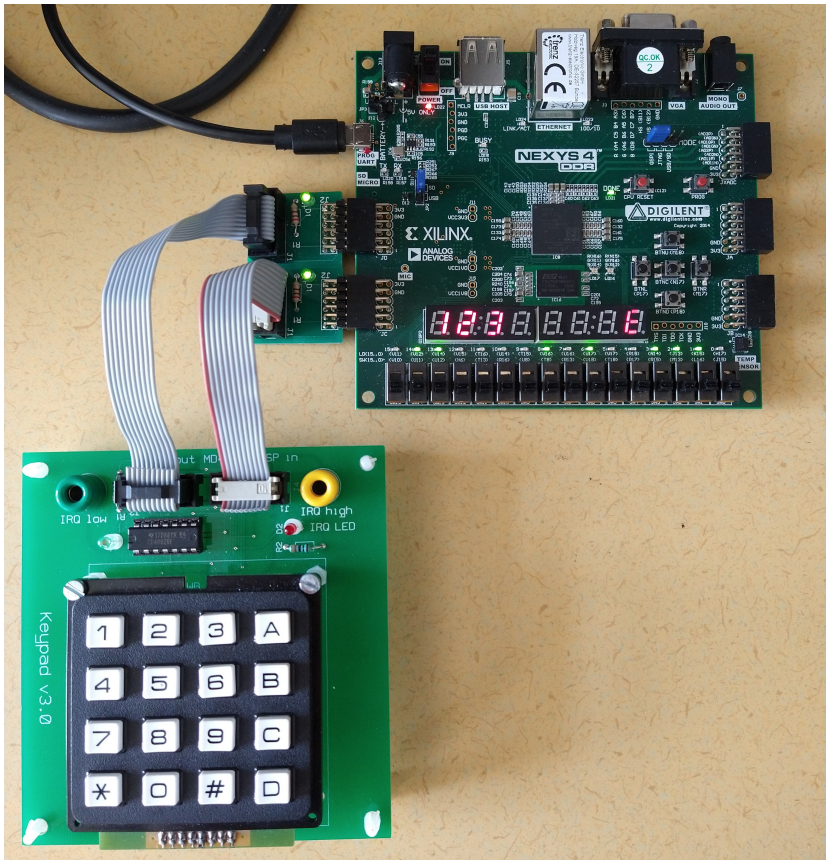
CHALMERS



Öppen design av säkerhetsdosa för tvåfaktorsautentiseringssystem

Kandidatarbete vid Data- och Informationsteknik

Henrik Andersson, Daniel Bylinka, David Frielingsdorf,
Johan Hellström, Sabine Randow, Adam Thunberg.

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2021
www.chalmers.se

Kandidatarbete 2021

Öppen design av säkerhetsdosa för tvåfaktorsautentiseringssystem

Henrik Andersson
Daniel Bylinka
David Frielingsdorf
Johan Hellström
Sabine Randow
Adam Thunberg

CHALMERS

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborg, Sverige 2021

Öppen design av säkerhetsdosa för tvåfaktorsautentiseringssystem
Henrik Andersson, Daniel Bylinka, David Frielingsdorf,
Johan Hellström, Sabine Randow, Adam Thunberg

Handledare: Lars Svensson, Institutionen för Data- och Informationsteknik
Examinator: Torbjörn Tjelldén, Institutionen för Data- och Informationsteknik

Kandidatarbete 2021
Institutionen för Data- och Informationsteknik
Chalmers tekniska högskola
412 96 Göteborg, Sverige
Telefon +46 31 772 1000

Omslag: Egen bild på den utvecklade koddosan.

Typsatt i LaTeX
Göteborg, Sverige 2021

Sammandrag

I dagens samhälle där mer och mer arbete lämnas till datorer är möjligheten att identifiera användare i digitala tjänster av stor vikt. Detta är speciellt viktigt för bland annat banktjänster över internet. Många banker har löst detta genom att dela ut koddosor till sina kunder. Dosorna innehåller hemliga nycklar som används för att bekräfta att personen som försöker använda tjänsten har tillgång till sin dosa. Problematiken med detta är att bankdosorna tillhör bankerna, och för att implementera en liknande lösning i andra system för att identifiera användare behöver man utveckla sin egna produkt från grunden. Vårt projekt ämnar att underlätta integreringen av ett sådant system för dem som vill genom att utveckla en öppen design av en koddosa med tillhörande autentiseringsmjukvara.

Vårt inloggningssystem består av två delar. Den ena är själva dosan vars design är utvecklad i VHDL för syntetisering till FPGA-kort; den andra delen är en autentiseringsmodul till Linux för att demonstrera tjänsten. Inloggningsprocessen fungerar likt de flesta banktjänster. Vid försök till inloggning ger modulen en utmaningskod för dosan. Koden skrivs in i dosan som tillhandahåller en svarskod, som ska vara nära omöjlig att hitta utan en hemlig kryptografisk nyckel. Denna svarskod beräknas genom att kombinera nyckeln med utmaningskoden och sedan används hashfunktionen SHA-3 för att få fram ett unikt värde. Svarskoden matas sedan in i modulen som gör samma beräkningar som dosan. Om svarskoden är identisk så accepteras inloggningsförsöket.

Koden för detta projekt finns öppet tillgänglig med licensen GPL.

Abstract

In a society where more work is continually left for computers, online identification has never been of greater importance. One would struggle to come up with a better example than online banks. Many have tried to solve this using two-factor authentication with small external keypads. These contain cryptographic keys unique to its user that are used to verify their identity online. A problem is that these devices are proprietary and are owned by the banks. To implement a similar solution you would have to design the devices from scratch yourself. Our project has intends to facilitate this process by developing a foundation for such a system.

Our system consists of two parts. Firstly, the codebox which has been developed in VHDL for synthesizing and programming an FPGA. Secondly we have an authentication module for Linux used to demonstrate the system by identifying and logging in a user. The login process is similar to that of any online bank: a user attempting to log in receives a challenge code from the PAM module. The challenge code is then fed to the keypad, which in turn generates a response code calculated with the cryptographic key. This response code is calculated by using the hash function SHA-3 on a concatenation of the secret key and challenge code. The response code is fed to the PAM module, which has preliminarily performed identical calculations and expects the same result. Lastly the PAM-module determines the determined code and the code received from the user are the same; if so the user is logged in.

The code from this project is openly accessible and uses the GPL license.

Förord

Vi vill tacka Lars Svessons som varit vår handledare under projektet. Du har svarat på alla våra frågor och hjälpt oss framåt när vi fastnat.

Tack till Team Keccak för tillåtelsen att använda grafik från deras hemsida.

Innehåll

Ordlista

AES: *Advanced Encryption Standard*. En symmetrisk krypteringsalgoritm ofta använd av amerikanska myndigheter och har inbyggt stöd i många processorer.

Bit: Engelsk förkortning för *binary digit*. En bit kan anta ett av två värden, antingen 1 eller 0, och är den minsta komponenten för binärt kodad information.

Byte: Åtta bits av information, se Bit.

ChaCha20: En chifferfunktion som är mer effektiv hos system där processorn inte har inbyggt stöd för AES-accelerering[1]. Krypteringsalgoritmen utför enbart matematiska manipulationer av indatan, vilket leder till att processen är avsevärt snabbare än exempelvis AES.

Demon: Svensk översättning av *daemon*, vilket inom Unix-liknande operativsystem är ett program som är avsett att köras i bakgrunden på en dator utan att användaren startar det. Brukar kallas för tjänst på andra operativsystem.

ECC: *Elliptic Curve Cryptography* (elliptisk kryptering). En asymmetrisk kryptering baserad på elliptiska kurvor.

FPGA: *Field-Programmable Gate Array*. En integrerad krets som kan programmeras om.

GnuPG: *GNU Privacy Guard*. En mjukvarusamling som implementerar *OpenPGP*-standarden, används för digital kryptering, signering och nyckelhantering [2].

GCC: *GNU Compiler Collection*. En samling kompilatorer som består av bland annat C, C++ och Fortran[3].

Hash/Hashvärde: Ett tal framtaget ur ett annat, oftast mycket större, tal enligt en specifik deterministisk algoritm. Hasher används ofta för elektroniska signaturer och kan även ibland benämnas kondensat.

Hashalgoritm: Funktion som beräknar ett hashvärde av bestämd längd utifrån godtycklig data.

HMAC: *Keyed-Hash Message Authentication Code*. En specifik typ av *Message Authentication Code* (MAC) som består av en kryptografisk hashfunktion och en hemlig kryptografisk nyckel [4].

Keccak: En familj av kryptografiska primitiver varav de säkraste instanserna ingår i SHA-3.

Kondensat: Ett mindre tal som representerar ett större tal, även benämnt hash eller hashvärde.

MAC: *Message Authentication Code*. Kallas ibland för *tag*. En MAC är en kod som verifierar att ett meddelande ej modifierats och kommer från rätt avsändare.

Makefil: En fil som beskriver förhållanden mellan källkodsfiler och kompilerade program, och anger kommandon som krävs för att kompilera programmen [5].

NIST: *National Institute of Standards of Technology*. En Amerikansk myndighet som fastställer standarder för bland annat teknisk utrustning.

OpenSSL: OpenSSL är en open source-implementation av bland annat SSL- och TLS-protokollen men också diverse hash- och krypteringsalgoritmer [6].

root: Administratörskontot i Unix-liknande operativsystem.

RSA: *Rivest-Shamir-Adlerman*. Den första asymmetriska krypteringsalgoritmen [7].

SHA-3: *Secure Hash Algorithm-3*. Senaste medlemmen av *Secure Hash Algoritm*-familjen av hashalgoritmer [8].

Slumpfrö: Ett tal använt för att definiera initialtillståndet av en pseudoslumpmässig nummergenerator.

SSL/TLS: Transport Layer Security (TLS) är ett öppet kryptografiskt kommunikationsprotokoll för säkert utbyte av information mellan datorsystem [9]. TLS är efterträdaren av den nu utfasade SSL.

Svarskod: Ett värde som beräknas utifrån en utmaningskod.

Switch-sats: En kontrollmekanism i programmering som styr programflödet. En switch-sats har flera fall (cases) med olika kod, switch:en väljer ett utav fallen efter en variabel angiven av utvecklaren.

Utmaningskod: Ett godtyckligt värde som ska matas in i en separat enhet för att generera en svarskod.

VHDL: *Very high speed integrated circuit hardware description language*. Ett hårdvarubeskrivande språk som används för att specificera och konstruera digitala kretsar.

XOR: Ett logiskt villkor som betyder “antingen A eller B, men inte båda”.

1 Inledning

Idag används koddosor bland annat av bankkunder för att styrka sin identitet på distans hos internetbanker, exempelvis SEB [10]. Det är ett av många tecken på att fler vardagliga ärenden görs via datorer genom internet, därav är det viktigt att kunna säkerställa identiteten hos en användare av en digital tjänst. I dessa fall är det oftast inte tillräckligt med identifiering via lösenord. Många banker likt SEB har försökt lösa detta problem genom att dela ut dosor som innehåller hemliga nycklar. De kan användas för att bekräfta att den som försöker använda tjänsten är i besittning av denna specifika koddosan.

Ett problem med många av de säkerhetsdosor som används idag är att designen av hårdvaran i koddosan inte är tillgänglig för allmänheten. Därmed har inte användare eller organisationer möjligheten att kontrollera att en koddosa verkligen nyttjar säkra metoder, utan måste blint lita på den bank eller organisation som producerat dosan.

Vi föreslår därför en koddosa där programkoden och hårdvarudesignen finns fritt tillgängliga enligt så kallad ”Open-Source” programvara. Koddosan bör fungera snarlikt en säkerhetsdosa där användaren matar in en kod och läser in svaret i ett system som kontrollerar detta; i detta projektet används Linux inloggningssystem i demonstrationssyfte för att autentisera en användare.

1.1 Syfte

Poängen är att ge användare av säkerhetsdosor samma möjlighet för lärdom om hårdvaran som utvecklarna av produkten i hoppet att detta skall öka säkerhet och tillit i sådana system. Detta innebär också att källkod behöver vara väl konstruerad och dokumenterad.

Syftet med projektet är också att undersöka processen av att konstruera en koddosa. Därtill ska vi diskutera möjliga följder (för- och nackdelar, användningsområden) av att släppa en öppen design av det här slaget.

1.2 Problemformulering

Vi har utvecklat en prototyp av en säkerhetsdosa, som genom tvåfaktorautentisering bidrar till att lösa problemet nämnt i avsnitt 1, och har gjort designen öppet tillgänglig på internet [11]. Undersökning och utvecklingen har skett under en period av åtta veckor vilket ledde till att implementeringen inte var alltför stor eller invecklad.

Gruppen fick tillgång till ett FPGA-kort vilket medförde en restriktion: designen behövde få plats på FPGA-kortets begränsade antal logikceller. Detta gav upphov till en undersökning av storlek av existerande implementeringar av algoritmer på FPGA-kort. Slutligen behövde vi välja en lämplig öppen licens [12] för projektets källkod.

Vi behöver också utveckla ett komplement till koddosan för att demonstrera ett fullständigt system. Detta gör vi med hjälp av Linux-PAM [13] och mjukvara skriven i C vars upp-

gift är att logga in användaren när PAM bekräftat att användaren är i besittning av rätt dosa.

1.3 Avgränsningar

Projektet utfördes på begränsad tid vilket medförde några initiala avgränsningar. Dessa och fler diskuteras i avsnitt 7.7 som handlar om vidareutveckling av produkten.

1.3.1 Grafiska gränssnitt

Förutom hur användaren behöver se vilket tillstånd PAM-modulen är i och eventuella svarskoder som behöver visas har inget grafiskt gränssnitt utvecklats. Dels för att användargränssnitt inte är projektets fokus och dels för att projektets syfte är att förse utvecklare med en öppen grund att bygga på. Dessutom kan gränssnittet variera drastiskt efter behov och tjänst; därav hade det vart en utmaning att utveckla en universellt användbar visuell design på begränsad tid. Dessa resonemang gäller även för eventuella gränssnitt av hårdvaran som utvecklas för att generera svarskoder och därmed verifiera användarens identitet.

1.3.2 Hashning

Vi kommer inte att gå djupare än att kort förklara hur de hashalgoritmer som används är implementerade. Att skriva egna hashalgoritmer ligger bortom gruppmedlemmarnas kunskap och skulle riskera att en eller flera delar av produkten implementeras med okända fel och säkerhetsbrister. Produkten kommer därför att använda sig av ett par väl etablerade bibliotek och teknologier och därmed baseras på existerande kunskap och arbete tillgängligt på internet.

2 Teknisk bakgrund

I följande rubriker förklaras grundläggande koncept som är centrala i produktens utveckling. För att enklare förstå det som utförts inom projektet rekommenderas förståelse för grunderna inom datorteknik, kryptering och hashalgoritmer. För att förstå konstruktionen av koddosan krävs dessutom en viss förståelse för digital hårdvara och det hårdvarubeskrivande programspråket VHDL, vilket motsvarar centrala innehållet i kursen Digital Konstruktion (EDA322) på Chalmers. För att förstå logiken för verifikation i vår mjukvara krävs viss kunskap om programspråket C samt viss kännedom om systemet Linux-PAM [13] och programbiblioteket OpenSSL [6].

2.1 Kryptering

Ordet kryptografi kommer från de två grekiska orden ”κρυπτός” (dölja), följt av ”γράφειν” (skrift) [14, s.2]. Vanligtvis när man pratar om kryptering så menar man antingen transformationen av klartext till kryptotext eller återställningen från kryptotext till klartext; där klartexten är det ursprungliga meddelandet och kryptotexten är en till synes meningslös och oläslig form av klartexten. För att utföra krypteringen och se till att endast valda parter kan återställa en kryptotext används gemensamma hemliga nycklar eller nyckelpar.

2.2 Krypteringsalgoritmen RSA

RSA är ett asymmetriskt [15] kryptosystem som utvecklades 1977 av Ron Rivest, Adi Shamir och Leonard Adleman, där akronymnen RSA kommer ifrån dess utvecklares efternamn.

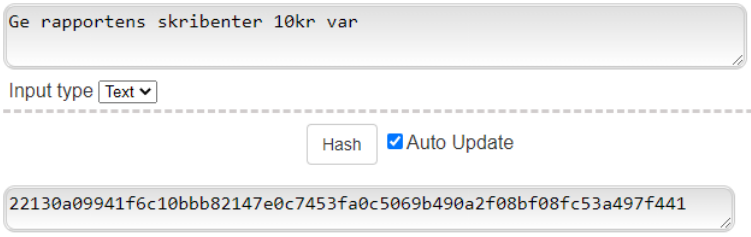
RSA använder sig av ett nyckelpar som består av en privat nyckel och en offentlig nyckel. Nycklarna är egentligen (ofta väldigt stora) primtal och respektive nyckel behöver sitt egna unika tal. Vi börjar med att välja extremt stora primtal p och q. p och q multipliceras för att få produkten n. Sen behöver också $t = (p-1)\cdot(q-1)$ beräknas. Därefter behöver ett primtal e beräknas som är relativt prima till t (t får inte vara delbart med e). Till sist beräknar vi talet d genom formeln $d \cdot e = 1 mod t$. Vår offentliga nyckel är n och e och vår privata nyckel är n och d.

Eftersom orimligt många beräkningar behövs för att finna den privata nyckeln ur den offentliga nyckeln (om ett tillräckligt stort primtal väljs) kan den offentliga nyckeln delas ut till vem som helst utan några större risker. Den offentliga nyckeln kan användas för att kryptera meddelanden medans de krypterade meddelandena bara kan återställas med den privata nyckeln.

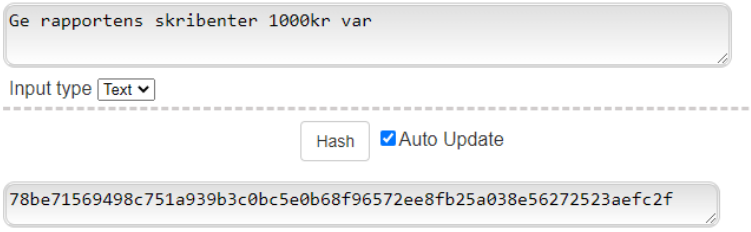
2.3 Hashning

En hashalgoritm komprimerar meddelanden av varierande längd till ett resultat med bestämd längd av tillsynes (men egentligen inte alls) slumpmässigt vald data [16]. Tanken är

att det ska vara nära omöjligt att manipulera indatan för att framkalla ett önskat hashresultat och därmed ska slutanvändare med stor säkerhet kunna bedöma data som opåverkad och intakt [17, s.3]. Eftersom varje möjlig unik indata inte kan få ett unikt kondensat är det viktigt att man inte systematiskt kan manipulera indata för att få ett önskat kondensat. Därmed ska man med hög säkerhet kunna verifiera att indatan inte har förändrats [18] om algoritmen är välutvecklad. Ett exempel på detta kan ses i figur 1.



(a) Originalmeddelandet och dess SHA256-kondensat.



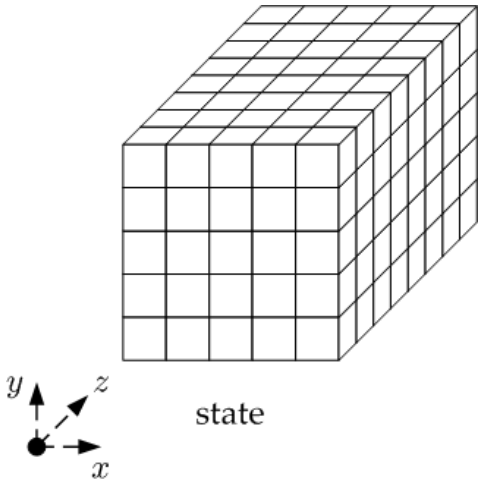
(b) Meddelandet med en lite högre pengarsumma, och dess SHA256-kondensat.

Figur 1: Två snarlika meddelanden vars SHA256-kondensat är mycket olika, beräknade med [19].

Som sagt används hashalgoritmer i stor utsträckning för att verifiera att meddelanden och filer är opåverkade. Och genom den principen kan den också användas för att bekräfta att en entitet har tillgång till samma nyckel som en annan vilket är varför hashning har varit av intresse i detta projektet.

2.4 SHA-3

SHA-3 är en algoritm som blev utsedd inom ramen för en tävling utlyst av den amerikanska organisationen NIST[20] för att ta fram en ny och säker hashalgoritm. Vinnaren av denna tävling var algoritmen Keccak som då blev SHA-3 [21][8]. Keccak grundar sig på en så kallad ”Sponge Construction” [22] och kan genom denna generera en hash av bestämd längd från en obestämd längd indata. Det finns olika versioner av SHA-3 och i detta projekt fokuserar vi på SHA3-256 som har fast längd utdata på 256 bitar [23].



Figur 2: Visuallisering av ett tillstånd i SHA-3 skapad av Team Keccak. Koordinaterna [x,y] kommer alltid vara 5 bitar långa men djupet av z-koordinaten bestäms av w i tabell 1.

Tabell 1: Tabellen visar värdena på variablerna som bestämmer hur SHA3-256 opererar[24, 23]. Notera att r och c tillsammans får samma längd som tillståndets längd b.

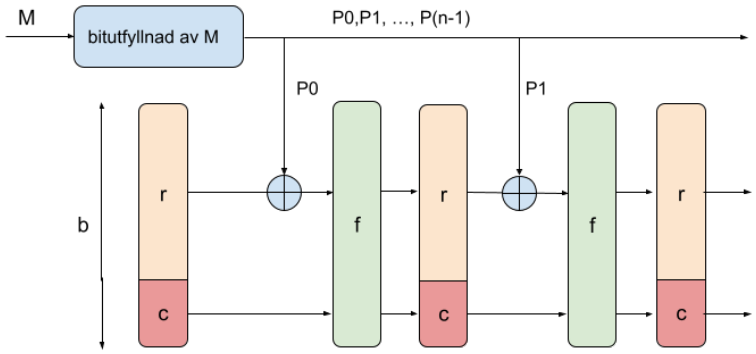
Konstant	Värde för SHA3-256
b	1600
r	1088
c	512
w	25
l	6
n	24
utdata	256

Keccak har en mängd parametrar som ändras beroende på vilken version man använder. Dessa parametrar är summerade i tabell 1 och beskrivna av Teamkeccak som [23]:

- b som är längden på permutationer [25] samt bredden av tillståndet i sponge construct.
- Varje tillstånd är formaterat som en 5x5 matris sedd i figur 2 där varje x-y koordinat har längden w. Detta är vad funktionen f kommer operera på. Men w förhåller sig till b så att $w = 25b$.
- r och c är bittakten och kapaciteten av tillståndet. Från början av hashfunktionen tar r värdet av tillståndet och $c = 0$. Därefter matas tillståndet $[r, c]$ in i funktionen f. Om f producerar utdata längre än r placeras resterande del av värdet i c. För SHA3-256 blir $[r, c] = [1088, 512]$.
- Utdatan av SHA3-256 är 256 bitar lång. Värdet bestäms från de 256 första bitarna av

r när sponge construct är färdig.

De specifika värdena på parametrarna kan ses i tabell 1. Men funktionen f består av fem operationer benämnda *theta, rho, pi, chi* och *iota* $(\theta, \rho, \pi, \chi, \iota)$ som beskrivs grundligt av Team Keccak [23]. Men alla opererar på tillståndsmatrisen i figur 2 och funktionen f producerar alltid utdata på formen $[r, c]$.



Figur 3: Bilden visar en sponge constructions absorberande fas. Den visar förenklat processen ett meddelande tar genom den absorberande fasen. Parametrarnas värde finns i tabell 1. Kortfattat delas meddelandet M i lika långa delar {P0, P1, ..., P(n-1)} som en efter en transformeras av funktionen f och sedan XOR:as med de andra transformerade delarna av meddelandet.

En sponge construction består av en absorberande fas, som ses i figur 3, som sedan följs av en klämmande fas [23]. I den absorberande utförs alla beräkningar och i den klämmande fasen avläses utdatan från det slutliga tillståndet. Eftersom en sponge construct tar indata av obestämd längd behöver den antagligen delas i mindre delar av lika längd. Låt oss kalla den ursprungliga indatan för meddelandet M. För att kunna delas upp i lika långa bitar enligt SHA-3:s standard behöver M bitutfyllas [24]. Detta sker genom att bitutfylla indatan M tills den har en längd som är en multipel av r, alltså en multipel av 1088 bitar. Bitutfyllnaden börjar och slutar på '1' med '0':or emellan. Därefter delas M i n delar som är r bitar långa benämnda {P0, P1, ..., P(n-1)} vilka används i sponge construct som beskrivet i figur 3.

Efter meddelandets bitutfyllnad och uppdelning hanteras delarna {P0, P1, ..., P(n-1)} i följd vilket kan ses i figur 3. Det hela börjar med tillståndet av längd b som initialt är satt till 0 och uppdelat i $[r, c]$ enligt tabell 1. Tillståndet kommer upprepat att XOR:as med delarna {P0, P1, ..., P(n-1)} av meddelandet M och därefter gå igenom funktionen f i n omgångar. Utdatan från funktionen f blir det nya tillståndet som XOR:as med nästa del av meddelandet M. Processen upprepas tills alla delar av meddelandet har gått igenom funktionen och returnerar ett tillstånd av längd b uppdelat i $[r, c]$.

I den klämmande fasen används de första 256 bitarna av tillståndets r-del som utdata för

hela sponge constructen samt hashfunktionen SHA3-256.

2.5 Pseudo-slumpmässiga tal och entropi

Ett problem inom kryptografi är när man känner till algoritmen finns det en möjlighet att man kan arbeta sig baklänges tills man hittar ursprungsvärdet. Därför använder många av dagens krypteringalgoritmer sig av slumpmässiga tal i någon del av sin process, till exempel för att generera nycklar eller för att bitutfylla ett meddelande. Det finns två sätt att generera dessa slumpmässiga tal, pseudo-slumpmässigt eller genom att samla på sig någon typ av information i form av entropi.

Problemet med att generera slumpmässiga siffror är att datorer är deterministiska, vilket innebär att alla tillstånd som kan komma att ske är baserade på ett utgångstillstånd. Detta leder till ett problem för kryptografin då om en illvillig part kan förutspå vilka siffror som kommer att användas i algoritmen som används för att kryptera data, kan denna part teoretiskt sett knäcka vilka nycklar eller bitutfyllning som kommer att användas. Att använda pseudo-slumpmässiga nummergeneratorer för detta är ett sätt att kringgå problemet. Dessa generatorer fungerar på så sätt att de skapar en till synes helt slumpmässig sträng av siffror med hjälp av ett slumpfrö. Denna sträng uppfyller alla krav på slumpmässighet med en nackdel: alla pseudo-slumpmässiga strängar av tal kommer förr eller senare upprepa sig. Ett exempel på detta är *Middle-Square metoden*. Algoritmen genererar en talföljd genom att ta ett fyrsiffrigt tal upphöjt i två, om talet inte är åtta siffror långt så fylls talet med ledande nollor, sedan används de fyra siffrorna i mitten upphöjt i två för att fortsätta algoritmen och detta fortsätter tills en önskad längd på siffersträngen uppnås. Problemet med en metod som *Middle-Square* är att om ett ”dåligt” slumpfrö används så kan perioden bli för kort för att vara användbar, till exempel så ger startvärdet 2916 en talföljd med följande siffror som används för att generera nästkommande siffror:

$$2916 \rightarrow 5030 \rightarrow 3009 \rightarrow 540 \rightarrow 2916 \rightarrow \cdots$$

vilket leder till en period på fyra omgångar.

Ett alternativ till att använda pseudo-slumpmässiga generatorer är att låta något som inte är deterministiskt bestämma vilken talföljd som skapas. I många fall så samlar operativsystemet in information på olika vis, knapptryckningar på tangentbordet, rörelse av musen eller till och med värden som samlas in från hårdvarudelar i maskinen. Denna typ av information kallas för entropi, från en term inom fysiken som beskriver mängden kaos i ett system. I Unix-liknande system så sparas denna information i två speciella filer, /dev/random och /dev/urandom.

2.6 MAC

MAC, kort för *message authentication code*, är ett litet stycke information som används för att autentisera ett meddelande [26]. Genom detta verifierar att information från en känd sändare faktiskt kom från sagd sändare.

2.7 VHDL och FPGA-kretsar

VHDL är ett hårdvarubeskrivande språk. Det utvecklades på 1980-talet på initiativ av den Amerikanska försvarsmakten för att bättre dokumentera digitala system [27]. Därtill ville man minska kostnaderna för militärens elektroniska system. Dokumentationen offentliggjordes 1985 [27]. Idag definierar IEEE 1076-2019 VHDL som läsbart för både maskin och människa, därav stödjer det utveckling, verifikation, syntetisering samt testning av hårdvarudesign [28].

FPGA-kretsar beskrivs av Xilinx som matriser av block vars logik kan programmeras och att blocken har programmerbara signalkopplingar [29]. Korten kan programmeras efter tillverkning och har alltså en dynamisk design till skillnad från ASIC. FPGA-kretsar har exempelvis använts av Perseverance, den Mars Rover som landade på Mars den 18 februari 2021 för att hitta mikroorganismer, för att kunna omprogrammera roboten och ge den nya funktioner under dess uppdrag [30].

2.8 PAM

PAM (pluggable authentication modules) är ett autentiseringssystem till Linux och flera andra Unix-liknande system. Ett program som begär användarautentisering kan anropa PAM som då anropar en viss PAM-modul specificerad av systemadministratören [13]. En PAM-modul är ett delat programbibliotek som innehåller vissa specifika funktioner för att kunna implementera autentisering via PAM. Själva autentiseringen sker på olika sätt beroende på modulen i fråga (exempelvis genom att begära ett lösenord). I detta projekt har vi skrivit en PAM-modul som utför autentisering med hjälp av utmanings- och svarskoder genom koddosan.

3 Metod

Detta kapitel beskriver det material som användes för att utveckla produkten och arbetsprocessen under projektet.

3.1 Material

Här anges de resurser som var nödvändiga för att slutföra projektet. Delar av materialet behövs endast om man ska skriva koden från början. Alltså behövs endast hårdvarukomponenterna införskaffas om man ämnar återskapa projektet med vår öppna källkod. Det är värt att notera att det finns många alternativ till programmen och hårdvarukomponenterna ute på marknaden.

3.1.1 Programvara och digitala verktyg

Här anges digitala program samt programbibliotek som användes för att utveckla koddosan och den tillhörande PAM-modulen:

1. Ett program för att syntetisera VHDL-designen till gate-level representation på ett FPGA-kort. Detta projekt använde *Vivado Design Suite* från Xilinx [31].
2. Ett program för att skriva, kompilera och simulera VHDL-kod. Detta projekt använde *ModelSim* från Intel®[32].
3. Linux-PAM för demonstration av användarflödet av produkten. [13]
4. Utveckling av PAM-modulen krävde en Linuxmiljö med PAM-biblioteket installerat. Den som arbetade i en Windows-miljö behövde installera *Subsystem for Linux* [33].
5. OpenSSLs crypto-bibliotek varifrån vi använde några delar i produkten [34].
6. Den dator som ska köra PAM-modulen behöver ha GCC för att kompilera C-kod [3].
7. GitLab användes under utvecklingen dels för att strukturera arbetet efter agila metoder, dels för att synkronisera arbetet med Git [35].

3.1.2 Litteratur

Nedan listar vi litteratur och dokumentation som var till hjälp för med att förstå vissa områden av projektet:

1. Alla i projektgruppen hade läst *Digital Design Using VHDL* [36]. Denna bok förklarar dels grunderna för programmering i VHDL, dels grunderna om logik-komponenter i elektroniska system.
2. Dokument från Keccaks skapare som förklarar hur Keccak (inkluderar SHA-3) kan implementeras i hårdvara samt mjukvara [37].

3. Dokumentation av programbiblioteket från OpenSSL [6]. Här hittar man koncisa definitioner och förklaringar av olika funktioner, bland annat hash- och krypteringsalgoritmer.

3.1.3 Hårdvara och komponenter

Här listar vi den hårdvara samt komponenter som behövdes för att konstruera koddosan.

1. Ett FPGA-kort av modellen *Nexys 4 DDR* från Digilent, men dess produktion har upphört [38]. Däremot säljs en kompatibel ersättare: *Nexys A7* [39].
2. En knappsats som i vårt fall var tillhandahållen av Chalmers. Det finns många alternativ, t.ex *Pmod KYPD* från Digilent [40].

3.2 Arbetsprocessen

Projektgruppens arbetsprocess var baserad på agila metoder. Det hade varit optimalt att ha korta möten varje vardag men eftersom medlemmarnas scheman skilde sig åt fick vi kompromissa. Istället för dagliga möten hållits varannan dag vid olika tider så att alla i gruppen hade möjlighet att komma. Under möten har gruppmedlemmar redogjort vad de arbetat med och åstadkommit, följt upp mål från tidigare möten och diskuterat vad deras fokus ska vara framtill nästkommande möte. På detta sätt stagnerade inte arbetet. Endast måndags- och fredagsmöten protokollfördes då onsdagsmötet var korta avstämningar. På fredagsmöten var gruppens handledare inbjuden.

Vi använde Gitlab för att organisera projektet. Varje sprint blev en milstolpe och mål blev ärenden på Gitlab. Allt utvecklingsarbete samt alla mötesprotokoll har laddats upp på Gitlab kontinuerligt.

Efter drygt halva projektets gång bestämde vi oss att skriva en statusrapport i Discord varje dag man arbetat eller planerar att arbeta, även på dagar vi inte hade möten. Här uppmuntrade vi varandra att skriva var och när vi kunde sätta oss och arbeta dagen efter vilket gav andra i gruppen möjligheter att pararbeta.

4 Genomförande

Här beskrivs märkvärdiga beslut som påverkade arbetets gång.

4.1 Undersökning för lämplig verifikationsalgoritm

Vi har undersökt vilken algoritm som bäst skulle åstadkomma våra mål gällande identifikation av svarskoder från koddosan. I undersökningen behandlades först och främst säkerheten hos ett antal populära krypteringsalgoritmer och fysiska möjligheten att implementera dessa på FPGA-kort. De kryptografiska algoritmer som undersökts är krypteringsalgoritmerna *Advanced Encryption Standard* (AES) [41], *Elliptic Curve Cryptography* (ECC) [42], *Rivest-Shamir-Adlerman* (RSA) [43] och *ChaCha20* [1].

Dessa valdes då de ansågs vara populära och utförligt undersökts under åren. Men endast en behövs, således behövde de jämföras. Vi bestämde först att implementera RSA, men insåg senare att hashalgoritmer verkade lämpligare för det tänkta systemet.

4.2 Byte från krypteringsalgoritm till hashalgoritm

Den ursprungliga planen var att implementera RSA i mjuk- och hårdvara. Beslutet att använda RSA grundades i tanken att algoritmen är relativt simpel jämfört med andra algoritmer, men också på grund av den omfattande kunskapen och tidigare arbeten som finns tillgänglig på internet. Senare insåg vi att en hashalgoritm eller kontrollsumma behövdes för att trunkera resultatet på ett lämpligt sätt. Då bestämde vi oss istället för att planera om vår utveckling och använda en hashalgoritm som verifikationsmetod. Keccak valdes som denna hashalgoritm då det är den senaste medlemmen i SHA-familjen av standarder. Den anses vara ”oknäckt” och bedöms vara det i många år till [8]. SHA-2 används spritt än idag men SHA-3 planerades som en ännu säkrare algoritm som ska kunna ersätta SHA-2 när det eventuellt behövs.

Genom att använda hashalgoritmen har vi konstruerat ett MAC-system. Mjukvaran producerar en utmaningskod, därefter utförs en hashning med användarens hemliga nyckel som konkateneras med utmaningskoden i både mjukvaran och koddosan. Koddosans utdata (hash) matas sedan in till mjukvaran som verifierar om koddosans resultat är korrekt eller ej. Både mjukvaran och hårdvaran ska alltså genomföra identiska operationer och resultaten bör vara identiska eftersom hashning är deterministisk. Denna metod har härletts från vanliga sätt att verifiera att information härstammar från en specifik person eller entitet (se avsnitt om MAC 2.6).

5 Systembeskrivning

Här beskrivs systemets beståndsdelar och funktionalitet i sin helhet. Kort sagt är systemet ett samspel mellan den externa koddosan och en godtycklig mjukvara som har möjligheten att verifiera att koddosan använder sig av en hemlig nyckel som endast en människa (eller entitet) har tilldelats i samband med koddosan. Kommande avsnitt 5.2 beskriver alltså designen rörande själva koddosan medan avsnitt 5.3 beskriver mjukvarans funktionalitet.

5.1 Tolka indata från knappsatsen

För att bäst simulera det teoretiska användarflödet använder vi en extern knappsats (bilaga B) och en segmentdisplay på FPGA-kortet för att demonstrera systemets användarflöde.

5.2 Hårdvara

Hårdvaran består av ett samspel av olika VHDL-moduler. Dessa moduler är sammanbundna i en källkodsfil som använder alla moduler som komponenter, där in- och utdata kopplas för att realisera systemet (se bilaga A.2 för interaktionsschema).

5.2.1 Keccak-padding

Denna modulen ansvarar för att beräkna hashresultat med SHA-3. Filen är baserad på källkod som är hämtad från grundarna av Keccak och finns beskriven i avsnitt 3.1. Motivet att hitta en färdig grundimplementering dök upp efter vi uppskattade att utvecklingstiden av Keccak från grunden hade långt överstigit vår utvecklingstid. Detta medförde dock nya problem; exempelvis utförs ingen bitutfyllnad, utan indatan antas i modulen redan vara färdig för beräkning. På grund av detta stämde ursprungligen inte hashresultaten från mjukvara och hårdvara överens och orsaken var först inte känd. Lösningen på detta problem diskuteras vidare i avsnitt 7.3.

För att underlätta produktion av ett större antal koddosor har vi utvecklat en funktion som läser in nyckeln från en textfil vid kompilering. Vid syntetisering av VHDL-koden kan då nya nycklar placeras i en vald sökväg så den unika hemliga nyckeln bakas in i varje ny koddosa. Som organisation hade man då enklare kunnat massproducera unika koddosor.

5.2.2 Knappsats

För att ta reda på vilken knapp (om någon) som är nedtryckt skickas en fyra-bitars one-hot signal till knappsatsen. En one-hot signal är ett antal nollor och en enstaka etta som i detta fall representerar vilken kolumn man för tillfället läser av beroende på vilken position ettan befinner sig i. Följt av detta läser man av en fyra-bitars one-hot insignal som

representerar vilken kolumn (om någon) som är aktiv. Om en rad visar sig vara aktiv kan man genom diagrammet i bilaga B reda ut vilken knapp som är nedtryckt.

När knapptrycket identifierats lagras det avlästa värdet i ett register och siffran visas i segmentskärmen. Detta görs genom en switch-sats som täcker alla relevanta kombinationer av kolumner och rader. Om en tvåa tryckts ner på knappsatsen skrivs värdet 2 i binär form till ett register som representerar den totala summan av inmatningen, därefter skrivs ”2” ut till segmentskärmen. Om användaren trycker på knappen ”B” subtraheras 2 från summaregistret och tvåan avlägsnas från segmentskärmen.

5.3 Programlogik

Mjukvaran som gruppen har utvecklat består av en PAM-modul som med hjälp av OpenSSL-biblioteket verifierar en användares identitet. Det finns fyra olika program i mjukvarudelen som beskrivs i detta kapitel. Översiktligt hanteras nycklarna av program som körs med särskild behörighet genom att de installeras som *setgid* [44]. Detta försäkrar att endast användare med rätt behörighet kan läsa av nycklarna som är lagrade i systemet. Nedan beskrivs i följande ordning PAM-modulen, dess hjälpprogram, ett nyckelhanteringsprogram och ett avlusningsprogram.

5.3.1 pam_dosa.so

PAM-modulen definerar de två funktionerna som krävs av en autentiseringsmodul: *pam_sm_setcred* och *pam_sm_authenticate*. Den förstnämnda används för att ge applikationen tillgång till de användaruppgifter som modulen har fått tillgång till. Denna modul har däremot ej utvecklats med några sådana uppgifter och därmed returnerar modulen omedelbart utan att göra något vid ett sådant funktionsanrop. Den andra funktionen, *pam_sm_authenticate*, används för att autentisera användaren vilket sker genom att modulen startar hjälpprogrammet *dosa_chkcode* som exekveras med en särskild behörighetsnivå för att kunna läsa av nyckeldatabasen. Hjälpprogrammets returvärde bestämmer modulens returvärde.

5.3.2 dosa_chkcode

När en användare finns i systemet kan ett försök till autentisering ske. Databasen som sparar användarnas information är en klartextfil med en rad per användare, där en rad består av användarnamnet följt med ett kolon och sedan användarens nyckel.

För att få tag på en användares nyckel används *get_key*-funktionen. Dess syfte är att hitta den användare som försöker logga in i systemet och sedan fylla i en nyckel-struktur med den hämtade nyckeln och längden på denna nyckel. Om en användare inte finns i databasen leder det till att hela programmet ger tillbaka returvärdet *PAM_USER_UNKNOWN* och avslutar inloggningsförsöket. Om nyckeln har en inkorrekt längd kan detta leda till att hashningen utförs på ett felaktigt sätt och kan därmed leda till en svaghet i systemet. Därför hanteras en felaktig nyckellängd med att informationen i nyckel-strukturen frigörs och

att värdet *PAM_AUTH_ERR* returneras, vilket betyder att ett fel i autentiseringsprocessen skett och inloggningsförsöket även i detta fallet avslutas.

När en giltig användare försöker autentisera sig mot systemet så genereras en åtta-siffrig utmaningskod och skrivs till terminalen som ska använda denna kod i sin personliga koddosa för att få sin korrekta svarskod. Detta sköts via *try_authenticate*-funktionen som med hjälp av ett par hjälpfunktioner hanterar all interaktion med användaren.

För att generera utmaningskoden så använder sig programmet av *getrandom*-funktionen i C för att generera en slumpmässig utmaningskod på 8 siffror.

```
int get_code(uint32_t *code_ret) {
  if (getrandom(code_ret, sizeof(*code_ret), GRND_NONBLOCK) != sizeof(*code_ret))
    return -1;
  // Truncate the first 4 bytes of the message to 26 bits
  *code_ret &= 0x3ffffff;
  return 0;
}
```

Getrandom använder urandom som källa för entropi [45] för att generera sina siffror, vilket är den rekommenderade metoden för att skapa kryptografiskt säkra pseudoslumptal för Linuxsystem [46]. Funktionen returnerar antalet bytes som kopierades till *code_ret*, vilket ger oss en möjlighet att kontrollera att det fanns tillräckligt med entropi i källan för att generera önskat antal bytes. Ett problem med *getrandom* är när entropi-poolen inte har initierats rätt så kommer *getrandom* att blockeras, detta löses genom att sätta flaggan GRND_NONBLOCK. Den ser till att vid eventuell blockering så returneras -1, vilket gör att vi kan jämföra returvärdet med storleken av *code_ret* för att garantera att de är lika stora. I detta fallet så krävs endast 4 bytes för vårt ändamål vilket innebär att if-satsen kommer att klara av jämförelsen så länge urandom är initierad eftersom *getrandom* garanterar att läsningar upp till 256 bytes lyckas med en initierad entropikälla [45].

Om vi skulle använda alla 4 bytes av utmaningskoden som genererats utan att hantera det på något vis så skulle vi kunna få bitmönster som är mycket större än åtta siffror när det tolkas decimalt, därför används raden $*code\_ret\& = 0x3FFFFFF$; som genom en bitvis AND-operation sätter bitmönstret till 26 användbara bits. Anledningen till detta är att 2^{26} är 67108864_{10} vilket garanterar att ett tal som är längre än åtta siffror inte kan genereras.

Dessa siffror hashas sedan med hjälp av SHA-3 algoritmen tillsammans med användarens nyckel för att beräkna den förväntade svarskoden som användaren ska få tillbaka från koddosan om användaren är den som den utger sig för att vara. Vid matchande svarskoder returnerar *try_authenticate* PAM_SUCCESS till *main*-funktionen och PAM-modulen vet därmed att inloggningen har lyckats. Om svarskoderna inte matchar så returneras PAM_AUTH_ERR och räknaren som håller reda på antalet inloggningsförsök som är kvar dekrementeras. Efter en felaktig svarskod genereras en ny utmaningskod av *try_authenticate*. Antalet försök som en användare får innan *main*-funktionen returnerar PAM_AUTH_ERR till PAM-modulen kan konfigureras vid kompileringstid.

5.3.3 dosa_manage_key

Det här programmet används för att generera, ta bort, uppdatera eller visa nycklar för användare i systemet. Användare kan fritt använda programmet för att hantera nyckeln för sitt egna konto. *root*-användaren, eller användare som är medlemmar i en utsedd grupp, kan även hantera nycklar tillhörande andra konton.

Programmet har fem olika lägen som specificeras av användaren i kommandotolken:

- *insert*: Genererar en nyckel för den specificerade användaren. Om användaren inte redan har en nyckel förs den nya nyckeln in i nyckeldatabasen och skrivs ut till en fil. Det finns dessutom inställningar som gör att nyckeln läses in från en fil istället för att genereras.
- *replace*: Gör samma sak som *insert*-läget förutom att den nya nyckeln förs in i databasen oavsett om användaren redan har en nyckel.
- *delete*: Tar bort den specificerade användarens nyckel från databasen.
- *print*: Skriver ut den specificerade användarens nyckel i kommandotolken.
- *write*: Skriver ut den specificerade användarens nyckel till en fil.

Filformatet som nyckeln skrivs ut i består av 12 rader med 16 hexadecimala siffror. En sådan fil kan användas för att syntetisera en bitstream som innehåller användarens privata nyckel.

5.3.4 hash_with_keyfile

Detta program används endast i utvecklings- och avlusningssyfte för att simulera verifikationsprocessen. Användaren specificerar själv utmaningskoden som används samt en fil där nyckeln lagras. Programmet hashar den givna koden och skriver ut hela tillståndet innan och efter hashingen, samt den trunkerade svarskoden som hårdvaran bör generera om den är syntetiserad med samma nyckelfil.

6 Resultat

Först presenterar vi systemets interaktionsflöde, följt av testresultat som visar att produkten fungerar. Slutligen går vi igenom alla moduler som skrivits till hårdvaran samt mjukvaran.

6.1 Verifikationsprocessen

Processen sker enligt följande paragraf och beskrivs principiellt i interaktionsflödet i bilaga A.2.

För att säkerställa funktionaliteten av systemet genererar vi slumpmässigt en nyckel som både mjuk- och hårdvaran ska använda. Denna nyckeln placeras först i en textfil så mjukvaran kan läsa av den och syntetiseras i ett fält av register i samband med resten av designen så att nyckeln är hårdkodad i kortet. Användaren ska mata in sitt användarnamn och skärmen ska visa en slumpmässigt genererad utmaningskod. Hashen utifrån nyckeln och utmaningskoden ska vid det här laget redan vara beräknad av mjukvaran som nu väntar sig indata. Användaren ska läsa av utmaningskoden och mata in den till kortet genom knappsatsen när kortet är startat med designen på plats. När användaren matat in koden och tryckt på knappen som signalerar att man är färdig bör ett åttasiffrigt nummer visas på skärmen. Detta ska matas in av användaren till PAM-skärmen. Om alla steg följts korrekt bör mjukvaran verifiera att summan stämmer överens med de initiala beräkningarna och logga in användaren. Om användaren matat in antingen utmaningskoden eller svarskoden fel bör mjukvaran säga ifrån och hindra användaren från att loggas in.

Produktens verifikationsprocess kan ses i bilaga A.2 och fungerar på följande sätt:

1. Mjukvaran tillhandahåller en utmaningskod bestående av åtta siffror som matas in i koddosan av användaren.
2. Koddosan genererar en svarskod bestående av åtta siffror baserat med utmaningskoden, koddosans nyckel samt krypteringsalgoritmen. Samtidigt gör mjukvaran samma beräkningar och genererar en svarskod.
3. Hårdvarans svarskod anges till mjukvaran som jämför båda svarskoder. Om de är likadana verifieras användaren.

För att denna verifikationsprocess ska fungera krävs två typer av funktionalitet:

- Koddosan måste ur en given kod kunna generera en svarskod som inte går att ta fram utan tillgång till nyckeln.
- Tjänstens verifikationssystem måste kunna beräkna om svarskoden är den rätta eller inte.

6.2 Konfiguration av vår Keccak-implementering

Bakgrundsfakta för SHA-3 finns beskrivet i avsnitt 2.4. Summerat är parametrarnas värde för koddosans SHA-3-instans:

- Längden på tillstånd och permutationer, $b = 1600$.
- Längden på element av tillstånd, $w = 64$.
- Antalet rundor av *sponge construtions*, $n = 24$.
- Bittakten $r = 1088$.
- Kapacitetsvärdet $c = 512$.

6.3 Testresultat

Vi genererade en ny nyckel på 96-bytes (se bilaga D.1 för nyckelns värde). Nyckeln syntetiserades med designen i Vivado och bitstreamen överfördes till FPGA-kortet. Vi fyllde i användarnamn och lösenord av en Linux-användare i en virtuell maskin med Arch Linux med PAM-modulen och mjukvaran installerad. PAM pausade inloggningen och visade en utmaningskod (se bilaga D.2 för bild). Vi matade in denna på knappsatsen som är kopplad till FPGA-kortet och fick ett åttasiffrigt resultat (vår hash). Detta matades sedan in som svarskod i PAM-modulen som loggade in oss (se bilaga D.3 för bild). För säkerhetens skull upprepade vi denna process fast avsiktligt matade in felaktiga nummerföljder vilket ledde till att PAM-modulen nekade inloggningsförsöket (se bilaga D.4).

Denna process upprepades ännu en gång med en ny slumpgenererad nyckel och åter visade sig systemet fungera precis som väntat. Vi testade även med nycklar som avsiktligt skiljde sig mellan koddosan och den virtuella maskinen (se bilaga D.1 för koddosans nyckel och bilaga D.5 för mjukvarans nyckel vid det här testet). Koddosans svarskoder blev nu nekade av mjukvaran (se bilaga D.6 för bild). Systemet fungerade åter precis som väntat. Vi testade också att generera koder från en dosa med felaktig nyckel. Alla försök misslyckades och därmed anser vi prototypen fungera som förväntat.

Inloggningsprocessen har även testats med olika kontrollvärden som tillåts i PAM [47]. Dessa värden bestämmer betydelsen av att en moduls autentisering misslyckas. Till exempel kan en misslyckad autentisering bara spela någon roll om det inte finns några andra moduler tillgängliga, annars kan en lyckad autentisering hos modulen krävas. Testning av kontrollvärdena skedde innan nyckelhanteringsprogrammet hade skapats, och ett misstag i den manuella nyckelhanteringen ledde till att vi blev utelåsta från den virtuella maskinen och behövde återställa den till ett sparat läge. Utelåsning när vi inte har tillgång till nyckeln var ju självklart det vi förväntade oss, så det här testet ser vi som lyckat. Det ledde också till utvecklingen av nyckelhanteringsprogrammet eftersom vi efter insåg att mjukvaran bör ha ett smidigt sätt att hantera nycklar.

7 Diskussion

I detta avsnitt diskuteras resultaten av projektet samt vad som hände under processen. Det som diskuterades under projektet var till exempel vilken krypteringsalgoritm som produkten skulle använda men under utvecklingen uppkom dels problem med implementeringen och dels insåg vi att en hashalgoritm hade varit inte bara lämpligare men också enklare att använda för att uppnå samma resultat. Vi diskuterar även lämpliga licenser för den öppna källkoden. Därefter följer en diskussion om etik och vidare sammanfattar vi vilka möjliga utvecklingar som kan göras för projektet.

7.1 Verifikationsalgoritmer

Under projektets gång lades mycket tid på att införskaffa information om de olika krypteringsalgoritmerna och verifikationsmetoder som vi potentiellt kunde använda. Eftersom vi i slutändan beslöt oss för att använda SHA-3 blev den utförda utvecklingen av krypteringsalgoritmer obsolet.

Trots detta anser vi inte att tiden var bortslösad. Koncept och informationspunkter som vi lärde oss gav en bättre förståelse för kryptografi i allmänhet. Därav kunde vi bättre avgöra vilken slags säkerhet vår koddosa skulle kräva.

En viktig faktor som skiljer dessa algoritmer åt är huruvida de använder sig av symmetriska eller asymmetriska nycklar. AES, ChaCha20 använder symmetriska nycklar medan ECC och RSA använder sig av asymmetriska nycklar. Principiellt använder vi symmetriska nycklar med vår slutliga algoritm, SHA-3, då nycklarna på både mjukvaru- och hårdvarusidan är samma. Algoritmerna som använder asymmetriska nycklar är inom vissa användningsområden lämpligare än de som bygger på symmetriska nycklar. Om nycklarna hos ett verifikationssystem som använder asymmetrisk kryptering hamnar i fel händer krävs nycklarna från både hårdvaru- och mjukvarusidan för att verifikation ska ske. I verifikation med koddosan skrivs dock åtta siffror in- och ut från dosan, vilket inför stora begränsningar för asymmetrisk kryptering då krypteringens blockstorlek inte kan överskrida de åtta siffrorna. För RSA, där blockstorleken är kopplad till nyckelstorleken, innebär detta att nycklarna som kan användas i koddosan är väldigt lätta att knäcka.

7.2 Vilken licens?

Vår avsikt i det här projektet var att koden för program och hårdvara skulle kunna spridas och modifieras genom en öppen källkod. Detta innebär att programkoden får spridas och modifieras av vem som helst vilket möjliggör en typ av samarbete inom öppen programvara där alla kan bidra och dra nytta av produkten. Det finns flera licenser anpassade för öppen källkod; vi har valt licensen GPL som är en av dessa [48]. GPL innebär förutom att vem som helst får sprida och modifiera koden att den som tillhandahåller kompilerade program även behöver ha möjligheten att tillhandahålla källkoden till dessa program. GPL innebär även att modifikationer av programkod skyddad enligt GPL samt program

som innehåller kod taget ur ett program skyddat enligt GPL också måste använda GPL, om det på något sätt offentliggörs [49]. Detta innebär att programvara skyddad enligt GPL är fri att sprida och modifiera och att dessa friheter kommer överföras till eventuell annan programvara härledd ur den GPL-skyddade programvaran. Detta bidrar till att alla kan dra nytta av det arbete som läggs på vidare utveckling av produkten.

Användningen av OpenSSL inom mjukvaran inför dock problem med licensieringen. OpenSSLs licensiering består av två licenser som inte är kompatibla med GPL [50]. Detta innebär att distribution av mjukvara som är statiskt eller dynamiskt länkad till OpenSSL inte tillåts under vanliga omständigheter. GPL har ett systembiblioteksundantag som tillåter länkning till inkompatibla bibliotek om alla användare av mjukvaran rimligtvis kan förväntas ha tillgång till biblioteket [51]. Undantagets applicering på OpenSSL-biblioteket skiljer mellan olika Linux-distributioner. Fedora är en av distributionerna som anser att OpenSSL är ett systembibliotek [52], men de är tydliga om att de inte talar för någon annan. För att uppnå kompatibilitet oavsett användarens Linuxdistribution föreslår OpenSSL-projektet att ett länkningsundantag läggs till i mjukvarans licens [53]. Vi har valt att släppa koden utan något sådant undantag, främst för att vi inte distribuerar någon kompilerad form av mjukvaran. Dessutom anser vi att OpenSSL ingår i tillräckligt många Linuxdistributioner för att möjliga användare kan förväntas ha tillgång till biblioteket.

7.3 Implementeringsproblem

Under utveckling uppstod ett antal insikter vilket ledde till att slutprodukten inte helt motsvarar ursprungsplaneringen. Eftersom vi hade regelbundna möten under arbetstiden och uppdaterade varandra om problem som tog mer än ett par dagar att lösa så kördes inte arbetet fast, förutom en etapp i mitten av utvecklingen av hårdvarans implementering av RSA.

7.3.1 Mjukvara

Det fanns från början en hel del implementeringar och bibliotek som vi kunde dra lärdom av när tanken var att implementera en krypteringsalgoritm som AES eller RSA i ett programspråk som C. Därför gick arbetet i mjukvara (relativt till hårdvara) väldigt snabbt.

Vi beslöt oss ursprungligen att implementera RSA och huvudsakligen autentisera användaren genom den algoritmen. Beslutet grundade sig på åldern, populariteten och den enorma kunskapsgrunden som redan finns om algoritmen. Det är etablerat att RSA inte mäter sig med hänsyn till effektivitet till algoritmer såsom ECC och AES [54], men om man använder nycklar som är tillräckligt stora anses det fortfarande vara säkert [55]. Dessutom hade ingen i gruppen omfattande erfarenhet eller kunskap gällande krypteringsalgoritmer och RSA ansågs vara enklare att förstå och implementera.

När väl planeringen och utvecklingen börjat insåg vi att vår logik var bristfällig; vår ursprungliga tanke var simpel: att verifiera att båda parter innehar samma nyckel. Hur skulle vi åstadkomma detta genom RSA? Antingen genom kryptering, dekryptering, eller båda.

Vi bestämde oss först att kryptera datan i mjukvaran och sedan dekryptera datan i hårdvaran. Men då insåg vi att resultaten blev för långa för att matas in och ut från hårdvaran, så vi behövde trunkera den krypterade datan. Vi valde att utföra en modulooperation för att korta ner resultatet till åtta siffror. Detta innebar att vi endast kom att utnyttja en publik nyckel för att kryptera meddelandet eftersom det inte går att dekryptera ett ändrat meddelande, och återställningen av chiffret utelämnades helt och hållet. Inte långt efter insåg vi att detta flöde liknar en HMAC och att användning av en hashalgoritm är mer lämplig.

När vi sedan bestämde oss för att byta från RSA till en hashalgoritm så gjorde vi en del utredningar om vad som skulle kunna passa och vilka tidigare utvecklade bibliotek som fanns tillgängliga. Vi bestämde oss för att utveckla en implementering som använde HMAC med SHA-3 som hashalgoritm. Efter en grundläggande implementering av HMAC hade skrivits klart i C så hade gruppen som helhet fått lite bättre förståelse inom ämnet. Slutligen insåg vi att HMAC verkade vara överflödigt, så vi skrev om programmet för att endast använda SHA-3. Tack vare OpenSSL-biblioteket krävde detta inte en stor insats.

Det uppstod även problem att installera mjukvaran på olika system. Det berodde på att vissa Linuxdistributioner, som Debian och Ubuntu, använder en *multiarch*-arkitektur [56] i syfte att förenkla installation och körning av program som är kompilerade för olika datorarktitekturer. För att lösa detta utvidgades mjukvarans Makefil [57] till att först kontrollera om det finns en *multiarch*-mapp på systemet; om den inte finns används en standardmapp. Med ett mer avancerat byggsystem hade problemet troligtvis inte uppstått från första början, men erfarenhet av andra byggsystem än *make* saknades i gruppen och därför fortsatte gruppen att endast använda *make* för att hantera kompilering.

7.3.2 Hårdvara

Utöver de anledningarna beskrivna i avsnitt 4.2 - gällande byte av algoritm - stötte vi på ett flertal problem under utvecklingen. En av dem var att vi inte visste hur många bitar data vi behövde för att utföra moduloaritmetiken som krypteringsalgoritmen RSA kräver. Vid sökning hittade vi Montgomery Modular Multiplication algoritmen [58] som kunde lösa detta problemet. Men innan utvecklingen hade nått ett meningsfullt stadie planerade vi om arbetet för implementationen av SHA-3 istället, så Montgomery Modular Multiplication förblev oanvänd.

Som tidigare nämnt i avsnitt 5.2.1 så uppstod det ett problem i jämförelsen mellan mjuk- och hårdvarans hashresultat tidigt under implementeringsfasen. Detta berodde på att hårdvarans testvektorer använde sig av 10^*1-bitutfyllnad [59] enligt Keccaks definition medan mjukvaran först konkatenerar ‘01’ med meddelandet innan det fylls ut med 10^*1-bitutfyllnad enligt SHA-3 standarden. Dessutom bitutfyllde hårdvaran inte indatan, utan förväntade sig att indatan redan skulle vara bitutfylld, vilket ledde till ytterligare komplikationer innan gruppen insåg det.

Efter att hashingen fungerade i både mjukvara och hårdvara återstod problemet att skriva

information till skärmen på FPGA-kortet. Detta medförde ett par problem som grundades i konverteringen av decimaltal till och från hexadecimala tal, vilket innebar att vi behövde multiplicera (vid inmatning) och dividera (vid utmatning) vår data. Vi löste detta genom att lägga till ett minne som lagrar inmatningen så att man kunde utföra multiplikation på varje siffra. Detta fungerar eftersom man kan dela upp ett tal i sina ensiffriga värden, exempelvis är

$$3540_{10} = 3 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 0 \times 10^0$$

För utmatningen användes åtta stycken 4-bitars tal som vardera representerade en av de åtta ensiffriga talen. Processen som användes för att beräkna detta kan enklare förstås med ett flödesdiagram (se bilaga A.1).

Att utveckla en koddosa innebär inte nödvändigtvis en saknad av eventuella etiska problem. Eftersom vi inte erbjuder en komplett tjänst, bara grundstenen, kommer kanske den som använder vår källkod vidareutveckla systemet till något som anses problematiskt. Den centrala frågan blir om det vi, utvecklarna, som är skyldiga om någon gör något "dåligt" eller olagligt med vår produkt. Detta är bara en tanke, fler konkreta tankar finns i följande delavsnitt.

7.4 Etiska implikationer av att designa ett skyddat system

Vid utveckling av skyddade system så hänger mycket på att man som utvecklare har tillit i sina användare om den produkt som man vill att de ska använda faktiskt är säker. En risk som alltid finns när användare interagerar med ett system där källkoden inte är känd är att utvecklarna implementerat en bakdörr i systemet. Det kan handla om en utvecklare som har illvilliga mål eller att en stat vill kunna komma åt information utan att behöva be om den. Att utveckla sitt system och därefter släppa källkoden öppet kommer snabbt leda till att någon av dess användare upptäcker om det med mening har introducerats några svagheter i systemet.

Men vår design består bara av ett par koncisa källkodsfiler och utvecklades med ursprungligen minimal kunskap om centrala ämnen för projektet. Därmed hade illvilliga krafter antagligen lika enkelt som oss kunnat bygga deras egna implementation.

7.5 Avsiktliga fel i kompilatorer och användarintegritet

Eftersom koden vi skrivit antingen måste syntetiseras eller kompileras är det möjligt för organisationen bakom kompilatorn att lägga in bakdörrar i koddosan eller PAM-modulen. I fallet av mjukvara hjälper det inte i det extrema fallet att ha tillgång till källkoden för kompilatorn eftersom även denna måste kompileras av den potentiellt illvilliga kompilatorn som kan lägga in samma fel i den nya [60]. I vårt fall använder vi oss av GCC för kompilering av PAM-modulen och Vivado för programmering av FPGA-kortet. Vivado har inte öppen källkod så även utan en äventyrad kompilator kan vi inte undgå eventuella bakdörrar från Vivado.

En bakdörr i koddosan vore dock ganska meningslös. För en felaktig implementering av SHA-3 skulle det krävas att PAM-modulen gjorde samma felaktiga och osäkra hashning vilket skulle kräva att kompilatorn som användes till denna ändrade koden i PAM-modulen på samma sätt som Vivado ändrat koden i koddosan. Detta skulle innebära att en stor internationell konspiration är på gång, vilket inte hade varit helt och hållet omöjligt; men frågan som behöver ställas då är vad som kan vinnas från en sådan invecklad och organiserad insats.

Om PAM-modulen och/eller mjukvaran kan manipuleras kan man enkelt manipulera processen, till exempel genom en specifik svarskod som alltid fungerar oavsett utmaningskoden. Säkerheten ligger slutligen på PAM-modulen vilket gör att det inte finns något rimligt sätt att äventyra endast koddosan.

En annan potentiell säkerhetsrisk ligger i nyckeldistributionen. Det hade varit möjligt för ett skadligt program att kapa nyckelgenerationen och tvinga användningen av ett begränsat antal kända nycklar. På så sätt hade den illvilliga parten kunnat framkalla svarskoder som egentligen bara ägarna av koddosorna hade kunnat producera. I fallet av en okänd *remote administration tool* [61] eller ett *rootkit* [62] kan man försöka identifiera och ta bort dem eller ominstallera sitt system. Sen kan också exempelvis pseudoslumpmässiga nummergeneratorns binära exekveringsfil vara äventyrad. För att förebygga dessa & dylika attacker hade man kunnat testa utdatan för programmet ansvarigt för generationen av nycklar. Man kan göra *test för slumpmässighet* [63]; dessa ämnar att tillsammans beräkna frekvensen av alla siffror men också olika mönster. En upprepande sifferföljd 123456789... hade godkänts av den mest enkla frekvenstesten, men den är inte särskilt slumpmässig; därför används också bland annat korrelationstester [64] för att avgöra hur förutsägbar den är.

7.6 Tillit hos användaren

Eftersom systemet består av flera komponenter som interagerar med varandra finns det utrymme för säkerhetsbrister. Om vi antar att dosans logik inte syntetiseras av en kompilator som har manipulerats för att införa brister i systemet är det genom källkoden man avgör varken den är säker eller inte. Den använder sig endast av kända beräkningar (såsom med SHA-3) och denna funktionalitet kan bekräftas. Hur nyckeln hashas tillsammans med meddelandet och huruvida det anses vara en god lösning går också att bedömas av den som är villig.

Användaren kan åtminstone vara säker på att dosan inte kan bli hackad på distans. Detta grundar sig på att dosan, som den är tänkt, inte ska ha någon internetuppkoppling. Om vi också förutsätter att dosan inte blir fysiskt stulen kan antagandet göras att den hemliga nyckeln förblir säker.

Trots att dosans design är offentlig behöver inte systemet den autentiseras med vara det. Vår PAM-modul finns tillgänglig för nedladdning men en organisation som eventuellt integrerar vår produkt kommer antagligen utveckla sitt egna autentiseringssystem anpassat

för tjänsten de erbjuder; och denna tjänstens funktionalitet kan förbli hemlig. GPL påstår endast att verk som härletts från ett annat GPL-licensierat verk också måste publiceras med den licensen; detta villkoret omfattar inte det komplementerande systemet som ska byggas för säkerhetsdosan. Visserligen kan då användaren lita på säkerhetsdosan där vår design är i verk, men ändsystemet går inte nödvändigtvis att inspektera då det är utom vår kontroll.

7.7 Vidareutveckling

I nuläget så finns det inget sätt för koddosan att veta att den är i rätta händer. Med lite vidareutveckling skulle en personlig PIN-kod kunna lagras i varje koddosa så det blir svårare att stjäla en koddosa och felaktigt identifiera sig som ägaren. Detta skulle även kunna kombineras med att koddosan inaktiveras efter ett satt antal felaktiga inloggningsförsök, vilket hade förhindrat råstyrkeattacker mot PIN-koden. Sådana begränsningar finns i många existerande säkerhetsdosor idag och därför är det viktigt att vår produkt inte saknar sådana väsentliga egenskaper.

7.7.1 Koddosan

VHDL-koden är nu skräddarsydd för en specifik modell av FPGA-kort: Digilents Nexys A7 [65]. Koden för att läsa av pinnar och att mata ut information till segmentskärmen är mer eller mindre sammanvävd med den mer universella koden relaterad till Keccak-implementeringen. En intresserad part hade då behövt städa upp och skriva om mycket av koden för att interagera med annan hårdvara. För att underlätta detta arbetet hade vi exempelvis kunnat dela upp våra VHDL-moduler mer effektivt.

VHDL-koden använder sig av en implementering av SHA-3 som inte är specifikt konstruerad för att minimera storlek tagen på kortet eller för att maximera beräkningshastighet. Att lägga till sådana implementeringar (vilka finns tillgängliga från *Keccak Team*) skulle vara fördelaktigt. Dels då en potentiell kund skulle ha fler valmöjligheter, samt att kostnaden för produktion skulle kunna minskas.

7.7.2 Mjukvara

Mjukvaran som utvecklades för projektet var menad som ett bevis för att produkten går att integrera och som ett sätt för oss att demonstrera produkten i sin helhet. Den skulle kunna användas för att bättre identifiera användare i Linux-system. Därtill skulle mjukvaran kunna vara användbar för att implementera SHA-3 i andra typer av mjukvarusystem. Ett av målen med projektet var för källkoden att vara öppen för hämtning av intresserade utvecklare; och genom att inte koppla ett grafiskt användargränssnitt görs själva produkten mer modulär och lättare att integrera i andra projekt.

Eftersom mjukvaran endast använder sig av en liten delmängd av OpenSSLs funktionalitet, nämligen Base64-kodning samt SHA-3 hashing, skulle en egen implementering av

dessa algoritmer vara lämplig. Detta skulle minska mjukvarans användningskrav samt öppna dörrarna för mer specialiserade och optimerade algoritmer. Licensieringskomplikationerna som beskrevs i 7.2 skulle dessutom elimineras.

7.7.3 Mjukvarugränssnitt

För att genomföra autentiseringen startar PAM-modulen ett hjälpprogram som tar över PAM-modulens *stdin* och *stdout* medan PAM-modulen inväntar hjälpprogrammets resultat. Detta fungerar bra när PAM-autentiseringen begärs från ett program som körs i en terminal men är inte lämpligt för ett mer mångsidigt autentiseringssystem där autentiseringen kan ske genom ett grafiskt användargränssnitt, där användaren oftast inte har tillgång till *stdin* och *stdout*. Modulen skulle kunna utvidgas till att använda ett *pipe* eller en *socket* för interprocesskommunikation med hjälpprogrammet och aktivt kommunicera med programmet istället för att vänta på en returkod. Det finns programbibliotek som kan användas för att abstrahera sådan kommunikation. Ett programbibliotek som skulle kunna användas som utgångspunkt för vidareutveckling av gränssnittet är *Assuan*, vars bibliotek och protokoll utvecklades med syftet att hantera interprocesskommunikationen mellan GnuPG-komponenter som *pinentry*-programmen och *gpg-agent*-demonen [66]. Det använder en server-klient arkitektur som liknar modulens design. *dosa_chkcode*-programmet skulle kunna utvecklas till att implementera en *Assuan*-server och PAM-modulen skulle kunna starta en lämplig klient beroende på modulens konfigurering och omständigheterna som den anropas under.

7.8 Hur COVID-19-pandemin påverkat oss

Detta projekt har, likt många andra arbeten under åren 2020-2021, påverkats av den pågående covid-19-pandemin. Endast delar av gruppen har träffats fysiskt i det grupprum vi fick tilldelat för att arbeta med FPGA-kortet. Annars har all kontakt skett online via Discord vilket medförde svårigheterna med att arbeta på distans; exempelvis förekom det tekniska problem vid möten. Trots Covid-19-pandemin har större delen av arbetet fungerat på distans eftersom mycket har handlat om att programmera och testa kod på våra egna datorer. En ytterligare sak som bidrog till att arbetet kunde utföras på distans var möjligheten att ansluta sig till en dator på Chalmers nätverk med hjälp av *Remote Desktop*. Detta innebar att vi kunde använda programvara som behövdes för att simulera VHDL-koden utan att vi behövde fysiskt befinna oss på campus.

7.9 Ursprungliga tidsplan och avvikelser

Vår ursprungliga tidsplan (se bilaga E för Gantt-schemat) förhöll vi oss till mer eller mindre ganska väl. Vi planerade om våra sprintdatum på möten vid behov så fort vi insåg att det hade varit lämpligare. Produktutveckling skedde i stort sett den perioden som angetts och samma sak gäller testning som pågick längre än produktutveckling i syfte att demonstrera och dokumentera funktionaliteten för rapporten. Vi har använt den originella planeringen

som grund för när vi ändrade planeringen. Några tydliga saker som ändrades var:

- Istället för sprints på två veckor så blev de oftast 3 veckor långa.
- Vi fick tre tydliga parallella spår som alltid arbetades med: Knappsatsen, implementering av kryptering på FPGA och utvecklingen av PAM-modulen.
- Mot slutet av produktuvecklingen slutade vi planera med sprints. Vi blev avslappnade i och med att vi hade ca 3 veckors marginal till förhandsgranskningens deadline när produkten var färdigutvecklad samt testad. Arbetet flöt fortfarande på med rapporten, men sprints hade fortfarande kunnat gynna oss för att sätta specifika skrivande-mål.

Så fort produktutvecklingen nått sitt slut slutade också mer eller mindre sprintplaneringen, så vi övergick till enkla uppgifter på en veckobasis. Detta fungerade bättre gällande rapportskrivande men vi hade med fördel kunnat använda anslagstavlan mer aktivt för att bättre hantera ärenden vi enkelt kan formulera.

8 Slutsatser och sammanfattning

Ingen av gruppmedlemmarna hade initialt en omfattande förståelse eller erfarenhet med kryptografiska koncept eller datasäkerhet. Detta innebär att undersökning av grundläggande koncept inom nämnda ämnen ingick som inledande arbete innan utvecklingen av prototypen påbörjades. Som följd av detta hann vi knappt fördjupa oss ytterligare i något av koncepten vi har berört i projektet. Som nämnt tidigare hade vi med fördel kunnat utveckla våra egna mjukvarobibliotek för att undvika extra nedladdningar och installationer. Dessutom hade intresserade parter bara behöva ladda hem ett program som täcker huvudfunktionaliteten som behövs för produkten, men för att kunna utveckla dessa bibliotek i mjukvara hade en djup förståelse av SHA-3 behövts. Optimeringar i hårdvara behövs inte nödvändigtvis eftersom programmet är såpass enkelt, och många av optimeringarna har redan gjorts i VHDL av Keccak Team [67].

För att reda ut eventuella säkerhetsbrister hade förståelse av olika typer av attacker mot hårdvara behövts. En tänkbar attack mot säkerhetsdosan är *differential fault analysis*, då fel i kryptografiska implementationer framkallas genom att utsätta hårdvaran för oväntad fysisk belastning i försök att avslöja dess interna tillstånd [68]. Men för ordentlig analys av en sådan attack krävs eventuellt fördjupad förståelse av bland annat överklockning, elektriska fält, magnetfält och joniserad strålning. Detta är en relevant attack då den potentiellt kan användas för att extrahera nyckeln från koddosor som kör vår design; och eftersom designen är öppen för allmänheten kan det bli svårare att dölja nyckeln. Genom en ökad förståelse av nämnda områden av vetenskap kanske det är möjligt att återställbart förvränga nyckeln så den blir svårare att identifiera genom DFA eller liknande attacker.

En framtida problemställning hade varit att utveckla en praktiskt färdig produkt i ett låst kretskort; med tillhörande skal, skärm, knappar och batteriparti. Designen av alla dessa komponenter hade varit öppet tillgängliga så man hade kunnat 3d-skriva skalen och köpa in skärmar. På så sätt hade det varit avsevärt enklare att anta vår design, då den är närmare fullständig än innan.

Vi har lyckats utveckla en prototyp av koddosa genom vilken man med hög säkerhet kan avgöra om en person är i besittning av just denna dosan, och genom en tillhörande Linux-PAM modul har det fullständiga tänkta flödet demonstrerats. Designen av detta system är öppet tillgänglig på internet och kan laddas hem och modifieras för att utöka ett mindre robust lösenordssystem. Men varken undersökningar eller teori om vår produkt skulle mäta sig mot andra liknande tjänster såsom e-identitet [69] (vilket är en komplett tjänst som går att beställa) har inte etablerats.

Sammanfattningsvis är större delen av det tänkta syftet (se avsnitt 1.1) uppnått: Processen att konstruera en säkerhetsdosa har undersökts. Möjliga för- och nackdelar med att offentliggöra en design av detta slag har diskuterats. Framtida användare av produkten kommer ha samma möjlighet av förståelse av produkten som framtida organisationer som producerar dem har då de också kan läsa källkoden. Om detta ökar säkerhet och tillit är svårt att avgöra, men säkerheten av produkten är åtminstone tillgänglig för analys. Källkoden för

dosan som nämnt i avsnitt 7.7.1 kunde varit mycket mer modulär och generell vilket var ett mål i syfte att motivera och underlätta integrering av vår produkt. Vårt komplement till koddosan, alltså själva autentiseringstjänsten, fungerade dock precis som väntat och demonstrerade funktionaliteten av vår produkt som önskat.

Referenser

[1] D. J. Bernstein, "Chacha, a variant of salsa20," Jan. 2008. [https://cr.yp.to/chacha/chacha-20080128.pdf; Hämtad 19 februari 2021].

[2] "The gnu privacy guard." [https://gnupg.org; Hämtad 6 maj 2021].

[3] "Gcc, the gnu compiler collection," 5 2021. [https://gcc.gnu.org/; Hämtad 14 maj 2021].

[4] J. Li, L. Wu, and X. Zhang, "An efficient hmac processor based on the sha-3 hash function," in *2017 IEEE 12th International Conference on ASIC (ASICON)*, pp. 252–255, 2017.

[5] "make(1) - linux man page." [https://linux.die.net/man/1/make]; Hämtad 11 maj 2021.

[6] "Openssl library documentation." [https://www.openssl.org/docs; Hämtad 28 april 2021].

[7] "Rsa." [https://it-ord.idg.se/ord/rsa/; Hämtad 5 april 2021].

[8] "Nist releases sha-3 cryptographic hash standard." [https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard; Hämtad 18 mars 2021].

[9] "What is tls (transport layer security)?." [https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/; Hämtad 3 juni 2021].

[10] "Internetbanken privat." [https://id.seb.se/ibp/digipass/pnr; Hämtad 25 januari, 2021].

[11] H. Andersson, D. Bylinka, D. Frielingsdorf, J. Hellström, S. Randow, and A. Thunberg, "2021-datx02-17 källkod." [https://chalmersuniversity.box.com/s/v0goc1moyqlb1kawb5eb1md2gm9lrw2w].

[12] "What is open source?." [https://opensource.com/resources/what-open-source; Hämtad 11 maj 2021].

[13] A. G. Morgan and T. Kukuk, "The linux-pam system administrators' guide." [http://www.linux-pam.org/Linux-PAM-html/sag-overview.html; Hämtad 16 april 2021].

[14] S. Padhye, R. A. Sahu, and V. Saraswat, *Introduction to cryptography*. 6000 Broken Sound Parkway, NW, Suite 300: CRC Press, 2018.

[15] "Asymmetric cryptography - an overview." [https://www.sciencedirect.com/topics/computer-science/asymmetric-cryptography; Hämtad 11 maj 2021].

[16] T. Riley, Sean och Scott, “Hashing algorithms och security - computerphile,” Nov. 2013.

[17] J. P. Conley *et al.*, *Encryption, Hashing, PPK, and Blockchain: A Simple Introduction*. Nashville: Vanderbilt University, Department of Economics, 2019.

[18] S. Riley and M. Pound, “Sha: Secure hashing algorithm - computerphilee,” Apr. 2017.

[19] “Sha256.” [https://emn178.github.io/online-tools/sha256.html;Hämtad 15 april 2021].

[20] “National institute of standards and technology.” [https://www.nist.gov/; Hämtad 26 april 2021].

[21] “Nist selects winner of secure hash algorithm (sha-3) competition.” [https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition; Hämtad 26 april 2021].

[22] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “The sponge and duplex constructions.”

[23] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “Keccak specifications summary.”

[24] A. Anand, “Breaking down : Sha-3 algorithm.” [https://infosecwriteups.com/breaking-down-sha-3-algorithm-70fe25e125b6; Hämtad 19 april 2021].

[25] “Permutation.” [https://mathworld.wolfram.com/Permutation.html; Hämtad 3 juni 2021].

[26] “Message authentication code.” [https://www.sciencedirect.com/topics/computer-science/message-authentication-code; Hämtad 3 juni 2021].

[27] D. R. Coelho, *The VHDL handbook*. Springer Science & Business Media, 2012.

[28] “Ieee 1076-2019 - ieee standard for vhdl language reference manual.” [https://standards.ieee.org/standard/1076-2019.html; Hämtad 11 april 2021].

[29] “What is an fpga? field programmable gate array.” [https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html; Hämtad 11 april 2021].

[30] F. Fallahlalehzari, “How does the mars perseverance rover benefit from fpgas as the main processing units?.” [https://www.aldec.com/en/company/blog/188--how-does-the-mars-perseverance-rover-benefit-from-fpgas-as-the-main-processing-units; Hämtad 26 april 2021].

[31] “Vivado design suite.” [https://www.xilinx.com/products/design-tools/vivado.html; Hämtad 1 april 2021].

[32] “Modelsim*-intel® fpga edition software.” [https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html; Hämtad 1 april 2021].

[33] “Windows subsystem for linux installation guide for windows 10.” [https://docs.microsoft.com/en-us/windows/wsl/install-win10; Hämtad 1 april 2021].

[34] “Libcrypto api.” [https://wiki.openssl.org/index.php/Libcrypto_API; Hämtad 26 april 2021].

[35] “Gitlab is the open devops platform.” [https://about.gitlab.com/; Hämtad 22 april 2021].

[36] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design Using VHDL, A systems approach*. Shaftesbury Rd, Cambridge CB2 8BS, Storbritannien: Cambridge University Press, 2015.

[37] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, “Keccak implementation overview,” tech. rep., 2012. [https://keccak.team/files/Keccak-implementation-3.2.pdf; Hämtad 1 april 2021].

[38] “Nexys4™ fpga board reference manual.” [https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys4/documenatation/Nexys4_RM_VB1_Final_3.pdf; Hämtad 18 februari 2021].

[39] “Nexys a7: Fpga trainer board recommended for ece curriculum.” [https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/?_ga=2.106870275.145252530.1617260843-780827856.1617260843; Hämtad 1 april 2021].

[40] “Pmod kypd: 16-button keypad.” [https://store.digilentinc.com/pmod-kypd-16-button-keypad/; Hämtad 1 april 2021].

[41] “Advanced encryption standard.” [https://it-ord.idg.se/ord/advanced-encryption-standard/; Hämtad 5 april 2021].

[42] “elliptisk kryptering.” [https://it-ord.idg.se/ord/elliptisk-kryptering/; Hämtad 5 april 2021].

[43] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, 1978.

[44] D. Zomaya, “Linux file permissions: Understanding setuid, setgid, and the sticky bit,” Jan. 2021. [https://www.cbtnuggets.com/blog/technology/system-admin/linux-file-permissions-understanding-setuid-setgid-and-the-sticky-bit; Hämtad 10 maj 2021].

[45] “getrandom(2) - linux manual page.” [https://man7.org/linux/man-pages/man2/getrandom.2.html; Hämtad 2 april 2021].

[46] T. Hühn, "Myths about /dev/urandom." [https://www.2uo.de/myths-about-urandom/; Hämtad 2 april 2021].

[47] A. G. Morgan and T. Kukuk, "4.1. configuration file syntax." [http://linux-pam.org/Linux-PAM-html/sag-configuration-file.html; Hämtad 14 maj 2021].

[48] "Gnu general public license," June 2007. [https://www.gnu.org/licenses/gpl-3.0.html; Hämtad 9 april 2021].

[49] "Frequently asked questions about the gnu licenses - gnu project - free software foundation," June 2007. [https://www.gnu.org/licenses/gpl-faq.html#GPLRequireSourcePostedPublic; Hämtad 9 maj 2021].

[50] "Various licenses and comments about them - gnu project - free software foundation." [https://www.gnu.org/licenses/license-list.html#OpenSSL; Hämtad 25 april 2021].

[51] "Frequently asked questions about the gnu licenses - gnu project - free software foundation." [https://www.gnu.org/licenses/gpl-faq.html#SystemLibraryException; Hämtad 25 april 2021].

[52] "Licensing:faq - fedora project wiki." [https://fedoraproject.org/wiki/Licensing:FAQ?rd=Licensing/FAQ#What.27s_the_deal_with_the_OpenSSL_license.3F; Hämtad 25 april 2021].

[53] "Frequently asked questions." [https://www.openssl.org/docs/faq.html#LEGAL2; Hämtad 25 april 2021].

[54] "Comparative analysis of des, aes, rsaencryption algorithms." [https://www.ijemr.net/DOC/ComparativeAnalysisOfDESAESRSAEncryptionAlgorithms.pdf; Hämtad 14 maj 2021].

[55] B. Kaliski, "Twirl and rsa key size." [https://web.archive.org/web/20170417095741/https://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm; Hämtad 13 maj 2021].

[56] "Multiarch - debian wiki." [https://wiki.debian.org/Multiarch; Hämtad 11 maj 2021].

[57] "Makefiles (gnu make)." [https://www.gnu.org/software/make/manual/html_node/Makefiles.html; Hämtad 13 maj 2021].

[58] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[59] "Sha-3 10*1 padding." [https://keccak.team/sponge_duplex.html; Hämtad 3 juni 2021].

[60] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[61] “What is a remote administration tool (rat)?.” [https://www.mcafee.com/blogs/consumer/what-is-rat; Hämtad 13 maj 2021].

[62] “Rootkits defined: What they do, how they work, and how to remove them.” [RootkitsDefined:WhatTheyDo,HowTheyWork,andHowtoRemoveThem; Hämtad 13 maj 2021].

[63] M. J. Strube, “Tests of randomness for pseudorandom number generators.” [https://link.springer.com/content/pdf/10.3758/BF03203701.pdf; Hämtad 13 maj 2021].

[64] P. Hellekalek, “On correlation analysis of pseudorandom numbers.” [https://link.springer.com/chapter/10.1007/978-1-4612-1690-2_16; Hämtad 13 maj 2021].

[65] “Nexys a7 reference.” [https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/start; Hämtad 28 april 2021].

[66] “Libassuan.” [https://gnupg.org/software/libassuan/index.html; Hämtad 3 jun 2021].

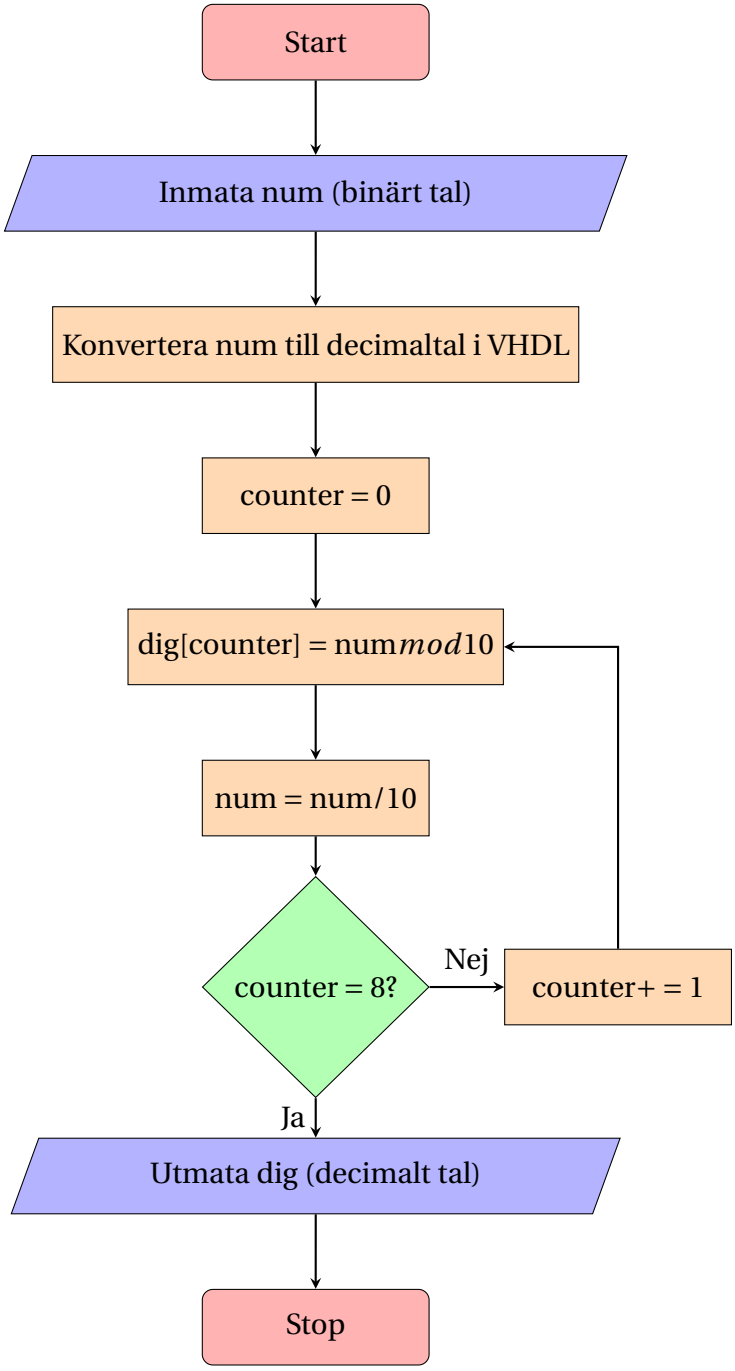
[67] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “Hardware resources.” [https://keccak.team/hardware.html; Hämtad 17 mars 2021].

[68] “Differential fault analysis of secret key cryptosystems.” [https://link.springer.com/content/pdf/10.1007/BFb0052259.pdf; Hämtad 3 juni 2021].

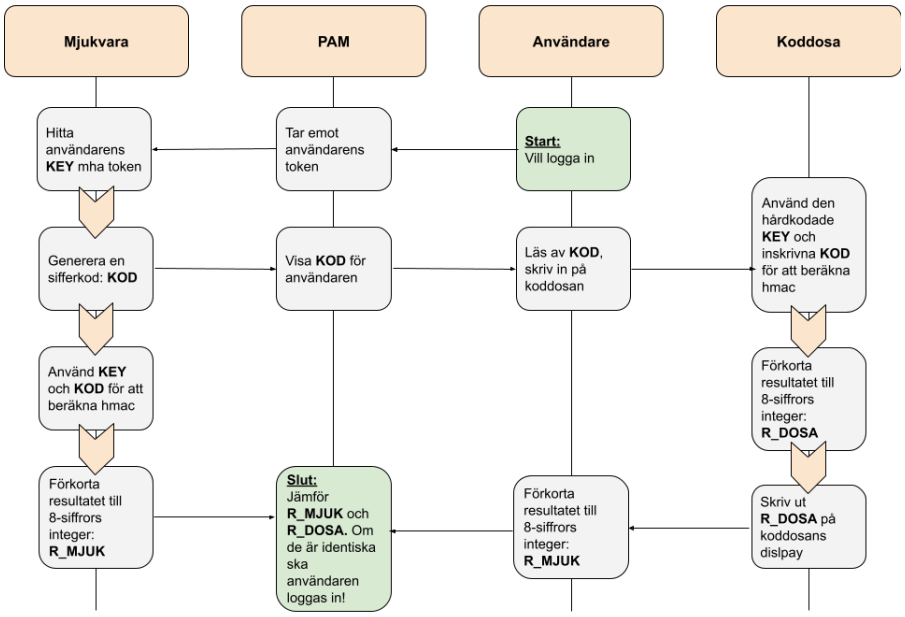
[69] “Engångskoder på smidig dosa för säker inloggning.” [https://e-identitet.se/auth/tvafaktorsautentisering-2fa/sakerhetsdosa/; Hämtad 29 april 2021].

Bilagor

A Diagram



Figur A.1: Flödesdiagram över hur den decimala till binära konverteringen går till. Åtta ensiffriga tal sparas i dig som sedan kan matas ut på display.



Figur A.2: Ett diagram av interaktionsflödet för produktens autentiseringsprocess. Processen startar hos användaren och leder till två parallella uträkningar hos mjukvaran och koddosan. Processen slutar hos PAM med att beräkningarna jämförs.

B DigiFlisp knappsats

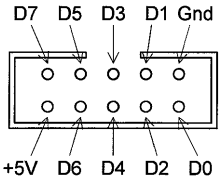
CHALMERS
Institutionen för data- och informationsteknik

2017-04-11

Keypad

Beskrivning

Keypadkortet innehåller ett helt passivt tangentbord där man lägger höga insignaler på de fyra raderna och avkänner kollumnernas värden. De kolumner som då ger hög signal är hopkopplade med någon rad via en nedtryck tangent.
Kopplingen kan ge interruptsignal då någon tangent är nedtryckt. Det finns två interruptsignaler IRQ low på banankontakt *J3* och IRQ high på banankontakt *J4* som ger låg respektive hög interruptsignal. Detta för att kortet skall kunna användas med system som använder olika aktiv interruptnivå. Interruptsignal indikeras i båda fallen via tänd lysdiod *D2*.
För kompatibilitet med labsystemet FLISP och med mikrodatorsystemet MD407 så kan anslutning ske på två olika sätt. Bit 0-3 i flatkabelkontakt *J1* ger alltid de fyra kolumnerna medan raderna antingen kan anslutas via bit 4-7 i kontakt *J1* eller via bit 4-7 i flatkabelkontakt *J2*.
Kontaktdonen i de två fallen visas i *Figur 1* medan *Tabell 1* respektive *Tabell 2* ger deras pinkonfigureringar.
För korrekt funktion så måste de processoringångar som är anslutna till ROW1 – ROW4 vara definierade med pulldown.



Figur 1 Flatkabelkontakt

Kontakt-ben	*Namn*	*Kontakt-ben*	*Namn*
1	GND	6	ROW1
2	COL1	7	ROW2
3	COL2	8	ROW3
4	COL3	9	ROW4
5	COL4	10	+5 V

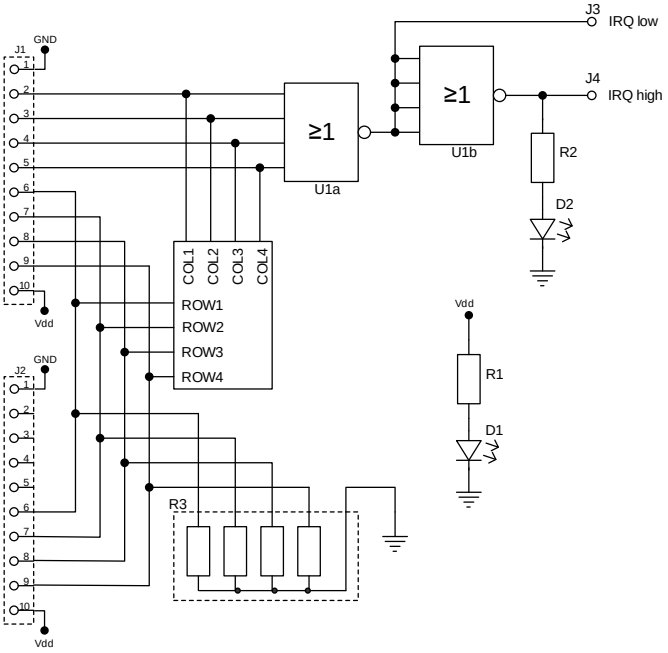
Tabell 1 Flatkabelkontakt J1

Kontakt-ben	*Namn*	*Kontakt-ben*	*Namn*
1	GND	6	ROW1
2	NC	7	ROW2
3	NC	8	ROW3
4	NC	9	ROW4
5	NC	10	+5 V

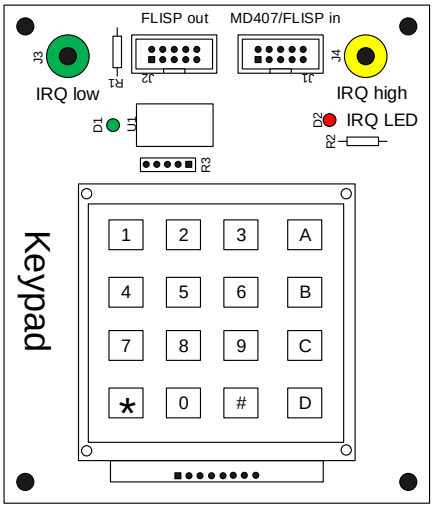
Tabell 2 Flatkabelkontakt J2

Figur 2 ger kortets kopplingsschema, *Figur 3* ger kortets kretskortslayout och *Tabell 3* ger komponentlistan.

CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för datorteknik
Besöksadress: Rännvägen 6
412 96 Göteborg



Figur 2: Keypad, kretsschema



Figur 3: Keypad, kretskortslayout

Beteckning	*Komponent*
J1, J2	Rakt flatkabeldon 2x5 stift
J3	Bananhylsa grön
J4	Bananhylsa gul
J11	Stiftlist 2x2 stift
S1	Tangentbord 4x4
R1, R2	330 Ω
R3	Resistansnät 1x4 100 kΩ
D1	LED grön
D2	LED röd
U1	CD4002

Tabell 3: Komponentlista

C DigiFlisp flatkabelanslutning

CHALMERS

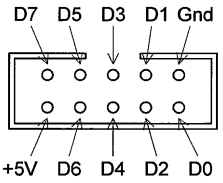
Institutionen för data- och informationsteknik

2017-04-11

10-polig flatkabelkontakt till 12-polig Digilentkontakt

Beskrivning

Kortet är avsett för att kunna ansluta Digilent-labmoduler med 12-polig kontakt till mikrodatorsystem med 10-poliga kontakter.
Kortet sammankopplas med mikrodatorsystemet via en 10-polig flatkabel med pinkonfiguration enligt *Figur 1* som även anges i *Tabell 1*.

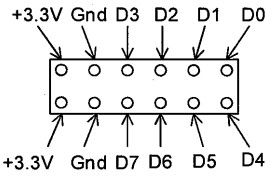


Figur 1 Flatkabelkontakt J1

Kontakt-ben	*Namn*	*Kontakt-ben*	*Namn*
1	GND	6	D4
2	D0	7	D5
3	D1	8	D6
4	D2	9	D7
5	D3	10	+5 V

Tabell 1 Flatkabelkontakt J1

Digilentmodulerna har en stiftlist som direkt ansluts till kortets Digilentsystemets hylslist. Hylslisten har en pinkonfiguration enligt *Figur 2* som även anges i *Tabell 2*.



Figur 2 Digilent stiftkontakt J2

Kontakt-ben	*Namn*	*Kontakt-ben*	*Namn*
1	3.3 V	7	D2
2	3.3 V	8	D6
3	GND	9	D1
4	GND	10	D5
5	D3	11	D0
6	D7	12	D4

Tabell 2 Digilent stiftkontakt J2

CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för datorteknik
Besöksadress: Rännvägen 6
412 96 Göteborg

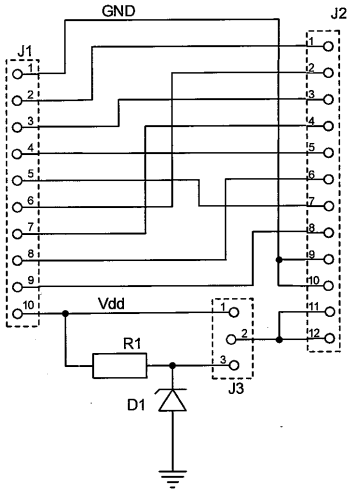
Digilentmodulerna är avsedda för spänningen 3,3 V medan mikrodatorsystemet levererar 5 V. Kortet ger därför möjlighet att sänka utspänningen från 5 V till 3,3 V. Via bygel *J3* bestäms om spänningen skall sänkas eller inte. *J3*:s benkonfiguration finns i *Tabell 3*.

Kontaktkortets kretsschema visas i *Figur 3* medan kortlayouten visas i *Figur 4*.

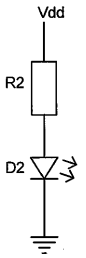
I *Tabell 4* ges kortets komponentlista.

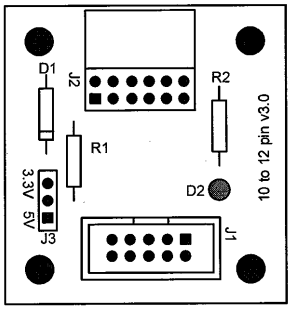
Kontakt-ben	*Namn*
1	5 V
5	Till modul
6	3,3 V

Tabell 3 Bygel J3



Figur 3 Kontaktkortets kretsschema





Figur 4 Kontaktkortets layout

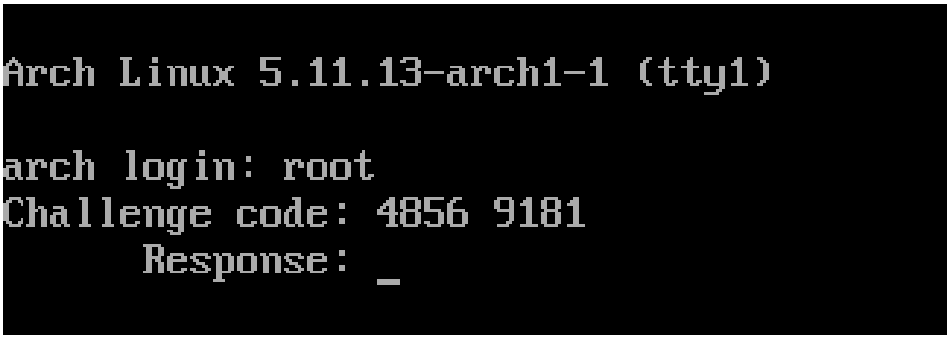
Beteckning	*Komponent*
J1	Rakt flatkabeldon 2x5 stift
J2	Vinklad hylsklist 2x6 stift
J3	Rak stiftlist 1x3 stift
R1,R2	330 Ω
D1	Zenerdiod 3,3 V
D2	LED grön

Tabell 4 Komponentlista

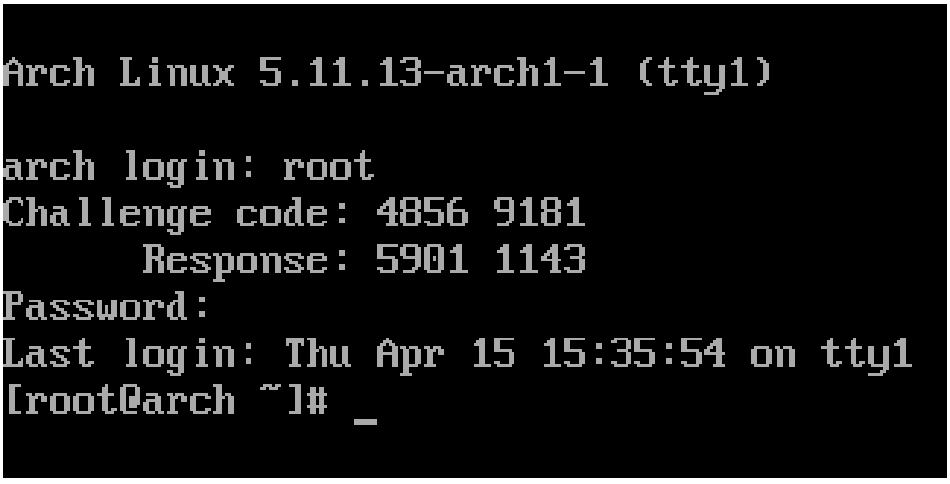
D Testprocess

49EAACE06DE4AAB0
8481F6DE7678A439
E69CAC0E9BFB18B2
2735E1A47619DFA6
F4A163C0C747D321
9E408844EE93C158
378763C69D3E27F9
9E74DC5A117A9885
081E58BA9CDB3436
2F6E2378FAE6C742
1E3931F787DAE7EF
C9F2F907308BA31E

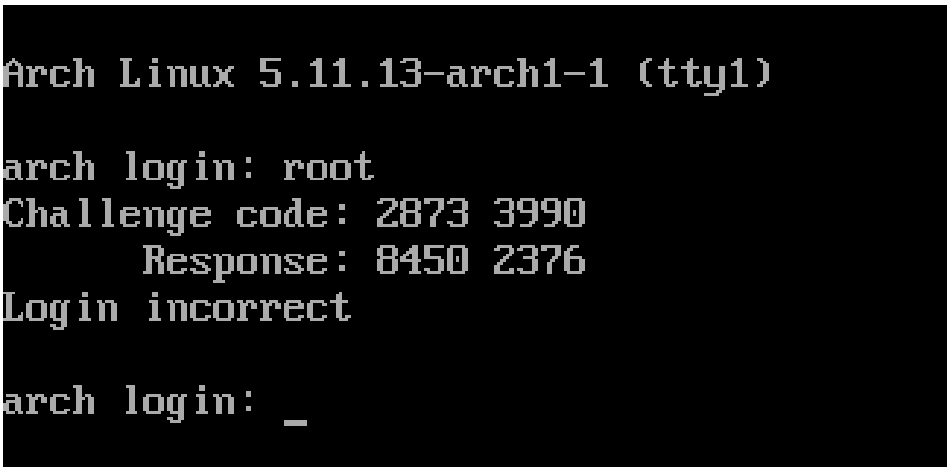
Figur D.1: Nyckeln använd i testning av användarflödet



Figur D.2: Skärmdump när PAM väntar sig en svarskod



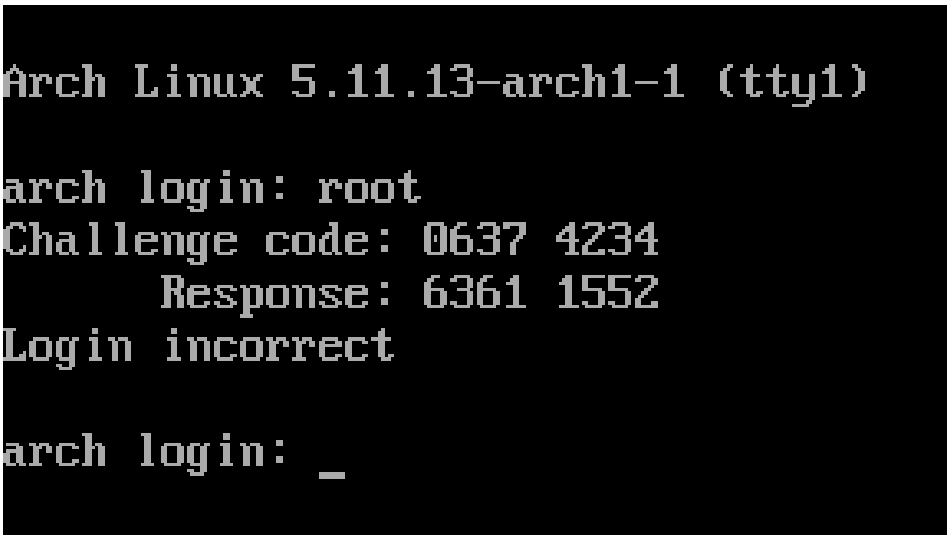
Figur D.3: Skärmdump när användaren loggats in efter svarskoden är korrekt



Figur D.4: Skärmdump när svarskoden är inkorrekt

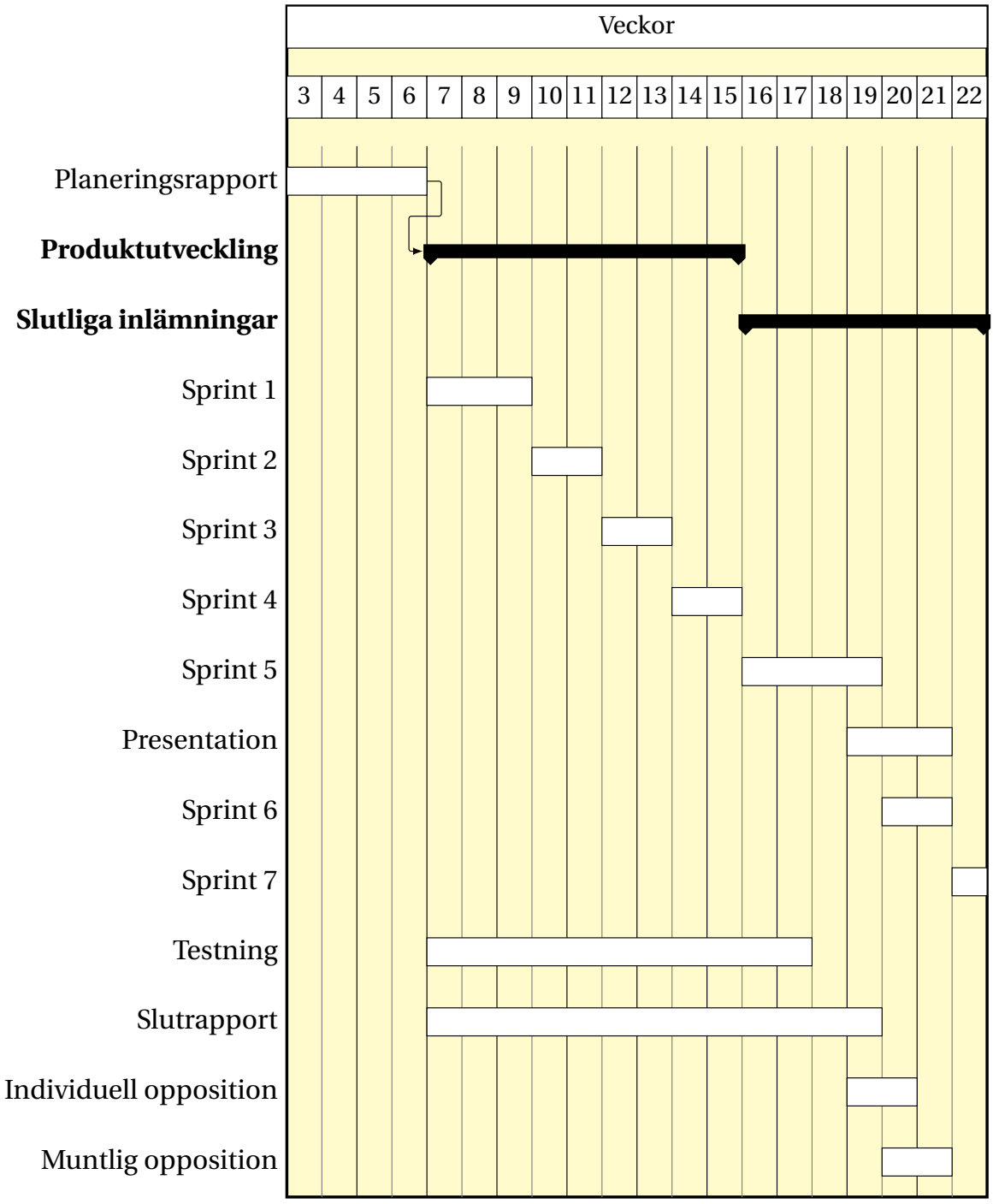
D4588DD47C52C669
F291D55564FA8F8F
B3DD93AD812361F6
D5EA1942D68050E0
4906DC82FB8DD02C
4C8862DE21BE4E9E
2720897EC3E5774D
BA5C19482BFD63DF
2593A53602CFE2A2
EDBFEC2401FA1491
7A7B9AFF9A20A353
B7D4EBB9D0B1CF0F

Figur D.5: Nyckeln använd i mjukvaran under testning med felaktig nyckel i koddosan



Figur D.6: Skärmdump när svarskoden genererad av en koddosa med fel nyckel matas in

E Tidsplanering från planeringsrapporten



Institutionen för data- och informationsteknik
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sverige
www.chalmers.se

CHALMERS