



Universitatea Tehnică din Cluj-Napoca

Facultatea de Automatică și Calculatoare

Structura sistemelor de calcul

Sistem de autentificare bazat pe criptografie asimetrică (RSA)

Student: Semeniuc Giulia-Carla

Grupa: Grupa 30235

Îndrumător: Sopterean Andrei Mihai

Cuprins

1 Rezumat	3
2 Introducere	4
2.1 Contextul temei și tendințe tehnologice	4
2.2 Domeniul de studiu și terminologia de bază	4
2.3 Limitările soluțiilor tradiționale de autentificare	5
2.4 Problema de rezolvat și obiectivele proiectului	5
2.5 Soluția propusă	5
2.6 Structura raportului	6
3 Fundamentare teoretică	6
3.1 Criptografia asimetrică și autentificarea	6
3.2 Algoritmul RSA — fundamente matematice	6
3.3 Semnătura digitală RSA	7
3.4 Autentificarea challenge-response	7
3.5 Soluții existente și poziționarea proiectului	8
4 Proiectare și implementare	8
4.1 Metoda experimentală utilizată	8
4.2 Alternative de proiectare și soluția aleasă	9
4.3 Arhitectura generală a sistemului	9
4.4 Descrierea modulelor software	10
4.5 Algoritmii implementați	10
4.6 Detalii de implementare și comunicație	11
4.7 Manual de utilizare	11
5 Rezultate experimentale	11
5.1 Instrumente de proiectare utilizate	11
5.2 Implementarea sistemului și resurse utilizate	12
5.3 Procedura de testare	12
5.4 Rezultate obținute	13
5.5 Analiza și interpretarea rezultatelor	13
5.6 Dificultăți întâlnite și soluții adoptate	13
6 Concluzii	13
Bibliografie	14
Anexa A: Cod client PC (Python)	15

Anexa B: Protocol de autentificare	21
Anexa C: Implementare RSA	24
Anexa D: Comunicație UART	26
Anexa E: Aplicație embedded	27

1 Rezumat

Autentificarea sigură reprezintă o cerință esențială în sistemele embedded moderne, unde accesul neautorizat poate conduce la compromiterea funcționalității sau a datelor procesate. Problema abordată a fost implementarea unui mecanism de autentificare sigur, care să eliminate utilizarea parolelor clasice și transmiterea informațiilor sensibile în clar, asigurând totodată protecția împotriva atacurilor de tip replay. Obiectivele proiectului au inclus definirea unui protocol de autentificare de tip challenge-response, implementarea acestuia pe o platformă embedded și dezvoltarea unui client PC capabil să realizeze operațiile criptografice necesare. Metoda de rezolvare a constat în utilizarea semnăturilor digitale RSA, implementarea aplicației embedded în limbajul C, dezvoltarea clientului PC în Python și utilizarea comunicației seriale UART pentru schimbul de mesaje. Rezultatele obținute au arătat că sistemul a permis autentificarea corectă în cazul cheilor valide și a respins toate tentativele neautorizate. În concluzie, soluția implementată a demonstrat că autentificarea bazată pe criptografie asimetrică este adecvată pentru sisteme embedded cu cerințe de securitate ridicate.

2 Introducere

2.1 Contextul temei și tendințe tehnologice

Dezvoltarea accelerată a sistemelor embedded conectate la rețea a condus la o creștere semnificativă a cerințelor de securitate. Dispozitivele utilizate în aplicații precum Internet of Things, automatizări industriale, sisteme de monitorizare, echipamente medicale sau infrastructuri critice sunt frecvent expuse la rețele nesigure și pot deveni ţinte ale accesului neautorizat. În multe dintre aceste aplicații, compromiterea unui singur dispozitiv poate avea consecințe severe asupra întregului sistem.

Tendințele tehnologice actuale indică o tranziție de la sisteme izolate către arhitecturi distribuite, în care dispozitivele embedded comunică frecvent cu alte noduri sau cu sisteme centrale. Această creștere a interconectivității amplifică suprafața de atac și face necesară utilizarea unor mecanisme de securitate avansate. Autentificarea entităților participante la comunicare reprezintă una dintre primele bariere împotriva accesului neautorizat.

În ultimii ani, soluțiile de securitate bazate exclusiv pe mecanisme simple, precum parolele statice sau identificatorii fixați la nivel hardware, au devenit insuficiente. Protocolele moderne de securitate utilizează mecanisme criptografice care oferă garanții mai puternice privind identitatea, integritatea și autenticitatea entităților implicate. În acest context, criptografia asimetrică și semnăturile digitale sunt utilizate pe scară largă în protocoale standardizate, precum TLS sau mecanismele de autentificare bazate pe certificate digitale [1].

2.2 Domeniul de studiu și terminologia de bază

Domeniul de studiu al acestui proiect este securitatea sistemelor embedded, cu accent pe mecanismele de autentificare criptografică. Autentificarea reprezintă procesul prin care o entitate își dovedește identitatea în fața unui sistem, fiind un pas distinct de autorizare, care stabilește drepturile de acces ulterior autentificării.

Criptografia asimetrică utilizează o pereche de chei criptografice: o cheie publică, care poate fi distribuită liber, și o cheie privată, care trebuie protejată împotriva accesului neautorizat. Această separare permite realizarea unor mecanisme de securitate în care informațiile sensibile nu sunt transmise direct prin canalul de comunicație.

Algoritmul RSA este unul dintre cei mai cunoscuți algoritmi de criptografie asimetrică și este utilizat atât pentru criptare, cât și pentru realizarea semnăturilor digitale. Semnătura digitală reprezintă un mecanism prin care o entitate poate demonstra autenticitatea unui mesaj și posesia unei chei private, fără a divulga această cheie. Aceste proprietăți fac ca RSA să fie potrivit pentru implementarea mecanismelor de autentificare în sisteme cu cerințe sporite de securitate [2].

2.3 Limitările soluțiilor tradiționale de autentificare

Multe sisteme embedded utilizează în continuare mecanisme simple de autentificare, bazate pe parole statice, coduri PIN sau identificatori fixați în firmware. Aceste soluții prezintă limitări importante din punct de vedere al securității, întrucât informațiile de autentificare pot fi interceptate, reutilizate sau extrase prin acces fizic sau logic la dispozitiv.

În plus, gestionarea securizată a parolelor este dificilă în medii embedded cu resurse limitate. Lipsa unor mecanisme avansate de stocare securizată sau a unor canale de comunicație criptate poate conduce la expunerea credențialelor. De asemenea, parolele statice sunt vulnerabile la atacuri de tip replay, în care un mesaj de autentificare valid este capturat și retransmis ulterior pentru a obține acces neautorizat.

Aceste limitări evidențiază necesitatea unor mecanisme de autentificare mai robuste, care să nu se bazeze pe secrete statice și să ofere protecție împotriva reutilizării mesajelor valide.

2.4 Problema de rezolvat și obiectivele proiectului

Problema principală abordată în cadrul acestui proiect este realizarea unui mecanism de autentificare sigur pentru un sistem embedded, care să fie rezistent la atacuri comune și să funcționeze eficient în condițiile unor resurse hardware limitate. Sistemul trebuie să permită verificarea identității unui client fără a transmite informații sensibile și fără a se baza pe parole statice.

Obiectivele principale ale proiectului au fost:

- proiectarea unui mecanism de autentificare bazat pe criptografie asimetrică RSA;
- definirea și implementarea unui protocol de tip challenge-response pentru prevenirea atacurilor de tip replay;
- realizarea unei aplicații embedded capabile să verifice identitatea unui client;
- dezvoltarea unui client PC care să gestioneze operațiile criptografice și comunicarea cu sistemul embedded;
- evaluarea funcțională și de securitate a soluției implementate.

2.5 Soluția propusă

Soluția propusă se bazează pe utilizarea semnăturilor digitale RSA pentru autentificare. În cadrul mecanismului implementat, sistemul embedded generează o provocare aleatorie (nonce), care este transmisă clientului. Clientul semnează această provocare utilizând cheia privată RSA și transmite semnatura înapoi către sistemul embedded.

Semnătura este verificată cu ajutorul cheii publice corespunzătoare, permitând validarea identității clientului. Această abordare elimină necesitatea transmiterii parolelor și asigură faptul că doar entitatea care deține cheia privată poate fi autentificată. Utilizarea unui nonce unic pentru fiecare sesiune previne reutilizarea mesajelor de autentificare și oferă protecție împotriva atacurilor de tip replay.

Comparativ cu soluțiile traditionale, mecanismul propus oferă un nivel superior de securitate și este compatibil cu extinderi ulterioare, precum integrarea certificatelor digitale sau utilizarea unor protocoale standardizate de securitate [3].

2.6 Structura raportului

Raportul este structurat astfel: Secțiunea 2 prezintă contextul general al temei, definește problema abordată și descrie soluția propusă. Secțiunea 3 introduce noțiunile teoretice necesare înțelegерii criptografiei asimetrice și a mecanismelor de autentificare utilizate. Secțiunea 4 detaliază proiectarea și implementarea soluției, incluzând arhitectura sistemului și protocolul de comunicație. Secțiunea 5 prezintă rezultatele experimentale și analiza acestora, iar Secțiunea 6 conține concluziile și direcțiile de dezvoltare viitoare.

3 Fundamentare teoretică

3.1 Criptografia asimetrică și autentificarea

Criptografia asimetrică se bazează pe utilizarea unei perechi de chei: o cheie publică și o cheie privată. Cheia publică poate fi distribuită liber, în timp ce cheia privată este păstrată secretă, astfel încât o entitate poate demonstra posesia cheii private fără a o divulga. Algoritmii asimetrici sunt folosiți pe scară largă în autentificare, semnături digitale și schimb de chei, oferind un nivel superior de securitate comparativ cu metodele bazate pe parole statice sau chei partajate [1][3].

Autentificarea reprezintă procesul prin care un sistem verifică identitatea entităților participante. În sistemele moderne, aceasta poate include metode bazate pe criptografie asimetrică, în care posesia unei chei private valide este dovada identității. Astfel, autentificarea asimetrică evită transmiterea informațiilor sensibile prin canalul de comunicație și oferă rezistență la diverse atacuri, inclusiv la captarea pasivă a traficului [1][7].

3.2 Algoritmul RSA — fundamente matematice

Algoritmul RSA, denumit după Ronald Rivest, Adi Shamir și Leonard Adleman, reprezintă unul dintre cei mai cunoscuți algoritmi de criptografie asimetrică și este utilizat în practică atât pentru criptare, cât și pentru semnături digitale [3]. Generarea cheilor RSA pornește

de la alegerea a două numere prime mari p și q și calcularea modulului:

$$n = p \cdot q.$$

Funcția lui Euler $\varphi(n)$ este definită ca:

$$\varphi(n) = (p - 1)(q - 1).$$

Se alege un exponent public e astfel încât:

$$\gcd(e, \varphi(n)) = 1,$$

iar exponentul privat d este calculat ca invers modular al lui e modulo $\varphi(n)$:

$$e \cdot d \equiv 1 \pmod{\varphi(n)}.$$

Cheia publică este perechea (n, e) , iar cheia privată este d . Securitatea teoretică a RSA se bazează pe dificultatea factorării lui n atunci când p și q sunt mari și generate corespunzător [3][20].

3.3 Semnătura digitală RSA

Pentru autentificare se utilizează, de regulă, semnăturile digitale RSA. În loc să semneze direct un mesaj M , se calculează mai întâi un hash $H = \text{hash}(M)$, unde $\text{hash}(\cdot)$ este o funcție criptografică de tipul SHA-256 sau similar. Semnătura S este calculată aplicând cheia privată asupra hash-ului:

$$S \equiv H^d \pmod{n}.$$

Verificarea semnăturii se realizează cu cheia publică:

$$H' \equiv S^e \pmod{n},$$

și dacă $H' = H$, semnătura este considerată validă. Această proprietate asigură atât autenticitatea entității semnatari, cât și integritatea datelor semnate [3].

3.4 Autentificarea challenge-response

Mecanismele de autentificare de tip challenge-response sunt utilizate pentru a preveni atacurile de tip replay, în care un mesaj de autentificare valid este reutilizat ulterior de un atacator pentru a obține acces neautorizat. Într-un protocol challenge-response, verificatorul emite o provocare (challenge), de obicei un nonce aleator, iar entitatea care

se autentifică trebuie să răspundă cu un răspuns care depinde criptografic de această provocare [turn0search1]. Acest model este utilizat pe scară largă în autentificarea la distanță, fie în autentificarea bazată pe parole, fie în scheme mai avansate cu semnături digitale [turn0search1][turn0search14].

În contextul RSA, mecanismul presupune ca serverul să genereze un nonce N , să îl trimită clientului, iar clientul să întoarcă o semnătură asupra hash-ului nonce-ului:

$$S \equiv \text{hash}(N)^d \pmod{n}.$$

Verifierul validează semnătura cu cheia publică și confirmă autenticitatea entității. Deoarece fiecare provocare este unică, acest mecanism previne reutilizarea răspunsurilor valide în sesiuni diferite.

3.5 Soluții existente și poziționarea proiectului

Literatura de specialitate conține multiple abordări ale autentificării criptografice, de la scheme bazate pe parole și şiruri unice la protocole complete precum TLS sau EAP pentru rețele complexe [turn0search16]. Metodele moderne includ autentificarea fără parole (passwordless), în care dispozitivele client semnează digital provocările pentru validare [turn0search14]. Cu toate acestea, integrarea unor protocole complete poate fi dificilă pe platforme embedded cu resurse limitate, deoarece acestea impun overhead de comunicație și procesare.

Proiectul de față se poziționează ca o soluție optimizată pentru autentificare criptografică într-un mediu embedded, combinând semnături digitale RSA cu un protocol de tip challenge-response adaptat comunicăției seriale UART. Această abordare oferă un bun compromis între securitate și eficiență, permitând validarea identității fără introducerea complexității unor protocole complete.

4 Proiectare și implementare

4.1 Metoda experimentală utilizată

Pentru realizarea obiectivelor proiectului a fost aleasă o abordare experimentală bazată pe o implementare software distribuită, formată dintr-un client PC și o aplicație embedded. Soluția a fost dezvoltată și testată pe un sistem real, utilizând comunicație serială UART între cele două componente. Această abordare a permis evaluarea funcțională și de securitate a mecanismului de autentificare în condiții apropiate de un scenariu real de utilizare.

Clientul PC a fost implementat în limbajul Python, fiind responsabil de gestionarea cheii private RSA și de realizarea operațiilor criptografice. Aplicația embedded a fost

implementată în limbajul C și are rolul de verificator al autentificării. Alegerea unei implementări software, în detrimentul unui simulator sau model analitic, a permis observarea directă a comportamentului sistemului și a limitărilor practice impuse de mediul embedded.

4.2 Alternative de proiectare și soluția aleasă

În etapa de proiectare au fost analizate mai multe alternative pentru realizarea mecanismului de autentificare. O primă alternativă a constat în utilizarea autentificării bazate pe parole sau chei partajate. Această soluție a fost abandonată din cauza vulnerabilităților cunoscute, precum expunerea secretelor și susceptibilitatea la atacuri de tip replay.

O altă alternativă analizată a fost utilizarea unui protocol complet de securitate, precum TLS, care oferă autentificare și criptare end-to-end. Deși această soluție este standardizată și robustă, a fost considerată prea complexă pentru scopul proiectului și dificil de integrat pe o platformă embedded cu resurse limitate.

Soluția aleasă a fost autentificarea bazată pe criptografie asimetrică RSA, utilizând semnături digitale într-un protocol de tip challenge-response. Această abordare oferă un compromis favorabil între securitate și complexitate, permitând autentificarea fără transmiterea secretelor și fără overhead-ul unui protocol complet.

4.3 Arhitectura generală a sistemului

Arhitectura sistemului este compusă din trei module principale: clientul PC, modulul embedded și canalul de comunicație serială UART. Fiecare modul are responsabilități bine definite, iar interacțiunea dintre acestea este realizată prin mesaje structurate. Figura 1 și Figura 2 prezintă arhitectura generală a sistemului de autentificare și fluxul de date dintre componente sale. La nivel hardware, sistemul embedded este implementat pe platforma Zynq, utilizând un design realizat în Vivado, în care logica software rulează pe procesorul ARM și gestionează comunicația și verificarea criptografică. La nivel funcțional, clientul PC inițiază procesul de autentificare, iar aplicația embedded, dezvoltată în Vitis, generează o provocare aleatorie (nonce), verifică semnătura digitală RSA primită și transmite rezultatul autentificării. Comunicația dintre clientul PC și sistemul embedded se realizează prin intermediul interfeței UART, asigurând schimbul controlat de mesaje pe durata întregului proces.

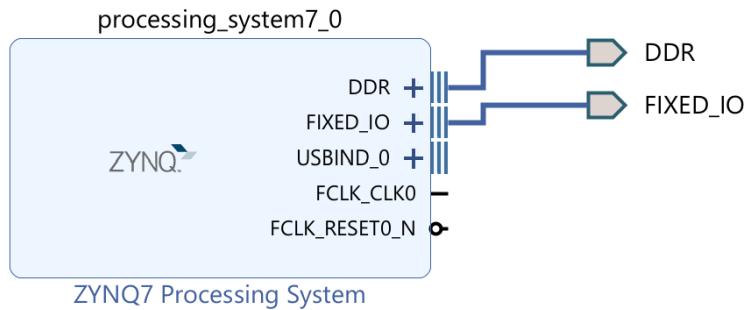


Figura 1: Diagramă bloc a sistemului embedded implementat pe platforma Zynq

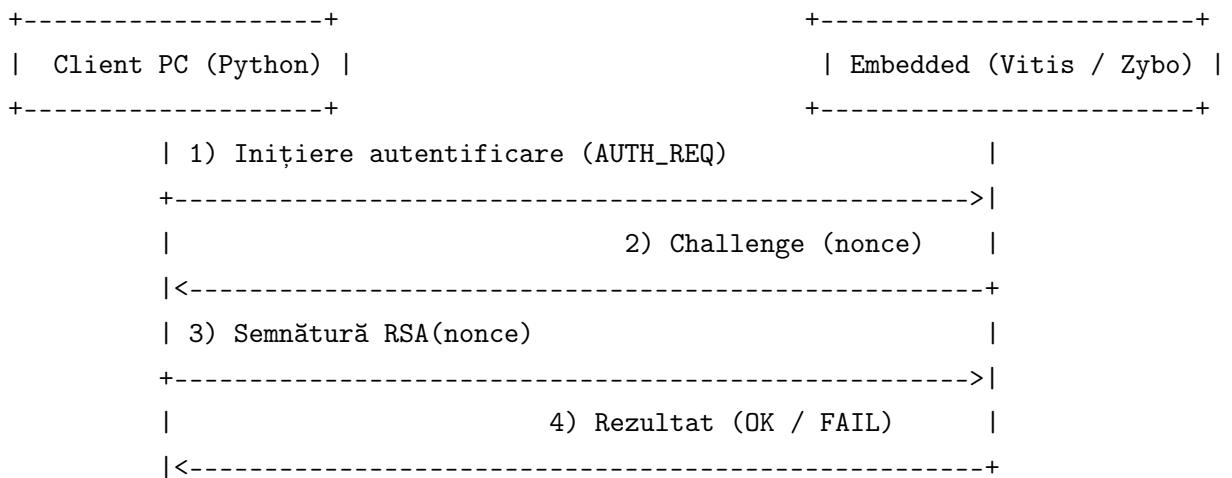


Figura 2: Diagrama de flux a autentificării dintre clientul PC și sistemul embedded

4.4 Descrierea modulelor software

Clientul PC are rolul de inițiator al autentificării și de executor al operațiilor criptografice. Acesta gestionează cheia privată RSA și realizează semnarea provocărilor primite de la sistemul embedded. Separarea logicii criptografice pe client reduce expunerea cheii private și limitează impactul unui eventual atac asupra dispozitivului embedded.

Modulul embedded are rolul de verificator al autentificării. Acesta deține cheia publică RSA asociată clientului și verifică semnaturile digitale primite. Modulul este responsabil de generarea provocărilor aleatorii și de decizia finală de acceptare sau respingere a autentificării.

4.5 Algoritmii implementați

Algoritmul principal implementat este mecanismul de autentificare de tip challenge-response bazat pe semnături digitale RSA. Fluxul algoritmic poate fi rezumat astfel:

1. clientul transmite un mesaj de inițiere a autentificării;

2. modulul embedded generează un nonce aleator și îl transmite clientului;
3. clientul calculează semnătura digitală asupra nonce-ului folosind cheia privată RSA;
4. semnătura este transmisă modulului embedded;
5. modulul embedded verifică semnătura utilizând cheia publică RSA;
6. autentificarea este acceptată sau respinsă în funcție de rezultatul verificării.

Utilizarea unui nonce unic pentru fiecare sesiune asigură prospețimea mesajelor și previne atacurile de tip replay.

4.6 Detalii de implementare și comunicație

Comunicația dintre clientul PC și sistemul embedded se realizează printr-un protocol simplu peste UART, structurat pe cadre care conțin tipul mesajului, lungimea și datele utile. Această structură permite extinderea ulterioară a protocolului cu noi tipuri de mesaje, fără modificări majore ale arhitecturii.

Arhitectura software a aplicației embedded este modulară, separând funcțiile de comunicație, logica protocolului și verificarea criptografică. Această separare facilitează testarea și întreținerea codului. Interfața cu utilizatorul este minimă, fiind realizată prin mesaje de stare transmise prin canalul de comunicație.

4.7 Manual de utilizare

Pentru utilizarea sistemului este necesar un calculator personal cu un sistem de operare compatibil cu limbajul Python și o conexiune serială către dispozitivul embedded. Clientul PC trebuie configurat cu cheia privată RSA corespunzătoare cheilor publice stocate pe sistemul embedded.

Procesul de utilizare presupune inițierea aplicației client, stabilirea conexiunii seriale și declanșarea procedurii de autentificare. Sistemul embedded răspunde automat la cererile primite și indică rezultatul autentificării prin mesaje transmise înapoi către client. Nu este necesară interacțiune suplimentară din partea utilizatorului în timpul procesului de autentificare.

5 Rezultate experimentale

5.1 Instrumente de proiectare utilizate

Implementarea și testarea sistemului de autentificare au fost realizate utilizând un set de instrumente software și hardware adecvate dezvoltării sistemelor embedded. Clientul PC

a fost implementat în limbajul Python și rulat pe un calculator personal cu sistem de operare Linux. Pentru dezvoltarea aplicației embedded a fost utilizat mediul Vitis, iar designul hardware al platformei Zynq a fost realizat în Vivado.

Platforma hardware utilizată a fost o placă de dezvoltare bazată pe arhitectura Zynq, care integrează un procesor ARM și logică programabilă. Comunicația dintre clientul PC și sistemul embedded a fost realizată prin intermediul interfeței UART. Pentru testare și depanare au fost utilizate utilitare de monitorizare a comunicației seriale și mesaje de debug transmise prin consolă.

5.2 Implementarea sistemului și resurse utilizate

Soluția propusă a fost implementată sub forma unei aplicații software care rulează pe procesorul ARM al platformei Zynq. Logica de autentificare, gestionarea protocolului și verificarea semnăturilor RSA sunt realizate exclusiv la nivel software, fără accelerare hardware dedicată. O parte din modulele software utilizate în cadrul implementării se bazează pe implementări open-source disponibile sub licență MIT, care au fost adaptate și integrate în arhitectura proprie a sistemului, cu respectarea termenilor de licențiere. Logica de autentificare, structura protocolului și integrarea componentelor reprezintă contribuția proprie.

Întrucât proiectul nu a urmărit optimizarea utilizării resurselor din logica programabilă, analiza consumului de resurse FPGA nu a constituit un obiectiv principal. Accentul a fost pus pe corectitudinea funcțională a mecanismului de autentificare și pe validarea fluxului de comunicație dintre clientul PC și aplicația embedded.

5.3 Procedura de testare

Testarea sistemului a fost realizată printr-o serie de scenarii controlate, menite să valideze corectitudinea funcțională și proprietățile de securitate ale mecanismului de autentificare. Procedura de testare a inclus următoarele etape:

- inițierea autentificării de către clientul PC;
- generarea și transmiterea provocării (nonce) de către sistemul embedded;
- semnarea provocării utilizând cheia privată RSA pe clientul PC;
- verificarea semnături digitale pe sistemul embedded;
- transmiterea rezultatului autentificării către client.

Au fost testate atât cazuri de autentificare validă, cât și situații în care semnătura transmisă nu corespunde provocării generate, pentru a verifica respingerea corectă a autentificării.

5.4 Rezultate obținute

Rezultatele experimentale au demonstrat funcționarea corectă a mecanismului de autentificare implementat. În cazul utilizării cheilor RSA valide, sistemul embedded a acceptat autentificarea și a transmis un mesaj de confirmare către clientul PC. În scenariile în care semnătura nu era validă sau era asociată unei provocări anterioare, autentificarea a fost respinsă.

Capturile de ecran obținute în timpul testării ilustrează schimbul de mesaje dintre clientul PC și sistemul embedded, precum și rezultatul procesului de autentificare. Acestea confirmă implementarea corectă a fluxului challenge-response și protecția împotriva reutilizării mesajelor de autentificare.

5.5 Analiza și interpretarea rezultatelor

Analiza rezultatelor indică faptul că utilizarea semnăturilor digitale RSA într-un mecanism de autentificare de tip challenge-response oferă un nivel ridicat de securitate, fără a introduce o complexitate excesivă în implementare. Deși operațiile RSA implică un cost computațional mai ridicat comparativ cu metodele bazate pe parole sau chei partajate, acest cost este acceptabil în contextul autentificărilor ocasionale.

Un avantaj important observat este eliminarea necesității transmiterii secretelor prin canalul de comunicație. De asemenea, utilizarea unui nonce unic pentru fiecare sesiune a prevenit cu succes atacurile de tip replay în scenariile testate.

5.6 Dificultăți întâlnite și soluții adoptate

Una dintre dificultățile întâlnite a fost sincronizarea corectă a mesajelor transmise prin interfața UART, în special în cazul mesajelor de dimensiuni variabile, precum semnăturile RSA. Această problemă a fost rezolvată prin definirea unui protocol de comunicație structurat, care include informații despre tipul și lungimea mesajelor.

O altă provocare a fost gestionarea timpului de execuție al operațiilor criptografice pe platforma embedded. Pentru a evita blocarea sistemului, operațiile de verificare au fost integrate într-o logică de control simplă, care permite tratarea corectă a mesajelor primite fără pierderi de date.

6 Concluzii

Proiectul a abordat problema autentificării sigure într-un sistem embedded, într-un context în care metodele clasice bazate pe parole prezintă vulnerabilități semnificative. Obiectivul principal a fost realizarea unui mecanism de autentificare care să evite transmiterea informațiilor sensibile și să ofere protecție împotriva atacurilor de tip replay. Acest obiectiv

a fost atins prin proiectarea și implementarea unui protocol de autentificare bazat pe criptografie asimetrică RSA și mecanismul challenge-response.

O contribuție importantă a proiectului constă în integrarea directă a semnăturilor digitale RSA într-un protocol simplu, adaptat unui mediu embedded și comunicăției seriale UART. De asemenea, separarea clară între clientul PC și sistemul embedded a permis protejarea cheii private și o arhitectură modulară, ușor de înțeles și extins.

Printre avantajele soluției realizate se numără nivelul ridicat de securitate oferit de criptografia asimetrică, eliminarea parolelor statice și protecția eficientă împotriva reutilizării mesajelor de autentificare. În același timp, proiectul prezintă și anumite limitări, în special legate de costul computațional al operațiilor RSA și de lipsa unei infrastructuri complete de management al cheilor sau certificatelor digitale.

Soluția implementată poate fi utilizată ca bază pentru aplicații care necesită autentificare sigură, precum sisteme de control embedded, dispozitive IoT, echipamente industriale sau prototipuri educaționale pentru studierea mecanismelor criptografice. Simplitatea arhitecturii o face potrivită pentru medii în care resursele sunt limitate, dar cerințele de securitate sunt ridicate.

Direcțiile de dezvoltare viitoare includ optimizarea performanței prin utilizarea accelerării hardware pentru operațiile criptografice, integrarea autentificării mutuale între client și sistemul embedded, precum și extinderea soluției prin utilizarea certificatelor digitale și a unor protocole standardizate. De asemenea, sistemul ar putea fi adaptat pentru comunicări securizate peste rețea, extinzând mecanismul de autentificare către scenarii distribuite mai complexe.

Bibliografie

- [1] Stallings, W., *Cryptography and Network Security: Principles and Practice*, 7th Edition, Pearson Education, 2017.
- [2] Rivest, R. L., Shamir, A., Adleman, L., “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120–126.
- [3] Kaliski, B., “PKCS #1: RSA Cryptography Specifications”, RFC 8017, Internet Engineering Task Force (IETF), 2016, <https://www.rfc-editor.org/rfc/rfc8017>.
- [4] Wikipedia, “Challenge-response authentication”, https://en.wikipedia.org/wiki/Challenge-response_authentication
- [5] Sklavos, N., Chatzigeorgiou, I., *Digital Signatures for Embedded Systems*, Springer, 2010.

Anexa A

Cod client PC (Python)

```
1 import serial, struct
2
3 PORT = "COM3"           # Zybo COM port
4 BAUD = 115200
5
6 # --- RSA keys ---
7 p = 0xe2176ecf
8 q = 0xc3a906c9
9 n = 0xaccd20dde4a5da87
10 e = 0x10001           # 0x10001 = 65537
11 d = 0x15bbfb3d18801bd1      # private exponent - pe pc
12
13 def modexp(base, exp, mod):
14     res = 1
15     base %= mod
16     while exp:
17         if exp & 1:
18             res = (res * base) % mod
19         exp >>= 1
20         base = (base * base) % mod
21     return res
22
23 def crc16(data: bytes) -> int:
24     crc = 0xFFFF
25     for b in data:
26         crc ^= b << 8
27         for _ in range(8):
28             if crc & 0x8000:
29                 crc = (crc << 1) ^ 0x1021
30             else:
31                 crc <= 1
32         crc &= 0xFFFF
33     return crc
34
35 def send_msg(ser, mtype, payload: bytes):
36     hdr = bytes([0xAA, mtype, (len(payload)>>8)&0xFF, len(payload)
37                 &0xFF])
38     crc = crc16(hdr[1:]) ^ crc16(payload)
```

```

38     ser.write(hdr + payload + struct.pack(">H", crc))
39
40 def recv_msg(ser):
41     while True:
42         b = ser.read(1)
43         if not b:
44             return None
45         if b[0] == 0xAA:
46             break
47     hdr = ser.read(3)
48     if len(hdr) < 3:
49         return None
50     mtype, lhi, llo = hdr
51     length = (lhi<<8) | llo
52     payload = ser.read(length)
53     crc_raw = ser.read(2)
54     if len(crc_raw) < 2:
55         return None
56     crc_recv, = struct.unpack(">H", crc_raw)
57     crc = crc16(bytes([mtype, lhi, llo])) ^ crc16(payload)
58     if crc != crc_recv:
59         print("CRC error")
60     return None
61
62     return mtype, payload
63
64 with serial.Serial(PORT, BAUD, timeout=1) as ser:
65     for i in range(3):
66         print(f"\n==== Test {i+1} ===")
67         # 1. send AUTH_REQUEST
68         send_msg(ser, 0x01, b"")
69         print("AUTH_REQUEST sent")
70
71         # 2. receive CHALLENGE
72         m = recv_msg(ser)
73         if not m or m[0] != 0x02:
74             print("No challenge received")
75             break
76         chall = int.from_bytes(m[1], "big")
77         print("Challenge:", hex(chall))
78
# 3. make signature

```

```

79         sig = modexp(chall, d, n)
80         if i == 1:
81             # deliberately break the signature on the 3rd run
82             print("!! CORRUPTING signature for test 3 !!")
83             sig ^= 1
84         sig_bytes = sig.to_bytes(8, "big")
85         send_msg(ser, 0x03, sig_bytes)
86         print("AUTH_RESPONSE sent (sig =", hex(sig), ")")
87
88     # 4. receive result
89     m = recv_msg(ser)
90     if not m or m[0] != 0x04:
91         print("No result received")
92         break
93     print("AUTH", "OK" if m[1][0] == 1 else "FAIL")

```

```

1 import random
2 from math import gcd
3
4 def is_prime(n):
5     if n < 2: return False
6     if n % 2 == 0:
7         return n == 2
8     i = 3
9     while i * i <= n:
10        if n % i == 0:
11            return False
12        i += 2
13    return True
14
15 def rand_prime_32():
16     while True:
17         x = random.randrange(2**31, 2**32-1) | 1    # impar
18         if is_prime(x):
19             return x
20
21 def egcd(a, b):
22     if b == 0:
23         return 1, 0, a
24     x1, y1, g = egcd(b, a % b)
25     return y1, x1 - (a // b) * y1, g
26

```

```

27 def modinv(a, m):
28     x, y, g = egcd(a, m)
29     if g != 1:
30         raise ValueError("nu exista invers modular")
31     return x % m
32
33 def gen_keys():
34     p = rand_prime_32()
35     q = rand_prime_32()
36     while q == p:
37         q = rand_prime_32()
38
39     n = p * q
40     phi = (p - 1) * (q - 1)
41     e = 65537
42     if gcd(e, phi) != 1:
43         # rareori, atunci repe i
44         return gen_keys()
45
46     d = modinv(e, phi)
47     return p, q, n, e, d, phi
48
49 if __name__ == "__main__":
50     p, q, n, e, d, phi = gen_keys()
51     print(f"p = {p:#x}")
52     print(f"q = {q:#x}")
53     print(f"n = {n:#x}")
54     print(f"e = {e:#x}")
55     print(f"d = {d:#x}")
56     print(f"phi= {phi:#x}")

```

```

1 import serial, struct
2
3 PORT = "COM3"           # Zybo COM port
4 BAUD = 115200
5
6 # --- RSA keys ---
7 p = 0xe2176ecf
8 q = 0xc3a906c9
9 n = 0xaccd20dde4a5da87
10 e = 0x0000000000010001      # 0x10001 = 65537
11 d = 0x15bbfb3d18801bd1      # private exponent - pe pc

```

```

12
13 def modexp(base, exp, mod):
14     res = 1
15     base %= mod
16     while exp:
17         if exp & 1:
18             res = (res * base) % mod
19         exp >>= 1
20         base = (base * base) % mod
21     return res
22
23 def crc16(data: bytes) -> int:
24     crc = 0xFFFF
25     for b in data:
26         crc ^= b << 8
27         for _ in range(8):
28             if crc & 0x8000:
29                 crc = (crc << 1) ^ 0x1021
30             else:
31                 crc <=> 1
32     crc &= 0xFFFF
33     return crc
34
35 def send_msg(ser, mtype, payload: bytes):
36     hdr = bytes([0xAA, mtype, (len(payload)>>8)&0xFF, len(payload)
37                  &0xFF])
38     crc = crc16(hdr[1:]) ^ crc16(payload)
39     ser.write(hdr + payload + struct.pack(">H", crc))
40
41 def recv_msg(ser):
42     while True:
43         b = ser.read(1)
44         if not b:
45             return None
46         if b[0] == 0xAA:
47             break
48     hdr = ser.read(3)
49     if len(hdr) < 3:
50         return None
51     mtype, lhi, llo = hdr
52     length = (lhi<<8) | llo

```

```

52     payload = ser.read(length)
53     crc_raw = ser.read(2)
54     if len(crc_raw) < 2:
55         return None
56     crc_recv, = struct.unpack(">H", crc_raw)
57     crc = crc16(bytes([mtype, lhi, llo])) ^ crc16(payload)
58     if crc != crc_recv:
59         print("CRC error")
60     return None
61
62
63 # Define what each test does
64 # "good"          -> correct signature
65 # "flip"          -> flip 1 bit of sig
66 # "wrong_d"       -> sign with d+1 (wrong key)
67 # "zero"          -> sig = 0
68 test_modes = [
69     "good",        # 1
70     "good",        # 2
71     "good",        # 3
72     "good",        # 4
73     "good",        # 5
74     "flip",        # 6 -> bad
75     "wrong_d",     # 7 -> bad
76     "zero",        # 8 -> bad
77     "good",        # 9
78     "good",        # 10
79 ]
80
81 with serial.Serial(PORT, BAUD, timeout=1) as ser:
82     for i, mode in enumerate(test_modes, start=1):
83         print(f"\n==== Test {i} ({mode}) ====")
84
85         # 1. send AUTH_REQUEST
86         send_msg(ser, 0x01, b"")
87         print("AUTH_REQUEST sent")
88
89         # 2. receive CHALLENGE
90         m = recv_msg(ser)
91         if not m or m[0] != 0x02:
92             print("No challenge received, got:", m)

```

```

93         break
94     chall = int.from_bytes(m[1], "big")
95     print("Challenge:", hex(chall))
96
97     # 3. compute base correct signature
98     sig = modexp(chall, d, n)
99
100    # apply corruption depending on mode
101    if mode == "flip":
102        print("!! FLIPPING lowest bit of signature !!")
103        sig ^= 1
104
105    elif mode == "wrong_d":
106        print("!! Using WRONG private key d+1 !!")
107        sig = modexp(chall, d + 1, n)
108
109    elif mode == "zero":
110        print("!! Using ZERO as signature !!")
111        sig = 0
112
113    # else "good": leave sig as-is
114
115    sig_bytes = sig.to_bytes(8, "big")
116    send_msg.ser, 0x03, sig_bytes)
117    print("AUTH_RESPONSE sent (sig =", hex(sig), ")")
118
119    # 4. receive result
120    m = recv_msg.ser)
121    if not m or m[0] != 0x04:
122        print("No result received, got:", m)
123        break
124    print("AUTH", "OK" if m[1][0] == 1 else "FAIL")

```

Anexa B

Protocol de autentificare

```

1 // protocol.h
2 #pragma once
3 #include <stdint.h>
4

```

```

5 #define MSG_START_BYTE      0xAA
6
7 #define MSG_AUTH_REQUEST    0x01
8 #define MSG_AUTH_CHALLENGE  0x02
9 #define MSG_AUTH_RESPONSE   0x03
10 #define MSG_AUTH_RESULT     0x04
11
12 typedef struct {
13     uint8_t type;
14     uint16_t length;
15     uint8_t payload[64]; // challenge+sig
16 } Message;
17
18 uint16_t crc16(const uint8_t *data, uint16_t len);
19
20 void send_message(uint8_t type, const uint8_t *payload, uint16_t
length);
21 int recv_message(Message *msg);

```

```

// protocol.c
#include "protocol.h"
#include "uart.h"

4

5 uint16_t crc16(const uint8_t *data, uint16_t len)
{
7     uint16_t crc = 0xFFFF;
8     for (uint16_t i = 0; i < len; i++) {
9         crc ^= (uint16_t) data[i] << 8;
10        for (uint8_t j = 0; j < 8; j++) {
11            if (crc & 0x8000)
12                crc = (crc << 1) ^ 0x1021;
13            else
14                crc <= 1;
15        }
16    }
17    return crc;
18 }

20 void send_message(uint8_t type, const uint8_t *payload, uint16_t
length)
{
21     uint8_t header[4];

```

```

23     header[0] = MSG_START_BYTE;
24     header[1] = type;
25     header[2] = (length >> 8) & 0xFF;
26     header[3] = length & 0xFF;
27
28     uint16_t crc = crc16(&header[1], 3);           // type + len
29     crc ^= crc16(payload, length);
30
31     uart_send_bytes(header, 4);
32     if (length) uart_send_bytes(payload, length);
33
34     uint8_t crc_bytes[2] = { crc >> 8, crc & 0xFF };
35     uart_send_bytes(crc_bytes, 2);
36 }
37
38 int recv_message(Message *msg)
39 {
40     uint8_t b;
41
42     // caut start byte
43     do {
44         uart_recv_bytes(&b, 1);
45     } while (b != MSG_START_BYTE);
46
47     uint8_t hdr[3];
48     uart_recv_bytes(hdr, 3);
49     msg->type = hdr[0];
50     msg->length = ((uint16_t)hdr[1] << 8) | hdr[2];
51     if (msg->length > sizeof(msg->payload)) return -1;
52
53     if (msg->length) uart_recv_bytes(msg->payload, msg->length);
54
55     uint8_t crc_raw[2];
56     uart_recv_bytes(crc_raw, 2);
57     uint16_t crc_recv = ((uint16_t)crc_raw[0] << 8) | crc_raw[1];
58
59     uint8_t t[3] = { msg->type, hdr[1], hdr[2] };
60     uint16_t crc = crc16(t, 3);
61     crc ^= crc16(msg->payload, msg->length);
62
63     return (crc == crc_recv) ? 0 : -2;

```

```
64 }
```

Anexa C

Implementare RSA

```
1 // rsa.h
2 #pragma once
3 #include <stdint.h>
4
5 typedef struct {
6     uint64_t n;
7     uint64_t e;
8 } RsaPublicKey;
9
10 uint64_t rsa_modexp(uint64_t base, uint64_t exp, uint64_t mod);
11 int rsa_verify_challenge(uint64_t challenge, uint64_t signature,
12                         const RsaPublicKey *pub);
13
14 // public key a clientului (PC)
15 extern const RsaPublicKey g_client_pubkey;
```

```
1 // rsa.c
2 #include "rsa.h"
3
4 // public key from PC
5 const RsaPublicKey g_client_pubkey = {
6     .n = 0xaccd20dde4a5da87ULL,
7     .e = 0x0000000000010001ULL // 0x10001 = 65537
8 };
9
10 static uint64_t mul_mod(uint64_t a, uint64_t b, uint64_t mod)
11 {
12     uint64_t res = 0;
13     a %= mod;
14
15     while (b) {
16         if (b & 1) {
17             if (res >= mod - a) {
18                 res = res - (mod - a);
19             } else {
```

```

20             res += a;
21         }
22     }
23
24     b >= 1;
25
26     if (a >= mod - a) {
27         a = a - (mod - a);
28     } else {
29         a += a;
30     }
31 }
32
33 return res;
34 }

35
36 uint64_t rsa_modexp(uint64_t base, uint64_t exp, uint64_t mod)
37 {
38     uint64_t result = 1 % mod;
39     base %= mod;
40
41     while (exp) {
42         if (exp & 1) {
43             result = mul_mod(result, base, mod);
44         }
45         exp >= 1;
46         if (exp) {
47             base = mul_mod(base, base, mod);
48         }
49     }
50
51     return result;
52 }

53
54 int rsa_verify_challenge(uint64_t challenge, uint64_t signature,
55                         const RsaPublicKey *pub)
56 {
57     uint64_t recovered = rsa_modexp(signature, pub->e, pub->n);
58     return (recovered == challenge);
59 }
```

Anexa D

Comunicație UART

```
1 // uart.h
2 #pragma once
3 #include <stdint.h>
4 #include "xuartps.h"
5
6 int uart_init(void);
7 void uart_send_bytes(const uint8_t *buf, uint16_t len);
8 uint16_t uart_recv_bytes(uint8_t *buf, uint16_t len);
```

```
1 // uart.c
2 #include "uart.h"
3 #include "xparameters.h"
4 #include "xuartps.h"
5
6 static XUartPs UartPs;
7
8 #define UART_BASEADDR XPAR_XUARTPS_0_BASEADDR // <- asta e baza
9     UART1
10
11 #define UART_BAUDRATE 115200
12
13 int uart_init(void)
14 {
15     XUartPs_Config *cfg;
16     int status;
17
18     // LookupConfig primește base address
19     cfg = XUartPs_LookupConfig(UART_BASEADDR);
20     if (cfg == NULL) {
21         return -1;
22     }
23
24     status = XUartPs_CfgInitialize(&UartPs, cfg, cfg->BaseAddress);
25     if (status != XST_SUCCESS) {
26         return -2;
27     }
28
29     XUartPs_SetBaudRate(&UartPs, UART_BAUDRATE);
30
31     return 0;
```

```

29 }
30
31 void uart_send_bytes(const uint8_t *buf, uint16_t len)
32 {
33     uint16_t sent = 0;
34     while (sent < len) {
35         int n = XUartPs_Send(&UartPs, (uint8_t *) (buf + sent), len
36             - sent);
37         if (n > 0) sent += n;
38     }
39
40 uint16_t uart_recv_bytes(uint8_t *buf, uint16_t len)
41 {
42     uint16_t recv = 0;
43     while (recv < len) {
44         int n = XUartPs_Recv(&UartPs, buf + recv, len - recv);
45         if (n > 0) recv += n;
46     }
47     return recv;
48 }
```

Anexa E

Aplicație embedded

```

1 /*
2  ****
3 * Copyright (C) 2023 Advanced Micro Devices, Inc. All Rights
4 * Reserved.
5 * SPDX-License-Identifier: MIT
6 ****
7 */
8
9 #ifndef __PLATFORM_H_
10 #define __PLATFORM_H_
11
12 #ifndef SDT
13 #include "platform_config.h"
14 #endif
```

```
12
13 void init_platform();
14 void cleanup_platform();
15
16 #endif
```

```
1 /*
2 * Copyright (C) 2023 Advanced Micro Devices, Inc. All Rights
3 * Reserved.
4 * SPDX-License-Identifier: MIT
5 */
6
6 #include "xparameters.h"
7 #include "xil_cache.h"
8
9 #ifndef SDT
10 #include "platform_config.h"
11#endif
12
13 #ifdef STDOUT_IS_16550
14 #include "xuartns550_l.h"
15
16 #define UART_BAUD 9600
17#endif
18
19 void
20 enable_caches()
21{
22 #ifdef __PPC__
23     Xil_ICacheEnableRegion(CACHEABLE_REGION_MASK);
24     Xil_DCacheEnableRegion(CACHEABLE_REGION_MASK);
25 #elif __MICROBLAZE__
26 #ifdef XPAR_MICROBLAZE_USE_ICACHE
27     Xil_ICacheEnable();
28 #endif
29 #ifdef XPAR_MICROBLAZE_USE_DCACHE
30     Xil_DCacheEnable();
31 #endif
32#endif}
```

```

33 }
34
35 void
36 disable_caches()
37 {
38 #ifdef __MICROBLAZE__
39 #ifdef XPAR_MICROBLAZE_USE_DCACHE
40     Xil_DCacheDisable();
41 #endif
42 #ifdef XPAR_MICROBLAZE_USE_ICACHE
43     Xil_ICacheDisable();
44 #endif
45 #endif
46 }
47
48 void
49 init_uart()
50 {
51 #ifdef STDOUT_IS_16550
52     XUartNs550_SetBaud(STDOUT_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ,
53                         UART_BAUD);
54     XUartNs550_SetLineControlReg(STDOUT_BASEADDR,
55                                   XUN_LCR_8_DATA_BITS);
56 #endif
57 /* Bootrom/BSP configures PS7/PSU UART to 115200 bps */
58 }
59
60 void
61 init_platform()
62 {
63     enable_caches();
64     init_uart();
65 }
66
67 void
68 cleanup_platform()
69 {
70     disable_caches();
71 }
```

```

1 // auth.h
2 #pragma once
```

```

3 void handle_auth_session(void);

1 // auth.c
2 #include "auth.h"
3 #include "protocol.h"
4 #include "rsa.h"
5 #include "xil_printf.h"

6
7 static uint64_t g_challenge_counter = 1;

8
9 static uint64_t generate_challenge(void)
10 {
11     // creste un contor
12     return g_challenge_counter++;
13 }

14
15 void handle_auth_session(void)
16 {
17     Message msg;
18     int st;

19
20     xil_printf("Waiting for AUTH_REQUEST...\r\n");
21
22     st = recv_message(&msg);
23     if (st != 0 || msg.type != MSG_AUTH_REQUEST) {
24         xil_printf("Bad AUTH_REQUEST (st=%d, type=%d)\r\n", st, msg
25             .type);
26         return;
27     }
28
29     xil_printf("AUTH_REQUEST received.\r\n");
30
31     // trimite challenge
32     uint64_t challenge = generate_challenge();
33     uint8_t chall_bytes[8];
34     for (int i = 0; i < 8; i++)
35         chall_bytes[7 - i] = (challenge >> (8 * i)) & 0xFF;
36
37     send_message(MSG_AUTH_CHALLENGE, chall_bytes, 8);
38     xil_printf("CHALLENGE: 0x%08x%08x\r\n",
39                 (uint32_t)(challenge >> 32), (uint32_t)challenge);

```

```

40 // primeste raspuns
41 st = recv_message(&msg);
42 if (st != 0 || msg.type != MSG_AUTH_RESPONSE || msg.length != 8) {
43     xil_printf("Bad AUTH_RESPONSE (st=%d, len=%d)\r\n", st, msg.length);
44     return;
45 }
46
47 uint64_t signature = 0;
48 for (int i = 0; i < 8; i++)
49     signature = (signature << 8) | msg.payload[i];
50
51 xil_printf("Signature: 0x%08x%08x\r\n",
52             (uint32_t)(signature >> 32), (uint32_t)signature);
53
54 int ok = rsa_verify_challenge(challenge, signature, &
55     g_client_pubkey);
56 uint8_t res = ok ? 1 : 0;
57 send_message(MSG_AUTH_RESULT, &res, 1);
58
59 xil_printf("AUTH %s\r\n", ok ? "OK" : "FAIL");
}

```

```

1 #include "platform.h"
2 #include "xil_printf.h"
3 #include "uart.h"
4 #include "auth.h"
5
6 int main(void)
{
7     init_platform();
8
9     if (uart_init() != 0) {
10         xil_printf("UART init failed\r\n");
11         while (1);
12     }
13
14     xil_printf("RSA Auth Server ready.\r\n");
15
16     while (1) {
17         handle_auth_session();
18

```

```
19    }
20
21    cleanup_platform();
22    return 0;
23 }
```