, επιτρέποντάς του να εξερευνήσει τις διάφορες καταστάσεις και μεταβάσεις με δομημένο τρόπο. Η προσέγγιση του AFLML είναι ιδιαίτερα χρήσιμη για τον έλεγχο λογισμικού με πολύπλοκες συμπεριφορές που εξαρτώνται από την κατάσταση.

1.3.5 Σχετική έρευνα

Οι υλοποιήσεις πρωτοκόλλων έχουν αναλυθεί διεξοδικά για διάφορα είδη ευπαθειών και σφαλμάτων. Ο TLS-Attacker [12] χρησιμεύει ως ένα αποδοτικό framework για τον έλεγχο υλοποιήσεων TLS. Οι de Ruiter και Poll [8] ήταν από τους πρώτους που χρησιμοποίησαν συστηματικό state fuzzing και ανέλυσαν τις μηχανές κατάστασης TLS που αποκτήθηκαν με τη χρήση model learning. Παρόμοια δουλειά έχει γίνει για το DTLS από τους Fiterau-Brostean et al. [13], που οδήγησε στη δημιουργία του εργαλείου DTLS-Fuzzer [14], ένα state fuzzing framework βασισμένο στον TLS-Attacker. Πιο πρόσφατα, ο Σαγώνας και ο Τυπάλδος [15] εφάρμοσαν επίσης το protocol state fuzzing σε υλοποιήσεις EDHOC και τις ανέλυσαν. Αυτές οι δουλειές, ωστόσο, βασίζονται στη μη αυτόματη επιθεώρηση του μοντέλου της μηχανής καταστάσεων για την εύρεση σφαλμάτων. Μια αυτοματοποιημένη black-box τεχνική για την ανίχνευση σφαλμάτων μηχανής καταστάσεων σε υλοποιήσεις καταστασιακών πρωτοκόλλων δικτύου προτάθηκε αργότερα από τους Fiterau-Brostean et al. [16].

Επιπλέον, έχουν παρουσιαστεί πολυάριθμες τεχνικές stateful fuzzing. Για παράδειγμα, το AFLNetLegion [17], μια επέκταση του AFLNet, παρουσιάζει έναν καινοτόμο αλγόριθμο για την επιλογή καταστάσεων. Ο SGFuzz [18] είναι ένας stateful greybox fuzzer που βασίζεται στο LibFuzzer, χρησιμοποιώντας πρόσθετη

ανατροφοδότηση για την πλοήγηση στο χώρο καταστάσεων των stateful συστημάτων λογισμικού με στόχο την αποκάλυψη stateful σφαλμάτων. Το NSFuzz [19] είναι ένα εργαλείο fuzzing σχεδιασμένο για stateful υπηρεσίες δικτύου, που χρησιμοποιεί στατική ανάλυση για τον εντοπισμό βρόχων δικτυακών συμβάντων και την εξαγωγή μεταβλητών κατάστασης, επιτυγχάνοντας έτσι γρήγορο συγχρονισμό εισόδου/εξόδου και αποτελεσματικό state-aware fuzzing μέσω lightweight instrumentation σε χρόνο μεταγλώττισης.

Έχει επίσης πραγματοποιηθεί σημαντικός όγκος έρευνας αφιερωμένος στη συστηματική αξιολόγηση και σύγκριση διαφορετικών τεχνικών fuzzing. Για παράδειγμα, οι Poncelet et al. [2] συζήτησαν τις προκλήσεις που σχετίζονται με την αξιολόγηση των fuzzers, δεδομένου του τεράστιου αριθμού των διαθέσιμων εργαλείων fuzzing και του περιορισμένου χρόνου για την αξιολόγησή τους.

Οι πρόσφατες εξελίξεις στην έρευνα με βάση το fuzzing έχουν διαδραματίσει καθοριστικό ρόλο στον εντοπισμό ευπαθειών σε υλοποιήσεις πρωτοκόλλων. Έχουν καταβληθεί προσπάθειες για τη δημιουργία μιας συστηματικής επισκόπησης στον τομέα του stateful fuzzing [20] και του protocol fuzzing [21]. Αυτές οι εργασίες επισκόπησης προσφέρουν συστηματικές συγκρίσεις και ταξινομήσεις των fuzzers, και επίσης επισημαίνουν τις προκλήσεις και τις ευκαιρίες για μελλοντική έρευνα σε αυτόν τον τομέα.

1.4 Πρωτόκολλα

Σε αυτό το κεφάλαιο στόχος μας είναι να θέσουμε τις βάσεις για την κατανόηση των περίπλοκων προτύπων και διαδικασιών που διέπουν την ασφαλή και αποτελεσματική ανταλλαγή δεδομένων στο υπό δοκιμή σύστημά μας. Αυτό το κεφάλαιο όχι μόνο θα πλαισιώσει τη σημασία κάθε πρωτοκόλλου στο ευρύτερο οικοσύστημα, αλλά και θα θέσει τις βάσεις για τη μετέπειτα εξερεύνηση των τεχνικών stateful fuzzing που εφαρμόζονται σε αυτά τα πρωτόκολλα.

1.4.1 CoAP

Το Constrained Application Protocol (CoAP) [22] είναι ένα πρωτόκολλο επικοινωνίας προσαρμοσμένο για συσκευές με περιορισμένους πόρους, που συνήθως συναντώνται στον τομέα του Διαδικτύου των Πραγμάτων (IoT). Ακολουθεί ένα μοντέλο client-server παρόμοιο με το HTTP, αλλά είναι βελτιστοποιημένο για αλληλεπιδράσεις από μηχάνημα σε μηχάνημα. Το CoAP λειτουργεί πάνω από πρωτόκολλα μεταφοράς που επικεντρώνονται στο datagram, όπως το UDP, και είναι σχεδιασμένο για ασύγχρονη επικοινωνία.

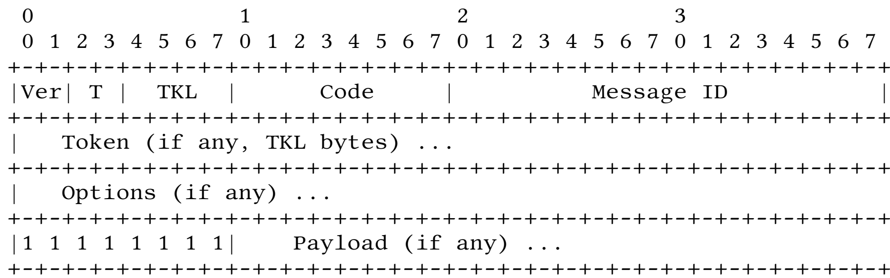
Τα μηνύματα CoAP έχουν συμπαγή δυαδική μορφή και αποτελούνται από μια επικεφαλίδα 4 byte που ακολουθείται από ένα Token μεταβλητού μήκους για συσχέτιση, μια ακολουθία επιλογών και ένα προαιρετικό payload. Τα μηνύματα μπορούν να είναι τεσσάρων τύπων: Επιβεβαιώσιμα, μη επιβεβαιώσιμα, επιβεβαίωση και επαναφορά. Τα επιβεβαιώσιμα μηνύματα υποστηρίζουν αξιοπιστία με επαναμεταδόσεις, ενώ τα μη επιβεβαιώσιμα μηνύματα δεν επαληθεύονται. Τα αιτήματα και οι αποκρίσεις μεταφέρονται σε αυτούς τους τύπους μηνυμάτων και οι αποκρίσεις μπορούν να μεταφερθούν σε μηνύματα επιβεβαίωσης.

Στο CoAP, μια απάντηση προσδιορίζεται από το πεδίο Code στην επικεφαλίδα, υποδεικνύοντας το αποτέλεσμα της κατανόησης και της εκπλήρωσης μιας αίτησης. Τα τρία ανώτερα bit του 8-bit Response Code καθορίζουν την κατηγορία απάντησης, ενώ τα πέντε κατώτερα bit παρέχουν πρόσθετες λεπτομέρειες. Οι κατανοητοί από τον άνθρωπο συμβολισμοί για τους κωδικούς CoAP έχουν τη μορφή "c.dd", όπου "c" είναι η κλάση σε δεκαδικό αριθμό και "dd" είναι οι λεπτομέρειες ως διψήφιο δεκαδικό αριθμό. Για παράδειγμα, το ¨Forbidden¨ αναπαρίσταται ως 4.03, που αντιστοιχεί στην δεκαεξαδική τιμή κωδικού 8 bit 0x83 (4*0x20+3) ή δεκαδική τιμή 131 (4*32+3).

Η μορφή του μηνύματος CoAP απεικονίζεται στο Σχήμα 1.1, και οι κωδικοί απόκρισης CoAP παρατίθενται στο Σχήμα 1.2.

1.4.2 EDHOC

Το πρωτόκολλο Ephemeral Diffie-Hellman Over COSE (EDHOC) [7] είναι ένα συμπαγές και ελαφρύ πρωτόκολλο ανταλλαγής κλειδιών που έχει σχεδιαστεί για περιβάλλοντα με υψηλούς περιορισμούς, στοχεύοντας κυρίως στην υποδομή IoT. Προσφέρει χαρακτηριστικά ασφαλείας όπως προστασία ταυτότητας, διαπραγμάτευση κρυπτογράφησης και μυστικότητα προς τα εμπρός. Το πρωτόκολλο



Σχήμα 1.1: Μορφή CoAP μηνυματος

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

Σχήμα 1.2: Κωδικοί απόκρισης του CoAP

βασίζεται στο COSE για την κρυπτογραφία, στο CBOR για την κωδικοποίηση και στο CoAP για τη μεταφορά.

Το EDHOC ορίζει δύο κύριους ρόλους: Initiator και Responder. Αυτοί οι ρόλοι δεν συνδέονται με συγκεκριμένα διαδικτυακά πρωτόκολλα μεταφοράς, επιτρέποντας ευελιξία για διάφορες εφαρμογές. Μια ανταλλαγή κλειδιών EDHOC αφορά πέντε μηνύματα, με τα *message_1*, *message_2* και *message_3* να είναι υποχρεωτικά. Το *message_4* είναι προαιρετικό και το error_message είναι διαθέσιμο και για τους δύο ρόλους. Τα μηνύματα αυτά κωδικοποιούνται με τη χρήση στοιχείων δεδομένων CBOR.

Τα αναγνωριστικά session παίζουν καθοριστικό ρόλο στο EDHOC. Κάθε μέλος επιλέγει ένα αναγνωριστικό για την αναγνώριση του τρέχοντος session. Ο Initiator επιλέγει το C_I και το στέλνει στο *message_1*, ενώ ο Responder επιλέγει το C_R και το στέλνει στο *message_2*. Αυτά τα αναγνωριστικά επιτρέπουν τη συσχέτιση μηνυμάτων και την ανάκτηση της κατάστασης του πρωτοκόλλου κατά τη διάρκεια του session και μπορούν επίσης να χρησιμοποιηθούν για σκοπούς σε επίπεδο εφαρμογής, όπως το OSCORE.

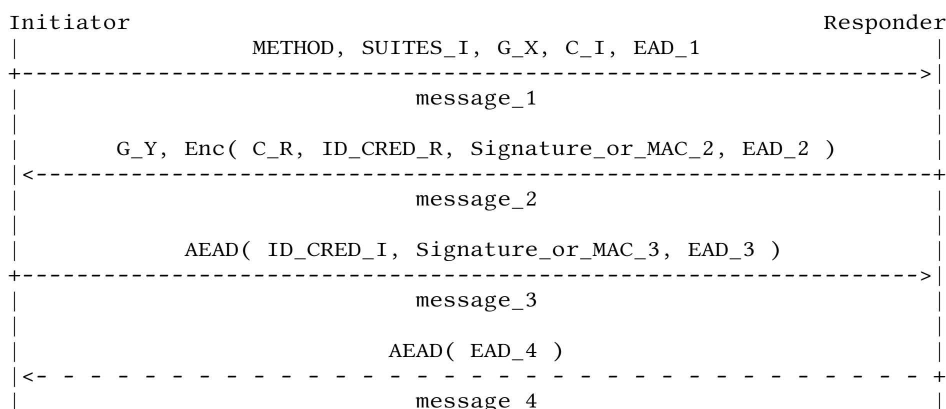
Οι παράμετροι ελέγχου ταυτότητας περιλαμβάνουν:

- Authentication Keys: Αυτά είναι τα δημόσια κλειδιά που χρησιμοποιούνται για τον έλεγχο ταυτότητας. Μπορεί να είναι είτε ένα κλειδί υπογραφής είτε ένα στατικό κλειδί Diffie-Hellman, ανάλογα με τη μέθοδο ελέγχου ταυτότητας.
- Authentication Credentials: Πρόκειται για τα δημόσια κλειδιά αυθεντικοποίησης του Initiator και του Responder, τα οποία αναφέρονται ως CRED_I και CRED_R αντίστοιχα. Χρησιμοποιούνται για την επαλήθευση της ακεραιότητας του άλλου μέλους και για την επαλήθευση της απόδειξης κατοχής του ιδιωτικού κλειδιού. Συνήθως δεν μεταφέρονται στο EDHOC, αλλά παρέχονται διαφορετικά.
- Authentication Credential Identifiers: Πρόκειται για αναγνωριστικά που χρησιμοποιούνται για την αναγνώριση των αντίστοιχων αποθηκευμένων διαπιστευτηρίων ελέγχου ταυτότητας. Είναι μικρότερα σε μέγεθος από τα διαπιστευτήρια που αναγνωρίζουν και μεταφέρονται κατά τη διάρκεια του EDHOC. Το αναγνωριστικό του Responder, ID_CRED_R, μεταφέρεται στο *message_2* και το αναγνωριστικό του Initiator, ID_CRED_I, μεταφέρεται στο *message_3*.
- External Authorization Data (EAD): Πρόκειται για εξωτερικά δεδομένα εφαρμογής που μπορούν να ενσωματωθούν σε κάθε μήνυμα του πρωτοκόλλου EDHOC. Συμπεριλαμβάνονται για να μειώσουν τις διαδρομές μετ΄ επιστροφής, τον αριθμό των ανταλλασσόμενων μηνυμάτων και να απλοποιήσουν την επεξεργασία.

Τα Ephemeral public keys (G_X και G_Y) ανταλλάσσονται στο *message_1* και στο *message_2*, και χρησιμεύουν ως πηγή τυχαιότητας για κάθε session. Αυτά τα κλειδιά είναι απαραίτητα για τη διασφάλιση της ασφάλειας της ανταλλαγής κλειδιών.

Μετά την επιτυχή επεξεργασία του *message_3*, τα μέλη μπορούν να αντλήσουν παραμέτρους του πλαισίου ασφαλείας OSCORE.

Η ροή μηνυμάτων EDHOC απεικονίζεται στο Σχήμα 1.3. Η αλληλεπίδραση μεταξύ του client και του server, με τον server να ενεργεί ως responder και τον client ως initiator, απεικονίζεται στο Σχήμα 1.4.



Σχήμα 1.3: Ροή μηνυμάτων EDHOC

```
Client    Server
  |          |
  +--------->| Header: POST (Code=0.02)
  |   POST   | Uri-Path: "/.well-known/edhoc"
  |          | Content-Format: application/cid-edhoc+cbor-seq
  |          | Payload: true, EDHOC message_1
  |          |
  |<---------+ Header: 2.04 Changed
  |   2.04   | Content-Format: application/edhoc+cbor-seq
  |          | Payload: EDHOC message_2
  |          |
  +--------->| Header: POST (Code=0.02)
  |   POST   | Uri-Path: "/.well-known/edhoc"
  |          | Content-Format: application/cid-edhoc+cbor-seq
  |          | Payload: C_R, EDHOC message_3
  |          |
  |<---------+ Header: 2.04 Changed
  |   2.04   | Content-Format: application/edhoc+cbor-seq
  |          | Payload: EDHOC message_4
  |          |
```

Σχήμα 1.4: Ροή μηνυμάτων EDHOC με τον server ως responder

1.4.3 OSCORE

Το Object Security for Constrained RESTful Environments (OSCORE) [23] είναι μια μέθοδος για την προστασία του CoAP σε επίπεδο εφαρμογής, χρησιμοποιώντας CBOR Object Signing and Encryption (COSE). Στοχεύει στην κατοχύρωση ασφάλειας από άκρο σε άκρο μεταξύ δύο τερματικών σημείων CoAP, διασφαλίζοντας ότι οι ενδιάμεσοι δεν μπορούν να αλλοιώσουν ή να αποκτήσουν πρόσβαση σε πεδία μηνυμάτων που δεν σχετίζονται με τις καθορισμένες λειτουργίες τους. Στην ουσία, το OSCORE λαμβάνει ένα μη προστατευμένο μήνυμα CoAP και το μετατρέπει σε ασφαλές, προστατεύοντας όχι μόνο το payload αλλά και τις πλήρως προστατευμένες επιλογές CoAP, τους αρχικούς κωδικούς REST της αίτησης και της απόκρισης και ορισμένα τμήματα του URI που παραπέμπουν στους πόρους των μηνυμάτων αίτησης.

Για να χρησιμοποιηθεί το πρωτόκολλο OSCORE, τα εμπλεκόμενα μέρη πρέπει να δημιουργήσουν ένα κοινό πλαίσιο ασφαλείας για την επεξεργασία των αντικειμένων COSE. Αυτό απαιτεί την ασφαλή και πιστοποιημένη ανταλλαγή βασικών πληροφοριών και υλικού κλειδιών, που παρέχεται από ένα κατάλληλο πρωτόκολλο ανταλλαγής κλειδιών.

1.5 Προετοιμασία Πειραμάτων

1.5.1 Συστήματα που χρησιμοποιήθηκαν

uOSCORE-uEDHOC Το υπό δοκιμή σύστημα που χρησιμοποιούμε είναι μια υλοποίηση σε γλώσσα C των πρωτοκόλλων IETF OSCORE [23] και EDHOC [7] που ονομάζεται uOSCORE-uEDHOC. Πιο συγκεκριμένα, χρησιμοποιούμε ένα από τα παρεχόμενα δείγματα που υλοποιεί έναν responder server που χρησιμοποιεί EDHOC και OSCORE.

Αυτό το SUT χρησιμοποιεί τυχαία παραγόμενα κλειδιά, καθιστώντας το ακατάλληλο για fuzzing. Η τυχαιότητα είναι κακή για το fuzzing λόγω της αδυναμίας αναπαραγωγής των ίδιων αποτελεσμάτων με τα ίδια seeds. Εάν αναπαράγουμε μια έγκυρη ακολουθία αιτημάτων client, για παράδειγμα, μπορεί να αποτύχει να περάσει τον έλεγχο εγκυρότητας του server, επειδή κάθε instance του server είναι τυχαιοποιημένο. Οι απαντήσεις του server είναι διαφορετικές μεταξύ δύο εκτελέσεων και ο server αναμένει διαφορετικά αιτήματα. Αν θέλουμε τα ίδια αιτήματα να προκαλούν την ίδια συμπεριφορά στον server, πρέπει να απο-τυχαιοποιήσουμε το SUT. Με αυτόν τον τρόπο ελπίζουμε ότι θα μπορούμε επίσης να αναπαράγουμε όποια crashes και hangs βρούμε.

Το SUT βασίζεται σε εξωτερικές βιβλιοθήκες για την κρυπτογράφηση. Η κρυπτογραφική βιβλιοθήκη Mbed TLS είναι η κύρια πηγή τυχαιότητας. Οι υλοποιήσεις πρωτοκόλλων βασίζονται σε τέτοιες πηγές τυχαιότητας προκειμένου να είναι ασφαλείς. Όμως, στην περίπτωσή μας χρειαζόμαστε ντετερμινιστική λειτουργία. Έτσι, αφαιρέσαμε την τυχαία πηγή, αντικαθιστώντας μια κλήση συστήματος που χρησιμοποιείται για να γεμίσει ένα buffer με τυχαία δεδομένα, γεμίζοντας το buffer με προκαθορισμένα δεδομένα αντ' αυτού.

EDHOC-Fuzzer Ο EDHOC-Fuzzer [15] είναι ένα εργαλείο σχεδιασμένο για την ανάλυση των υλοποιήσεων του πρωτοκόλλου EDHOC. Αυτό το εργαλείο χρησιμοποιεί protocol state fuzzing για να δημιουργήσει μια καλή προσέγγιση του υποκείμενου μοντέλου μηχανής καταστάσεων μιας υλοποίησης EDHOC.

Ο EDHOC-Fuzzer έχει τη δυνατότητα να στέλνει μεμονωμένες abstract ακολουθίες εισόδου στο SUT. Επεκτάθηκε επίσης με μια επιλογή εξαγωγής των αιτημάτων ακριβώς πριν από την αποστολή τους στο SUT. Εάν αυτές οι επιλογές χρησιμοποιηθούν μαζί, το εργαλείο μπορεί να λειτουργήσει ως concretizer, μετατρέποντας τις abstract ακολουθίες εισόδου σε concrete που μπορούν να χρησιμοποιηθούν ως seeds με τον AFLML ή άλλους fuzzers.

AFLNet Δεδομένου ότι ο AFLNet [5] είναι για συγκεκριμένα πρωτόκολλα, έπρεπε να τον επεκτείνουμε για την υποστήριξη EDHOC. Προκειμένου να προσθέσουμε υποστήριξη για ένα άλλο πρωτόκολλο έπρεπε να υλοποιήσουμε δύο συναρτήσεις, μία για να διαχωρίσουμε τα συγκολλημένα αιτήματα και μία για να εξάγουμε τους κωδικούς απόκρισης από τις συγκολλημένες απαντήσεις. Αυτό συνήθως γίνεται είτε με την εύρεση ενός διαχωριστικού που είναι συγκεκριμένο για κάθε πρωτόκολλο είτε με την εύρεση του μεγέθους του μηνύματος από την επικεφαλίδα του. Όμως, αυτό ήταν αδύνατο για το EDHOC, αφού δεν υποστηρίζει κανένα

Πίνακας 1.1: Επιλογές που χρησιμοποιήθηκαν για τον AFLNet

Επιλογή	Περιγραφή
-P COAP	Χρήση του COAP ως πρωτοκόλλο εφαρμογής προς δοκιμή
-m none	Απενεργοποίηση ορίου μνήμης
-D 50000	Ορισμός χρόνου αναμονής αρχικοποίησης server σε 50000 μs
-q 3	Ορισμός αλγορίθμου επιλογής κατάστασης FAVOR
-s 3	Ορισμός αλγορίθμου επιλογής seed FAVOR
-R	Ενεργοποίηση τελεστών μετάλλαξης region-level
-E	Ενεργοποίηση του state-aware mode
-K	Τερματισμός του server μετά την κατανάλωση όλων των μηνυμάτων
-W 50	Ορισμός του χρόνου αναμονής απόκρισης σε 50 ms

από τα δύο. Έτσι, έπρεπε να τροποποιήσουμε τον AFLNet ώστε να χειρίζεται τα μηνύματα με άλλον τρόπο.

Αρχικά, ο AFLNet χρησιμοποιεί raw αρχεία (συγκολλημένα μηνύματα), ως seeds, ενώ ο StateAFL δέχεται αρχεία seed εισόδου σε replayable μορφή. Αυτή η μορφή έχει αποθηκευμένο πριν από κάθε μήνυμα το μέγεθός του. Επιπλέον, ο StateAFL διαθέτει μια καθολική συνάρτηση για τον διαχωρισμό των αποθηκευμένων αιτημάτων, δεδομένου ότι η replayable μορφή εξαλείφει την ανάγκη ειδικής υλοποίησης για κάθε πρωτόκολλο. Δανειστήκαμε οπότε κάποιο κώδικα από τον StateAFL για αυτά τα χαρακτηριστικά, καθιστώντας έτσι τον AFLNet συμβατό με replayable είσοδο.

Αλλάξαμε επίσης τον τρόπο αποθήκευσης των απαντήσεων. Όπως και στα αιτήματα, αποθηκεύουμε πρώτα το μέγεθός τους, επιτρέποντας την πρόσβαση σε καθεμία από αυτές μέσα σε ένα buffer με αποθηκευμένες απαντήσεις.

Ο κωδικός απόκρισης των μηνυμάτων CoAP αποθηκεύεται στο δεύτερο byte. Με το πρόβλημα του διαχωρισμού λυμένο, μπορούμε εύκολα να εξάγουμε τους κωδικούς απόκρισης.

Τροφοδοτούμε το AFLNet με ένα μόνο seed που περιέχει τα μηνύματα EDHOC message_1 και message_3 ακολουθούμενα από ένα OSCORE application message. Αυτή η ακολουθία αντιστοιχεί σε μια ανταλλαγή κλειδιών ακολουθούμενη από μια κρυπτογραφημένη ανταλλαγή μηνυμάτων και έχει γίνει concretized με τη χρήση του EDHOC-Fuzzer.

Οι επιλογές που χρησιμοποιήθηκαν φαίνονται στον πίνακα 1.1.

StateAFL Τροφοδοτούμε τον StateAFL [6] με ένα μόνο seed που περιέχει τα μηνύματα message_1 και message_3 ακολουθούμενα από ένα OSCORE application message, όπως ακριβώς και τον AFLNet. Χάρη στην ευελιξία του, δεν χρειάστηκαν τροποποιήσεις.

Οι επιλογές που χρησιμοποιήθηκαν φαίνονται στον πίνακα 1.2.

AFLML Για να δημιουργήσουμε τα seeds για τον AFLML μαθαίνουμε πρώτα τη μηχανή καταστάσεων με τον EDHOC-Fuzzer. Στη συνέχεια, χρησιμοποιώντας

Πίνακας 1.2: Επιλογές που χρησιμοποιήθηκαν για τον StateAFL

Επιλογή	**Περιγραφή**
-D 50000	Ορισμός χρόνου αναμονής αρχικοποίησης server σε 50000 μs
-q 3	Ορισμός αλγορίθμου επιλογής κατάστασης FAVOR
-s 3	Ορισμός αλγορίθμου επιλογής seed FAVOR
-R	Ενεργοποίηση τελεστών μετάλλαξης region-level
-E	Ενεργοποίηση του state-aware mode
-K	Τερματισμός του server μετά την κατανάλωση όλων των μηνυμάτων
-W 50	Ορισμός του χρόνου αναμονής απόκρισης σε 50 ms

Πίνακας 1.3: Επιλογές που χρησιμοποιήθηκαν για τον AFLML

Επιλογή	**Περιγραφή**
-m none	Απενεργοποίηση ορίου μνήμης
-D 50000	Ορισμός χρόνου αναμονής αρχικοποίησης server σε 50000 μs
-q 3	Ορισμός αλγορίθμου επιλογής κατάστασης FAVOR
-s 3	Ορισμός αλγορίθμου επιλογής seed FAVOR
-R	Ενεργοποίηση τελεστών μετάλλαξης region-level
-E	Ενεργοποίηση του state-aware mode
-K	Τερματισμός του server μετά την κατανάλωση όλων των μηνυμάτων
-W 50	Ορισμός του χρόνου αναμονής απόκρισης σε 50 ms

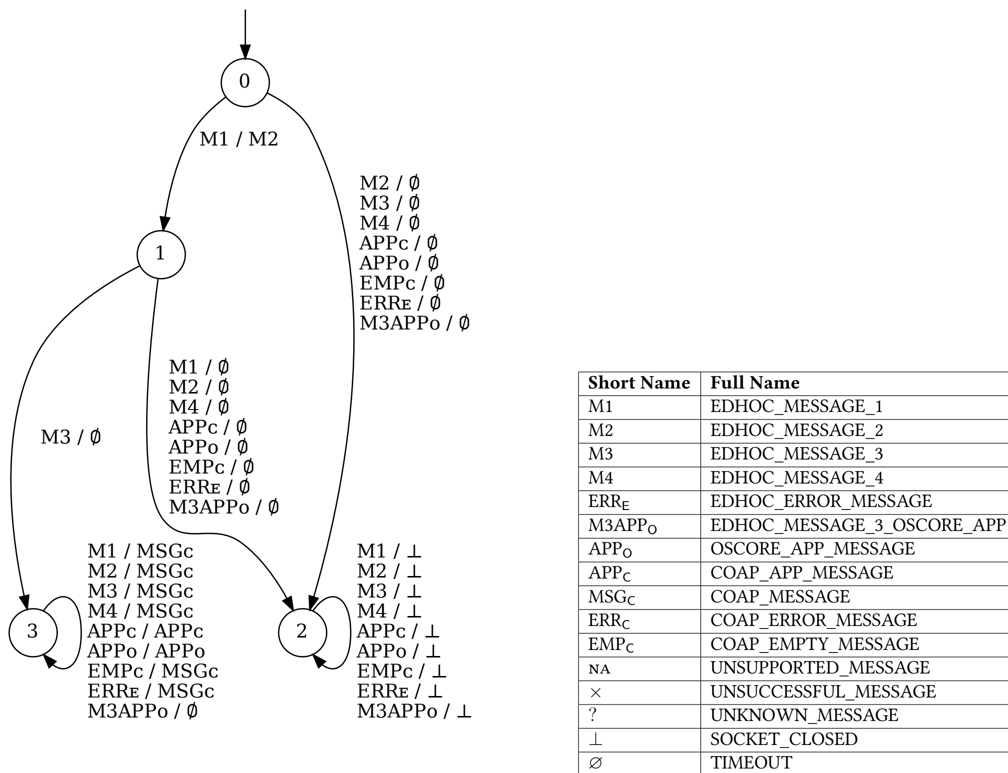
ένα script, εξερευνούμε τα πιθανά μονοπάτια μέσα στην μηχανή καταστάσεων και δημιουργούμε abstract seed αρχεία με βάση αυτά. Στη συνέχεια, κάνουμε concretize αυτά τα seeds με τον EDHOC-Fuzzer και τα χρησιμοποιούμε ως είσοδο.

Οι επιλογές που χρησιμοποιήθηκαν φαίνονται στον πίνακα 1.3.

1.5.2 Ρυθμίσεις

Επανάληψη πειραμάτων Παρά τις προσπάθειές μας να αφαιρέσουμε την τυχαίοτητα του SUT, μπορεί να υπάρχει ακόμα κάποια τυχαιότητα. Επίσης, οι fuzzers είναι τυχαιοποιημένοι. Κατά συνέπεια, δύο εκτελέσεις του ίδιου fuzzer μπορεί να μην δίνουν ακριβώς τα ίδια αποτελέσματα. Για να ελαχιστοποιήσουμε την επίδραση της τυχαιότητας, πραγματοποιήσαμε δέκα ανεξάρτητα πειράματα για κάθε fuzzer. Κάθε πείραμα εκτελείται σε ένα απομονωμένο Docker container χρησιμοποιώντας τα scripts εκτέλεσης του ProFuzzBench [24].

Πλατφόρμα Όλα τα πειράματα διεξήχθησαν σε server με τέσσερις επεξεργαστές Intel(R) Xeon(R) E5-4650 @ 2.70GHz και 128 GB RAM με Debian GNU/Linux 12.4.



Short Name	Full Name
M1	EDHOC_MESSAGE_1
M2	EDHOC_MESSAGE_2
M3	EDHOC_MESSAGE_3
M4	EDHOC_MESSAGE_4
ERR_E	EDHOC_ERROR_MESSAGE
$M3APP_O$	EDHOC_MESSAGE_3_OSCORE_APP
APP_O	OSCORE_APP_MESSAGE
APP_C	COAP_APP_MESSAGE
MSG_C	COAP_MESSAGE
ERR_C	COAP_ERROR_MESSAGE
EMP_C	COAP_EMPTY_MESSAGE
NA	UNSUPPORTED_MESSAGE
×	UNSUCCESSFUL_MESSAGE
?	UNKNOWN_MESSAGE
⊥	SOCKET_CLOSED
∅	TIMEOUT

Σχήμα 1.5: Μοντέλο μηχανής καταστάσεων από τον EDHOC-Fuzzer.

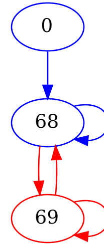
1.6 Αποτελέσματα

Χρησιμοποιώντας τη διάταξη που περιγράψαμε, τρέξαμε κάθε fuzzer για 24 ώρες. Συγκρίνουμε τους fuzzers ως προς την ακρίβεια στη παραγωγή μηχανών καταστάσεων, την κάλυψη ακμών και γραμμών του προγράμματος και τον αριθμό των crashes που βρέθηκαν.

1.6.1 Σύγκριση μηχανών καταστάσεων

Αρχικά, συγκρίνουμε τα τρία διαφορετικά μοντέλα μηχανών καταστάσεων του SUT: το μοντέλο από τον EDHOC-Fuzzer (Σχήμα 1.5), το μοντέλο από τον AFLNet (Σχήμα 1.6) και το μοντέλο από τον StateAFL (Σχήμα 1.7).

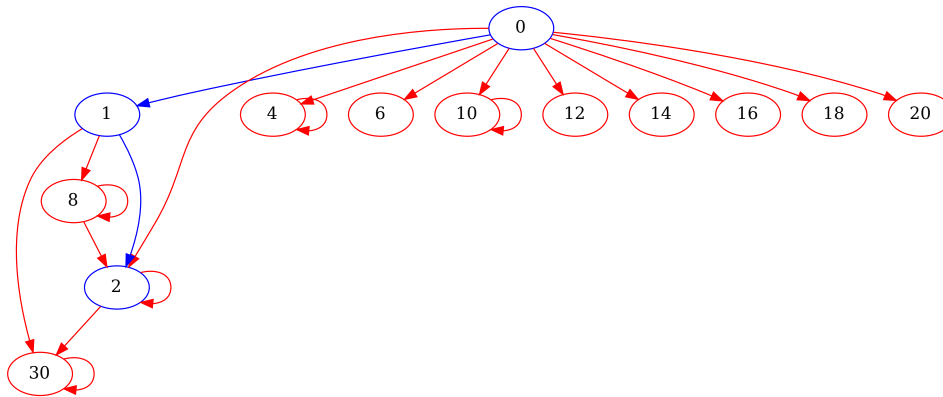
Δεδομένου ότι χρησιμοποιούμε Active Automata Learning (AAL) για την κατασκευή του μοντέλου μηχανής καταστάσεων για τον AFLML με τον EDHOC-Fuzzer, αυτό είναι προφανώς το πιο ακριβές μοντέλο και θα το χρησιμοποιήσουμε ως αναφορά. Παρουσιάζεται ως μηχανή Mealy (δηλαδή οι ακμές του είναι επισημασμένες με ετικέτες της μορφής I/O που δείχνουν τις σχέσεις μεταξύ εισόδων και εξόδων) και περιέχει συνολικά τέσσερις καταστάσεις (με ετικέτες 0 έως 3). Οι άλλοι fuzzers υποδεικνύουν τις καταστάσεις και τις ακμές που ανακαλύφθηκαν από τα αρχικά seeds με μπλε χρώμα και τις υπόλοιπες με κόκκινο. Η κατάσταση 0 αντιπροσωπεύει την αρχική κατάσταση σε όλα τα μοντέλα.



Σχήμα 1.6: Μοντέλο μηχανής καταστάσεων από τον AFLNet.

Ο AFLNet δημιουργεί μια κατάσταση για κάθε κωδικό απόκρισης. Η μηχανή καταστάσεών του έχει μόνο τρεις καταστάσεις, με τις ετικέτες 0, 68 και 69. Παρατηρήσαμε ότι τα M2 και APP_O έχουν κωδικό απόκρισης 68 (Changed) και το MSG_C έχει κωδικό απόκρισης 69 (Content). Έτσι, μπορούμε απευθείας να υποθέσουμε ότι η κατάσταση 68 αντιπροσωπεύει την κατάσταση 1 στο μοντέλο AAL, επειδή η κατάσταση 1 επιστρέφει M2, και η κατάσταση 69 αντιπροσωπεύει την κατάσταση 3, επειδή η κατάσταση 3 επιστρέφει MSG_C. Το πρώτο μήνυμα στο αρχικό seed, το M1, ανακαλύπτει την ακμή από το 0 στο 68. Έτσι μπορούμε να επαληθεύσουμε ότι η κατάσταση 68 αντιπροσωπεύει την κατάσταση 1 στο μοντέλο AAL. Το επόμενο μήνυμα στο αρχικό seed είναι το M3 και θα περιμέναμε να ανακαλύψει μια κατάσταση παρόμοια με την κατάσταση 3 στο μοντέλο AAL. Όμως, δεν κάνει κάτι τέτοιο ακόμα, επειδή ο AFLNet βασίζεται σε κωδικούς απόκρισης για την ανακάλυψη καταστάσεων και η ακολουθία M1-M3 οδηγεί σε timeout μετά την αποστολή του M3. Αντ' αυτού, μια τέτοια κατάσταση, η κατάσταση 69, μπορεί να ανακαλυφθεί αργότερα με την ακολουθία M1-M3-M3 ή M1-M3-M1 χρησιμοποιώντας μια μετάλλαξη η οποία προσθέτει τμήματα της υπάρχουσας εισόδου. Το τελευταίο μήνυμα στο αρχικό seed είναι το APP_O και ανακαλύπτει την ακμή 68-68. Τέτοια ακμή δεν υπάρχει στο μοντέλο AAL και η δημιουργία της είναι αποτέλεσμα του ότι τα M2 και APP_O έχουν τον ίδιο κωδικό απόκρισης και ο AFLNet δεν έχει ανακαλύψει ακόμα την κατάσταση 69. Η ακμή 69-68 είναι επίσης αποτέλεσμα του πρώτου. Η ακμή 69-69 μπορεί να ανακαλυφθεί χρησιμοποιώντας μία από τις παρακάτω ακολουθίες: M1-M3-M3-M3, M1-M3-M3-M1, M1-M3-M1-M3, M1-M3-M1-M3, M1-M3-M1-M1. Ο AFLNet αδυνατεί επίσης να ανακαλύψει την κατάσταση 2, όπως και την κατάσταση 1, επειδή δεν υπάρχει κατάλληλη ακολουθία που να επιστρέφει κάποιο κωδικό απόκρισης. Για παράδειγμα, η ακολουθία M1-M1-M1-M1 οδηγεί σε timeout ακολουθούμενο από τερματισμό λειτουργίας του server.

Σε αντίθεση με τον AFLNet, ο StateAFL είναι λιγότερο σταθερός με τη δημιουργία της μηχανής καταστάσεων, με αποτέλεσμα να προκύπτουν διάφορα μοντέλα μεταξύ των εκτελέσεων, τα οποία όμως μοιράζονται κάποιες ομοιότητες. Το πρώτο μήνυμα στο αρχικό seed, M1, ανακαλύπτει την ακμή από την κατάσταση 0 στην κατάσταση 1. Το επόμενο μήνυμα στο αρχικό seed, M3, δεν ανακαλύπτει νέα κατάσταση, πιθανόν επειδή το περιεχόμενο της μακροχρόνιας μνήμης πριν και μετά τη μετάβαση από την κατάσταση 1 στην κατάσταση 3 στο μοντέλο AAL είναι παρόμοιο. Αντ' αυτού, το τελευταίο μήνυμα στο αρχικό seed, APP_O, ανακαλύπτει



Σχήμα 1.7: Μοντέλο μηχανής καταστάσεων από τον StateAFL (εν μέρει).

την κατάσταση 2, η οποία μοιάζει με την κατάσταση 3 στο μοντέλο AAL. Αν και η αρχική μηχανή καταστάσεων του StateAFL μπορεί να θεωρηθεί βελτίωση της αρχικής μηχανής καταστάσεων του AFLNet, ο StateAFL αντιμετωπίζει αργότερα το πρόβλημα ότι δημιουργεί διπλές καταστάσεις. Υπάρχουν πολλές δυσερμήνευτες καταστάσεις με μία εισερχόμενη μετάβαση από την κατάσταση 0, την αρχική κατάσταση, και καμία εξερχόμενη μετάβαση. Παρόλο που μπορούμε να δούμε ότι ορισμένα τμήματα της μηχανής καταστάσεων StateAFL μοιάζουν με το μοντέλο AAL, στις περισσότερες εκτελέσεις υπάρχουν παραπλανητικές καταστάσεις και μεταβάσεις. Συνοψίζοντας, ο StateAFL αποτυγχάνει τόσο να διαχωρίσει διαφορετικές καταστάσεις όσο και να δημιουργήσει καταστάσεις που έχουν όλες νόημα.

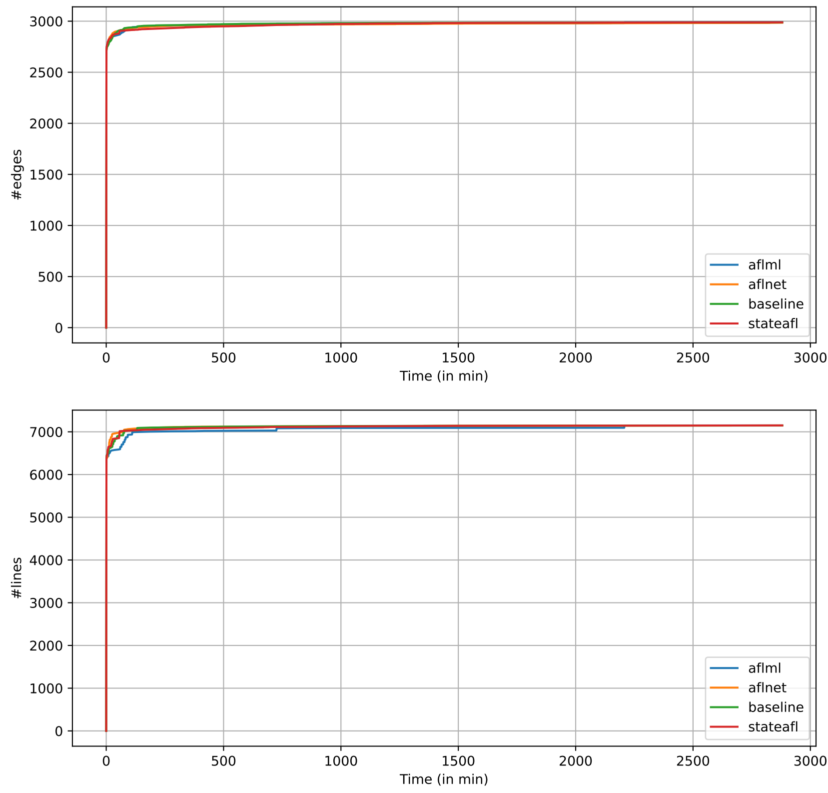
1.6.2 Σύγκριση κάλυψης κώδικα

Η κάλυψη ακμών και γραμμών κώδικα μετρήθηκε με την πάροδο του χρόνου για καθέναν από τους τρεις fuzzers χρησιμοποιώντας τα scripts ανάλυσης του ProFuzzBench. Ο AFLNet αξιολογήθηκε δύο φορές, μία φορά χωρίς τις σημαίες `-E` και `-q`, λειτουργώντας ως baseline για την αξιολόγηση της επίδρασης της επίγνωσης κατάστασης.

Τα γραφήματα κάλυψης ακμών και γραμμών απεικονίζονται στο Σχήμα 1.8. Η ανάλυσή τους δείχνει ότι όλοι οι fuzzers επιτυγχάνουν σχεδόν μέγιστη κάλυψη μέσα στις πρώτες δύο ώρες, μετά από τις οποίες ο ρυθμός αύξησης της κάλυψης μειώνεται. Η διαφορά στις επιδόσεις τους όσον αφορά την κάλυψη είναι ασήμαντη.

1.6.3 Σύγκριση των Crashes

Οι fuzzers που βασίζονται στον AFL, όπως αυτοί που αξιολογήθηκαν στη σύγκρισή μας, κατηγοριοποιούν τα crashes ως “unique” εάν παρουσιάζουν διαφορετικά προφίλ κάλυψης. Για να διασφαλίσουμε την εγκυρότητα αυτών των crashes, χρησιμοποιούμε ένα script που φιλτράρει τα ψευδώς θετικά αποτελέσματα εκτελώντας εκ νέου τα replayable crashes και παρακολουθώντας την έξοδο από το



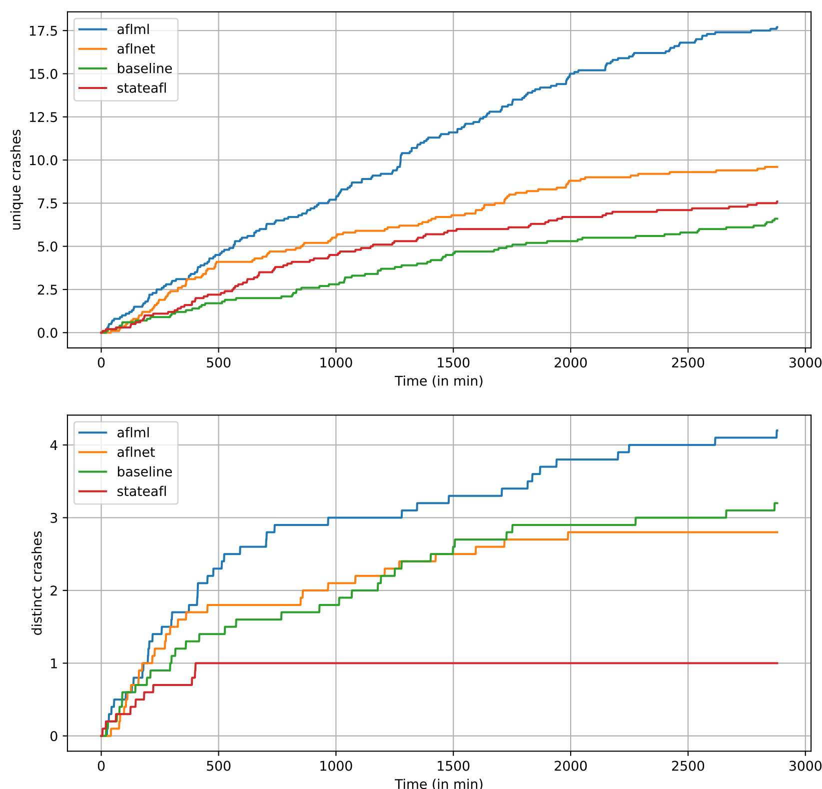
Σχήμα 1.8: Κάλυψη κώδικα των διαφόρων fuzzers με την πάροδο του χρόνου.

```
SUMMARY: AddressSanitizer: SEGV (/lib/x86_64-linux-gnu/libc.so.6+0xbbaeb)
```

Σχήμα 1.9: AddressSanitizer summary

instrumented SUT. Το SUT που χρησιμοποιείται για αυτή την επαλήθευση είναι το ίδιο που χρησιμοποιήσαμε κατά τη διαδικασία fuzzing και έχει μεταγλωττιστεί με τον AddressSanitizer (ASAN), ο οποίος βοηθά στον εντοπισμό σφαλμάτων μνήμης. Αναλύουμε το summary του AddressSanitizer, το οποίο περιέχει λεπτομέρειες σχετικά με το πού και πώς κράσαρε το πρόγραμμα, ως ένδειξη της ποικιλομορφίας των crashes. Στο Σχήμα 1.9 παρέχεται ένα παράδειγμα ενός AddressSanitizer summary. Αναφερόμαστε σε crashes με διαφορετικά summaries ως "*distinct*". Τα distinct crashes έχουν μεγαλύτερη πιθανότητα να σχετίζονται με διαφορετικά bugs σε σύγκριση με τα "unique" crashes. Ο μέσος όρος (από 10 πειράματα) των "unique" και "distinct" crashes παρουσιάζεται στο Σχήμα 1.10. Το AFLNet χωρίς τις σημαίες `-E` και `-q` χρησιμεύει ως baseline.

Ο AFLML επιδεικνύει ανώτερες επιδόσεις όσον αφορά την ικανότητα ανακάλυψης των περισσότερων unique crashes. Η διαφορά γίνεται αρκετά σαφής ύστερα από τις πρώτες δέκα ώρες, μετά τις οποίες παρατηρείται σταδιακή μείωση της απόδοσης των άλλων fuzzers. Ανάμεσα στους υπόλοιπους fuzzers, ο AFLNet



Σχήμα 1.10: Μέσος αριθμός unique και distinct crashes που προκλήθηκαν από διαφορετικούς fuzzers με την πάροδο του χρόνου.

έχει τις καλύτερες επιδόσεις και ο baseline τις χειρότερες. Ωστόσο, η διαφορά απόδοσης μεταξύ AFLNet, StateAFL και baseline είναι σχετικά μικρή.

Τις πρώτες τρεις ώρες, οι baseline, AFLNet και AFLML επιδεικνύουν συγκρίσιμες επιδόσεις στην ανακάλυψη distinct crashes. Ωστόσο, οι επιδόσεις των baseline και AFLNet μειώνονται σταδιακά με την πάροδο του χρόνου και τελικά το AFLML τους ξεπερνά. Όσον αφορά την ποικιλομορφία των crashes, το baseline και το AFLNet παρουσιάζουν παρόμοιες επιδόσεις. Αντίθετα, το StateAFL υστερεί σημαντικά, καταφέρνοντας να ανακαλύψει μόνο ένα distinct crash. Αυτό θα μπορούσε να αποδοθεί στην παραπλανητική μηχανή καταστάσεων που δημιουργεί.

Συνολικά, εντοπίσαμε δέκα distinct crashes. Πέντε ήταν ανιχνεύσιμα μόνο με τον AFLML, ενώ τρία από αυτά ανιχνεύθηκαν μόνο μία φορά. Ένα distinct crash όμως, το οποίο ανιχνεύθηκε σε ορισμένες εκτελέσεις από τον AFLNet και τον baseline, δεν ήταν ανιχνεύσιμο από τον AFLML. Επίσης, ο AFLNet δεν μπορούσε να εντοπίσει ένα distinct crash, το οποίο ήταν ανιχνεύσιμο από τον baseline. Μόνο ένα distinct crash ήταν ανιχνεύσιμο από τον StateAFL.

1.7 Συμπεράσματα

Η παρούσα εργασία διερεύνησε και αξιολόγησε πειραματικά ορισμένες εκδοχές του stateful fuzzing, μιας εξειδικευμένης μορφής fuzz testing που λαμβάνει υπόψη την κατάσταση του συστήματος που εξετάζεται. Η μελέτη επικεντρώθηκε σε μια νέα προσέγγιση όπου το υποκείμενο μοντέλο μηχανής κατάστασης του υπό δοκιμή συστήματος μαθαίνεται πριν από την πραγματική διαδικασία fuzzing, αντί κατά τη διάρκειά της.

Τα αποτελέσματα έδειξαν ότι η εκμάθηση του υποκείμενου μοντέλου μηχανής κατάστασης του υπό δοκιμή συστήματος πριν από την πραγματική διαδικασία fuzzing μπορεί να οδηγήσει σε πιο ακριβές και αποτελεσματικό fuzz testing και να βελτιώσει την απόδοση του fuzzing. Ωστόσο, διαπιστώθηκε επίσης ότι κάθε fuzzer έχει συμβιβασμούς και ότι υπάρχει η κατάλληλη για τον καθένα περίπτωση χρήσης. Για παράδειγμα, ο StateAFL δεν απαιτεί καμία τροποποίηση και λειτουργεί απευθείας, ενώ ο AFLNet δίνει ελαφρώς καλύτερα αποτελέσματα με ελάχιστη τροποποίηση. Ο AFLML δίνει τα καλύτερα αποτελέσματα, αλλά το στήσιμό του απαιτεί ένα τμήμα εκμάθησης μοντέλων που είναι ειδικό για κάθε πρωτόκολλο.

Εντοπίσαμε επίσης συγκεκριμένα σημεία προς βελτίωση για ορισμένους fuzzers. Για παράδειγμα, ο StateAFL κατασκευάζει μια μηχανή καταστάσεων που δεν είναι απολύτως ακριβής, γεγονός που με τη σειρά του επηρεάζει αρνητικά την απόδοσή του. Από την άλλη πλευρά, ο AFLNet δυσκολεύεται με τον χειρισμό των timeouts, γεγονός που υποδηλώνει ότι θα ήταν ίσως επωφελές να αντιμετωπίζει τα timeouts ως κωδικούς απόκρισης. Αυτές οι παρατηρήσεις υπογραμμίζουν την πολυπλοκότητα που συνεπάγεται ο σχεδιασμός fuzzers που επιτυγχάνουν ισορροπία μεταξύ υψηλής απόδοσης και ευκολίας χρήσης.

Επιπλέον, καταλήγουμε στο συμπέρασμα ότι πρέπει να γίνουν περισσότερες δοκιμές για να αξιολογηθεί η επίδραση των stateful fuzzers. Η αξιολόγηση των εργαλείων fuzz testing με ακρίβεια και συνέπεια αποτελεί μεγάλη πρόκληση, ιδίως αν ληφθεί υπόψη η συνεχής εξέλιξη των τεχνικών και η ενσωμάτωση όλο και πιο σύνθετων ρυθμίσεων χαρακτηριστικών σε αυτά τα εργαλεία.

Καθώς τα συστήματα λογισμικού συνεχίζουν να εξελίσσονται και να γίνονται πιο πολύπλοκα, ο ρόλος του fuzzing στη διατήρηση της αξιοπιστίας και της ασφάλειας του λογισμικού θα γίνεται όλο και πιο κρίσιμος. Η παρούσα μελέτη έχει συνεισφέρει (σε μικρό βαθμό) στον τομέα του fuzz testing. Ελπίζουμε ότι η μελλοντική έρευνα θα συνεχίσει να αξιοποιεί αυτά τα ευρήματα για την περαιτέρω βελτίωση της αποτελεσματικότητας και της αποδοτικότητας του fuzz testing.

1.8 Μελλοντική έρευνα

Το stateful fuzzing είναι ένα εξελισσόμενο πεδίο με πολλές ευκαιρίες για μελλοντική έρευνα. Τα ακόλουθα σημεία περιγράφουν πιθανές κατευθύνσεις για την πρόοδο του πεδίου:

- **Εργαλεία φιλικά προς τον χρήστη**: Η ανάπτυξη φιλικών προς το χρήστη διεπαφών και documentation για τα εργαλεία stateful fuzzing μπορεί να διευκολύνει την είσοδο νέων χρηστών. Οι προσπάθειες για τη δημιουργία πιο προσιτών εργαλείων θα βοηθήσουν στην ευρύτερη υιοθέτηση πρακτικών stateful fuzzing.
- **Fuzzing as a Service**: Η διερεύνηση της έννοιας του ¨Fuzzing as a Service¨ θα μπορούσε να καταστήσει τις προηγμένες τεχνικές fuzzing πιο προσιτές σε ένα ευρύτερο κοινό. Η ανάπτυξη πλατφορμών βασισμένων στο cloud που προσφέρουν δυνατότητες stateful fuzzing θα μπορούσε να διευκολύνει τους προγραμματιστές να υιοθετήσουν αυτές τις μεθόδους χωρίς την ανάγκη εξειδικευμένης τεχνογνωσίας.
- **Βελτιστοποίηση των στρατηγικών Fuzzing**: Η διερεύνηση διαφόρων στρατηγικών fuzzing και της αποτελεσματικότητάς τους σε διαφορετικά πλαίσια μπορεί να οδηγήσει στην ανάπτυξη πιο αποτελεσματικών διαδικασιών fuzzing. Αυτό περιλαμβάνει τη μελέτη προσαρμοζόμενων τεχνικών fuzzing που μπορούν να προσαρμόζονται δυναμικά με βάση την ανατροφοδότηση από το υπό δοκιμή σύστημα.
- **Ενσωμάτωση με άλλες τεχνικές ελέγχου**: Ο συνδυασμός του stateful fuzzing με άλλες μεθοδολογίες ελέγχου, όπως ο έλεγχος πρωτοκόλλων με χρήση συμβολικής εκτέλεσης (π.χ., [25, 26]) ή στατικής ανάλυσης, θα μπορούσε να αποφέρει ενδιαφέροντα αποτελέσματα. Η έρευνα σχετικά με τους πιο αποτελεσματικούς τρόπους ενσωμάτωσης αυτών των τεχνικών θα ήταν επωφελής.
- **Σχεδιασμός ανεξάρτητος από πρωτόκολλα**: Για να ενισχυθεί η ευελιξία του stateful fuzzing, η μελλοντική έρευνα θα μπορούσε να επικεντρωθεί στη δημιουργία protocol-agnostic protocol state fuzzers. Αυτοί οι fuzzers θα πρέπει να είναι ικανοί να χειρίζονται ένα ευρύ φάσμα πρωτοκόλλων με ελάχιστες ρυθμίσεις. Ένας τέτοιος σχεδιασμός θα απλοποιούσε τη διαδικασία προσαρμογής του fuzzer σε νέα ή custom πρωτόκολλα.
- **Βελτιωμένη εκμάθηση μηχανών καταστάσεων**: Μελλοντικές εργασίες θα μπορούσαν να επικεντρωθούν στη βελτίωση των αλγορίθμων που χρησιμοποιούνται για την εκμάθηση μηχανών κατάστασης πριν από το fuzzing. Αυτό περιλαμβάνει την ανάπτυξη τεχνικών που μπορούν να συλλάβουν με μεγαλύτερη ακρίβεια πολύπλοκες συμπεριφορές και καταστάσεις εφαρμογών, καθώς και μεθόδους για τη μείωση του χρόνου εκμάθησης χωρίς να μειώνεται η ποιότητα του μοντέλου μηχανής καταστάσεων που μαθαίνεται.
- **Η μηχανική μάθηση (ML) και η τεχνητή νοημοσύνη (AI) στο Fuzzing**: Η αξιοποίηση των εξελίξεων στις τεχνολογίες ML και AI για τη βελτίωση του stateful fuzzing θα μπορούσε να αποτελέσει σημαντικό βήμα προόδου. Αυτό θα μπορούσε να περιλαμβάνει τη χρήση AI για την πρόβλεψη των πιο πιθανών καταστάσεων στις οποίες μπορεί να υπάρχουν ευπάθειες ή για τη βελτιστοποίηση της ίδιας της διαδικασίας fuzzing.

Με την αντιμετώπιση αυτών των ζητημάτων, ο τομέας του stateful fuzzing μπορεί να συνεχίσει να αναπτύσσεται και να συμβάλλει στην ενίσχυση της ασφάλειας

λογισμικού. Ο απώτερος στόχος είναι η δημιουργία ενός ασφαλέστερου ψηφιακού περιβάλλοντος μέσω του συστηματικού εντοπισμού και μετριασμού των πιθανών ευπαθειών.

Chapter 2

Introduction

Software testing is an essential activity for ensuring the quality and security of software systems. However, manual testing can be time-consuming, expensive and error-prone. Therefore, automated testing techniques have been developed to reduce the human effort which is required and increase the effectiveness of testing. One of the most popular and successful automated testing techniques is *fuzz testing* (aka *fuzzing*) [1], which involves providing random or semi-random inputs to the System Under Test (SUT) and observing its behavior. Fuzzing can uncover crashes and security vulnerabilities that may not be easily detectable by other testing methods. In recent years, it has been shown to be especially useful for testing complex systems where security is crucial, such as network protocol implementations [2, 3].

However, fuzzing is not a one-size-fits-all solution. The effectiveness of fuzzing techniques varies depending on the type of system being tested. Some systems have intricate internal states that change based on the messages they receive and send. In this thesis, we focus on *stateful fuzzing*, which takes into account the system's state and it can test systems with different phases and transitions more effectively than stateless fuzzing.

One of the challenges of stateful fuzzing is to infer the state machine model of the SUT, which describes its possible states and transitions. The state machine model can help the fuzzer to generate valid and meaningful inputs that can trigger different behaviors of the SUT. Most of the existing stateful fuzzers infer the state machine model during the fuzzing process, by observing either the messages sent over the network or the memory of the system and clustering it into different states. However, these approaches can be inefficient and inaccurate. The inputs used during the fuzzing process are suboptimal for learning the state machine, because they do not represent the protocol's language effectively and may miss some states or transitions.

In this thesis, we present a novel approach for stateful fuzzing that learns a close approximation of the underlying state machine model of the implementation under test before the actual fuzzing process, rather than inferring it during it. We use a technique called *active automata learning* (aka *model learning* [4]), which iteratively queries the SUT with carefully selected inputs and learns a state machine model from the responses.

We compare this approach with two state-of-the-art fuzzers, AFLNet [5] and StateAFL [6], which are stateful fuzzers based on AFL/AFL++ [9, 10]. The protocol we use for this comparison is EDHOC (Ephemeral Diffie-Hellman Over COSE) [7], a lightweight and secure key exchange protocol that is designed for constrained devices and networks. We measure the code coverage and the number of vulnerabilities discovered by each fuzzer. We also discuss the strengths, limitations and challenges of each approach and suggest directions for future research.

2.1 Contributions

This thesis contributes to the field of network protocol security through an extensive comparative examination of current stateful fuzzing techniques applied to network protocol implementations. By systematically evaluating and comparing stateful fuzzers, we provide some insights into the strengths and weaknesses of these techniques and identify areas for future research. We hold that the performance comparison results we report and the conclusions we draw from them advance the collective understanding within the realm of stateful and protocol fuzzing. Additionally, this work establishes a foundation for future research endeavors, providing examples of fuzzer setup and preparation of the SUT for fuzzing.

2.2 Outline

The rest of the thesis is organized as follows:

- Chapter 3 covers the essential theoretical background of stateful fuzzing and briefly overviews the fuzzers that we used.
- Chapter 4 provides technical details about the protocols under test.
- Chapter 5 describes the setup of the experiments.
- Chapter 6 presents the results of our experiments and discusses them.
- Chapter 7 draws some conclusions and lists ideas for possible future work.

Chapter 3

Background

3.1 Fuzzing

Fuzzing [1], also known as fuzz testing, is an automated testing technique that is used to discover vulnerabilities and software defects by sending a large number of invalid, random, or unexpected inputs to a program or system. The goal of fuzzing is to discover issues, such as crashes, failing code assertions, memory leaks and buffer overflows in the target software, which can include applications, libraries, or network protocol implementations.

Typically, fuzzers are used to test programs that take structured inputs. This structure is specified, for example, in a file format or protocol and distinguishes valid from invalid input. An effective fuzzer generates semi-valid inputs that on the one hand are "valid enough" in that they are not directly rejected by the parser and on the other hand are "invalid enough" to expose corner cases that have not been properly dealt with. For the purpose of security, input that crosses a trust boundary is often the most useful.

Fuzzers can be categorized into:

Generation-based fuzzers: They require detailed knowledge of the program's input format and often use a configuration file to generate test cases that can deeply test the target program.

Mutation-based fuzzers: They start with some valid initial inputs and create new test cases by mutating these inputs. They are easier to use because they require less effort from the testing team and no input specifications.

Fuzzing approaches can also be classified into the following general categories:

Black-box fuzzing: This approach involves testing a system without any knowledge of its internal structure (e.g. its source code) or any of its implementation details.

White-box fuzzing: This method involves a more in-depth analysis of the system's internal structure (e.g. its code base) by more targeted and specialized fuzzing inputs based on the system's specific components and logic.

Gray-box fuzzing: This kind of fuzzing provides a balance between efficiency and effectiveness by conducting a biased random search over the domain of program inputs using a feedback function from observed test executions. This feedback is typically based on some notion of code coverage (e.g., line or branch coverage), which guides the mutations towards more advanced test cases.

The initial inputs of mutation-based fuzzers, like those we evaluate, represent a sequence of bytes. We will refer to them as *concrete seeds*, whilst we will refer to seeds that just hold the type or types of the data (i.e., the type of messages as specified by the protocol) as *abstract seeds*. An abstract seed may describe multiple concrete seeds.

More information about fuzzing can be found on some recent survey papers on the subject [27, 28, 29].

3.2 Stateful Fuzzing

Stateful fuzzing focuses on finding vulnerabilities in programs that have internal states and react to inputs based on their current state. What makes stateful fuzzing unique is its ability to understand and explore the program's current state during the testing process. In contrast to conventional fuzzing methods, stateful fuzzing keeps track of the program's behavior or internal variables and uses this information to infer the underlying state machine model and the current state. This allows it to generate test cases that are finely tailored to specific program states.

Having awareness of the program's state, stateful fuzzing can more accurately mimic real-world scenarios and interactions. This approach is especially valuable when testing complex software systems like network protocols and web applications, where their behavior is determined by the sequence of inputs and the program's internal state. Some bugs may only be exposed in certain states, which require a specific sequence of inputs to reach.

The main challenge of stateful fuzzing is to cover the state space of the system without having an explicit specification of the protocol. This involves partially uncovering the state space of the protocol and incorporating strategies for state identification.

3.3 Active Automata Learning

Active Automata Learning (AAL) [4] is an automated method for constructing a state machine model that closely matches the system's behavior. It does so by providing inputs and observing the outputs of the System Under Learning (SUL). This process involves two phases:

1. **Hypothesis Construction:** In this phase, sequences of input symbols are sent to the system. The responses from the system are used to create a so called *hypothesis*, i.e., a candidate model of the system, usually expressed as a finite state automaton. The goal is to make a model that produces the same

outputs as the system for all input sequences sent to the system. For input sequences not used, the model makes assumptions based on what it has learned.

2. **Hypothesis Validation:** In this phase, the hypothesis model is tested to see if it accurately represents the system. More input sequences are sent to both the system and the model. If there are any differences in their outputs, that indicates that the model needs a refinement and the process returns to the hypothesis construction phase to refine the model. If no differences are found, the learning process ends and the hypothesis model is considered a good approximation of the system.

If this process does not terminate, then it is possible that SUT's behavior cannot be modeled by a deterministic finite state automaton.

3.4 Protocol State Fuzzing

Protocol State Fuzzing [8] is a technique that uses active automata learning, in order to infer the state machine of a protocol's implementation. The learned models are used to expose logical flaws via non-standard or unexpected message sequences. By fuzzing sequences of messages, rather than individual messages, this technique can uncover deeper vulnerabilities that might be missed by other methods.

The learning setup for this technique requires a *Learner*, a *Mapper* and a SUL. The Learner is responsible for querying the SUL and receiving its responses, effectively running the learning algorithm. However, the Learner does not know how to convert abstract input symbols into concrete protocol messages. This is where the Mapper comes in.

The Mapper is an intermediate component that tries to bridge the gap between the Learner's abstract input alphabet and the concrete protocol messages used by the SUL. It transforms abstract input symbols from the finite input alphabet into concrete protocol messages that are sent to the SUL, filling in the necessary details. It also maps the concrete messages from the SUL to abstract symbols of the output alphabet, removing any protocol-specific unnecessary details.

3.5 AFL

American Fuzzy Lop (AFL) [9, 10, 11] is an effective fuzzer that excels in speed, reliability, and ease of use. It employs a clever combination of mutational and coverage-guided techniques. It mutates a set of test cases to reach previously unexplored areas of the program. When a test case uncovers new coverage, it is saved in the test case queue.

AFL uses a queue to manage test cases it tries. By default, it follows a FIFO (First-In-First-Out) policy, processing them in the order they are added. On top of that, AFL prioritizes smaller and faster test cases to improve efficiency. It marks these as "favored". When selecting the next test case, AFL skips non-favored cases with a high probability if there's at least one favored option available. Otherwise, cases that have been tried before are skipped with a higher probability.

The mutations in AFL are classified into two main categories: deterministic and havoc. Deterministic mutations involve single, predetermined alterations to the content of test cases, such as bit flips, additions, substitutions with integers from a predefined set of interesting values, and more. Havoc mutations are randomly stacked and can involve altering the size of the test case by adding or deleting portions of the input. Furthermore, AFL may merge two test cases into one and then apply havoc in a later stage known as the splicing stage.

This genetic algorithm and the favored seed selection are embedded in AFLNet, StateAFL and AFLML.

3.6 Stateful Fuzzers

Stateful fuzzers are a class of fuzzes that leverage the concept of system states to guide the fuzzing process. They are particularly effective for testing software with complex state-dependent behaviors.

3.6.1 AFLNet

AFLNet [5] is a greybox fuzzer designed for protocol implementations. AFLNet, based on AFL [9], adopts a genetic algorithm to generate the best inputs. On top of that it utilizes state feedback to direct the fuzzing process

It begins with recorded message exchanges between a server and a real client, called seeds, eliminating the need for any protocol specifications or message grammars. AFLNet takes on the role of a client, replaying the modified versions of the original message sequence sent to the server. It keeps those modifications that prove effective in expanding code or state coverage.

To identify the states that the server goes through when a sequence of messages is sent, AFLNet relies on the server's response codes. Using this feedback, AFLNet determines which state to focus on next using several heuristics. For example, to identify rarely exercised states it selects a state with a probability inversely proportional to the proportion of mutated message sequences that have exercised it and to maximize the probability of discovering new state transitions, AFLNet prioritizes a state that has been particularly successful in contributing to increased code or state coverage when previously selected.

3.6.2 StateAFL

StateAFL [6] is a greybox fuzzer designed for network servers. One of its salient points is that it does not require manual customization such as protocol models, protocol parsers, and learning frameworks. Instead, it uses lightweight dynamic analysis of the target program.

StateAFL works by instrumenting the target server at compile-time, inserting probes on memory allocations and network I/O operations. During runtime, it infers the current protocol state of the target server by taking snapshots of long-lived memory

areas. It then uses a type of hashing algorithm, which allows for some level of variation, to convert the contents of the memory into a unique identifier for that state. This allows StateAFL to recognize a state even if small changes have occurred in the memory. By recognizing these states, StateAFL can incrementally build a protocol state machine to guide fuzzing.

StateAFL has been integrated with a large set of network servers for popular protocols, without any manual customization to accommodate the protocol. Experimental results show that StateAFL can achieve comparable or even better code coverage and bug detection than customized fuzzing.

3.6.3 AFLML

AFLML is a greybox fuzzer that can be utilized in conjunction with Protocol State Fuzzing. Although based on AFLNet, AFLML does not extract response codes from messages. It has no knowledge of the protocol language and it is not able to identify the type of messages sent by the System Under Test (SUT).

The main idea behind AFLML is that fuzzing performance can be improved if the state machine of the system to fuzz is known in advance. The fuzzing process of this fuzzer involves the following steps:

1. Model Learning: The first step consists of learning the state machine of the SUT using some active automata learning technique.
2. Abstract Seed Generation: After learning the SUT's state machine, abstract seeds corresponding to potential transitions in the state machine are produced. These seeds can be thought of as message type sequences that trigger specific state changes in the SUT. The goal is to create a transition cover that encapsulates seeds for all possible transitions.
3. Concretization: In this step, abstract seeds are concretized, which means that concrete input data is generated from the abstract seeds. Concretization involves creating valid input data that corresponds to the abstract states and transitions identified in the state machine. These concrete inputs are sent to the SUT during fuzzing.
4. Fuzzing: With the concrete seeds in hand, the fuzzer can start the actual fuzzing process. AFLML can initiate the fuzzing from any state of the SUT using these seeds, allowing it to thoroughly explore different states and transitions within the SUT's state machine. AFLML uses a round-robin approach for state selection without any heuristic.

By understanding the state machine of the SUT, AFLML can guide the fuzzing process more effectively, allowing it to explore different states and transitions in a structured manner. AFLML's approach is particularly useful for testing software with complex state-dependent behaviors. In such cases, understanding and controlling the state transitions is crucial for effective testing and the detection of vulnerabilities. The state transitions in these systems can often lead to unexpected behaviors or vulnerabilities that are not apparent under normal operation. Moreover, AFLML can generate test inputs that can reach deeper into the software's state

space. This can potentially expose vulnerabilities that are hidden in rarely visited states or transitions. Thus, AFLML provides a more thorough and effective fuzzing approach compared to other (stateful) fuzzing techniques.

3.7 Related Work

Protocol implementations have been thoroughly analyzed for different kinds of vulnerabilities and bugs. TLS-Attacker [12] serves as an efficient framework for testing TLS implementations. de Ruiter and Poll [8] were among the first to use systematic state fuzzing and analyze TLS state machines learned using model learning. Similar work has been done for DTLS by Fiterau-Brostean et al. [13], leading to the creation of DTLS-Fuzzer tool [14], a state fuzzing framework based on TLS-Attacker. More recently, Sagonas and Typaldos [15] have applied protocol state fuzzing to EDHOC implementations and analyzed them. These works, however, rely on manual inspection of the state machine model to find bugs. An automated black-box technique for detecting state machine bugs in stateful network protocol implementations was later proposed by Fiterau-Brostean et al. [16].

In addition, numerous stateful fuzzing techniques have been introduced. For example, AFLNetLegion [17], an extension of AFLNet, presents a novel and principled algorithm for state selection. SGFuzz [18] is a stateful greybox fuzzer that builds upon LibFuzzer, utilizing additional feedback to navigate the state space of stateful software systems with the goal of revealing stateful bugs. NSFuzz [19] is a fuzzing solution designed for stateful network services, employing static analysis to pinpoint network event loops and extract state variables, thereby achieving rapid I/O synchronization and efficient state-aware fuzzing through lightweight compile-time instrumentation.

There has also been a significant amount of research dedicated to the systematic evaluation and comparison of different fuzzing techniques. For instance, Poncelet et al. [2] discussed the challenges associated with evaluating fuzzers, given the vast number of available fuzzing tools and the limited time for their evaluation.

Recent advancements in fuzzing-based research have played a crucial role in identifying vulnerabilities within protocol implementations. Efforts have been made to establish a systematic overview in the field of stateful fuzzing [20] and protocol fuzzing [21]. These survey papers offer systematic comparisons and classifications of fuzzers, and also highlight the challenges and opportunities for future research in this area.

Chapter 4

Protocols

In the evolving landscape of digital communication, protocols serve as the backbone of connectivity, especially in the realm of the Internet of Things (IoT). As we delve into this chapter, we aim to lay the groundwork for understanding the intricate standards and procedures that govern the secure and efficient exchange of data within a (security) network implementation. This chapter will not only contextualize the significance of each protocol within the larger ecosystem, but also set the stage for the subsequent exploration of stateful fuzzing techniques applied to these protocols.

4.1 CoAP

Constrained Application Protocol (CoAP) [22] is a communication protocol tailored for devices with limited resources, commonly found in the IoT domain. It follows a client-server model similar to HTTP, but is optimized for machine-to-machine interactions. CoAP operates over datagram-oriented transport protocols like UDP and is designed for asynchronous communication.

CoAP messages have a compact binary format and consist of a 4-byte header followed by a variable-length Token for correlation, a sequence of options encoded with delta values and an optional payload. Options can have different formats, including empty, opaque, uint (unsigned integer) and (UTF-8 encoded) string. Messages can be of four types: Confirmable, Non-confirmable, Acknowledgement and Reset. Confirmable messages support reliability with retransmissions, while Non-confirmable messages are not acknowledged. Requests and responses are carried in these message types and responses can be carried in Acknowledgement messages.

In CoAP, a response is identified by the Code field in the header, indicating the outcome of understanding and fulfilling a request. The upper three bits of the 8-bit Response Code define the response class, while the lower five bits provide additional details. Human-readable notations for CoAP Codes are in the format "c.dd," where "c" is the class in decimal and "dd" is the detail as a two-digit decimal. For instance, "Forbidden" is represented as 4.03, corresponding to an 8-bit code value of hexadecimal 0x83 (4*0x20+3) or decimal 131 (4*32+3).

The CoAP message format is depicted in Figure 4.1; the CoAP response codes are listed in Figure 4.2.

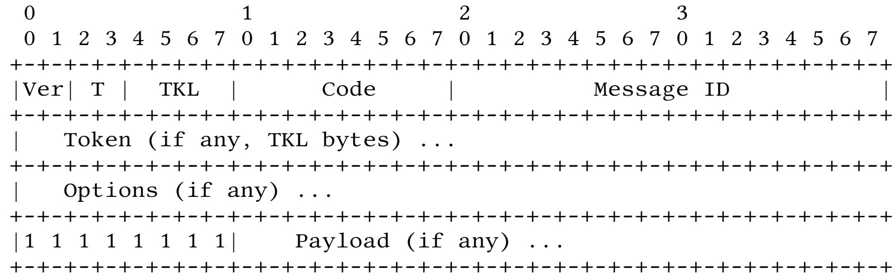


FIGURE 4.1: CoAP Message Format

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

FIGURE 4.2: CoAP Response Codes

4.2 EDHOC

The Ephemeral Diffie-Hellman Over COSE (EDHOC) [7] protocol is a compact and lightweight key exchange protocol designed for highly constrained environments, primarily targeting the IoT infrastructure. It offers security features like identity protection, cipher-suite negotiation and forward secrecy. The protocol relies on COSE for cryptography, CBOR for encoding and CoAP for transport.

EDHOC defines two main roles: Initiator and Responder. These roles are not tied to specific web transfer protocols, allowing flexibility for various applications. An EDHOC key exchange involves five messages, with *message_1*, *message_2* and *message_3* being mandatory. *message_4* is optional and error_message is available for both roles. These messages are encoded using CBOR data items.

Connection Identifiers play a crucial role in EDHOC. Each peer selects an identifier to identify the current session. Initiator chooses C_I and sends it in *message_1*, while Responder selects C_R and sends it in *message_2*. These identifiers enable message correlation and protocol state retrieval during the session and they can also be used for application-level purposes, such as OSCORE.

Authentication parameters include:

- Authentication Keys: These are public keys used for authentication. They can be either a signature key or a static Diffie-Hellman key, depending on the authentication method.
- Authentication Credentials: These are the public authentication keys of the Initiator and the Responder, referred to as CRED_I and CRED_R respectively. They are used to verify the integrity of the other peer and to verify proof-of-possession of the private key. They are usually not transported in EDHOC, but are provisioned otherwise.
- Authentication Credential Identifiers: These are identifiers used to recognize the corresponding stored authentication credentials. They are smaller in size than the credentials they identify, and are transported during EDHOC. The identifier of the Responder, ID_CRED_R, is transported in *message_2* and the identifier of the Initiator, ID_CRED_I, is transported in *message_3*.
- External Authorization Data (EAD): These are external application data that can be integrated into each message of the EDHOC protocol. They are included to reduce round trips, the number of exchanged messages, and simplify processing.

The EDHOC cipher suite consists of: EDHOC AEAD algorithm, EDHOC hash algorithm, EDHOC MAC length in bytes for static authentication, EDHOC key exchange algorithm, EDHOC signature algorithm for signed authentication, Application AEAD algorithm and Application hash algorithm. Cipher suite negotiation in EDHOC happens separately from the main exchange. The Initiator initially sends its preferred cipher suites in *message_1*, and the Responder responds with an error message containing its supported cipher suites, ending the current session. The Initiator must resolve this disagreement in the next *message_1*.

Ephemeral public keys (G_X and G_Y) are exchanged in *message_1* and *message_2*, serving as a source of randomness for each session. These keys are essential for ensuring the security of the key exchange.

The application profile is maintained by each peer and contains information necessary for processing and verification. It includes details about the authentication method, the use of *message_4* and the handling of external authorization data.

Finally, after successfully processing *message_3*, peers can derive OSCORE security context parameters, such as the OSCORE Master Secret and Master Salt, using the EDHOC-Exporter interface.

The EDHOC message flow is illustrated in Figure 4.3. The interaction between the client and server, with the server acting as the responder and the client as the initiator, is depicted in Figure 4.4.

4.3 OSCORE

Object Security for Constrained RESTful Environments (OSCORE) [23] is a method for application-layer protection of CoAP, using CBOR Object Signing and Encryption (COSE). It aims to establish end-to-end security between two CoAP endpoints,

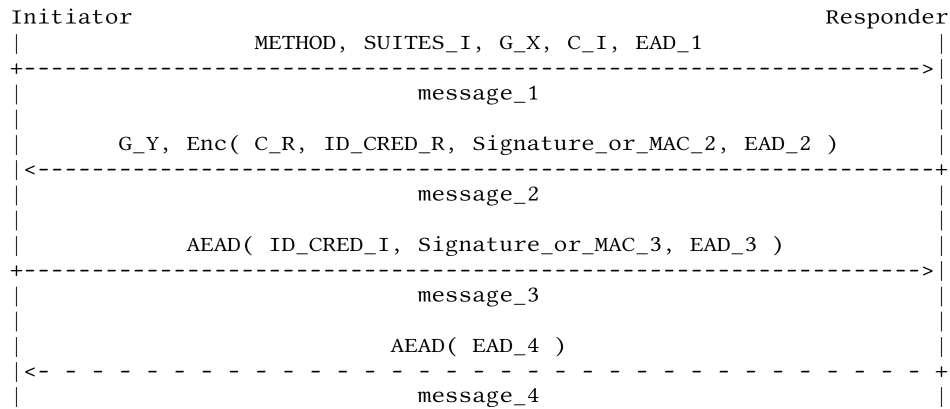


FIGURE 4.3: EDHOC message flow

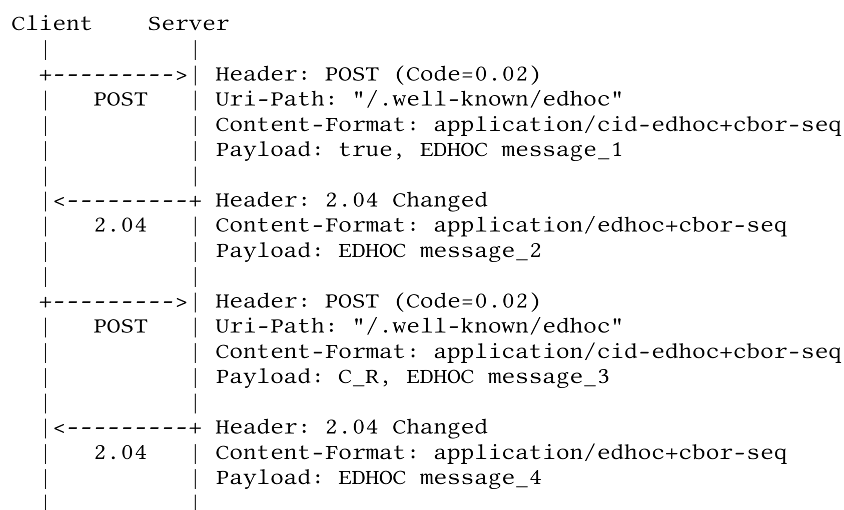


FIGURE 4.4: EDHOC message flow with server as responder

ensuring that intermediaries cannot tamper with or gain access to message fields unrelated to their designated operations. In essence, OSCORE takes an unprotected CoAP message and transforms it into a secure one, protecting not only the payload but also fully protected CoAP options, original request and response REST codes and certain parts of the URI pointing to the resources in the request messages.

For the OSCORE protocol to be used, the involved parties must establish a shared security context for processing COSE objects. This requires the secure and authenticated exchange of essential information and keying material, provided by a suitable key exchange protocol.

Chapter 5

Experimental Setup

5.1 Systems Used

5.1.1 uOSCORE-uEDHOC

The System Under Test we are using, called uOSCORE-uEDHOC, is a C implementation for constrained (and non-constrained) devices of the IETF protocols OSCORE [23] and EDHOC [7]. More specifically, we use one of the provided samples that implements a responder server that uses both EDHOC and OSCORE.

This SUT uses randomly generated keys, making it inappropriate for fuzzing. Randomness is bad for fuzzing because of the inability to reproduce the same results with the same seeds. If we replay a valid sequence of client requests for example, it might fail to pass the server validation check, because each server instance is randomized. Server responses are different between two runs, and the server expects different requests. If we want the same requests to trigger the same behavior on the server we need to de-randomize the SUT. That way, we will hopefully also be able to reproduce any crashes and hangs we find.

The SUT relies on external libraries for cryptography. MbedTLS cryptographic library is the primary source of randomness. Protocol implementations rely on such sources of randomness in order to be secure. But, in our case we need deterministic operation. So, we removed the random source, replacing a system call used to fill a buffer with random data, filling the buffer with predetermined data instead.

5.1.2 EDHOC-Fuzzer

EDHOC-Fuzzer [15] is a tool designed to analyze the implementations of the EDHOC protocol. This tool uses protocol state fuzzing to generate a close approximation of the underlying state machine model of an EDHOC implementation.

EDHOC-Fuzzer has an option to send individual abstract input sequences to the SUT. It was also extended with another option to export the requests right before they are sent to the SUT. If these options are used together, the tool can act as a concretizer, converting the abstract input sequences to concrete ones that can be used as seeds with AFLML or other fuzzers.

TABLE 5.1: Options used for AFLNet

Option	Description
-P COAP	Use COAP as the application protocol to be tested
-m none	Disable memory limit
-D 50000	Set Server initialization waiting time to 50000 μs
-q 3	Set FAVOR state selection algorithm
-s 3	Set FAVOR seed selection algorithm
-R	Enable region-level mutation operators
-E	Enable state-aware mode
-K	Kill the server after consuming all request messages
-W 50	Set response waiting time to 50 ms

5.1.3 AFLNet

Since AFLNet [5] requires a component with some knowledge of the protocol which is used, we needed to extend it for EDHOC support. In order to add support for another protocol, we had to implement two functions: one to separate concatenated requests, and one to extract response codes from concatenated responses. This is usually done either by finding a delimiter specific for each protocol or by finding the message size from the header. But, this was impossible for EDHOC, since it doesn't support either of those. So, we had to modify AFLNet to handle messages another way.

Originally, AFLNet uses raw files (concatenated messages) as seeds, while StateAFL accepts input seed files in replayable format. This format precedes each message with its size. Additionally, StateAFL features a generic function to separate buffered requests, since replayable format eliminates the necessity of protocol specific implementations. So, we borrowed some code from StateAFL for these features, thereby making AFLNet compatible with replayable format.

We also changed how responses are stored. Like requests, we precede them with their size, allowing for iteration within a buffer with stored responses.

The response code of CoAP messages is stored in the second byte. With the separation problem solved, we can easily extract the response codes.

We feed AFLNet with a single seed containing the EDHOC messages message_1 and message_3 followed by an OSCORE application message. This sequence corresponds to a key exchange followed by an encrypted message exchange and it has been concretized using EDHOC-Fuzzer.

The options we used are shown in Table 5.1.

5.1.4 StateAFL

We feed StateAFL [6] with a single seed containing the message_1 and message_3 followed by an OSCORE application message, just like AFLNet. Thanks to its versatility, no modifications were needed.

The options we used are shown in Table 5.2.

TABLE 5.2: Options used for StateAFL

Option	Description
-D 50000	Set Server initialization waiting time to 50000 µs
-q 3	Set FAVOR state selection algorithm
-s 3	Set FAVOR seed selection algorithm
-R	Enable region-level mutation operators
-E	Enable state-aware mode
-K	Kill the server after consuming all request messages
-W 50	Set response waiting time to 50 ms

TABLE 5.3: Options used for AFLML

Option	Description
-m none	Disable memory limit
-D 50000	Set Server initialization waiting time to 50000 µs
-q 3	Set FAVOR state selection algorithm
-s 3	Set FAVOR seed selection algorithm
-R	Enable region-level mutation operators
-E	Enable state-aware mode
-K	Kill the server after consuming all request messages
-W 50	Set response waiting time to 50 ms

5.1.5 AFLML

To create the seeds for AFLML, we first learn the uOSCORE-uEDHOC state machine using EDHOC-Fuzzer. Then, using a script, we explore the possible traces through the state machine and create abstract seed files based on these. Afterwards, we concretize those seeds with EDHOC-Fuzzer, and use them as input.

The options we used are shown in Table 5.3.

5.2 Settings

Platform All the experiments were conducted on a server with four Intel(R) Xeon(R) E5-4650 @ 2.70GHz CPUs and 128 GB of RAM running Debian GNU/Linux 12.4.

Number of Runs Despite our best efforts to de-randomize the SUT, there may be still some randomness left. Also, the fuzzers are randomized. They apply random mutations for example. In consequence, two runs of the same fuzzer may not give exactly the same results. In order to minimize the impact of randomness, we conducted ten independent experiments for each fuzzer. Each experiment runs on an isolated Docker container using the ProFuzzBench [24] execution scripts.

Response Waiting Time The `-W` option configures the polling timeout determining the maximum duration the fuzzer will wait for responses. If this value is set too low, the fuzzer may miss important feedback from the SUT due to premature

timeouts. Also, misreported crashes can be increased. On the other hand, if this value is set too high, it can decrease the number of executions per second, which in turn negatively affects the fuzzer's effectiveness.

How to select an optimal response waiting time is not obvious. We conducted experiments with waiting times of 20 ms, 50 ms, and 100 ms. Our findings suggest that a 50 ms waiting time is a good choice, striking a balance between performance and accuracy.

Figure 5.1 provides a comparison of edge and line coverage across various waiting times and fuzzers. Average crashes discovered by each configuration are depicted in Figure 5.2.

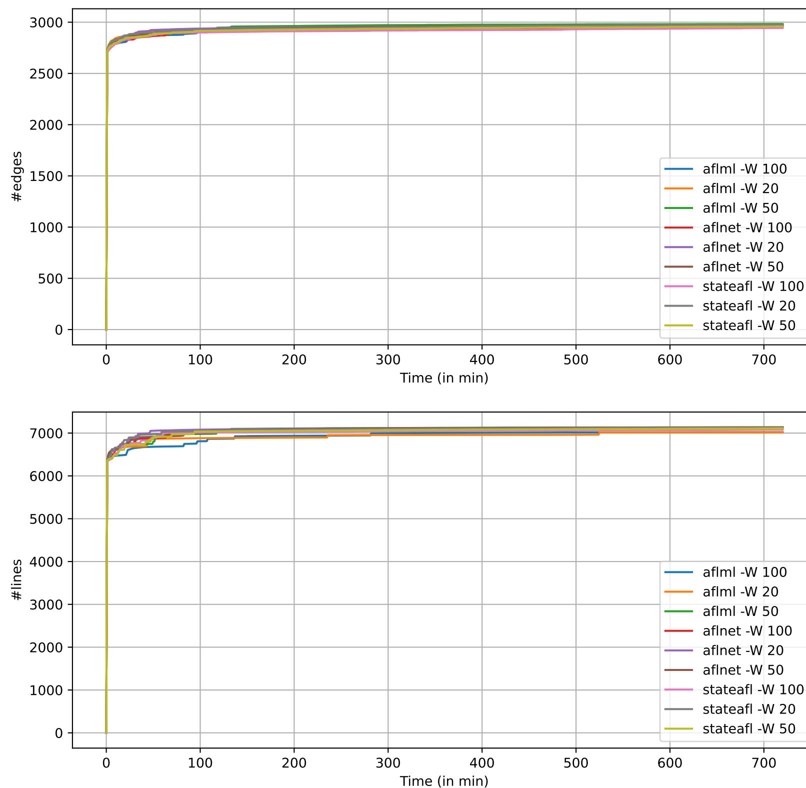


FIGURE 5.1: Code coverage varying the response waiting time.

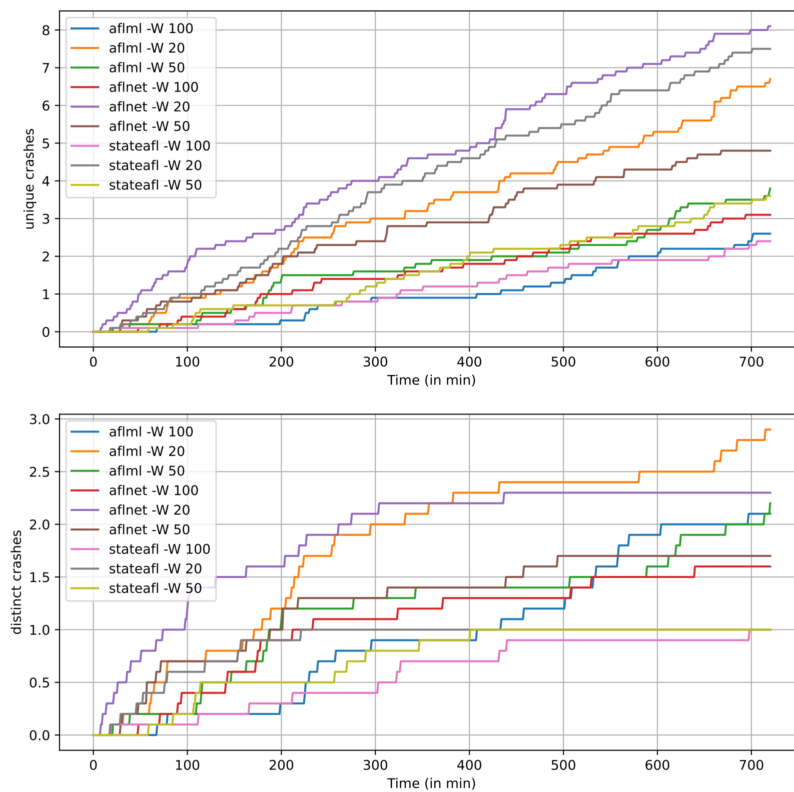


FIGURE 5.2: Number of crashes (averages from ten runs) varying the response waiting time.

Chapter 6

Results

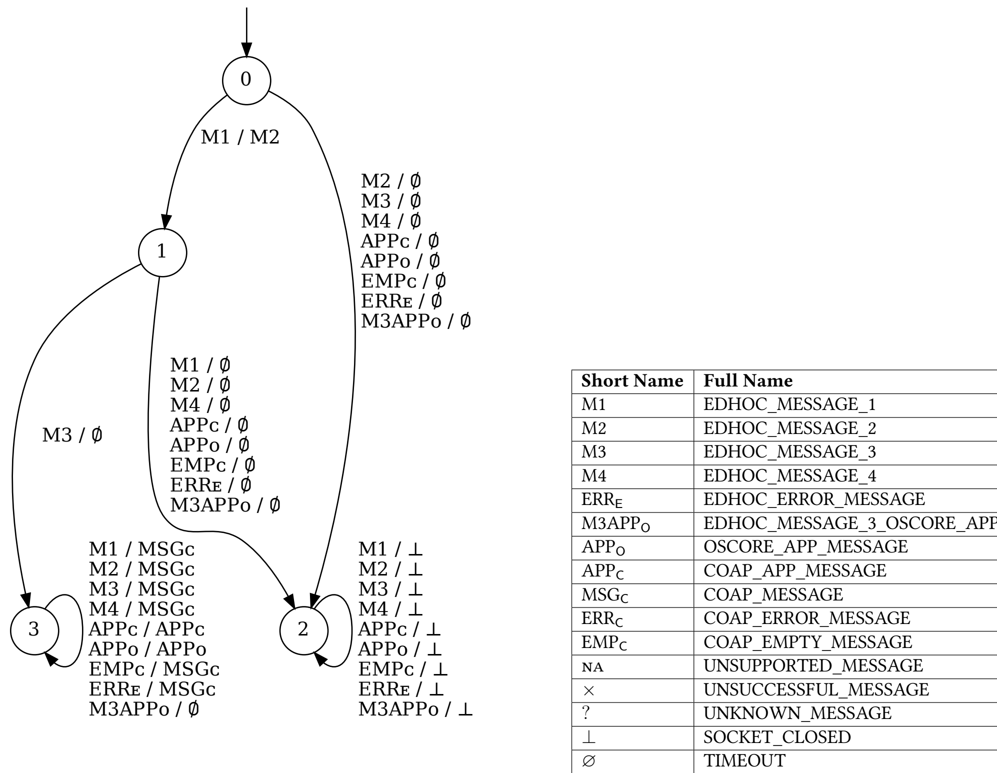
Using the setup we described, we run each fuzzer for 24 hours. We compare the fuzzers in terms of accuracy of state machine generation, edge and line coverage, and number of crashes found.

6.1 State Machine Comparison

First, we compare the three different learned state machine models of the SUT: the EDHOC-Fuzzer model (Figure 6.1), the AFLNet model (Figure 6.2), and the StateAFL model (Figure 6.3).

Since we use Active Automata Learning (AAL) for constructing the state machine model for AFLML with EDHOC-Fuzzer, this is of course the most accurate model and we will use it as a reference. It is shown as a Mealy machine (i.e., its edges are annotated with labels of the form I/O showing relationships between inputs and outputs), and contains four states in total (labeled 0 to 3). The other fuzzers indicate the states and edges discovered by the initial seeds with blue and the rest with red. State 0 represents the initial state in all models.

AFLNet creates a state for each response code. Its state machine has only three states, labeled 0, 68, and 69. We observed that M2 and APP_O have response code 68 (Changed) and MSG_C has response code 69 (Content). So, we can immediately speculate that state 68 represents state 1 in the AAL model, because state 1 returns M2, and state 69 represents state 3, because state 3 returns MSG_C. The first message in the initial seed, M1, discovers the edge from 0 to 68. So we can verify that state 68 represents state 1 in the AAL model. The next message in the initial seed is M3 and we would expect it to discover a state similar to state 3 in the AAL model. But, it doesn't yet because AFLNet relies on response codes for state discovery and the sequence M1-M3 results in a timeout after M3 is sent. Instead, such a state, state 69, can be discovered later with the sequence M1-M3-M3 or M1-M3-M1 using the a mutation that adds portions of the existing input. The last message in the initial seed is APP_O and it discovers the edge 68-68. Such an edge doesn't exist in the AAL model and its creation is a result of M2 and APP_O having the same response code and AFLNet not having discovered state 69 yet. The edge 69-68 is also a result of the former. The edge 69-69 can be discovered using one of the following sequences: M1-M3-M3-M3, M1-M3-M3-M1, M1-M3-M1-M3, M1-M3-M1-M1. AFLNet is also unable to discover state 2, like state 1, because there is no appropriate sequence



Short Name	Full Name
M1	EDHOC_MESSAGE_1
M2	EDHOC_MESSAGE_2
M3	EDHOC_MESSAGE_3
M4	EDHOC_MESSAGE_4
ERR$_E$	EDHOC_ERROR_MESSAGE
M3APP$_O$	EDHOC_MESSAGE_3_OSCORE_APP
APP$_O$	OSCORE_APP_MESSAGE
APP$_C$	COAP_APP_MESSAGE
MSG$_C$	COAP_MESSAGE
ERR$_C$	COAP_ERROR_MESSAGE
EMP$_C$	COAP_EMPTY_MESSAGE
NA	UNSUPPORTED_MESSAGE
×	UNSUCCESSFUL_MESSAGE
?	UNKNOWN_MESSAGE
⊥	SOCKET_CLOSED
∅	TIMEOUT

FIGURE 6.1: EDHOC-Fuzzer learned model of the SUT.

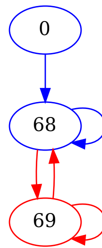


FIGURE 6.2: AFLNet state machine model.

that returns a response code. For example, the sequence M1-M1-M1 results in a timeout followed by a server shutdown.

Unlike AFLNet, StateAFL is less consistent with its state machine creation resulting in various models between runs that share some similarities though. The first message in the initial seed, M1, discovers the edge from state 0 to state 1. The next message in the initial seed, M3, does not discover a new state, probably because the long lived memory content before and after the transition from state 1 to state 3 in the AAL model is similar. Instead, the last message in the initial seed, APP$_O$, discovers state 2, which resembles state 3 in the AAL model. Although the initial state machine of StateAFL can be considered an improvement to the initial state machine of AFLNet, StateAFL is later plagued by its algorithm creating duplicate

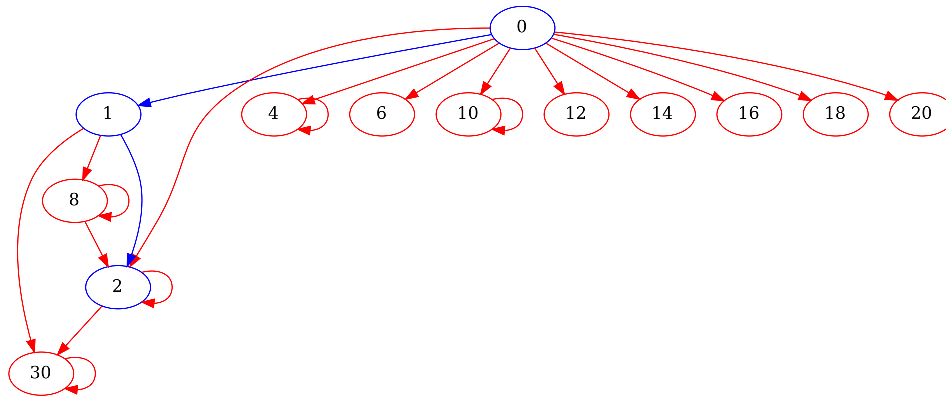


FIGURE 6.3: StateAFL state machine model (partial).

states. There are a lot of ambiguous states with one incoming transition from state 0, the initial state, and no outgoing transitions. Although we can see that some parts of the StateAFL state machine resemble the AAL model, in most runs there are dubious states and transitions. To sum up, StateAFL fails both to separate different states and to create states that are all meaningful.

6.2 Coverage Comparison

Edge and line coverage were tracked over time for each of the three fuzzers using the ProFuzzBench analysis scripts. AFLNet was evaluated twice, once without the `-E` and `-q` flags, serving as a baseline to assess the impact of state awareness.

Edge and line coverage graphs are depicted in Figure 6.4. Their analysis indicates that all fuzzers achieve near-maximum coverage within the first two hours, after which the rate of coverage growth diminishes. The difference in their coverage performance is insignificant.

Coverage percentages are not very useful, because a lot of the lines and edges are unreachable. For instance, uOSCORE-uEDHOC server uses a lot of external libraries, but not all of their functions. Still, coverage percentage graphs appear in Figure 6.5.

6.3 Crashes Comparison

AFL-based fuzzers, such as those evaluated in our comparison, categorize crashes as “unique” if they exhibit different coverage profiles. To ensure the validity of these crashes, we employ a script that filters out false positives by re-executing the replayable crashes and monitoring the output from the instrumented SUT. The SUT utilized for this verification is the same we used during the fuzzing process and it is compiled with AddressSanitizer (ASAN), which helps detect memory errors. We analyze the AddressSanitizer summary, which contains details about where and how the program crashed, as an indicator of crash diversity. An example of an

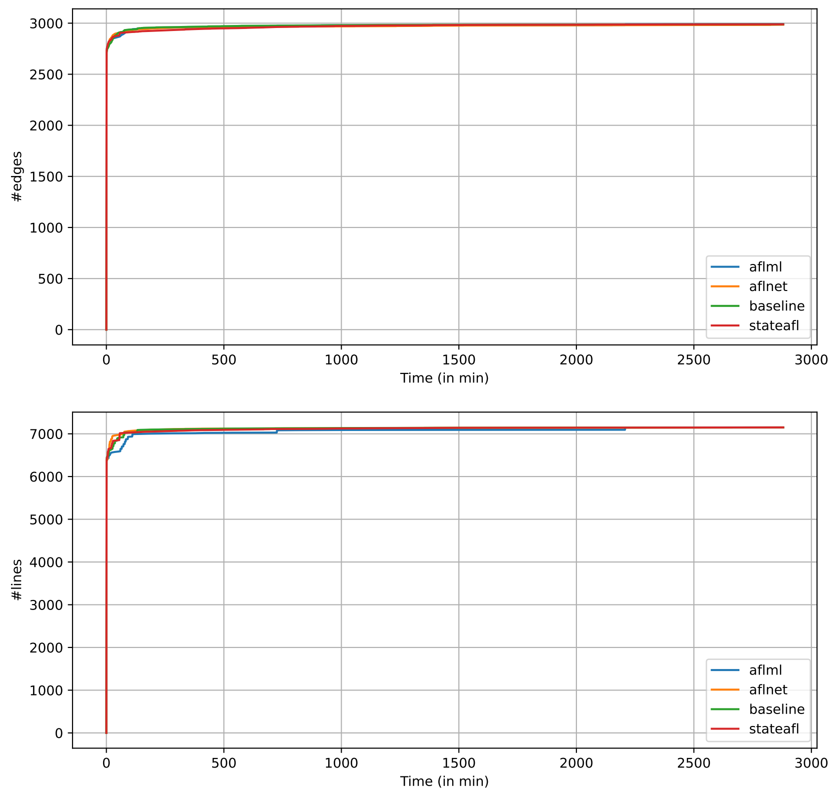


FIGURE 6.4: Code coverage of the different fuzzers over time (absolute numbers).

AddressSanitizer summary is provided in Figure 6.6. We refer to crashes with different summaries as "*distinct*". Distinct crashes have a higher probability of being associated with separate bugs compared to "unique" crashes. The average (over 10 experiments) "unique" and "distinct" crashes are shown in Figure 6.7. AFLNet without the `-E` and `-q` flags serves as a baseline.

AFLML demonstrates superior performance in terms of being able to discover most unique crashes. The difference becomes pretty clear after the first ten hours, post which there is a gradual decline in the performance of other fuzzers. Among the other fuzzers, AFLNet performs best and baseline performs worst. However, the performance gap between AFLNet, StateAFL and baseline is relatively small.

In the first three hours, baseline, AFLNet, and AFLML demonstrate comparable performance in the discovery of distinct crashes. However, the performance of baseline and AFLNet gradually deteriorates over time, and AFLML eventually outperforms them. When it comes to crash diversity, baseline and AFLNet exhibit similar performance. In contrast, StateAFL lags significantly behind, managing to discover only a single distinct crash. This could be attributed to the misleading state machine it creates.

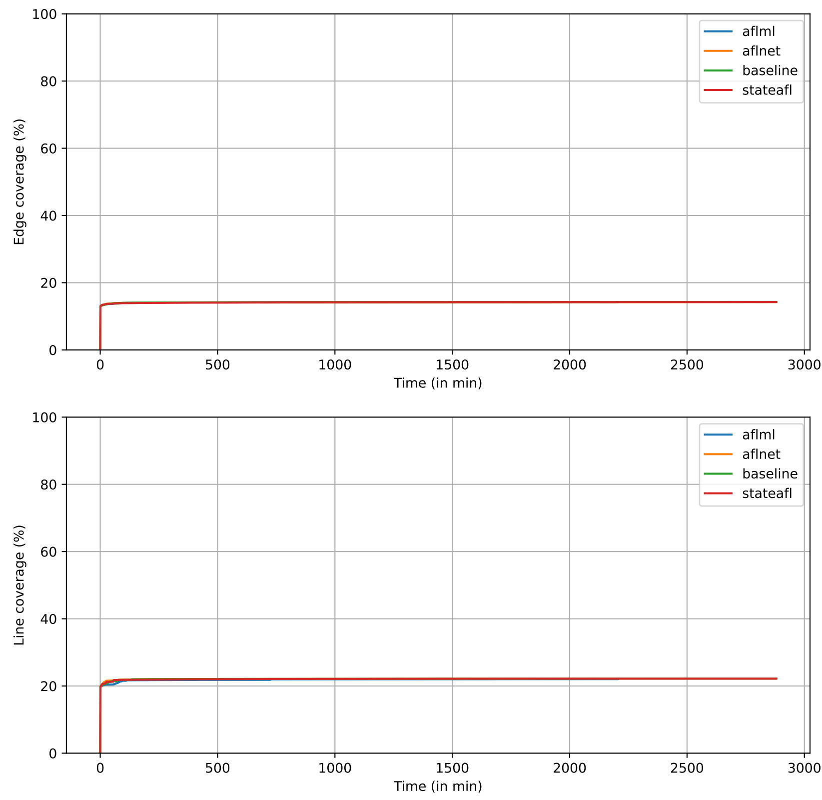


FIGURE 6.5: Code coverage of the different fuzzers over time (%).

```
SUMMARY: AddressSanitizer: SEGV (/lib/x86_64-linux-gnu/libc.so.6+0xbbaeb)
```

FIGURE 6.6: AddressSanitizer summary

In total, we detected ten distinct crashes. Five were only detectable by AFLML, while three of those were detected only once. One distinct crash though, that was detected in some runs by AFLNet and baseline, was undetectable by AFLML. Also, AFLNet consistently missed a distinct crash, that was detectable by baseline. Only one distinct crash was detectable by StateAFL.

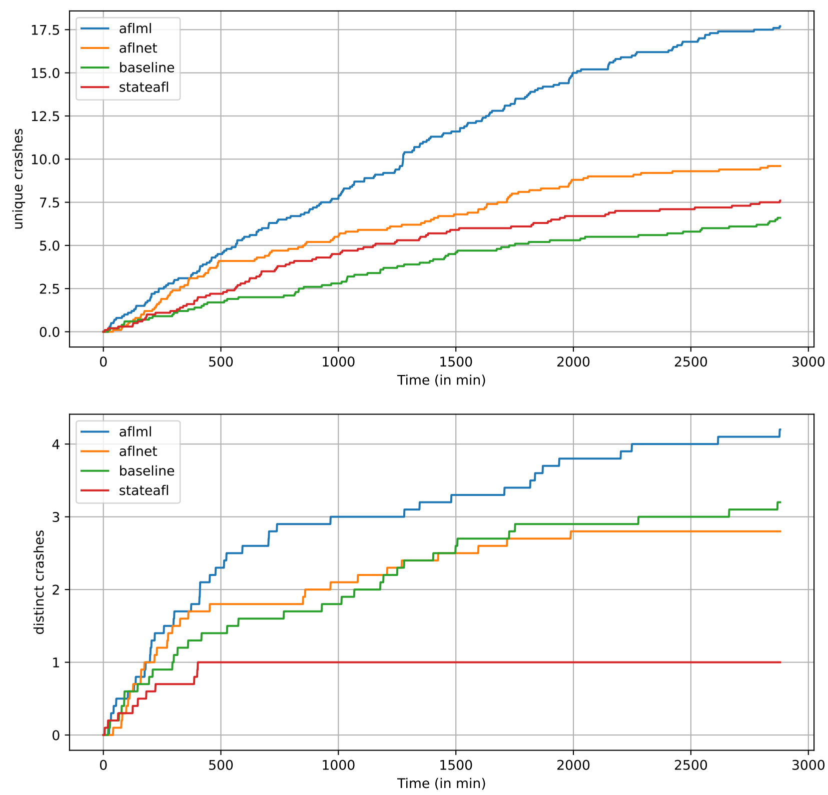


FIGURE 6.7: Average number of unique and distinct crashes triggered by different fuzzers over time.

Chapter 7

Conclusion

This thesis has explored and experimentally evaluated some variations of stateful fuzzing, a specialized form of fuzz testing that considers the state of the system being tested. The study was centered around a novel approach where the underlying state machine model of the system under test is learned before the actual fuzzing process, rather than during it.

The results showed that learning the underlying state machine model of the system under test before the actual fuzzing process can lead to more accurate and efficient fuzz testing and improve the fuzzing performance. However, it was also found that each fuzzer involves tradeoffs and has its place in the toolbox. For instance, StateAFL does not require any modification and works out of the box, while AFLNet gives slightly better results with minimal modification. AFLML gives the best results, but its setup requires a model learning component that is protocol specific.

We also pinpointed specific areas of improvement for certain fuzzers. For instance, StateAFL constructs a state machine that is not entirely accurate, which in turn negatively impacts its performance. On the other hand, AFLNet struggles with handling timeouts, suggesting that it might be beneficial to treat timeouts as response codes. These insights underscore the complexities involved in designing fuzzers that strike a balance between high performance and ease of use.

Furthermore, we conclude that more testing needs to be done to evaluate the impact of stateful fuzzers. Evaluating fuzz testing tools with accuracy and consistency poses a formidable challenge, especially considering the ongoing evolution of techniques and the integration of increasingly complex features settings into these tools.

As software systems continue to evolve and become more complex, the role of fuzzing in maintaining software reliability and security will only become more critical. This study has made some (small) contributions to the field of fuzz testing. We hope that future research will continue to build on these findings to further improve the effectiveness and efficiency of fuzz testing.

Future Work

Stateful fuzzing is an evolving field with numerous opportunities for future research. The following points outline potential directions for advancing the field:

- **User-Friendly Tooling**: Developing user-friendly interfaces and documentation for stateful fuzzing tools can lower the barrier to entry for new users. Efforts to create more accessible tools will help in the broader adoption of stateful fuzzing practices.
- **Fuzzing as a Service**: Exploring the concept of "Fuzzing as a Service" could make advanced fuzzing techniques more accessible to a broader audience. Developing cloud-based platforms that offer stateful fuzzing capabilities could make it easier for developers to adopt these methods without the need for specialized in-house expertise.
- **Optimization of Fuzzing Strategies**: Investigating various fuzzing strategies and their effectiveness in different contexts can lead to the development of more efficient fuzzing processes. This includes the study of adaptive fuzzing techniques that can dynamically adjust based on the feedback from the system under test.
- **Integration with Other Testing Techniques**: Combining stateful fuzzing with other testing methodologies, such as protocol testing using symbolic execution (e.g., [25, 26]) or static analysis, could yield interesting results. Research into the most effective ways to integrate these techniques would be beneficial.
- **Protocol-Agnostic Design**: To enhance the versatility of stateful fuzzing, future research could focus on creating protocol-agnostic protocol state fuzzers. These fuzzers should be capable of handling a wide range of protocols with minimal configuration. This involves developing generic algorithms that can infer the state machine of the SUT, regardless of the underlying protocol specifics. Such a design would simplify the process of adapting the fuzzer to new or custom protocols.
- **Enhanced State Machine Learning**: Future work could focus on improving the algorithms used to learn state machines before fuzzing. This includes the development of techniques that can more accurately capture complex application behaviors and states, as well as methods to reduce the learning time without compromising the quality of the learned state machine model.
- **Machine Learning (ML) and Artificial Intelligence (AI) in Fuzzing**: Leveraging advancements in ML and AI to enhance stateful fuzzing could be a significant step forward. This might include using AI to predict the most likely states where vulnerabilities may exist or to optimize the fuzzing process itself.

By addressing these areas, the field of stateful fuzzing can continue to grow and contribute to the enhancement of software security. The ultimate goal is to create a safer digital environment by systematically identifying and mitigating potential vulnerabilities.

Bibliography

[1] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[2] Clement Poncelet, Konstantinos Sagonas, and Nicolas Tsiftes. "So Many Fuzzers, So Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2022. Rochester, MI, USA: ACM, Oct. 2022, 95:1–95:12. DOI: 10.1145/3551349.3556946. URL: https://doi.org/10.1145/3551349.3556946.

[3] Bo Yu et al. "Poster: Fuzzing IoT Firmware via Multi-stage Message Generation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 2525–2527. DOI: 10.1145/3319535.3363247. URL: https://doi.org/10.1145/3319535.3363247.

[4] Frits W. Vaandrager. "Model learning". In: *Commun. ACM* 60.2 (2017), pp. 86–95. DOI: 10.1145/2967606. URL: https://doi.org/10.1145/2967606.

[5] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "AFLNET: A Greybox Fuzzer for Network Protocols". In: *13th IEEE International Conference on Software Testing, Validation and Verification*. ICST 2020. Porto, Portugal: IEEE, Oct. 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062. URL: https://doi.org/10.1109/ICST46399.2020.00062.

[6] Roberto Natella. "StateAFL: Greybox fuzzing for stateful network servers". In: *Empir. Softw. Eng.* 27.7 (2022), p. 191. DOI: 10.1007/S10664-022-10233-3. URL: https://doi.org/10.1007/s10664-022-10233-3.

[7] Göran Selander, John Preuß Mattsson, and Francesca Palombini. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. Internet-Draft draft-ietf-lake-edhoc-22. Work in Progress. Internet Engineering Task Force, Aug. 2023. 112 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/15/.

[8] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *24th USENIX Security Symposium, USENIX Security 15*. Ed. by Jaeyeon Jung and Thorsten Holz. Washington, D.C., USA: USENIX Association, Aug. 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[9] Michal Zalewski. *American fuzzy lop*. 2017. URL: http://lcamtuf.coredump.cx/afl.

[10] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies*. Ed. by Yuval Yarom and Sarah Zennou. WOOT 2020. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[11] Andrea Fioraldi et al. "Dissecting American Fuzzy Lop: A FuzzBench Evaluation". In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (2023), 52:1–52:26. DOI: 10.1145/3580596. URL: https://doi.org/10.1145/3580596.

[12] Juraj Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl et al. Vienna, Austria: ACM, Oct. 2016, pp. 1492–1504. DOI: 10.1145/2976749.2978411. URL: https://doi.org/10.1145/2976749.2978411.

[13] Paul Fiterau-Brostean et al. "Analysis of DTLS Implementations Using Protocol State Fuzzing". In: *29th USENIX Security Symposium, USENIX Security 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 2523–2540. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[14] Paul Fiterau-Brostean et al. "DTLS-Fuzzer: A DTLS Protocol State Fuzzer". In: *15th IEEE Conference on Software Testing, Verification and Validation*. ICST 2022. Valencia, Spain: IEEE, Apr. 2022, pp. 456–458. DOI: 10.1109/ICST53961.2022.00051. URL: https://doi.org/10.1109/ICST53961.2022.00051.

[15] Konstantinos Sagonas and Thanasis Typaldos. "EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Ed. by René Just and Gordon Fraser. ISSTA 2023. Seattle, WA, USA: ACM, July 2023, pp. 1495–1498. DOI: 10.1145/3597926.3604922. URL: https://doi.org/10.1145/3597926.3604922.

[16] Paul Fiterau-Brostean et al. "Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations". In: *30th Annual Network and Distributed System Security Symposium*. NDSS 2023. San Diego, CA, USA: The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/automata-based-automated-detection-of-state-machine-bugs-in-protocol-implementations/.

[17] Dongge Liu et al. "State Selection Algorithms and Their Impact on The Performance of Stateful Network Protocol Fuzzing". In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2022. Honolulu, HI, USA: IEEE, Mar. 2022, pp. 720–730. DOI: 10.1109/SANER53432.2022.00089. URL: https://doi.org/10.1109/SANER53432.2022.00089.

[18] Jinsheng Ba et al. "Stateful Greybox Fuzzing". In: *31st USENIX Security Symposium, USENIX Security 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 3255–3272. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[19] Fan Hu et al. "NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing - RCR Report". In: *ACM Trans. Softw. Eng. Methodol.* 32.6 (2023), 161:1–161:8. DOI: 10.1145/3580599. URL: https://doi.org/10.1145/3580599.

[20] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. *Fuzzers for stateful systems: Survey and Research Directions*. 2023. DOI: 10.48550/ARXIV.2301.02490. arXiv: 2301.02490 [cs.CR]. URL: https://doi.org/10.48550/arXiv.2301.02490.

[21] Xiaohan Zhang et al. *A Survey of Protocol Fuzzing*. 2024. arXiv: 2401.01568 [cs.CR]. URL: https://doi.org/10.48550/arXiv.2401.01568.

[22] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP).* Tech. rep. 7252. June 2014. 112 pp. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[23] Göran Selander et al. *Object Security for Constrained RESTful Environments (OSCORE).* Tech. rep. 8613. July 2019. 94 pp. DOI: 10.17487/RFC8613. URL: https://www.rfc-editor.org/info/rfc8613.

[24] Roberto Natella and Van-Thuan Pham. "ProFuzzBench: A benchmark for stateful protocol fuzzing". In: *30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* Ed. by Cristian Cadar and Xiangyu Zhang. ISSTA '21. Virtual Event, Denmark: ACM, July 2021, pp. 662–665. DOI: 10.1145/3460319.3469077. URL: https://doi.org/10.1145/3460319.3469077.

[25] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications". In: *IEEE Trans. Software Eng.* 40.7 (2014), pp. 695–709. DOI: 10.1109/TSE.2014.2323977. URL: https://doi.org/10.1109/TSE.2014.2323977.

[26] Hooman Asadian et al. "Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification". In: *IEEE Conference on Software Testing, Verification and Validation.* ICST 2022. IEEE, Apr. 2022, pp. 70–81. DOI: 10.1109/ICST53961.2022.00019. URL: https://ieeexplore.ieee.org/document/9787883.

[27] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: A Survey". In: *Cybersecur.* 1.1 (2018), p. 6. DOI: 10.1186/S42400-018-0002-Y. URL: https://doi.org/10.1186/s42400-018-0002-y.

[28] Patrice Godefroid. "Fuzzing: hack, art, and science". In: *Commun. ACM* 63.2 (2020), pp. 70–76. DOI: 10.1145/3363824. URL: https://doi.org/10.1145/3363824.

[29] Valentin J. M. Manès et al. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563. URL: https://doi.org/10.1109/TSE.2019.2946563.