Διερεύνηση και Αξιολόγηση Προηγμένων Τεχνικών Συμμετρικής Κρυπτογράφησης: Πλήρης Ανάλυση των AES και Ascon ως Λύσεις για Σύγχρονα Συστήματα Ασφαλείας

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΔΙΚΤΥΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΑΛΕΞΙΟΥ ΔΗΜΗΤΡΑ

ΑΜ 190

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: AES, DES, blowfish, 3DES, Ascon, NIST, SBOX, light cryptography, Symmetric Cryptography, Authenticated Encryption, C, IoT devices, Threats

ΠΕΡΙΛΗΨΗ

Η παρούσα πτυχιακή εργασία πραγματεύεται τη μελέτη και συγκριτική αξιολόγηση σύγχρονων συμμετρικών κρυπτογραφικών αλγορίθμων, με έμφαση στους AES και Ascon. Στο πρώτο μέρος αναλύεται διεξοδικά ο αλγόριθμος AES, ο οποίος αποτελεί το επικρατέστερο διεθνές πρότυπο συμμετρικής κρυπτογράφησης. Παρουσιάζονται η μαθηματική του δομή, οι τρόποι λειτουργίας του και τα βασικά σχήματα padding, καθώς και παραδείγματα που διευκολύνουν την κατανόηση της εσωτερικής λειτουργίας του. Ακολουθεί η θεωρητική παρουσίαση του αλγορίθμου Ascon, ενός σύγχρονου authenticated encryption σχήματος που έχει τυποποιηθεί από το NIST και αποτελεί σημαντική πρόταση της πρόσφατης ερευνητικής κοινότητας. Στο δεύτερο μέρος της εργασίας αναπτύσσεται η πειραματική διαδικασία, η οποία οργανώνεται σε δύο στάδια. Αρχικά, μελετάται η συμπεριφορά του AES σε διαφορετικά modes λειτουργίας και διαφορετικά padding schemes, με μετρήσεις χρόνου κρυπτογράφησης–αποκρυπτογράφησης και χρήσης μνήμης για αρχεία ποικίλου μεγέθους. Στη συνέχεια πραγματοποιείται συγκριτική αξιολόγηση του AES με τον Ascon128a, εξετάζοντας την απόδοσή τους σε πραγματικές συνθήκες. Τα συμπεράσματα της εργασίας προσφέρουν μια ολοκληρωμένη εικόνα της πρακτικής απόδοσης των δύο αλγορίθμων και αναδεικνύουν τα πλεονεκτήματα και τις ιδιαιτερότητές τους σε διαφορετικά σενάρια χρήσης, παρέχοντας ένα χρήσιμο υπόβαθρο για περαιτέρω μελέτη και εφαρμογή σύγχρονων κρυπτογραφικών τεχνικών. Τέλος, η εργασία ολοκληρώνεται με τις μελλοντικές προεκτάσεις, όπου συζητούνται πιθανές εφαρμογές των AES και Ascon, προτάσεις για περαιτέρω έρευνα, καθώς και τρόποι επέκτασης της παρούσας πειραματικής διαδικασίας σε διαφορετικές πλατφόρμες και τεχνολογικά περιβάλλοντα.

ΕΙΣΑΓΩΓΗ

Η συνεχώς αυξανόμενη ανάγκη για ασφαλή και αποδοτική προστασία δεδομένων καθιστά την επιλογή και αξιολόγηση κρυπτογραφικών αλγορίθμων κρίσιμη για μια ευρεία γκάμα εφαρμογών, από συμβατικούς υπολογιστές μέχρι συσκευές Internet of Things (IoT) και ενσωματωμένα συστήματα. Στο πλαίσιο αυτό, ο αλγόριθμος AES (Advanced Encryption Standard) αποτελεί το καθιερωμένο διεθνές πρότυπο συμμετρικής κρυπτογράφησης, χάρη στην υψηλή ασφάλεια, την ώριμη υλοποίηση και τη βελτιστοποιημένη απόδοση σε σύγχρονα συστήματα. Ωστόσο, η ταχεία ανάπτυξη συσκευών με περιορισμένους υπολογιστικούς πόρους—όπως αισθητήρες, μικροελεγκτές και φορητές συσκευές—καθώς και η εμφάνιση νέων απειλών (π.χ. κβαντικές επιθέσεις) έχουν δημιουργήσει την ανάγκη για πιο ελαφριές και ενεργειακά αποδοτικές κρυπτογραφικές λύσεις. Στο πλαίσιο αυτό αναδεικνύεται ο Ascon, ο οποίος επιλέχθηκε από το NIST ως το νέο διεθνές πρότυπο ελαφριάς κρυπτογραφίας το 2023, προσφέροντας υψηλή ασφάλεια, χαμηλή κατανάλωση πόρων και αντοχή έναντι μελλοντικών απειλών. Η παρούσα εργασία εξετάζει συγκριτικά τους αλγορίθμους AES και Ascon τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο.

Αρχικά παρουσιάζεται το θεωρητικό πλαίσιο των αλγορίθμων AES και Ascon, με έμφαση στα χαρακτηριστικά σχεδίασης, στις λειτουργίες τους και στα κριτήρια ασφάλειας που διέπουν τη χρήση τους σε σύγχρονα συστήματα. Στη συνέχεια ακολουθεί το Πειραματικό Μέρος, το οποίο υλοποιείται σε δύο στάδια. Στο Πειραματικό Μέρος 1 εξετάζεται η απόδοση του AES σε διαφορετικά modes λειτουργίας και διαφορετικά σχήματα padding (ECB–Zero Padding, ECB–PKCS#7, CBC–Zero Padding, CBC–PKCS#7), με σκοπό την αξιολόγηση του τρόπου με τον οποίο η επιλογή του mode επηρεάζει την ταχύτητα και τη συμπεριφορά του αλγορίθμου. Στο Πειραματικό Μέρος 2 πραγματοποιείται σύγκριση μεταξύ AES-GCM και Ascon-128a, με υλοποίηση κοινής ανταλλαγής κλειδιού μέσω ECDH και μετρήσεις χρόνου σε αρχεία διαφορετικού μεγέθους, ώστε να αναδειχθούν οι διαφορές απόδοσης μεταξύ ενός καθιερωμένου και ενός ελαφρού, σύγχρονου αλγορίθμου. Η συνδυαστική αυτή μεθοδολογία επιτρέπει μια ολοκληρωμένη αξιολόγηση των επιδόσεων και της καταλληλότητας των δύο αλγορίθμων σε πραγματικές συνθήκες.

ΑΝΑΣΚΟΠΗΣΗ ΒΙΒΛΙΟΓΡΑΦΙΑΣ

Η ΑΡΧΗ ΤΟΥ AES

Τον Ιανουάριο του 1997, το National Institute of Standards and Technology (NIST) προκήρυξε έναν ανοιχτό διαγωνισμό για την αντικατάσταση του απαρχαιωμένου αλγορίθμου DES (Data Encryption Standard), με στόχο την ανάπτυξη ενός σύγχρονου και ισχυρού αλγορίθμου συμμετρικής κρυπτογράφησης. Ο διαγωνισμός ολοκληρώθηκε το 2000 με την επιλογή του αλγορίθμου Rijndael, που είχε αναπτυχθεί από τους Joan Daemen και Vincent

Rijmen, ενώ η επίσημη έγκρισή του ως το νέο πρότυπο AES (Advanced Encryption Standard) πραγματοποιήθηκε τον Νοέμβριο του 2001 [1].

Ο AES, κατά την αρχική του έγκριση, δεν προοριζόταν για αυστηρά διαβαθμισμένες στρατιωτικές ή κρατικές πληροφορίες, αλλά για την προστασία ευαίσθητων — αλλά μη διαβαθμισμένων — δεδομένων, όπως οικονομικά, προσωπικά ή διοικητικά έγγραφα [2].

Παρ' όλ' αυτά τα σπουδαία χαρακτηριστικά που διέθεται, όπως η διαθεσιμότητά του χωρίς δικαιώματα χρήσης(royalty-free), η προσαρμοστικότητα σε διαφορετικές πλατφόρμες (versatility), η ευελιξία κλειδιού(key agility), η απλότητα του σχεδιασμού του (simplicity) και η χαμηλές απαιτήσεις μνήμης σε περιβάλλοντα με περιορισμένους πόρους, εδραίωσαν τον αλγόριθμο AES ως παγκόσμιο πρότυπο στον τομέα της κρυπτογραφίας.

Επιπροσθέτως, η ευρεία υιοθέτηση του AES από διεθνείς οργανισμούς, όπως οι ISO, IEEE και IETF, σε συνδυασμό με την ενσωμάτωσή του από φορείς όπως το ETSI, καθώς και από κατασκευαστές κρυπτογραφικών βιβλιοθηκών, αναδεικνύουν τη λειτουργική του αξία και την καθολική του αποδοχή στο πεδίο της σύγχρονης κρυπτογραφίας, καθιερώνοντας τον AES ως βασικό κρυπτογραφικό εργαλείο σε πληθώρα εφαρμογών υψηλής ασφάλειας, τόσο στον δημόσιο όσο και στον ιδιωτικό τομέα [2].

Τέλος, ο NIST υιοθέτησε τρεις εκδοχές της οικογένειας του αλγορίθμου Rijndael(cipher family), γνωστές ως AES-128, AES-192 και AES-256, όπου ο αριθμός υποδηλώνει το μήκος του κλειδιού σε bits. Και στις τρεις περιπτώσεις, το μήκος του block δεδομένων ορίζεται σταθερά στα 128 bits, παρότι ο Rijndael έχει σχεδιαστεί ώστε να υποστηρίζει και εναλλακτικά μεγέθη block και κλειδιών [3].

ΜΕΛΕΤΕΣ ΓΙΑ ΤΟΝ AES

Από την υιοθέτησή του ως πρότυπο κρυπτογράφησης, ο αλγόριθμος AES (Advanced Encryption Standard) έχει αποτελέσει αντικείμενο εκτεταμένης μελέτης και αξιολόγησης. Πλήθος ερευνητικών εργασιών έχουν επικεντρωθεί στη συγκριτική του ανάλυση έναντι άλλων δημοφιλών αλγορίθμων, όπως ο DES (Data Encryption Standard), ο Blowfish και ο RSA (Rivest-Shamir-Adleman), με στόχο την αξιολόγηση της απόδοσης, της ασφάλειας και της αποδοτικότητας σε διάφορα περιβάλλοντα.

Στο παρόν κεφάλαιο επιχειρείται η συγκριτική παρουσίαση προηγούμενων μελετών που εστιάζουν στην αξιολόγηση του αλγορίθμου AES σε σχέση με άλλους γνωστούς συμμετρικούς και ασύμμετρους αλγορίθμους κρυπτογράφησης. Η ανασκόπηση επικεντρώνεται σε βασικά τεχνικά χαρακτηριστικά, όπως η απόδοση, η ασφάλεια έναντι επιθέσεων και η κατανάλωση υπολογιστικών πόρων. Για την αποτίμηση των επιδόσεων του AES, επιλέχθηκαν δύο βασικές κατηγορίες μελετών: (α) βιβλιογραφικές / ανασκοπικές, οι οποίες αναλύουν συγκριτικά πορίσματα προγενέστερων ερευνών, και (β) πειραματικές, οι οποίες βασίζονται σε μετρήσεις από πραγματικά συστήματα και προγράμματα προσομοίωσης. Η διάκριση αυτή κρίνεται απαραίτητη ώστε να εντοπιστούν όχι μόνο ποσοτικά αποτελέσματα αλλά και θεωρητικές ερμηνείες και τάσεις της σχετικής ερευνητικής κοινότητας.

ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ / ΑΝΑΣΚΟΠΙΚΕΣ ΜΕΛΕΤΕΣ

Αρχικά, σύμφωνα με πρόσφατη μελέτη των Alemami et al., στην οποία συγκρίνονται διάφοροι αλγόριθμοι κρυπτογράφησης σε περιβάλλον cloud, οι αλγόριθμοι RSA και IDEA κρίνονται λιγότερο ασφαλείς σε σύγκριση με τους AES, Blowfish και DES. Ο AES διακρίνεται για την ικανότητά του να κρυπτογραφεί μεγάλους όγκους δεδομένων, για τον περιορισμένο χρόνο που απαιτεί για την κρυπτογράφηση και, γενικά, για την ταχύτητα και αποδοτικότητά του σε σύγκριση με όλους τους υπόλοιπους αλγορίθμους, καθιστώντας τον ως την πλέον κατάλληλη επιλογή σε περιβάλλοντα cloud [4].

Η μελέτη των Baig et al. (2024) εξετάζει συγκριτικά τους αλγορίθμους AES, RSA και 3DES, εστιάζοντας στην απόδοση και την ταχύτητα εκτέλεσης. Τα αποτελέσματα δείχνουν ότι ο AES προσφέρει υψηλό ρυθμό μετάδοσης δεδομένων και αποτελεσματικότητα, με σχετικά χαμηλό φόρτο μνήμης, καθιστώντας τον κατάλληλο για εφαρμογές που απαιτούν ταχύτητα και ασφάλεια. Ο RSA, από την άλλη, παρουσιάζει σημαντική καθυστέρηση κατά την κρυπτογράφηση αρχείων μεγάλου μεγέθους, κυρίως λόγω του αυξημένου φόρτου μνήμης και της ανάγκης για μεγαλύτερα κλειδιά. Ο 3DES επιχειρεί να βελτιώσει τις αδυναμίες του DES, όμως θεωρείται ξεπερασμένος, λόγω του περιορισμένου μεγέθους κλειδιού και της μειωμένης αποδοτικότητας. Συνολικά, οι συγγραφείς καταλήγουν ότι ο AES αποτελεί την πιο ισορροπημένη επιλογή μεταξύ ταχύτητας, ασφάλειας και αποδοτικότητας [5].

Τέλος, μία ακόμη συγκριτική μελέτη η οποία σχετίζεται με την απόδοση των συμμετρικών αλγορίθμων AES και DES σε επίπεδο cloud είναι αυτή των Basco et al., (2025) [6], η οποία επικεντρώνεται στα προβλήματα που αντιμετωπίζονται στο cloud computing, όπως η ιδιωτικότητα, η λειτουργικότητα των δεδομένων (απώλεια, απειλές κλπ.) και τέλος η ισχύς των δεδομένων. Η εργασία των Basco et al., αποτελεί μία βιβλιογραφική ανασκόπηση μεταξύ 18 ερευνών οι οποίες παρουσιάζουν και αναλύουν τα χαρακτηριστικά των αλγορίθμων AES και DES στο πλαίσιο της ασφάλειας δεδομένων και του cloud computing. Συμπερασματικά, ο AES υπερέχει του DES σε όλα τα επίπεδα. Σύμφωνα με τους συγγραφείς ο AES θεωρείται ισχυρότερος, παρέχει μεγαλύτερα υψηλότερο επίπεδο ασφάλειας και είναι ανθεκτικότερος σε σύγχρονες απειλές, όπως injection malware και απώλεια δεδομένων σε σχέση με τον DES. Ακόμη, είναι αποδοτικότερος σε επίπεδο επεξεργασίας, κυρίως σε cloud εφαρμογές, ενώ παρουσιάζει υψηλότερη ταχύτητα κρυπτογράφησης σε σχέση με τους DES και 3DES. Επιπροσθέτως, αν και ο DES είναι απλούστερος στην υλοποίηση ο AES αποτελεί έναν πιο ολοκληρωμένο και μελλοντικά βιώσιμο αλγόριθμο. Στον παρακάτω πίνακα 1 αναγράφονται τα βασικά στοιχεία των προαναφερθέντων ερευνών.

Μελέτη	**Αλγόριθμοι**	**Πλαίσιο μελέτης**	**Κριτήρια σύγκρισης**	**Συμπεράσματα για AES**
Alemami et al. (2025)	AES, RSA, IDEA, Blowfish, DES	Ασφάλεια και απόδοση σε cloud περιβάλλον	Ασφάλεια, ταχύτητα, αποδοτικότητα	Υπερέχει σε ασφάλεια και απόδοση, κατάλληλος για cloud

Baig et al. (2024)	AES, RSA, 3DES	OpenSSL environment	Ταχύτητα, Απόδοση	Ισορροπημένη επιλογή απόδοσης και ασφάλειας
Basco et al., (2025)	AES και DES	Επίπεδο cloud	Ιδιωτικότητα, λειτουργικότητα και ισχύς	Αποδοτικότερος, ταχύτερος, με μελλοντικές προεκτάσεις

ΠΕΙΡΑΜΑΤΙΚΕΣ ΜΕΛΕΤΕΣ

Η έρευνα των Manurung et al., (2025), εξετάζει και συγκρίνει τους αλγορίθμους AES, DES, 3DES και RC6 σε σχέση με την ασφάλεια των ψηφιακών αρχείων. Καταλήγει ότι ο AES χαρακτηρίζεται ως πιο ανθεκτικός στην ασφάλεια, πιο αποτελεσματικός σε χρόνο και σε χώρο και πολύ πιο γρήγορος, ιδιαίτερα σε εξειδικευμένες συσκευές, θέτοντας πιο αποτελεσματικό σε σύγκριση με τους υπόλοιπους αλγορίθμους [7].

Η μελέτη των Buhari et al., εξετάζει επίσης, συγκριτικά τις επιδόσεις και τα χαρακτηριστικά ασφαλείας των αλγορίθμων AES, 3DES και Blowfish. Τα αποτελέσματα δείχνουν ότι ο Blowfish παρουσιάζει την υψηλότερη ταχύτητα και απόδοση, ανεξαρτήτως μεγέθους αρχείου, ακολουθούμενος από τον AES, ο οποίος επιτυγχάνει καλή ισορροπία μεταξύ απόδοσης και ασφάλειας. Ο 3DES, λόγω της χαμηλής του ταχύτητας και απόδοσης, θεωρείται λιγότερο κατάλληλος για απαιτητικά συστήματα. Σε όρους μνήμης, ο 3DES διαθέτει το μικρότερο αποτύπωμα, καθιστώντας τον πιο κατάλληλο για περιορισμένα περιβάλλοντα, ενώ ο Blowfish είναι ο πιο απαιτητικός. Σε επίπεδο ασφάλειας, ο AES ξεχωρίζει ως ο πλέον αξιόπιστος, προσφέροντας ισχυρή προστασία έναντι ποικίλων απειλών [8] .

Οι Assa-Agyei & Olajide πραγματοποίησαν πειραματική αξιολόγηση των αλγορίθμων AES, Twofish και Blowfish με μέγεθος κλειδιού 128-bit. Σύμφωνα με τα αποτελέσματα ο AES παρουσίασε τη μικρότερη χρονική καθυστέρηση και ταχύτερη εκτέλεση, κρίθηκε καταλληλότερος για ασφαλή μεταφορά δεδομένων και υπερίσχυσε τόσο σε απόδοση όσο και σε ασφάλεια έναντι των άλλων δύο [9].

Η μελέτη του Patel, προσφέρει μια πιο διαφοροποιημένη εικόνα συγκριτικά με προηγούμενες μελέτες, εξετάζοντας την απόδοση των AES, DES και Blowfish σε μικρά και μεγάλα αρχεία κειμένου. Κύρια συμπεράσματα της έρευνας είναι ότι σε μικρά αρχεία (< 1000 bytes) ο DES έχει καλύτερη απόδοση (εκτελείται ταχύτερα), από την άλλη σε μεγάλα αρχεία (> 10.000 bytes) οι AES, DES και Blowfish παρουσιάζουν παρόμοια απόδοση. Ως προς τις ανάγκες μνήμης (memory requirements), ο Blowfish είναι πιο αποδοτικός, καθιστώντας τον κατάλληλο για περιβάλλοντα με περιορισμένη μνήμη ενώ η επιλογή Blowfish προτείνεται για εφαρμογές με περιορισμούς στη μνήμη [10].

Στη μελέτη τους, οι Buhari et al. πραγματοποίησαν πειραματική σύγκριση των αλγορίθμων AES και Blowfish, αξιολογώντας την απόδοσή τους σε διαφορετικούς τύπους αρχείων (κείμενο, ήχο, εικόνα, βίντεο). Τα αποτελέσματα δείχνουν ότι ο Blowfish υπερτερεί σε ταχύτητα και throughput, ειδικά σε μεγάλα και πολύπλοκα αρχεία, γεγονός που αποδίδεται στην ευελιξία στο μήκος κλειδιού. Παρόλα αυτά, δεν γίνεται αμφισβήτηση της ισχύος του AES ως ασφαλέστερη επιλογή. Οι συγγραφείς υποδεικνύουν ότι το Blowfish είναι πιο αποδοτικό χρονικά, αλλά ο AES παραμένει καταλληλότερος για εφαρμογές με αυστηρές απαιτήσεις ασφαλείας [11].

Άλλη μία ενδιαφέρουσα μελέτη είναι αυτή των Chattaraj et al., [12] με τίτλο "Performance Evaluation of Cryptographic Algorithms on Resource-constrained and GPU/TPU integrated Multi-core Processor Systems" παρέχει μια εκτενή συγκριτική αξιολόγηση των αλγορίθμων DES, 2DES, 3DES, AES και Blowfish, λαμβάνοντας υπόψη την απόδοσή τους σε διαφορετικά περιβάλλοντα και λειτουργικές καταστάσεις (modes) όπως CBC, ECB, CFB, OFB και CTR. Σύμφωνα με τα ευρήματα, ο αλγόριθμος AES, ιδιαίτερα σε λειτουργία CTR, υπερέχει σταθερά έναντι όλων των άλλων ως προς την ταχύτητα και την αποδοτικότητα, ειδικά σε μεγάλα μεγέθη αρχείων (100MB–1GB) και σε σύγχρονα πολυπύρηνα και cloud περιβάλλοντα. Ο Blowfish εμφανίζει καλύτερες επιδόσεις σε περιβάλλοντα περιορισμένων πόρων ή μικρού όγκου δεδομένων, ενώ οι 3DES και 2DES παρουσιάζουν σημαντικές καθυστερήσεις και θεωρούνται πλέον δεν είναι αποτελεσματικοί για απαιτητικές εφαρμογές. Η μελέτη επιβεβαιώνει τον AES ως την πιο κατάλληλη επιλογή για σύγχρονα και απαιτητικά υπολογιστικά περιβάλλοντα, προσφέροντας ιδανική ισορροπία μεταξύ ταχύτητας και ασφάλειας.

Τέλος, η μελέτη των Dibas & Sabri παρέχει μια εκτενή εμπειρική σύγκριση μεταξύ των συμμετρικών αλγορίθμων AES, 3DES, Blowfish και Twofish, με χρήση C# για την εφαρμογή και αξιολόγηση τους βάσει execution time, μνήμης, και ciphertext size σε διάφορα μεγέθη αρχείων (1KB έως 100MB). Τα κύρια συμπεράσματα έχουν ως εξής ο AES ήταν ο πιο αποδοτικός όσον αφορά το χρόνο εκτέλεσης (encryption/decryption), από την άλλη για αρχεία <10MB: ο 3DES ήταν καλύτερος από Blowfish, ενώ για αρχεία >10MB: Blowfish ξεπέρασε τον 3DES. Το Twofish είχε τα χειρότερα αποτελέσματα σε όλους τους τομείς. Ο AES και 3DES χρησιμοποίησαν λιγότερη μνήμη, ενώ Blowfish και Twofish κατανάλωσαν περισσότερη και παρήγαγαν μεγαλύτερο ciphertext [13].

Μελέτη	Αλγόριθμοι	Κριτήρια Σύγκρισης	Μέγεθος Κλειδιού	Γλώσσα Προγραμματισμού	Hardware/ Software	Συμπέρασμα για AES
Manurung et al. (2025)	AES, DES, 3DES, RC6	Ταχύτητα, Ασφάλεια, Απόδοση	128/192/256 bits	Java (μέσω Android SDK)	Android συσκευή	Πιο ανθεκτικός, αποτελεσματικός και γρήγορος
Buhari et al. (2025)	AES, 3DES, Blowfish	Ταχύτητα, Ασφάλεια, Κατανάλωση Μνήμης	128-bits	Java (με custom εφαρμογές EncryptionSpaceMonitor.java & EncryptionTimer.java)	Intel i5-3337U CPU @ 1.80GHz, 4GB RAM, Windows 11 Pro	Καλή ισορροπία απόδοσης-ασφάλειας, πιο ασφαλής
Assa-Agyei & Olajide (2023)	AES, Twofish, Blowfish	Ταχύτητα, Ασφάλεια	128-bits	Python	Laptop με Intel® Core™ i5-10210U @ 2.40GHz, 16 GB RAM, Windows 11 Pro (21H2)	Ταχύτερος και ασφαλέστερος
Patel (2019)	AES, DES, Blowfish	Ταχύτητα, Μνήμη	128/192/256 bits	Java (customized programs namely AESTextEnDnTime.java, DESTextEnDnTime.java and BlowfishTextEnDnTime.java)	Intel (R) Core (TM) i3-3110M CPU 2.4 GHz Windows 7 Professional	Παρόμοια απόδοση με άλλους σε μεγάλα αρχεία
Buhari et al. (2019)	AES, Blowfish	Ταχύτητα, Throughput	128/192/256 bits	Java (Netbeans IDE7.1.2 with default settings in jdk 7.1 development kit for JAVA)	Intel(R) Core(TM) i3 M370 @ 2.40GHz processor with 4 GB RAM on Windows 7 home premium, 6	Ασφαλέστερος, αλλά πιο αργός από Blowfish

Chattaraj et al. (2025)	DES, 2DES, 3DES, AES, Blowfish	Ταχύτητα, Αποδοτικότητα	128 bits	Python 3.11	6 διαφορετικές πλατφόρμες (από Raspberry Pi έως Google Cloud με TPUv2)	Καλύτερος σε CTR mode και μεγάλα αρχεία
Dibas & Sabri (2021)	AES, 3DES, Blowfish, Twofish	Χρόνος εκτέλεσης, Μνήμη, Ciphertext Size	256 bits	C#	LENOVO Ideapad Y700 with Processor Intel(R) Core (TM) i7-6700HQ CPU @ 2.60GHz, 4 Core(s), 8 Logical Processor(s) and 32 GB RAM. Microsoft Windows 10 Enterprise LTSC.	Πιο αποδοτικός σε χρόνο εκτέλεσης

Στον παραπάνω πίνακα παρουσιάζονται συνοπτικα τα αποτελέσματα των πειραματικών ερευνών, όπου φαίνεται καθαρά ότι ο αλγόριθμος AES υπερέχει σε όλα τα επίπεδα σε σύγκριση με άλλους αλγόριθμους κρυπτογράφησης.

Η ΘΕΩΡΙΑ ΤΟΥ AES - ΜΑΘΗΜΑΤΙΚΟ ΥΠΟΒΑΘΡΟ

Ο αλγόριθμος AES αποτελεί έναν ισχυρό εργαλείο κρυπτογράφησης, η υπεροχή του αυτή οφείλεται στο ισχυρό μαθηματικό του υπόβαθρο, το οποίο του προσδίδει υψηλή ασφάλεια, ταχύτητα, αποδοτικότητα, ευελιξία και επεκτασιμότητα. Η χρήση πεπερασμένων σωμάτων ($GF(2^8)$), η σταθερή και επαναλαμβανόμενη δομή των γύρων (SubBytes – ShiftRows – MixColumns – AddRoundKey), καθώς και ο μη γραμμικός πίνακας αντικατάστασης S-Box, που ενισχύει τη μη γραμμικότητα και την ασυμμετρία του συστήματος, αποτελούν βασικά χαρακτηριστικά της ασφάλειας του AES. Τέλος, η απλότητα στην υλοποίηση μέσω σταθερών πινάκων και πράξεων XOR, καθιστά τον AES ιδανικό τόσο για λογισμικό όσο και για υλικό, διατηρώντας τον έως και σήμερα ως ένα από τα πιο αξιόπιστα και ισχυρά κρυπτογραφικά εργαλεία. Παρακάτω παρουσιάζεται αρχικά η θεωρητική μελέτη του αλγορίθμου AES, καθώς και το πλήρες μαθηματικό και λειτουργικό πλαίσιο που διέπει τον αλγόριθμο αυτό. Η ανάλυση συνοδεύεται από ενδεικτικά παραδείγματα και επεξηγηματικές αναλύσεις με στόχο την

κατανόηση της εσωτερικής δομής, των μηχανισμών λειτουργίας και των κρυπτογραφικών διαδικασιών.

ΒΑΣΙΚΕΣ ΜΑΘΗΜΑΤΙΚΕΣ ΕΝΝΟΙΕΣ ΣΤΟΝ AES

Αρχικά, ως block ορίζεται μία ακολουθία bits καθορισμένου σταθερού μήκους, ενώ ως block cipher ορίζεται μία οικογένεια από μεταθέσεις του σταθερού μήκους blocks, παραμετροποιημένα από ένα κλειδί. Το κλειδί είναι μία ακολουθία από bits σταθερού μήκους και παραμένει σταθερό κατά τη διάρκεια των διαδοχικών εκτελέσεων του αλγορίθμου.
Η μαθηματική σχέση που ορίζει τα block cipher απεικονίζεται ως εξής:
$F: K \times \{0,1\}^n \rightarrow \{0,1\}^n$, $F_k(\cdot)$ bijection για κάθε k.

Για κάθε $k \epsilon K$, η συνάρτηση $F_k(x)$, αποτελεί μία αντιστρέψιμη(bijective) απεικόνιση στο σύνολο $\{0,1\}^n$, το οποίο πρακτικά σημαίνει ότι για κάθε έξοδο υπάρχει ακριβώς μία είσοδο [14].

Στον AES, το κλειδί λειτουργεί ως παράμετρος της συνάρτησης κρυπτογράφησης, καθορίζοντας ποια συγκεκριμένη μετάθεση εφαρμόζεται στο block. Το κλειδί είναι μία ακολουθία από bits σταθερού μήκους και παραμένει σταθερό κατά τη διάρκεια των διαδοχικών εκτελέσεων του αλγορίθμου. Όπως αναφέρθηκε και στα προηγούμενα τα διαθέσιμα μήκη κλειδιών στον AES είναι 128, 192 και 256 bits, ενώ το μήκος του block είναι σταθερά 128 bits ανεξαρτήτως κλειδιού[3]. Ως βασική μονάδα στον AES υιοθετείται το 1 byte, το οποίο αντιστοιχεί με 8 bits. Το κάθε bit ορίζεται ως μία ακολουθία μεταξύ δύο παρενθέσεων της μορφής $\{ b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0 \}$, για παράδειγμα {0 1 0 0 0 1 1 0}, με τα bits γράφονται σύμφωνα με την σημαντικότητα τους(MSB), από τα περισσότερα σημαντικά στα λιγότερα σημαντικά. Ο NIST προτείνει την χρήση δεκαεξαδικού συστήματος αρίθμησης, για λόγους συντομίας και άμεσης αναπαράστασης από δυαδικό σε δεκαεξαδικό καθώς και για να διευκολύνει την αλγεβρική επεξεργασία σε πεπερασμένα σώματα(Galois Fields, εδώ 2^8), εφόσον το κάθε byte χωρίζεται σε δύο τετράδες bits b_7, b_6, b_5, b_4 και b_3, b_2, b_1, b_0, από τα περισσότερο στα λιγότερο σημαντικά ψηφία και η απεικόνιση γίνεται με άμεσο τρόπο. Όπως, για παράδειγμα το byte {0 1 0 0 0 1 1 0} χωρίζεται σε δύο τετράδες {0 1 0 0 | 0 1 1 0} και στο δεκαεξαδικό ορίζεται στη μορφή { 4| 6} σύμφωνα με την αντιστοιχία του δεκαεξαδικού σε δυαδικό. Για να υπάρχει διαύγεια στον τρόπο απεικόνισης των δεδομένων καθώς και των κλειδιών χρησιμοποιούνται ακολουθίες 8k bits. Η πληροφορία γενικά απεικονίζεται σε οκτάδες: $r_0, r_1, r_2, \ldots, r_{8k-1}$, $k \epsilon N^*$. Για κάθε byte a_j με $0 \leq j \leq k-1$, ορίζεται το $a_j = \{r_{8j}, r_{8j+1}, r_{8j+2}, \ldots, r_{8j+7}\}$, δηλαδή μία οκτάδες bits τα οποία αντιστοιχούν σε κάθε bytes πληροφορίας, είτε αναφερόμαστε σε δεδομένα είτε σε κλειδιά. Άρα, το κάθε block πληροφορίας απεικονίζεται στη μορφή : $r_0, r_1, r_2, \ldots, r_{127}$, δηλαδή 128 bits ή 16 bytes με:

$a_0 = \{r_0, r_1, r_2, \ldots, r_7\}$,
$a_1 = \{r_8, r_1, r_2, \ldots, r_{15}\}$

και

$a_{15} = \{r_{120}, r_{121}, r_{122}, \ldots, r_{127}\}$.

Ο βασικός εσωτερικός μηχανισμός του AES καθορίζεται από έναν πίνακα 4×4(16 bytes) τα οποία αποτελούν τα bytes του κάθε block πληροφορίας(16 bytes). Κάθε block πληροφορίας διαβάζεται σειριακά, $a_0, a_1, a_2, \ldots, a_{15}$ και τοποθετούνται στον πίνακα κατακόρυφα σε στήλες δημιουργώντας τον πίνακα input bytes, ο οποίος είναι ως εξής:

in[r,c]=

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

Όπου r οι γραμμές και c οι στήλες του πίνακα, με $0 \leq r < 4$ και $0 \leq c < 4$. Άρα, σύμφωνα με τα παραπάνω ορίζεται ότι: $in_0 = a_0$, $in_4 = a_4$,..., $in_{15} = a_{15}$.
Με είσοδο τον πίνακα input bytes και σύμφωνα με τη σχέση s[r,c]=in[r+4c] δημιουργείται ο πίνακας state:

s[r,c]=

$s(0,0)$	$s(0,1)$	$s(0,2)$	$s(0,3)$
$s(1,0)$	$s(1,1)$	$s(1,2)$	$s(1,3)$
$s(2,0)$	$s(2,1)$	$s(2,2)$	$s(2,3)$
$s(3,0)$	$s(3,1)$	$s(3,2)$	$s(3,3)$

Σύμφωνα με τα παραπάνω: s[0,0]=in[0+4·0]=$in_0 = a_0$, s[0,1]=in[0+4· 1]=$in_4 = a_4$, …, s[3,3]=in[3+4·3]=$in_{15} = a_{15}$.
Ως λέξη(word) ορίζεται μία ακολουθία από 4 κατακόρυφα bytes, με κάθε block να ορίζεται από 4 λέξεις.

$v_0 =$

$s(0,0)$
$s(1,0)$
$s(2,0)$
$s(3,0)$

$v_1 =$

$s(0,1)$
$s(1,1)$
$s(2,1)$
$s(3,1)$

$v_2 =$

$s(0,2)$
$s(1,2)$
$s(2,2)$
$s(3,2)$

$v_3 =$

$s(0,2)$
$s(1,2)$
$s(2,2)$
$s(3,2)$

ΣΥΝΟΛΑ GALOIS

Όπως προαναφέρθηκε ο AES αλγόριθμος χρησιμοποιεί πεπερασμένα σώματα(GF) και συγκεκριμένα το $GF(2^8)$. Πριν παρουσιαστεί η υλοποίηση αυτού του πεδίου, είναι χρήσιμο να δοθούν ορισμένοι βασικοί ορισμοί από τη θεωρία πεπερασμένων σωμάτων καθώς και οι πράξεις στο σύνολο αυτό, ώστε να καταστεί σαφής η σημασία και η χρήση τους στα κρυπτογραφικά συστήματα όπως ο AES.

ΟΡΙΣΜΟΣ ΠΕΠΕΡΑΣΜΕΝΟΥ ΣΩΜΑΤΟΣ $GF(p^n)$

Έστω p ένας πρώτος αριθμός και n≥1 ένας ακέραιος. Ένα **πεπερασμένο σώμα** τάξης p^n, που συμβολίζεται ως F_{P^n} ή $GF(p^n)$, είναι ένα σύνολο από p^n αντικείμενα και δύο δυαδικές πράξεις, πρόσθεση και πολλαπλασιασμό, έτσι ώστε να ισχύουν οι ακόλουθες ιδιότητες:

1. Τα στοιχεία είναι κλειστά ως προς την πρόσθεση modulo p.
2. Τα στοιχεία είναι κλειστά ως προς τον πολλαπλασιασμό modulo p.
3. Για κάθε μη μηδενικό στοιχείο, υπάρχει πολλαπλασιαστικό αντίστροφο.

Με τον όρο κλειστό εννοούμε ότι αν πάρεις οποιαδήποτε δύο στοιχεία από το σύνολο και εφαρμόσεις την πράξη (πρόσθεση ή πολλαπλασιασμό, εδώ), το αποτέλεσμα παραμένει μέσα στο ίδιο σύνολο [15].

ΤΟ ΠΕΠΕΡΑΣΜΕΝΟ ΣΩΜΑ $GF(2^8)$

Στον AES χρησιμοποιείται το πεπερασμένο σώμα $GF(2^8)$ το οποίο έχει 256 στοιχεία. Όλα του τα στοιχεία του αναπαρίστανται ως πολυώνυμα βαθμού < 8 με συντελεστές στο GF(2) (δηλ. bits). Η πρόσθεση είναι bitwise XOR και ο πολλαπλασιασμός είναι πολλαπλασιασμός πολυωνύμων με μείωση modulo ένα μη αναγώγιμο πολυώνυμο βαθμού 8, τα οποία αναλύονται παρακάτω.

Τα στοιχεία του GF μπορούν να αναπαρασταθούν από πολυώνυμα το πολύ μέχρι εβδόμου βαθμού, με κάθε στοιχείο να μπορεί να πάρει τη μορφή:

$a(x) = a_7 \cdot x^7 + a_6 \cdot x^6 + a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$, με $a_i \epsilon \{0\ 1\}$.

Άρα, κάθε bit μπορεί να απεικονιστεί ως πολυώνυμο με την χρήση των πεπερασμένων σωμάτων. Για παράδειγμα: Το bit d = {1 0 1 1 0 0 1 0}, ορίζεται ως το πολυώνυμο $a(x) = 1 \cdot x^7 + 1 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x$. Οι πράξεις της πρόσθεσης και του πολλαπλασιασμού μεταξύ των στοιχείων του $GF(2^8)$ αποτελούν βασική θεωρία του σώματος και ορίζονται ως εξής:

ΠΡΟΣΘΕΣΗ: Η πρόσθεση στο $GF(2^8)$ ορίζεται ως μία πράξη XOR κατά bit πληροφορίας (bitwise) σύμφωνα με τους κανόνες της XOR, δηλαδή ισχύει ότι: $1 \oplus 1 = 0, 1 \oplus 0 = 1, 0 \oplus 1 = 1, 0 \oplus 0 = 0$.

Για παράδειγμα έστω τα πολυώνυμα: $a(x) = 1 \cdot x^7 + 1 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x$ και $b(x) = 1 \cdot x^7 + 1 \cdot x^6 + 1 \cdot x^4 + 1 \cdot x^3 + 1 \cdot x + 1$,

τότε η πρόσθεση θα ορίζεται ως εξής $c(x) = a(x) + b(x) = 1 \cdot x^6 + 1 \cdot x^5 + 1 \cdot x^3 + 1$.

Αν γίνει η πράξη και στην δυαδική μορφή τότε θα ορίζεται ως εξής:

$a = \{1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\}$

$b = \{1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\}$

$c = a + b = \{0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\}$, η XOR πραγματοποιείται σε αντιστοιχία bit προς bit, δηλαδή $c(i) = a(i) + b(i)$, με $0 \leq i \leq 7$.

ΠΟΛΛΑΠΛΑΣΙΑΣΜΟΣ: Στον πολλαπλασιασμό στο $GF(2^8)$ χρησιμοποιείται το σύμβολο ·. Ο πολλαπλασιασμός $d = b \cdot c$ γίνεται ως κλασικός πολλαπλασιασμός μεταξύ πολυωνύμων και στο γινόμενο γίνεται mod με το πολυώνυμο $m(x) = x^8 + x^4 + x^3 + x + 1$, που αποτελεί ένα μη αναγώγιμο πολυώνυμο και είναι κατάλληλο για πολλαπλασιασμό στο $GF(2^8)$ για να παραμείνει το αποτέλεσμα στο σώμα. Ο NIST προτείνει την χρήση της συνάρτησης xTIMES(b) για να πραγματοποιήσει τον πολλαπλασιασμό, η οποία είναι ιδιαίτερα χρήσιμη και αποδοτική σε σχέση με τον κλασικό πολλαπλασιασμό.

Αν $c(x) = \{02\}$ ή x, τότε το γινόμενο σύμφωνα με την συνάρτηση $xTIMES(b)$ ορίζεται ως εξής:

Αν b_7=0 τότε κάνουμε απλή ολίσθηση προς τα αριστερά στο b και το γινόμενο που προκύπτει είναι: $d = \{ b_6 b_5 b_4 b_3 b_2 b_1 b_0 0 \}$.

Αν b_7=1 τότε κάνουμε XOR του ολισθιμένου στοιχείου $\{ b_6 b_5 b_4 b_3 b_2 b_1 b_0 0 \}$ με το $\{ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1 \}$.

Ιδιότητες:

Ουδέτερο στοιχείο: ως ουδέτερο στοιχείο του πολλαπλασιασμού στο $GF(2^8)$ ορίζεται το {01} ή $\{ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \}$, και ισχύει ότι $b \cdot \{01\} = b$.

Επιμεριστική ιδιότητα του πολλαπλασιασμού ως προς XOR: ισχύει ότι $a \cdot (b \oplus c) = (a \oplus b) \cdot (a \oplus c)$.

Με την χρήση της συνάρτησης $xTIMES(b)$ και των ιδιοτήτων του XOR στην ομάδα $GF(2^8)$ μπορούν να γίνουν οι πράξεις.

Έστω $b = \{32\}$ ή $\{ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \}$. Τότε οι πράξεις έχουν ως εξής :

$\{32\} \cdot \{01\} = \{32\}$ ή $\{ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \} \cdot \{ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \} = \{ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \}$.

$\{32\} \cdot \{02\} = xTIMES(\{32\}) = \{64\}$ ή $\{ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \} \cdot \{ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \} = \{ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 0 \}$.

Εφόσον το b_7=0 πραγματοποιείται απλή ολίσθηση αριστερά και προκύπτει $\{ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 0 \}$.

$\{32\} \cdot \{03\} = \{32\} \cdot (\{01\} \oplus \{02\}) = (\{32\} \cdot \{01\}) \oplus (\{32\} \cdot \{02\}) = \{32\} \oplus \{64\} = \{56\}$

Εφόσον, οι πράξεις πραγματοποιούνται στο $GF(2^8)$ ισχύει ότι οι δυνάμεις του 2, δηλαδή τα δεδομένα {02},{04},{08},{10} κλπ, χρησιμοποιούν την ιδιότητα: $b \cdot \{04\} = b \cdot \{02\}^2$, $b \cdot \{08\} = b \cdot \{02\}^3$, $b \cdot \{10\} = b \cdot \{02\}^4$, όμοια και τα υπόλοιπα [2].

Προκύπτει λοιπόν ότι:

$$\{32\} \cdot \{04\} = xTIMES(xTIMES(32)) = xTIMES(\{56\}) = \{ac\}.$$

$\{32\} \cdot \{05\} = \{32\} \cdot (\{04\} \oplus \{01\}) = (\{32\} \cdot \{04\}) \oplus (\{32\} \cdot \{01\}) = \{ac\} \oplus \{32\} = \{9E\}$.

Κάνοντας διαδοχικά xTIMES μπορούν να υπολογιστούν σύντομα όλες οι τιμές, που θα χρειαστούν στα επόμενα βήματα του AES.

ΠΟΛΛΑΠΛΑΣΙΑΣΜΟΣ ΤΩΝ ΛΕΞΕΩΝ ΜΕ ΧΡΗΣΗ ΣΤΑΘΕΡΟΥ ΠΙΝΑΚΑ:

Ο AES ορίζει δύο μετατροπές στους πίνακες που ορίζονται από το διάνυσμα των λέξεων $v = [v_0\ v_1\ v_2\ v_3]$ και προκύπτει σαν έξοδο το διάνυσμα λέξεων $d = [d_0\ d_1\ d_2\ d_3]$ [1].

Και προκύπτει:

$$d_0 = (a_0 \bullet v_0) \oplus (a_3 \bullet v_1) \oplus (a_2 \bullet v_2) \oplus (a_1 \bullet v_3)$$
$$d_1 = (a_1 \bullet v_0) \oplus (a_0 \bullet v_1) \oplus (a_3 \bullet v_2) \oplus (a_2 \bullet v_3)$$
$$d_2 = (a_2 \bullet v_0) \oplus (a_1 \bullet v_1) \oplus (a_0 \bullet v_2) \oplus (a_3 \bullet v_3)$$
$$d_3 = (a_3 \bullet v_0) \oplus (a_2 \bullet v_1) \oplus (a_1 \bullet v_2) \oplus (a_0 \bullet v_3)$$

Βάση της θεωρίας υπολογίζεται ο πολλαπλασιαστικός αντίστροφος ενός $w \neq \{00\}$ και υπάρχει μοναδικός v^{-1} τέτοιος ώστε $v \cdot v^{-1} = \{01\}$ και ισχύει ότι $v^{-1} = v^{254}$.
Διαφορετικά, μπορεί να υπολογιστεί με την χρήση των πολυωνυμικών συναρτήσεων, όπου σύμφωνα με τον εκτεταμένο αλγόριθμο του Ευκλείδη ισχύει ότι: $v(x)a(x) + m(x)c(x) = 1$, όπου $m(x) = x^8 + x^4 + x^3 + x + 1$ και ως a(x) ορίζεται ο αντίστροφος του w(x) ως πολυώνυμο, δηλαδή αναπαριστά το αντίστροφο byte v^{-1}.

ΟΙ ΣΥΝΑΡΤΗΣΕΙΣ ΣΤΟΝ AES

Οι βασικές συναρτήσεις οι οποίες χρησιμοποιούνται στον AES συμβολίζονται ως εξής: CIPHER() και INVCIPHER(), πρακτικά ορίζουν την κρυπτογράφηση και την αποκρυπτογράφηση. Η κεντρική ιδέα του AES βασίζεται σε μία ακολουθία διαδοχικών μετατροπών που ονομάζονται γύροι. Κάθε γύρος δέχεται ως είσοδο ένα round key, το οποίο έχει μέγεθος 4 words(16 bytes) και προκύπτει από τον αλγόριθμο key expansion.

Η ΣΥΝΑΡΤΗΣΗ KEYEXPANSION

Ο KEYEXPANSION() δέχεται ως είσοδο το block cipher key και ως έξοδο έχει όλα τα round keys τα οποία θα χρησιμοποιηθούν σε όλους τους γύρους του AES. Το σύνολο των λέξεων που προκύπτουν ορίζεται ως key schedule. Το κάθε μέλος της οικογένειας AES απαιτεί διαφορετικό πλήθος μέγεθος κλειδιού, διαφορετικό πλήθος γύρων και ο αλγόριθμος KEYEXPANSION() παράγει διαφορετικό πλήθος λέξεων, όπως παρουσιάζεται στον Πίνακα 1.

AES Variant	**Κλειδί (Nk words)**	**Γύροι (Nr)**	**Συνολικά Words ()**
AES-128	4 (4x32=128)	10	44
AES-192	6 (6x32=192)	12	52
AES-256	8 (8x32=256)	14	60

ΠΙΝΑΚΑΣ 1

ΠΕΡΙΓΡΑΦΗ ΤΗΣ KEYEXPANSION():

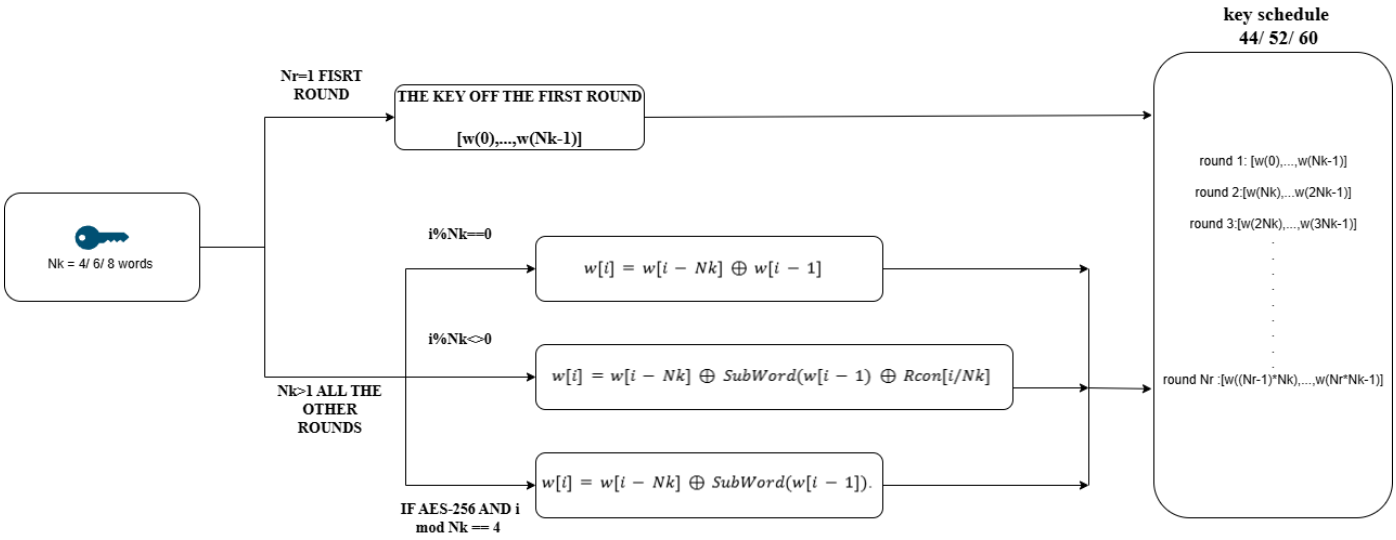
Συνολικά δημιουργούνται 4 × (Nr + 1) λέξεις (32-bit η κάθε μία), από το αρχικό κλειδί του AES. Οι Round Constants ορίζονται ως 10 σταθερές λέξεις των 32-bits που χρησιμοποιούνται για να διαφοροποιήσουν τα κλειδιά του κάθε γύρου. Ισχύει ότι: $Rcon[j] = [\, x^{j-1}, 00, 00, 00]$, $0 < j \leq Nr$. Για κάθε $j > 1$ ισχύει ότι $Rcon[j] = XTIMES(Rcon[j-1])$. Ο πίνακας αυτών των λέξεων έχει ως εξής:

j	Rcon[j]
1	[01, 00, 00, 00]
2	[02, 00, 00, 00]
3	[04, 00, 00, 00]
4	[08, 00, 00, 00]
5	[10, 00, 00, 00]
6	[20, 00, 00, 00]
7	[40, 00, 00, 00]
8	[80, 00, 00, 00]
9	[1b, 00, 00, 00]
10	[36, 00, 00, 00]

1. Διατηρεί τις πρώτες λέξεις(w[0]...w[Nk-1]), 4, 6 και 8 αντίστοιχα με AES-128, AES-196, AES-256, τα πρώτα Nk στοιχεία αποτελούν το αρχικό κλειδί.
2. Για τις υπόλοιπες λέξεις, οι οποίες χρησιμοποιούνται στις επαναληπτικές λειτουργίες του AES γίνεται το εξής:
    a. Αν $i \ mod \ Nk \ \neq 0$ τότε εφαρμόζει: $w[i] = w[i - Nk] \oplus w[i - 1]$
    b. Αν $i \ mod \ Nk = 0$ τότε ισχύει: $w[i] = w[i - Nk] \oplus SubWord(w[i - 1]) \oplus Rcon[i/Nk]$

Στην περίπτωση του AES-256 ισχύει επιπλέον ότι αν i mod Nk == 4 (δηλ. i πολλαπλάσιο του 4 αλλά όχι του Nk). $w[i] = w[i - Nk] \oplus SubWord(w[i - 1])$.

Στην παρακάτω εικόνα παρουσιάζεται το διάγραμμα ροής της διαδικασίας Key Schedule του αλγορίθμου AES, το οποίο περιγράφει αναλυτικά τη δημιουργία των υποκλειδιών (round keys) που χρησιμοποιούνται σε κάθε γύρο του αλγορίθμου, για όλες τις παραλλαγές AES-128, AES-192 και AES-256.



ΕΙΚΟΝΑ 1: Διάγραμμα λειτουργίας του Key Expansion στον αλγόριθμο AES.

Στην διαδικασία της αποκρυπτογράφησης χρησιμοποιούμε την ίδια συνάρτηση.

ΟΙ ΣΥΝΑΡΤΗΣΕΙΣ SUBBYTES / INVSUBBYTES

Η συνάρτηση SUBBYTES εφαρμόζεται κατά τη διαδικασία της κρυπτογράφησης και προκύπτει εφαρμόζοντας δύο διαδοχικά βήματα, όπως παρουσιάζονται παρακάτω και πραγματοποιεί μη γραμμική υποκατάσταση σε κάθε byte του state, μετατρέποντας κάθε byte b σε ένα νέο byte.

Βήμα 1: Υπολογισμός του αντίστροφου στο GF(2^8). Το byte b θεωρείται ως στοιχείο του πεπερασμένου σώματος $GF(2^8)$. Υπολογίζεται το πολλαπλασιαστικό του αντίστροφο(b'), δηλαδή το $b^{-1} mod\, m(x)$), όπου το μηδέν αντιστοιχεί στον εαυτό του (δηλαδή SBOX(0x00)=0x63).

Δηλαδή: αν b ≠ {00}, τότε βρίσκουμε τον αντίστροφο του b στο πεπερασμένο σώμα GF(2^8), αλλιώς b = {00}, τότε δεν έχει και το αφήνουμε ως έχει.

Βήμα 2: Affine Μετασχηματισμός (bitwise πράξη με XOR)
Στον αντίστροφο εφαρμόζεται μια γραμμική μετατροπή + XOR με το σταθερό byte c={01100011}.

Αυτό το βήμα υλοποιείται ως εξής:

$b_0[i] = b_i{}' \oplus b'_{(i+4)mod8}{}' \oplus b'_{(i+5)mod8} \oplus' b'_{(i+6)mod8} \oplus' b'_{(i+7)mod8} \oplus c_i{}'$ για i =0,...,7 και $c = \{0\ 1\ 1\ 0\ 0\ 0\ 1\ 1\}$.

Και σύμφωνα με αυτή τη σχέση ότι:

$$SBOX(b) = M \cdot b' \oplus c$$

Όπου M =

1	0	0	0	1	1	1	1	1
1	1	0	0	0	1	1	1	1
1	1	1	0	0	0	1	1	0
1	1	1	1	0	0	0	1	0
1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	0	1
0	0	0	1	1	1	1	1	0

Τελικά τα στοιχεία του SBOX ορίζονται ως εξής:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
4	9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
5	53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	3	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Με τον παραπάνω πίνακα μπορούμε να κάνουμε εύκολα την μετατροπή SUBBYTES για τα στοιχεία που εισάγονται.

Για παράδειγμα, έστω ότι:

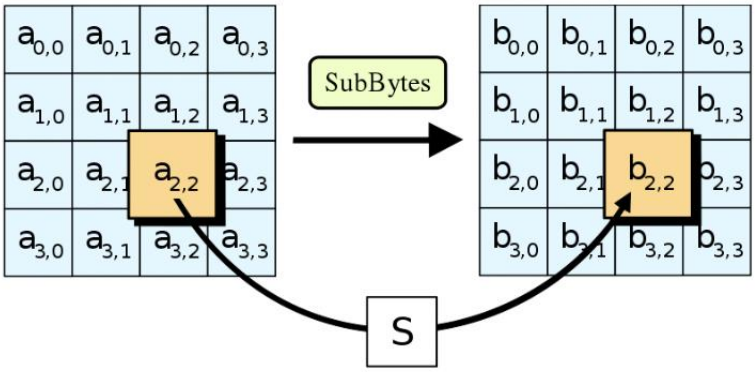
input_state =

19	a0	9a	e9
3d	f4	c6	f8
e3	e2	8d	48
be	2b	2a	8

Τότε το output_state, σύμφωνα με το SBOX γίνεται :

d4	e0	b8	1e
27	bf	b4	41
11	98	5d	52
ae	f1	e5	30

Το οποίο προκύπτει input_state(1,1)=19 σύμφωνα με τον πίνακα SBOX προκύπτει ότι output_state(1,1)=d4.



[16]

Η συνάρτηση INVSUBBYTES εφαρμόζεται κατά τη διαδικασία της αποκρυπτογράφησης και προκύπτει εφαρμόζοντας δύο διαδοχικά βήματα, όπως παρουσιάζονται παρακάτω και πραγματοποιεί μη γραμμική υποκατάσταση σε κάθε byte του state, μετατρέποντας κάθε byte b σε ένα νέο byte.

Βήμα 1: Αντίστροφος Affine Μετασχηματισμός (bitwise πράξη με XOR) : Αναιρείται ο αφινικός μετασχηματισμός που εφαρμόζεται στην κρυπτογράφηση.

Βήμα 2: Εφαρμογή της αντίστροφης πράξης στο σώμα $GF(2^8)$ (multiplicative inverse). Κάθε byte αντικαθίσταται με το πολλαπλασιαστικό του αντίστροφο στο $GF(2^8)$, εκτός από το 00.

Το SBOX της αποκρυπτογράφησης παρουσιάζεται παρακάτω:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB

1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
a	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
b	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
c	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
d	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
e	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
f	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

ΟΙ ΣΥΝΑΡΤΗΣΕΙΣ ShiftRows / InvShiftRows

Πραγματοποιείται μετατροπή σύμφωνα με τη σχέση $s'_{r,c} = s_{r,(c+r)mod4}$, $0 \leq r < 4, και\ 0 \leq r < 4$. Εφαρμόζεται στον πίνακα state και μετατοπίζει κυκλικά τις γραμμές του πίνακα προς τα αριστερά. Σύμφωνα με την σχέση $s'_{r,c} = s_{r,(c+r)mod4}$, η μετατόπιση δεν επηρεάζει την πρώτη γραμμή, ενώ οι υπόλοιπες γραμμές μετακινούνται κυκλικά ως εξής:

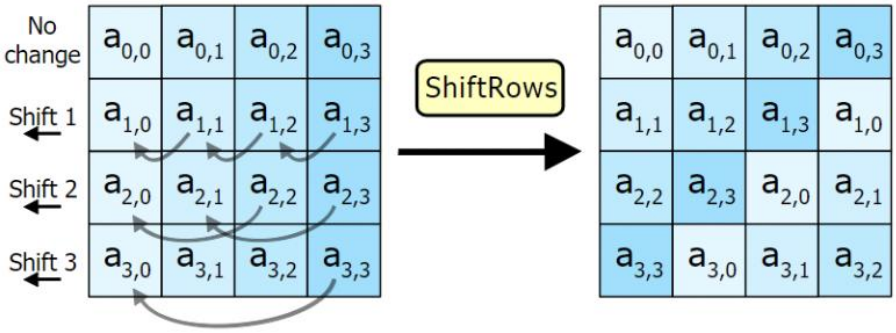
Γραμμή	**Μετατόπιση**
1η	Καμία
2η	1 θέση
3η	2 θέσεις
4η	3 θέσεις

Αν ως είσοδος οριστεί το παρακάτω:

d4	e0	b8	1e
27	bf	b4	41
11	98	5d	52
ae	f1	e5	30

Το output_ShiftRows προκύπτει:

d4	e0	b8	1e
bf	b4	41	27
5d	52	11	98
30	ae	f1	e5



[16]

Κατά τη διαδικασία της αποκρυπτογράφησης χρησιμοποιείται η συνάρτηση **InvShiftRows**, η οποία αποτελεί την αντίστροφη λειτουργία της ShiftRows της κρυπτογράφησης. Η InvShiftRows εφαρμόζεται στον πίνακα state (4×4 bytes) και πραγματοποιεί κυκλική μετατόπιση προς τα δεξιά στις γραμμές, αποκαθιστώντας τη διάταξη των byte πριν από τη ShiftRows. Αρχικά, η 1η γραμμή (γραμμή 0) δεν μετατοπίζεται, έπειτα η 2η γραμμή (γραμμή 1) μετατοπίζεται κυκλικά κατά 1 θέση προς τα δεξιά, η 3η γραμμή (γραμμή 2) μετατοπίζεται κυκλικά κατά 2 θέσεις προς τα δεξιά και η 4η γραμμή (γραμμή 3) μετατοπίζεται κυκλικά κατά 3 θέσεις προς τα δεξιά.

ΟΙ ΣΥΝΑΡΤΗΣΕΙΣ MixColumn / InvMixColumn

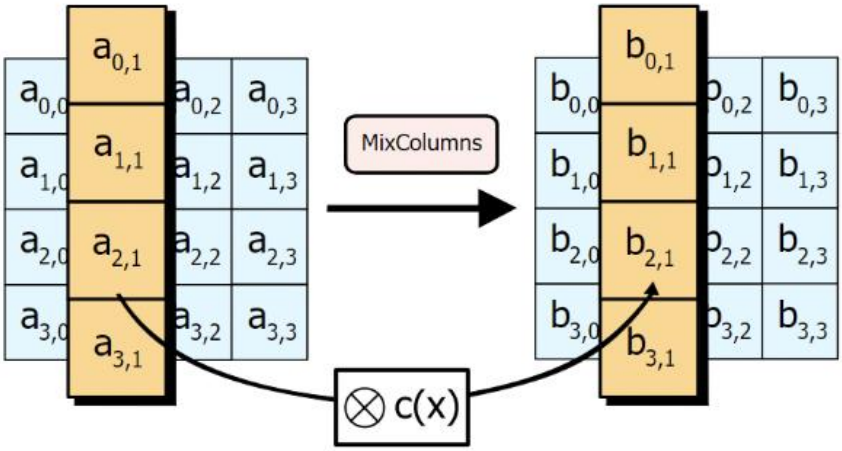
Ως $a = [a_0\, a_1\, a_2\, a_3\ ]$, με a_i σταθερές προκαθορισμένες από την δομή του AES και ορίζονται ως $a = [a_0\, a_1\, a_2\, a_3\ ] = [\{02\}, \{01\}, \{01\}, \{03\}]$, οι οποίες δημιουργούν τον πίνακα A:

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Η πράξη ορίζεται σύμφωνα με τη σχέση $s' = A \cdot s$, όπως ορίζεται ο πολλαπλασιασμός στο $G(2^8), με\ XOR\ και\ XTIMES$ συνάρτηση.

output_mixcolumns(s)=

48	28	b2	2b
f8	6	55	f9
d3	26	82	a1
7a	4c	b7	3b



[16]

Κατά τη διαδικασία της αποκρυπτογράφησης χρησιμοποιείται η συνάρτηση **InvMixColumns**, η οποία αποτελεί την αντίστροφη λειτουργία της MixColumns που εφαρμόζεται στην κρυπτογράφηση. Η συνάρτηση εφαρμόζει έναν **αντίστροφο γραμμικό μετασχηματισμό** στο σώμα **GF(2^8)**. Για κάθε στήλη του state, η InvMixColumns πραγματοποιεί πολλαπλασιασμούς και XOR με σταθερές τιμές, σύμφωνα με το εξής αντίστροφο πολυώνυμο:

0e	**0b**	**0d**	**09**
09	0e	0b	0d
0d	09	0e	0b
0b	0d	09	0e

ΟΙ ΣΥΝΑΡΤΗΣΕΙΣ AddRoundKey / InvAddRoundKey

Η AddRoundKey είναι μια πράξη XOR που εφαρμόζεται μεταξύ του state πίνακα και του αντίστοιχου round key, όπως προκύπτουν από το key schedule, σύμφωνα με τον τύπο:
$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{(4\star round+c)}]$, με $0 \leq c < 4$.
Έστω ότι έχουμε ένα roundkey:

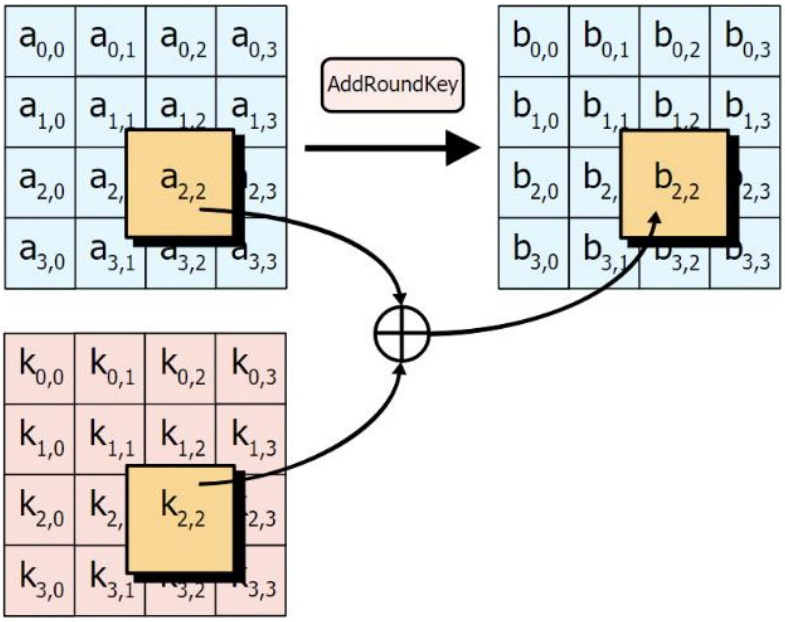
a4	ee	a2	cf
1a	9f	ba	f6
b6	d4	ea	0c
3f	b7	1f	49

Γίνεται XOR με το state byte προς byte με το state:

48	28	b2	2b
f8	6	55	f9
d3	26	82	a1
7a	4c	b7	3b

Και το αποτέλεσμα προκύπτει ως εξής:

ec	c6	10	e4
e2	99	ef	0f
65	f2	68	ad
45	fb	a8	72



[16]

Η συνάρτηση **InvAddRoundKey** εκτελεί bitwise πράξη XOR ανάμεσα στο state και το round key, όπως παρουσιάζεται στον παρακάτω τύπο:

$[s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] = [s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] \oplus [w'_{(4\star round+c)}]$, με $0 \leq c < 4$. Οι συναρτήσεις πραγματοποιούν ακριβώς την ίδια διαδικασία, εφαρμόζοντας την αντίστροφη σειρά των κλειδιών όπως αυτά έχουν δημιουργηθεί στο key schedule.

Η ΔΟΜΗ ΤΟΥ AES

ΔΙΑΔΙΚΑΣΙΑ ΚΡΥΠΤΟΓΡΑΦΗΣΗΣ (CIPHER)

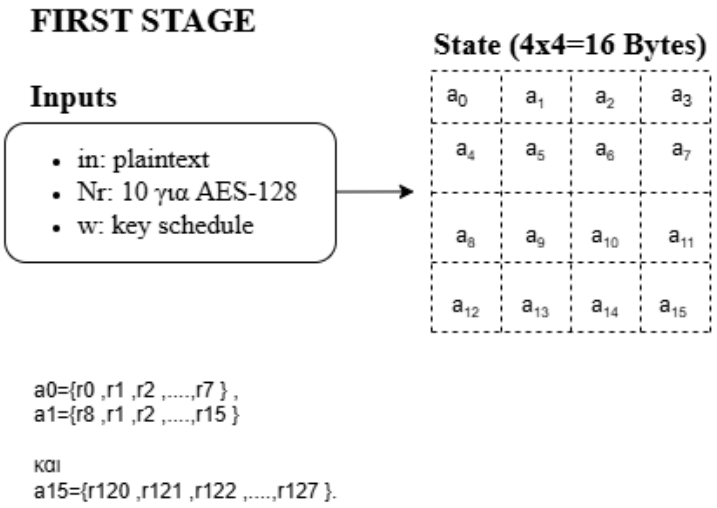
ΠΕΡΙΓΡΑΦΗ ΤΩΝ ΜΕΤΑΤΡΟΠΩΝ:

Κάθε γύρος χαρακτηρίζεται από τέσσερις μετατροπές οι οποίες είναι οι εξής: SubBytes(), ShiftRows(), MixColumns() και AddRoundKey().

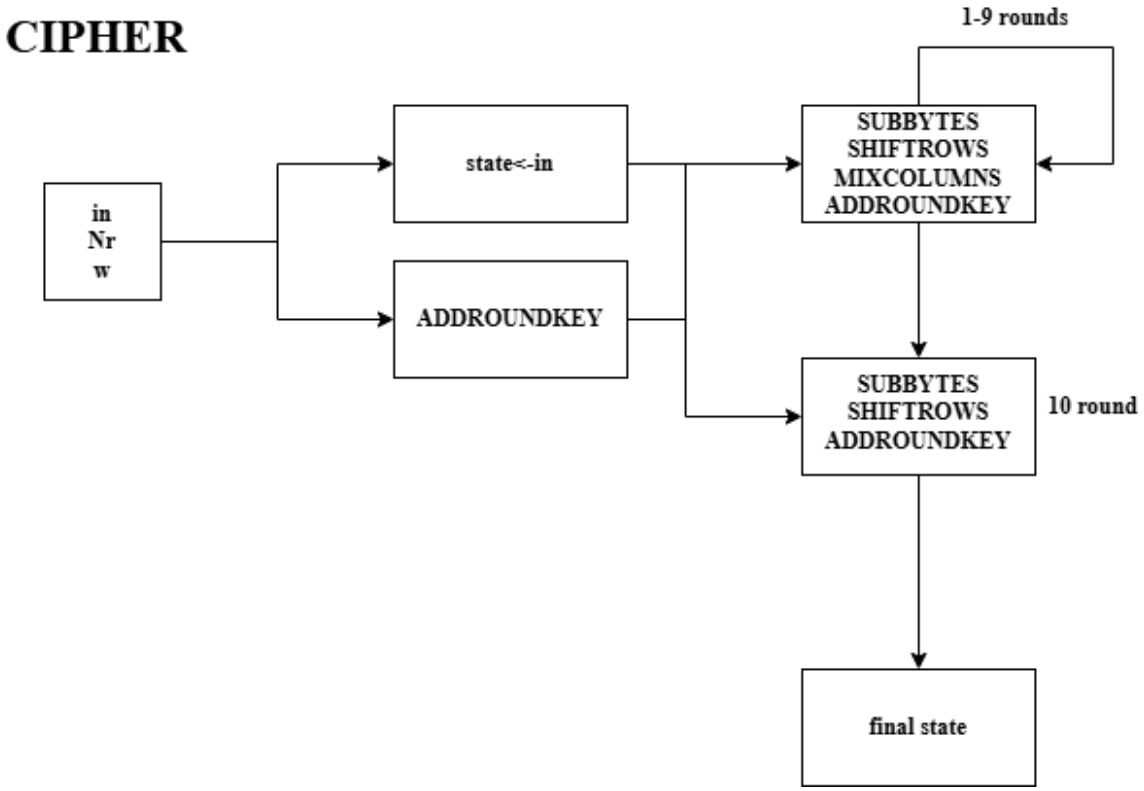
Η διαδικασία κρυπτογράφησης ενός block 128-bit δεδομένων με βάση τον αριθμό γύρων Nr και το key schedule w, που έχει παραχθεί μέσω της KeyExpansion(), περιγράφεται ως εξής:

Ονομάζουμε CIPHER(in, Nr, w) τη συνάρτηση του AES.

Ως Είσοδοι ορίζονται τα παρακάτω: in: το αρχικό block εισόδου (plaintext), Nr: αριθμός γύρων (10 για AES-128, 12 για AES-192, 14 για AES-256), w: το key schedule - πίνακας λέξεων (words) με τα round keys.



Η είσοδος αντιγράφεται στη μεταβλητή state, που είναι ο εσωτερικός πίνακας εργασίας (4x4 bytes), δηλαδή state=v. Στον πρώτο γύρο πραγματοποιείται μόνο τροποποίηση του αρχικού state μέσω του AddRoundKey, και πιο συγκεκριμένα εφαρμόζει bitwise XOR μεταξύ του state και των λέξεων v_0, v_1, v_2, v_3 . Δηλαδή: state[i][j]:=state[i][j]⊕key[i][j]. Πραγματοποιείται XOR στοιχείο προς στοιχείο στους δύο πίνακες. Ακολουθούν διαδοχικές μετατροπές από τις συναρτήσεις SubBytes(), ShiftRows(), MixColumns() και AddRoundKey(), για τους επόμενους γύρους εκτός από τον τελευταίο(εδώ πραγματοποιούνται 9 διαδοχικές μετατροπές). Στον τελευταίο γύρο πραγματοποιούνται όλες οι μετατροπές εκτός από το MixColumns()(εδώ νοείται ο δέκατος γύρος). Ως έξοδος ορίζεται το καινούργιο τροποποιημένο state. Το διάγραμμα ροής του AES-128 (10 βήματα) παρουσιάζεται παρακάτω στην εικόνα 2:



ΕΙΚΟΝΑ 2: ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΚΡΥΠΤΟΓΡΑΦΗΣΗ AES-128 10 ΓΥΡΟΙ

ΔΙΑΔΙΚΑΣΙΑ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗΣ (INVCIPHER)

ΠΕΡΙΓΡΑΦΗ ΤΩΝ ΜΕΤΑΤΡΟΠΩΝ

Κατά την αποκρυπτογράφηση, ο αλγόριθμος AES χρησιμοποιεί τις αντίστροφες μετατροπές των τεσσάρων βασικών λειτουργιών: **InvSubBytes(), InvShiftRows(), InvMixColumns() και AddRoundKey().**

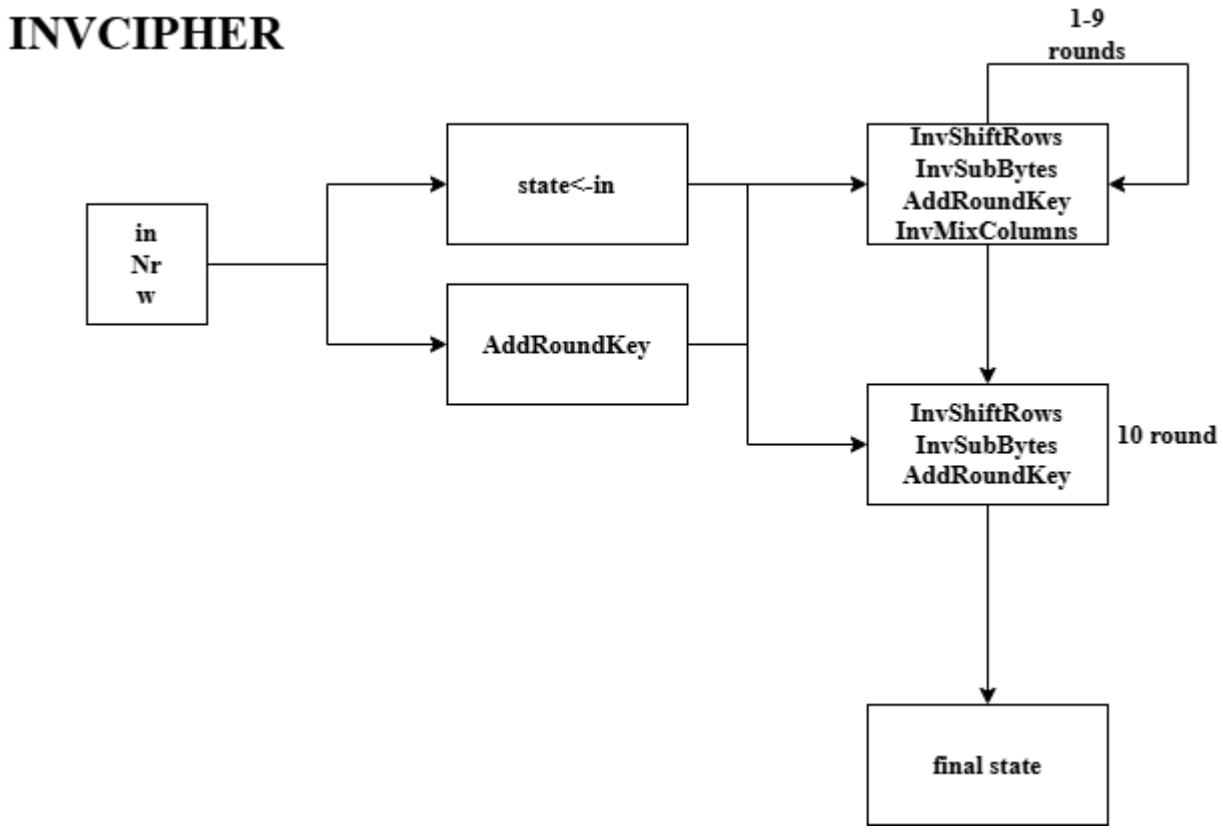
Η συνάρτηση **INVCIPHER** δέχεται:

- **in**: το block εισόδου (ciphertext) των 128 bit
- **Nr**: αριθμός γύρων (10 για AES-128, 12 για AES-192, 14 για AES-256)
- **w**: το key schedule, ένας πίνακας από λέξεις (words) που περιέχουν τα round keys και έχουν παραχθεί μέσω της KeyExpansion()

Το block εισόδου αντιγράφεται στον πίνακα state, έναν εσωτερικό πίνακα 4×4 bytes: instate=in.

Ο πρώτος γύρος της αποκρυπτογράφησης πραγματοποιεί μόνο την πράξη **AddRoundKey**, αλλά χρησιμοποιεί το **τελευταίο** `round key`: `state[i][j]:=state[i][j]⊕key[i][j]`.

Για τους επόμενους γύρους (δηλ. 9 γύρους στην περίπτωση AES-128) εφαρμόζονται διαδοχικά οι εξής αντίστροφες λειτουργίες, η **InvShiftRows()** – κυκλική μετατόπιση γραμμών προς τα δεξιά, ακολουθεί η **InvSubBytes()** – μη γραμμική υποκατάσταση μέσω inverse S-box, έπειτα εφραμόζεται **AddRoundKey()** – XOR με το κατάλληλο round key και τέλος πραγματοποιείται **InvMixColumns()** – αντίστροφος γραμμικός μετασχηματισμός στο GF(2^8). Στον τελευταίο γύρο πραγματοποιούνται όλες οι μετατροπές εκτός από το MixColumns()(εδώ νοείται ο δέκατος γύρος). Το διάγραμμα ροής του AES-128 (10 βήματα) παρουσιάζεται παρακάτω στην εικόνα 3:



ΕΙΚΟΝΑ 3: ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ AES-128 10 ΓΥΡΟΙ

Η παρούσα εργασία στοχεύει αρχικά στην αξιολόγηση της απόδοσης διαφορετικών αλγορίθμων κρυπτογράφησης μετρώντας τους χρόνους κρυπτογράφησης και αποκρυπτογράφησης σε διάφορα μεγέθη αρχείων και διαμορφώσεις συστήματος και

δευτερευόντως στη διεξαγωγή αξιολόγησης, ώστε να προσδιοριστούν ποιοι αλγόριθμοι είναι οι καταλληλότεροι για εφαρμογές πραγματικού κόσμου, όπως η ασφάλεια αποθήκευσης σε περιβάλλον cloud, όπως αυτά των χρηματοοικονομικών συναλλαγών και των συστημάτων επικοινωνίας.

Η επόμενη ενότητα περιγράφει το πειραματικό δοκιμαστικό πλαίσιο που χρησιμοποιήθηκε για τη σύγκριση της απόδοσης διαφορετικών αλγορίθμων κρυπτογράφησης και την παρουσίαση των αποτελεσμάτων με βάση διαφορετικά μεγέθη αρχείων και διαμορφώσεις συστήματος.

ΕΙΔΗ PADDING

Ο αλγόριθμος AES όπως προαναφέρθηκε στις προηγούμενες ενότητες λειτουργεί σε μπλοκ δεδομένων μεγέθους 16 bytes (128 bits). Αυτό σημαίνει ότι η πληροφορία που πρόκειται να κρυπτογραφηθεί χωρίζεται σε τμήματα των 16 bytes. Ωστόσο, σε επίπεδο εφαρμογής, το μήνυμα δεν έχει πάντοτε μήκος που να είναι ακριβές πολλαπλάσιο του 16. Για τον λόγο αυτό απαιτείται η προσθήκη επιπλέον δεδομένων στο τέλος του μηνύματος, ώστε το συνολικό μήκος να είναι πράγματι πολλαπλάσιο του μεγέθους του μπλοκ. Η διαδικασία αυτή ονομάζεται padding και καθορίζει τον τρόπο με τον οποίο γεμίζουν τα κενά bytes [17]. Σημαντικό είναι να επισημανθεί ότι ακόμα και αν το μήνυμα είναι πολλαπλάσιο του 16, προστίθεται ένα ολόκληρο block padding ώστε να γίνεται ξεκάθαρο κατά το βήμα της αποκρυπτογράφησης που τελειώνει η πληροφορία. Υπάρχουν διάφορες μέθοδοι padding, όπως το **Zero Padding,** το **ISO/IEC 7816-4**, το **PKCS#7** και το **ANSI X.923**.

Κάθε μέθοδος padding βασίζεται σε διαφορετική λογική και χρησιμοποιείται ανάλογα με τις απαιτήσεις και τις προδιαγραφές που θέτουν οι αναλυτές ή τα πρότυπα ασφαλείας. Έτσι, υπάρχουν αρκετές εναλλακτικές προσεγγίσεις για τη διαχείριση των κενών bytes, καθεμία με τα δικά της πλεονεκτήματα και μειονεκτήματα.

ZERO PADDING

Η μέθοδος **Zero Padding** θεωρείται μία από τις πιο απλές τεχνικές συμπλήρωσης δεδομένων. Όπως υποδηλώνει και το όνομά της, η λογική της βασίζεται στην προσθήκη μηδενικών (bytes με τιμή 0x00) ώστε να καλυφθούν τα κενά στοιχεία στο τέλος του μηνύματος. Η πληροφορία χωρίζεται σε μπλοκ δεδομένων των 16 bytes. Στην περίπτωση που το τελευταίο μπλοκ δεν έχει μήκος πολλαπλάσιο του 16, συμπληρώνεται με μηδενικά μέχρι να φτάσει το απαιτούμενο μέγεθος. Έτσι εξασφαλίζεται ότι κάθε μπλοκ έχει το ίδιο μέγεθος και μπορεί να υποβληθεί σωστά σε επεξεργασία από τον αλγόριθμο AES [18]. Δεν θεωρείται αξιόπιστος τρόπος padding εφόσον στο βήμα της αποκρυπτογράφησης σε περίπτωση που τα τελευταία στοιχεία του μηνύματος αποτελούνται από μηδενικά θα αφαιρεθούν κατά το βήμα απαλοιφής του padding με αποτέλεσμα να αλλοιωθεί το αρχικό μήνυμα.

PKCS#7

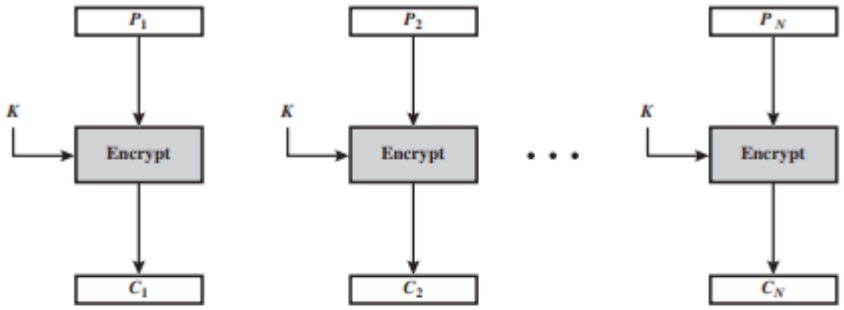
Η μέθοδος **PKCS#7** εφαρμόζεται βάσει του χώρου που απομένει στο τελευταίο block των 16 byte. Αν το μήκος του τελευταίου block δεν είναι ακριβές πολλαπλάσιο του 16, τότε προστίθενται τόσα bytes όσα λείπουν, με την τιμή κάθε byte ίση με τον αριθμό των byte που προστέθηκαν. Με αυτόν τον τρόπο το μήνυμα συμπληρώνεται ώστε να ευθυγραμμίζεται πάντα με το μέγεθος block του AES, ενώ το padding μπορεί να αφαιρεθεί με ασφάλεια κατά την αποκρυπτογράφηση [19].

MODE OPERATION

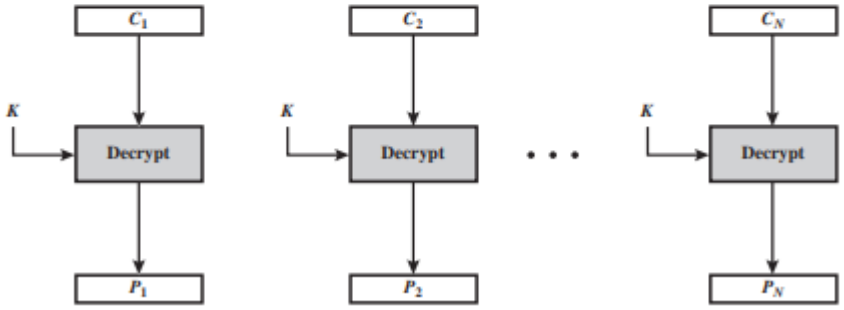
Ως Mode of Operation ονομάζεται ο τρόπος με τον οποίο ένας block cipher, όπως ο AES, εφαρμόζεται σε δεδομένα μεγαλύτερα από ένα block (16 bytes) και ορίζει τον τρόπο αλληλεπίδρασης των blocks μεταξύ τους. Διάφοροι τύποι modes λειτουργίας για τον αλγόριθμο AES έχουν προταθεί στη βιβλιογραφία, όπως τα ECB, CBC, CFB, OFB, CTR, GCM, EAX και άλλα. Στην παρούσα εργασία θα εξεταστούν οι ECB, CBC και GCM, οι οποίοι επιλέχθηκαν ώστε να αναδειχθούν οι διαφορές ανάμεσα σε ένα απλό mode (ECB), καθώς ένα κλασικό και του ευρέως χρησιμοποιούμενο mode με αλυσιδωτή εξάρτηση (CBC) και στην πιο σύνθετη μορφή GCM η οποία περιλαμβάνει ένα counter και ένα tag.

ECB MODE

Το Electronic Codebook (ECB), στο οποίο το plaintext επεξεργάζεται block-block και κάθε block κρυπτογραφείται με το ίδιο κλειδί όπως φαίνεται στο παρακάτω σχήμα:



(a) Encryption

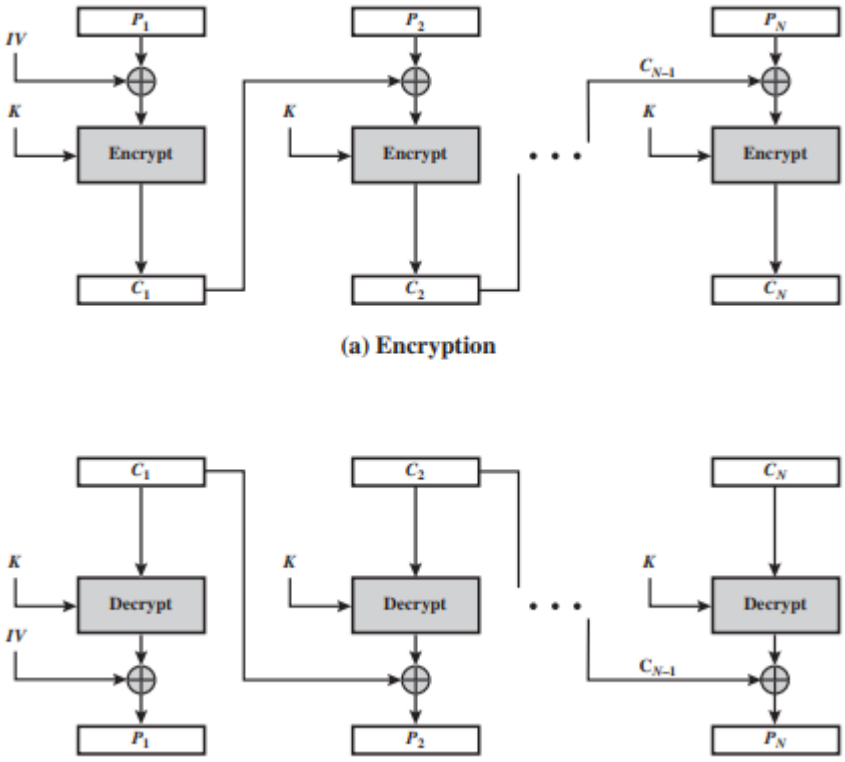


(b) Decryption

[20]

CBC MODE

Το Cipher Block Chaining (CBC) χρησιμοποιεί έναν μη προβλέψιμο IV (τυχαία ακολουθία αριθμών 16 bytes) ως αρχική είσοδο στο πρώτο block plaintext και σε κάθε επόμενο γίνεται XOR με το αποτέλεσμα του προηγούμενου block, όπως παρουσιάζεται στο σχήμα που ακολουθεί [21].



[20]

Η κρυπτογράφηση με CBC γίνεται:

- $C_1 = E_K(P_1 \oplus IV)$
- $C_j = E_K(P_j \oplus C_{j-1}), με\ j = 2, \ldots, n$

Η αποκρυπτογράφηση CBC γίνεται:

- $P_1 = D_K(C_1) \oplus IV$
- $P_j = D_K(C_{j-1}) \oplus C_{j-1}, με\ j = 2, \ldots, n$.

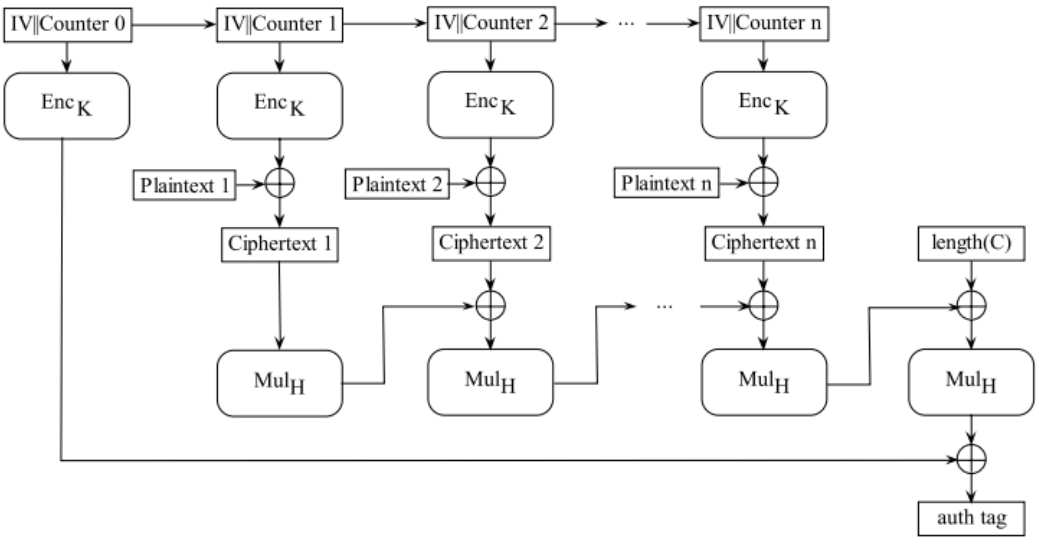
GCM MODE (Galois/Counter Mode)

Ο AES σε λειτουργία **GCM (Galois/Counter Mode)** είναι ένας αλγόριθμος Authenticated Encryption, ο οποίος συνδυάζει κρυπτογράφηση και αυθεντικοποίηση. Μέσω της συνάρτησης GHASH και της χρήσης Associated Data (AD), το GCM επιτυγχάνει έλεγχο ακεραιότητας και αυθεντικότητας τόσο για τα κρυπτογραφημένα δεδομένα όσο και για επιπλέον, μη μυστικά δεδομένα [22].

Στο παρακάτω σχήμα παρουσιάζεται η κρυπτογράφηση σε GCM mode, όπου χρησιμοποιείται ως είσοδο το αρχικό μήνυμα, αφού το έχει χωρίσει σε blocks των 128 bit (

Plaintext 1, Plaintext 2,..., Plaintext n) το IV, ένας μετρητής counter, με $i = 1, \ldots, n$. Σε αντιπαραβολή με τις προηγούμενες μεθόδους το GCM δεν χρειάζεται padding, το τελευταίο ελλειπές Plaintext γίνεται XOR με τα αντίστοιχα bits.

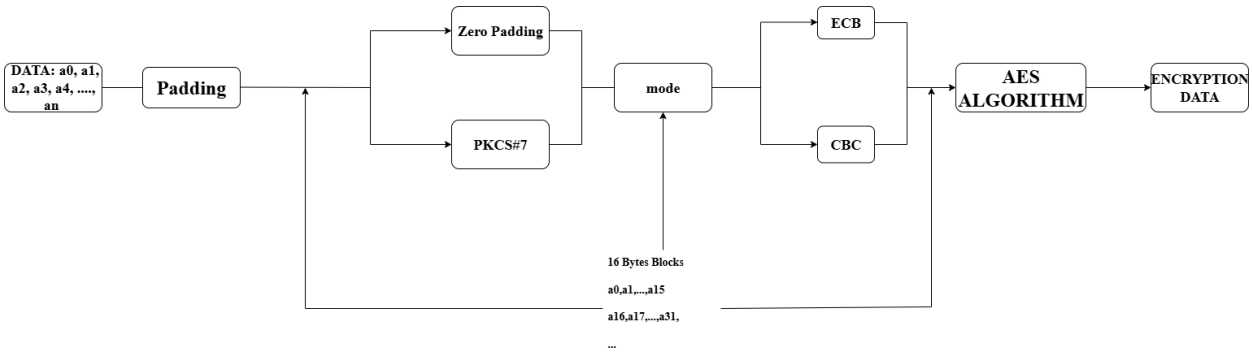
Αρχικά, το IV αποτελεί ένα μοναδικό 12 bytes (96 bits [22]) στοιχείο το οποίο αποτελεί και την αρχή της κρυπτογραφικής διαδικασίας, κάθε επόμενο IV διαφοροποιείται με την χρήση ενός μετρητή, σύμφωνα με τον τύπο. Το pre-counter block ορίζεται ως $J_0 = IV||0^{31}||1$, ενώ κάθε επόμενο counter blocks αυξάνει το J_0, σύμφωνα με τις σχέσεις $CB_1 = inc32(J_0)$ και για κάθε επόμενο $CB_i = inc32(CB_{i-1})$, όπου $int32(x)$ η συνάρτηση η οποία αυξάνει κατά 1 τα 32 τελευταία bits, πιο αναλυτικά αν $x = MSB_{96}(x)||LSB_{32}(x)$ και εφαρμόζεται $int32(x) = MSB_{96}(x)|||[(LSB_{32}(x) + 1) mod 2^{32}]$. Άρα, στα επόμενα blocks η είσοδος είναι το $IV||CB_i$. Τέλος, για κάθε block και σύμφωνα με τον τύπο $C_i = P_i \oplus AES_k(CB_i)$, δημιουργείται το τελικό κρυπτογραφημένο μήνυμα, το τελευταίο ελλειπές block ακολουθεί τη σχέση $C^* = P^* \oplus MSB_{|P^*|}E_K(CB_n))$. Παράλληλα στο τελικό βήμα πραγματοποιείται και η αυθεντικοποίηση. Τα Associated Data (AD) και όλα τα blocks του ciphertext C τροφοδοτούνται στη συνάρτηση GHASH με παράμετρο το H, παράγοντας ένα ενδιάμεσο 128-bit αποτέλεσμα: S = GHASH(H, AD, C)[23]. Το τελικό authentication tag T προκύπτει από τη σχέση $T = MSB_t(S \oplus AES_K(j_0))$, όπου t είναι το μήκος του tag T. Κατά την αποκρυπτογράφηση, το tag επανυπολογίζεται με τον ίδιο τύπο και, αν δεν ταυτίζεται με το λαμβανόμενο T, η διαδικασία θεωρείται αποτυχημένη και το μήνυμα απορρίπτεται. Με αυτό τον τρόπο επιτυγχάνεται η αυθεντικοποίηση του μηνύματος. Στο **ΣΧΗΜΑ 5** αποτυπώνεται η γραφική απεικόνιση του GCM mode.



ΣΧΗΜΑ 5

ΠΛΗΡΕΣ ΣΧΗΜΑ ΚΡΥΠΤΟΓΡΑΦΗΣΗΣ ΜΕ AES

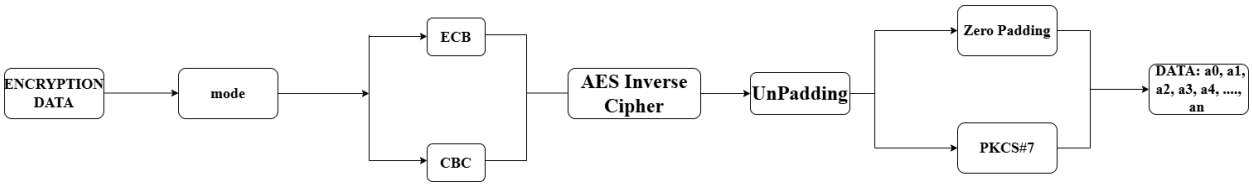
Κατά το βήμα της κρυπτογράφησης αρχικά εφαρμόζεται padding (π.χ.: ISO/IEC 7816-4, PKCS#7), έπειτα χρησιμοποιείται το επιλεγμένο Mode of Operation (π.χ.: ECB, CBC) και τέλος εφαρμόζεται ο αλγόριθμος AES, όπως παρουσιάζεται στο παρακάτω **ΣΧΗΜΑ 6**:



ΣΧΗΜΑ 6

ΠΛΗΡΕΣ ΣΧΗΜΑ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ ΜΕ AES

Κατά το βήμα της αποκρυπτογράφησης, αρχικά εφαρμόζεται το αντίστοιχο Mode of Operation (π.χ. ECB, CBC, GCM) ώστε να ανακτηθούν τα blocks του ciphertext σύμφωνα με τη λειτουργία του εκάστοτε mode. Στη συνέχεια εφαρμόζεται ο αλγόριθμος AES–Inverse Cipher (INVCIPHER) για την αναστροφή των γύρων κρυπτογράφησης και την ανάκτηση του αρχικού state. Τέλος, αφαιρείται το padding (π.χ. ISO/IEC 7816-4, PKCS#7), αποδίδοντας το τελικό plaintext, όπως φαίνεται στο παρακάτω **ΣΧΗΜΑ 7**.



ΣΧΗΜΑ 7

Η ΑΝΤΑΛΛΑΓΗ ΤΩΝ ΚΛΕΙΔΙΩΝ

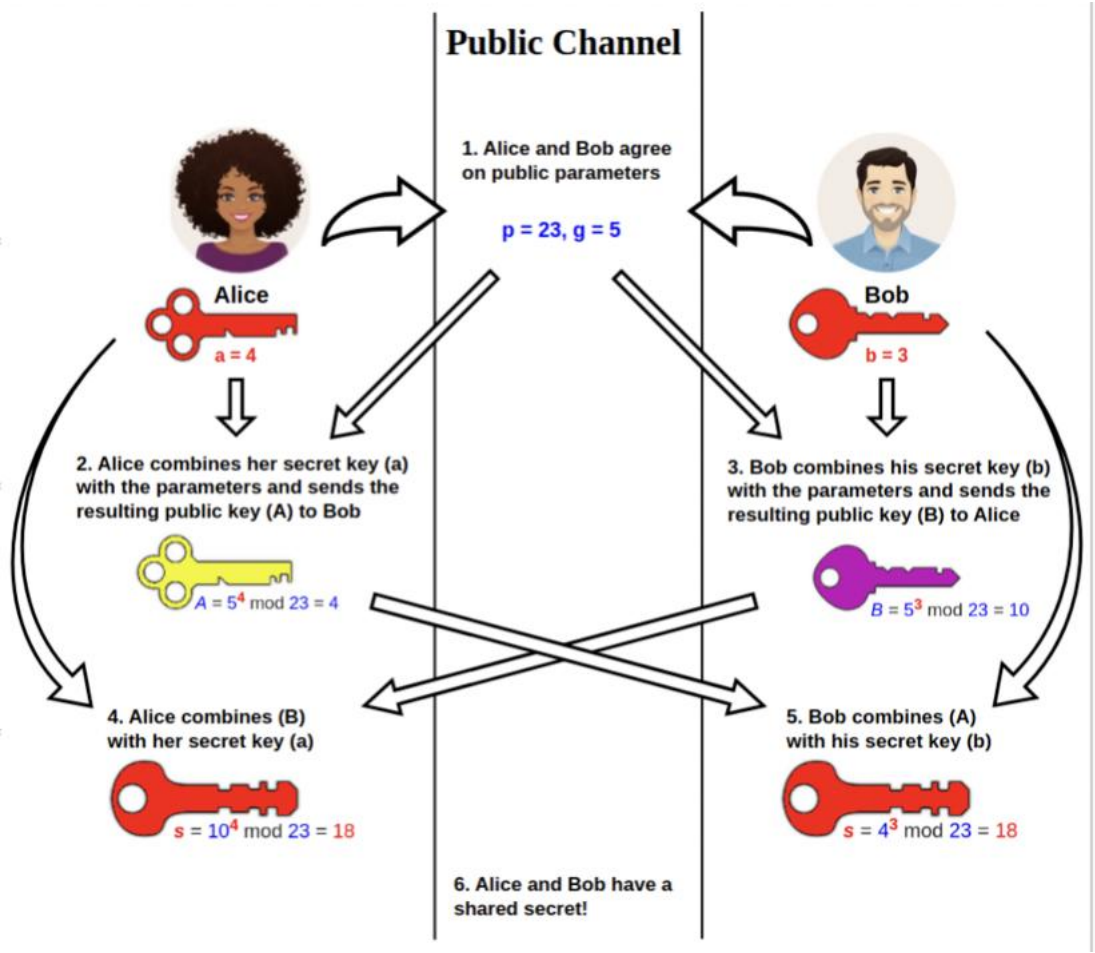
ΤΟ ΠΡΩΤΟΚΟΛΛΟ DIFFIE - HELMANN

Το πρωτόκολλο Diffie-Hellman (DH) αποτελεί έναν θεμελιώδη αλγόριθμο ανταλλαγής κλειδιών στην κρυπτογραφία, επιτρέποντας σε δύο μέρη να δημιουργήσουν ένα κοινό συμμετρικό AES κλειδί μέσω ενός μη ασφαλούς διαύλου επικοινωνίας. Η ασφάλεια του πρωτοκόλλου βασίζεται στη δυσκολία του προβλήματος του διακριτού λογαρίθμου το οποίο έχει μεγάλο υπολογιστικό κόστος, σχεδόν αδύνατο να υπολογιστεί σε λογικά χρονικά διαστήματα.

Σύμφωνα με τους Diffie & Hellman τα βήματα που χρειάζεται να γίνουν έχουν ως εξής:

1. Επιλογή δημοσίων παραμέτρων:
    a. Ένας μεγάλος πρώτος αριθμός p(συνήθως 2048 bytes και πάνω)
    b. Μία γεννήτορας του p, δηλαδή ένας αριθμός g τέτοιος ώστε τα $g^1, g^2, ..., g^{p-1} \bmod p$ να καλύπτουν όλους τους ακεραίους modulo p (Z_p^*).
2. Η Alice:
    a. Επιλέγει έναν μυστικό αριθμό a (ιδιωτικό κλειδί)
    b. Υπολογίζει $A=g^a \bmod p$ (δημόσιο κλειδί)
    c. Στέλνει το A στον χρήστη B
3. Ο Bob:
    a. Επιλέγει έναν μυστικό αριθμό b
    b. Υπολογίζει $B=g^b \bmod p$
    c. Στέλνει το B στον χρήστη A
4. Υπολογισμός του **κοινού μυστικού**:
    a. Ο A υπολογίζει: $s=B^a \bmod p=g^{ba} \bmod p$
    b. Ο B υπολογίζει: $s=A^b \bmod p=g^{ab} \bmod p$

Και οι δύο φτάνουν στο ίδιο κοινό κλειδί s, χωρίς να έχουν ανταλλάξει τους ιδιωτικούς τους αριθμούς.



Βήμα 1: Βρίσκω έναν αριθμό g τέτοιο ώστε τα $g^1, g^2, ..., g^{p-1} \bmod p$ να καλύπτουν όλους τους ακεραίους modulo p (δηλ. ένας γεννήτορας του Z_{17}^*). Στο παράδειγμα για p=17 υπολογίζεται ότι g=3.

Βήμα 2: Επιλογή μυστικών κλειδιών

Ο Χρήστης Α διαλέγει το ιδιωτικό κλειδί a=6. Ο Χρήστης Β διαλέγει το ιδιωτικό κλειδί b=15.

Βήμα 3: Δημιουργία δημόσιων κλειδιών

Χρησιμοποιούμε τον γεννήτορα g=3 και το p=17 και υπολογίζουμε τα Α και Β:

Η Alice υπολογίζει το δημόσιο κλειδί του Α: $A=g^a \bmod p=3^6 \bmod 17=729 \bmod 17=15$ και το στέλνει στον Bob. Ο Bob υπολογίζει το δημόσιο κλειδί του Β: $B=g^b \bmod p=3^{15} \bmod 17=6$ και το στέλνει στην Alice.

Βήμα 4: Κοινό Μυστικό Κλειδί

Η Alice υπολογίζει το μυστικό κλειδί εφόσον τώρα διαθέτει τα B και a: $s=B^a \bmod p=6^6 \bmod 17=46656 \bmod 17=2$

Ο Bob υπολογίζει το μυστικό κλειδί εφόσον τώρα διαθέτει τα A και b: $s=A^b \bmod p=15^{15} \bmod 17=2$

Τελικά και οι δύο έχουν το κοινό μυστικό κλειδί s=2, χωρίς αυτό να εκτεθεί στα δημόσιο μέρος της επικοινωνίας.

Όπως, αναφέρθηκε η ασφάλεια βασίζεται στο πρόβλημα διακριτού λογαρίθμου (Discrete Logarithm Problem). Ο attacker γνωρίζει τα στοιχεία που βρίσκονται στο δημόσιο χώρο, δηλαδή τα p, g, A και B και χρειάζεται να υπολογίσει το $s= B^a \bmod p= A^b \bmod p$. Αρκεί να βρει για ποια $x\epsilon\{1,\dots,p-1\}$ ισχύει ότι $A=g^x \bmod p$ ή $B=g^x \bmod p$. Αν ο πρώτος αριθμός p είναι σχετικά μικρός του υπολογιστικό κόστος είναι ελάχιστο, όταν όμως ο αριθμός p είναι πολύ μεγάλος πρώτος, έχουμε πολύ μεγάλο υπολογιστικό κόστος, αποφεύγοντας έτσι τις επιθέσεις της μορφής brute force attack.

ΤΟ DISCRETE LOGARITHM PROBLEM (DLP)

Ο υπολογισμός του α από τη γνωστή σχέση $A=g^a \bmod p$. Το σύνολο των πιθανών τιμών του a είναι τεράστιο, αν για παράδειγμα το p είναι 2048-bit, τότε $a\in\{1,2,\dots,p-2\}$, δηλαδή της τάξης $2^{2048} \sim 22048$ τιμές, με αποτέλεσμα το υπολογιστικό κόστος να είναι τεράστιο και τελικά να είναι σχεδόν αδύνατος ο υπολογισμός.

ΕΛΛΕΙΠΤΙΚΕΣ ΚΑΜΠΥΛΕΣ ΣΤΗΝ ΚΡΥΠΤΟΓΡΑΦΙΑ

Η ασφάλεια της ανταλλαγής κλειδιών Diffie-Hellman βασίζεται στη δυσκολία επίλυσης του προβλήματος του διακριτού λογαρίθμου. Ωστόσο, η ραγδαία ανάπτυξη των υπολογιστικών δυνατοτήτων έχει καταστήσει το πρωτόκολλο Diffie-Hellman λιγότερο αποτελεσματικό ως προς την παροχή ισχυρής ασφάλειας στα σύγχρονα συστήματα. Για τον λόγο αυτό έχουν αναπτυχθεί νεότερα πρωτόκολλα ανταλλαγής κλειδιών, όπως το Elliptic Curve Diffie-Hellman (ECDH). Το ECDH είναι ένα ανώνυμο πρωτόκολλο συμφωνίας κλειδιού που επιτρέπει σε δύο μέρη, να δημιουργήσουν ένα κοινό μυστικό κλειδί μέσω ενός μη ασφαλούς καναλιού, χρησιμοποιώντας ζεύγη δημόσιου-ιδιωτικού κλειδιού τα οποία βασίζονται σε ελλειπτικές καμπύλες.

Ως ελλειπτική καμπύλη E η οποία συχνά αναφέρεται ως καμπύλη Weierstrass ορίζεται η εξίσωση $y^2 + a_1 x\, y + a_3 y = x^3 + a_2\ x^2 + a_4 x + a_6$, με $a_1, a_2, a_3, a_4, a_6 \epsilon\, K$, όπου η διακρίνουσα Δ ορίζεται ως εξής:

$$d_2 = a_1{}^2 + 4a_2$$

$$d_4 = 2a_4 + a_1 a_3$$

$$d_6 = 2a_3{}^2 + 4a_6$$

$$d_8 = a_1{}^2\, a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2$$

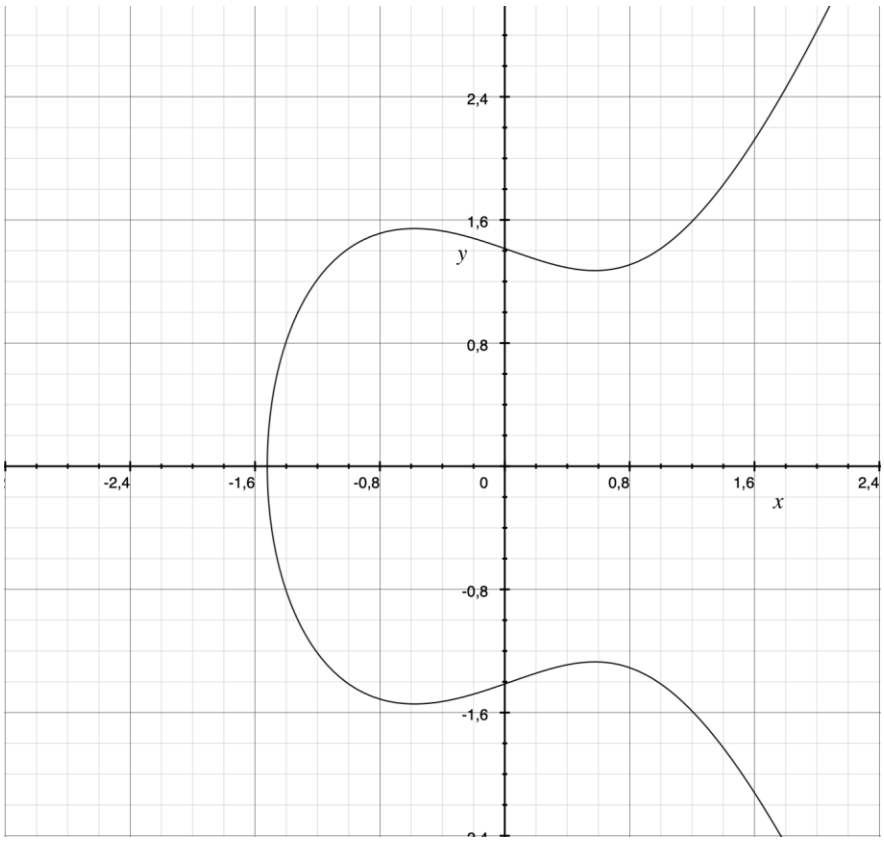
και $\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2 d_4 d_6$.

Αν L είναι οποιαδήποτε επέκταση του σώματος K, τότε το σύνολο των L-ορθολογικών σημείων της ελλειπτικής καμπύλης E είναι της μορφής:

$E(L) = \{(x, y)\ \epsilon\ LxL\colon\ y^2 + a_1 x\ y + a_3 y = x^3 + a_2\ \ x^2 + a_4 x + a_6\} \cup \{\infty\}$, τα ζεύγη (x,y) είναι λύσεις της εξίσωσης με συντελεστές στο L και το ∞ είναι το σημείο στο άπειρο (ειδικό στοιχείο που προστίθεται στο σύνολο).

Λέμε ότι η καμπύλη είναι ορισμένη πάνω στο K,εφόσον ισχύει ότι $a_1, a_2, a_3, a_4, a_6 \epsilon\ K$ και γράφεται $E|K$, δηλαδή η ελλειπτική καμπύλη E ορίζεται πάνω στο K.

Για παράδειγμα η ελλειπτική καμπύλη $E_1\colon y^2 = x^3 - x + 2$, πάνω στο ℜ απεικονίζεται κάπως έτσι:



Ως ελλειπτική καμπύλη E ορίζεται η εξίσωση $y^2 = x^3 + a\ \ x + b$, με $4a^3 + 27b^2 \neq 0$ $a, b \in Q$. Έπειτα, γίνεται επιλογή ενός πρώτου αριθμού p, και η εξίσωση της ελλειπτικής καμπύλης διαμορφώνεται $y^2 = x^3 + a\ \ x + b\ mod\ p$, με $a, b\ \epsilon\ F_p$, με την προϋπόθεση $4a^3 + 27b^2 \neq 0(modp)$. Το πλήθος των σημείων της ελλειπτικής καμπύλης E πάνω από το πεδίο F_p ορίζεται το $N_p = \#E(F_p)$.

Το χαρακτηριστικό ενός σώματος K, που συμβολίζεται ως char(K), είναι ένας θεμελιώδης αριθμητικός «δείκτης» του σώματος. Ορίζεται ως ο μικρότερος θετικός ακέραιος n τέτοιος ώστε $n \cdot 1 = 1 + 1 + \ldots + 1 = 0, στο\ K$. Σε περίπτωση που δεν υπάρχει τέτοιος ακέραιος, τότε λέμε ότι

το χαρακτηριστικό είναι 0. Για παράδειγμα στα σώματα των ρητών Q ή των πραγματικών $\Re$ ισχύει ότι $char(Q) = char(\Re) = 0$. Αντίστοιχα, στα σώματα $F_p = Z/pZ$ ισχύει ότι $char(F_p) = p$. Συνοπτικά οι κατηγορίες των ελλειπτικών καμπυλών ανάλογα με την τιμή του char μπορούν να κατηγοριοποιηθούν ως εξής:

Χαρακτηριστικό του K	Επιτρεπτή μορφή εξίσωσης	Τύπος καμπύλης	Διακρίνουσα
char(K) ≠ 2,3	$y^2 = x^3 + ax + b$	-	$\Delta = -16(4a^3 + 27b^2)$
char(K) = 2, a1 ≠ 0	$y^2 + xy = x^3 + ax^2 + b$	no-supersingular	$\Delta = b$
char(K) = 2, a1 = 0	$y^2 + cy = x^3 + ax + b$	ordinary	$\Delta = c^4$
char(K) = 3, $a1^2 \neq -a2$	$y^2 = x^3 + ax^2 + b$	no-supersingular	$\Delta = -a^3b$
char(K) = 3, $a1^2 = -a2$	$y^2 = x^3 + ax + b$	ordinary	$\Delta = -a^3$

Σύμφωνα με τον NIST [24] οι ελλειπτικές καμπύλες που χρησιμοποιούνται στην κρυπτογραφία χρειάζεται να πληρούν τις εξής προϋποθέσεις:

1. Υποκείμενο πεπερασμένο σώμα: Το σώμα GF(q) πρέπει να είναι είτε πρωτογενές (GF(p)) είτε δυαδικό $GF(2^m)$, όπου το m είναι πρώτος αριθμός.
2. Τάξη της καμπύλης: Κάθε καμπύλη E πάνω από $GF(q)$πρέπει να έχει τάξη |E| = h·n, όπου n είναι μεγάλος πρώτος αριθμός, h είναι μικρός ακέραιος (ο συντελεστής cofactor), συνεπώς $gcd(h, n) = 1$, και πρέπει να ισχύει $h \leq 2^{10}$.
3. Σημείο βάσης: Κάθε καμπύλη E πρέπει να έχει σταθερό σημείο βάσης G με πρώτη τάξη n.
4. Αποφυγή επιθέσεων anomalous curve: Η τάξη της καμπύλης δεν πρέπει να είναι ίση με q, ώστε να αποφεύγονται ειδικές επιθέσεις.
5. Μεγάλος embedding degree: Το πρόβλημα διακριτού λογαρίθμου στην καμπύλη E μεταφέρεται σε $GF(q^k)$, όπου το k είναι το embedding degree (ελάχιστος θετικός ακέραιος με $q^k \equiv 1 \bmod n$). Πρέπει να ισχύει $k \geq 2^{10}$. Οι καμπύλες της NIST έχουν embedding degree πολύ μεγαλύτερο, κοντά στην τάξη n.
6. Πεδίο ενδομορφισμών: Για καμπύλη E με ίχνος t, ο αριθμός $Disc = t^2 - 4q$ σχετίζεται με τη διακρίνουσα του πεδίου ενδομορφισμών. Δεν επιβάλλεται περιορισμός στο square-free μέρος του |Disc| (εκτός από τις καμπύλες που χρησιμοποιούνται σε pairing-based κρυπτογραφία). Το t λέγεται ίχνος της καμπύλης (trace of Frobenius) και υπολογίζεται από τη σχέση $|E(GF(q))| = q + 1 - t$, όπου $|E(GF(q))|$ το πλήθος των σημείων της καμπύλης πάνω από $GF(q)$. Επίσης, ισχύει $|t| \leq 2\sqrt{q}$, εξασφαλίζοντας ότι το t είναι αρκετά μικρό.

Όλες οι παραπάνω προϋποθέσεις που έχουν τεθεί από το πρότυπο NIST SP 800-186 οδηγούν στο συμπέρασμα ότι για λόγους ασφάλειας χρησιμοποιούνται αποκλειστικά κανονικές (ordinary) ελλειπτικές καμπύλες. Οι supersingular καμπύλες αποκλείονται εφόσον έχουν μικρό embedding degree (άρα επιτρέπουν pairing-based επιθέσεις) και δεν διαθέτουν μεγάλο πρώτο παράγοντα στην τάξη τους. Συμπερασματικά, για αλγορίθμους ανταλλαγής κλειδιού (π.χ. ECDH), αλλά και για υπογραφή (ECDSA), η NIST συστήνει μόνο κανονικές ελλειπτικές καμπύλες ορισμένες πάνω από (GF(p)) είτε δυαδικό $GF(2^m)$.

Ο NIST συνιστά συγκεκριμένα δεκαπέντε ελλειπτικές καμπύλες οι οποίες παρουσιάζονται στον παρακάτω πίνακα, μαζί με το Security Strength σε bits του κάθε αλγορίθμου:

Security Strength (bits)	**Προτεινόμενες καμπύλες**
112	P-224, K-233, B-233
128	P-256, W-25519, Curve25519, Edwards25519, K-283, B-283
192	P-384, K-409, B-409
224	W-448, Curve448, Edwards448, E448
256	P-521, K-571, B-571

Η P-224

Σε πρώτο πλάνο θα γίνει μία αναφορά στην P-224, καθώς και τα χαρακτηριστικά αυτής. Η P-224 αποτελεί μία pseudorandom Weierstrass curve πάνω από prime field και η μορφή της είναι $E: y^2 = x^3 + ax + b(modp)$, με Domain Parameters $D = (p, h, n, Type, a, b, G\{seed, c\})$, όπου:

- p : prime modulus, ο πρώτος αριθμός που ορίζει το μέγεθος του πεδίου GF(p). Η γενική μορφή του p είναι : $p = \sum_i \quad c_i \cdot 2^{e_i}$, με $c_i \epsilon \{-2, -1, 0, 1, 2\}$.

- h : cofactor, Ορίζεται από τη σχέση |E|=h· n, είναι μικρός ακέραιος (π.χ. 1, 2 ή 4) και για τις pseudorandom prime curves, ορίζεται πάντα h=1.
- n : order of base point, ο αριθμός των σημείων στο κυκλικό υποσύνολο που παράγεται από το σημείο βάσης G, στις NIST curves, το n είναι μεγάλος πρώτος αριθμός (π.χ. 224 bit για την P-224).
- $Type$: το είδος της καμπύλης (π.χ.: pseudorandom Weierstrass curve)

- a : ο συντελεστής του x, οι NIST prime curves ορίζονται με a=−3(modp).
- b : παράγεται ψευδοτυχαία από το Seed μέσω SHA-1.
- $G\{seed, c\}$:
    - seed → Ένα 160-bit string που δημοσιεύεται από τη NIST και αποτελεί την είσοδο στον SHA-1, για να παραχθεί ο συντελεστής b.
    - c → Το αποτέλεσμα του SHA-1 hashing του Seed και χρησιμοποιείται για να ελεγχθεί ότι η κατασκευή του b έγινε σωστά.

Σύμφωνα με τον NIST [24] η P-224 έχει ως παραμέτρους τις παρακάτω:
p = 26959946667150639794667015087019630673557916260026308143510066298881 ή$p = 2^{224} - 2^{96} + 1$ (=0xffffffff ffffffff ffffffff ffffffff 00000000 00000000 00000001)
n = 26959946667150639794667015087019625940457807714424391721682722368061 (=(p+1) − h· n = 0xe95c 1f470fc1 ec22d6ba a3a3d5c5)
SEED = bd713447 99d5c7fc dc45b59f a3b9ab8f 6a948bc5
c = 5b056c7e 11dd68f4 0469ee7f 3c7a7d74 f7d12111 6506d031 218291fb (=0x5b056c7e 11dd68f4 0469ee7f 3c7a7d74 f7d12111 6506d031 218291fb)
b = b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943 2355ffb4 (=0xb4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943 2355ffb4)
Gx = b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6 115c1d21 (=0xb70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6 115c1d21)
Gy = bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199 85007e34 (=0xbd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199 85007e34)

ΠΡΑΞΕΙΣ ΣΤΙΣ ΕΛΛΕΙΠΤΙΚΕΣ ΚΑΜΠΥΛΕΣ SHORT WEIERSTRASS

Η μορφή της ελλειπτικής καμπύλης Short Weierstrass, όπως προαναφέρθηκε ορίζεται ως εξής: $E: y^2 = x^3 + ax + b(modp), με\ p > 3$. Κρίνεται απαραίτητο να αναφερθούν οι ιδιότητες των πράξεων:

ΠΡΟΣΘΕΣΗ : $P + Q$, έστω $P = (x_1, y_1)$ και $Q = (x_2, y_2)$,

- Αν $P \neq Q$, τότε ορίζεται $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$. Ως $P + Q$ ορίζεται το ζεύγος (x_3, y_3), με $x_3 = \lambda^2 - x_1 - x_2(modp)$ και $y_3 = \lambda \cdot (x_1 - x_3) - y_1(modp)$.

- Αν $P = Q$, τότε ορίζεται $\lambda = \frac{3 \cdot x_1^2 + a}{2 \cdot y_1}$. Ως $P + Q$ ορίζεται το ζεύγος (x_3, y_3), με $x_3 = \lambda^2 - 2x_1 (mod p)$ και $y_3 = \lambda \cdot (x_1 - x_3) - y_1 (mod p)$.

ΠΟΛΛΑΠΛΑΣΙΑΣΜΟΣ : $k \cdot Q = Q + Q + \ldots + Q$, ο πολλαπλασιασμός ορίζεται ως διαδοχική πρόσθεση, ενώ για λόγους υπολογιστικής [24] ευκολίας χρησιμοποιούνται κατά τη διάρκεια του υπολογισμού η μέθοδος διπλασιασμού και προσθήκης (double-and-add). Αν για παράδειγμα χρειάζεται να υπολογιστεί το τετραπλάσιο του $P = (x_1, y_1)$, δηλαδή $4 \cdot P$ τότε αρχικά υπολογίζεται το $2 \cdot P = P + P$, $3 \cdot P = 2 \cdot P + P$, $4 \cdot P = 2(2 \cdot P)$ κλπ.

ΟΥΔΕΤΕΡΟ ΣΤΟΙΧΕΙΟ : ως ουδέτερο στοιχείο εδω ορίζεται το ∞ και ισχύει ότι $P + \infty \, (mod p) = \infty + P \, (mod p) = P \, (mod p)$, για κάθε $P \epsilon E(K)$.
ΑΝΤΙΘΕΤΟ : ορίζεται το $-P$ για το οποίο ισχύει $P + (-P)(mod p) = \infty$.

ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ ΔΙΑΚΡΙΤΟΥ ΛΟΓΑΡΙΘΜΟΥ (DLP) ΣΤΙΣ ΕΛΛΕΙΠΤΙΚΕΣ ΚΑΜΠΥΛΕΣ

Έστω E/F_p μια ελλειπτική καμπύλη ορισμένη E πάνω από πεπερασμένο σώμα F_p, και $S, T \epsilon \, E(F_p)$, όπου S, T δύο σημεία της καμπύλης. Το **Πρόβλημα Διακριτού Λογαρίθμου σε Ελλειπτικές Καμπύλες (ECDLP)** ζητά να βρεθεί ένας ακέραιος m τέτοιος ώστε $S \equiv m \cdot T$. Στην κρυπτογραφία, η δυσκολία επίλυσής του αποτελεί τη θεμελιώδη βάση πάνω στην οποία οικοδομούνται πολλά πρωτόκολλα ασφαλείας που βασίζονται στις ελλειπτικές καμπύλες [25]. Για να γίνει πιο κατανοητό από τον αναγνώστη παρακάτω παρουσιάζεται ένα παράδειγμα ECDLP σε μία καμπύλη με λίγα στοιχεία:

Έστω το πεπερασμένο σύνολο: $F_5 = \{0,1,2,3,4,5\}$ και η ελλειπτική καμπύλη $E : y^2 \equiv x^3 + x (mod 5)$ η οποία ορίζεται πάνω σε αυτό. Τα σημεία που ικανοποιούν την εξίσωση είναι: $E(F_5) = \{\infty, (0,0), (2,1), (2,4), (3,1), (3,4)\}$. Έστω ότι διαλέγουμε τα σημεία T=(2,1) και S=(0,0). Το πρόβλημα του διακριτού λογαρίθμου αναζητά τα σημεία που ικανοποιούν τη σχέση $S \equiv m \cdot T$. Η λύση έχει ως εξής: Αρχικά, υπολογίζονται τα διαδοχικά πολλαπλάσια του T έως ότου προκύψει το S. Δηλαδή : 1T=(2,1), 2T=(3,1), 3T=(0,0). Το οποίο καταλήγει στο S. Άρα, εδώ υπολογίζεται εύκολα στο οτι m=3.

Φυσικά, για καμπύλες με σχετικά μικρές τιμές του p, ο υπολογισμός σημείων και η επίλυση του προβλήματος διακριτού λογαρίθμου μπορούν να πραγματοποιηθούν σχετικά εύκολα. Αντίθετα, σε ελλειπτικές καμπύλες όπως η P-224, το πρόβλημα αυτό καθίσταται υπολογιστικά αδύνατο, γεγονός που το καθιστά ιδιαίτερα κατάλληλο ως θεμέλιο για την κατασκευή κρυπτογραφικών πρωτοκόλλων.

ΤΟ ΠΡΩΤΟΚΟΛΛΟ DIFFIE-HELLMAN ΣΕ ΕΛΛΕΙΠΤΙΚΕΣ ΕΛΛΕΙΠΤΙΚΕΣ ΚΑΜΠΥΛΕΣ (ECDH)

Η δομή του πρωτοκόλλου Diffie-Hellman σε ελλειπτικές καμπύλες βασίζεται στην ίδια ακριβώς λογική με αυτή του Diffie-Hellman. Ορίζονται δημόσιοι παράμετροι μέσο ενός μη ασφαλούς διαύλου επικοινωνίας, οι επιμέρους πλευρές επιλέγουν από ένα μυστικό ακέραιο.

Πρώτο βήμα Ορισμός Δημόσιων Παραμέτρων:

- Όλοι συμφωνούν σε μία **ελλειπτική καμπύλη** Ε πάνω από F_p .
- Επιλέγεται ένα σημείο **G** της καμπύλης (γεννήτορας), γνωστό σε όλους

Δεύτερο βήμα Ορισμός Ιδιωτικών Κλειδιών:

- Η **Alice** διαλέγει έναν μυστικό ακέραιο $a\epsilon[1, n-1]$.
- Ο **Bob** διαλέγει έναν μυστικό ακέραιο $b\epsilon[1, n-1]$.

Τρίτο βήμα Υπολογισμός Τιμών από μέρους των δύο πλευρών:

- Η **Alice** υπολογίζει το $A = aG$ και το στέλνει στον Bob.
- Ο **Bob** υπολογίζει το $B = bG$ και το στέλνει στην Alice.

Τέταρτο βήμα Υπολογισμός του κοινού κλειδιού:

- Η **Alice** υπολογίζει $k = aB = a(bG) = (ab)G$.
- Ο **Bob** υπολογίζει $k = bA = b(aG) = (ba)G = (ab)G$.

Και οι δύο πλευρές υπολογίζουν την τιμή k, χωρίς τελικά να γίνεται γνωστό η αριθμητική τιμή του κοινού κλειδιού.

Θα παρουσιαστεί παράλληλα ένα παράδειγμα για να γίνει κατανοητό πως λειτουργεί το πρωτόκολλο Diffie-Hellman πάνω σε ελλειπτικές καμπύλες.

ΠΑΡΑΔΕΙΓΜΑ

Έστω ότι θα εργαστούμε στην $GF(17)$ με την καμπύλη $E: y^2 = x^3 + 2x + 2(mod17)$ και το σημείο $G = (5,1)$, γεννήτορας της ομάδας πάνω στην καμπύλη Ε.

ΒΗΜΑ 1 (ΙΔΙΩΤΙΚΑ ΚΛΕΙΔΙΑ): Έστω ότι η Alice επιλέγει $a = 5$ και ο Bob επιλέγει $b = 7$.

ΒΗΜΑ 2 (ΔΗΜΟΣΙΑ ΚΛΕΙΔΙΑ): Η κάθε πλευρά υπολογίζει τα δημόσια κλειδιά σύμφωνα με τα ιδιωτικά τους ως εξής: $A = a \cdot G = 5 \cdot G = (9,16)$, $B = b \cdot G = 7 \cdot G = (0,6)$ και τα Α και Β στέλνονται μέσω του μη ασφαλούς διαύλου στην άλλη πλευρά.

ΒΗΜΑ 3 (ΥΠΟΛΟΓΙΣΜΟΣ ΚΟΙΝΟΥ ΜΥΣΤΙΚΟΥ ΚΛΕΙΔΙΟΥ):

Η Alice υπολογίζει το $K = a \cdot B = 5 \cdot (0,6) = (10,11)$.

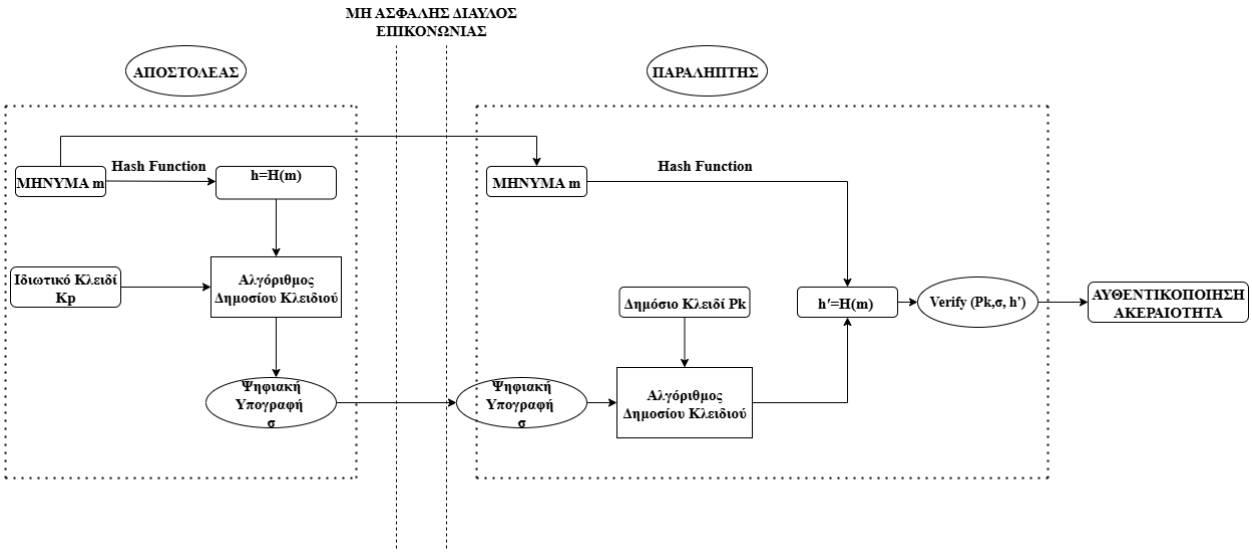
Για τον υπολογισμό του $5 \cdot G$ εργαζόμαστε ως εξής: $5 \cdot G = 4 \cdot G + G$, ενώ $4 \cdot G = 2(2 \cdot G)$ και $2 \cdot G = G + G$, όπως αυτά παρουσιάστηκαν στις πράξεις πάνω στις ελλειπτικές καμπύλες.

Ο Bob υπολογίζει το $K = b \cdot A = 7 \cdot (9,16) = (10,11)$, το γινόμενο $7 \cdot G$ υπολογίζεται όπως και πριν με τη χρήση της μεθόδου διπλασιασμού και προσθήκης.

ΨΗΦΙΑΚΗ ΥΠΟΓΡΑΦΗ ECDSA

Στα σύγχρονα κρυπτογραφικά πρωτόκολλα απαραίτητο κρίνεται το στάδιο της αυθεντικοποίησης, το οποίο πραγματοποιείται με τη μέθοδο της ψηφιακής υπογραφής [26]. Η ψηφιακή υπογραφή αποτελεί το μέσο με το οποίο αποδεικνύεται η ταυτότητα του αποστολέα του μηνύματος. Η γενική λογική της ψηφιακής υπογραφής έχει ως εξής:

Ο αποστολέας θέλει να στείλει ένα μήνυμα m. Αρχικά υπολογίζει τη σύνοψη (hash) h=H(m) με μια κρυπτογραφική συνάρτηση κατακερματισμού (π.χ. SHA-256). Στη συνέχεια, με το ιδιωτικό του κλειδί και τον αλγόριθμο υπογραφής (π.χ. ECDSA), δημιουργεί την ψηφιακή υπογραφή σ=Sign(sk,h). Μέσω του μη ασφαλούς διαύλου επικοινωνίας αποστέλλει το ζεύγος (m,σ). Ο παραλήπτης λαμβάνει (m,σ), υπολογίζει ξανά το h′=H(m) με την ίδια hash όπως αυτή έχει οριστεί από το πρωτόκολλο, και έπειτα εκτελεί τον αλγόριθμο επαλήθευσης με το δημόσιο κλειδί του αποστολέα: Verify(pk,h′,σ). Αν ο έλεγχος επαληθευτεί, τότε το μήνυμα θεωρείται αυθεντικό και ακέραιο ενώ ο αποστολέας αυθεντικοποιείται. Η διαδικασία της ψηφιακής υπογραφής φαίνεται στο παρακάτω σχήμα:



Στην ψηφιακή υπογραφή η οποία βασίζεται στις ελλειπτικές καμπύλες χρησιμοποιεί την θεωρία των ελλειπτικών καμπυλών.

ΠΡΩΤΟ ΒΗΜΑ : ΟΡΙΣΜΟΣ ΔΗΜΟΣΙΟΥ ΚΑΙ ΙΔΙΩΤΙΚΟΥ ΚΛΕΙΔΙΟΥ

Έστω G ένας γεννήτορας τάξης n και ένα ιδιωτικό κλειδί $d \epsilon [1, n-1]$, τέτοιος ώστε $Q = d \cdot G \epsilon E(F_p)$, με Q να αποτελεί και το δημόσιο κλειδί και βρίσκεται πάνω στην ελλειπτική καμπύλη $E: y^2 = x^3 + ax + b(modp)$, με $Q \neq \infty$

ΔΕΥΤΕΡΟ ΒΗΜΑ : ΔΗΜΙΟΥΡΓΙΑ ΤΗΣ ΨΗΦΙΑΚΗΣ ΥΠΟΓΡΑΦΗΣ

Σε συνδυασμό με το μήνυμα (m) που θέλει να στείλει ο αποστολέας και του ιδιωτικού κλειδιού δημιουργείται η ψηφιακή υπογραφή (r, s).

Αρχικά, ορίζεται ένας τυχαίος αριθμός $k \epsilon [1, n-1]$ και ορίζεται το σημείο $P = k \cdot G = (P_x, P_y)$. Αν $P = \infty$ τότε επιλέγεται εκ νέου k. Έτσι, υπολογίζονται οι τιμές της ψηφιακής υπογραφής:

$r = P_x(modn)$, αν $r = 0$ επιλέγεται νέο k.

$e = H(m)$, η hash τιμή του μηνύματος (π.χ. SHA-256)

$s = k^{-1}(e + d \cdot r)(modn)$.

ΤΡΙΤΟ ΒΗΜΑ : ΕΠΑΛΗΘΕΥΣΗ ΤΗΣ ΨΗΦΙΑΚΗΣ ΥΠΟΓΡΑΦΗΣ ΚΑΙ ΤΗΣ ΟΝΤΟΤΗΤΑΣ ΤΟΥ ΑΠΟΣΤΟΛΕΑ

Ο αποστολέας στέλνει στον παραλήπτη το μήνυμα m μαζί με την ψηφιακή υπογραφή (r, s) και το δημόσιο κλειδί Q. Αρχικά, ο παραλήπτης υπολογίζει τη hash τιμή του μηνύματος $e = H(m)$, η συνάρτηση hash (π.χ. SHA-256) έχει οριστεί από τα δύο μέλη σύμφωνα με το πρωτόκολλο που έχουν ορίσει, άρα η τιμή αυτή δίνει το ίδιο αποτέλεσμα με την τιμή που έχει υπολογίσει και ο αποστολέας. Έπειτα, υπολογίζει το $w = s^{-1}(modn)$ και το $X = e \cdot w \cdot G + r \cdot w \cdot Q$. Ορίζεται $x = X_x(modn)$. Αν, $x = r$ τότε έχουμε επαλήθευση της υπογραφής και της οντότητας του αποστολέα.

ΑΡΙΘΜΗΤΙΚΟ ΠΑΡΑΔΕΙΓΜΑ ΨΗΦΙΑΚΗΣ ΥΠΟΓΡΑΦΗΣ

Έστω η ελλειπτική καμπύλη $E: y^2 = x^3 + 2x + 2$ στο F_{17}, γεννήτορας $G = (5,1)$ και $n = 19$.

ΠΡΩΤΟ ΒΗΜΑ : ΟΡΙΣΜΟΣ ΔΗΜΟΣΙΟΥ ΚΑΙ ΙΔΙΩΤΙΚΟΥ ΚΛΕΙΔΙΟΥ

$d = 5$ **και** $Q = d \cdot G = (9, 16)$

ΔΕΥΤΕΡΟ ΒΗΜΑ : ΔΗΜΙΟΥΡΓΙΑ ΤΗΣ ΨΗΦΙΑΚΗΣ ΥΠΟΓΡΑΦΗΣ

Έστω το μήνυμα m με τιμή hash $e = H(m) = 1$ και τυχαίος αριθμός $k = 8$.

$P = k \cdot G = 8 \cdot G = (13,7)$, με $r = P_x(modn) = 13$.

$s = k^{-1}(e + d \cdot r)(modn) = 8^{-1}(1 + 5 \cdot 13)(mod17) = 13$.

Άρα, η ψηφιακή υπογραφή έχει ως εξής : $(r, s) = (13, 13)$

ΤΡΙΤΟ ΒΗΜΑ : ΕΠΑΛΗΘΕΥΣΗ ΤΗΣ ΨΗΦΙΑΚΗΣ ΥΠΟΓΡΑΦΗΣ ΚΑΙ ΤΗΣ ΟΝΤΟΤΗΤΑΣ ΤΟΥ ΑΠΟΣΤΟΛΕΑ

Ο αποστολέας λαμβάνει το μήνυμα m, το δημόσιο κλειδί Q και την ψηφιακή υπογραφή.

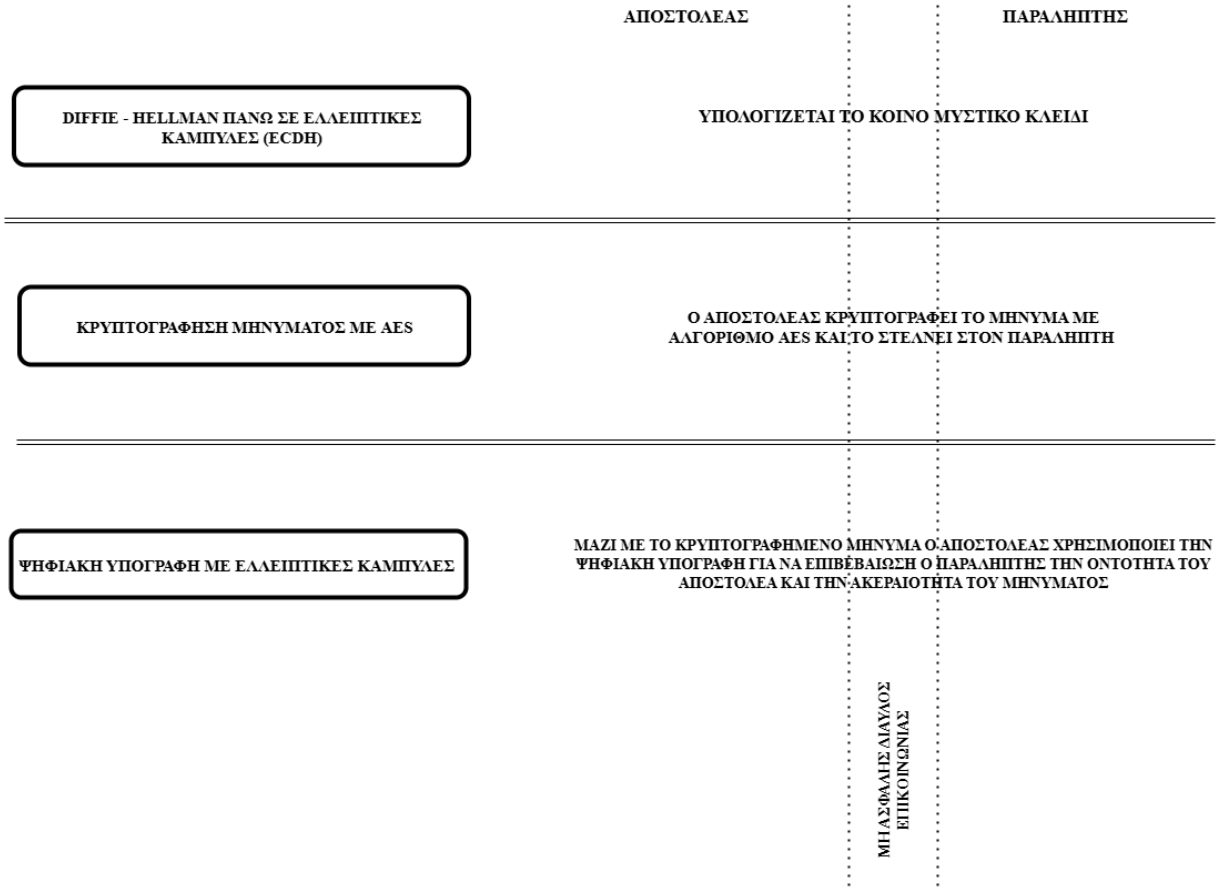
Υπολογίζει την τιμή hash του μηνύματος $e = H(m) = 1$.

Έπειτα με την σχέση $w = s^{-1}(modn) = 13^{-1} mod 19 = 3$ και $X = e \cdot w \cdot G + r \cdot w \cdot Q = (13,7)$.

Ισχύει ότι $X_x = r = 13$, άρα έχουμε επαλήθευση της ταυτότητας του αποστολέα και ακεραιότητα του μηνύματος.

ΜΟΝΤΕΛΟ ΕΠΙΚΟΙΝΩΝΙΑΣ (ECDH–AES–ECDSA)

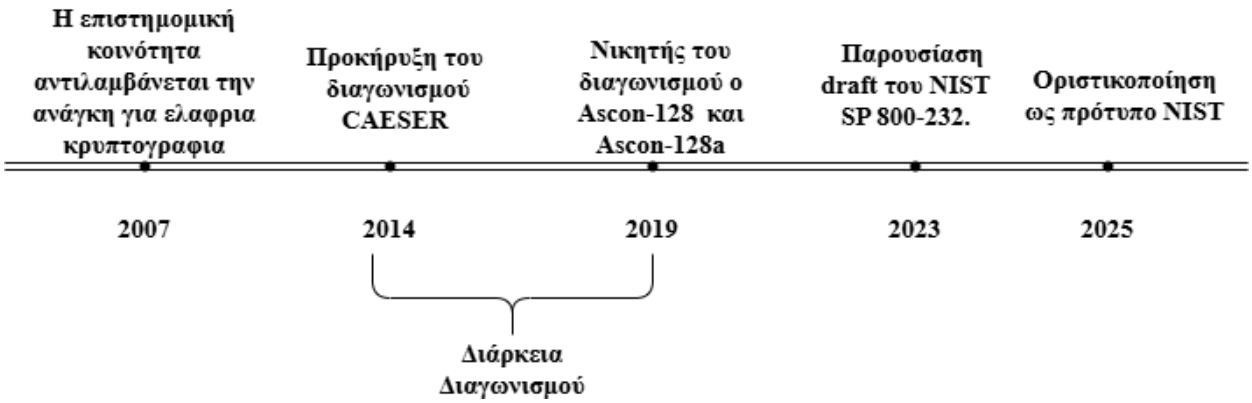
Στην παρούσα εργασία, αρχικά χρησιμοποιείται το πρωτόκολλο Diffie–Hellman πάνω σε ελλειπτικές καμπύλες (ECDH) για την παραγωγή κοινού μυστικού, από το οποίο, μέσω HKDF-SHA256, προκύπτει το συμμετρικό κλειδί AES. Στη συνέχεια ο αποστολέας κρυπτογραφεί το μήνυμα με AES σε ένα από τα modes ECB ή CBC (με PKCS#7 ή Zero padding. Έπειτα δημιουργεί ψηφιακή υπογραφή ECDSA πάνω στα ακριβή πεδία που μεταδίδονται, ώστε να διασφαλίζονται αυθεντικότητα και ακεραιότητα. Προς τον παραλήπτη αποστέλλονται το Header, το κρυπτοκείμενο και η υπογραφή. Ο παραλήπτης επικυρώνει πρώτα την υπογραφή με το δημόσιο κλειδί του αποστολέα και, εφόσον είναι έγκυρη, παράγει το ίδιο συμμετρικό κλειδί μέσω ECDH/HKDF και αποκρυπτογραφεί το μήνυμα. Με αυτόν τον τρόπο επιτυγχάνεται εμπιστευτικότητα (AES) και αυθεντικότητα/ακεραιότητα (ECDSA).



Ο ΑΛΓΟΡΙΘΜΟΣ ASCON

Η ραγδαία εξάπλωση των ευφυών και διασυνδεδεμένων συσκευών, όπως τα συστήματα Internet of Things (IoT), οι αισθητήρες, τα φορητά συστήματα και οι ενσωματωμένες εφαρμογές, έχει δημιουργήσει τις κατάλληλες συνθήκες για την ανάγκη εφαρμογής ελαφριάς κρυπτογραφίας σε αυτά τα περιβάλλοντα. Ήδη από το 2007, παρατηρείται η αυξανόμενη ανάγκη για ανάπτυξη "ελαφρών" κρυπτογραφικών αλγορίθμων τύπου block (lightweight block ciphers), σχεδιασμένων ειδικά για ενσωματωμένα και περιορισμένων πόρων συστήματα.

Το 2014 ο NIST στο διαγωνισμό CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) παρουσίασε την οικογένεια αλγορίθμων Ascon, τον αλγόριθμο Ascon-128, με κλειδί 128 bit και τον αλγόριθμο Ascon-96, με κλειδί 96 bit. Οι ενημερωμένες εκδόσεις v1.1 και v1.2 περιλαμβανε κάποιες τροποποιήσεις, όπως η αλλαγή στο πλήθος των γύρων, με την Ascon-128 να τροποποιείται σε Ascon-128a. Ενώ ακολουθεί το 2015 η ανάπτυξη της Διαδικασίας Τυποποίησης Ελαφριάς Κρυπτογραφίας (Lightweight Cryptography Standardization Process) με στόχο την δημιουργία κρυπτογραφικών πρωτοκόλλων κατάλληλα διαμορφωμένων για συσκευές με περιορισμένους πόρους. Έπειτα Το 2019, οι Ascon-128 και Ascon-128a επιλέχθηκαν ως η πρώτη επιλογή για περιπτώσεις ελαφριάς κρυπτογράφησης με έλεγχο αυθεντικότητας στο τελικό χαρτοφυλάκιο του διαγωνισμού CAESAR. Τέλος, το 2023, το Εθνικό Ινστιτούτο Προτύπων και Τεχνολογίας των Ηνωμένων Πολιτειών (NIST) ανακοίνωσε την επιλογή της οικογένειας αλγορίθμων Ascon, η οποία σχεδιάστηκε από τους Dobraunig, Eichlseder, Mendel και Schläffer, με σκοπό να προσφέρει αποδοτικές κρυπτογραφικές λύσεις για συσκευές με περιορισμένους πόρους, όπως συσκευές Internet of Things (IoT), ενσωματωμένα συστήματα και αισθητήρες χαμηλής κατανάλωσης. Η οικογένεια Ascon αναδείχθηκε ως μια βιώσιμη εναλλακτική λύση σε περιπτώσεις όπου ο Advanced Encryption Standard (AES) δεν αποδίδει βέλτιστα, ανταποκρινόμενη στην αυξανόμενη ανάγκη για ελαφριά κρυπτογραφία (lightweight cryptography) σε συσκευές με περιορισμένους υπολογιστικούς και ενεργειακούς πόρους. Τα βασικά δομικά στοιχεία της οικογένειας Ascon περιλαμβάνουν την κρυπτογράφηση με έλεγχο αυθεντικότητας και συσχετισμένα δεδομένα (AEAD), τη συνάρτηση κατακερματισμού (hash function) και την επεκτάσιμη συνάρτηση εξόδου (XOF – eXtendable Output Function). Κύριο χαρακτηριστικό της οικογένειας Ascon αποτελεί η αρχιτεκτονική βασισμένη σε μεταθέσεις (permutation-based design), η οποία προσφέρει ισχυρή ασφάλεια, υψηλή αποδοτικότητα και ευελιξία, καθιστώντας την ιδανική για σύγχρονα περιβάλλοντα με περιορισμένους πόρους [27] [28]. Οι αλγόριθμοι που περιλαμβάνονται στο πρωτόκολλο **NIST SP 800-232** [27] είναι τέσσερις, ο Ascon-AEAD128, ο Ascon-Hash256, ο Ascon-XOF128 και ο Ascon-CXOF128.



Ο αλγόριθμος **Ascon-AEAD128** / Ascon-128a αποτελεί ένα σχήμα AEAD βασισμένο σε nonce, με ισχύς ασφάλειας 128 bit στο πλαίσιο χρήσης ενός μόνο κλειδιού.

Ο αλγόριθμος **Ascon-Hash256** ορίζεται ως μια κρυπτογραφική συνάρτηση κατακερματισμού που παράγει σύνοψη (hash) μήκους 256 bit για τα εισερχόμενα μηνύματα, προσφέροντας ισχύ ασφάλειας 128 bit.

Ο αλγόριθμος **Ascon-XOF128** χαρακτηρίζεται ως επεκτάσιμη συνάρτηση εξόδου (XOF), όπου το μέγεθος εξόδου μπορεί να οριστεί από τον χρήστη, με υποστηριζόμενη ισχύ ασφάλειας έως 128 bit.

Ο αλγόριθμος **Ascon-CXOF128** αποτελεί μία προσαρμοσμένη εκδοχή της XOF, που επιτρέπει στον χρήστη να καθορίσει συμβολοσειρά προσαρμογής (customization string) και μέγεθος εξόδου, με υποστηριζόμενη ισχύ ασφάλειας έως 128 bit.

Τέλος, η οικογένεια αλγορίθμων Ascon αποτελεί χαρακτηριστικό παράδειγμα sponge-based σχεδίασης, καθώς ακολουθεί τη λογική απορρόφησης (absorbing) και εκτόνωσης (squeezing). Στην αρχή απορροφά τα associated data και το nonce μέσα στο εσωτερικό state, κατά το στάδιο της Αρχικοποίησης και της Επεξεργασίας των Συνδεδεμένων Δεδομένων, ώστε αυτά να επηρεάσουν την τελική αυθεντικοποίηση, και στη συνέχεια "εκτονώνει" το state για να παραχθούν το κρυπτογράφημα και το authentication tag [28].

ΜΕΛΕΤΕΣ ΓΙΑ ΤΟΝ ASCON

Ο αλγόριθμος Ascon128a, ήδη από την επιλογή και τυποποίησή του στο πλαίσιο του διαγωνισμού NIST Lightweight Cryptography (2023), έχει αποτελέσει αντικείμενο εκτεταμένης μελέτης τόσο ως προς την απόδοσή του όσο και ως προς την ανθεκτικότητά του σε σύγχρονες και μελλοντικές μορφές επιθέσεων. Η βιβλιογραφία γύρω από τον Ascon περιλαμβάνει πλήθος ερευνητικών εργασιών, οι οποίες εξετάζουν τη συμπεριφορά του σε διαφορετικά περιβάλλοντα

υλικού και λογισμικού, καθώς και τη συγκριτική του θέση έναντι άλλων δημοφιλών συμμετρικών κρυπταλγορίθμων. Στο παρόν κεφάλαιο παρουσιάζονται συγκεντρωτικά οι σημαντικότερες προηγούμενες μελέτες που αφορούν τον αλγόριθμο Ascon, με στόχο την αξιολόγησή του σε όρους απόδοσης, ασφάλειας, κατανάλωσης πόρων και καταλληλότητας για υλοποίηση σε συστήματα IoT. Η ανάλυση οργανώνεται σε δύο κύριες κατηγορίες, τις βιβλιογραφικές και θεωρητικές ανασκοπήσεις και στις πειραματικές και εφαρμοσμένες μελέτες. Η διάκριση αυτή επιτρέπει όχι μόνο την αποτίμηση των ποσοτικών μετρήσεων, αλλά και την εξαγωγή ουσιαστικών συμπερασμάτων.

Οι Sarasha et al στην εργασία τους [29] πραγματοποίησαν μια ολοκληρωμένη αξιολόγηση της απόδοσης του Ascon σε μια σειρά από διαφορετικές πλατφόρμες Arduino, όπως Arduino DUE, Arduino Mega2560, Arduino Nano Every και Arduino Nano ESP32. Η μελέτη περιλαμβάνει ανάλυση χρόνων εκτέλεσης, κατανάλωσης πόρων και αποδοτικότητας των υλοποιήσεων, ενώ στο τέλος προσφέρεται μια λεπτομερής συγκριτική παρουσίαση των αποτελεσμάτων που αναδεικνύει τις διαφορές και τα πλεονεκτήματα κάθε πλατφόρμας. Τα συμπεράσματα έχουν ως εξής: οι τρεις εκδόσεις του Ascon έχουν καλύτερη απόδοση από το AES-128 στο Arduino DUE, με το Ascon-128a να είναι η ταχύτερη έκδοση. Ωστόσο, το AES-128 είναι σαφώς πιο αποδοτικό από οποιαδήποτε έκδοση του Ascon στο Arduino Mega2560. Τέλος, η συμπεριφορά των τριών εκδόσεων του Ascon και του AES-128 στο Arduino Nano ESP32 είναι αρκετά παρόμοια, αν και ο αλγόριθμος με τη χαμηλότερη απόδοση είναι το AES-128, ενώ το Ascon-128a εμφανίζει ελαφρώς καλύτερη επίδοση. Συνολικά, τα αποτελέσματα αυτά δείχνουν ότι η επιλογή του καταλληλότερου αλγορίθμου εξαρτάται σε μεγάλο βαθμό από τα διαθέσιμα υλικά χαρακτηριστικά της εκάστοτε συσκευής, γεγονός που υπογραμμίζει τη σημασία αξιολόγησης των κρυπτογραφικών υλοποιήσεων σε πραγματικά περιβάλλοντα χαμηλών πόρων, όπως αυτά που συναντώνται σε εφαρμογές IoT.

Οι Gewehr et al. [30] στην εργασία τους παρουσίασαν μία συγκριτική αξιολόγηση των Ascon, AES-128 σε λειτουργία CCM και ChaCha20-Poly1305 σε έναν πυρήνα RISC-V χαμηλής πολυπλοκότητας, με και χωρίς εξειδικευμένες επεκτάσεις συνόλου εντολών, αναλύοντας βελτιώσεις σε απόδοση, ενεργειακή αποδοτικότητα, χρήση μνήμης και επιβάρυνση σε επιφάνεια. Συνολικά, τα αποτελέσματα δείχνουν ότι η χρήση εξειδικευμένων επεκτάσεων συνόλου εντολών σε RISC-V συστήματα μπορεί να επιφέρει σημαντικές βελτιώσεις στην απόδοση, στην ενεργειακή αποδοτικότητα και στη χρήση μνήμης, καθιστώντας τόσο τον Ascon όσο και τον AES-128 ιδιαίτερα αποδοτικούς αλγορίθμους για ενσωματωμένα περιβάλλοντα χαμηλής πολυπλοκότητας.

Οι Kitahara et al. [31] σύγκριναν τους Ascon, Grain-128AEAD και TinyJambu, ως προς την απόδοση των υλοποιήσεων και ανέλυσαν τα χαρακτηριστικά τους βάσει των αποτελεσμάτων. Επιβεβαίωσαν ότι οι αλγόριθμοι Ascon και Xoodyak, οι οποίοι παρουσιάζουν υψηλή απόδοση σε άλλη πλατφόρμα 32-bit, υπερέχουν επίσης στο Cortex-M0. Ο Ascon αποδεικνύεται ως καταλληλότερος για σενάρια όπου το μήκος των συσχετιζόμενων δεδομένων (Associated Data – AD) είναι μικρό, καθώς η επεξεργασία AD απαιτεί μεγαλύτερο χρόνο από την επεξεργασία του

κειμένου (PT). Το TinyJambu είναι κατάλληλο ως lightweight block cipher λόγω της απλής round function και του μικρού block length. Επιβεβαίωσαν, επίσης ότι οι Ascon, Grain-128AEAD και TinyJambu, έχουν μικρότερο μέγεθος ROM από το AES-GCM.

Οι Radhakrishnan et al. [32] υλοποίησαν μία συγκριτική μελέτη μεταξύ των AES-128 και ASCON στο πλαίσιο του ίδιου πρωτοκόλλου ασφαλείας (CANsec). Οι πολύπλοκες μαθηματικές διεργασίες που απαιτεί ο AES συνεπάγονται αυξημένη επεξεργαστική ισχύ, γεγονός που αποτελεί σημαντικό μειονέκτημα σε περιβάλλοντα μειωμένων πόρων, όπως αυτά των IoT συσκευών. Αντίθετα, ο ASCON προσφέρει ισχυρό ανταγωνισμό στον AES, τόσο ως προς την ασφαλή υλοποίηση όσο και ως προς την εμπιστοσύνη που έχει αποκτήσει από την κρυπτογραφική κοινότητα, καθιστώντας τον ιδιαίτερα κατάλληλο για χρήση σε συστήματα IoT.

Ο Weng στην εργασία του "Performance and Energy Evaluation of Lightweight Cryptography for Small IoT Devices," [33] συγκρίνει τους κρυπτογραφικούς αλγορίθμους Acorn, Ascon, Speck και AES-128-GCM, με τους πρώτους τρεις να παρουσιάζουν αξιοσημείωτη αποδοτικότητα σε μικρές IoT συσκευές σε σύγκριση με τον τελευταίο.

Μελέτη	Πλατφόρμα / Περιβάλλον	Αντικείμενο Μελέτης	Κύρια Συμπεράσματα
Sarasha et al. [29]	Arduino DUE, Mega2560, Nano Every, Nano ESP32	Απόδοση Ascon (128/128a/96) vs AES-128 σε embedded πλατφόρμες	Στο Arduino DUE οι εκδόσεις του Ascon υπερέχουν του AES-128, με το Ascon-128a ταχύτερο. Στο Mega2560 ο AES-128 είναι πιο αποδοτικός. Στο Nano ESP32 οι αλγόριθμοι έχουν παρόμοια συμπεριφορά, με το AES-128 πιο αργό. Η επιλογή αλγορίθμου εξαρτάται από τα hardware χαρακτηριστικά.
Gewehr et al. [30]	RISC-V low-complexity core με/χωρίς ISA extensions	Σύγκριση Ascon, AES-128-CCM, ChaCha20-Poly1305 σε RISC-V	Οι ISA επεκτάσεις βελτιώνουν σημαντικά απόδοση/ενέργεια/μνήμη.Ascon & AES-128 γίνονται ιδιαίτερα αποδοτικοί με hardware acceleration.Κατάλληλοι για embedded περιβάλλοντα χαμηλής πολυπλοκότητας.
Kitahara et al. [31]	Cortex-M0	Σύγκριση Ascon, Grain-128AEAD, TinyJambu	Ascon και Xoodyak αποδίδουν πολύ καλά στο Cortex-M0. Ascon κατάλληλος όταν το Associated Data είναι μικρό. TinyJambu κατάλληλος ως lightweight block cipher. Και οι τρεις έχουν μικρότερο ROM footprint από AES-GCM.
Radhakrishnan et al. [32]	CANsec security protocol σε IoT context	Σύγκριση AES-128 vs ASCON σε ίδια υλοποίηση πρωτοκόλλου	Ο AES απαιτεί αυξημένη υπολογιστική ισχύ λόγω πολύπλοκων πράξεων. Ο ASCON έχει χαμηλότερο κόστος, υψηλή ασφάλεια και θεωρείται πιο κατάλληλος για IoT. Ο ASCON κερδίζει σε ισορροπία ασφάλειας–αποδοτικότητας.

Weng [33]	Μικρές IoT συσκευές	Performance & Energy Evaluation: Acorn, Ascon, Speck, AES-128-GCM	Acorn, Ascon και Speck είναι σημαντικά πιο αποδοτικοί από AES-128-GCM σε μικρά IoT devices. Το AES-GCM υστερεί χωρίς hardware acceleration. Οι lightweight αλγόριθμοι έχουν σαφές πλεονέκτημα σε low-power συστήματα.

ΕΣΩΤΕΡΙΚΗ ΔΟΜΗ ΚΑΙ ΛΕΙΤΟΥΡΓΙΑ ΤΟΥ ASCON-128a

Ο αλγόριθμος Ascon αρχικά οργανώνει τα δεδομένα εισόδου σε 320 bits τα οποία αντιστοιχούν σε 5 λέξεις των 64 bits (8 bytes), δηλαδή $S = S_0||S_1||S_2||S_3||S_4$, με $S_0 = \{S_0[0], S_0[1], \dots, S_0[7]\}$, … , $S_4 = \{S_4[0], S_4[1], \dots, S_4[7]\}$, με $S_k[L] = (a_0, a_1, \dots, a_7)$, $k\epsilon[0,4]$ και $L\epsilon[0,7]$ [27].

Η οικογένεια Ascon χρησιμοποιεί κάποιες βασικές συναρτήσεις στη λειτουργίας της, οι οποίες αναλύονται στα επόμενα.

ΒΑΣΙΚΕΣ ΣΥΝΑΡΤΗΣΕΙΣ

PARSE FUNCTION parse(X,r)

Σπάει το bitstring X σε μπλοκ μήκους r. Τα πλήρη μπλοκ είναι $X_0, X_1, \dots, X_{l-1}, \underline{X_l}$ και το τελευταίο $\underline{X_l}$ μπορεί να είναι μικρότερο ή κενό αν $|X|mod r = 0$. Το μήκος των block εξαρτάται από το μέλος της οικογένειας που χρησιμοποιείται κάθε φορά, ο Ascon128a χρησιμοποιεί rate=128.

PADDING FUNCTION pad(X,r)

Προσθέτει bits στο τέλος του τελευταίου μπλοκ ώστε να γίνει πολλαπλάσιο του r. Η διαδικασία που ακολουθείται έχει ως εξής προσθέτει αρχικά το bit 1 στο τέλος του bitstring X και ακολουθείται από 0^j , με $j = (-|X| - 1) mod r$ και το bitstring γίνεται $X' = X||1||0^j$.

ΜΕΤΑΘΕΣΕΙΣ ΣΤΟΝ ΑΛΓΟΡΙΘΜΟ ASCON

Κάθε γύρος εφαρμόζει τρεις επιμέρους λειτουργίες, πάντα με αυτή τη σειρά $P = p_L \circ p_S \circ p_C$, η μετατροπή p_C (Constant Addition), η μετατροπή p_s (Substitution) και την p_L(Linear Diffusion) και πραγματοποιεί από 1 έως 16 γύρους. Οι μετατροπές περιγράφονται αναλυτικά παρακάτω.

$$P_C \longrightarrow P_S \longrightarrow P_L$$

ROUND FUNCTION - CONSTANT ADDITION p_C

Η πρώτη μετατροπή περιλαμβάνει μετατροπή στη τρίτη λέξη του αρχικού S, εφαρμόζοντας την πράξη $S2 = S2 \oplus c\_i$, όπου c_i αποτελεί μία γνωστή τιμή η οποία αλλάζει σε κάθε γύρο σύμφωνα με τον τύπο $c_i = const_{16-rnd+i}$ πίνακα:

i	**const$_i$ (εξάδες)**
0	0x000000000000003c
1	0x000000000000002d
2	0x000000000000001e
3	0x000000000000000f
4	0x00000000000000f0
5	0x00000000000000e1
6	0x00000000000000d2
7	0x00000000000000c3
8	0x00000000000000b4
9	0x00000000000000a5
10	0x0000000000000096
11	0x0000000000000087

12	0x0000000000000078
13	0x0000000000000069
14	0x000000000000005a
15	0x000000000000004b

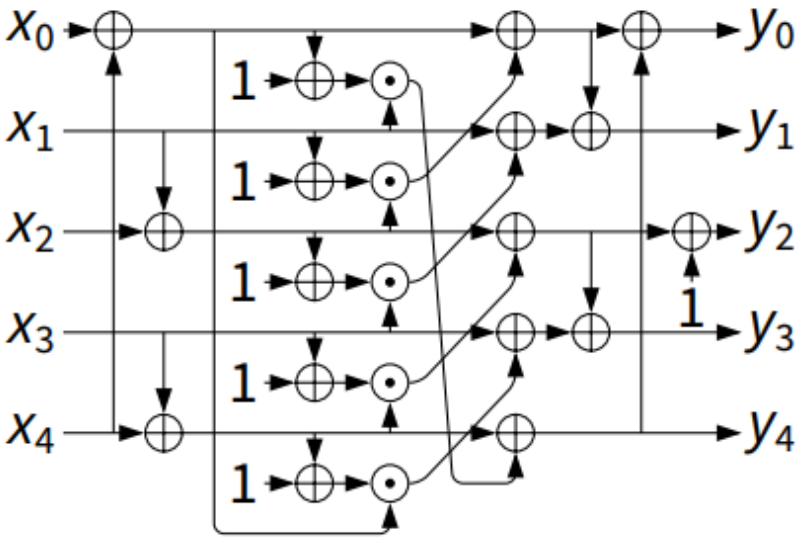
Οι αλγόριθμοι Ascon-p[16], χρησιμοποιεί 16 γύρους ισχύει, δηλαδή round=16, $c_i = const_{16-16+i} = const_i$ χρησιμοποιεί όλους τους όρους του πίνακα.

Οι αλγόριθμοι Ascon-p[12], χρησιμοποιεί 12 γύρους ισχύει, δηλαδή round=12, $c_i = const_{16-12+i} = const_{4+i}$ και χρησιμοποιούνται οι σταθερές $const_4$ έως $const_{15}$.

Οι αλγόριθμοι Ascon-p[8], χρησιμοποιεί 8 γύρους ισχύει, δηλαδή round=8, $c_i = const_{16-8+i} = const_{8+i}$ και χρησιμοποιούνται οι σταθερές $const_8$ έως $const_{15}$.

SUBSTITUTION -S-BOX- p_S

Έστω $S = (S_0, S_1, S_2, S_3, S_4)$. Αρχικά επιλέγεται το j bit, $j\epsilon[0-63]$, από κάθε λέξη S_i με $i\epsilon[1-5]$, δηλαδή $(S_0(0,j), S_1(1,j), S_2(2,j), S_3(3,j), S_4(4,j))$. Αυτές οι τιμές περνούν από ένα S-box και προκύπτουν οι τιμές $(y_0, y_1, y_2, y_3, y_4)$ οι οποίες προκύπτουν ως εξής:



Άρα, προκύπτει ότι:

$$y_0 = x_4 \cdot x_1 \oplus x_3 \oplus x_2 \cdot x_1 \oplus x_2 \oplus x_1 \cdot x_0 \oplus x_1 \oplus x_0$$
$$y_1 = x_4 \oplus x_3 \cdot x_2 \oplus x_3 \cdot x_1 \oplus x_3 \oplus x_2 \cdot x_1 \oplus x_2 \oplus x_1 \oplus x_0$$
$$y_2 = x_4 \cdot x_3 \oplus x_4 \oplus x_2 \oplus x_1 \oplus 1$$

$$y_3 = x_4 \cdot x_0 \oplus x_4 \oplus x_3 \cdot x_0 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$
$$y_4 = x_4 \cdot x_1 \oplus x_4 \oplus x_3 \oplus x_1 \cdot x_0 \oplus x_1$$

Ο πίνακας S-box σύμφωνα με τον NIST έχει την παρακάτω μορφή:

x (hex)	x (binary)	SBOX(x) (hex)	SBOX(x) (binary)
0	**00000**	**04**	**00100**
1	**00001**	**0B**	**01011**
2	**00010**	**1F**	**11111**
3	**00011**	**14**	**10100**
4	**00100**	**1A**	**11010**
5	**00101**	**15**	**10101**
6	**00110**	**09**	**01001**
7	**00111**	**02**	**00010**
8	**01000**	**1B**	**11011**
9	**01001**	**05**	**00101**
A	**01010**	**08**	**01000**
B	**01011**	**12**	**10010**
C	**01100**	**1D**	**11101**
D	**01101**	**03**	**00011**
E	**01110**	**06**	**00110**
F	**01111**	**1C**	**11100**
10	**10000**	**1E**	**11110**
11	**10001**	**13**	**10011**
12	**10010**	**07**	**00111**
13	**10011**	**0E**	**01110**
14	**10100**	**00**	**00000**
15	**10101**	**0D**	**01101**
16	**10110**	**11**	**10001**
17	**10111**	**18**	**11000**
18	**11000**	**10**	**10000**
19	**11001**	**0C**	**01100**
1A	**11010**	**01**	**00001**
1B	**11011**	**19**	**11001**
1C	**11100**	**16**	**10110**
1D	**11101**	**0A**	**01010**
1E	**11110**	**0F**	**01111**
1F	**11111**	**17**	**10111**

linear diffusion layer - p_L

Για κάθε 64-bit λέξη Si φτιάχνεις το $\Sigma_i(S_i)$ με **XOR** του ίδιου του Si και δύο **κυκλικών δεξιών περιστροφών** (right rotations, σημειώνονται με >>>), σύμφωνα με τις σχέσεις:

$\Sigma_0(S_0) = S_0 \oplus (S_0 >>> 19) \oplus (S_0 >>> 28)$

$\Sigma_1(S_1) = S_1 \oplus (S_1 >>> 61) \oplus (S_1 >>> 39)$

$\Sigma_2(S_2) = S_2 \oplus (S_2 >>> 1) \oplus (S_2 >>> 6)$

$\Sigma_3(S_3) = S_3 \oplus (S_3 >>> 10) \oplus (S_3 >>> 17)$

$\Sigma_4(S_4) = S_4 \oplus (S_4 >>> 7) \oplus (S_4 >>> 41)$

ASCON-AEAD128a

Ascon-AEAD128.enc — Encryption

Η συνάρτηση κρυπτογράφησης ορίζεται ως εξής: $Ascon - AEAD128.enc(K, N, A, P) = (C, T)$.
Παίρνει ως εισόδους:

- το **κλειδί** K (128 bits),
- το **nonce** N (128 bits),
  τα **associated data** A (μηνύματα που δεν κρυπτογραφούνται αλλά πρέπει να αυθεντικοποιηθούν),
- και το **plaintext** P.

Επιστρέφει ως εξόδους :

- το **ciphertext** C (ίδιο μήκος με το P),
- και το **authentication tag** T (128 bits).

Ο αλγόριθμος Ascon-AEAD128 στο πλαίσιο της κρυπτογράφησης περιλαμβάνει 4 φάσεις την αρχικοποίηση, την επεξεργασία των Associated Data, την Κρυπτογράφηση και το τελικό στάδιο. Η διαδικασία Ascon-p [rnd] γίνεται ακριβώς με την ίδια σειρά όπως και στην διαδικασία της κρυπτογράφησης.

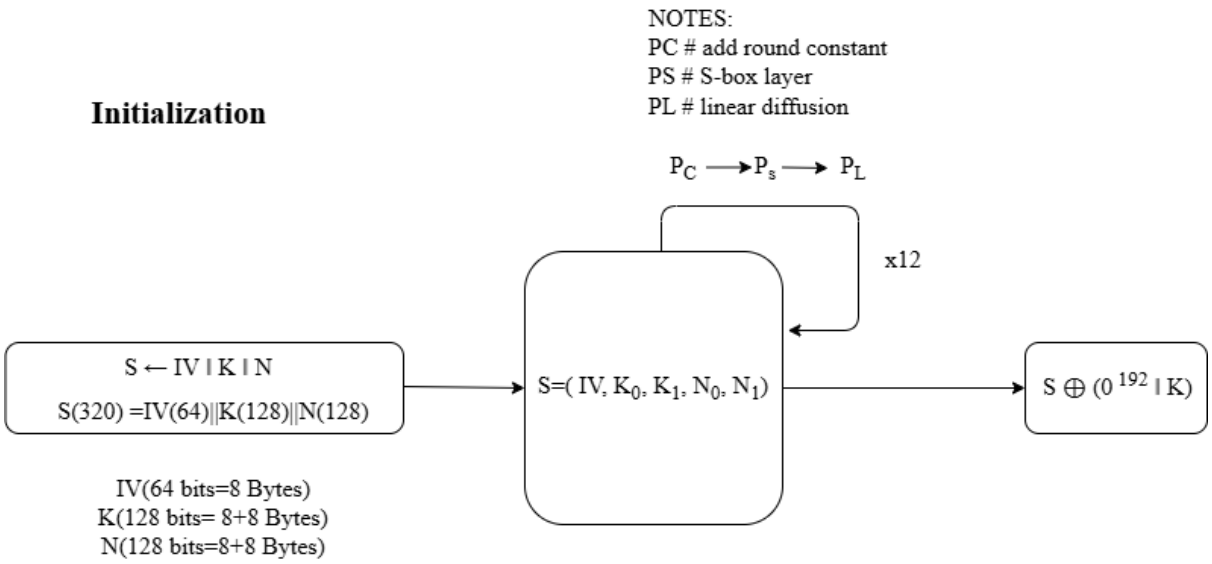
INITIALIZATION (ΑΡΧΙΚΟΠΟΙΗΣΗ)

Στο στάδιο της αρχικοποίησης οι Είσοδοι είναι το IV (σταθερά 64 bits), το κλειδί Key K (128 bits) και τέλος το Nonce N (128 bits). Αυτά συνδυάζονται σε ένα state 320 bits με S=IV‖K‖N και εφαρμόζεται η Ascon-p[12] permutation (12 γύροι).
Σύμφωνα με τον ψευδοκώδικα ισχύει ότι:
IV ← 0x00001000808c0001 : Σταθερά αρχικοποίησης 64-bit (κωδικοποιεί παραμέτρους αλγορίθμου).

S ← IV ‖ K ‖ N : Φτιάχνεις το 320-bit state: 64 (IV) ‖ 128 (K) ‖ 128 (N).

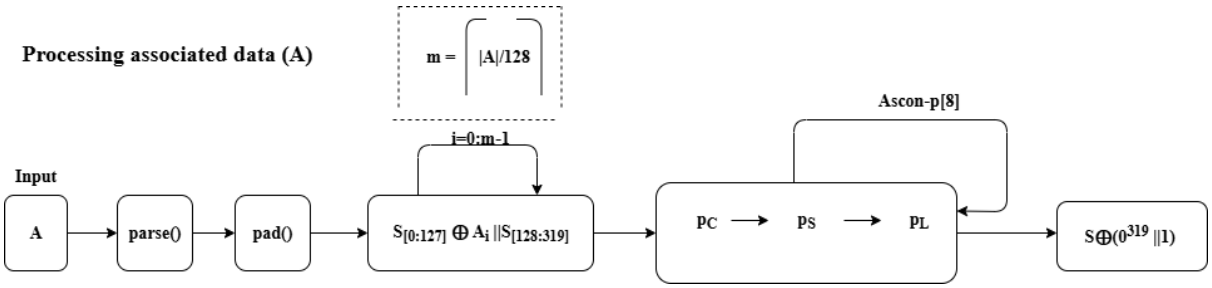
S ← Ascon-p12 : Επαναλαμβάνει την permutation με 12 γύρους.

$S \leftarrow S \oplus (\boldsymbol{0^{192}} \| K)$ **:** Εισάγεται εκ νέου το κλειδί (XOR) στα **τελευταία 128 bits** του state.
Η είσοδος στην αρχικοποίηση αποτελεί το $S = (IV, K_0, K_1, N_0, N_1)$
Το στάδιο της αρχικοποίησης παρουσιάζεται στο παρακάτω διάγραμμα:



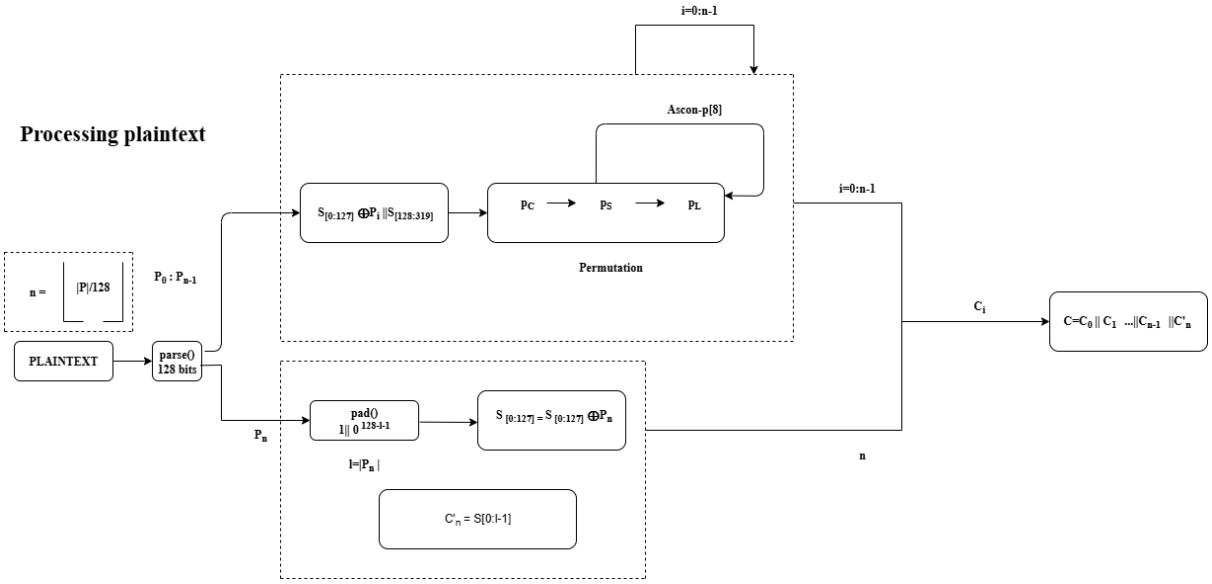
PROCESSING OF ASSOCIATED DATA (ΕΠΕΞΕΡΓΑΣΙΑ ΤΩΝ ASSOCIATED DATA)

Τα Associated Data (A) είναι δεδομένα αυθεντικοποίησης τα οποία δεν κρυπτογραφούνται, επομένως δεν είναι μυστικά. Επεξεργάζονται από τον Ascon πριν από το μήνυμα, ώστε να επηρεάσουν την τελική ετικέτα (authentication tag) και, συνεπώς, την εγκυρότητα του μηνύματος. Στο στάδιο αυτό, επεξεργάζονται τα Associated Data (A) ως εξής: Αρχικά, με τη χρήση της συνάρτησης parse(), τα A χωρίζονται σε blocks των 128 bits. Έπειτα εφαρμόζεται η συνάρτηση pad() ώστε να πραγματοποιηθεί το κατάλληλο γέμισμα στο τελευταίο block, εφόσον αυτό δεν είναι πλήρες. Στη συνέχεια, πραγματοποιούνται διαδοχικές m πράξεις σύμφωνα με τον τύπο $S_{[0:127]} \oplus A_i || S_{[128:319]}$, με $i \epsilon [0: m-1]$, όπου το m υπολογίζεται ως το πλήθος των επαναλήψεων και είναι ίσο με το ceiling του $\frac{|A|}{128}$. Έπειτα, το αποτέλεσμα μετά τα διαδοχικά XOR το $S'_{[0:319]}$ εισάγεται στο Ascon-p και πραγματοποιείται επανάληψη 8 φορές, για το Ascon128a. Τέλος, πραγματοποιείται η πραξη $S \oplus (0^{319} || 1)$. Η φάση **Processing Associated Data (A)** στον αλγόριθμο **Ascon-128a** είναι υπεύθυνη για την απορρόφηση των δεδομένων αυθεντικοποίησης στο εσωτερικό state του αλγορίθμου. Η συνολική διαδικασία απεικονίζεται παρακάτω σε μορφή διαγράμματος:



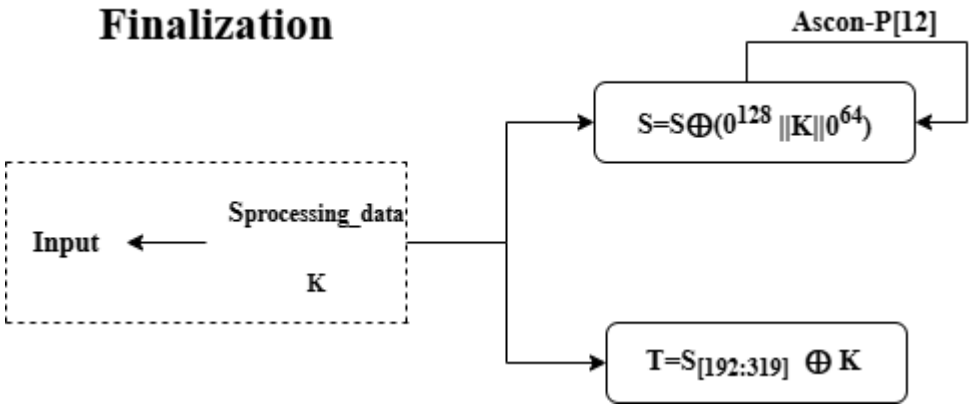
PROCESSING OF PLAINTEXT (ΚΡΥΠΤΟΓΡΑΦΗΣΗ)

Αρχικά, με τη χρήση της συνάρτησης parse(), τα Plaintext ($P = P_0||P_1||\dots||\underline{P_n}$) χωρίζονται σε blocks των 128 bits. Στα block τα οποία είναι πλήρη, δηλαδή έχουν μέγεθος 128 εφαρμόζεται ο τύπος $C_i = S_{[0:127]} \oplus P_i$. Το $C_i = S_{[0:127]} \oplus P_i||S_{[128:319]}$ εισάγεται στο Ascon-p[8]. Η παραπάνω διαδικασία εφαρμόζεται τόσες φορές όσα είναι τα πλήρη block , δηλαδή το floor του $n = \frac{|P|}{128}$. Στο τελευταίο ελλιπές block, αν υπάρχει εφαρμόζεται αρχικά padding και έπειτα γίνεται χρήση της σχέσης $C' = S_{[0:127]} \oplus P_n$. Το τελικό C προκύπτει $C = C_0||C_1\dots||C_{n-1}||C'_n$

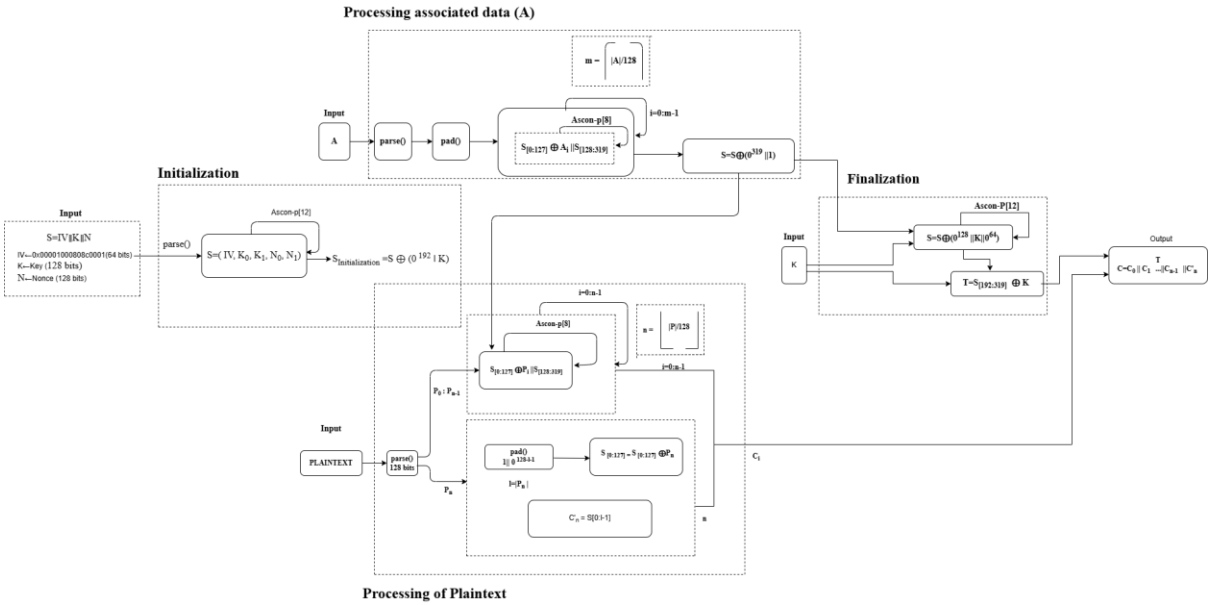


FINALIZATION (ΤΕΛΙΚΟ ΣΤΑΔΙΟ)

Στο τελικό στάδιο της διαδικασίας ως input ορίζεται το S το αποτέλεσμα του προηγούμενου σταδίου, δηλαδή του και υπολογίζονται τα S και P σύμφωνα με τις σχέσεις: $S = S \oplus (\boldsymbol{0^{128}}||K||\boldsymbol{0^{64}})$ και $T = S_{[192:319]} \oplus K$. Τέλος, στο S εφαρμόζεται Ascon-p[12]. Η διαδικασία του τελικού σταδίου αποτυπώνεται στο παρακάτω διάγραμμα.



ΤΕΛΙΚΟ ΔΙΑΓΡΑΜΜΑ Ascon-AEAD128 ΚΡΥΠΤΟΓΡΑΦΗΣΗ



Ascon-AEAD128.dec — decryption

INITIALIZATION (ΑΡΧΙΚΟΠΟΙΗΣΗ)

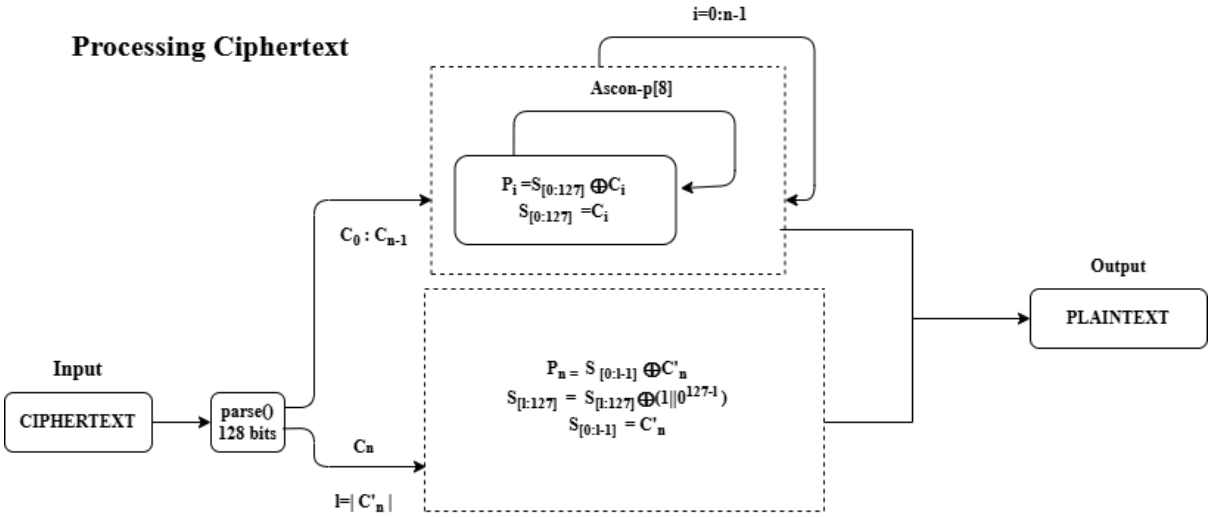
Η διαδικασία του Initialization είναι ακριβώς ίδια με αυτή της κρυπτογράφησης και έχει παρουσιαστεί στα προηγούμενα.

PROCESSING OF ASSOCIATED DATA (ΕΠΕΞΕΡΓΑΣΙΑ ΤΩΝ ASSOCIATED DATA)

Η διαδικασία του Processing associated data είναι ακριβώς ίδια με αυτή της κρυπτογράφησης και έχει παρουσιαστεί στα προηγούμενα.

PROCESSING OF CIPHERTEXT (ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ)

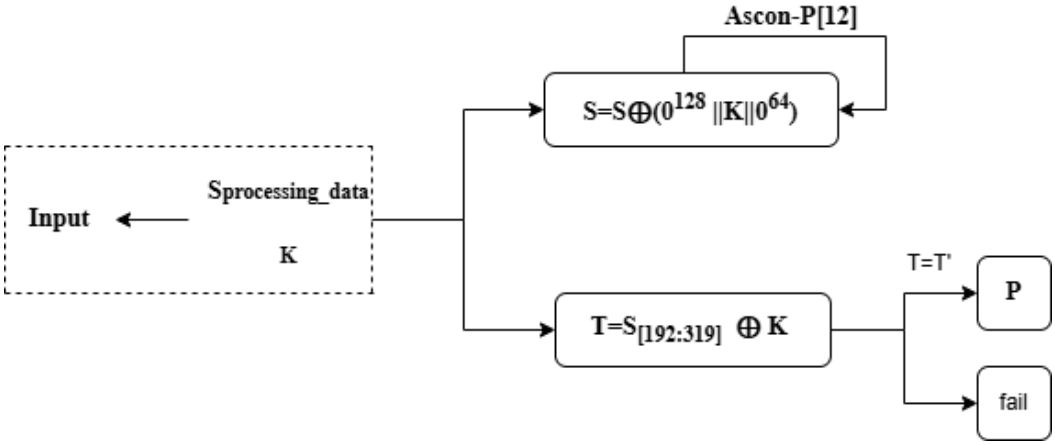
Το κρυπτογραφημένο μήνυμα C εισάγεται αρχικά σε μία συνάρτηση **parse()**, η οποία το χωρίζει σε τμήματα των 128 bits, $C_{\blacksquare} = C_1||C_2||\dots||C'_i, με\ i = ceil(\frac{n}{128})$. Για κάθε block $C_i\ με\ i\epsilon 0: n-1$, εφαρμόζεται $P_i = S[\boldsymbol{0}: \boldsymbol{127}] \oplus C_i\ και\ S[\boldsymbol{0}: \boldsymbol{127}] = C_i$ και εσωτερικά πραγματοποιείται Ascon-p[8]. Για το τελευταίο, μερικό block C_n', εφαρμόζεται ξεχωριστή επεξεργασία που περιλαμβάνει padding και ενημέρωση του state πριν το τελικό στάδιο, σύμφωνα με τους τύπους $P_n = S[\boldsymbol{0}: l-\boldsymbol{1}] \oplus C_n', S_{[l:127]} = S_{[l:127]} \oplus (\boldsymbol{1}||\boldsymbol{0}^{127-l}), S_{[0:l-1]} = C_n'$.



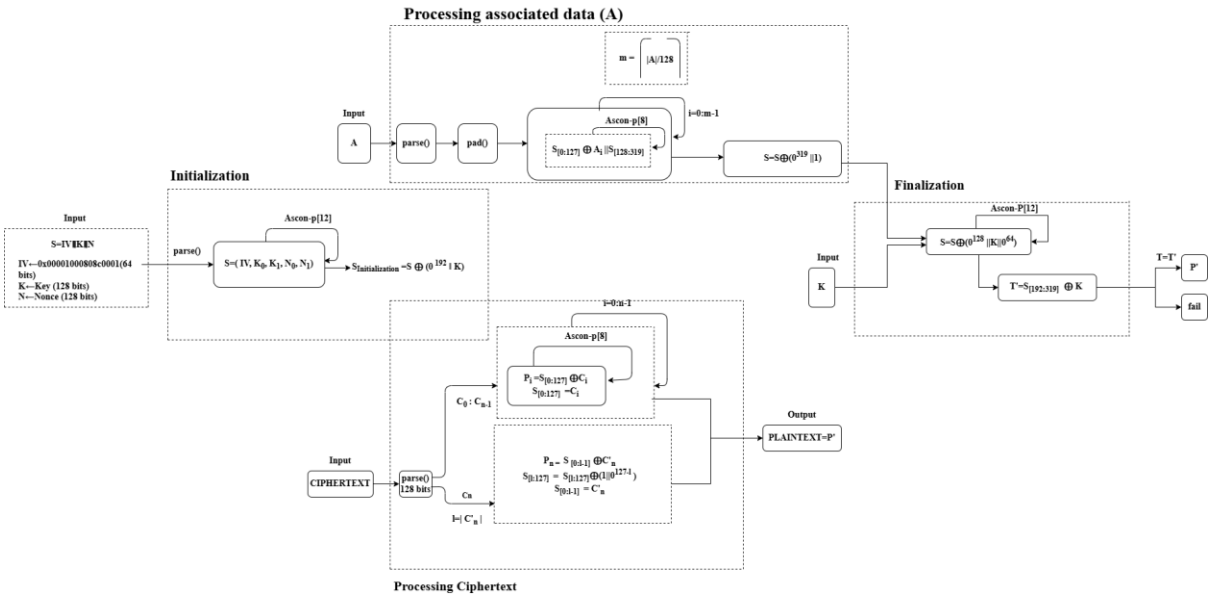
FINALIZATION (ΤΕΛΙΚΟ ΣΤΑΔΙΟ)

Στο τελικό στάδιο της διαδικασίας ως input ορίζεται το S το αποτέλεσμα του προηγούμενου σταδίου, δηλαδή του και υπολογίζονται τα S και P σύμφωνα με τις σχέσεις: $S = S \oplus (\boldsymbol{0}^{\boldsymbol{128}}||K||\boldsymbol{0}^{\boldsymbol{64}})$ και $T' = S_{[192:319]} \oplus K$. Τέλος, στο S εφαρμόζεται Ascon-p[12]. Αν $T = T'$ Η διαδικασία του τελικού σταδίου αποτυπώνεται στο παρακάτω διάγραμμα.

Finalization



ΤΕΛΙΚΟ ΔΙΑΓΡΑΜΜΑ Ascon-AEAD128 ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ



ΠΑΡΑΔΕΙΓΜΑ ΜΕ ΓΝΩΣΤΟ PLAINTEXT

ΚΡΥΠΤΟΓΡΑΦΗΣΗ

INITIALIZATION (ΑΡΧΙΚΟΠΟΙΗΣΗ)

Το input της **Αρχικοποίησης** διαμορφώνεται ως εξής: **S ← IV ‖ K ‖ N,** δηλαδή `320-bit state: 64 (IV) ‖ 128 (K) ‖ 128 (N).`

IV ← 0x00001000808c0001 : Σταθερά αρχικοποίησης 64-bit (Σύμφωνα με το NIST, έχει ορίσει ως IV αυτή τη σταθερά).
K ← A3B1C9F02E4D7F8A1234567890ABCDEF (128-bit, hex, τυχαίο κλειδί).
N ← (128-bit — π.χ. counter ή τυχαίο): N = 00000000000000000000000000000001 (counter) **ή** N = 4F3A2C1B9E8D7F60123456789ABCDEF0 (τυχαίο)

1 ΒΗΜΑ: Εφαρμόζεται η συνάρτηση parse στο S. Άρα, το S διαμορφώνεται ως εξής:
S=(0x00001000808c0001||A3B1C9F02E4D7F8A||1234567890ABCDEF||0000000000000000 ||0000000000000001)
Έπειτα εφαρμόζεται Ascon-p ως εξής: $P_C \rightarrow P_S \rightarrow P_L$, 12 φορές.

Η συνάρτηση P_C

Σύμφωνα με τον τύπο $c_i = const_{16-rnd+i}$ για rnd=12 η P_C διαμορφώνεται $c_i = const_{16-12+i} = const_{4+i}$. . Άρα, η συνάρτηση χρησιμοποιεί τα $const_4$ έως $const_{15}$, σύμφωνα με τον πίνακα των c_i.

Εφαρμόζεται στο $S = S_0||S_1||S_2||S_3||S_4$, με $S_2 = S_2 \oplus c_i$.

Ως $S_2 = (1234567890ABCDEF)$ ορίζεται αυτό και γίνεται XOR με το αντίστοιχο c_i.

Η συνάρτηση P_S

Για κάθε $S = S_0||S_1||S_2||S_3||S_4$. Για $j = \boldsymbol{0,1,\dots,63}$, ορίζονται $S_0[j], S_1[j], S_2[j], S_3[j], S_4[j]$. Εφαρμόζει για j=0 και παίρνει τα πέντε πρώτα bits από κάθε λέξη $S_0[\boldsymbol{0}], S_1[\boldsymbol{0}], S_2[\boldsymbol{0}], S_3[\boldsymbol{0}], S_4[\boldsymbol{0}]$,.., μέχρι j=63 , $S_0[\boldsymbol{63}], S_1[\boldsymbol{63}], S_2[\boldsymbol{63}], S_3[\boldsymbol{63}], S_4[\boldsymbol{63}]$.

Με βασική σχέση την εξής:

$$y_0 = x_4 \cdot x_1 \oplus x_3 \oplus x_2 \cdot x_1 \oplus x_2 \oplus x_1 \cdot x_0 \oplus x_1 \oplus x_0$$
$$y_1 = x_4 \oplus x_3 \cdot x_2 \oplus x_3 \cdot x_1 \oplus x_3 \oplus x_2 \cdot x_1 \oplus x_2 \oplus x_1 \oplus x_0$$
$$y_2 = x_4 \cdot x_3 \oplus x_4 \oplus x_2 \oplus x_1 \oplus 1$$
$$y_3 = x_4 \cdot x_0 \oplus x_4 \oplus x_3 \cdot x_0 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$
$$y_4 = x_4 \cdot x_1 \oplus x_4 \oplus x_3 \oplus x_1 \cdot x_0 \oplus x_1$$

Για $j = 0$ ορίζεται το $S_0[\boldsymbol{0}] = x_0, S_1[\boldsymbol{0}] = x_1, S_2[\boldsymbol{0}] = x_2, S_3[\boldsymbol{0}] = x_3, S_4[\boldsymbol{0}] = x_4$.

Η συνάρτηση P_L

Στην **πρώτη** επανάληψη για i=0 γίνεται:

Το $S_2 = (1234567890ABCDEF)$ και $S_2 = S_2 \oplus c_0$.

$S'_2 = 0x1234567890ABCDEF \oplus 0x00000000000000F0 = 0x1234567890ABCB1F$.

Έπειτα το S'_2 προσαρμόζεται στο καινούργιο $S = S_0||S_1||S'_2||S_3||S_4$.

Άρα, το S διαμορφώνεται ως εξής:

S=(0x00001000808c0001||A3B1C9F02E4D7F8A||*0x1234567890ABCB1F*||0000000000000000||0000000000000001)

Η συνάρτηση P_S

Για ευκολία θα ορίσουμε το S ως ένα πίνακα 5x64. Η αρίθμηση των bits σε κάθε λέξη του state ακολουθεί little-endian σειρά, δηλαδή ξεκινά από τα δεξιά (το λιγότερο σημαντικό bit).

	bit_0	bit_1	…	bit_62	bit_63
S_0	S_0[0]	S_0[1]	…	S_0[62]	S_0[63]
S_1	S_1[0]	S_1[1]	…	S_1[62]	S_1[63]
S_2	S_2[0]	S_2[1]	…	S_2[62]	S_2[63]
S_3	S_3[0]	S_3[1]	…	S_3[62]	S_3[63]
S_4	S_4[0]	S_4[1]		S_4[62]	S_4[63]

Σε αυτό το βήμα εφαρμόζει SBOX κατά στήλη,

Για $j = \boldsymbol{0,1,\ldots,63}$, με $S_i[j]$.

Για $j = 0$

$S\_0[0] = x_0$	$S\_1[0] = x_1$	$S\_2[0] = x_2$	$S\_3[0] = x_3$	$S\_4[0] = x_4$

Άρα, $(x_0, x_1, x_2, x_3, x_4)^T = (1,0,1,0,1)^T$ και η έξοδος διαμορφώνεται ως εξής: $(x_0, x_1, x_2, x_3, x_4)^T = (0,1,1,0,1)^T$.

Οι πράξεις διαμορφώνονται σύμφωνα με τους τύπους:

$$y_0 = 1 \cdot 0 \oplus 0 \oplus 1 \cdot 0 \oplus 1 \oplus 0 \cdot 1 \oplus 0 \oplus 1 = 0$$

Όμοια υπολογίζονται και τα υπόλοιπα και η διαδικασία επαναλαμβάνεται μέχρι j=63, δηλαδή όλες οι πεντάδες περνάνε από αυτή τη διαδικασία.

$$S_0' = 0xB3B5CFF83E6FFF9E$$
$$S_1' = 0xB3B5CFF83E63FF9F$$
$$S_2' = 0x4E7A607741194B6B$$
$$S_3' = 0xB1858F883E6AB494$$

και

$S_4' = 0xA3B1C9F02E417F8B$.

.

<u>Η συνάρτηση P_L</u>

Ως είσοδος ορίζεται τα $S_0', S_1', S_2', S_3', S_4'$

$$\Sigma_0(S'_0) = S'_0 \oplus (S'_0 >>> 19) \oplus (S'_0 >>> 28)$$

$$\Sigma_1(S'_1) = S'_1 \oplus (S'_1 >>> 61) \oplus (S'_1 >>> 39)$$

$$\Sigma_2(S'_2) = S'_2 \oplus (S'_2 >>> 1) \oplus (S'_2 >>> 6)$$

$$\Sigma_3(S'_3) = S'_3 \oplus (S'_3 >>> 10) \oplus (S'_3 >>> 17)$$

$$\Sigma_4(S'_4) = S'_4 \oplus (S'_4 >>> 7) \oplus (S'_4 >>> 41).$$

$S'_0 = 10110011\ 10110101\ 11001111\ 11111000\ 00111110\ 01101111\ 11111111\ 10011110$

Το οποίο αποτελεί και την είσοδο στο $\Sigma_0(S_0)$.

$$\Sigma_0(S'_0) = 101010101011100111100000011001011011110011001100000001111101000 0$$

ή $\Sigma_0(S'_0) = 0xAAB9E065BCCC07D0$.

Όμοια και στα υπόλοιπα.

$$\Sigma_1(S'_1) = F8E28C07A939FA4C$$

$$\Sigma_2(S'_2) = 1E7C3F72E3A53977$$

$$\Sigma_3(S'_3) = C477DB1DF56903A5$$

$$\Sigma_4(S'_4) = A9233C707E756DB7$$

Τελικό μέρος του Initialization :

$$S'_{Initialization} = S'' \oplus (0^{192}||K)$$

$$S''_i = (AAB9E065BCCC07D0||F8E28C07A939FA4C||1E7C3F72E3A53977||$$

$$C477DB1DF56903A5||A9233C707E756DB7)$$

Πραγματοποιείται XOR με το $(0^{192}||K)$, όπου $K = A3B1C9F02E4D7F8A1234567890ABCDEF$.

$$S'_{Initialization} = (AAB9E065BCCC07D0||F8E28C07A939FA4C||$$

$1E7C3F72E3A53977||67C614EDDB2C7C2F||BB172A08EEDEA050)$.

PROCESSING OF ASSOCIATED DATA (ΕΠΕΞΕΡΓΑΣΙΑ ΤΩΝ ASSOCIATED DATA)

Τα Associated Data **(A)** δεν αποτελούν τυχαία συμβολοσειρά, αλλά ορίζονται από το πρωτόκολλο ή το σύστημα που κάνει χρήση του αλγορίθμου Ascon. Στην παρούσα υλοποίηση, για λόγους απλότητας και ενδεικτικού παραδείγματος, θεωρείται ότι τα A αντιστοιχούν στο ID μιας συσκευής IoT, το οποίο έχει σταθερή τιμή και αποτελεί γνωστή πληροφορία. Είναι σημαντικό να σημειωθεί ότι το A δεν αποτελεί μυστικό δεδομένο, δεν κρυπτογραφείται, αλλά συμμετέχει στη διαδικασία αυθεντικοποίησης ώστε να διασφαλιστεί η ακεραιότητα και η ταυτότητα της συσκευής.

Έστω $A = ID_{IoT} = 1122334455667788$.

Εφαρμόζεται αρχικά **parse** και **pad** στα **Associated Data (A)** και παίρνει την παρακάτω μορφή:

$A' = (\mathbf{1122334455667788||8000000000000000})$.
Το S που προκύπτει σύμφωνα με τον τύπο$S_{[0:127]} \oplus A'_i||S_{[128:319]}$.

$$(0xAAB9E065BCCC07D0||0xF8E28C07A939FA4C) \oplus$$

$$(\mathbf{1122334455667788||8000000000000000}) = (\mathbf{BB9BD321E9AA7058||78E28C07A939FA4C})$$

Άρα, τελικά η συνάρτηση Ascon-p[8] εφαρμόζει 8 γύρους της εσωτερικής permutation

(pC + pS + pL) πάνω στο state :

$$(BB9BD321E9AA7058||78E28C07A939FA4C||$$

$$1E7C3F72E3A53977||67C614EDDB2C7C2F||BB172A08EEDEA050)\ .$$

Και το τελικό S μετά τους 8 γύρους διαμορφώνεται ως εξής:

$$S = (\mathbf{44}EF\mathbf{36}F\mathbf{58}A\mathbf{702}B\mathbf{02}||BFBC\mathbf{33358}AC\mathbf{2}DA2F||\mathbf{2B109}D\mathbf{61}C3FC3E\mathbf{48}$$
$$\mathbf{499}CC\mathbf{8780}C\mathbf{81}C4D\mathbf{2}||\mathbf{25822}D\mathbf{16}D7EF\mathbf{97}A\mathbf{2})$$

PROCESSING OF PLAINTEXT (ΚΡΥΠΤΟΓΡΑΦΗΣΗ)

Για λόγους εκπαιδευτικούς ορίζεται ως plaintext το P = "Hello Ascon Example Encryption Test!"
=(48656C6C6F204173636F6E204578616D706C6520456E6372797074696F6E205465737421)

Αρχικά εφαρμόζεται η συνάρτηση parse στο plaintext P και ορίζεται $n = 2$

$$P_0 = (48656C6C6F204173636F6E204578616D)$$

$$P_1 = (706C6520456E6372797074696F6E2054)$$

$P_2 = (65737421)$

Σύμφωνα με τον τύπο $C_i = S_{[0:127]} \oplus P_i$ εφαρμόζεται Ascon-p 8 φορές στα επιμέρους P_i, τα οποία έχουν πλήρες μέγεθος, όπου $S_{[0:127]}$, όπως αυτά έχουν προκύψει από το Processing Associated Data.

$$C_0 = S_{[0:127]} \oplus P_0 = (0C8A5A99E5506A71\ DCD35D15CFBABB42)$$

$$C_1 = S_{[0:127]} \oplus P_1 = (7D81E2E82D5050E5\ D53540E71CA7810A)$$

Το $P_2 = (65737421)$ αποτελεί ένα ελλιπές τμήμα και ακολουθείται διαφορετική διαδικασία, αρχικά εφαρμόζεται padding και έπειτα XOR με το S.

$$P'_2 = (6573742180000000000000000000000)$$

$$C_2 = S_{[0:127]} \oplus P'_2 = (8D4C5145940AE4F46E4FD0DEA56DF204)$$

Τελικά, στο κρυπτογραφημένο μήνυμα επιλέγονται τα πρώτα $|C_2|$ στοιχεία.

Άρα, $C_2 = (8D4C5145)$

Το τελικό κρυπτογραφημένο μήνυμα είναι το εξής:

$C = (0C8A5A99E5506A71\ DCD35D15CFBABB427D81E2E82D5050E5\ D53540E71CA7810A$

$8D4C5145)$

FINALIZATION (ΤΕΛΙΚΟ ΣΤΑΔΙΟ)

Στο τελικό στάδιο της κρυπτογράφησης παράγεται το tag. Ως είσοδοι ορίζονται τα S και K, όπως αυτά αποτυπώνονται παρακάτω:

$$S = (8D4C5145940AE4F4 \parallel 6E4FD0DEA56DF204 \parallel 96083E0E2436086A \parallel 5B2FF61296E93089 \parallel 8AAFD78662CDF737)$$

$$K = (A3B1C9F02E4D7F8A\ 1234567890ABCDEF)$$

$$S = S \oplus (\boldsymbol{0^{128}} ||K|| \boldsymbol{0^{64}}) = (AEE\boldsymbol{387725}FC\boldsymbol{8}BB77 \parallel BDE\boldsymbol{0E67A424}FF\boldsymbol{54}C \parallel \boldsymbol{68}F727CD\boldsymbol{87D91722} \parallel 7156D285D96CA077 \parallel 13583E037D20FFA4)$$

Και το tag προκύπτει από τη σχέση $T = S_{[192:319]} \oplus K$.

Άρα, προκύπτει ότι $T = (D2E71B75F721DFFD016C687BED8B324B)$.

Το output της συνολικής κρυπτογραφικής αποτελεί το C και το T.

ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ

Ο παραλήπτης λαμβάνει το ζεύγος (C, T) καθώς επίσης τα A και N. Το κλειδί αποτελεί γνωστή μυστική πληροφορία μεταξύ των δύο μερών και η ανταλλαγή του έχει πραγματοποιηθεί σε προηγούμενο χρόνο.

INITIALIZATION (ΑΡΧΙΚΟΠΟΙΗΣΗ)

Η αρχικοποίηση πραγματοποιείται όπως ακριβώς και στο στάδιο της κρυπτογράφησης. Δηλαδή,

$$S'_{Initialization} = (AAB9E065BCCC07D0||F8E28C07A939FA4C||$$

$1E7C3F72E3A53977||67C614EDDB2C7C2F||BB172A08EEDEA050)$.

PROCESSING OF ASSOCIATED DATA (ΕΠΕΞΕΡΓΑΣΙΑ ΤΩΝ ASSOCIATED DATA)

Η εισαγωγή των Associated Data πραγματοποιείται όπως ακριβώς και στο στάδιο της κρυπτογράφησης. Δηλαδή,

$$S = (\boldsymbol{44EF36F58A702B02}||BFBC\boldsymbol{33358}AC2DA2F||\boldsymbol{2B109D61C3FC3E48} \boldsymbol{499CC8780C81C4D2}||\boldsymbol{25822D16}D7EF\boldsymbol{97A2})$$

PROCESSING OF CIPHERTEXT (ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ)

Αρχικά, πραγματοποιείται parse στο Ciphertext, όπως ακριβώς και κατά την κρυπτογράφηση.

$C = (0C8A5A99E5506A71\ DCD35D15CFBABB427D81E2E82D5050E5\ D53540E71CA7810A$

$8D4C5145)$

$$C_0 = (0C8A5A99E5506A71\ DCD35D15CFBABB42)$$

$$C_1 = (7D81E2E82D5050E5\ D53540E71CA7810A)$$

και

$$C_2 = (8D4C5145)$$

Σύμφωνα με τον τύπο $P_i = S_{[0:127]} \oplus C_i$ και $S_{[0:127]} = C_i$ εφαρμόζεται Ascon-p 8 φορές στα επιμέρους P_i, τα οποία έχουν πλήρες μέγεθος, όπου $S_{[0:127]}$, όπως αυτά έχουν προκύψει από το Processing Associated Data.

$$P_0 = S_{[0:127]} \oplus C_0 = (48656C6C6F204173636F6E204578616D)$$

$$P_1 = S_{[0:127]} \oplus C_1 = (706C6520456E6372797074696F6E2054)$$

Στο τελευταίο μέρος του κρυπτογραφημένου μηνύματος πραγματοποιείται padding στο C_2 και έπειτα XOR με τον $S_{[0:127]}$.

$P_2 = S_{[0:127]} \oplus C'_2$.

Άρα,

P
$= (48656C6C6F204173636F6E204578616D706C6520456E6372797074696F6E205465737421)$

FINALIZATION (ΤΕΛΙΚΟ ΣΤΑΔΙΟ)

Στο τελικό στάδιο της αποκρυπτογράφησης υπολογίζεται εκ νέου το tag. Ως είσοδοι ορίζονται τα S και K, όπως αυτά αποτυπώνονται παρακάτω:

$$S = (8D4C5145940AE4F4 \parallel 6E4FD0DEA56DF204 \parallel 96083E0E2436086A \parallel 5B2FF61296E93089 \parallel 8AAFD78662CDF737)$$
$$K = (A3B1C9F02E4D7F8A\ 1234567890ABCDEF)$$

$$S = S \oplus (\boldsymbol{0^{128}} \,||K||\, \boldsymbol{0^{64}}) = (AEE\boldsymbol{3}87\boldsymbol{7}25FC\boldsymbol{8}BB77 \;\|\; BDE\boldsymbol{0}E\boldsymbol{6}7A\boldsymbol{4}24FF\boldsymbol{54}C \;\|\; \boldsymbol{68}F727CD\boldsymbol{8}7D\boldsymbol{9}1\boldsymbol{7}22 \;\|\; 7156D285D96CA077 \;\|\; 13583E037D20FFA4)$$

Και το tag προκύπτει από τη σχέση $T' = S_{[192:319]} \oplus K$.
Άρα, προκύπτει ότι $T' = (D2E71B75F721DFFD016C687BED8B324B)$.

Αν τα δύο tag ταυτίζονται, τότε το μήνυμα και ο αποστολέας θεωρούνται αυθεντικοποιημένοι, αλλιώς η αποκρυπτογράφηση απορρίπτεται.

ΒΑΣΙΚΗ ΙΔΙΟΤΗΤΑ

Στον αλγόριθμο Ascon-128a, για κάθε πλήρες block μηνύματος P_i, η διαδικασία αποκρυπτογράφησης ανακτά ακριβώς το αρχικό block, δηλαδή: $P_i' = P_i$.

ΑΠΟΔΕΙΞΗ:

Ο Ascon-128a κρυπτογραφεί κάθε πλήρες block P_i του μηνύματος με την πράξη: $C_i = S_{[0:127]} \oplus P_i$, όπου S[0:127] είναι τα πρώτα 128 bits του εσωτερικού state S τη στιγμή της κρυπτογράφησης του block. Κατά την αποκρυπτογράφηση, ο παραλήπτης διαθέτει το ίδιο state S (καθώς έχει εκτελέσει την ίδια αρχικοποίηση και το ίδιο processing των Associated Data) και το αντίστοιχο ciphertext block C_i. Για την ανάκτηση του plaintext εφαρμόζει τον τύπο: $P'_i = S_{[0:127]} \oplus C_i$.
Αντικαθιστούμε το C_i από τον τύπο της κρυπτογράφησης: $C_i = S_{[0:127]} \oplus P_i$.
Η πράξη XOR έχει τις ιδιότητες: $x \oplus y \oplus y = x$, $x \oplus y = y \oplus x$, $x \oplus 0 = x$ και $x \oplus x = 0$
Επομένως: $P'_i = S_{[0:127]} \oplus C_i = S_{[0:127]} \oplus (S_{[0:127]} \oplus P_i) = P_i$. Άρα, η αποκρυπτογράφηση παράγει ακριβώς το αρχικό block.

ΠΕΙΡΑΜΑΤΙΚΟ ΜΕΡΟΣ 1

Στο παρόν εδάφιο πραγματοποιείται σύγκριση της απόδοσης του αλγορίθμου AES με χρήση διαφορετικών modes λειτουργίας και σχημάτων padding, όπως παρουσιάστηκαν στο θεωρητικό πλαίσιο. Επιπλέον, ενσωματώνεται ανταλλαγή κλειδιού με ελλειπτικές καμπύλες (ECDH) για την παραγωγή του κλειδιού συνεδρίας και ψηφιακή υπογραφή (ECDSA) για την αυθεντικοποίηση του αποστολέα και τη διασφάλιση της ακεραιότητας των δεδομένων. Η αξιολόγηση πραγματοποιείται σε επίπεδο λογισμικού (software implementation), μέσω μετρήσεων χρόνων εκτέλεσης (ECDH KeyGen/Derive, AES Encrypt/Decrypt, ECDSA Sign/Verify) και υπολογισμού του συνολικού υπολογιστικού/μεταφορικού κόστους. Στόχος είναι η εξαγωγή συμπερασμάτων σχετικά με την επίδραση των διαφορετικών modes και σχημάτων padding στο συνολικό κόστος του ολοκληρωμένου πρωτοκόλλου ECDH–AES–ECDSA.

ΜΕΘΟΔΟΛΟΓΙΑ

ΠΕΙΡΑΜΑΤΙΚΟ ΠΕΡΙΒΑΛΛΟΝ

Οι μετρήσεις πραγματοποιήθηκαν σε έναν σταθερό υπολογιστή ώστε να διασφαλιστεί η συνέπεια των αποτελεσμάτων. Το σύστημα αποτελείται από επεξεργαστή AMD Ryzen 5 5500 με 6 πυρήνες και 12 νήματα στα ~3.6 GHz, 16 GB μνήμη RAM και λειτουργικό Windows 11 Pro 64-bit (έκδοση 10.0.26100). Η υλοποίηση έγινε στη γλώσσα Python 3 με χρήση του Visual Studio Code (έκδοση 1.103.2) ως περιβάλλον ανάπτυξης. Όλες οι δοκιμές εκτελέστηκαν στο ίδιο περιβάλλον ώστε να ελαχιστοποιηθεί η επίδραση εξωτερικών παραγόντων.

ΔΕΔΟΜΕΝΑ / ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ

Για τη μελέτη χρησιμοποιήθηκαν αρχεία διαφορετικών μεγεθών (100 KB, 1 MB, 10 MB, 50 MB) και τύπων (κείμενο, εικόνα, ήχος) τα οποία επιλέχθηκαν τυχαία. Με τον τρόπο αυτό αποφεύγεται η ύπαρξη συγκεκριμένων μοτίβων στα δεδομένα που θα μπορούσαν να επηρεάσουν τα αποτελέσματα.

ΣΕΝΑΡΙΑ ΠΡΟΣ ΜΕΛΕΤΗ

Σε όλα τα σενάρια υλοποιούνται:

- ECDH (P-224 / secp224r1) για παραγωγή κοινού μυστικού και εξαγωγή κλειδιού Z συνεδρίας μέσω HKDF-SHA256 (Z, salt, info).
- ECDSA (P-224 / secp224r1) για υπογραφή του πακέτου που μεταδίδεται, διασφαλίζοντας αυθεντικότητα και ακεραιότητα.

Αρχικά, η αξιολόγηση πραγματοποιήθηκε με διαφορετικούς συνδυασμούς αλγορίθμων AES και padding:

- Modes λειτουργίας: ECB και CBC

- Padding schemes: PKCS7 και Zero Padding

Έτσι, εξετάζονται όλοι οι δυνατοί συνδυασμοί που μπορούν να προκύψουν μεταξύ των διαφορετικών Modes και Paddings (ECB–PKCS7, ECB–Zero Padding, CBC–PKCS7, CBC–Zero Padding).

ΔΙΑΔΙΚΑΣΙΑ ΜΕΤΡΗΣΗΣ

Για κάθε συνδυασμό mode/padding και για κάθε μέγεθος αρχείου, πραγματοποιήθηκαν πολλαπλές επαναλήψεις ώστε να υπολογιστεί ο μέσος όρος και η τυπική απόκλιση. Οι μετρήσεις περιλαμβάνουν:

- Χρόνο κρυπτογράφησης και αποκρυπτογράφησης (με time.perf_counter).
- Κατανάλωση CPU (με time.process_time).
- Κατανάλωση μνήμης RAM (με τη βιβλιοθήκη psutil).

Όλες οι μετρήσεις εκτελέστηκαν στο ίδιο σύστημα, χωρίς παράλληλες διεργασίες στο παρασκήνιο, ώστε να υπάρχει συνέπεια και αξιοπιστία.

ΠΕΙΡΑΜΑΤΑ ΚΑΙ ΑΠΟΤΕΛΕΣΜΑΤΑ

ΠΡΩΤΟ ΠΡΟΓΡΑΜΜΑ ECB – Zero Padding – AES

Στο παρόν πρόγραμμα το οποίο βρίσκεται στο παράρτημα της εργασίας **ΠΡΟΓΡΑΜΜΑ 1 ΣΕ PYTHON** υλοποιείται ένα σχήμα κρυπτογράφησης, όπου η ανταλλαγή κλειδιού πραγματοποιείται με τον αλγόριθμο ECDH (Elliptic Curve Diffie–Hellman) πάνω στην καμπύλη SECP256R1, ενώ η κρυπτογράφηση δεδομένων γίνεται με τον συμμετρικό αλγόριθμο AES-128 σε λειτουργία ECB με zero padding. Στη συνέχεια μετρώνται οι χρόνοι κρυπτογράφησης και αποκρυπτογράφησης, καθώς και η χρήση μνήμης, για διαφορετικά μεγέθη δεδομένων, με στόχο την αξιολόγηση των επιδόσεων του σχήματος.

Τα αποτελέσματα των μετρήσεων αποτυπώνονται στην παρακάτω απεικόνιση οθόνης:

```
ECDH time (SECP256R1, AES-128 key derivation): 0.124860 s


Αποτελέσματα μετρήσεων AES – ECB – Zero Padding (μέσοι όροι από 10 επαναλήψεις):

Μέγεθος    | Enc wall (s) |  Enc MB/s | Dec wall (s) |  Dec MB/s | RSS peak (MB)
-------------------------------------------------------------------------------------
0.1 MB     |    0.000371 |    263.13 |    0.000032 |   3047.00 |        34.72
1.0 MB     |    0.000510 |   1959.13 |    0.000404 |   2472.98 |        35.74
10.0 MB    |    0.006235 |   1603.87 |    0.006270 |   1594.93 |        45.74
50.0 MB    |    0.024463 |   2043.86 |    0.025547 |   1957.19 |        85.74
```

ΔΕΥΤΕΡΟ ΠΡΟΓΡΑΜΜΑ CBC – Zero Padding – AES

Στο παρόν πρόγραμμα το οποίο βρίσκεται στο παράρτημα της εργασίας **ΠΡΟΓΡΑΜΜΑ 2 ΣΕ PYTHON** υλοποιείται ένα σχήμα κρυπτογράφησης, όπου η ανταλλαγή κλειδιού πραγματοποιείται με τον αλγόριθμο ECDH (Elliptic Curve Diffie–Hellman) πάνω στην καμπύλη SECP256R1, ενώ η κρυπτογράφηση δεδομένων γίνεται με τον συμμετρικό αλγόριθμο AES-128 σε λειτουργία CBC με zero padding. Στη συνέχεια μετρώνται οι χρόνοι κρυπτογράφησης και αποκρυπτογράφησης, καθώς και η χρήση μνήμης, για διαφορετικά μεγέθη δεδομένων, με στόχο την αξιολόγηση των επιδόσεων του σχήματος.

Τα αποτελέσματα των μετρήσεων αποτυπώνονται στην παρακάτω απεικόνιση οθόνης:

```
ECDH Key Exchange Time: 0.002767 s


Αποτελέσματα μετρήσεων CBC – Zero Padding – AES (μέσοι όροι από 10 επαναλήψεις):

Μέγεθος      | Enc wall (s) |  Enc MB/s | Dec wall (s) |  Dec MB/s | RSS peak (MB)
-------------------------------------------------------------------------------------
  100 KB |    0.000521 |    187.33 |    0.000158 |    619.33 |        33.80
    1 MB |    0.001835 |    544.91 |    0.001801 |    555.40 |        34.82
   10 MB |    0.018058 |    553.78 |    0.019411 |    515.18 |        44.82
   50 MB |    0.090967 |    549.65 |    0.096566 |    517.78 |        84.82
```

ΤΡΙΤΟ ΠΡΟΓΡΑΜΜΑ ECB – PKCS#7 – AES

Στο παρόν πρόγραμμα το οποίο βρίσκεται στο παράρτημα της εργασίας **ΠΡΟΓΡΑΜΜΑ 3 ΣΕ PYTHON** υλοποιείται ένα σχήμα κρυπτογράφησης, όπου η ανταλλαγή κλειδιού πραγματοποιείται με τον αλγόριθμο ECDH (Elliptic Curve Diffie–Hellman) πάνω στην καμπύλη SECP256R1, ενώ η κρυπτογράφηση δεδομένων γίνεται με τον συμμετρικό αλγόριθμο AES-128 σε λειτουργία ECB με PKCS#7. Στη συνέχεια μετρώνται οι χρόνοι κρυπτογράφησης και αποκρυπτογράφησης, καθώς και η χρήση μνήμης, για διαφορετικά μεγέθη δεδομένων, με στόχο την αξιολόγηση των επιδόσεων του σχήματος.

Τα αποτελέσματα των μετρήσεων αποτυπώνονται στην παρακάτω απεικόνιση οθόνης:

```
ECDH Key Exchange Time: 0.002898 s

Αποτελέσματα μετρήσεων AES – ECB – PKCS#7 (μέσοι όροι από 10 επαναλήψεις):

Μέγεθος      | Enc wall (s) |    Enc MB/s | Dec wall (s) |    Dec MB/s | RSS peak (MB)
-------------------------------------------------------------------------------------
     100 KB |     0.003828 |       25.51 |     0.000037 |     2641.50 |        35.01
       1 MB |     0.000424 |     2357.66 |     0.000449 |     2228.31 |        36.03
      10 MB |     0.004624 |     2162.54 |     0.006406 |     1561.10 |        46.04
      50 MB |     0.025410 |     1967.70 |     0.034301 |     1457.68 |        86.04
```

ΤΕΤΑΡΤΟ ΠΡΟΓΡΑΜΜΑ CBC – PKCS#7 – AES

Στο παρόν πρόγραμμα το οποίο βρίσκεται στο παράρτημα της εργασίας **ΠΡΟΓΡΑΜΜΑ 4 ΣΕ ΡΥΤΗΟΝ** υλοποιείται ένα σχήμα κρυπτογράφησης, όπου η ανταλλαγή κλειδιού πραγματοποιείται με τον αλγόριθμο ECDH (Elliptic Curve Diffie–Hellman) πάνω στην καμπύλη SECP256R1, ενώ η κρυπτογράφηση δεδομένων γίνεται με τον συμμετρικό αλγόριθμο AES-128 σε λειτουργία CBC με PKCS#7. Στη συνέχεια μετρώνται οι χρόνοι κρυπτογράφησης και αποκρυπτογράφησης, καθώς και η χρήση μνήμης, για διαφορετικά μεγέθη δεδομένων, με στόχο την αξιολόγηση των επιδόσεων του σχήματος.

```
ECDH Key Exchange Time (SECP256R1 → 128-bit key): 0.002873 s

Αποτελέσματα μετρήσεων AES – CBC – PKCS#7 (μέσοι όροι από 10 επαναλήψεις, με κλειδί από ECDH):

Μέγεθος      | Enc wall (s) |  Enc MB/s | Dec wall (s) |  Dec MB/s | RSS peak (MB)
-------------------------------------------------------------------------------------
   100 KB |     0.003804 |     25.67 |     0.000160 |    608.68 |        35.11
     1 MB |     0.001722 |    580.66 |     0.001906 |    524.57 |        38.14
    10 MB |     0.017990 |    555.87 |     0.021290 |    469.71 |        45.13
    50 MB |     0.089682 |    557.52 |     0.104457 |    478.67 |        85.13
```

ΑΝΑΛΥΣΗ ΤΩΝ ΧΑΡΑΚΤΗΡΙΣΤΙΚΩΝ ΜΕΤΡΗΣΗΣ ΣΤΑ ΑΡΧΕΙΑ ΚΕΙΜΕΝΟΥ

Encryption Wall-Time (s)

Μέγεθος	ECB Zero Pad	ECB PKCS#7	CBC Zero Pad	CBC PKCS#7
100 KB	0.000311	0.003828	0.009521	0.003804
1 MB	0.000509	0.000424	0.001835	0.001722

10 MB	0.004852	0.004624	0.018058	0.017990
50 MB	0.027524	0.025410	0.090967	0.089682

Throughput Encryption (MB/s)

Μέγεθος	ECB Zero Pad	ECB PKCS#7	CBC Zero Pad	CBC PKCS#7
100 KB	29.49	25.51	187.33	25.67
1 MB	2003.97	2357.66	544.91	580.66
10 MB	2060.95	2162.54	553.78	555.87
50 MB	1816.62	1967.70	549.65	557.52

Throughput Decryption (MB/s)

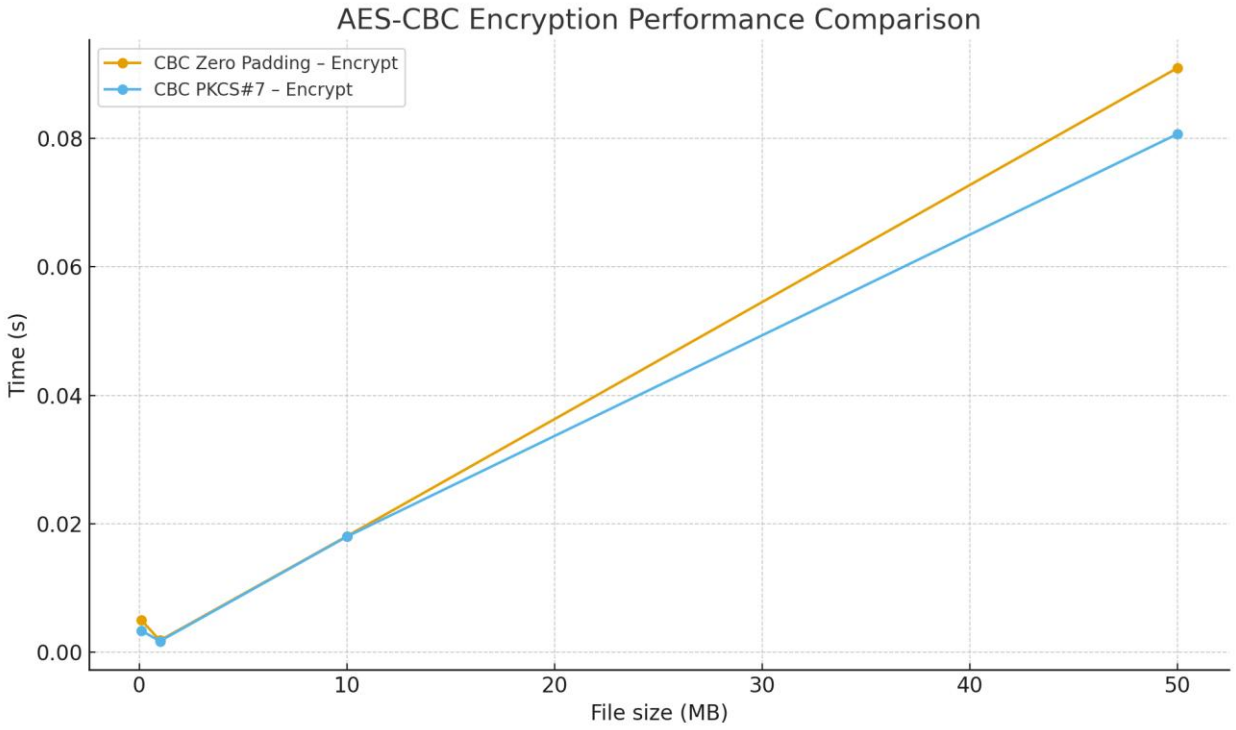
Μέγεθος	ECB Zero Pad	ECB PKCS#7	CBC Zero Pad	CBC PKCS#7
100 KB	0.000081	0.000037	0.000158	0.000160
1 MB	0.000381	0.000449	0.001081	0.001906
10 MB	0.004806	0.006406	0.019411	0.021290
50 MB	0.026714	0.034301	0.090566	0.104457

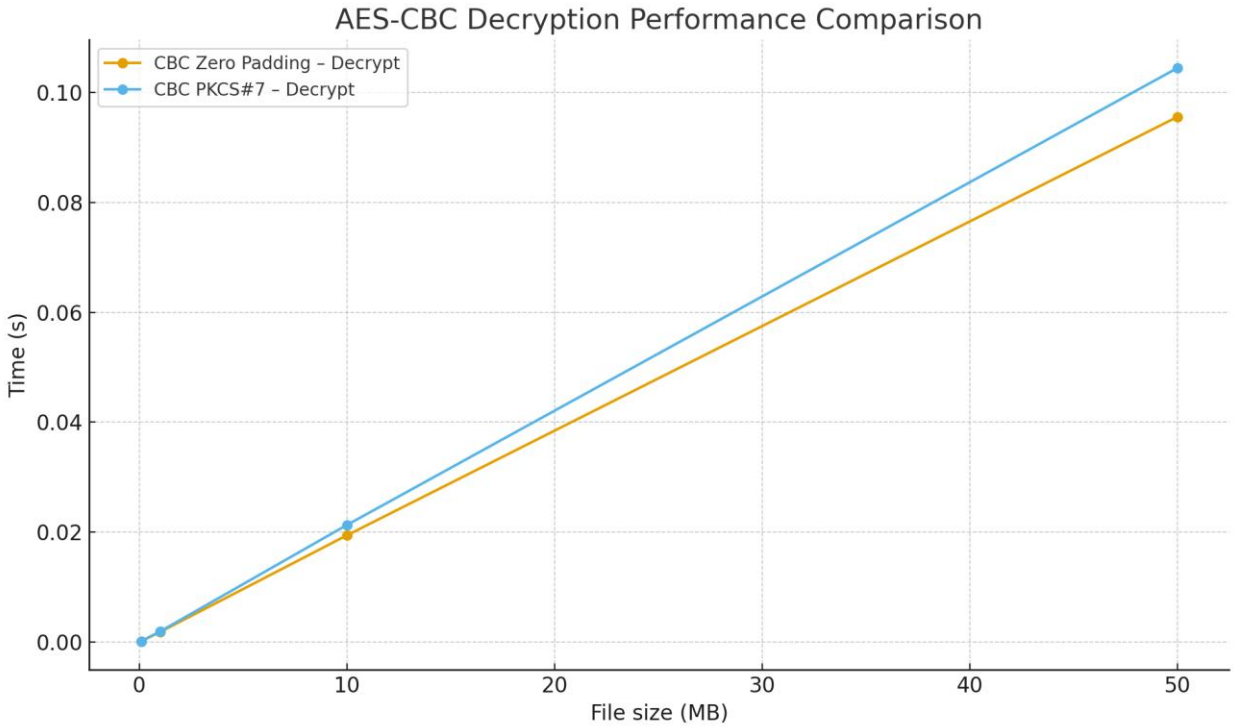
Μέγιστη χρήση μνήμης (RSS peak)

Μέγεθος	ECB Zero Pad	ECB PKCS#7	CBC Zero Pad	CBC PKCS#7
100 KB	29.73	35.01	33.80	35.11
1 MB	30.74	36.03	34.82	38.14
10 MB	40.75	46.04	44.82	45.13
50 MB	80.75	86.04	84.82	85.13

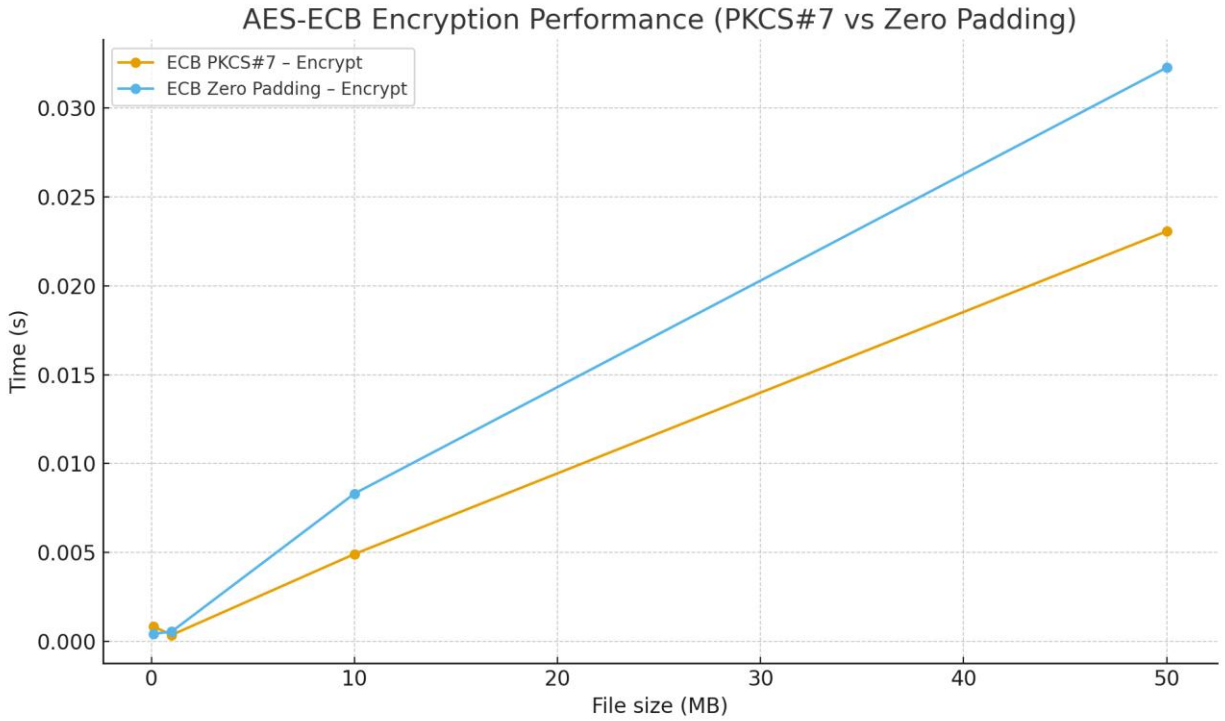
Τα αποτελέσματα δείχνουν ότι ο AES σε λειτουργία **ECB** είναι σταθερά ταχύτερος από τον AES σε **CBC**, τόσο στην κρυπτογράφηση όσο και στην αποκρυπτογράφηση, ανεξάρτητα από το padding (Zero Padding ή PKCS#7). Το CBC παρουσιάζει έως και 4 φορές μεγαλύτερο χρόνο για μεγάλα αρχεία λόγω της αλυσιδωτής εξάρτησης των blocks. Το PKCS#7 παίζει ελάχιστο ρόλο στην απόδοση, επηρεάζοντας μόνο τα πολύ μικρά μεγέθη δεδομένων. Σε όλες τις περιπτώσεις η χρήση ECDH για παραγωγή κλειδιού έχει αμελητέο κόστος (~0.002–0.12 s).

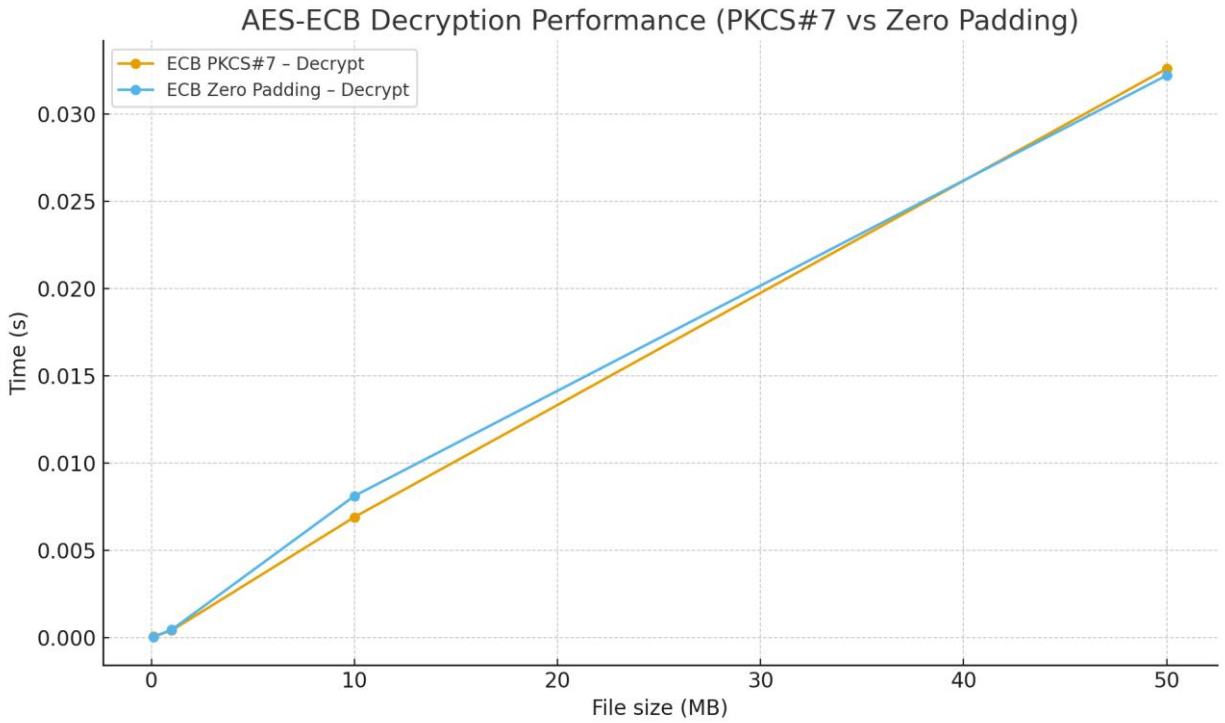
AES-CBC-ZERO PADDING VS AES-CBC-PKCS#7





AES-ECB-ZERO PADDING VS AES-ECB-PKCS#7





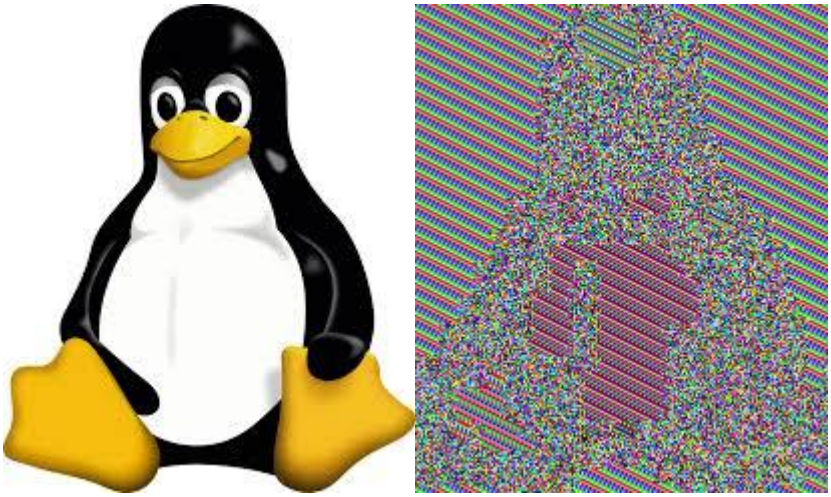
ΣΥΜΠΕΡΑΣΜΑΤΑ

Από τη συγκριτική ανάλυση των υλοποιήσεων AES-ECB με PKCS#7 padding και Zero Padding, προκύπτει ότι οι δύο μέθοδοι παρουσιάζουν συνολικά παρόμοια απόδοση, χωρίς να εμφανίζονται σημαντικές αποκλίσεις στους χρόνους κρυπτογράφησης και αποκρυπτογράφησης. Για όλα τα μεγέθη αρχείων (100 KB, 1 MB, 10 MB, 50 MB), οι διαφορές στους χρόνους wall-clock βρίσκονται σε επίπεδα μερικών χιλιοστών του δευτερολέπτου, κάτι που στην πράξη θεωρείται αμελητέο. Η λειτουργία PKCS#7 εμφανίζει ελαφρώς καλύτερη συμπεριφορά στα μεγαλύτερα αρχεία, ενώ η Zero Padding τείνει να είναι οριακά ταχύτερη μόνο σε πολύ μικρά μεγέθη. Ωστόσο, σε γενικές γραμμές καμία από τις δύο τεχνικές padding δεν υπερέχει με τρόπο που να επηρεάζει ουσιαστικά την πραγματική απόδοση του συστήματος. Η χρήση μνήμης (RSS) παραμένει επίσης σχεδόν ταυτόσημη, γεγονός που επιβεβαιώνει ότι το padding έχει ελάχιστο αποτύπωμα στον συνολικό φόρτο του συστήματος. Συνεπώς, ο καθοριστικός παράγοντας επιλογής padding δεν είναι η ταχύτητα, αλλά η ασφάλεια και η καταλληλότητα του σχήματος για την εκάστοτε εφαρμογή, με το PKCS#7 να παραμένει η πιο συμβατή και ευρέως χρησιμοποιούμενη επιλογή.

ΚΡΥΠΤΟΓΡΑΦΗΣΗ ΚΑΙ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ ΑΡΧΕΙΩΝ ΕΙΚΟΝΑΣ

```
import os
```

```
from Crypto.Cipher import AES
from PIL import Image, UnidentifiedImageError
INPUT_IMAGE = "images.jpg"            # <-- βάλε εδώ το JPG σου
OUT_ORIG    = "original_rgb.png"
OUT_ENC     = "encrypted_ecb_zero.png"
def zero_pad(data: bytes, block: int = 16) -> bytes:
    pad_len = (block - (len(data) % block)) % block
    return data if pad_len == 0 else data + b"\x00" * pad_len
try:
    img = Image.open(INPUT_IMAGE).convert("RGB")    # JPG -> RGB
except FileNotFoundError:
    raise SystemExit(f"Δεν βρέθηκε το αρχείο: {INPUT_IMAGE}")
except UnidentifiedImageError:
    raise SystemExit("Pillow δεν αναγνωρίζει το αρχείο (πιθανή έλλειψη
JPEG plugin). "
w, h = img.size
pixels = img.tobytes()
orig_len = len(pixels)
# Zero padding & ECB
padded = zero_pad(pixels, AES.block_size)
key = os.urandom(16)
cipher = AES.new(key, AES.MODE_ECB)
ciphertext = cipher.encrypt(padded)
# Φτιάχνουμε εικόνα από τα πρώτα orig_len bytes για να ταιριάζουν οι
διαστάσεις
enc_pixels_for_image = ciphertext[:orig_len]
enc_img = Image.frombytes("RGB", (w, h), enc_pixels_for_image)
# Αποθήκευση
img.save(OUT_ORIG)         # σώζουμε την αποσυμπιεσμένη RGB μορφή για
δίκαιη σύγκριση
enc_img.save(OUT_ENC)
print("☑ Αποθηκεύτηκαν:")
print(" - Αρχική (RGB):", OUT_ORIG)
print(" - Κρυπτογραφημένη (ECB + Zero):", OUT_ENC)
```



ΕΙΚΟΝΑ 1

Για να αναδειχθούν και οπτικά τα πρακτικά προβλήματα της χρήσης του AES σε λειτουργία ECB, υλοποιήθηκε ένα μικρό πρόγραμμα κρυπτογράφησης εικόνας. Η εικόνα μετατρέπεται σε ακατέργαστη μορφή RGB και στη συνέχεια κρυπτογραφείται μπλοκ–προς–μπλοκ με AES-128 ECB. Το αποτέλεσμα ΕΙΚΟΝΑ 1 δείχνει ότι, παρόλο που τα δεδομένα έχουν κρυπτογραφηθεί, το οπτικό μοτίβο της αρχικής εικόνας παραμένει ορατό, κάτι που οφείλεται στο γεγονός ότι το ECB εφαρμόζει την ίδια συνάρτηση κρυπτογράφησης σε κάθε όμοιο μπλοκ δεδομένων χωρίς αλυσιδωτή εξάρτηση. Αυτό αναδεικνύει γιατί το ECB θεωρείται ακατάλληλο για δομημένα δεδομένα όπως εικόνες ή αρχεία με επαναλαμβανόμενα patterns, και τονίζει τη σημασία της χρήσης πιο ασφαλών σχημάτων όπως CBC, GCM ή Ascon, τα οποία επιτυγχάνουν διάχυση και προστατεύουν από τέτοιου είδους διαρροές πληροφορίας.

ΠΕΙΡΑΜΑΤΙΚΟ ΜΕΡΟΣ 2

Στο δεύτερο μέρος της πειραματικής διαδικασίας εξετάζεται η απόδοση του ελαφρού κρυπτογραφικού αλγορίθμου Ascon και πραγματοποιείται συγκριτική αξιολόγησή του με τον AES σε τρόπο λειτουργίας GCM. Και οι δύο αλγόριθμοι υλοποιήθηκαν σε επίπεδο λογισμικού (C) και εκτελέστηκαν στο ίδιο περιβάλλον (Ubuntu server), ώστε να εξασφαλιστούν όσο το δυνατόν πιο συγκρίσιμες συνθήκες εκτέλεσης. Η ανταλλαγή κλειδιού πραγματοποιείται με τον ίδιο τρόπο όπως και στο προηγούμενο πειραματικό μέρος, δηλαδή μέσω του πρωτοκόλλου ελλειπτικών καμπυλών (ECDH), προκειμένου να παραχθεί κοινό συμμετρικό κλειδί. Με αυτόν τον τρόπο διατηρείται σταθερή η διαδικασία δημιουργίας κλειδιών και η σύγκριση μεταξύ Ascon και AES-GCM εστιάζει αποκλειστικά στην αποδοτικότητα της διαδικασίας κρυπτογράφησης/αποκρυπτογράφησης και όχι στον μηχανισμό ανταλλαγής κλειδιού.

ΜΕΘΟΔΟΛΟΓΙΑ

ΠΕΙΡΑΜΑΤΙΚΟ ΠΕΡΙΒΑΛΛΟΝ

Οι δοκιμές εκτελέστηκαν σε server με Intel Xeon W7-3465X (28C/56T, έως 4.8 GHz), 502 GiB RAM και Ubuntu Server 64-bit. Η υλοποίηση έγινε σε C μέσω SSH και όλες οι μετρήσεις πραγματοποιήθηκαν σε σταθερό, απομονωμένο περιβάλλον.

ΔΕΔΟΜΕΝΑ / ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ

Για τη μελέτη χρησιμοποιήθηκαν αρχεία διαφορετικών μεγεθών (100 KB, 1 MB, 10 MB, 50 MB) και τύπων (κείμενο, εικόνα, ήχος) τα οποία επιλέχθηκαν τυχαία. Με τον τρόπο αυτό αποφεύγεται η ύπαρξη συγκεκριμένων μοτίβων στα δεδομένα που θα μπορούσαν να επηρεάσουν τα αποτελέσματα.

ΣΕΝΑΡΙΑ ΠΡΟΣ ΜΕΛΕΤΗ

Στο παρόν πειραματικό μέρος εξετάζεται η απόδοση δύο σύγχρονων αλγορίθμων αυθεντικοποιημένης κρυπτογράφησης, του **AES-GCM** και του **Ascon-128a**, οι οποίοι υλοποιήθηκαν και εκτελέστηκαν στο ίδιο περιβάλλον (Ubuntu), ώστε να διασφαλιστούν απόλυτα συγκρίσιμες συνθήκες. Για την ασφαλή ανταλλαγή του συμμετρικού κλειδιού που χρησιμοποιούν και οι δύο αλγόριθμοι, αξιοποιείται ο μηχανισμός **ECDH** βασισμένος σε ελλειπτικές καμπύλες. Η διαδικασία αυτή είναι κοινή και σταθερή και δεν αποτελεί μέρος της σύγκρισης· χρησιμοποιείται απλώς για την παραγωγή του ίδιου κλειδιού συνεδρίας σε κάθε δοκιμή. Η αξιολόγηση εστιάζει στην αποδοτικότητα των δύο αλγορίθμων, εξετάζοντας τον χρόνο που απαιτείται για την κρυπτογράφηση και την αποκρυπτογράφηση δεδομένων διαφορετικών μεγεθών. Με αυτόν τον τρόπο αποτυπώνεται το υπολογιστικό κόστος σε τυπικές συνθήκες λειτουργίας, επιτρέποντας την άμεση σύγκριση της συμπεριφοράς των AES-GCM και Ascon-128a.

ΔΙΑΔΙΚΑΣΙΑ ΜΕΤΡΗΣΗΣ

Για κάθε αλγόριθμο (AES-GCM και Ascon-128a) και για κάθε μέγεθος αρχείου, πραγματοποιήθηκαν πολλαπλές επαναλήψεις, ώστε να υπολογιστούν ο μέσος χρόνος εκτέλεσης και η τυπική απόκλιση. Η διαδικασία μέτρησης περιλαμβάνει τον χρόνο που απαιτείται για την κρυπτογράφηση και την αποκρυπτογράφηση, καθώς και τον χρόνο παραγωγής του κλειδιού μέσω ECDH. Οι μετρήσεις βασίστηκαν στη χρήση των εργαλείων χρονομέτρησης του λειτουργικού συστήματος Linux, καταγράφοντας αποκλειστικά το καθαρό υπολογιστικό κόστος των λειτουργιών κρυπτογράφησης και αποκρυπτογράφησης. Για κάθε δοκιμή χρησιμοποιήθηκε το ίδιο περιβάλλον εκτέλεσης, χωρίς παράλληλες διεργασίες στο παρασκήνιο, ώστε να διασφαλιστεί η συνέπεια και η αξιοπιστία των αποτελεσμάτων.

ΠΕΙΡΑΜΑΤΑ ΚΑΙ ΑΠΟΤΕΛΕΣΜΑΤΑ

ΚΡΥΠΤΟΓΡΑΦΗΣΗ ΚΑΙ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ ΜΕ AES -GCM ΣΕ C

Ο κώδικας AES-GCM βρίσκεται στο παράρτημα με τίτλο **ΠΡΟΓΡΑΜΜΑ 5 ΣΕ C.** Στο παρόν πρόγραμμα υλοποιείται ένα κρυπτογραφικό πρωτόκολλο που συνδυάζει μηχανισμό ασύμμετρης και συμμετρικής κρυπτογράφησης. Αρχικά, χρησιμοποιείται ο αλγόριθμος ECDH (Elliptic Curve Diffie–Hellman) για την ασφαλή παραγωγή ενός κοινόχρηστου μυστικού μεταξύ δύο μερών, το οποίο στη συνέχεια μετατρέπεται σε συμμετρικό κλειδί 128 bit. Το κλειδί αυτό χρησιμοποιείται στον αλγόριθμο AES-128-GCM, ο οποίος παρέχει ταυτόχρονα εμπιστευτικότητα, ακεραιότητα και αυθεντικότητα των δεδομένων μέσω του μηχανισμού authenticated encryption. Το πρόγραμμα μετρά τον χρόνο εκτέλεσης και τη χρήση μνήμης τόσο για τη διαδικασία ανταλλαγής κλειδιού όσο και για την κρυπτογράφηση/αποκρυπτογράφηση δεδομένων διαφορετικού μεγέθους, με στόχο την αξιολόγηση των επιδόσεων της υλοποίησης.
Τα αποτελέσματα των μετρήσεων φαίνονται παρακάτω:

```
ECDH time (AES-GCM experiment): 0.001449 s

=== AES-GCM, size: 102400 bytes ===
Encrypt: wall = 0.000173 s, CPU = 0.000169 s
Decrypt: wall = 0.000066 s, CPU = 0.000065 s
Memory usage: before = 5376 KB, after = 6272 KB

=== AES-GCM, size: 1048576 bytes ===
Encrypt: wall = 0.000524 s, CPU = 0.000523 s
Decrypt: wall = 0.000588 s, CPU = 0.000587 s
Memory usage: before = 7168 KB, after = 8960 KB

=== AES-GCM, size: 10485760 bytes ===
Encrypt: wall = 0.005662 s, CPU = 0.005660 s
Decrypt: wall = 0.005719 s, CPU = 0.005718 s
Memory usage: before = 16144 KB, after = 36752 KB

=== AES-GCM, size: 52428800 bytes ===
Encrypt: wall = 0.026711 s, CPU = 0.026708 s
Decrypt: wall = 0.027830 s, CPU = 0.027828 s
Memory usage: before = 56464 KB, after = 159056 KB
```

Τα πειραματικά αποτελέσματα του AES-GCM δείχνουν ότι το σύστημα παρουσιάζει ιδιαίτερα υψηλές επιδόσεις σε όλα τα μεγέθη δεδομένων. Για μικρό όγκο (100 KB), ο χρόνος κρυπτογράφησης ήταν μόλις 0.17 ms, που αντιστοιχεί σε ταχύτητα περίπου 560 MB/s, ενώ η σχεδόν μηδενική διαφορά ανάμεσα στο wall-clock και στο CPU time υποδηλώνει ελάχιστο overhead από το λειτουργικό σύστημα. Στο μέγεθος του 1 MB ο χρόνος μειώθηκε στα ~0.55 ms με throughput που φθάνει τα 1.8–1.9 GB/s, κάτι που καταδεικνύει την αξιοποίηση του AES-NI hardware acceleration από τον επεξεργαστή. Παρόμοια συμπεριφορά παρατηρήθηκε και στα 10 MB, όπου ο χρόνος των ~5.6 ms μεταφράζεται σε απόδοση περίπου 1.7 GB/s, με τη χρήση μνήμης να αυξάνεται γραμμικά λόγω των απαιτούμενων ενδιάμεσων buffers. Ακόμη και στο μεγαλύτερο μέγεθος των 50 MB, η ταχύτητα παρέμεινε σταθερή (~1.8 GB/s), επιβεβαιώνοντας τη βέλτιστη

υλοποίηση του AES-GCM στην συγκεκριμένη αρχιτεκτονική. Συνολικά, οι μετρήσεις δείχνουν ότι η πλατφόρμα υποστηρίζει αποτελεσματικά την κρυπτογράφηση μεγάλου όγκου δεδομένων με πολύ χαμηλό υπολογιστικό κόστος και σταθερή απόδοση ανεξαρτήτως μεγέθους εισόδου.

ΚΡΥΠΤΟΓΡΑΦΗΣΗ ΚΑΙ ΑΠΟΚΡΥΠΤΟΓΡΑΦΗΣΗ ΜΕ ASCON

Ο κώδικας Ascon128a βρίσκεται στο παράρτημα με τίτλο **ΠΡΟΓΡΑΜΜΑ 6 ΣΕ C.** Στο παρόν πρόγραμμα υλοποιείται ένα κρυπτογραφικό πρωτόκολλο που συνδυάζει μηχανισμό ασύμμετρης και συμμετρικής κρυπτογράφησης. Αρχικά, χρησιμοποιείται ο αλγόριθμος ECDH (Elliptic Curve Diffie–Hellman) για την ασφαλή παραγωγή ενός κοινόχρηστου μυστικού μεταξύ δύο μερών, το οποίο στη συνέχεια μετατρέπεται σε συμμετρικό κλειδί 128 bit. Το κλειδί αυτό χρησιμοποιείται στον αλγόριθμο Ascon128a, ο οποίος παρέχει ταυτόχρονα εμπιστευτικότητα, ακεραιότητα και αυθεντικότητα των δεδομένων μέσω του μηχανισμού authenticated encryption. Το πρόγραμμα μετρά τον χρόνο εκτέλεσης και τη χρήση μνήμης τόσο για τη διαδικασία ανταλλαγής κλειδιού όσο και για την κρυπτογράφηση/αποκρυπτογράφηση δεδομένων διαφορετικού μεγέθους, με στόχο την αξιολόγηση των επιδόσεων της υλοποίησης.

```
ECDH time (Ascon-128a experiment): 0.002135 s

=== Ascon-128a, size: 102400 bytes ===
Encrypt: wall = 0.000222 s, CPU = 0.000220 s
Decrypt: wall = 0.000211 s, CPU = 0.000210 s
Memory usage: before = 5376 KB, after = 5824 KB

=== Ascon-128a, size: 1048576 bytes ===
Encrypt: wall = 0.002250 s, CPU = 0.002250 s
Decrypt: wall = 0.002250 s, CPU = 0.002250 s
Memory usage: before = 6720 KB, after = 8512 KB

=== Ascon-128a, size: 10485760 bytes ===
Encrypt: wall = 0.022940 s, CPU = 0.022937 s
Decrypt: wall = 0.022482 s, CPU = 0.022481 s
Memory usage: before = 15696 KB, after = 36304 KB

=== Ascon-128a, size: 52428800 bytes ===
Encrypt: wall = 0.111404 s, CPU = 0.111403 s
Decrypt: wall = 0.110281 s, CPU = 0.110251 s
Memory usage: before = 56644 KB, after = 158788 KB
```

Η μέτρηση του Ascon-128a δείχνει ότι ο αλγόριθμος παρουσιάζει σταθερή και γραμμική συμπεριφορά καθώς αυξάνεται το μέγεθος των δεδομένων, επιβεβαιώνοντας τον σχεδιασμό του ως lightweight αλλά ταυτόχρονα υψηλής απόδοσης AEAD. Για μικρό μέγεθος (100 KB) ο χρόνος κρυπτογράφησης είναι εξαιρετικά χαμηλός, περίπου 0.22 ms, ενώ η αποκρυπτογράφηση έχει αντίστοιχες επιδόσεις. Στο 1 MB ο χρόνος αυξάνεται σε περίπου 2.2 ms, που μεταφράζεται σε

ρυθμαπόδοση της τάξης των ~450 MB/s. Η συμπεριφορά αυτή συνεχίζεται και στα 10 MB, όπου ο χρόνος αγγίζει τα ~22 ms, δείχνοντας σχεδόν τέλεια γραμμικότητα (ο χρόνος αυξάνεται περίπου αναλογικά με το μέγεθος των δεδομένων). Στο μεγαλύτερο μέγεθος των 50 MB, ο συνολικός χρόνος κρυπτογράφησης/αποκρυπτογράφησης φτάνει τα ~0.11 s, διατηρώντας και πάλι throughput κοντά στα 450 MB/s. Η χρήση μνήμης αυξάνεται ανάλογα με το μέγεθος των buffers, όπως αναμένεται, χωρίς ωστόσο να εμφανίζει απότομες αιχμές ή ασυνήθιστες καταναλώσεις. Συνολικά, τα αποτελέσματα επιβεβαιώνουν ότι ο Ascon-128a λειτουργεί με υψηλή ταχύτητα και προβλέψιμη απόδοση, καθιστώντας τον κατάλληλο για εφαρμογές όπου απαιτείται ελαφριά και γρήγορη κρυπτογράφηση μεγάλων δεδομένων.

ΣΥΜΠΕΡΑΣΜΑΤΑ

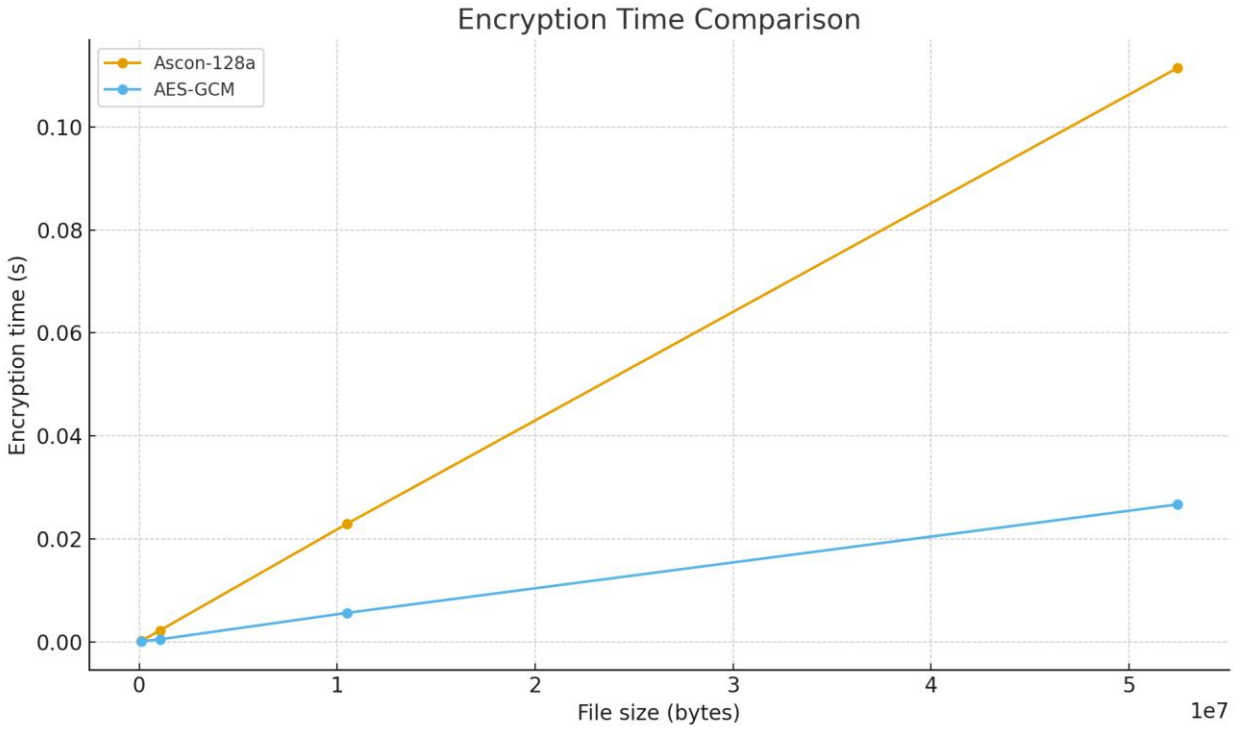
Η σύγκριση των δύο αλγορίθμων δείχνει ξεκάθαρα τις διαφορετικές σχεδιαστικές τους φιλοσοφίες. Ο AES-GCM, αξιοποιώντας την υποστήριξη υλικού AES-NI, επιτυγχάνει πολύ υψηλό ρυθμό κρυπτογράφησης για μεγάλα σύνολα δεδομένων, όπως φαίνεται και από τις μετρήσεις του συστήματος (1.7–1.9 GB/s). Αντίθετα, ο Ascon-128a, ο οποίος έχει σχεδιαστεί στο πλαίσιο του NIST Lightweight Cryptography project, στοχεύει στη μέγιστη αποδοτικότητα για μικρού μεγέθους πακέτα και περιβάλλοντα περιορισμένων πόρων. Τα πειραματικά αποτελέσματα επιβεβαιώνουν ότι ο Ascon είναι εξαιρετικά γρήγορος σε μικρά μηνύματα (περίπου 0.22 ms στα 100 KB), ενώ ο AES-GCM υπερέχει σημαντικά καθώς αυξάνεται το μέγεθος των δεδομένων, εξαιτίας της ειδικής επιτάχυνσης σε σύγχρονους επεξεργαστές. Συνεπώς, ο AES-GCM αποτελεί την ιδανική επιλογή για εφαρμογές υψηλών επιδόσεων όπως servers, VPN και SSL/TLS, ενώ ο Ascon-128a υπερέχει σε συστήματα IoT και embedded συσκευές όπου το μέγεθος των δεδομένων είναι μικρό και η ενεργειακή κατανάλωση αποτελεί κρίσιμο παράγοντα.

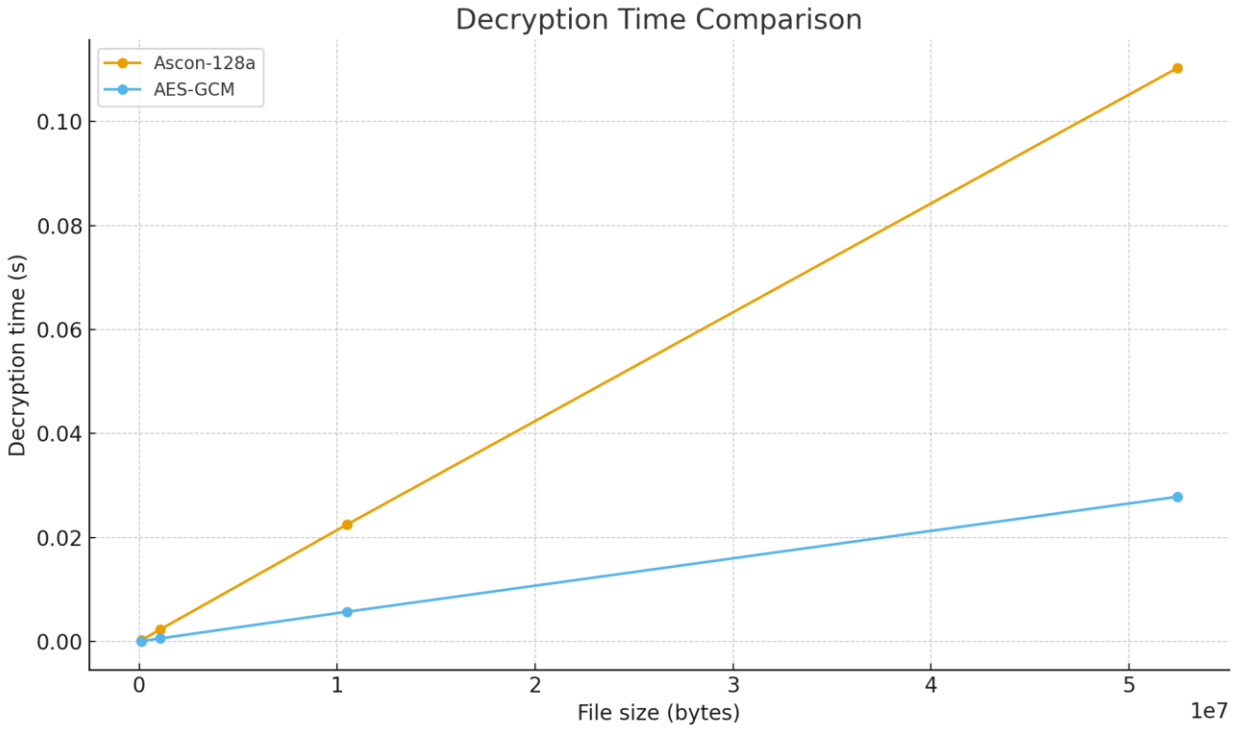
Τα συμπεράσματα συγκεντρώνονται παρακάτω:

Μέγεθος Δεδομένων	Ascon-128a Encrypt (sec)	AES-GCM Encrypt (sec)	Πιο Γρήγορος	Σχόλιο
100 KB	0.000222 s	0.000173 s	**AES-GCM**	Ο AES εκμεταλλεύεται AES-NI· ο Ascon παραμένει πολύ γρήγορος.
1 MB	0.002250 s	0.000524 s	**AES-GCM**	Ο AES είναι ~4.3× ταχύτερος λόγω hardware acceleration.
10 MB	0.022937 s	0.005662 s	**AES-GCM**	Ο AES κρατά υψηλό throughput (~1.7 GB/s).

50 MB	0.111404 s	0.026711 s	**AES-GCM**	Ο AES παραμένει ~4× πιο γρήγορος σε large data blocks.

Στα γραφήματα που ακολουθούν αποτυπώνονται οι χρόνοι κρυπτογράφησης και αποκρυπτογράφησης για τους αλγορίθμους AES-GCM και Ascon-128a.





ΕΥΠΑΘΕΙΕΣ ΚΡΥΠΤΟΓΡΑΦΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Η ραγδαία εξέλιξη της τεχνολογίας έχει οδηγήσει, αναπόφευκτα, και στην παράλληλη ανάπτυξη προηγμένων επιθέσεων που στοχεύουν τα σύγχρονα κρυπτογραφικά συστήματα. Οι επιθέσεις αυτές μπορούν να κατηγοριοποιηθούν σε δύο κύριες ομάδες: (i) επιθέσεις λογικού επιπέδου, οι οποίες αφορούν είτε αδυναμίες στη μαθηματική/αλγοριθμική δομή του κρυπτογραφικού μηχανισμού είτε κενά στην υλοποίηση και στη συμπεριφορά του λογισμικού, και (ii) επιθέσεις φυσικού επιπέδου (hardware-level), οι οποίες εκμεταλλεύονται τις μικροαρχιτεκτονικές ή ηλεκτρονικές ιδιότητες της συσκευής για την εξαγωγή ευαίσθητων πληροφοριών μέσω side channels ή fault injections. Η διάκριση αυτή επιτρέπει μια πιο συγκροτημένη κατανόηση του τοπίου των απειλών και διευκολύνει την αξιολόγηση της ασφάλειας σύγχρονων κρυπτογραφικών εφαρμογών σε πραγματικά περιβάλλοντα.

ΕΠΙΘΕΣΕΙΣ ΛΟΓΙΚΟΥ ΕΠΙΠΕΔΟΥ (ALGORITHM AND SOFTWARE LAYER)

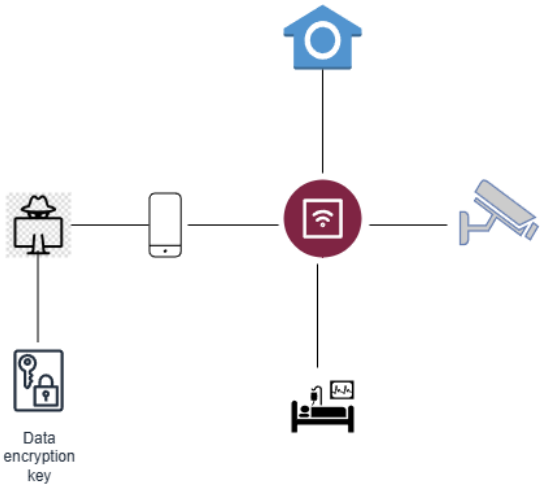
Οι επιθέσεις λογικού επιπέδου που στοχεύουν το αλγοριθμικό ή μαθηματικό υπόβαθρο των κρυπτογραφικών συστημάτων εστιάζουν αποκλειστικά στη θεωρητική δομή του αλγορίθμου, ανεξάρτητα από το hardware ή την υλοποίηση σε λογισμικό. Στόχος τους είναι να εντοπίσουν αδυναμίες στη σχεδίαση, στη διάχυση και στη συμπεριφορά των S-Boxes, ή στις γραμμικές και

μη γραμμικές λειτουργίες. Στην κατηγορία αυτή ανήκουν επιθέσεις όπως η Differential Cryptanalysis, η οποία αναλύει πώς διαφορές στην είσοδο επηρεάζουν το ciphertext· οι Algebraic και Cube Attacks, που διατυπώνουν τον κρυπτογραφικό αλγόριθμο ως σύστημα πολυωνυμικών εξισώσεων ώστε να λυθεί αποδοτικότερα από brute force. Ιδιαίτερο ενδιαφέρον παρουσιάζει η επίθεση meet-in-the-middle (MITM) [34]. Στις meet-in-the-middle επιθέσεις, ο αντίπαλος εκμεταλλεύεται τη δυνατότητα διάσπασης του κρυπτογραφικού σχήματος σε τμήματα και πραγματοποιεί παράλληλη διερεύνηση της κρυπτογράφησης και της αποκρυπτογράφησης του αλγορίθμου, αναζητώντας τα επιμέρους κομμάτια στα οποία μπορεί να πραγματοποιήσει συσχέτιση. Η μεθοδολογία αυτή μειώνει σημαντικά την υπολογιστική πολυπλοκότητα σε σχέση με μια ευθεία επίθεση brute force, ιδίως όταν συνδυάζεται με επιπλέον περιορισμούς τιμών (value constraints) που περιορίζουν τον χώρο των ενδιάμεσων παραμέτρων, όπως έχει παρουσιαστεί σε πρόσφατες εργασίες για τον AES. Τέλος, η Truncated Differential Cryptanalysis (TDC) αποτελεί επέκταση της κλασικής διαφορικής κρυπτανάλυσης, στην οποία εξετάζονται μερικώς καθορισμένες διαφορές αντί για πλήρως συγκεκριμένες τιμές. Η τεχνική αυτή επιτρέπει τη χρήση ευρύτερων και υψηλότερης πιθανότητας διαφορικών μονοπατιών, γεγονός που την καθιστά ιδιαίτερα αποτελεσματική στη μελέτη σύγχρονων block ciphers και sponge-based αλγορίθμων.

ΕΠΙΘΕΣΕΙΣ ΣΕ ΕΠΙΠΕΔΟ ΥΛΙΚΟΥ (HARDWARE LAYER)

Οι επιθέσεις σε επίπεδο hardware εκμεταλλεύονται φυσικές ιδιότητες και διαρροές πληροφορίας από το ίδιο το υλικό, όπως η κατανάλωση ισχύος, ο ηλεκτρομαγνητικός θόρυβος ή τα σφάλματα λειτουργίας, όπως οι επιθέσεις πλευρικού καναλιού.

Μία από τις ευπάθειες που αναφέρονται εκτενώς στη διεθνή βιβλιογραφία και απασχολούν την ερευνητική κοινότητα είναι οι επιθέσεις πλευρικού καναλιού (Side-Channel Attacks – SCAs), οι οποίες βασίζονται στην ανάλυση πλευρικών καναλιών (Side-Channel Analysis – SCA). Σε αντίθεση με τις παραδοσιακές επιθέσεις που στοχεύουν στη μαθηματική δομή του αλγορίθμου AES, οι SCAs εκμεταλλεύονται φυσικές διαρροές κατά την εκτέλεσή του, όπως μεταβολές στην κατανάλωση ισχύος, ηλεκτρομαγνητικές εκπομπές (EM), τον χρόνο εκτέλεσης ή τη συμπεριφορά της cache [35] [36]. Στο παρόν σχήμα, ο επιτιθέμενος εντοπίζει μία ευαίσθητη συσκευή (π.χ. έξυπνο ρολόι, έξυπνο κινητό), από την οποία μπορεί να ανακτήσει το μυστικό κλειδί. Αυτό οδηγεί σε μη εξουσιοδοτημένη πρόσβαση σε όλα τα συνδεδεμένα συστήματα του IoT οικοσυστήματος. Οι δύο βασικές μορφές Side Channel Attacks είναι η Correlation Power Analysis (CPA) και η differential power analysis (DPA). Οι επιθέσεις SCAs αποτελούν εμφανίζονται τόσο στον EAS όσο και στον Ascon.



Οι Template Attacks [37] αποτελούν μια ειδική κατηγορία των Side-Channel Attacks (SCAs), με την προϋπόθεση ότι ο επιτιθέμενος διαθέτει ένα αντίγραφο της συσκευής από την οποία θέλει να εξαγάγει πληροφορία. Στη φάση του profiling, πριν την πραγματική επίθεση, ο επιτιθέμενος καταγράφει το side-channel αποτύπωμα του αντιγράφου για διάφορους συνδυασμούς plaintext και κλειδιών, ώστε να δημιουργήσει στατιστικά πρότυπα (templates) της διαρροής. Έπειτα, κατά την φάση της επίθεσης, καταγράφει το side-channel αποτύπωμα της συσκευής-στόχου και το συγκρίνει με τα πρότυπα που έχουν δημιουργηθεί στο προηγούμενο βήμα, προκειμένου να εντοπίσει το κλειδί που χρησιμοποιήθηκε. Τέλος, η επίθεση DPA [38] στον AES εφαρμόζεται ώστε να εξαχθεί το μυστικό κλειδί (128 bit) και πραγματοποιείται καταγράφοντας τις ιχνογραφήσεις ισχύος (power traces) οι οποίες παράγονται κατά τις υπολογιστικές διεργασίες του αλγορίθμου.

ΤΡΟΠΟΙ ΜΕΤΡΙΑΣΜΟΥ ΤΩΝ ΕΠΙΘΕΣΕΩΝ

Οι Asfand et al. [38] στην έρευνα τους πρότειναν για τις επιθέσεις DPA ένα νέο τροποποιημένο masking scheme το οποίο αποτελείται από Boolean masking για τις γραμμικές πράξεις και multiplicative masking για τις μη γραμμικές πράξεις του AES.

ΣΥΜΠΕΡΑΣΜΑΤΑ - ΜΕΛΛΟΝΤΙΚΕΣ ΠΡΟΕΚΤΑΣΕΙΣ

Η παρούσα πτυχιακή εργασία εστιάζει στην ανάλυση, αξιολόγηση και πειραματική μελέτη του αλγορίθμου AES, τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο, ενώ παράλληλα εξετάζει την απόδοσή του σε σχέση με έναν νεότερο lightweight αλγόριθμο, τον Ascon, ο οποίος έχει υιοθετηθεί από το NIST ως πρότυπο για Lightweight Cryptography. Στόχος της εργασίας ήταν να αναδείξει την πρακτική αξία του AES, να διερευνήσει πλεονεκτήματα και περιορισμούς των

διαφορετικών modes λειτουργίας του, καθώς και να αξιολογήσει τη σχετική του αποδοτικότητα σε σύγκριση με σύγχρονες εναλλακτικές λύσεις. Στο Πειραματικό Μέρος 1, πραγματοποιήθηκε εκτενής μελέτη του AES σε διαφορετικά modes και padding schemes — συγκεκριμένα σε ECB Zero Padding, ECB PKCS#7, CBC Zero Padding και CBC PKCS#7. Για καθένα από τα modes μετρήθηκαν χρόνοι κρυπτογράφησης, αποκρυπτογράφησης και throughput για διάφορα μεγέθη αρχείων (100 KB, 1 MB, 10 MB και 50 MB). Από τα αποτελέσματα προέκυψε ότι οι διαφορές στις επιδόσεις μεταξύ των padding schemes είναι μικρές, ενώ καθίσταται σαφές ότι το ECB, παρά τις υψηλές ταχύτητες, παρουσιάζει σοβαρά προβλήματα ασφάλειας—κάτι που αποτυπώθηκε και στο πείραμα κρυπτογράφησης εικόνας όπου διατηρείται οπτικά το μοτίβο. Η λειτουργία CBC εμφανίστηκε ασφαλέστερη και με σταθερή απόδοση, χωρίς σημαντικό υπολογιστικό κόστος σε σχέση με το ECB. Στο Πειραματικό Μέρος 2, πραγματοποιήθηκε σύγκριση μεταξύ AES-GCM και Ascon-128a. Η μεθοδολογία περιλάμβανε κοινό σχήμα ανταλλαγής κλειδιού μέσω ECDH, ώστε να εξασφαλιστεί κοινός 128-bit session key. Και οι δύο αλγόριθμοι εξετάστηκαν σε κρυπτογράφηση και αποκρυπτογράφηση μεγάλων αρχείων. Τα αποτελέσματα έδειξαν ότι ο AES-GCM παραμένει ιδιαίτερα αποδοτικός στις περισσότερες περιπτώσεις, χάρη στις βελτιστοποιήσεις AES-NI των σύγχρονων επεξεργαστών. Ο Ascon, παρότι ελαφρύτερος και σχεδιασμένος για συστήματα περιορισμένων πόρων, παρουσιάζει ανταγωνιστική απόδοση σε μικρά μεγέθη δεδομένων, όμως ο AES-GCM διατηρεί σημαντικό πλεονέκτημα σε περιβάλλοντα με hardware acceleration. Συνολικά, η εργασία καταδεικνύει ότι ο AES εξακολουθεί να αποτελεί τον πλέον αξιόπιστο και αποδοτικό αλγόριθμο για γενικής χρήσης εφαρμογές κρυπτογράφησης, ενώ ο Ascon αποτελεί πολλά υποσχόμενη επιλογή για lightweight και IoT συσκευές. Τα πειραματικά αποτελέσματα αναδεικνύουν τις διαφορές μεταξύ των modes του AES και υποστηρίζουν την ανάγκη επιλογής κατάλληλου mode ανάλογα με τις απαιτήσεις ασφαλείας και απόδοσης.

Με βάση τα αποτελέσματα αυτής της μελέτης, μια σημαντική μελλοντική προέκταση αφορά τη διερεύνηση της συμπεριφοράς των αλγορίθμων σε IoT και IoNT [39] συσκευές και μικροελεγκτές, όπου οι περιορισμοί σε επεξεργαστική ισχύ, μνήμη και ενεργειακή κατανάλωση καθιστούν τους ελαφρούς αλγορίθμους ιδιαίτερα κρίσιμους. Η απόδοση του AES και του Ascon σε τέτοια περιβάλλοντα μπορεί να οδηγήσει σε χρήσιμα συμπεράσματα σχετικά με την επιλογή του κατάλληλου κρυπτογραφικού σχήματος για συστήματα πραγματικού χρόνου, αισθητήρες, smart devices ή embedded εφαρμογές. Επιπλέον, μελλοντική έρευνα μπορεί να επεκταθεί στην ανάλυση επιθέσεων και πρακτικών ευπαθειών των υλοποιήσεων, καθώς η ασφάλεια δεν εξαρτάται μόνο από τον αλγόριθμο, αλλά και από τον τρόπο ενσωμάτωσής του στο λογισμικό και στο υλικό. Επιπροσθέτως, η αξιολόγηση των αλγορίθμων σε ετερογενή υπολογιστικά περιβάλλοντα, καθώς και η μελέτη της ανθεκτικότητάς τους απέναντι σε μετα-κβαντικές απειλές, συνιστούν σημαντικές κατευθύνσεις για μελλοντική διεύρυνση της έρευνας. Τέλος, μια ιδιαίτερα ενδιαφέρουσα κατεύθυνση για μελλοντική έρευνα αποτελεί η διερεύνηση τεχνικών κρυπτανάλυσης που βασίζονται στη μηχανική μάθηση, και ειδικότερα σε μοντέλα βαθιών νευρωνικών δικτύων. Οι προσεγγίσεις αυτές έχουν δείξει ότι μπορούν να ενισχύσουν επιθέσεις side-channel, όπως η power analysis, μέσω αυτοματοποιημένης εξαγωγής χαρακτηριστικών και αναγνώρισης προτύπων σε δεδομένα διαρροής [40]. Η εφαρμογή τους σε υλοποιήσεις των AES και Ascon θα μπορούσε να προσφέρει ένα πιο ολοκληρωμένο πλαίσιο αξιολόγησης της

ανθεκτικότητάς τους έναντι σύγχρονων και εξελιγμένων μορφών φυσικών επιθέσεων, αναδεικνύοντας πιθανά σημεία ευπάθειας που δεν εντοπίζονται με κλασικές τεχνικές.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] National Institute of Standards and Technology (US), 'Advanced Encryption Standard (AES)', National Institute of Standards and Technology (U.S.), Washington, D.C., NIST FIPS 197-upd1, Μαΐου 2023. doi: 10.6028/NIST.FIPS.197-upd1.

[2] J. Daemen και V. Rijmen, *The Design of Rijndael*. στο Information Security and Cryptography. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. doi: 10.1007/978-3-662-04722-4.

[3] N. I. of Standards και Technology, 'Advanced Encryption Standard (AES)', U.S. Department of Commerce, Washington, D.C., Federal Information Processing Standards Publication FIPS 197-upd1, 2001. Ημερομηνία πρόσβασης: 25 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://doi.org/10.6028/NIST.FIPS.197-upd1

[4] Y. Alemami, A. Al-Ghonmein, K. Al-Moghrabi, και M. Afendee, 'Cloud data security and various cryptographic algorithms Corresponding Author', *Int. J. Electr. Comput. Eng. IJECE*, τ. 13, σελ. 1867–1879, Απριλίου 2023, doi: 10.11591/ijece.v13i2.pp1867-1879.

[5] M. H. M. Baig, H. B. U. Haq, και W. Habib, 'A Comparative Analysis of AES, RSA, and 3DES Encryption Standards based on Speed and Performance', *Manag. Sci. Adv.*, τ. 1, τχ. 1, Art. τχ. 1, Νοεμβρίου 2024, doi: 10.31181/msa1120244.

[6] 'Comparison of Data Encryption Standard (DES) and Advanced Encryption Standard (AES) in Security Issues of Cloud Computing: A Literature Review', στο *Smart Innovation, Systems and Technologies*, Singapore: Springer Nature Singapore, 2025, σελ. 189–200. doi: 10.1007/978-981-96-0147-9_16.

[7] R. A. Manurung, S. Sutarman, και S. Efendi, 'Comparative Analysis of the Performance of Four Symmetric Algorithms on Digital File Security', *J. Inform. Telecommun. Eng.*, τ. 8, τχ. 2, σελ. 152–164, Ιανουαρίου 2025, doi: 10.31289/jite.v8i2.13978.

[8] B. A. Buhari *κ.ά.*, 'Performance and Security Analysis of Symmetric Data Encryption Algorithms: AES, 3DES and Blowfish', *Int. J. Adv. Netw. Appl.*, τ. 16, τχ. 04, σελ. 6473–6486, 2025, doi: 10.35444/ijana.2025.16404.

[9] K. Assa-Agyei και F. Olajide, 'A Comparative Study of Twofish, Blowfish, and Advanced Encryption Standard for Secured Data Transmission', *Int. J. Adv. Comput. Sci. Appl.*, τ. 14, τχ. 3, 2023, doi: 10.14569/ijacsa.2023.0140344.

[10] K. Patel, 'Performance analysis of AES, DES and Blowfish cryptographic algorithms on small and large data files', *Int. J. Inf. Technol.*, τ. 11, τχ. 4, σελ. 813–819, Δεκεμβρίου 2019, doi: 10.1007/s41870-018-0271-4.

[11] B. A. Buhari, A. A. Obiniyi, K. Sunday, και S. Shehu, 'Performance Evaluation of Symmetric Data Encryption Algorithms: AES and Blowfish', *Saudi J. Eng. Technol.*, τ. 04, τχ. 10, σελ. 407–414, Οκτωβρίου 2019, doi: 10.36348/sjeat.2019.v04i10.002.

[12] D. Chattaraj, S. K. B, B. Maji, και S. Tadkal, 'Performance Evaluation of Cryptographic Algorithms on Resource-constrained and GPU/TPU integrated Multi-core Processor Systems', στο *2025 6th International Conference on Control, Communication and Computing (ICCC)*, Φεβρουαρίου 2025, σελ. 1–6. doi: 10.1109/ICCC64910.2025.11077192.

[13] H. Dibas και K. E. Sabri, 'A comprehensive performance empirical study of the symmetric algorithms:AES, 3DES, Blowfish and Twofish', στο *2021 International Conference on Information Technology (ICIT)*, Ιουλίου 2021, σελ. 344–349. doi: 10.1109/ICIT52682.2021.9491644.

[14] J. Katz και Y. Lindell, *Introduction to modern cryptography*. στο Chapman & Hall/CRC cryptography and network security. Boca Raton, Fla.: Chapman & Hall/CRC, 2008.

[15] O. D. Matteo, 'A very brief introduction to finite fields'.

[16] B.-H. Lee, E. Dewi, και M. Wajdi, *Data security in cloud computing using AES under HEROKU cloud*. 2018, σελ. 5. doi: 10.1109/WOCC.2018.8372705.
[17] S. Frankel, K. R. Glenn, και S. G. Kelly, 'The AES-CBC Cipher Algorithm and Its Use with IPsec', Internet Engineering Task Force, Request for Comments RFC 3602, Ιουνίου 2003. doi: 10.17487/RFC3602.
[18] 'Folie 1'. Ημερομηνία πρόσβασης: 9 Σεπτέμβριος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://cosec.bit.uni-bonn.de/fileadmin/user_upload/teaching/11ss/blockciphers/Talks/Tim_Syben.pdf
[19] K. Haria, R. Shah, V. Jain, και R. Mangrulkar, 'Enhanced image encryption using AES algorithm with CBC mode: a secure and efficient approach', *Iran J. Comput. Sci.*, τ. 7, τχ. 3, σελ. 589–605, Σεπτεμβρίου 2024, doi: 10.1007/s42044-024-00191-y.
[20] W. Stallings, 'Cryptography and Network Security: Principles and Practice'.
[21] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, και R. Cammarota, 'Post-Quantum Lattice-Based Cryptography Implementations: A Survey', *ACM Comput. Surv.*, τ. 51, τχ. 6, σελ. 1–41, Νοεμβρίου 2019, doi: 10.1145/3292548.
[22] 'NIST SP 800-38D, Recommendationfor Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC'. Ημερομηνία πρόσβασης: 12 Νοέμβριος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf
[23] E. Käsper και P. Schwabe, 'Faster and Timing-Attack Resistant AES-GCM', στο *Cryptographic Hardware and Embedded Systems - CHES 2009*, C. Clavier και K. Gaj, Επιμ., Berlin, Heidelberg: Springer, 2009, σελ. 1–17. doi: 10.1007/978-3-642-04138-9_1.
[24] L. Chen, D. Moody, A. Regenscheid, A. Robinson, και K. Randall, 'Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters', National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-186, Φεβρουαρίου 2023. doi: 10.6028/NIST.SP.800-186.
[25] J. H. Silverman, 'The Xedni Calculus and the Elliptic Curve Discrete Logarithm Problem', *Des. Codes Cryptogr.*, τ. 20, τχ. 1, σελ. 5–40, Απριλίου 2000, doi: 10.1023/A:1008319518035.
[26] W. J. Buchanan, J. Gilchrist, και K. Finlow-Bates, 'ECDSA Cracking Methods', 9 Απρίλιος 2025, *arXiv*: arXiv:2504.07265. doi: 10.48550/arXiv.2504.07265.
[27] M. Sonmez Turan, 'Ascon-Based Lightweight Cryptography Standards for Constrained Devices', National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-232, 2025. doi: 10.6028/NIST.SP.800-232.
[28] C. Dobraunig, M. Eichlseder, F. Mendel, και M. Schläffer, 'Ascon v1.2: Lightweight Authenticated Encryption and Hashing', *J. Cryptol.*, τ. 34, τχ. 3, σελ. 33, Ιουλίου 2021, doi: 10.1007/s00145-021-09398-9.
[29] V. Sarasa Laborda, L. Hernández-Álvarez, L. Hernández Encinas, J. I. Sánchez García, και A. Queiruga-Dios, 'Study About the Performance of Ascon in Arduino Devices', *Appl. Sci.*, τ. 15, τχ. 7, σελ. 4071, Ιανουαρίου 2025, doi: 10.3390/app15074071.
[30] C. Gewehr *κ.ά.*, 'Hardware Acceleration of Authenticated Encryption with Associated Data via RISC-V Instruction Set Extensions in Low Power Embedded Systems', στο *2024 IEEE 15th Latin America Symposium on Circuits and Systems (LASCAS)*, Οκτωβρίου 2024, σελ. 1–5. doi: 10.1109/LASCAS60203.2024.10506132.
[31] T. Kitahara, R. Hira, Y. Hara-Azumi, D. Miyahara, Y. Li, και K. Sakiyama, 'Optimized Software Implementations of Ascon, Grain-128AEAD, and TinyJambu on ARM Cortex-M0', στο *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*, Αυγούστου 2022, σελ. 316–322. doi: 10.1109/CANDARW57323.2022.00030.
[32] 'Efficiency and Security Evaluation of Lightweight Cryptographic Algorithms for Resource-Constrained IoT Devices'. Ημερομηνία πρόσβασης: 25 Νοέμβριος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://www.mdpi.com/1424-8220/24/12/4008
[33] D. Weng, 'Performance and Energy Evaluation of Lightweight Cryptography for Small IoT Devices', στο *2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, Ιουλίου 2023, σελ. 289–295. doi: 10.1109/UEMCON59035.2023.10316062.
[34] X. Dong, J. Liu, Y. Wei, W. Gao, και J. Chen, 'Meet-in-the-middle attacks on AES with value

constraints', *Des. Codes Cryptogr.*, τ. 92, τχ. 9, σελ. 2423–2449, Σεπτεμβρίου 2024, doi: 10.1007/s10623-024-01396-9.
[35] S. Naserelden, N. Alias, A. Altigani, A. Mohamed, και S. Badreddine, 'Advance attacks on AES: A comprehensive review of side channel, fault injection, machine learning and quantum techniques', τ. 9, σελ. 2471–2486, Απριλίου 2025, doi: 10.55214/25768484.v9i4.6586.
[36] S. Ahmed *κ.ά.*, 'Lightweight AES Design for IoT Applications: Optimizations in FPGA and ASIC With DFA Countermeasure Strategies', *IEEE Access*, τ. 13, σελ. 22489–22509, 2025, doi: 10.1109/ACCESS.2025.3533611.
[37] S. Chari, J. R. Rao, και P. Rohatgi, 'Template Attacks', στο *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, çetin K. Koç, και C. Paar, Επιμ., Berlin, Heidelberg: Springer, 2003, σελ. 13–28. doi: 10.1007/3-540-36400-5_3.
[38] M. Asfand Hafeez, M. Mazyad Hazzazi, H. Tariq, A. Aljaedi, A. Javed, και A. R. Alharbi, 'A Low-Overhead Countermeasure against Differential Power Analysis for AES Block Cipher', *Appl. Sci.*, τ. 11, τχ. 21, σελ. 10314, Ιανουαρίου 2021, doi: 10.3390/app112110314.
[39] A. Alabdulatif, N. N. Thilakarathne, Z. K. Lawal, K. E. Fahim, και R. Y. Zakari, 'Internet of Nano-Things (IoNT): A Comprehensive Review from Architecture to Security and Privacy Challenges', *Sensors*, τ. 23, τχ. 5, σελ. 2807, Ιανουαρίου 2023, doi: 10.3390/s23052807.
[40] M. Degré, P. Derbez, L. Lahaye, και A. Schrottenloher, 'New models for the cryptanalysis of ASCON', *Des. Codes Cryptogr.*, τ. 93, τχ. 6, σελ. 2055–2072, Ιουνίου 2025, doi: 10.1007/s10623-025-01572-5.

ΠΑΡΑΡΤΗΜΑ

ΠΡΟΓΡΑΜΜΑ 1 ΣΕ PYTHON

```
import os
import time
import psutil
from Crypto.Cipher import AES
# για ECDH
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.backends import default_backend
# ----------------------------------------
# ECDH: παραγωγή συμμετρικού κλειδιού 128 bit
# ----------------------------------------
def ecdh_derive_key():
```

```
    """
    Χρησιμοποιεί Elliptic Curve Diffie–Hellman (SECP256R1)
    για να παραγάγει κοινό μυστικό και παίρνει τα πρώτα 16 bytes
    ως κλειδί AES-128.
    Επιστρέφει (key, time_sec).
    """
    t0 = time.perf_counter()
    # private keys για τα δύο μέρη
    priv_a = ec.generate_private_key(ec.SECP256R1(), backend=default_backend())
    priv_b = ec.generate_private_key(ec.SECP256R1(), backend=default_backend())
    # shared secrets (πρέπει να είναι ίσα)
    shared_a = priv_a.exchange(ec.ECDH(), priv_b.public_key())
    shared_b = priv_b.exchange(ec.ECDH(), priv_a.public_key())
    assert shared_a == shared_b
    # για το πείραμα, παίρνουμε τα πρώτα 16 bytes ως AES-128 key
    key = shared_a[:16]
    t1 = time.perf_counter()
    return key, (t1 - t0)
# ---------------------------------------
# Ρυθμίσεις
# ---------------------------------------
sizes = [
    100 * 1024,       # 100 KB
    1 * 1024 * 1024,  # 1 MB
    10 * 1024 * 1024, # 10 MB
    50 * 1024 * 1024  # 50 MB
]
repetitions = 10
# ---------------------------------------
# Zero padding helpers
# ---------------------------------------
def zero_pad(data: bytes, block_size: int = AES.block_size) -> bytes:
    padding_len = (block_size - (len(data) % block_size)) % block_size
    return data if padding_len == 0 else data + b"\x00" * padding_len
def zero_unpad(data: bytes) -> bytes:
    return data.rstrip(b"\x00")
# ---------------------------------------
```

```
# RSS μνήμης
# ---------------------------------------
proc = psutil.Process(os.getpid())
def rss_mb() -> float:
    return proc.memory_info().rss / (1024 * 1024)
# ---------------------------------------
# Ένας καθαρός κύκλος ECB: encrypt -> decrypt
# ---------------------------------------
def run_cycle_ecb(plaintext: bytes, key: bytes):
    padded = zero_pad(plaintext, AES.block_size)
    # --- Κρυπτογράφηση ---
    t0_wall = time.perf_counter()
    t0_cpu = time.process_time()
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(padded)
    enc_wall = time.perf_counter() - t0_wall
    enc_cpu = time.process_time() - t0_cpu
    # --- Αποκρυπτογράφηση ---
    t1_wall = time.perf_counter()
    t1_cpu = time.process_time()
    decipher = AES.new(key, AES.MODE_ECB)
    decrypted = decipher.decrypt(ciphertext)
    unpadded = zero_unpad(decrypted)
    dec_wall = time.perf_counter() - t1_wall
    dec_cpu = time.process_time() - t1_cpu
    # προαιρετικός έλεγχος ορθότητας
    if unpadded != plaintext:
        raise ValueError("Decryption / unpadding failed!")
    return {
        "enc_wall": enc_wall,
        "enc_cpu": enc_cpu,
        "dec_wall": dec_wall,
        "dec_cpu": dec_cpu,
    }
# ---------------------------------------
# Κύριο πρόγραμμα
# ---------------------------------------
```

```
# 1. ECDH key exchange
aes_key, ecdh_time = ecdh_derive_key()
print(f"ECDH time (SECP256R1, AES-128 key derivation): {ecdh_time:.6f} s\n")
results = []
for size in sizes:
    # δημιουργούμε δεδομένα στη μνήμη (χωρίς δίσκο)
    plaintext = os.urandom(size)
    metrics = []
    rss_before = rss_mb()
    for _ in range(repetitions):
        m = run_cycle_ecb(plaintext, aes_key)
        metrics.append(m)
    rss_after = rss_mb()
    rss_peak = max(rss_before, rss_after)
    # μέσοι όροι
    avg_enc_wall = sum(m["enc_wall"] for m in metrics) / repetitions
    avg_enc_cpu = sum(m["enc_cpu"] for m in metrics) / repetitions
    avg_dec_wall = sum(m["dec_wall"] for m in metrics) / repetitions
    avg_dec_cpu = sum(m["dec_cpu"] for m in metrics) / repetitions
    # throughput σε MB/s
    size_mb = size / (1024 * 1024)
    enc_throughput = size_mb / avg_enc_wall if avg_enc_wall > 0 else 0.0
    dec_throughput = size_mb / avg_dec_wall if avg_dec_wall > 0 else 0.0
    results.append({
        "size_bytes": size,
        "size_mb": size_mb,
        "enc_wall": avg_enc_wall,
        "enc_cpu": avg_enc_cpu,
        "dec_wall": avg_dec_wall,
        "dec_cpu": avg_dec_cpu,
        "enc_throughput": enc_throughput,
        "dec_throughput": dec_throughput,
        "rss_peak": rss_peak,
    })
# ------------------------------------
# Αναφορά
# ------------------------------------
```

```
print("\nΑποτελέσματα μετρήσεων AES – ECB – Zero Padding "
      f"(μέσοι όροι από {repetitions} επαναλήψεις):\n")

print(f"{'Μέγεθος':10} | {'Enc wall (s)':>11} | {'Enc MB/s':>9} | "
      f"{'Dec wall (s)':>11} | {'Dec MB/s':>9} | {'RSS peak (MB)':>12}")
print("-" * 92)
for r in results:
    # εδώ διορθώσαμε το '0 MB' ώστε το 100 KB να φαίνεται ως 0.1 MB
    size_label = f"{r['size_mb']:.1f} MB"
    print(f"{size_label:10} | "
          f"{r['enc_wall']:11.6f} | {r['enc_throughput']:9.2f} | "
          f"{r['dec_wall']:11.6f} | {r['dec_throughput']:9.2f} | "
          f"{r['rss_peak']:12.2f}")
```

ΠΡΟΓΡΑΜΜΑ 2 ΣΕ PYTHON

```
import os
import time
import psutil
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from cryptography.hazmat.primitives.asymmetric import ec

# ---------------------------------------
# ECDH Key Exchange (P-256)
# ---------------------------------------
def ecdh_derive_key():
    privA = ec.generate_private_key(ec.SECP256R1())
    privB = ec.generate_private_key(ec.SECP256R1())
    pubA = privA.public_key()
    pubB = privB.public_key()
    shared_A = privA.exchange(ec.ECDH(), pubB)
    shared_B = privB.exchange(ec.ECDH(), pubA)
    assert shared_A == shared_B
    # Παίρνουμε τα πρώτα 16 bytes (128-bit key)
    return shared_A[:16]
# ---------------------------------------
# Zero padding helpers
```

```
# ---------------------------------------
def zero_pad(data: bytes, block_size: int = AES.block_size) -> bytes:
    padding_len = (block_size - (len(data) % block_size)) % block_size
    return data if padding_len == 0 else data + b"\x00" * padding_len
def zero_unpad(data: bytes) -> bytes:
    return data.rstrip(b"\x00")
# ---------------------------------------
# RSS μνήμης
# ---------------------------------------
proc = psutil.Process(os.getpid())
def rss_mb() -> float:
    return proc.memory_info().rss / (1024 * 1024)
# ---------------------------------------
# CBC encrypt -> decrypt
# ---------------------------------------
def run_cycle_cbc(plaintext: bytes, key: bytes):
    padded = zero_pad(plaintext, AES.block_size)
    # --- Κρυπτογράφηση ---
    iv = get_random_bytes(16)
    t0_wall = time.perf_counter()
    t0_cpu = time.process_time()
    cipher = AES.new(key, AES.MODE_CBC, iv=iv)
    ciphertext = cipher.encrypt(padded)
    enc_wall = time.perf_counter() - t0_wall
    enc_cpu = time.process_time() - t0_cpu
    # --- Αποκρυπτογράφηση ---
    t1_wall = time.perf_counter()
    t1_cpu = time.process_time()
    decipher = AES.new(key, AES.MODE_CBC, iv=iv)
    decrypted = decipher.decrypt(ciphertext)
    unpadded = zero_unpad(decrypted)
    dec_wall = time.perf_counter() - t1_wall
    dec_cpu = time.process_time() - t1_cpu
    return {
        "enc_wall": enc_wall,
        "enc_cpu": enc_cpu,
        "dec_wall": dec_wall,
```

```
        "dec_cpu": dec_cpu,
    }
# ------------------------------------
# Κύριο πρόγραμμα
# ------------------------------------
# Μετράμε χρόνο ECDH και παίρνουμε session key
t0 = time.perf_counter()
session_key = ecdh_derive_key()
t1 = time.perf_counter()
print(f"\nECDH Key Exchange Time: {t1 - t0:.6f} s\n")
sizes = [
    100 * 1024,       # 100 KB
    1 * 1024 * 1024,  # 1 MB
    10 * 1024 * 1024, # 10 MB
    50 * 1024 * 1024  # 50 MB
]
repetitions = 10
results = []
for size in sizes:
    plaintext = os.urandom(size)
    metrics = []
    rss_before = rss_mb()
    for _ in range(repetitions):
        m = run_cycle_cbc(plaintext, session_key)
        metrics.append(m)
    rss_after = rss_mb()
    rss_peak = max(rss_before, rss_after)
    avg_enc_wall = sum(m["enc_wall"] for m in metrics) / repetitions
    avg_enc_cpu  = sum(m["enc_cpu"]  for m in metrics) / repetitions
    avg_dec_wall = sum(m["dec_wall"] for m in metrics) / repetitions
    avg_dec_cpu  = sum(m["dec_cpu"]  for m in metrics) / repetitions
    size_mb = size / (1024 * 1024)
    enc_throughput = size_mb / avg_enc_wall if avg_enc_wall > 0 else 0.0
    dec_throughput = size_mb / avg_dec_wall if avg_dec_wall > 0 else 0.0
    results.append({
        "size_bytes": size,
        "size_mb": size_mb,
```

```
        "enc_wall": avg_enc_wall,
        "enc_cpu": avg_enc_cpu,
        "dec_wall": avg_dec_wall,
        "dec_cpu": avg_dec_cpu,
        "enc_throughput": enc_throughput,
        "dec_throughput": dec_throughput,
        "rss_peak": rss_peak
    })
# ---------------------------------------
# Αναφορά
# ---------------------------------------
print("\nΑποτελέσματα μετρήσεων CBC – Zero Padding – AES "
      f"(μέσοι όροι από {repetitions} επαναλήψεις, χωρίς I/O):\n")
print(f"{'Μέγεθος':12} | {'Enc wall (s)':>11} | {'Enc MB/s':>9} | "
      f"{'Dec wall (s)':>11} | {'Dec MB/s':>9} | {'RSS peak (MB)':>12}")
print("-" * 90)
for r in results:
    size_bytes = r["size_bytes"]
    # Ετικέτα μεγέθους: KB κάτω από 1MB, αλλιώς MB
    if size_bytes < 1024 * 1024:
        size_label = f"{size_bytes // 1024} KB"
    else:
        size_label = f"{size_bytes // (1024 * 1024)} MB"
    print(f"{size_label:>8} | "
          f"{r['enc_wall']:11.6f} | {r['enc_throughput']:9.2f} | "
          f"{r['dec_wall']:11.6f} | {r['dec_throughput']:9.2f} | "
          f"{r['rss_peak']:12.2f}")
```

ΠΡΟΓΡΑΜΜΑ 3 ΣΕ PYTHON

```
import os
import time
import psutil
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
```

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
# ---------------------------------------
# ECDH Key Exchange (P-256)
# ---------------------------------------
def ecdh_derive_key():
    privA = ec.generate_private_key(ec.SECP256R1())
    privB = ec.generate_private_key(ec.SECP256R1())
    pubA = privA.public_key()
    pubB = privB.public_key()
    shared_A = privA.exchange(ec.ECDH(), pubB)
    shared_B = privB.exchange(ec.ECDH(), pubA)
    assert shared_A == shared_B
    # Παίρνουμε 128 bit key (16 bytes)
    return shared_A[:16]
# ---------------------------------------
# PKCS#7 padding
# ---------------------------------------
def pkcs7_pad(data: bytes, block_size: int = AES.block_size) -> bytes:
    padding_len = block_size - (len(data) % block_size)
    return data + bytes([padding_len]) * padding_len
def pkcs7_unpad(data: bytes) -> bytes:
    padding_len = data[-1]
    return data[:-padding_len]
# ---------------------------------------
# RSS memory usage
# ---------------------------------------
proc = psutil.Process(os.getpid())
def rss_mb() -> float:
    return proc.memory_info().rss / (1024 * 1024)
# ---------------------------------------
# CBC encrypt/decrypt cycle
# ---------------------------------------
def run_cycle_ecb_pkcs7(plaintext: bytes, key: bytes):
    padded = pkcs7_pad(plaintext)
    # Encrypt
    t0_wall = time.perf_counter()
```

```
    t0_cpu = time.process_time()
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(padded)
    enc_wall = time.perf_counter() - t0_wall
    enc_cpu = time.process_time() - t0_cpu
    # Decrypt
    t1_wall = time.perf_counter()
    t1_cpu = time.process_time()
    decipher = AES.new(key, AES.MODE_ECB)
    decrypted = decipher.decrypt(ciphertext)
    unpadded = pkcs7_unpad(decrypted)
    dec_wall = time.perf_counter() - t1_wall
    dec_cpu = time.process_time() - t1_cpu
    return {
        "enc_wall": enc_wall,
        "enc_cpu": enc_cpu,
        "dec_wall": dec_wall,
        "dec_cpu": dec_cpu,
    }
# ---------------------------------------
# MAIN
# ---------------------------------------
sizes = [
    100 * 1024,       # 100 KB
    1 * 1024 * 1024,  # 1 MB
    10 * 1024 * 1024, # 10 MB
    50 * 1024 * 1024  # 50 MB
]
repetitions = 10
# --- Μέτρηση ECDH ---
t0 = time.perf_counter()
session_key = ecdh_derive_key()
t1 = time.perf_counter()
print(f"\nECDH Key Exchange Time: {t1 - t0:.6f} s\n")
results = []
for size in sizes:
    plaintext = os.urandom(size)
```

```
    metrics = []
    rss_before = rss_mb()
    for _ in range(repetitions):
        metrics.append(run_cycle_ecb_pkcs7(plaintext, session_key))
    rss_after = rss_mb()
    rss_peak = max(rss_before, rss_after)
    avg_enc_wall = sum(m["enc_wall"] for m in metrics) / repetitions
    avg_enc_cpu  = sum(m["enc_cpu"]  for m in metrics) / repetitions
    avg_dec_wall = sum(m["dec_wall"] for m in metrics) / repetitions
    avg_dec_cpu  = sum(m["dec_cpu"]  for m in metrics) / repetitions
    size_mb = size / (1024 * 1024)
    enc_throughput = size_mb / avg_enc_wall
    dec_throughput = size_mb / avg_dec_wall
    results.append({
        "size_bytes": size,
        "size_mb": size_mb,
        "enc_wall": avg_enc_wall,
        "dec_wall": avg_dec_wall,
        "enc_throughput": enc_throughput,
        "dec_throughput": dec_throughput,
        "rss_peak": rss_peak
    })
# ---------------------------------------
# DISPLAY RESULTS
# ---------------------------------------
print("\nΑποτελέσματα μετρήσεων AES – ECB – PKCS#7 "
      f"(μέσοι όροι από {repetitions} επαναλήψεις):\n")
print(f"{'Μέγεθος':12} | {'Enc wall (s)':>11} | {'Enc MB/s':>11} | "
      f"{'Dec wall (s)':>11} | {'Dec MB/s':>11} | {'RSS peak (MB)':>12}")
print("-" * 95)
for r in results:
    size_bytes = r["size_bytes"]
    # Ετικέτα μεγέθους (KB για μικρά, MB για μεγάλα)
    if size_bytes < 1024 * 1024:
        size_label = f"{size_bytes//1024} KB"
    else:
        size_label = f"{size_bytes//(1024*1024)} MB"
```

```
    print(f"{size_label:>12} | "
          f"{r['enc_wall']:11.6f} | {r['enc_throughput']:11.2f} | "
          f"{r['dec_wall']:11.6f} | {r['dec_throughput']:11.2f} | "
          f"{r['rss_peak']:12.2f}")
```

ΠΡΟΓΡΑΜΜΑ 4 ΣΕ PYTHON

```
import os
import time
import psutil
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
# ---------------------------------------
# ECDH Key Exchange (P-256 / SECP256R1)
# ---------------------------------------
def ecdh_derive_key():
    # Private keys για τα δύο μέρη
    privA = ec.generate_private_key(ec.SECP256R1())
    privB = ec.generate_private_key(ec.SECP256R1())
    pubA = privA.public_key()
    pubB = privB.public_key()
    # Υπολογισμός shared secret και από τις δύο πλευρές
    shared_A = privA.exchange(ec.ECDH(), pubB)
    shared_B = privB.exchange(ec.ECDH(), pubA)
    # Πρέπει να είναι ίσα (έλεγχος ορθότητας ECDH)
    assert shared_A == shared_B
    # Παίρνουμε τα πρώτα 16 bytes (128-bit key) για AES-128
    return shared_A[:16]
# ---------------------------------------
# Ρυθμίσεις
# ---------------------------------------
sizes = [
    100 * 1024,       # 100 KB
    1 * 1024 * 1024,  # 1 MB
    10 * 1024 * 1024, # 10 MB
    50 * 1024 * 1024  # 50 MB
```

```
]
repetitions = 10
# ---------------------------------------
# PKCS#7 padding helpers
# ---------------------------------------
def pkcs7_pad(data: bytes, block_size: int = AES.block_size) -> bytes:
    padding_len = block_size - (len(data) % block_size)
    return data + bytes([padding_len]) * padding_len
def pkcs7_unpad(data: bytes) -> bytes:
    if not data:
        return data
    padding_len = data[-1]
    if padding_len < 1 or padding_len > AES.block_size:
        raise ValueError("Bad PKCS#7 padding")
    return data[:-padding_len]
# ---------------------------------------
# RSS μνήμης
# ---------------------------------------
proc = psutil.Process(os.getpid())
def rss_mb() -> float:
    return proc.memory_info().rss / (1024 * 1024)
# ---------------------------------------
# Ένας καθαρός κύκλος CBC (PKCS#7, χωρίς I/O)
# ---------------------------------------
def run_cycle_cbc_pkcs7(plaintext: bytes, key: bytes):
    padded = pkcs7_pad(plaintext, AES.block_size)
    # --- Κρυπτογράφηση ---
    iv = get_random_bytes(16)  # CBC χρειάζεται IV
    t0_wall = time.perf_counter()
    t0_cpu = time.process_time()
    cipher = AES.new(key, AES.MODE_CBC, iv=iv)
    ciphertext = cipher.encrypt(padded)
    enc_wall = time.perf_counter() - t0_wall
    enc_cpu = time.process_time() - t0_cpu
    # --- Αποκρυπτογράφηση ---
    t1_wall = time.perf_counter()
    t1_cpu = time.process_time()
```

```
    decipher = AES.new(key, AES.MODE_CBC, iv=iv)
    decrypted = decipher.decrypt(ciphertext)
    unpadded = pkcs7_unpad(decrypted)
    dec_wall = time.perf_counter() - t1_wall
    dec_cpu = time.process_time() - t1_cpu
    return {
        "enc_wall": enc_wall,
        "enc_cpu": enc_cpu,
        "dec_wall": dec_wall,
        "dec_cpu": dec_cpu,
    }
# ---------------------------------------
# Κύριο πρόγραμμα
# ---------------------------------------
# Παράγουμε AES-128 κλειδί μέσω ECDH
t_ecdh_start = time.perf_counter()
session_key = ecdh_derive_key()
t_ecdh_end = time.perf_counter()
ecdh_time = t_ecdh_end - t_ecdh_start

print(f"\nECDH Key Exchange Time (SECP256R1 → 128-bit key): {ecdh_time:.6f} s\n")
results = []
for size in sizes:
    plaintext = os.urandom(size)
    metrics = []
    rss_before = rss_mb()
    for _ in range(repetitions):
        m = run_cycle_cbc_pkcs7(plaintext, session_key)
        metrics.append(m)
    rss_after = rss_mb()
    rss_peak = max(rss_before, rss_after)
    # Μέσοι όροι
    avg_enc_wall = sum(m["enc_wall"] for m in metrics) / repetitions
    avg_enc_cpu  = sum(m["enc_cpu"]  for m in metrics) / repetitions
    avg_dec_wall = sum(m["dec_wall"] for m in metrics) / repetitions
    avg_dec_cpu  = sum(m["dec_cpu"]  for m in metrics) / repetitions
    # Throughput (MB/s)
```

```
    size_mb = size / (1024 * 1024)
    enc_throughput = size_mb / avg_enc_wall if avg_enc_wall > 0 else 0.0
    dec_throughput = size_mb / avg_dec_wall if avg_dec_wall > 0 else 0.0
    results.append({
        "size_bytes": size,
        "size_mb": size_mb,
        "enc_wall": avg_enc_wall,
        "enc_cpu": avg_enc_cpu,
        "dec_wall": avg_dec_wall,
        "dec_cpu": avg_dec_cpu,
        "enc_throughput": enc_throughput,
        "dec_throughput": dec_throughput,
        "rss_peak": rss_peak,
    })
# ---------------------------------------
# Αναφορά
# ---------------------------------------
print("\nΑποτελέσματα μετρήσεων AES – CBC – PKCS#7 "
      f"(μέσοι όροι από {repetitions} επαναλήψεις, με κλειδί από ECDH):\n")

print(f"{'Μέγεθος':12} | {'Enc wall (s)':>11} | {'Enc MB/s':>9} | "
      f"{'Dec wall (s)':>11} | {'Dec MB/s':>9} | {'RSS peak (MB)':>12}")
print("-" * 90)
for r in results:
    size_bytes = r["size_bytes"]
    if size_bytes < 1024 * 1024:
        size_label = f"{size_bytes // 1024} KB"
    else:
        size_label = f"{size_bytes // (1024 * 1024)} MB"
    print(f"{size_label:>8} | "
          f"{r['enc_wall']:11.6f} | {r['enc_throughput']:9.2f} | "
          f"{r['dec_wall']:11.6f} | {r['dec_throughput']:9.2f} | "
          f"{r['rss_peak']:12.2f}")
```

ΠΡΟΓΡΑΜΜΑ 5 ΣΕ C

```
#include <stdio.h>
```

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/rand.h>
#include <openssl/ec.h>
#define KEY_LEN 16       // 128-bit key
#define IV_LEN 12        // 96-bit IV for AES-GCM
#define TAG_LEN 16       // 128-bit tag
void handle_errors(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}
// wall-clock διαφορά σε δευτερόλεπτα
double timespec_diff_sec(struct timespec start, struct timespec end) {
    double sec  = (double)(end.tv_sec - start.tv_sec);
    double nsec = (double)(end.tv_nsec - start.tv_nsec) / 1e9;
    return sec + nsec;
}
// ανάγνωση τρέχουσας χρήσης μνήμης (VmRSS) σε KB
long get_memory_usage_kb(void) {
    FILE *f = fopen("/proc/self/status", "r");
```

```
    if (!f) return -1;
    char line[256];
    long mem_kb = -1;
    while (fgets(line, sizeof(line), f)) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            // μορφή γραμμής: "VmRSS:   12345 kB"
            sscanf(line, "VmRSS: %ld kB", &mem_kb);
            break;
        }
    }
    fclose(f);
    return mem_kb;
}
// ------------------------- ECDH: derive 128-bit key -----------------------------
void ecdh_derive_key(uint8_t *out_key, size_t out_len) {
    EVP_PKEY_CTX *pctx = NULL;
    EVP_PKEY_CTX *kctx = NULL;
    EVP_PKEY_CTX *dctx = NULL;
    EVP_PKEY *params = NULL;
    EVP_PKEY *keyA = NULL;
    EVP_PKEY *keyB = NULL;
    if (out_len < KEY_LEN) {
        fprintf(stderr, "Output buffer too small for key\n");
        exit(EXIT_FAILURE);
    }
```

```
// EC params για καμπύλη P-256
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
if (!pctx)
   handle_errors("EVP_PKEY_CTX_new_id");
if (EVP_PKEY_paramgen_init(pctx) <= 0)
   handle_errors("EVP_PKEY_paramgen_init");
if (EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, NID_X9_62_prime256v1) <= 0)
   handle_errors("Set curve nid");
if (EVP_PKEY_paramgen(pctx, &params) <= 0)
   handle_errors("EVP_PKEY_paramgen");
// Keypair A
kctx = EVP_PKEY_CTX_new(params, NULL);
if (!kctx)
   handle_errors("EVP_PKEY_CTX_new A");
if (EVP_PKEY_keygen_init(kctx) <= 0)
   handle_errors("EVP_PKEY_keygen_init A");
if (EVP_PKEY_keygen(kctx, &keyA) <= 0)
   handle_errors("EVP_PKEY_keygen A");
EVP_PKEY_CTX_free(kctx);
kctx = NULL;
// Keypair B
kctx = EVP_PKEY_CTX_new(params, NULL);
if (!kctx)
   handle_errors("EVP_PKEY_CTX_new B");
```

```
if (EVP_PKEY_keygen_init(kctx) <= 0)
    handle_errors("EVP_PKEY_keygen_init B");
if (EVP_PKEY_keygen(kctx, &keyB) <= 0)
    handle_errors("EVP_PKEY_keygen B");
// ECDH derive (A με public B)
dctx = EVP_PKEY_CTX_new(keyA, NULL);
if (!dctx)
    handle_errors("EVP_PKEY_CTX_new derive");
if (EVP_PKEY_derive_init(dctx) <= 0)
    handle_errors("EVP_PKEY_derive_init");
if (EVP_PKEY_derive_set_peer(dctx, keyB) <= 0)
    handle_errors("EVP_PKEY_derive_set_peer");
size_t secret_len = 0;
if (EVP_PKEY_derive(dctx, NULL, &secret_len) <= 0)
    handle_errors("EVP_PKEY_derive length");
uint8_t *secret = malloc(secret_len);
if (!secret)
    handle_errors("malloc secret");
if (EVP_PKEY_derive(dctx, secret, &secret_len) <= 0)
    handle_errors("EVP_PKEY_derive");
if (secret_len < KEY_LEN)
    handle_errors("Shared secret too small");
// για το πείραμα: παίρνουμε τα πρώτα 16 bytes ως κλειδί
memcpy(out_key, secret, KEY_LEN);
free(secret);
```

```
    EVP_PKEY_free(keyA);
    EVP_PKEY_free(keyB);
    EVP_PKEY_free(params);
    EVP_PKEY_CTX_free(pctx);
    EVP_PKEY_CTX_free(kctx);
    EVP_PKEY_CTX_free(dctx);
}
// ----------------------------- AES-GCM Encrypt / Decrypt -------------------------------
int aes_gcm_encrypt(const uint8_t *plaintext, int plaintext_len,
                    const uint8_t *aad, int aad_len,
                    const uint8_t *key,
                    const uint8_t *iv, int iv_len,
                    uint8_t *ciphertext,
                    uint8_t *tag) {
    EVP_CIPHER_CTX *ctx = NULL;
    int len = 0;
    int ciphertext_len = 0;
    int ret = 0;
    ctx = EVP_CIPHER_CTX_new();
    if (!ctx)
        handle_errors("EVP_CIPHER_CTX_new");
    if (EVP_EncryptInit_ex(ctx, EVP_aes_128_gcm(), NULL, NULL, NULL) != 1)
        goto err;
    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL) != 1)
        goto err;
```

```
    if (EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv) != 1)
        goto err;
    if (aad && aad_len > 0) {
        if (EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len) != 1)
            goto err;
    }
    if (EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len) != 1)
        goto err;
    ciphertext_len = len;
    if (EVP_EncryptFinal_ex(ctx, ciphertext + len, &len) != 1)
        goto err;
    ciphertext_len += len;
    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, TAG_LEN, tag) != 1)
        goto err;
    ret = ciphertext_len;
err:
    if (ret == 0)
        handle_errors("AES-GCM encrypt failed");
    EVP_CIPHER_CTX_free(ctx);
    return ret;
}
int aes_gcm_decrypt(const uint8_t *ciphertext, int ciphertext_len,
                    const uint8_t *aad, int aad_len,
                    const uint8_t *tag,
                    const uint8_t *key,
```

```
                const uint8_t *iv, int iv_len,
                uint8_t *plaintext) {
    EVP_CIPHER_CTX *ctx = NULL;
    int len = 0;
    int plaintext_len = 0;
    int ret = -1;
    ctx = EVP_CIPHER_CTX_new();
    if (!ctx)
        handle_errors("EVP_CIPHER_CTX_new");
    if (EVP_DecryptInit_ex(ctx, EVP_aes_128_gcm(), NULL, NULL, NULL) != 1)
        goto end;
    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL) != 1)
        goto end;
    if (EVP_DecryptInit_ex(ctx, NULL, NULL, key, iv) != 1)
        goto end;
    if (aad && aad_len > 0) {
        if (EVP_DecryptUpdate(ctx, NULL, &len, aad, aad_len) != 1)
            goto end;
    }
    if (EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len) != 1)
        goto end;
    plaintext_len = len;
    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, TAG_LEN, (void *)tag) !=
1)
        goto end;
```

```
    if (EVP_DecryptFinal_ex(ctx, plaintext + len, &len) != 1) {
        fprintf(stderr, "AES-GCM tag verification failed!\n");
        plaintext_len = -1;
        goto end;
    }
    plaintext_len += len;
    ret = plaintext_len;
end:
    EVP_CIPHER_CTX_free(ctx);
    return ret;
}
// ------------------------------------ main ------------------------------------------
int main(void) {
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();
    // μεγέθη δεδομένων
    size_t sizes[] = {100 * 1024, 1 * 1024 * 1024, 10 * 1024 * 1024, 50 * 1024 * 1024};
    size_t num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    uint8_t session_key[KEY_LEN];
    struct timespec t_start, t_end;
    clock_t cpu_start, cpu_end;
    // ECDH (μετράει μόνο wall-clock )
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    ecdh_derive_key(session_key, KEY_LEN);
    clock_gettime(CLOCK_MONOTONIC, &t_end);
```

```
double ecdh_time = timespec_diff_sec(t_start, t_end);
printf("ECDH time (AES-GCM experiment): %.6f s\n\n", ecdh_time);
uint8_t aad[16];
RAND_bytes(aad, sizeof(aad));
for (size_t i = 0; i < num_sizes; i++) {
    size_t data_len = sizes[i];
    printf("=== AES-GCM, size: %zu bytes ===\n", data_len);
    uint8_t *plaintext  = malloc(data_len);
    uint8_t *ciphertext = malloc(data_len);
    uint8_t *decrypted  = malloc(data_len);
    if (!plaintext || !ciphertext || !decrypted)
        handle_errors("malloc");
    RAND_bytes(plaintext, data_len);
    uint8_t iv[IV_LEN];
    uint8_t tag[TAG_LEN];
    RAND_bytes(iv, sizeof(iv));
    long mem_before = get_memory_usage_kb();
    // ---- Encrypt ----
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    cpu_start = clock();
    int clen = aes_gcm_encrypt(plaintext, (int)data_len,
                               aad, sizeof(aad),
                               session_key,
                               iv, IV_LEN,
                               ciphertext,
```

```
                    tag);
cpu_end = clock();
clock_gettime(CLOCK_MONOTONIC, &t_end);
double enc_wall = timespec_diff_sec(t_start, t_end);
double enc_cpu  = (double)(cpu_end - cpu_start) / CLOCKS_PER_SEC;
// ---- Decrypt ----
clock_gettime(CLOCK_MONOTONIC, &t_start);
cpu_start = clock();

int plen = aes_gcm_decrypt(ciphertext, clen,
                    aad, sizeof(aad),
                    tag,
                    session_key,
                    iv, IV_LEN,
                    decrypted);
cpu_end = clock();
clock_gettime(CLOCK_MONOTONIC, &t_end);
double dec_wall = timespec_diff_sec(t_start, t_end);
double dec_cpu  = (double)(cpu_end - cpu_start) / CLOCKS_PER_SEC;
long mem_after = get_memory_usage_kb();
if (plen != (int)data_len || memcmp(plaintext, decrypted, data_len) != 0) {
   fprintf(stderr, "AES-GCM verification failed!\n");
} else {
   printf("Encrypt: wall = %.6f s, CPU = %.6f s\n",
         enc_wall, enc_cpu);
```

```
            printf("Decrypt: wall = %.6f s, CPU = %.6f s\n",
                dec_wall, dec_cpu);
            printf("Memory usage: before = %ld KB, after = %ld KB\n\n",
                mem_before, mem_after);
        }
        free(plaintext);
        free(ciphertext);
        free(decrypted);
    }
    EVP_cleanup();
    ERR_free_strings();
    return 0;
}
```

Για compile το ecdh_aes_gcm.c:

```
gcc ecdh_aes_gcm.c -o ecdh_aes_gcm \
   -Wall -Wextra -O2 \
   -lcrypto -lssl
```

Για να τρέξει:
./ecdh_aes_gcm

ΠΡΟΓΡΑΜΜΑ 6 ΣΕ C

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/ec.h>
```

```
#include <openssl/rand.h>

#define KEY_LEN 16   // 128-bit key

// ----------------- helpers: errors, time diff, memory usage -------------------

void handle_errors(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

// wall-clock διαφορά σε δευτερόλεπτα
double timespec_diff_sec(struct timespec start, struct timespec end) {
    double sec  = (double)(end.tv_sec - start.tv_sec);
    double nsec = (double)(end.tv_nsec - start.tv_nsec) / 1e9;
    return sec + nsec;
}

// ανάγνωση τρέχουσας χρήσης μνήμης (VmRSS) σε KB
long get_memory_usage_kb(void) {
    FILE *f = fopen("/proc/self/status", "r");
    if (!f) return -1;
    char line[256];
    long mem_kb = -1;
    while (fgets(line, sizeof(line), f)) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            sscanf(line, "VmRSS: %ld kB", &mem_kb);
            break;
        }
    }
    fclose(f);
    return mem_kb;
}

// ------------------------- ECDH: derive 128-bit key -------------------------

void ecdh_derive_key(uint8_t *out_key, size_t out_len) {
    EVP_PKEY_CTX *pctx = NULL;
    EVP_PKEY_CTX *kctx = NULL;
    EVP_PKEY_CTX *dctx = NULL;
    EVP_PKEY *params = NULL;
    EVP_PKEY *keyA = NULL;
    EVP_PKEY *keyB = NULL;

    if (out_len < KEY_LEN) {
        fprintf(stderr, "Output buffer too small for key\n");
        exit(EXIT_FAILURE);
```

```
}

// EC params για καμπύλη P-256
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
if (!pctx)
    handle_errors("EVP_PKEY_CTX_new_id");
if (EVP_PKEY_paramgen_init(pctx) <= 0)
    handle_errors("EVP_PKEY_paramgen_init");
if (EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, NID_X9_62_prime256v1) <= 0)
    handle_errors("Set curve nid");
if (EVP_PKEY_paramgen(pctx, &params) <= 0)
    handle_errors("EVP_PKEY_paramgen");

// Keypair A
kctx = EVP_PKEY_CTX_new(params, NULL);
if (!kctx)
    handle_errors("EVP_PKEY_CTX_new A");
if (EVP_PKEY_keygen_init(kctx) <= 0)
    handle_errors("EVP_PKEY_keygen_init A");
if (EVP_PKEY_keygen(kctx, &keyA) <= 0)
    handle_errors("EVP_PKEY_keygen A");
EVP_PKEY_CTX_free(kctx);
kctx = NULL;

// Keypair B
kctx = EVP_PKEY_CTX_new(params, NULL);
if (!kctx)
    handle_errors("EVP_PKEY_CTX_new B");
if (EVP_PKEY_keygen_init(kctx) <= 0)
    handle_errors("EVP_PKEY_keygen_init B");
if (EVP_PKEY_keygen(kctx, &keyB) <= 0)
    handle_errors("EVP_PKEY_keygen B");

// ECDH derive (A με public B)
dctx = EVP_PKEY_CTX_new(keyA, NULL);
if (!dctx)
    handle_errors("EVP_PKEY_CTX_new derive");
if (EVP_PKEY_derive_init(dctx) <= 0)
    handle_errors("EVP_PKEY_derive_init");
if (EVP_PKEY_derive_set_peer(dctx, keyB) <= 0)
    handle_errors("EVP_PKEY_derive_set_peer");

size_t secret_len = 0;
if (EVP_PKEY_derive(dctx, NULL, &secret_len) <= 0)
    handle_errors("EVP_PKEY_derive length");

uint8_t *secret = malloc(secret_len);
if (!secret)
```

```
        handle_errors("malloc secret");
    if (EVP_PKEY_derive(dctx, secret, &secret_len) <= 0)
        handle_errors("EVP_PKEY_derive");

    if (secret_len < KEY_LEN)
        handle_errors("Shared secret too small");

    // για το πείραμα: παίρνουμε τα πρώτα 16 bytes ως κλειδί
    memcpy(out_key, secret, KEY_LEN);

    free(secret);
    EVP_PKEY_free(keyA);
    EVP_PKEY_free(keyB);
    EVP_PKEY_free(params);
    EVP_PKEY_CTX_free(pctx);
    EVP_PKEY_CTX_free(kctx);
    EVP_PKEY_CTX_free(dctx);
}

// ------------------------- Ascon-128a implementation ------------------------

typedef struct {
    uint64_t x0, x1, x2, x3, x4;
} ascon_state_t;

// x86_64 optimized rotation macros
#define ROR64(x, n) (((x) >> (n)) | ((x) << (64 - (n))))

// permutation
static inline __attribute__((always_inline))
void ascon_permutation(ascon_state_t *s, int rounds) {
    uint64_t t0, t1, t2, t3, t4;

    for (int i = 0; i < rounds; i++) {
        // Addition of round constant
        s->x2 ^= ((0x0F - i) << 4) | i;

        // Substitution layer
        s->x0 ^= s->x4;
        s->x4 ^= s->x3;
        s->x2 ^= s->x1;

        t0 = s->x0;
        t1 = s->x1;
        t2 = s->x2;
        t3 = s->x3;
        t4 = s->x4;
```

```
        // Non-linear layer
        s->x0 = t0 ^ (~t1 & t2);
        s->x1 = t1 ^ (~t2 & t3);
        s->x2 = t2 ^ (~t3 & t4);
        s->x3 = t3 ^ (~t4 & t0);
        s->x4 = t4 ^ (~t0 & t1);

        // Linear layer additions
        s->x1 ^= s->x0;
        s->x0 ^= s->x4;
        s->x3 ^= s->x2;
        s->x2 = ~s->x2;

        // Linear diffusion layer
        s->x0 ^= ROR64(s->x0, 19) ^ ROR64(s->x0, 28);
        s->x1 ^= ROR64(s->x1, 61) ^ ROR64(s->x1, 39);
        s->x2 ^= ROR64(s->x2, 1)  ^ ROR64(s->x2, 6);
        s->x3 ^= ROR64(s->x3, 10) ^ ROR64(s->x3, 17);
        s->x4 ^= ROR64(s->x4, 7)  ^ ROR64(s->x4, 41);
    }
}

// memory load/store
static inline uint64_t load64(const uint8_t *bytes) {
    uint64_t value;
    memcpy(&value, bytes, 8);
    return value;
}

static inline void store64(uint8_t *bytes, uint64_t value) {
    memcpy(bytes, &value, 8);
}

// Initialize state for Ascon-128a
static void ascon128a_init(ascon_state_t *s, const uint8_t *key, const uint8_t *nonce) {
    uint64_t k0 = load64(key);
    uint64_t k1 = load64(key + 8);
    uint64_t n0 = load64(nonce);
    uint64_t n1 = load64(nonce + 8);

    s->x0 = 0x80800c0800000000ULL;
    s->x1 = k0;
    s->x2 = k1;
    s->x3 = n0;
    s->x4 = n1;

    ascon_permutation(s, 12);
```

```
    s->x3 ^= k0;
    s->x4 ^= k1;
}

// Process associated data
static void ascon128a_process_associated_data(ascon_state_t *s,
                                              const uint8_t *ad,
                                              size_t ad_len) {
    if (ad_len > 0) {
        size_t i = 0;

        while (i + 16 <= ad_len) {
            uint64_t ad0 = load64(ad + i);
            uint64_t ad1 = load64(ad + i + 8);

            s->x0 ^= ad0;
            s->x1 ^= ad1;
            ascon_permutation(s, 8);

            i += 16;
        }

        if (i < ad_len) {
            uint64_t ad0 = 0, ad1 = 0;
            size_t remaining = ad_len - i;

            memcpy(&ad0, ad + i, (remaining > 8) ? 8 : remaining);
            if (remaining > 8) {
                memcpy(&ad1, ad + i + 8, remaining - 8);
            }

            if (remaining < 8) {
                ad0 |= 0x80ULL << (56 - (remaining * 8));
            } else if (remaining < 16) {
                ad1 |= 0x80ULL << (56 - ((remaining - 8) * 8));
            }

            s->x0 ^= ad0;
            s->x1 ^= ad1;
            ascon_permutation(s, 8);
        }
    }

    s->x4 ^= 1;
}

// Encrypt blocks
static void ascon128a_encrypt_blocks(ascon_state_t *s,
```

```
                            uint8_t *ciphertext,
                            const uint8_t *plaintext,
                            size_t plaintext_len) {
    size_t i = 0;

    while (i + 16 <= plaintext_len) {
        uint64_t p0 = load64(plaintext + i);
        uint64_t p1 = load64(plaintext + i + 8);

        s->x0 ^= p0;
        s->x1 ^= p1;

        store64(ciphertext + i, s->x0);
        store64(ciphertext + i + 8, s->x1);

        i += 16;

        if (i < plaintext_len) {
            ascon_permutation(s, 8);
        }
    }

    if (i < plaintext_len) {
        uint64_t p0 = 0, p1 = 0;
        size_t remaining = plaintext_len - i;

        memcpy(&p0, plaintext + i, (remaining > 8) ? 8 : remaining);
        if (remaining > 8) {
            memcpy(&p1, plaintext + i + 8, remaining - 8);
        }

        s->x0 ^= p0;
        s->x1 ^= p1;

        memcpy(ciphertext + i, &s->x0, (remaining > 8) ? 8 : remaining);
        if (remaining > 8) {
            memcpy(ciphertext + i + 8, &s->x1, remaining - 8);
        }

        if (remaining < 8) {
            s->x0 ^= 0x80ULL << (56 - (remaining * 8));
        } else if (remaining < 16) {
            s->x1 ^= 0x80ULL << (56 - ((remaining - 8) * 8));
        }
    }
}

// Finalize and generate tag
```

```
static void ascon128a_finalize(ascon_state_t *s, const uint8_t *key, uint8_t *tag) {
    uint64_t k0 = load64(key);
    uint64_t k1 = load64(key + 8);

    s->x2 ^= k0;
    s->x3 ^= k1;

    ascon_permutation(s, 12);

    s->x3 ^= k0;
    s->x4 ^= k1;

    store64(tag, s->x3);
    store64(tag + 8, s->x4);
}

// AEAD encrypt
int ascon128a_aead_encrypt(uint8_t* ciphertext, size_t* ciphertext_len,
                           const uint8_t* plaintext, size_t plaintext_len,
                           const uint8_t* associated_data, size_t ad_len,
                           const uint8_t* nonce, const uint8_t* key) {
    ascon_state_t s;

    ascon128a_init(&s, key, nonce);
    ascon128a_process_associated_data(&s, associated_data, ad_len);
    ascon128a_encrypt_blocks(&s, ciphertext, plaintext, plaintext_len);

    uint8_t tag[16];
    ascon128a_finalize(&s, key, tag);
    memcpy(ciphertext + plaintext_len, tag, 16);

    *ciphertext_len = plaintext_len + 16;
    return 0;
}

// AEAD decrypt με αποκρυπτογράφηση + έλεγχο tag
int ascon128a_aead_decrypt(uint8_t* plaintext, size_t* plaintext_len,
                           const uint8_t* ciphertext, size_t ciphertext_len,
                           const uint8_t* associated_data, size_t ad_len,
                           const uint8_t* nonce, const uint8_t* key) {
    if (ciphertext_len < 16) return -1;

    size_t data_len = ciphertext_len - 16;   // χωρίς το tag
    const uint8_t *tag_in = ciphertext + data_len;

    ascon_state_t s;
    ascon128a_init(&s, key, nonce);
    ascon128a_process_associated_data(&s, associated_data, ad_len);
```

```
size_t i = 0;

// full blocks
while (i + 16 <= data_len) {
    uint64_t c0 = load64(ciphertext + i);
    uint64_t c1 = load64(ciphertext + i + 8);

    uint64_t p0 = s.x0 ^ c0;
    uint64_t p1 = s.x1 ^ c1;

    store64(plaintext + i, p0);
    store64(plaintext + i + 8, p1);

    s.x0 = c0;
    s.x1 = c1;

    i += 16;

    if (i < data_len) {
        ascon_permutation(&s, 8);
    }
}

// partial block
if (i < data_len) {
    uint64_t c0 = 0, c1 = 0;
    size_t remaining = data_len - i;

    memcpy(&c0, ciphertext + i, (remaining > 8) ? 8 : remaining);
    if (remaining > 8) {
        memcpy(&c1, ciphertext + i + 8, remaining - 8);
    }

    uint64_t p0 = s.x0 ^ c0;
    uint64_t p1 = s.x1 ^ c1;

    memcpy(plaintext + i, &p0, (remaining > 8) ? 8 : remaining);
    if (remaining > 8) {
        memcpy(plaintext + i + 8, &p1, remaining - 8);
    }

    s.x0 = c0;
    s.x1 = c1;

    if (remaining < 8) {
        s.x0 ^= 0x80ULL << (56 - (remaining * 8));
    } else if (remaining < 16) {
```

```
            s.x1 ^= 0x80ULL << (56 - ((remaining - 8) * 8));
        }
    }

    // υπολογισμός tag και σύγκριση
    uint8_t tag_calc[16];
    ascon128a_finalize(&s, key, tag_calc);

    unsigned diff = 0;
    for (int j = 0; j < 16; j++) {
        diff |= (unsigned)(tag_calc[j] ^ tag_in[j]);
    }

    if (diff != 0) {
        // tag mismatch
        return -1;
    }

    *plaintext_len = data_len;
    return 0;
}

// ------------------------------ main: benchmark ------------------------------

int main(void) {
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    // ίδια μεγέθη με το AES-GCM πρόγραμμα
    size_t sizes[] = {100 * 1024, 1 * 1024 * 1024, 10 * 1024 * 1024, 50 * 1024 * 1024};
    size_t num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    uint8_t session_key[KEY_LEN];
    struct timespec t_start, t_end;
    clock_t cpu_start, cpu_end;

    // ECDH (μετράει μόνο wall-clock)
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    ecdh_derive_key(session_key, KEY_LEN);
    clock_gettime(CLOCK_MONOTONIC, &t_end);
    double ecdh_time = timespec_diff_sec(t_start, t_end);
    printf("ECDH time (Ascon-128a experiment): %.6f s\n\n", ecdh_time);

    // AAD 16 bytes
    uint8_t aad[16];
    RAND_bytes(aad, sizeof(aad));

    for (size_t i = 0; i < num_sizes; i++) {
```

```
size_t data_len = sizes[i];
printf("=== Ascon-128a, size: %zu bytes ===\n", data_len);

uint8_t *plaintext  = malloc(data_len);
uint8_t *ciphertext = malloc(data_len + 16); // + tag
uint8_t *decrypted  = malloc(data_len);
if (!plaintext || !ciphertext || !decrypted)
    handle_errors("malloc");

// τυχαία δεδομένα
RAND_bytes(plaintext, data_len);

uint8_t nonce[16];
RAND_bytes(nonce, sizeof(nonce));

long mem_before = get_memory_usage_kb();

// ---- Encrypt ----
size_t clen = 0;
clock_gettime(CLOCK_MONOTONIC, &t_start);
cpu_start = clock();

int enc_res = ascon128a_aead_encrypt(ciphertext, &clen,
                                plaintext, data_len,
                                aad, sizeof(aad),
                                nonce, session_key);

cpu_end = clock();
clock_gettime(CLOCK_MONOTONIC, &t_end);

double enc_wall = timespec_diff_sec(t_start, t_end);
double enc_cpu  = (double)(cpu_end - cpu_start) / CLOCKS_PER_SEC;

// ---- Decrypt ----
size_t plen = 0;
clock_gettime(CLOCK_MONOTONIC, &t_start);
cpu_start = clock();

int dec_res = ascon128a_aead_decrypt(decrypted, &plen,
                                ciphertext, clen,
                                aad, sizeof(aad),
                                nonce, session_key);

cpu_end = clock();
clock_gettime(CLOCK_MONOTONIC, &t_end);

double dec_wall = timespec_diff_sec(t_start, t_end);
double dec_cpu  = (double)(cpu_end - cpu_start) / CLOCKS_PER_SEC;
```

```
        long mem_after = get_memory_usage_kb();

        if (enc_res != 0 || dec_res != 0 || plen != data_len ||
            memcmp(plaintext, decrypted, data_len) != 0) {
            fprintf(stderr, "Ascon-128a verification failed!\n");
        } else {
            printf("Encrypt: wall = %.6f s, CPU = %.6f s\n",
                   enc_wall, enc_cpu);
            printf("Decrypt: wall = %.6f s, CPU = %.6f s\n",
                   dec_wall, dec_cpu);
            printf("Memory usage: before = %ld KB, after = %ld KB\n\n",
                   mem_before, mem_after);
        }

        free(plaintext);
        free(ciphertext);
        free(decrypted);
    }

    EVP_cleanup();
    ERR_free_strings();
    return 0;
}
```