FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Obrany proti útokům postranními kanály založené na dynamické rekonfiguraci FPGA
Student:	Bc. Jan Brejník
Vedoucí:	Ing. Stanislav Jeřábek
Studijní program:	Informatika
Studijní obor:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Prozkoumejte možnosti využití dynamické rekonfigurace jako protiopatření vůči útokům postranními kanály na implementaci šifer PRESENT a AES v FPGA. Seznamte se s technikami dynamické rekonfigurace. Jako zdroj použijte především [1]. Postupy zde uvedené reimplementujte pro šifru PRESENT, inspirujte se jimi a prozkoumejte možnosti aplikace těchto metod pro implementaci šifry AES. Využijte FPGA osazené na měřících deskách, které jsou k dispozici na KČN, a proveďte alespoň několik pokusných měření.

Seznam odborné literatury

[1] P. Sasdrich, A. Moradi, O. Mischke, and T. Güneysu: "Achieving side-channel protection with dynamic logic reconfiguration on modern FPGAs," Journal of Cryptographic Engineering, no. 2, pp. 107–121, June 2014.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 3. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Obrany proti útokům postranními kanály založené na dynamické rekonfiguraci FPGA

Bc. Jan Brejník

Katedra číslicového návrhu
Vedoucí práce: Ing. Stanislav Jeřábek

10. ledna 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. ledna 2019 .....

Odkaz na tuto práci

Brejník, Jan. *Obrany proti útokům postranními kanály založené na dynamické rekonfiguraci FPGA*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Programovatelná hradlová pole (FPGA) disponují schopností dynamické rekonfigurace, díky které mohou být částečně přeprogramovány za běhu, a to bez nutnosti vnějšího zásahu. Jeden a ten samý výpočet tak může být v různých okamžicích realizován různým způsobem. Konkrétní způsob realizace v daný čas není pro případného útočníka známý, a proto je pro něj obtížnější využít informací uniklých postranními kanály k útoku, kterým by získal citlivé informace. Tato diplomová práce navazuje na článek [1], který popisuje aplikaci tří různých ochran na šifrovací algoritmus PRESENT. V rámci této práce byly tyto publikované ochrany aplikovány na šifrovací algoritmy PRESENT, SERPENT a AES. Algoritmus AES byl navíc implementován dvěma způsoby - první způsob je založen na postupu z [1], druhý způsob pak používá konečné kompozitní těleso pro implementaci S-Boxu, což vyžaduje méně CFGLUTů.

Klíčová slova Útoky postranními kanály, CPA, dynamická rekonfigurace, kompozitní těleso, maskování, AES, SERPENT, PRESENT

Abstract

Field Programmable Gate Arrays (FPGAs) have an ability of dynamic reconfiguration, which allows them to be reprogrammed at runtime by itself.

One computation can be implemented in different ways at different times. An actual way at a specific time is not known for an attacker and therefore it is much more difficult to use side-channel leakage to gain sensitive information. This diploma thesis follows the paper [1], which describes usage of three different countermeasures on PRESENT encryption algorithm. In this thesis, all these countermeasures were applied to PRESENT, SERPENT and AES. AES algorithm was implemented in two ways. The first way is based on the approach described in [1]. The second way uses a composite finite field to implement S-Box and therefore needs less CFGLUTs.

Keywords Side-channel attacks, CPA, dynamic reconfiguration, composite field, masking, AES, SERPENT, PRESENT

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Útoky postranními kanály jsou útoky, které se nesnaží útočit přímo na kryptografickou podstatu algoritmu, ale na jeho implementaci. Tímto útokem může být možné získat citlivou informaci zevnitř zařízení, třeba šifrovací klíč. Tu je možné zjistit například pomocí odběrové analýzy, při které útočník sleduje velikost odebíraného proudu, která závisí na hodnotách operandů, se kterými se uvnitř zařízení pracuje [5]. Dalším takovým útokem je analýza časová (timing attack) [6] nebo akustická [7].

Existují různé ochrany proti těmto útokům. Jde například o

- maskování (masking) [8], randomizuje mezivýsledky při výpočtu, díky čemuž spotřeba nezávisí na skutečných mezihodnotách
- skrývání (hiding) [9], které se snaží eliminovat únik informace, například zarušením sledovaného signálu

Tato práce navazuje na článek [1], který představuje použití tří různých ochran na šifrovací algoritmus PRESENT a využívá přitom dynamickou rekonfiguraci. V rámci této diplomové práce byly ochrany použity pro zabezpečení šifrovacích algoritmů PRESENT, SERPENT a AES. Je zde popsán způsob implementace těchto ochran, různé implementační detaily a dále výsledky měření úspěšnosti těchto ochran.

Text této práce je rozdělen na několik částí. První kapitola, Analýza, se zabývá popisem ochran a diskutuje způsob jejich použití pro jednotlivé šifry. Druhá kapitola, Implementace dynamické rekonfigurace, se zabývá obecným popisem realizace rekonfigurace. Pro její snazší implementaci totiž byly vytvořeny podpůrné nástroje, které jsou také popsány v této kapitole. Samotné podstatě této práce se pak věnuje třetí kapitola, Implementace šifrovacích algoritmů.

Další dvě kapitoly se pak již zabývají vyhodnocováním odolnosti proti útoku. Čtvrtá kapitola, Testování a měření, popisuje způsob měření spotřeby

a vyhodnocování výsledků. Dále rozebírá i některé nalezené příčiny úniku informace a jejich odstranění. Pátá kapitola, Výsledky měření, prezentuje konečné dosažené výsledky.

Poslední, šestá kapitola, Budoucí práce, obsahuje některé myšlenky, které by mohly pomoci pro další práci. Prezentuje jedno další protiopatření, které by mohlo eliminovat únik informace a několik tipů, které by mohly zefektivnit další práci.

V prvním dodatku této práce je popsán způsob útoku na implementaci algoritmu PRESENT, kterým je možné zjistit šifrovací klíč. Ve druhém dodatku je pak stručný přehled všech realizovaných implementací.

Kapitola 1

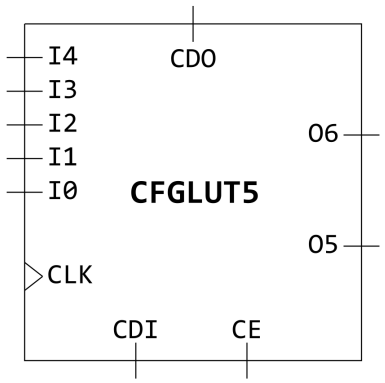
Analýza

Tato kapitola se zabývá teoretickou analýzou úkolu této práce. Nejprve je ve stručnosti popsán princip dynamické rekonfigurace a její stěžejní komponenta `CFGLUT5`. Dále jsou popsány ochrany prezentované v [1] a jsou diskutovány možnosti jejich použití u šifrovacích algoritmů PRESENT, SERPENT a AES. Další část se pak věnuje popisu realizace AES S-Boxu s použitím kompozitního tělesa (prezentováno v [10]) a způsobu ochrany této realizace.

1.1 Dynamická rekonfigurace

Dynamická rekonfigurace je schopnost FPGA být částečně přeprogramováno bez zásahu zvnější, přičemž nová konfigurace je vypočítána rovněž uvnitř FPGA. Existuje více možností, jak realizovat dynamickou rekonfiguraci. Lze využít například blokovou RAM (BRAM), komponentu SRL16, CFGLUT [1] nebo distribuovanou paměť (distributed RAM).

V této práci bude použita komponenta `CFGLUT5` dostupná pro FPGA Xi-



Obrázek 1.1: CFGLUT5, schématická značka

linx Spartan 6 [11] (ale i pro jiné řady). Jde o komponentu, která realizuje logickou funkci s pěti vstupy a jedním výstupem, přičemž daná funkce může být zcela libovolná a nastavená až za běhu. Vstupem CFGLUTu jsou porty `I0 - I5`, funkční hodnota funkce je pak na výstupním portu `O5`. Kromě toho je k dispozici ještě port `O6`, který realizuje čtyřvstupovou funkci, tj. nebere v úvahu hodnotu vstupního portu `I5` a generuje takový výstup, jako by hodnota portu `I5` byla `0`. Tyto zmíněné porty jsou propojeny pouze kombinační logikou, tedy výstupy `O5` a `O6` jsou přenastaveny bezprostředně po změně vstupů.

Pro rekonfiguraci slouží porty `CDO`, `CDI` a `CE`. Na `CFGLUT5` lze pohlížet jako na posuvný registr, kde `CDI` je vstupem, `CDO` je výstupem a signál `CE` povoluje zápis (a tedy i posun). Port `CLK` představuje hodinový signál a řídí posouvání hodnot v posuvném registru (pokud má `CE` hodnotu `1`). Hodnota portu `O5`, resp. `O6` je pak určena hodnotou bitu v posuvném registru, adresa tohoto bitu je určena vstupními porty `I0 - I5`.

Grafické znázornění rekonfigurace pro náhodně zvolenou funkci zadanou pravdivostní tabulkou je na obrázku 1.2.

1.1.1 Řazení CFGLUTů pro zvýšení počtu vstupních bitů

`CFGLUT5` může realizovat funkci s až 5 vstupními bity. V praxi ale 5 vstupů může být málo, například S-Box algoritmu AES má 8 vstupních bitů. V takovém případě je možné použít více CFGLUTů spojených do kaskády, přičemž každý další bit zdvojnásobuje počet CFGLUTů. Pokud potřebujeme realizovat n-vstupovou funkci $F(x_{n-1}, ..., x_0)$ s využitím m-vstupových CFGLUTů a $n > m$, bude potřeba $2^{(n-m)}$ CFGLUTů. Každý z nich má pouze m vstupů, na které budou přivedeny signály $x_{m-1}, ...x_0$. Pro každou možnou hodnotu zbývajících signálů $x_{n-1}, ..., x_m$ pak existuje jeden CFGLUT. Konkrétně tedy:

- nultý CFGLUT realizuje funkci

$$F_0(x_{m-1}, ..., x_0) = F(0, ..., 0, 0, 0, x_{m-1}, ..., 0)$$

- první CFGLUT realizuje funkci

$$F_1(x_{m-1}, ..., x_0) = F(0, ..., 0, 0, 1, x_{m-1}, ..., 0)$$

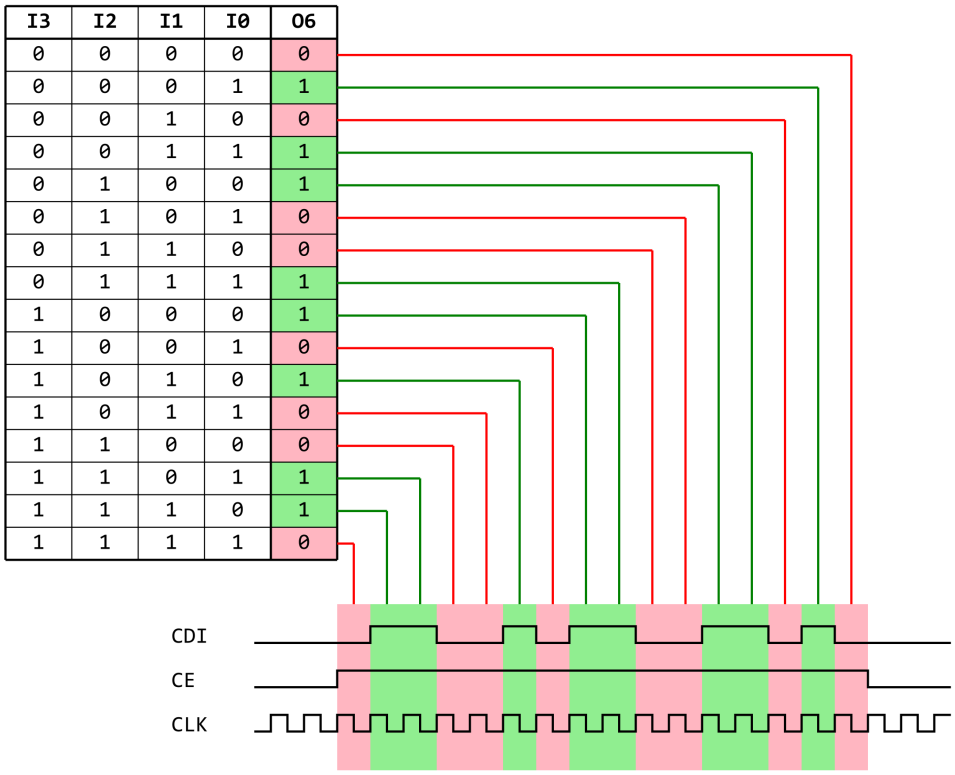
- druhý CFGLUT realizuje funkci

$$F_2(x_{m-1}, ..., x_0) = F(0, ..., 0, 1, 0, x_{m-1}, ..., 0)$$

- třetí CFGLUT realizuje funkci

$$F_3(x_{m-1}, ..., x_0) = F(0, ..., 0, 1, 1, x_{m-1}, ..., 0)$$

- atd.



Obrázek 1.2: Grafické znázornění rekonfigurace CFGLUTu. Nahrávaná logická funkce je na obrázku popsána pravdivostní tabulkou.

- a poslední, $(2^{(n-m)} - 1)$-tý CFGLUT realizuje funkci

$$F_{2^{(n-m)}-1}(x_{m-1}, ..., x_0) = F(1, ..., 1, 1, 1, x_{m-1}, ..., x_0)$$

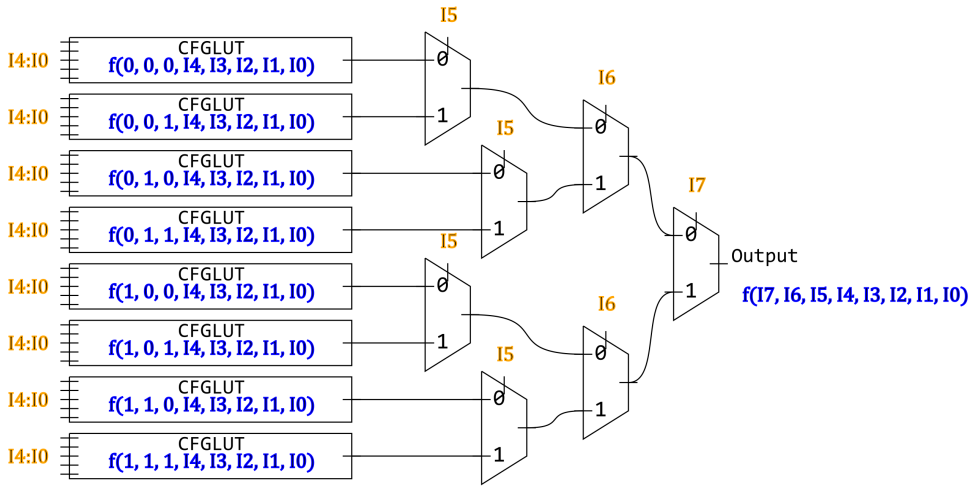
Multiplexor na základě skutečných hodnot signálů $x_{n-1}, ..., x_m$ vybere výstup správného CFGLUTu. Tento přístup lze použít pro funkci s libovolným počtem vstupních bitů, nicméně je nutné brát v úvahu, že počet potřebných CFGLUTů roste exponenciálně. Díky tomu, že CFGLUTy mají výstupní port `CDO`, je možné propojit porty `CDO` a `CDI` sousedních CFGLUTů a všechny je konfigurovat skrz jeden vodič sériově.

Například, pro osmivstupovou ($n = 8$) funkci je zapotřebí $2^{(n-m)} = 2^{(8-5)} = 8$ instancí komponenty `CFGLUT5` ($m = 5$). Zapojení je na obrázku 1.3.

Je-li $n \leq m$, bude potřeba právě jeden CFGLUT.

Vzhledem k tomu, že `CFGLUT5` realizuje funkci s právě jedním výstupním bitem, je nutné ještě navíc počet CFGLUTů vynásobit počtem výstupních bitů realizované funkce.

Tento způsob rozkladu je vlastně Shannonova expanze [12].



Obrázek 1.3: Zapojení osmi pětivstupových CFGLUTů pro realizaci osmivstupové funkce.

1.2 Použité ochrany

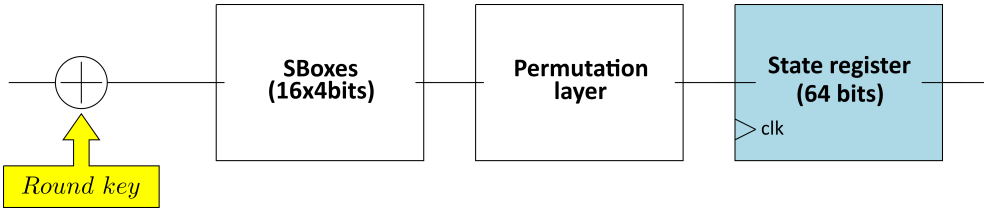
Na obrázcích 1.4 až 1.7 jsou jednotlivé ochrany popisované v následujícím textu schématicky znázorněny. Na těchto obrázcích je vždy jedna runda algoritmu PRESENT včetně stavového registru bez zpětné smyčky mezi výstupem rundy a vstupem (následující) rundy. Popiskem `Reg` je označen stavový registr, popisek `Perm` pak označuje permutaci bitů.

1.2.1 Dekompozice S-Boxu

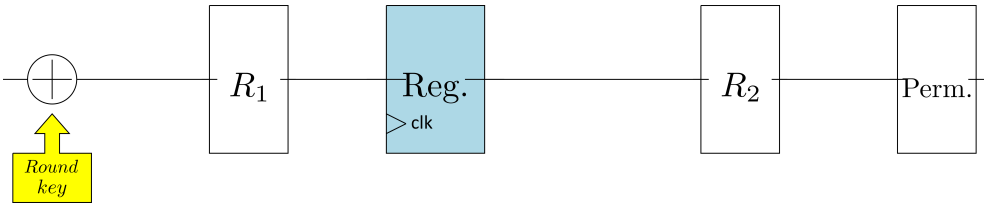
Dekompozice S-Boxu je jedna z ochran proti útoku postranními kanály, která byla publikována v článku [1]. Základní myšlenkou je rozdělení nelineární substituční části (S-Boxu) na dvě bijekce R_1, R_2, které dohromady dávají původní S-Box, tedy musí pro ně platit tato rovnice:

$$R_2(R_1(x)) = SBox(x) \tag{1.1}$$

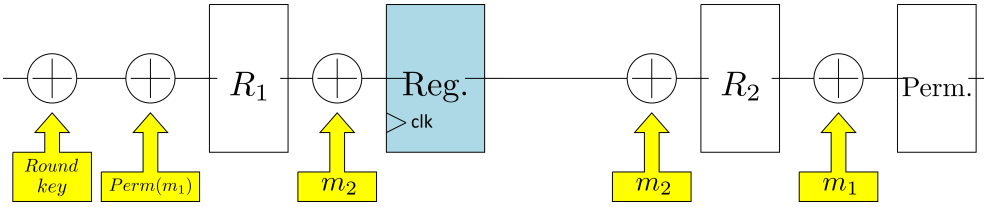
Tyto dvě bijekce zapojené za sebou nahradí původní S-Box. Bijekce nemusí kromě výše uvedené rovnice splňovat žádné další podmínky, existuje tedy velké množství možných párů (R_1, R_2) a díky dynamické rekonfiguraci lze pro každé šifrování použít jiný. Teoreticky existuje $(2^n)!$ různých n-bitových bijekcí R_1. Pro každou z nich lze najít právě jednu bijekci R_2 takovou, aby platila výše uvedená rovnice. Vzhledem k faktoriálu jde tedy o obrovské množství různých párů (R_1, R_2), mezi kterými lze náhodně vybírat.



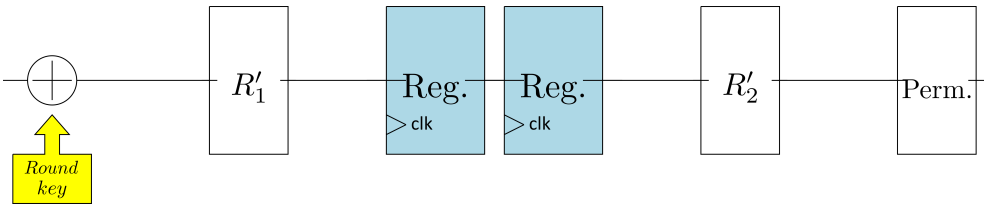
Obrázek 1.4: Schématické znázornění jedné rundy PRESENTu bez jakýchkoliv ochran



Obrázek 1.5: S-Box je dekomponován na dvě bijekce R_1 a R_2



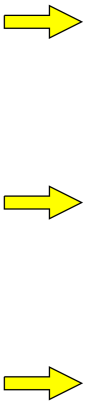
Obrázek 1.6: Booleovské maskování náhodnými hodnotami m_1 a m_2.



Obrázek 1.7: Booleovské maskování je realizováno uvnitř bijekce $R'_1(x) = R_1(Perm(m_1) \oplus x) \oplus m_2$ a $R'_2(x) = R_2(m_2 \oplus x) \oplus m_1$. Kromě toho je přidán další stavový registr (register precharge)

x	$R_1(x)$
0	5
1	15
2	**12**
3	8
4	11
5	6
6	4
7	3
8	2
9	0
10	13
11	14
12	1
13	**9**
14	10
15	7

x	$R_2(x)$
0	9
1	15
2	13
3	2
4	0
5	10
6	7
7	6
8	4
9	**14**
10	3
11	8
12	**5**
13	1
14	11
15	12



x	$R_1(x)$
0	5
1	15
2	**9**
3	8
4	11
5	6
6	4
7	3
8	2
9	0
10	13
11	14
12	1
13	**12**
14	10
15	7

x	$R_2(x)$
0	9
1	15
2	13
3	2
4	0
5	10
6	7
7	6
8	4
9	**5**
10	3
11	8
12	**14**
13	1
14	11
15	12

Obrázek 1.8: Příklad na obrázku znázorňuje náhodnou modifikaci dekomponovaného S-Boxu. Levá strana zachycuje stav před modifikací, pravá strana pak znázorňuje stav po záměně řádků 2 a 13.

Párem řádků se v tomto textu myslí libovolná dvojice řádků pravdivostní tabulky. Záměna páru řádků pak znamená vzájemné prohození hodnot v těchto dvou řádcích.

Stavový registr je umístěn mezi bijekce R_1 a R_2, hodnota mezi nimi je totiž pro útočníka nepredikovatelná *"Since most side-channel attacks on symmetric block ciphers target the output of the non-linear substitution layer, it might be beneficial to avoid the storage of the S-box outputs into such registers. [...] By moving the state register into the substitution layer, we split the standard S-box up into two (random) mappings. Hence, we never store a correct S-box output into a register but only (randomly) mapped values"* [1]

Ačkoliv bijekce může být vybrána zcela náhodně před každým šifrováním, není v mém návrhu generována vždy nová, ale je nějakým způsobem modifikována bijekce předchozí. Tento přístup je jednodušší a neměl by nijak oslabit ochranu (v [1] je také použit tento způsob). Na začátku (po resetu) je jedna z bijekcí identita a druhá S-Box. Před šifrováním je první bijekce modifikována záměnou několika dvojic řádků. Podle hodnot zaměněných řádků jsou pak i zaměněny řádky v druhé bijekci a tím je zajištěno, že složení obou bijekcí dává dohromady stále původní S-Box. Jeden krok modifikace (tj. záměna jednoho páru řádků) je graficky znázorněn na příkladu na obrázku 1.8.

V mé implementaci bude před každým šifrováním prováděna záměna osmi

párů řádků (na rozdíl od řešení popisovaného v [1]). Je to z toho důvodu, že plánuji i implementaci AESu, který obsahuje 8bitový S-Box a jeho pravdivostní tabulka má tak výrazně více řádků. Navíc bude možné experimentálně ověřit vliv počtu zaměňovaných párů řádků na účinnost této ochrany.

Kromě toho se mé řešení bude odlišovat od [1] i tím, že nesdílí bijekce R_1 a R_2 všemi instancemi S-Boxů, ale pro každý S-Box jsou bijekce různé. Do budoucna to opět umožní experimentálně ověřit, zda tato změna zvýší ochranu proti útokům a má tedy smysl (i přes větší nároky na FPGA).

1.2.1.1 AES

Protože AES obsahuje 8bitový S-Box, je nutné jej realizovat více CFGLUTy, jak je uvedeno v části 1.1. V rámci této práce je navržen a implementován ještě jeden způsob, který používá kompozitní těleso pro výpočet S-Boxu a potřebuje menší počet CFGLUTů. Ten je prezentován v části 1.3.

1.2.1.2 SERPENT

Algoritmus SERPENT se od algoritmů PRESENT nebo AES odlišuje tím, že definuje 8 různých S-Boxů, které se během jednoho šifrování střídají napříč rundami. Nabízí se dvě možnosti, jak řešit dekompozici:

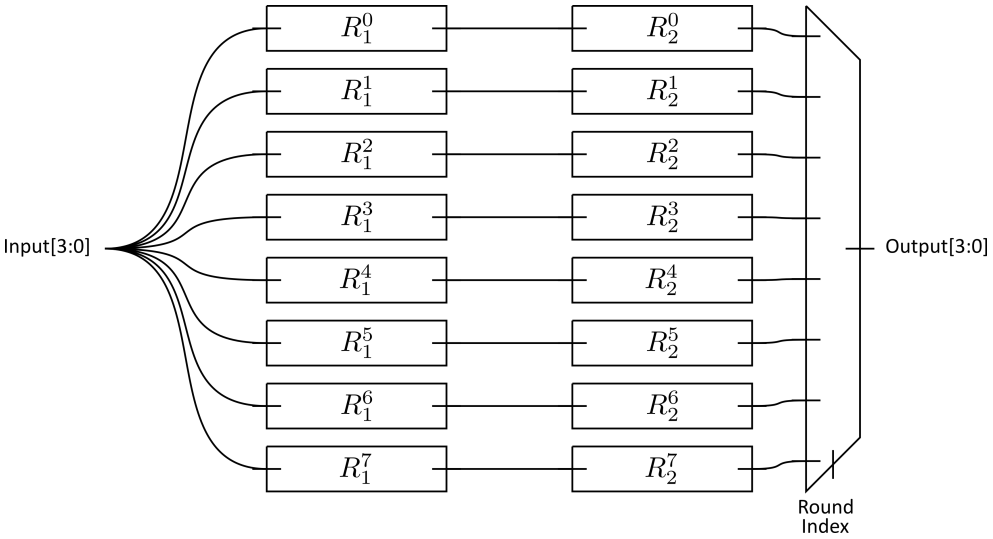
- Každý z osmi S-Boxů bude dekomponován na 2 bijekce. Celkem tedy bude realizováno 16 bijekcí a v každé rundě se pomocí multiplexorů vybere ta správná dvojice. Naznačeno na obrázku 1.9.
- První bijekce bude pro všechny S-Boxy společná. Tato možnost je méně náročná, protože stačí realizovat pouze 9 bijekcí. V implementaci bude použita tato varianta. Naznačeno na obrázku 1.10.

(Pro jednoduchost není na odkazovaných obrázcích znázorněn stavový registr, který je ve skutečnosti umístěn mezi levou a pravou částí.)

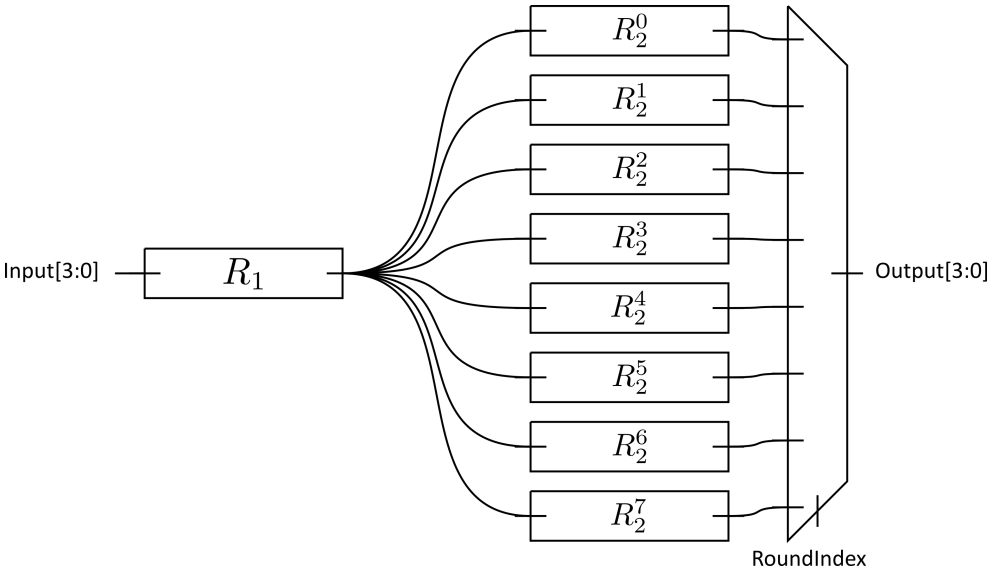
1.2.2 Booleovské maskování

Booleovské maskování randomizuje mezivýsledky při šifrování. Do rundy je vloženo přičítání náhodných dat (operace XOR) a díky tomu jsou následné mezihodnoty pro útočníka nepredikovatelné. Aby však nebyla narušena správnost výsledku, je nutné umístit do rundy ještě další operaci XOR, která eliminuje změnu způsobenou prvním XORem a obnoví správnou hodnotu mezivýsledku. Úkolem je tedy najít, kam do rundy lze vložit tyto dva XORy, aniž by byla ovlivněna správnost výsledku.

V následujícím textu bude první XOR s náhodnou hodnotou (který randomizuje mezivýsledky) označován jako "přičítání"a druhý jako "odečítání". Protože se pohybujeme v binárním tělese, jde v obou případech o stejnou



Obrázek 1.9: PRESENT, dekompozice S-Boxu, varianta 1



Obrázek 1.10: PRESENT, dekompozice S-Boxu, varianta 2

operaci XOR, nicméně toto rozlišení umožňuje vyjádřit, která operace randomizuje hodnoty a která naopak obnovuje zpět správnou hodnotu.

1.2.2.1 PRESENT

V článku [1] jsou uvedeny dva způsoby, jak pomocí maskování ochránit algoritmus PRESENT. Náhodné binární vektory (získané generátorem náhodných čísel) jsou označeny m_1 a m_2 a jejich délka je stejná jako délka celého bloku, tedy 64 bitů v případě algoritmu PRESENT.

1. Lze přičíst $Perm^{-1}(m_1)$ za druhou částí dekomponovaného S-Boxu a odečíst m_1 před první částí dekomponovaného S-Boxu. V tomto případě se přičítá jiná hodnota než odečítá. Důvodem je, že mezi přičtením a odečtením se nachází permutace bitů. Proto se musí přičítat inverzní permutace m_1.

   Druhou možností je přičítat m_1 a odečítat $Perm(m_1)$. Ve své implementaci zvolím tuto možnost, protože není nutné implementovat inverzi permutace, ale stačí pouze permutaci, která už stejně musí být implementována (je součástí rundy). V tomto případě je rozdíl minimální, ale například u algoritmu AES by bylo nezbytné implementovat inverzi operace MixColumns, která je složitější.

2. Lze přičíst náhodnou hodnotu m_2 za první částí dekomponovaného S-Boxu a odečíst náhodnou hodnotu m_2 před druhou částí dekomponovaného S-Boxu. Tím zjevně nebude ovlivněna správnost výsledku, neboť mezi přičtením a odečtením není žádná operace, pouze stavový registr, který lze z tohoto pohledu považovat za identitu.

Schéma jedné rundy s touto ochranou a s dekomponovaným S-Boxem je na obrázku 1.6.

Protože přičítání i odečítání náhodných hodnoty je vždy přímo před nebo přímo za bijekcí R_1 či R_2, není nutné do rundy přidávat XOR, ale přičtení a odečtení může být již součástí této bijekce. Tedy konfigurace CFGLUTů (jež realizují bijekce R_1 a R_2) bude vytvořena tak, aby prováděla XORování na vstupu i na výstupu.

Jinými slovy, CFGLUTy nebudou realizovat bijekce R_1 a R_2, ale bijekce R'_1 a R'_2:

$$R'_1(x) = R_1(x \oplus Perm(m_1)) \oplus m_2 \tag{1.2}$$

$$R'_2(x) = R_2(x \oplus m_2) \oplus m_1 \tag{1.3}$$

U prvního maskování (m_1) je nutné brát v úvahu, že zatímco náhodná hodnota se přičítá v jedné rundě, příslušné odečtení je až v rundě následující. Proto by v první rundě mělo být odečítání vynecháno. V tomto případě by

ale vynechání odečítání bylo neefektivní, bylo by nutné rekonfigurovat CFGLUTy, což je náročná operace (z hlediska počtu potřebných taktů). Proto se v první rundě před bijekci R_1 vkládá operace XOR s hodnotou $Perm(m_1)$, která odečítání eliminuje.

V poslední rundě je pak nutné ještě odečíst hodnotu $Perm(m_1)$ k získání správného šifrového textu (u algoritmu PRESENT je šifrovým textem hodnota po přičtení posledního rundovního klíče, substituční část a permutace se v poslední rundě vynechává).

1.2.2.2 Maskování při realizaci lineárních transformací

V předchozím textu je u algoritmu PRESENT ukázáno, že je-li maska přičítána před permutací a odečítána až za ní, nelze odečítat stejnou hodnotu jako byla přičtena, ale je nutné jí nejprve permutovat. Následující text se zabývá zobecněním této skutečnosti.

Předpokládejme vektorový prostor nad konečným tělesem $GF(2)$, kde prvky tohoto prostoru tvoří n-prvkové vektory, jejichž jednotlivé složky jsou z tělesa $GF(2)$.

Dále předpokládejme libovolnou transformaci f, která transformuje jeden n-bitový vektor. Pokud chceme výpočet transformace f ochránit maskováním, můžeme masku m přičíst před vstupem do transformace a na výstupu odečíst $f(m)$, jak je naznačeno na obrázku 1.11. Z něj je také patrné, že výsledek bude správný právě když

$$f(x) = f(x \oplus m) \oplus f(m) \tag{1.4}$$

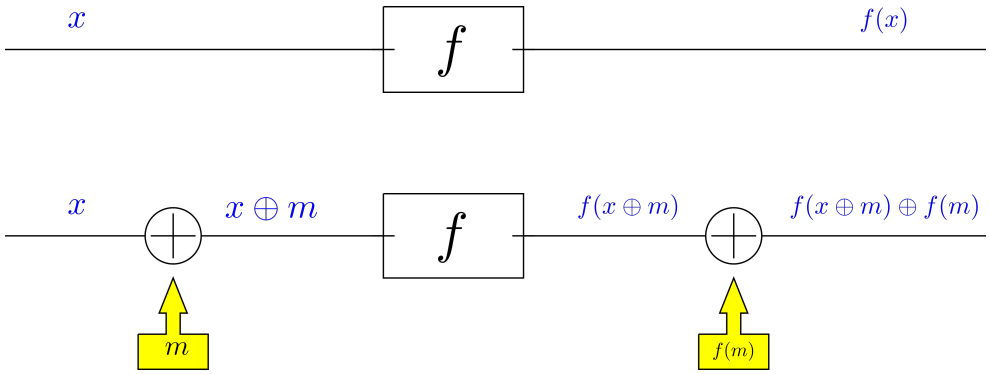
Tato podmínka je splněna, pokud f je lineární transformace [13]. Lze tedy říci, že booleovským maskováním lze ochránit výpočet libovolné lineární transformace a to tak, že ke vstupní hodnotě se přičte náhodná hodnota m a od výstupní hodnoty se odečte $f(m)$.

Maskováním lze ochránit i XORování s konstantou, například s rundovním klíčem. V tomto případě se pro odmaskování odečítá ta samá hodnota, která se přičítala. Lze to snadno dokázat. Předpokládejme hodnotu x, ke které se má přičíst rundovní klíč k. Bez maskování bude výsledkem $x \oplus k$. S maskováním bude výsledkem $(x \oplus m) \oplus k$. Odečteme-li od něj zase stejnou masku dostaneme $((x \oplus m) \oplus k) \oplus m = x \oplus k$.

1.2.2.3 SERPENT

Booleovské maskování lze u algoritmu SERPENT použít obdobným způsobem, jako u PRESENTu a proto není nutné jej podrobněji rozepisovat. Zmíním pouze dvě zásadní skutečnosti:

- SERPENT obsahuje místo permutace lineární transformaci, maskování lze tedy provést stejným způsobem, jako je popsáno výše.



Obrázek 1.11: Maskování libovolné lineární transformace

- V poslední rundě se neaplikuje lineární transformace, ale místo ní se přičítá další rundovní klíč. Pro odmaskování je tedy nutné odečíst m_1, nikoliv $L_{SERPENT}(m_1)$, kde $L_{SERPENT}$ označuje lineární transformaci použitou v algoritmu SERPENT.

1.2.2.4 AES

Maskování lze provést i u algoritmu AES podobně jako u předchozích:

- Algoritmus AES používá kromě nelineárního S-Boxu (SubBytes) operace ShiftRows a MixColumns a přičtení rundovního klíče. ShiftRows je vlastně permutace, MixColumns je lineární transformace (násobení maticí).

- V poslední rundě se vynechává operace MixColumns, pro odmaskování je tedy nutné odečíst $ShiftRows(m_1)$, v předchozích rundách se odečítá $MixColumns(ShiftRows(m_1))$.

1.2.3 Register precharge

Ačkoliv maskování zcela randomizuje mezivýsledky, nemá žádný vliv na rozdíl mezi hodinovými takty. Jinak řečeno, má-li nějaký mezivýsledek v taktu j hodnotu h_0 a v taktu $j+1$ hodnotu h_1, pak rozdíl mezi nimi je při použití maskování $h_0 \oplus h_1$. Stejný je ale i při použití maskování: $(h_0 \oplus m) \oplus (h_1 \oplus m) = h_0 \oplus h_1$.

V článku [1] je proto použita ještě ochrana register precharge. Spočívá v tom, že šifrování reálných dat je proloženo šifrováním bloku náhodných dat. Toto opatření příliš nezvyšuje nároky na FPGA, jedinou úpravou je zdvojení stavového registru (první z nich ukládá data, se kterými se v aktuálním taktu nepracuje) a upravit řadič. Tato ochrana ale zdvojnásobí potřebný počet hodinových taktů.

Schéma jedné rundy šifry PRESENT ochráněné všemi uvedenými ochranami (včetně register precharge) je na obrázku 1.7.

1.3 Realizace AES S-Boxu s využitím kompozitního tělesa

Algoritmus AES se odlišuje od ostatních zde diskutovaných algoritmů tím, že používá 8bitový S-Box. Ten lze sice také dekomponovat na dvě bijekce, ale pro realizaci jednoho výstupního bitu je potřeba 8 CFGLUTů (viz 1.1.1), na celý návrh potom 2 048 (8 CFGLUTů na jeden výstupní bit * 128 bitů na jeden blok * 2 bijekce). To je značně více než u algoritmu SERPENT, který používá stejně velký blok (128 bitů), ale má pouze 4bitový S-Box a stačí mu tak celkem 256 CFGLUTů (1 CFGLUT na jeden výstupní bit * 128 bitů na jeden blok * 2 bijekce). Kromě počtu potřebných CFGLUTů je třeba brát v úvahu i nezbytné multiplexory. Tato část se zabývá ještě jedním způsobem ochrany S-Boxu, který si vystačí s o něco menším počtem CFGLUTů.

Tento způsob realizace vychází z článku [10], ve kterém je popsán výpočet S-Boxu s použitím kompozitního tělesa.

1.3.1 Konečná tělesa

V této části se pracuje s konečnými tělesy a proto je předpokládána jejich základní znalost. Potřebné informace lze nalézt v [14] (jde o části 2.1 a 2.2).

V následujícím textu i v samotné implementaci jsou použita některá konkrétní tělesa, která jsou definována v tabulkách 1.1. Každý řádek v této tabulce představuje jedno rozšířené těleso (extended field). V rámci jedné tabulky je pak vyjádřen vztah mezi tělesy. Platí, že těleso na jednom řádku je rozšířením tělesa na řádku pod ním. Těleso F_B je kompozitní těleso vzniklé rozšířením tělesa F_{B2}. Těleso F_C je kompozitní těleso vzniklé rozšířením tělesa F_{C2}, které je rozšířením tělesa F_{C3}.

Na přiloženém DVD jsou ve složce GfReports informace o zde používaných tělesech a izomorfismech (seznamy prvků těles, jejich inverze, matice izomorfismů apod.).

Prvky konečných těles jsou zde zapisovány po složkách, podobně jako vektory. Například jednotkový prvek tělesa $GF(2^8)$ bude zapisován jako

$$(0, 0, 0, 0, 0, 0, 0, 1)$$

.

1.3.2 AES S-Box

Dle specifikace algoritmu AES se při výpočtu S-Boxu pohlíží na jednotlivé osmice bitů jako na prvky z Rijndaelova tělesa. Výpočet pak probíhá tak, že

Tabulka 1.1: Použitá konečná tělesa

Název	Řád	Primitivní prvek	Ireducibilní polynom
F_{AES}	256	$(0,0,0,0,0,0,1,0)$	$P_{AES}(x) = x^8 + x^4 + x^3 + x + 1$

Název	Řád	Primitivní prvek	Ireducibilní polynom
F_A	256	$(0,0,0,0,0,0,1,0)$	$P_{R1}(x) = x^8 + x^4 + x^3 + x^2 + 1$

Název	Řád	Primitivní prvek	Ireducibilní polynom
F_B	256	$((0,0,0,1),(0,0,0,0))$	$P_1(x) =$ $(0,0,0,1)x^2 + (0,0,0,1)x +$ $(1,0,0,1)$
F_{B2}	16	$(0,0,1,0)$	$Q_1(x) = 1x^4 + x + 1$

Název	Řád	Primitivní prvek	Ireducibilní polynom
F_C	256	$(((0,0),(0,1)),((0,0),(0,0)))$	$P_1'(x) =$ $((0,0),(0,1))x^2 +$ $((0,0),(0,1))x +$ $((1,1),(1,1))$
F_{C2}	16	$((0,1),(0,0))$	$P_2(x) = (0,1)x^2 + (0,1)x +$ $(1,0)$
F_{C3}	4	$(1,0)$	$Q_2(x) = 1x^2 + 1x + 1$

se nejprve určí multiplikativní inverze prvku, která je následně transformována afinní transformací. Při ní se už na osmici pohlíží jako na binární vektor. Transformace se skládá z násobení maticí (8×8) a přičtení (XOR) konstantního 8prvkového vektoru. S-Box lze tedy v FPGA realizovat i tímto výpočtem. Velkou nevýhodou ale je, že výpočet multiplikativní inverze je náročný. Může být ale zjednodušen použitím konečného kompozitního tělesa.

1.3.3 Izomorfismus mezi Rijndaelovým tělesem a kompozitním tělesem F_C

Pro snadný výpočet multiplikativní inverze je nutné převést prvek do kompozitního tělesa. V [14] je uveden algoritmus, který dokáže nalézt izomorfismus mezi dvěma tělesy (pochopitelně se stejným řádem), kde zdrojové těleso je běžné rozšířené těleso definované jedním ireducibilním polynomem (v citované práci pojmenován R) a kompozitním tělesem, které je určené dvěma polynomy (v citované práci jde o polynomy P a Q). Uvedený algoritmus ale předpokládá, že všechny polynomy jsou primitivní, což v případě ireducibilního polynomu Rijndaelova tělesa není splněno.

Z tohoto důvodu bude prvek z tělesa F_{AES} nejprve izomorfismem zobrazen

na prvek z F_A. Pak již lze použít citovaný algoritmus na nalezení izomorfismu mezi F_A a F_B a následně mezi F_B a F_C. Druhým možným řešením by bylo nepoužívat algoritmus z [14], ale přímo hledat izomorfismus mezi F_{AES} a F_C algoritmem popsaným v následující části.

1.3.3.1 Izomorfismus mezi Rijndaelovým tělesem a tělesem F_A

Zobrazení $Iso_1 : F_{AES} \rightarrow F_A$ je izomorfismem, pokud $\forall a, b \in F_{AES}$ platí [15]:

$$Iso(a) + Iso(b) = Iso(a + b) \tag{1.5}$$

$$Iso(a) \cdot Iso(b) = Iso(a \cdot b) \tag{1.6}$$

Vzhledem k malému počtu prvků lze pro nalezení izomorfismu použít naivní algoritmus:

1. Vybrat libovolný primitivní prvek α z prvního tělesa a libovolný primitivní prvek β z druhého tělesa.

2. Zkonstruovat zobrazení $Iso : F_{AES} \rightarrow F_1$ tak, že $Iso(0) = 0$, $Iso(\alpha) = \beta$, $Iso(\alpha^2) = \beta^2$, $Iso(\alpha^3) = \beta^3$ atd.

3. Ověřit, zda zobrazení je isomorfismem, tedy zda splňuje podmínky 1.5 a 1.6. Pokud ano, je izomorfismus nalezen. V opačném případě se opakuje postup s jiným párem primitivních prvků.

Z podstaty izomorfismu se musí primitivní prvek jednoho tělesa vždy mapovat na primitivní prvek druhého tělesa, proto se v kroku jedna zkouší pouze dvojice primitivních prvků. Jakmile se prvek α zobrazí na prvek β, musí být ostatní prvky zobrazeny způsobem uvedeným v bodě 2. Každý jiný způsob by zjevně nesplňoval podmínku 1.6.

Tímto algoritmem jsem určil, že zobrazením prvku $(0,0,0,0,0,0,1,0) \in F_{AES}$ na $(0,0,0,0,0,0,1,1) \in F_A$ vznikne izomorfismus. Toto zobrazení je lineární, takže jej lze zapsat maticí:

$$Iso_1(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \times x$$

$$Iso_1^{-1}(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \times x$$

1.3.3.2 Izomorfismus mezi F_A a F_B

Izomorfismus mezi tělesem F_A a F_B byl nalezen algoritmem popsaným v [14]. Tento algoritmus zde nebude popisován, v tabulkách 1.2 a 1.3 jsou pouze uvedeny jednotlivé kroky jeho běhu.

Výstupem algoritmu je matice pro zobrazení elementu z tělesa F_A do tělesa F_B.

$$Iso_2(x) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \times x$$

$$Iso_2^{-1}(x) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \times x$$

1.3.3.3 Izomorfismus mezi F_B a F_C

Tento izomorfismus byl také nalezen algoritmem [14]. V tomto případě stačilo provést pouze jednu iteraci algoritmu. Následující matice převádějí obě části vstupního 8bitového vektoru, tedy levá horní čtvrtina je stejná jako pravá dolní čtvrtina. Ostatní prvky matice jsou nulové.

Tabulka 1.2: Běh algoritmu na nalezení izomorfismu

t	Element	Komentář
1	$((0;0;0;1);(0;0;0;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((1;0;0;1);(1;1;1;1))$
2	$((0;0;0;1);(1;0;0;1))$	Element byl označen v kroku $t = 1$
3	$((1;0;0;0);(1;0;0;1))$	Element není primitivní
4	$((0;0;0;1);(0;1;0;0))$	Element byl označen v kroku $t = 0$
5	$((0;1;0;1);(1;0;0;1))$	Element není primitivní
6	$((1;1;0;0);(1;0;1;1))$	Element není primitivní
7	$((0;1;1;1);(0;1;1;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((1;1;1;0);(0;0;0;0))$
8	$((0;0;0;1);(1;0;1;0))$	Element byl označen v kroku $t = 1$
9	$((1;0;1;1);(1;0;0;1))$	Element není primitivní
10	$((0;0;1;0);(1;1;0;0))$	Element není primitivní
11	$((1;1;1;0);(0;0;0;1))$	Izomorfismus nenalezen, $P(\alpha^t) = ((0;1;1;0);(1;1;0;0))$
12	$((1;1;1;1);(0;1;1;1))$	Element není primitivní
13	$((1;0;0;0);(1;1;1;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((0;1;0;1);(0;1;1;0))$
14	$((0;1;1;0);(0;1;0;0))$	Element byl označen v kroku $t = 7$
15	$((0;0;1;0);(0;0;1;1))$	Element není primitivní
16	$((0;0;0;1);(0;0;0;1))$	Element byl označen v kroku $t = 1$
17	$((0;0;0;0);(1;0;0;1))$	Element není primitivní
18	$((1;0;0;1);(0;0;0;0))$	Element není primitivní
19	$((1;0;0;1);(1;1;0;1))$	Izomorfismus nenalezen, $P(\alpha^t) = ((0;0;0;0);(1;1;1;0))$
20	$((0;1;0;0);(1;1;0;1))$	Element není primitivní
21	$((1;0;0;1);(0;0;1;0))$	Element není primitivní
22	$((1;0;1;1);(1;1;0;1))$	Element byl označen v kroku $t = 11$
23	$((0;1;1;0);(1;1;0;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((1;0;0;0);(1;1;1;0))$
24	$((1;0;1;0);(0;0;1;1))$	Element není primitivní
25	$((1;0;0;1);(0;1;0;1))$	Element není primitivní
26	$((1;1;0;0);(1;1;0;1))$	Element byl označen v kroku $t = 13$
27	$((0;0;0;1);(0;1;1;0))$	Element není primitivní
28	$((0;1;1;1);(1;0;0;1))$	Element byl označen v kroku $t = 7$
29	$((1;1;1;0);(1;0;1;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((1;0;0;0);(0;1;0;1))$
30	$((0;1;0;0);(0;1;1;1))$	Element není primitivní
31	$((0;0;1;1);(0;0;1;0))$	Izomorfismus nenalezen, $P(\alpha^t) = ((0;1;0;0);(1;1;0;1))$
32	$((0;0;0;1);(1;0;0;0))$	Element byl označen v kroku $t = 1$

Tabulka 1.3: Běh algoritmu na nalezení izomorfismu, pokračování tabulky 1.2

t	Element	Komentář
33	$((1;0;0;1);(1;0;0;1))$	Element není primitivní
34	$((0;0;0;0);(1;1;0;1))$	Element není primitivní
35	$((1;1;0;1);(0;0;0;0))$	Element není primitivní
36	$((1;1;0;1);(1;1;1;1))$	Element není primitivní
37	$((0;0;1;0);(1;1;1;1))$	Izomorfismus nalezen, $P(\alpha^t) = ((0;0;0;0);(0;0;0;0))$

$$Iso_3(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times x$$

$$Iso_3^{-1}(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times x$$

1.3.4 Výpočet multiplikativní inverze v kompozitním tělese

V následujícím textu výraz $A(x)/B(x)$ představuje podíl polynomů A a B. Výraz $A(x) \mod B(x)$ pak představuje zbytek po dělení polynomů. Platí, že

$$(A(x)/B(x)) \cdot B(x) + (A(x) \mod B(x)) = A(x)$$

.

Multiplikativní inverzi lze vypočítat pomocí rozšířeného Euklidova algoritmu (EEA). Zde se právě projeví význam použití kompozitního tělesa. Pro něj totiž stačí provést nejvýše jednu iteraci EEA. U původního tělesa F_{AES} by to bylo více. Díky tomu je i implementace v FPGA výrazně jednodušší, vystačí si pouze s kombinační logikou.

Předpokládejme, že chceme určit inverzi prvku z tělesa F_C. Na prvek tohoto tělesa lze pohlížet jako na polynom $A(x) = ax + b$, kde $a, b \in F_{C2}$. Běh

Tabulka 1.4: EEA pro obecný prvek tělesa F_C

Zbytek	Podíl	Koeficienty	Koeficienty
$P(x)$		1	0
$A(x)$		0	1
$P(x) \mod A(x)$	$P(x)/A(x)$	$1 - 0(P(x)/A(x))$	$0 - 1(P(x)/A(x))$

EEA pro obecný prvek a obecný ireducibilní polynom $P(x) = x^2 + \alpha * x + \beta$ definující těleso je znázorněn v tabulce 1.4.

Pro řádky EEA platí Bézoutova rovnost, tedy v posledním řádku platí

$$P(x) \mod A(x) = 1 \cdot P(x) - 1 \cdot (P(x)/A(x)) \cdot A(x)$$

Po vydělení obou stran výrazem $(P(x) \mod A(x))$ pak platí:

$$1 = \frac{P(x)}{P(x) \mod A(x)} - \frac{(P(x)/A(x))}{P(x) \mod A(x)} \cdot \frac{A(x)}{1}$$

Z toho plyne, že inverze prvku $A(x)$ je $(P(x)/A(x))/(P(x) \mod A(x))$. Je tedy nutné určit podíl a zbytek po dělení $P(x)/A(x)$.

Podíl:

$$(x^2 + \alpha x + \beta)/(ax + b) = a^{-1}x + a^{-1}(\alpha - a^{-1}b)$$

Zbytek po dělení:

$$\begin{array}{rrrrrrrrrrr}
(x^2 & + & & \alpha x & + & & \beta) & / & (ax & + & b) & = \ldots \\
\cline{1-7}
-x^2 & - & a^{-1}bx & & & & & & & & & \\
\cline{1-7}
 & & (\alpha - a^{-1}b)x & + & & \beta & & & & & & \\
 & - & (\alpha - a^{-1}b)x & - & a^{-1}(\alpha b - a^{-1}b^2) & & & & & & & \\
\cline{1-7}
 & & & & \beta - a^{-1}b\alpha + a^{-2}b^2 & & & & & & &
\end{array}$$

Inverze:

$$\frac{P(x)/A(x)}{P(x) \mod A(x)} = \frac{a^{-1}x + a^{-1}(\alpha - ba^{-1})}{\beta - a^{-1}b\alpha + b^2a^{-2}} = \frac{ax + a(\alpha - ba^{-1})}{a^2\beta - ab\alpha + b^2} =$$

$$\frac{ax + a(\alpha - ba^{-1})}{a^2\beta - ab\alpha + b^2} = \frac{ax + a(\alpha - ba^{-1})}{a^2\beta - b(a\alpha - b)} = \frac{a}{a^2\beta - b(a\alpha - b)}x + \frac{a\alpha - b}{a^2\beta - b(a\alpha - b)}$$

Provedené úpravy:

- První rovnítko: dosazení podílu a zbytku
- Druhé rovnítko: vynásobení čitatele i jmenovatele a^2

- Třetí rovnítko: žádná úprava
- Čtvrté rovnítko: vytknutí b ve jmenovateli
- Páté rovnítko: rozdělení na dva zlomky, přepsání výrazu do tvaru $a^{inv}x + b^{inv}$

$$a^{inv} = \frac{a}{a^2\beta - b(a\alpha - b)} \tag{1.7}$$

$$b^{inv} = \frac{a\alpha - b}{a^2\beta - b(a\alpha - b)} \tag{1.8}$$

Možná realizace tohoto vzorce v FPGA je schématicky znázorněna na obrázku 1.12.

Výše uvedené odvození ale předpokládá, že polynom $A(x)$ je právě prvního stupně. Je nutné ošetřit i případ $a = 0$. Inverzí je pak zjevně b^{-1}. Dosazením $a = 0$ do rovnic 1.7 a 1.8 je vidět, že i v tomto případě dává správný výsledek.

Poslední možný případ je, že $a = 0$ a zároveň $b = 0$, tedy jde o nulový prvek. Ten multiplikativní inverzi nemá. Protože ale S-Box musí být definován pro všechny hodnoty, je stanoveno, že pro tento konkrétní případ se jako inverze nuly bude považovat zase nula. Dosazením do odvozeného vzorce je vidět, že pro nulu je jeho výsledkem také nula.

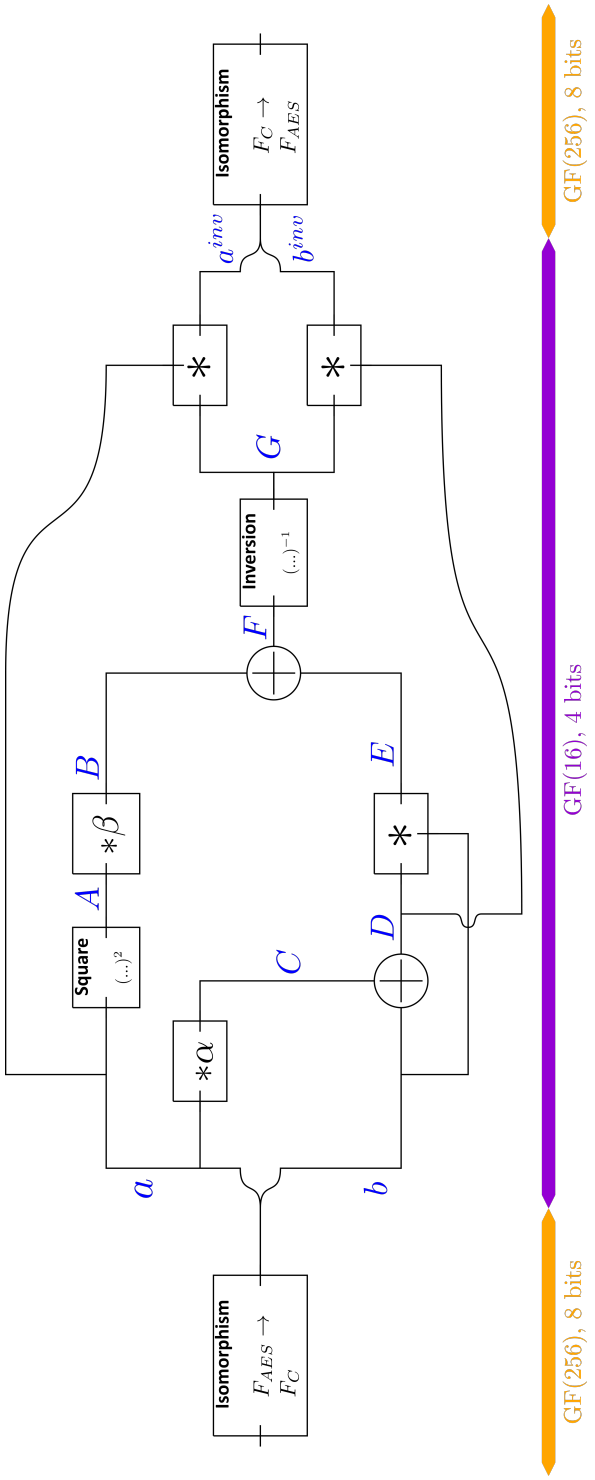
1.3.5 Minimalizace úniku informace při výpočtu multiplikativní inverze

Navržený způsob výpočtu multiplikativní inverze je nutné ochránit proti úniku informace postranními kanály. Lze použít maskování, tedy přičítán náhodné masky na výstupu jedné dílčí operace a její odečítání na vstupu do následující dílčí operace. Aby toho bylo možné dosáhnout, budou realizovány pomocí CFGLUTů a díky tomu se nikde během výpočtu nebude objevovat nezamaskovaná hodnota.

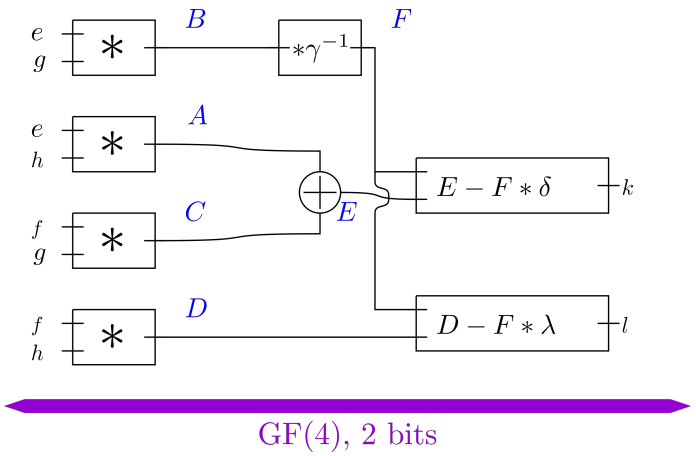
Vstupem následujících operací je jeden vstupní operand, a proto jsou realizovatelné pomocí 5vstupového CFGLUTu:

- Druhá mocnina (výstup na uzel A)
- Násobení konstantou β (výstup na uzel B)
- Násobení konstantou α (výstup na uzel C)
- Multiplikativní inverze (výstup na uzel G)

První dvě operace lze navíc sloučit do jedné, protože výstup jedné je použit pouze jako vstup do druhé. Násobení konstantou α (uzel C) pak nemusí být



Obrázek 1.12: Realizace multiplikativní inverze kombinační logikou



Obrázek 1.13: Realizace násobení v tělese GF(4). Všechny vodiče mají šířku 2 bity.

implementováno vůbec, pokud je zvolen ireducibilní polynom takový, že $\alpha = 1$, což v tomto případě je.

Komplikovanější situace je u násobení dvou prvků (výstup na uzel E, a^{inv}, b^{inv}). Do této operace vstupují 2 čtyřbitové prvky, tedy celkem 8 bitů, což již nelze snadno implementovat 5vstupovými CFGLUTy. Lze ale celou tuto operaci rozdělit na dílčí operace.

Předpokládejme dva různé prvky z tělesa F_{C2} zapsané ve tvaru polynomu jako $ex + f$ a $gx + h$ a jejich součin zapsaný ve tvaru $kx + l$. Pro obecnost dále označme koeficienty polynomu P_1:

$$\gamma = ((0,0),(0,1))$$

$$\delta = ((0,0),(0,1))$$

$$\lambda = ((1,1),(1,1))$$

Platí tedy, že $P_2(x) = \gamma x^2 + \delta x + \lambda$. Součin prvků je:

$$(ex + f) \cdot (gx + h) = (eg)x^2 + (fg + he)x + hf \mod P_2$$

$$k = (fg + he) - eg\gamma^{-1}\delta \tag{1.9}$$

$$l = (hf) - eg\gamma^{-1}\lambda \tag{1.10}$$

Schématické znázornění realizace tohoto výpočtu v FPGA je na obrázku 1.13. Ve výpočtu inverze je operace násobení celkem třikrát (viz obrázek 1.12), tedy pro jeden S-Box budou zapotřebí tři instance schématu z obrázku 1.13.

Podobně jako dříve budou dílčí operace realizovány CFGLUTy, jejichž konfigurace bude ochraňovat maskováním mezivýsledky. Do každé operace vstupují nejvýše 4 bity (každý vodič má šířku 2 bity a reprezentuje prvek z tělesa F_{C3}).

1.3.6 Příklad výpočtu S-Boxu pro hodnotu 0xE7

V této části je podrobně popsán postup výpočtu S-Boxu pro hodnotu 231 (šestnáctkově 0xE7), tedy $(1,1,1,0,0,1,1,1)$. Mezihodnoty při výpočtu jsou pojmenovány stejně jako ve schématech (1.12 a 1.13). V reálné implementaci vstupuje do výpočtu zamaskovaná hodnota, výstupem je zamaskovaná hodnota a zamaskované jsou i všechny mezivýsledky. Pro jednoduchost v tomto příkladu není maskování uvažováno.

1.3.6.1 Výpočet inverze

$$Iso_1((1,1,1,0,0,1,1,1)) = (1,0,0,1,1,1,1,0)$$

$$Iso_2((1,0,0,1,1,1,1,0)) = ((0,1,1,0),(0,0,1,0))$$

$$Iso_3((0,1,1,0),(0,0,1,0)) = (((0,0),(1,0)),((0,1),(0,0)))$$

$$a = ((0,0),(1,0))$$

$$b = ((0,1),(0,0))$$

$$A = ((0,0),(1,1))$$

$$B = ((1,0),(1,0))$$

$$C = ((0,0),(1,0))$$

$$D = ((0,1),(1,0))$$

$$E = ((1,1),(1,0))$$

$$F = ((0,1),(0,0))$$

$$G = ((1,1),(1,1))$$

$$a^{inv} = ((0,1),(0,1))$$

$$b^{inv} = ((0,1),(0,0))$$

$$Iso_3^{-1}(((0,1),(0,1)),((0,1),(0,0))) = ((0,0,1,1),(0,0,1,0))$$

$$Iso_2^{-1}((0,0,1,1),(0,0,1,0)) = (1,1,0,0,0,1,1,1)$$

$$Iso_1^{-1}(1,1,0,0,0,1,1,1) = (1,0,1,0,1,1,0,1)$$

1.3.6.2 Afinní transformace (součást AES S-Boxu)

Dále je nutné aplikovat afinní transformaci, která je součástí AES S-Boxu

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

1.3.6.3 Násobení 1 (výstup na uzel E)

$$e = (0, 1)$$
$$f = (1, 0)$$
$$g = (0, 1)$$
$$h = (0, 0)$$
$$A = (0, 0)$$
$$B = (0, 1)$$
$$C = (1, 0)$$
$$D = (0, 0)$$
$$E = (1, 0)$$
$$F = (0, 1)$$
$$k = (1, 1)$$
$$l = (1, 0)$$

1.3.6.4 Násobení 2 (výstup na uzel a^{inv})

$$e = (0, 0)$$
$$f = (1, 0)$$
$$g = (1, 1)$$
$$h = (1, 1)$$
$$A = (0, 0)$$
$$B = (0, 0)$$
$$C = (0, 1)$$

$$D = (0, 1)$$
$$E = (0, 1)$$
$$F = (0, 0)$$
$$k = (0, 1)$$
$$l = (0, 1)$$

1.3.6.5 Násobení 3 (výstup na uzel b^{inv})

$$e = (1, 1)$$
$$f = (1, 1)$$
$$g = (0, 1)$$
$$h = (1, 0)$$
$$A = (0, 1)$$
$$B = (1, 1)$$
$$C = (1, 1)$$
$$D = (0, 1)$$
$$E = (1, 0)$$
$$F = (1, 1)$$
$$k = (0, 1)$$
$$l = (0, 0)$$

Výsledkem je tedy prvek $(1, 0, 0, 1, 0, 1, 0, 0)$, který reprezentuje hodnotu 148, šestnáctkově 0x94.

1.4 Generátor náhodných čísel

Všechny popisované ochrany jsou silně závislé na náhodných číslech, která vlastně určují, jakým způsobem se provede rekonfigurace a jaký blok se použije pro register precharge. V reálném nasazení by tedy musel být součástí zařízení i kvalitní generátor náhodných čísel. V mém návrhu, který je určen primárně pro výzkumné a vyhodnocovací účely, jsem se rozhodl nepoužívat žádný generátor a místo něj náhodná data posílat do FPGA společně s otevřeným textem před každým šifrováním. V reálném nasazení by tím všechny ochrany ztratily smysl, nicméně v tomto případě přináší několik výhod:

- Proces šifrování je zcela deterministický, pokud by se v návrhu objevila chyba, bude snadné jí znovu nasimulovat a opravit.

- Lze rekonstruovat skutečné mezivýsledky, které se během výpočtu objevují. Díky tomu je například možné na základě korelace mezi očekávanou a reálnou spotřebou zjistit, v kterém časovém okamžiku probíhá vyhodnocování jednotlivých rund.
- Deaktivovat jednotlivé ochrany bez nutnosti nahrávat jiný bitstream. Jednoduše se místo náhodných dat pošlou data, která ochrany eliminují, například nulové masky.

1.4.1 Deaktivace jednotlivých ochran

Dekompozice S-Boxu potřebuje pro svou činnost indexy řádků v pravdivostní tabulce, které se mají vzájemně zaměnit. Pro deaktivaci tedy stačí místo těchto indexů odeslat samé nuly. Tím budou oba indexy záměny ukazovat na stejný řádek, takže operace nebude mít žádný efekt.

Maskování lze deaktivovat opět odesláním nul místo náhodných dat. Z vlastností operace XOR plyne, že $x \oplus 0 = x$ a nemá tedy žádný efekt.

Odlišné je to v případě ochrany register precharge. Pro její deaktivaci je nutné odeslat otevřený text jako blok náhodných dat. Tím se dosáhne toho, že jak při skutečném šifrování, tak i při šifrování bloku náhodných dat bude vstupem stejná hodnota a budou stejné i mezivýsledky. Rozdíl mezi po sobě jdoucími takty bude stejný, jako by ochrana nebyla implementována (stále ale potrvá dvakrát tolik taktů).

Kapitola 2

Implementace dynamické rekonfigurace

Úkolem této práce je použít dynamickou rekonfiguraci u třech různých algoritmů, a u algoritmu AES dokonce dvěma způsoby (základním způsobem podle [1] a dále s využitím kompozitního tělesa). Celkem jde tedy minimálně o 4 různé implementace. Z těchto důvodů jsem hledal nějaké obecné řešení, které by se dalo použít u všech variant. Na základě vlastností jednotlivých algoritmů jsem stanovil několik požadavků, které by toto řešení mělo splňovat:

- Schopnost realizovat funkce různých šířek. SERPENT a PRESENT používá 4bitový S-Box, zatímco AES používá 8bitový S-Box. Navržené řešení by tedy mělo podporovat libovolnou šířku bijekce, a to i šířku, která přesahuje počet vstupních bitů CFGLUTu (používám 5vstupový CFGLUT).

- Mělo by být možné maskovat vstupní a výstupní hodnotu funkce.

- Řešení by mělo podporovat dekompozici S-Boxu, tedy mělo by být schopné náhodně modifikovat dvě bijekce tak, aby jejich složení dávalo stále původní S-Box. Tato funkce by měla být volitelná, ne vždy je to potřeba.

- Každý algoritmus používá jinou definici S-Boxu a také jiný celkový počet S-Boxů. Tyto parametry tedy musí být nastavitelné.

Na základě těchto požadavků jsem vytvořil 4 VHDL entity a architektury, jejichž propojením lze realizovat dynamickou rekonfiguraci. Tyto entity hojně využívají generické parametry, aby se daly přizpůsobit potřebám jednotlivých algoritmů. Toto řešení je v dalším textu označované jako V1.

Poté jsem ale zjistil, že by bylo dobré, aby řešení splňovalo ještě další dvě podmínky:

- Řešení by mělo být schopné realizovat dekompozici různých S-Boxů naráz. Tato podmínka je kvůli algoritmu SERPENT, který v každé rundě používá jinou definici S-Boxu. Jinými slovy, navržené řešení by mělo být schopné dekomponovat S-Boxy způsobem uvedeným na obrázku 1.10.

- Stupeň paralelismu by měl být volitelný. Mělo by být možné volit mezi rekonfigurací nenáročnou na velikost FPGA, snadno přeložitelnou a plně paralelní rekonfigurací, které stačí méně hodinových taktů k provedení rekonfigurace (samozřejmě za cenu vyšších nároků na FPGA). To umožní řešit situace, kdy se nepodaří vygenerovat bitstream pro cílové FPGA.

Již vytvořené řešení by bylo velmi těžko rozšiřitelné tak, aby splňovalo i tyto podmínky, obzvláště bez VHDL-2008, které není v Xilinx ISE 14.7 podporováno. Proto jsem se rozhodl vytvořit jednoduchý program (dále nazývaný DynReconfGen), který pro konkrétní potřeby vygeneruje VHDL kód realizující rekonfiguraci. Součástí této práce jsou obě varianty. Protože však nástroj DynReconfGen plně nahrazuje první variantu (V1), a navíc je v některý ohledech lepší, byl použit pro implementování všech algoritmů a varianta V1 nebude podrobně popisována. Jde v podstatě o slepou vývojovou větev. Na přiloženém DVD je ukázka implementace algoritmu PRESENT s použitím V1. Pro jiné implementace nebylo V1 nikdy použito.

Při prvních měřeních se ukázalo, že způsob rekonfigurace generovaný nástrojem DynReconfGen není v některých případech dostatečný. Umí sice modifikovat dvě bijekce tak, aby jejich složení dávalo stále původní bijekci, ale pouze v jednoduchém případě, kdy výstup jedné bijekce je přímo vstupem druhé. Nepodporuje složitější větvení, respektive rozvětvené signály umí ochránit pouze maskováním. Vzhledem ke špatným výsledkům implementace AESu používající kompozitní těleso jsem se domníval, že by lepší výsledky mohla poskytnout implementace, která by mezivýsledky nejen maskovala, ale i náhodně nahrazovala za jiné hodnoty (podobně, jako se to děje u dekompozice S-Boxu). Proto jsem vytvořil další nástroj, DynReconfGenV3, který nepracuje nad jednotlivými funkcemi nezávisle, ale pracuje nad celým schématem. Jako v předchozím případě jde o generátor VHDL kódu. Tento nástroj byl použit pouze u implementace AESu využívající kompozitní těleso. Jistě by jej bylo možné použít i u všech ostatních, nicméně nemělo smysl se zbytečně věnovat předělávání již existujícího funkčního kódu, které by nemělo žádný přínos.

2.1 Dynamická rekonfigurace V1

Toto řešení se skládá ze 4 VHDL entit

- `BijectionSource`

- `DecomposedBijectionSource`

- `CfglutControllerBody`
- `BijectionsSet`

a dále jednoho VHDL balíčku `PackageDynReconf`, který obsahuje některé konstanty a společné funkce.

2.1.1 BijectionSource a DecomposedBijectionSource

Jejich úkolem je (po aktivaci signálem `Gen`) postupně, v několika taktech, odvysílat řádky pravdivostní tabulky příslušné bijekce do entity `CfglutController`. V případě entity `BijectionSource` je tato pravdivostní tabulka vždy stejná a odpovídá pravdivostní tabulce realizované bijekce. Složitější je entita `DecomposedBijectionSource`, která odesílá dvě pravdivostní tabulky (je tedy připojena k dvěma entitám `CfglutController`) a náhodně modifikuje obě tabulky tak, aby mezivýsledek byl randomizován, ale dohromady tvořily původní bijekci.

Jinak řečeno, entita `BijectionSource` se použije v případě, že chceme pouze maskovat vstup a výstup bijekce. Pokud chceme bijekci i dekomponovat na dvě části, je nutné použít entitu `DecomposedBijectionSource`. Ta je o něco náročnější na prostředky FPGA, protože musí uchovávat obě pravdivostní tabulky (pro tento účel se používají dvě distribuované RAM). Obě entity mají generické parametry `Width` (šířka implementované bijekce) a `Definition` (počáteční pravdivostní tabulka, pole typu `integer`).

2.1.2 CfglutController

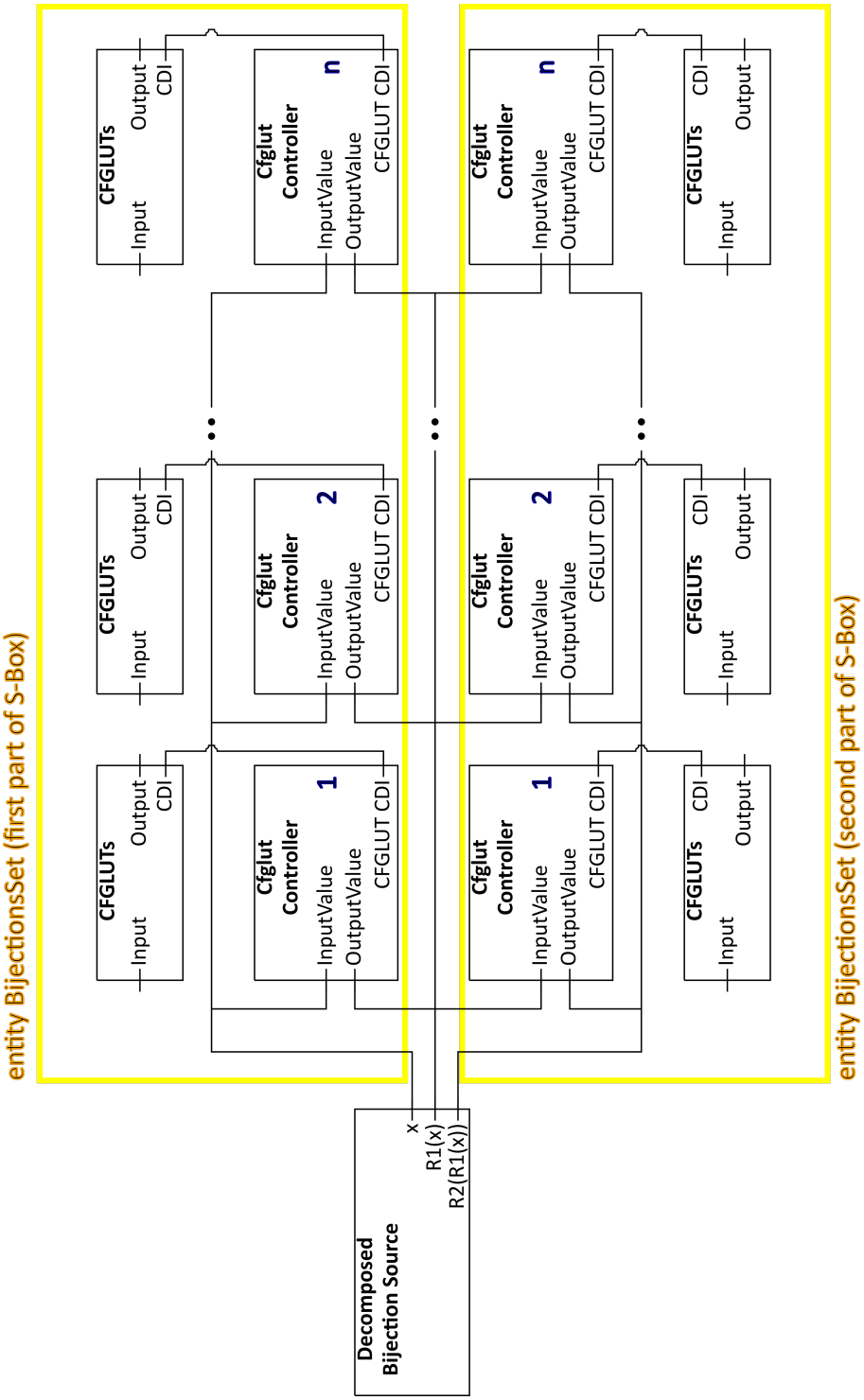
Vstupem této entity je pravdivostní tabulka. Ta je na základě náhodných masek přeuspořádaná tak, aby aby byla vstupní a výstupní hodnota bijekce maskována. Poté je tato tabulka odeslána do CFGLUTů. Pro uložení tabulky je opět použita distribuovaná RAM.

2.1.3 BijectionsSet

Tato entita se stará o propojení komponent `CfglutController` a `CFGLUT5`. Má generický parametr `BijectionsCount`, který určuje, kolik bijekcí se má realizovat (v jedné rundě je více S-Boxů).

2.1.4 Příklad použití pro dekompozici S-Boxu

Na obrázku 2.1 je schématicky znázorněno propojení popisovaných entit pro dekompozici S-Boxu. Šířka všech vodičů na obrázku odpovídá šířce bijekce (např. 4 bity v případě algoritmu PRESENT). Písmenem n je na obrázku označen celkový počet S-Boxů v jedné rundě (tedy $n = 64/4 = 16$ v případě algoritmu PRESENT). Komponenta `BijectionsSet` v horní polovině obrázku



Obrázek 2.1: Schéma znázorňuje propojení komponent při dekompozici S-Boxu.

realizuje první části S-Boxů, komponenta v dolní polovině pak druhé části S-Boxů. Obě části dohromady realizují původní S-Boxy. Komponenty CFGLUT jsou zapojeny na příslušné místo do rundy přes porty `Input` a `Output`. Komponenta `DecomposedBijectionSource` náhodně modifikuje obě bijekce a odesílá je do komponent `CfglutController`. Ty se pak starají o bitové maskování a odeslání konfigurace do CFGLUTů. Pro přehlednost nejsou na obrázku vyznačeny všechny porty komponent, chybí hodinový vstup a resetovací signál, řídící a stavové signály, vstupy pro náhodná data.

Počet taktů potřebných k odeslání pravdivostní tabulky z komponenty `DecommposedBijectionSource` odpovídá počtu řádků v pravdivostní tabulce (tedy 16 v případě algoritmu PRESENT). Poté se spustí odesílání konfigurace do CFGLUTů, které trvá stejný počet taktů (obě fáze tedy v případě algoritmu PRESENT trvají 32 taktů). První fázi lze provést během šifrování, protože neovlivňuje konfiguraci CFGLUTů.

2.2 Nástroj DynReconfGen

Tento nástroj je určen pro generování kódu dynamické rekonfigurace, a to na míru konkrétním požadavkům. Je obecně použitelný, lze jej použít v případech, kdy je potřeba:

- dekomponování jedné bijekce na dvě náhodné za sebou zapojené bijekcte
- implementování bijekce (ale obecně i libovolné funkce), jejíž vstup a výstup je ochráněn booleovským maskováním
- kombinaci obou výše uvedených možností

Vstupem tohoto programu je XML soubor definující parametry realizovaných bijekcí a výstupem je několik VHDL souborů, které stačí přidat do cílového projektu a propojit se zbytkem návrhu.

Tato kapitola se zabývá popisem tohoto nástroje. V první části je popsán z uživatelského pohledu, v druhé části je vysvětlen vnitřní princip fungování vygenerovaného kódu.

Poznámka k terminologii: Program byl původně určen pouze pro realizaci bijekcí, za určitých okolností však může fungovat obecně pro libovolnou funkci. Pojem bijekce je tak ve zdrojových kódech, konfiguračním XML, komentářích apod. použit chybně. Platí, že program lze použít k realizaci libovolné funkce, pokud není součástí dekomponovaného páru. Je-li součástí dekomponovaného páru, musí opravdu jít o bijekci.

2.2.1 Vstup programu

Vstupem programu je XML dokument s níže popsanou strukturou. Tento dokument je vlastně konfigurace, podle které se vygenerují příslušné zdrojové

Tabulka 2.1: Přepínače nástroje DynReconfGen

Přepínač	Popis
-i filename	Nastavuje umístění vstupního XML souboru. Tento argument je nepovinný. Pokud není uveden, použije se soubor input.xml v aktuální složce.
-o dirname	Nastavuje výstupní složku, do které budou uloženy vygenerované soubory. Tento argument je nepovinný, pokud není uveden, výstup se zapíše do složky DynReconfGen_Output, která je podsložkou aktuální složky.

kódy. V dalším textu bude tato konfigurace nazývána profil. Vstup a výstup se předává argumenty příkazové řádky, příslušné přepínače jsou popsány v tabulce 2.1

2.2.1.1 Kořenový element DynGenReconfInput

Tento element má jediný povinný atribut `Name`, který určuje název profilu. Ten bude použit pro pojmenování souborů, entit, portů a podobně.

2.2.1.2 Element Bijections

Tento element obsahuje jeden nebo více elementů `Bijection`. Každý z nich definuje jednu bijekci, která se má realizovat. Element `Bijection` má atributy popsané v tabulce 2.2

2.2.1.3 Element DecomposedPairs

Tento element obsahuje elementy `DecomposedPair`. Elementem `DecomposedPair` se vybírají bijekce, které se mají vzájemně náhodně modifikovat (dekompozice S-Boxu). Atributy jsou popsány v tabulce 2.4

Potomkem elementu `DecomposedPair` musí být právě jeden element `FirstPartBijection`, který určuje, která bijekce je první částí dekomponované bijekce, a dále jeden nebo více elementů `SecondPartBijection`, které ukazují na druhé části dekomponované bijekce. První část je vždy pouze jedna, druhých částí může být více (použité u algoritmu SERPENT, viz obrázek 1.10).

Tabulka 2.2: Atributy elementu `Bijection`

Atribut	Typ	Popis
`Name`	`string`	Název, který bude použit k pojmenování souvisejících objektů, zejména portů
`Width`	`integer` (> 0)	Šířka bijekce, tj. počet vstupních a zároveň výstupních bitů
`BitMasking`	`boolean`	Povolení bitového maskování na vstupu a výstupu bijekce
`NumberOfInstances`	`integer` (> 0)	Počet realizovaných kopií (instancí) této bijekce (např. počet S-Boxů paralelně v jedné rundě)
`PortsCount`	`boolean`	Počet portů k CFGLUTům. Tento parametr nastavuje míru paralelismu. Je-li tento atribut nastaven na hodnotu 1, CFGLUTy všech instancí budou konfigurovány jedním portem. Tento způsob je velmi nenáročný na zdroje, ale prodlouží proces rekonfigurace. Opačným případem je nastavení na stejnou hodnotu jako má atribut `NumberOfInstances`. V takovém případě má naopak každá instance vlastní port a všechny se rekonfigurují zároveň.
`Definition`	`string`	Název definice bijekce, vybírá pravdivostní tabulku. Nástroj má v sobě zabudované funkce uvedené v tabulce 2.3, nicméně lze do něj snadno přidat libovolné jiné definice.

Tabulka 2.3: Bijekce zabudované v nástroji DynReconfGen

Název	Šířka	Popis
`Identity_<W>`	`W`	Identita (`W` zastupuje kladné číslo, např. `Identity_4`)
`Increment_<W>`	`W`	Vrací vstup zvětšený o jedna (tato definice je určena spíše pro testování, `W` zastupuje kladné číslo, např. `Increment_4`)
`SerpentSBox_<N>`	4	SERPNET S-Box (`N` zastupuje index S-Boxu a musí být celé číslo z rozsahu 0 až 7, tedy například `SerpentSBox_0`)
`PresentSBox`	4	PRESENT S-Box
`AES_F<N>`	4	Dílčí operace při výpočtu AES S-Boxu s využitím kompozitního tělesa. (`N` zastupuje index operace a musí být celé číslo z rozsahu 1 až 6, tedy například `AES_F1`)
`AesSBox`	8	AES S-Box

Tabulka 2.4: Atributy elementu `DecomposedPair`

Atribut	Typ	Popis
`Name`	`string`	Název, který bude použit k pojmenování souvisejících objektů.
`SectionsCount`	`integer` (> 0)	Tento parametr určuje, do kolika sekcí budou rozděleny instance bijekce. Všechny instance v rámci jedné sekce jsou modifikovány stejným způsobem. Je-li tato hodnota nastavena na 1, jsou všechny instance modifikovány stejně a liší se pouze případným bitovým maskováním. Hodnota stejná jako počet instancí naopak znamená, že každá instance bude modifikována jiným způsobem. Z implementačních důvodů nemůže být tato hodnota větší než počet portů (jedním portem nelze konfigurovat instance z různých sekcí)
`SwapsCount`	`integer` (> 0)	Počet párů řádků, které budou při modifikace zaměněny.

2.2.2 Příklad vstupního XML souboru

Následující ukázkový XML dokument vygeneruje dynamickou rekonfiguraci pro ochránění algoritmu PRESENT.

```
1  <DynGenReconfInput Name="Present">
2    <Bijections>
3      <Bijection Name="R1"              Width="4"
4                 BitMasking="true"      PortsCount="32"
5                 NumberOfInstances="32" Definition="PresentSBox" />
6
7      <Bijection Name="R2"              Width="4"
8                 BitMasking="true"      PortsCount="32"
9                 NumberOfInstances="32" Definition="Identity_4" />
10   </Bijections>
11   <DecomposedPairs>
12     <DecomposedPair SectionsCount="32"
13                     Name="SBoxDecomp" SwapsCount="8">
14       <FirstPartBijection>R1</FirstPartBijection>
15       <SecondPartBijection>R2</SecondPartBijection>
16     </DecomposedPair>
17   </DecomposedPairs>
18 </DynGenReconfInput>
```

Toto je nejjednodušší případ, složitější je situace v případě algoritmu SERPENT. Pro něj zde z důvodu velikosti nebude uváděn celý vstupní soubor, ale je zde naznačen postup pro jeho vytvoření:

- Kód na řádcích 7-9 musí být zopakován celkem osmkrát, pokaždé s jiným názvem (např. `R2_0`, `R2_1` atd.) a jinou hodnotou atributu Definition (`SerpentSBox_0`, `SerpentSBox_1` atd.).

- Řádek 15 musí být zopakován také 8 krát, pro názvy zvolené v předchozím kroku (např. `R2_0`, `R2_1` atd.).

2.2.3 Výstup programu

Výstupem programu je několik VHDL souborů. Některé z nich jsou statické, tj. jejich obsah vůbec nezávisí na vstupním XML. Pokud bude v rámci jednoho projektu použit tento nástroj vícekrát pro různé profily, tyto soubory jsou sdíleny všemi konfiguracemi a nesmí být v projektu více než jednou (došlo by ke kolizi názvů). Názvy dynamických souborů závisí na názvu profilu, který je v následujícím textu nahrazen slovem `NAME`.

Pro použití vygenerovaných souborů je nutné vložit je do projektu a dále vytvořit instanci entity `DynReconfWrapper_NAME`. Šablona pro vytvoření instance je v souboru `Readme_NAME.txt`.

2.2.4 Rozhraní entity DynReconfWrapper

Rozhraní entity `DynReconfWrapper_NAME` závisí na konkrétním profilu. Níže je uvedeno rozhraní pro výše uvedený ukázkový profil (kód je částečně upraven pro lepší čitelnost, významem je zcela ekvivalentní).

```
1  entity DynReconfWrapper_Present is
2  port (
3    Clk          : in std_logic;
4    Rst          : in std_logic;
5    Calculate    : in std_logic;
6    Load         : in std_logic;
7    IsReady      : out std_logic;
8
9    --Bijection R1;
10   R1Input   : in std_logic_vector(4 * 32 - 1 downto 0);
11   R1Output  : out std_logic_vector(4 * 32 - 1 downto 0);
12   --Bijection R1, bit masking;
13   R1MaskIn  : in std_logic_vector(4 * 32 - 1 downto 0);
14   R1MaskOut : in std_logic_vector(4 * 32 - 1 downto 0);
15
16   --Bijection R2;
17   R2Input   : in std_logic_vector(4 * 32 - 1 downto 0);
18   R2Output  : out std_logic_vector(4 * 32 - 1 downto 0);
19   --Bijection R2, bit masking;
20   R2MaskIn  : in std_logic_vector(4 * 32 - 1 downto 0);
21   R2MaskOut : in std_logic_vector(4 * 32 - 1 downto 0);
22
23   --Decomposition pair SBoxDecomp;
24   SBoxDecompSwapIndicies
25         : in work.PackageDynReconf_Present.SBoxDecompIndiciesArraySet
26 );
27 end entity DynReconfWrapper_Present;
```

V rozhraní je pro každou bijekci (element `Bijection`) vytvořen port pro vstup a výstup bijekce

- `<BIJECTION_NAME>Input`
- `<BIJECTION_NAME>Output`

Oba porty jsou typu `std_logic_vector`, mají stejnou šířku, která je rovna součinu počtu instancí a šířky jedné bijekce. Je-li na bijekci zapnuto maskování, jsou vygenerovány ještě porty pro vstupní a výstupní masku

- `<BIJECTION_NAME>MaskIn`
- `<BIJECTION_NAME>MaskOut`.

Masky jsou brány v úvahu pouze během rekonfigurace. Naopak výstup dává správnou hodnotu pouze když rekonfigurace neprobíhá.

Pro každý dekomponovaný pár je vygenerován ještě port

- `<PAIR_NAME>SwapIndicies`,

který určuje řádky v pravdivostní tabulce bijekce, jenž budou při rekonfiguraci zaměněny. Typem tohoto portu je dvoudimenzionální pole, kde první dimenze určuje sekci a má rozsah 0 až `SectionsCount - 1`. Druhá dimenze pak určuje index páru, který má rozsah 0 až `SwapsCount - 1`. Prvkem tohoto pole je dvojice vektorů. Jejich šířka odpovídá šířce bijekce a jde vlastně o ukazatele na řádky pravdivostní tabulky, které budou zaměněny.

V reálném použití jsou na vstupy

- `<BIJECTION_NAME>MaskIn`
- `<BIJECTION_NAME>MaskIn`
- `<BIJECTION_NAME>SwapIndicies`

přivedena náhodná data. Pro testování a vyhodnocování může být některá z ochran vypnuta nebo zeslabena přivedením ne úplně náhodných dat. Samé nuly jako maska vypnou maskování, samé nuly na portu `<PAIR_NAME>SwapIndicies` deaktivují modifikace bijekcí.

Kromě těchto portů je pak součástí rozhraní i hodinový vstup `Clk`, resetovací signál `Rst`. Jediný stavový signál `IsReady` typu `std_logic` má hodnotu `1`, pokud je komponenta připravena na spuštění rekonfigurace. Řídící signály jsou dva, `Calculate` a `Load`. První z nich spustí první fázi rekonfigurace. Během této fáze není nijak ovlivněna konfigurace CFGLUTů, lze jí tedy spustit i během šifrování. Tato fáze je zodpovědná za modifikaci bijekcí, které jsou součástí dekomponovaného páru. Druhá fáze rekonfigurace se spustí signálem `Load`. Během této fáze jsou pravdivostní tabulky nahrány do CFGLUTů. Hodnoty masek se čtou během této fáze. Dokončení obou fází lze ověřit stavovým signálem `IsReady`. Po resetu jsou všechny CFGLUTy neinicializované a nedávají správné výsledky. Před prvním použitím je tedy vždy nutné spustit rekonfiguraci.

2.2.5 Testování vygenerovaného kódu

Součástí výstupu je i VHDL testbench v souboru `Tests/DynReconfTestBench_<NAME>.vhd`. Pokud simulátor nepodporuje `CFGLUT5`, je nutné nahradit instance komponenty `CFGLUT5` za instance komponenty `FakeCFGLUT5` (která je také výstupem nástroje `DynReconfGen`). Toto nahrazení je nutné udělat v souboru `CfglutTableSetBody.vhd`.

Testbench obsahuje stimulační proces, který k top-level entitě připojí náhodná data (tj. náhodné masky, náhodné indexy řádků k záměně a náhodné vstupy bijekcí), spustí rekonfiguraci a po jejím dokončení ověří, zda realizované bijekce vrací očekávané výstupy. Tento test je opakován stokrát a v případě

úspěchu je simulace ukončena vypsáním zprávy "Successfully completed". Cílem tohoto testu je ověřit, zda vygenerovaný VHDL kód se chová tak, jak je očekáváno.

2.2.6 Popis vygenerovaných souborů

Statické výstupní soubory:

- `PackageDynReconfCommon.vhd` je balíček, který obsahuje některé společné funkce a typy
- `CfglutTableSetBody.vhd` je architektura, která vytváří potřebné CFGLUTy pro jednu nebo více instancí bijekce. Vstupem je jeden konfigurační port a samotný vstup bijekce. Výstupem je pak výstup bijekce. Komponenta realizuje buď pouze jednu bijekci (v případě zcela paralelního režimu) nebo více bijekcí (částečně paralelní režim nebo zcela sériový režim). Počet vytvořených CFGLUTů závisí na šířce bijekce. Je-li bijekce širší než počet vstupních bitů CFGLUTu, je vytvořeno zapojení do kaskády popisované naznačené na obrázku 1.3.
- `CfglutTableSet.vhd` je entita pro předchozí architekturu.
- `FakeCFGLUT5.vhd` je komponenta, která má stejné rozhraní jako `CFGLUT5` a je vytvořena tak, aby bylo totožné i její chování. Pokud simulátor nepodporuje `CFGLUT5`, je možné jej pro simulaci a testování nahradit touto entitou. Tato entita není určena pro syntézu, syntézní nástroj v ní nerozpozná `CFGLUT5` a překlad se tak vůbec nepodaří, nebo bude velmi náročný a výsledek bude potřebovat velké množství zdrojů v FPGA.

Výstupní soubory přímo závislé na konkrétním profilu:

- `DynReconfCoreBody_<NAME>.vhd` obsahuje architekturu, která je jádrem samotné rekonfigurace. Uchovává pravdivostní tabulky, stará se o modifikace bijekcí (při dekompozici) a zajišťuje jejich odeslání do CFGLUTů (konkrétněji do komponenty `CfglutTableSet`) .
- `DynReconfWrapperBody_<NAME>.vhd` obsahuje architekturu, která zapouzdřuje jednu instanci komponenty `DynReconfCoreBody_<NAME>` a jednu nebo více komponent `CfglutTableSet`. Tato architektura je určena k instancování v cílovém projektu a zpřístupňuje uživateli požadovanou dynamickou rekonfiguraci prostřednictvím jednoduchého a přehledného rozhraní.
- `DynReconfCore_<NAME>.vhd` a `DynReconfWrapper_<NAME>.vhd` jsou soubory obsahující entity k výše uvedeným architekturám.

- `PackageDynReconf_<NAME>.vhd` je VHDL balíček, který obsahuje definice souvisejících konstant, typů a funkcí. Mezi konstanty patří například šířka všech bijekcí, počty instancí a portů, počet záměn u dekomponovaných bijekcí (`SwapsCount`) a v neposlední řadě samotnou definici všech bijekcí, tj. jejich počáteční pravdivostní tabulku. Pro uživatele může být zajímavá funkce `Init<PAIR_NAME>SwapIndiciesArraySet`, která dokáže z bitového vektoru příslušné délky inicializovat port `SwapIndicies` komponenty `DynReconfWrapper_<NAME>`.

- `Readme_<NAME>.txt` obsahuje šablonu pro instancování komponenty `DynReconfWrapper` a další podrobnosti, například mapování mezi porty a instancemi.

- `Tests/DynReconfTestBench_<NAME>.vhd` je testbench určený k otestování správného chování vygenerovaného kódu.

2.2.7 Vnitřní implementace architektury DynReconfCoreBody

Tato architektura obsahuje pro každý konfigurační port jednu distribuovanou RAM, která uchovává pravdivostní tabulku příslušné bijekce. Po zahájení první fáze rekonfigurace (signálem `Calculate`) je spuštěna modifikace dekomponovaných párů. Pro snazší popis tohoto procesu uvažujme dvě bijekci R_1 a R_2, které tvoří dekomponovaný pár, tj. jejich složením $R_2(R_1(x))$ vznikne původní bijekce. Postup modifikace je pak následující:

- V prvním taktu je přečten první z vybrané dvojice řádků v R_1 a na základě této hodnoty i první z dvojice řádků v R_2.

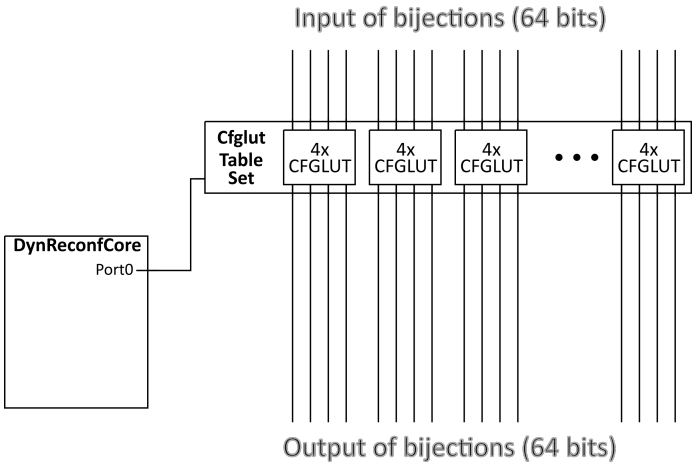
- V druhém taktu je přečten druhý z vybrané dvojice řádků v R_1, který je hned zapsán do prvního řádku vybrané dvojice. Stejný proces proběhne i nad tabulkou R_2.

- V třetím taktu je zapsána hodnota přečtená v prvním taktu do druhého řádku dvojice v R_1. Stejný procese proběhne i nad tabulkou R_2.

Z uvedeného plyne, že počet taktů potřebných k dokončení této fáze je přibližně `3 * SwapsCount`.

Ve druhé fázi rekonfigurace (signálem `Load`) jsou tabulky nahrány do CFGLUTů. Během toho se aplikuje i maskování, tj. k adrese řádku je přičtena (XOR) vstupní maska a výstupní maska je přičtena k hodnotě řádku pravdivostní tabulky. Počet potřebných taktů je roven počtu řádků pravdivostní tabulky.

Je-li v jednom profilu definováno více bijekcí s různou šířkou, nebo více dekomponovaných párů s různým počtem záměn (`SwapsCount`), potřebný počet taktů k dokončení první nebo druhé fáze je větší z nich.



Obrázek 2.2: Schéma komponenty vygenerované nástrojem DynReconfGen, úroveň paralelismu 1 (na obrázku nejsou znázorněny všechny CFGLUTy, v komponentě `CfglutTableSet` jich je celkem 16 čtveřic)

Není možné, aby více portů sdílelo jednu distribuovanou RAM, protože během odesílání tabulek do CFGLUTů je prováděno maskování, tedy je nutné adresovat řádky v závislosti na konkrétní hodně masky. Pokud je však maskování vypnuto, jsou konfigurace všech instancí připojených k jednomu portu stejné, a proto jsou nahrávány paralelně. Při vypnutém maskování nemá smysl používat paralelismus (tj. používat `PortsCount` větší než 1) vyjma situace, kdy počet sekcí (`SectionsCount`) je větší než 1. Jedním portem totiž nelze obsluhovat instance z různých sekcí.

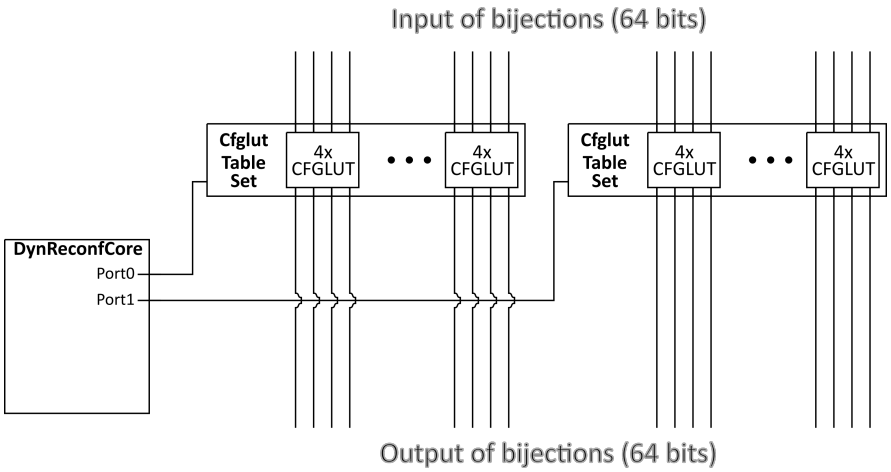
2.2.8 Porovnání různých stupňů paralelismu

Obrázky 2.2 až 2.5 demonstrují vnitřní zapojení vygenerované entity pro různé stupně paralelismu. Ve všech případech byl nástrojem DynReconfGen vygenerován kód pro 16 instancí bijekce šířky 4 bity.

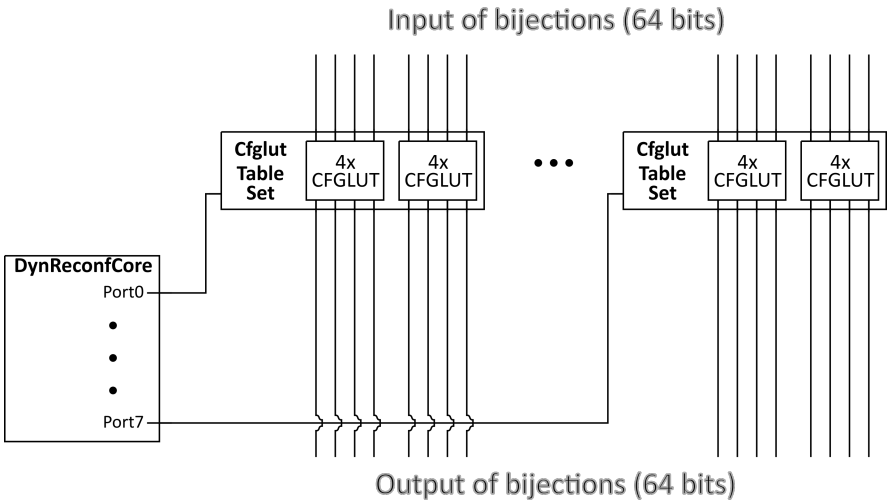
V případě obrázku 2.2 byl stupeň paralelismu nastaven na hodnotu 1. Všechny CFGLUTy jsou tedy připojeny přes jeden port a nahrávání jejich konfigurace probíhá sekvenčně. Nakonfigurování jedné instance trvá 16 taktů (16 řádků pravdivostní tabulky), celkem tedy $16 * 16 = 256$ taktů. Výhodou však je, že komponenta `DynReconfCore` je nenáročná, uvnitř obsahuje pouze jednu distribuovanou RAM 16x4 bitů.

Opakem je schéma na obrázku 2.5, kde je použit největší možný stupeň paralelismu (maximum je počet instancí). Nahrání konfigurace trvá 16 taktů, což je vykoupeno náročnou komponentou `DynReconfCore`.

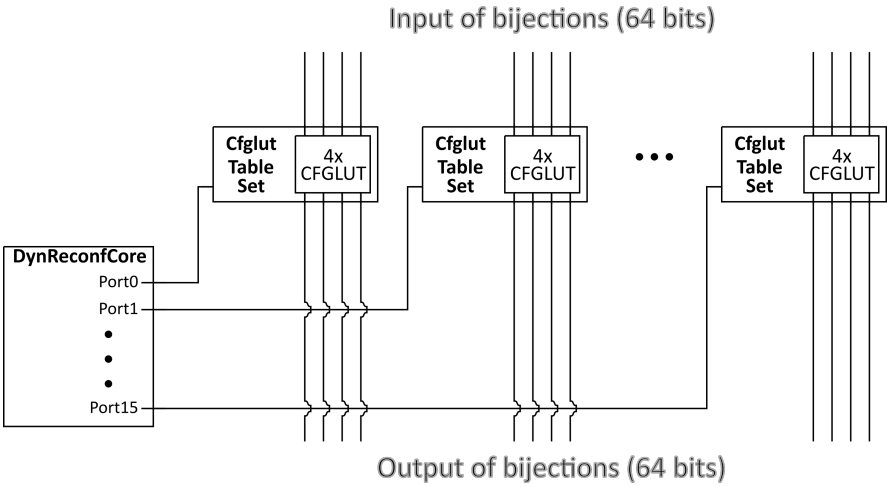
Úroveň paralelismu lze nastavit na libovolné celé číslo z rozsahu od 1 do počtu instancí, tedy v tomto případě od 1 do 16. Pokud však není počet



Obrázek 2.3: Schéma komponenty vygenerované nástrojem DynReconfGen, úroveň paralelismu 2 (na obrázku nejsou znázorněny všechny CFGLUTy, v každé komponentě `CfglutTableSet` je jich celkem 8 čtveřic)



Obrázek 2.4: Schéma komponenty vygenerované nástrojem DynReconfGen, úroveň paralelismu 8 (na obrázku nejsou znázorněny všechny komponenty `CfglutTableSet`, celkem jich je 8)



Obrázek 2.5: Schéma komponenty vygenerované nástrojem DynGenReconf, úroveň paralelismu 16 (na obrázku nejsou znázorněny všechny komponenty CfglutTableSet, celkem jich je 16)

instancí dělitelný stupněm paralelismu, k některým portům bude připojeno méně instancí než k ostatním a nebudou tak plně využity.

2.3 Nástroj DynReconfGenV3

Ačkoliv nástroj DynReconfGen dobře posloužil při implementaci algoritmů AES (včetně verze používající kompozitní těleso), PRESENT a SERPENT, objevil se po prvních měřeních jistý nedostatek a začal jsem uvažovat o vytvoření programu plnící stejný úkol, ale jenž je zcela jinak koncipován. Původní nástroj DynReconfGen pracuje na úrovni jednotlivých bijekcí (resp. funkcí). Vrátíme-li se k schématu na obrázku 1.12, tak s pomocí nástroje DynReconfGen lze vygenerovat VHDL kód realizující jednotlivé funkce v tomto schématu. Samotné propojení dílčích funkcí ale již musí být provedeno ručně uživatelem. Mezivýsledky mezi funkcemi jsou ochráněny maskováním. Nemohou už ale být ochráněny náhodnou záměnou hodnot, tak jak je tomu v případě dekompozice S-Boxu, při které jsou náhodně zaměňovány řádky v pravdivostních tabulkách.

Naproti tomu, nástroj DynReconfGenV3 pracuje nad celým schématem. Jeho vstupem je tedy schéma podobné tomu na obrázku 1.12. Schéma se musí skládat z funkcí a registrů, jiné prvky v současné době nejsou podporovány. Program pro takové schéma vygeneruje VHDL entitu, jejíž rozhraní tvoří pouze primární vstupy a výstupy zadaného schématu (samozřejmě kromě stavových a kontrolních signálů a dále také náhodných dat pro rekonfiguraci). Propojení dílčích funkcí uvnitř schématu zajistí program sám. Pro každou dílčí funkci lze zvolit způsob ochrany výstupní hodnoty. Na výběr je mezi

maskováním, překladem hodnot, nebo kombinací obojího. Překladem hodnot se v následujícím textu myslí proces, jehož vstupem jsou dva n-bitové vektory. Při překladu jsou pak výstupní hodnoty v pravdivostní tabulce, které se rovnají jednomu z těchto dvou vektorů, nahrazeny druhým z těchto vektorů. Jinými slovy, změní se tím způsob kódování hodnoty, význam zůstane stejný. Dílčí funkce, do kterých takto ochráněná hodnota vstupuje pak samozřejmě musí být upraveny tak, aby nedošlo ke změně výsledku, tedy aby nové kódování bylo správně interpretováno. O to se program postará automaticky, respektive vygeneruje takový VHDL kód, který se o to postará. Ve zdrojových kódech se pro označení této operace používá název "translation".

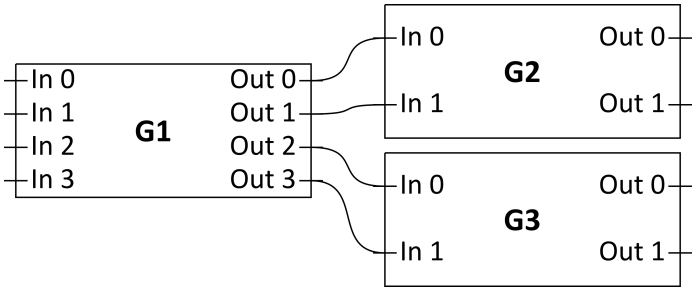
Situace v případě překladu hodnot je o něco složitější než u maskování. Maskování je vlastně operace XOR na úrovni jednotlivých bitů, každý jeden bit hodnoty je XORován s právě jedním bitem náhodné masky. Naproti tomu překlad hodnot pracuje na celém vektoru a k nahrazení dojde pouze pokud celý vektor je roven vstupnímu náhodnému vektoru. To není problém, pokud všechny výstupní bity jedné dílčí funkce vstupují společně do následující dílčí funkce. Problém nastane, pokud pouze část výstupních bitů vstupuje do další dílčí funkce.

Například, uvažujme schéma na obrázku 2.6, na kterém jsou tři dílčí funkce G1, G2, G3. Každý vodič v tomto schématu představuje jeden bit. Uvažujme, že chceme udělat překlad výstupu funkce G1 a náhodné vektory pro překlad jsou $s_1 = 0000$ a $s_2 = 1111$. Pokud bychom všechny výstupní hodnoty 0000 v pravdivostní tabulce přeložili na 1111 a obráceně, neexistuje způsob jak modifikovat funkce G2 a G3 tak, aby byla zachována správnost výsledku. Do funkce G2 totiž vstupují pouze dva bity z výstupu funkce G1. Pokud na vstup funkce G2 přijde hodnota 00, tak může jít o dva odlišné případy:

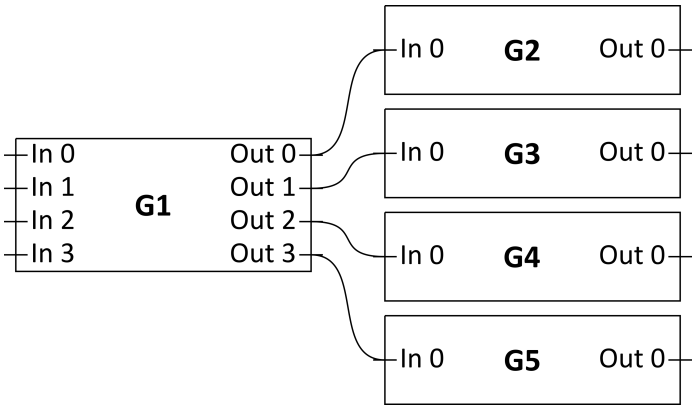
- Celým výstupem funkce byla hodnota 0000, tedy původní význam hodnoty je 1111 a funkce G2 by vstup měla interpretovat jako 11
- Celým výstupem funkce byla třeba hodnota 1100, která má svůj původní význam a funkce G2 by vstup měla interpretovat jako 00

Nelze tedy najít konfiguraci funkcí G2 a G3. Z tohoto důvodu je nutné provádět překlad pouze na podmnožině výstupních bitů. V tomto případě nezávisle překládat výstupní bity na pozicích 0 a 1 (podle páru dvoubitových náhodných vektorů) a výstupní bity na pozicích 2 a 3 (opět podle páru dvoubitových náhodných vektorů).

Díky tomu, že program zná celé schéma, postará se o správné rozdělení výstupních bitů do skupin. Zvláštní případ je vyobrazen na schématu 2.7. Zde je nutné rozdělit výstupní bity do čtyř skupin, tedy každý bit do samostatné skupiny. Toto je zvláštní případ, kdy překlad je v podstatě ekvivalentní maskování.



Obrázek 2.6: Překlady hodnot při větvení, příklad 1

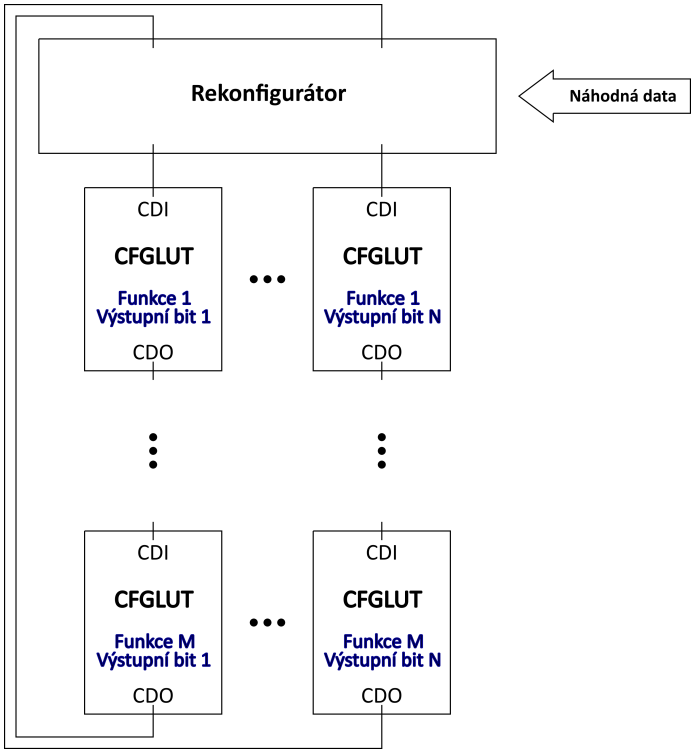


Obrázek 2.7: Překlady hodnot při větvení, příklad 2

2.3.1 Výstup programu

Výstupem programu je VHDL kód. Na rozdíl od předchozího nástroje DynReconfGen jde pouze o jeden dlouhý soubor obsahující jeden balíček (VHDL package), jednu entitu a k ní příslušnou architekturu. Rozhraní entity obsahuje řídící a stavové signály (hodinový signál, reset, ovládání rekonfigurace), náhodná data pro rekonfiguraci a samozřejmě primární vstupy a výstupy realizovaného schématu.

Trochu odlišný je způsob připojení CFGLUTů. Ty jsou propojeny do řetězců tak, že vstup CDI je připojen k výstupu CDO předchozícho CFGLUTu (resp. k CFGLUTu, který realizuje předchozí funkci v řetězci). Řetězců může být jeden nebo více. Uživatel si jednotlivě pro každou funkci definuje, do kterého řetězce má být připojena a v jakém pořadí. Díky tomu může volit úroveň paralelismu. Čím je řetězec kratší, tím je rekonfigurace rychlejší, ale za cenu větších nároků na FGPA. Způsob připojení CFGLUTů je znázorněn na obrázku 2.8



Obrázek 2.8: Propojení CFGLUTů vytvořené nástrojem DynReconfGenV3

2.3.2 Aktuální stav

Tímto programem byl vygenerován kód pro další verzi AESu používající kompozitní těleso. Tato verze je označována jako pokročilá (ve zdrojových kódech Advanced, nebo Adv).

Tento nástroj byl vyvíjen až v samém závěru práce a nelze jej považovat za dokončený. Ostatně jde o poměrně velký úkol, na kterém by se jistě našlo dost práce pro jednu další diplomovou práci. Nyní je asi nejpalčivějším nedostatkem načítání vstupu, které není prozatím nijak vyřešeno. V současné verzi je vstup přímo zapsán do zdrojových kódů programu. Myslím, že tento nástroj může být užitečným pomocníkem při výzkumu ochran založených na dynamické rekonfiguraci, a proto by mělo smysl pokračovat v jeho vývoji.

2.4 Shrnutí

Během této práce byly vytvořeny postupně tři různé způsoby, jak realizovat dynamicky se rekonfigurující zapojení. Motivací pro každý další způsob bylo odstranit nalezené nedostatky předchozího řešení.

První řešení, V1, je tvořené několika VHDL komponentami, které jsou navržené tak, aby byly obecně použitelné. Druhé řešení, DynReconfGen, je

generátor VHDL kódu, Umožňuje navíc volit úroveň paralelismu a řešit tak problémy s nemožností přeložit implementaci pro cílové FPGA. Třetí řešení, DynReconfGen, na rozdíl od předchozího napracuje nad dílčími funkcemi, ale jeho vstupem je celé zapojení.

Kapitola 3

Implementace šifrovacích algoritmů

V této části jsou popisovány implementační detaily. Je zde popsána struktura projektu a ve stručnosti nastíněn význam jednotlivých zdrojových souborů. Navržené řešení je konfigurovatelné mnoha parametry, jejichž význam je také vysvětlen v této kapitole.

3.1 Struktura projektu

Zdrojové soubory jsou rozděleny do 4 složek: `Aes`, `Serpent`, `Present` a `Common`. První tři jmenované obsahují soubory související s jednotlivými šifrovacími algoritmy. Složka `Common` obsahuje společné soubory. Jde zejména o UCF soubor a obálkovou entitu, která řídí šifrování a zajišťuje přenos dat po sériové lince. V dalším textu je ve stručnosti popsán význam jednotlivých souborů a parametrů, které lze u implementací volit.

V dalším textu je podrobněji popsána implementace jednotlivých algoritmů. Vzhledem k tomu, že jsou všechny implementace velmi podobné, je důkladněji popsán pouze PRESENT, u ostatních implementací jsou zmíněny jenom rozdíly oproti PRESENTu.

3.2 PRESENT

3.2.1 Konfigurovatelné parametry

U této implementace lze nastavit následující 4 parametry:

- `Mode`, kterým se volí délka šifrovacího klíče. Možné hodnoty jsou `PresentMode_Key80` a `PresentMode_Key128`
- `PresentProtectionMode`, kterým se vybírá způsob ochrany proti úniku informace. Možné hodnoty jsou `PresentProtection_Basic` a `Present-`

`Protection_BasicPar`. Liší se pouze tím, že v druhém případě se provádí rekonfigurace jednotlivých S-Boxů paralelně a jednotlivé S-Boxy jsou rekonfigurovány nezávisle na sobě (tj. bijekce $R1$ a $R2$ jsou v jednotlivých S-Boxech modifikovány podle jiných náhodných dat).

Z tohoto důvodu je v případě paralelní verze potřeba více náhodných bitů pro provedení rekonfigurace.

- `EnableRegPrecharge`, kterým se zapíná/vypíná ochrana Register precharge.
- `WriteEnMode`, která nastavuje režim zápisu do stavového registru. Je-li nastaven na `SleepWhenInactive`, zápis probíhá pouze při šifrování. Při volbě `FreeRunning` je zápis prováděn vždy, po celou dobu běhu FPGA.

Pro měření se používá tato konfigurace:

- `Mode` = `PresentMode_Key80`
- `PresentProtectionMode` = `PresentProtection_BasicPar`
- `EnableRegPrecharge` = 1

3.2.2 Zdrojové soubory

Hlavní komponenta se jmenuje `Present`. Ta využívá komponentu `PresentKeySchedule`, která je zodpovědná za výpočet jednotlivých rundovních klíčů, a dále komponentu `PresentPermutation`, která představuje kombinační logiku realizující permutaci, tedy lineární část rundy.

Dynamickou rekonfiguraci zajišťují soubory ve složce `DynReconf` nebo `DynReconfPar`. Obsah těchto složek je vygenerován nástrojem `DynReconfGen`.

Součástí je i jeden testbench ve složce `Tests`, kterým je možné spustit šifrování pro jeden konkrétní vstup (tento testbench nekontroluje správnost výsledku, kontrolu je nutné provést ručně).

Soubor `PackagePresent.vhd` pak obsahuje pomocné funkce, definici S-Boxu apod. V tomto souboru je také definován význam jednotlivých bitů bloku náhodných dat. Tedy je v něm uvedeno, která část bloku náhodných dat je použita jako maska, data pro modifikaci bijekcí $R1$ a $R2$, nebo jako náhodná data pro ochranu register precharge.

```
1 subtype  PresentRandomField_Mask1
2                 is natural range 63  downto 0;
3 subtype  PresentRandomField_Mask2
4                 is natural range 127 downto 64;
5 subtype  PresentRandomField_RandomBlock
6                 is natural range 191 downto 128;
7 subtype  PresentRandomField_SwapIndicies
8                 is natural range 255 downto 192;
```

3.3 SERPENT

- Lineární část rundy (lineární transformace) není v samostatné komponentě, ale je funkcí v souboru `PackageSerpent.vhd`.

- Jsou možné celkem 3 způsoby ochrany, oproti PRESENTu je zde navíc režim `SerpentProtection_BasicParLite`. Ten nabízí pouze částečný paralelismus při rekonfiguraci. Vždy dva sousední S-Boxy tvoří pár a jsou rekonfigurovány stejně po sobě. Páry jsou pak rekonfigurovány paralelně. Rekonfigurace tedy trvá dvakrát déle než u čistě paralelního režimu, ale je méně náročná. Tento režim byl přidán z důvodu nemožnosti přeložit čistě paralelní verzi pro desku Sakura-G.

- Implementace podporuje pouze délky klíče dělitelné 8.

- Pro měření se používá konfigurace se 128bitovým klíčem, režimem `SerpentProtection_BasicParLite` a zapnutým register precharge.

3.4 AES

- Lineární části rundy (ShiftRows a MixColumns) nejsou v samostatné komponentě, ale jsou funkcemi v souboru `PackageSerpent.vhd`.

- Je zde navíc soubor `PackageCompositeField.vhd`, který obsahuje definici izomorfismu do kompozitního tělesa, typy pro snazší práci s prvky konečného tělesa a podobné.

- Podporovány jsou všechny povolené délky klíče (128, 192, 256)

- Pro měření se používá konfigurace se 128bitovým klíčem a zapnutým register precharge.

3.4.1 Způsoby ochrany

Existuje celkem 6 způsobů ochrany. Je možné volit mezi klasickým způsobem jako u předchozích algoritmů a způsoby využívající kompozitní těleso k výpočtu hodnoty S-Boxu.

U klasického způsobu je opět k dispozici paralelní a sériový režim.

Způsoby `AesProtection_CompositeField` a `AesProtection_CompositeField` využívají kompozitní těleso a jsou vytvořené nástrojem DynReconfGen.

Způsoby `AesProtection_CompositeField_ADVANCED` a `AesProtection_CompositeField_ADVANCEDRegs` využívají také kompozitní těleso, ale jsou vytvořené nástrojem DynReconfGenV3. Verze s přídomkem `Regs` pak obsahuje navíc vložené registry pro eliminaci hazardů.

3.4.2 AES, výpočet S-Boxu v kompozitním tělese

Trochu podrobnější popis si zaslouží výpočet S-Boxu (respektive multiplikativní inverze) v kompozitním tělese. Výpočet vychází ze způsobu rozebíraného v části 1.3. Na obrázku 3.1 je schématicky znázorněn postup výpočtu inverze. Operace násobení je pak dále rozebrána na dílčí operace (obrázek 3.2).

3.4.2.1 Další volitelné registry při výpočtu inverze

Během prvních měření se ukázalo, že informace při šifrování prokazatelně uniká. Domníval jsem se, že únik by mohly způsobovat hazardy. Abych mohl tuto domněnku experimentálně potvrdit či vyvrátit, vložil jsem do výpočtu registry, které zabrání šíření hazardů (ale zvýší počet potřebných taktů). Registry musí být vloženy do všech větví rovnoměrně tak, aby byla zachována správnost výsledku. Registry jsou rozděleny do skupin pojmenovaných písmeny A - K a mohou být jednotlivě vloženy nebo vyjmuty nastavením konstant (`CompositeFieldSBox_RegA`, `CompositeFieldSBox_RegB` atd.) v souboru `PackageAes.vhd`. Díky tomu lze zkoumat jednotlivé varianty a experimentálně zjistit, které registry jsou přínosné.

Jsou-li volitelné registry aktivovány, příslušně se prodlouží potřebný počet taktů. V prvním taktu je na vstup rundy přiveden blok náhodných dat, ve druhém taktu pak otevřený text. To je stejné, jako u všech ostatních implementací. Rozdíl je až v dalších taktech, které jsou důsledkem vložení dodatečných registrů. V dalších taktech je opět na vstup připojen blok náhodných dat, stejný jako v prvním taktu. Jinými slovy, ve druhém taktu je do rundy přiveden otevřený text, v ostatních taktech pak blok náhodných dat. Z časového pohledu jsou tedy skutečná data obalena z obou stran náhodnými daty.

3.4.2.2 Dekompozice identity

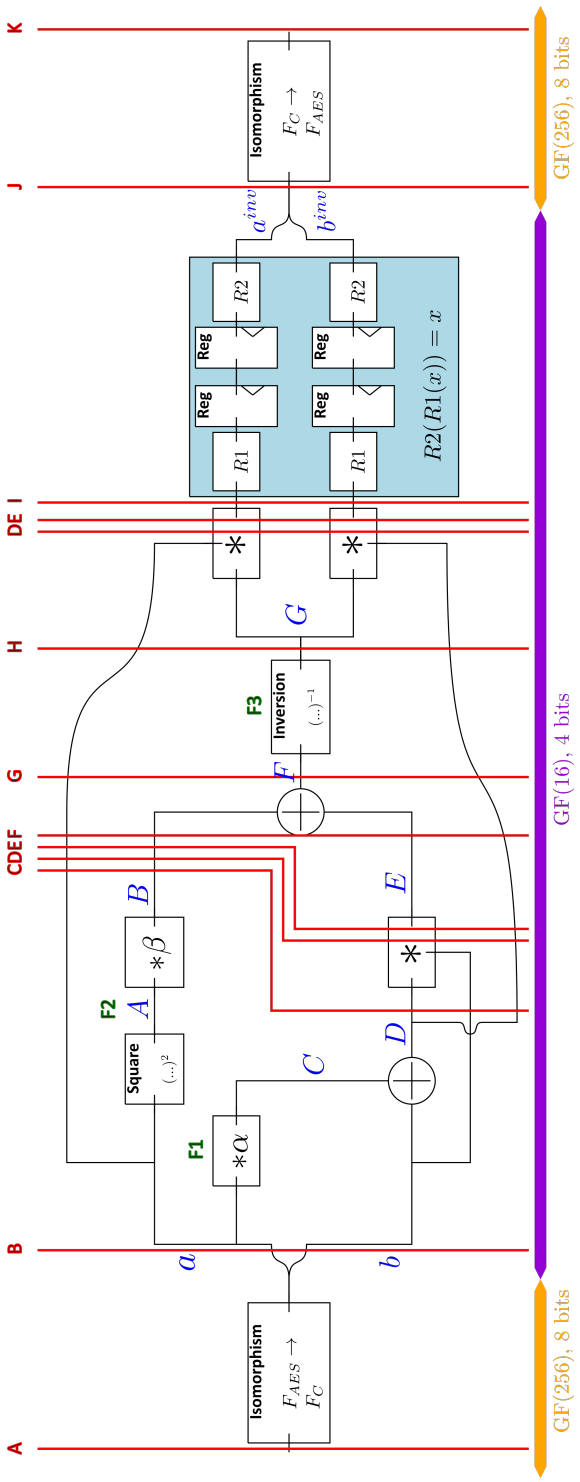
Protože se schéma výpočtu inverze větví, nelze zde aplikovat dekompozici nějaké bijekce tak jako v ostatních implementací. Proto je do výpočtu vloženo identické zobrazení, které neovlivní výpočet, ale lze jej dekomponovat a mezi dvě vzniklé bijekce vložit stavový registr tak, jako je to u předchozích implementací. Měřením je možné zjistit, zda toto zesložitění má nějaký přínos.

Tato dekompozice společně se stavovými registry je na obrázku 3.1 v modrém obdélníku.

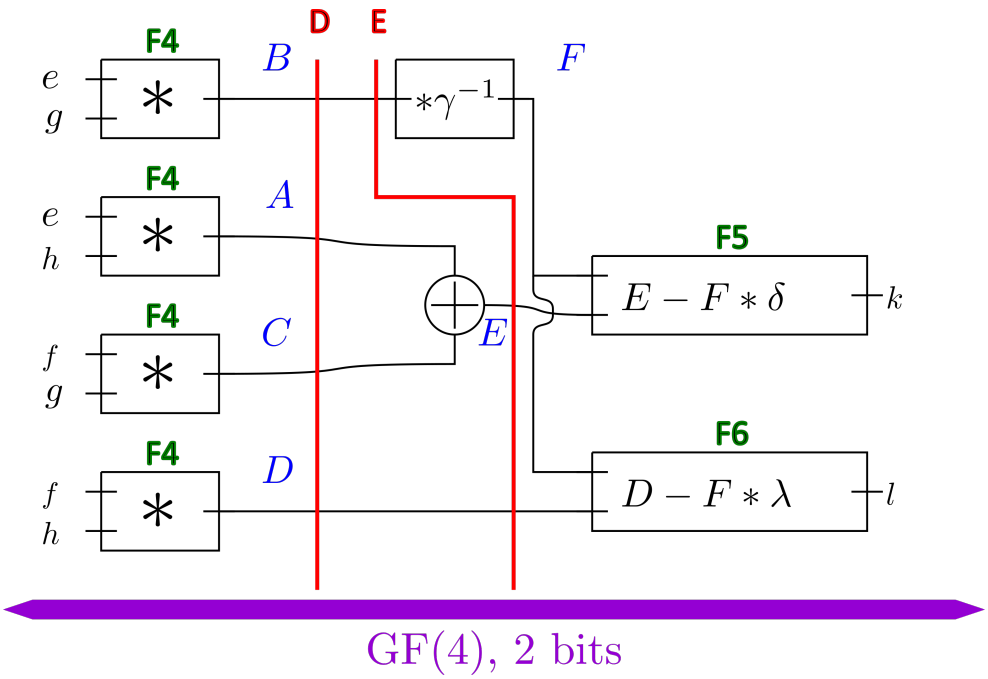
3.4.2.3 Sloučení operací

Z obrázku 3.1 je zřejmé, že uzel A se nijak nevětví, pouze přenáší výstup z jedné operace do operace jiné. Obě operace lze tedy sloučit do jedné (F2) a ušetřit tak CFGLUTy.

Stejné zjednodušení lze provést i v obrázku 3.2 u operace s výstupem na uzel F, která může být součástí operací F5 a F6.



Obrázek 3.1: Schéma znázorňuje výpočet multiplikativní inverze v kompozitním tělese. Obrázek dále obsahuje dva stavové registry rundy (register precharge). Další volitelné registry jsou vyznačeny červenou čárou.



Obrázek 3.2: Schéma znázorňuje výpočet součinu v tělese GF(4). Červenými čárami jsou znázorněny volitelné registry. Násobení je při výpočtu inverze použito celkem třikrát (viz obrázek 3.1)

Dále, na obrázku 3.1 je operace násobení konstantou F1. Vzhledem k volbě ireducibilního polynomu je konstantou jedna (prvek neutrální k násobení) a proto může být celá operace vynechána.

3.4.2.4 Pokročilá verze (DynReconfGenV3)

Pokročilá verze se od dříve popsané liší tím, že provádí překlad mezivýsledků. Z tohoto důvodu jsou i XORy realizovány CFGLUTy.

3.4.2.5 Počet CFGLUTů

Po těchto úpravách je pro násobení (obrázek 3.2) potřeba 12 CFGLUTů. Šířka výstupu všech operací je 2 bity.

- Operace s výstupem na uzel F je vynechána
- Operace F4 se vyskytuje celkem čtyřikrát, tedy celkem 8 CFGLUTů
- Operace F5 a F6 se vyskytuje každá jednou. Celkem tedy 4 CFGLUTy

Pro výpočet inverze je pak potřeba celkem 60 CFGLUTů. Dílčí operace mají výstup široký 4 bity.

- Operace F1 je vynechána
- Operace F2 vyžaduje 4 CFGLUTy
- Operace F3 vyžaduje také 4 CFGLUTy
- Ve výpočtu je třikrát násobení, tedy 3 * 12 = 36 CFGLUTů
- Dekomponovaná identita vyžaduje 16 CFGLUTů

Další LUTy jsou potřeba na izomorfismy mezi tělesy. V základní implementaci je potřeba 128 CFGLUTů, což je přibližně dvojnásobek. V základní verzi ale není nutné realizovat izomorfismy a afinní transformaci.

O něco vyšší je počet CFGLUTů v pokročilém režimu (při použití DynReconfGenV3), protože v takovém případě jsou pomocí CFGLUTů realizovány i operace XOR. V jednom S-Boxu jsou dva čtyřbitové XORy a tři dvoubitové XORy (jeden v každé násobičce). Počet CFGLUTů je v jednom S-Boxu je tedy větší o 14.

3.5 Obálková entita

Obálkovou entitou je myšlena entita, která řídí šifrování a stará se o komunikaci s počítačem, tedy přijímá po sériové lince data k šifrování, šifrovací klíč a blok náhodných dat. Ovládá výstupní signál `Trigger`, který je před začátkem šifrování uveden na krátký okamžik do stavu `1`. Tento signál slouží pro spuštění záznamu spotřeby, jde o synchronizační signál pro osciloskop. Po skončení šifrování je pak šifrový text odeslán zpět do počítače.

Na tuto entitu jsem si vytyčil tyto požadavky:

- Entita by měla být společná pro všechny algoritmy. Toto zjednoduší správu zdrojových souborů.
- Entita se bude starat o snížení frekvence hodinového signálu (dělička), které bude konfigurovatelné.
- Entita by měla komunikovat s počítače textově a data přenášet v šestnáctkovém režimu (znaky 0-9 a A-F). Měla by také kontrolovat, zda přijaté vektory mají správnou délku. To usnadní práci při ručním testování. Tento požadavek je do jisté míry diskutabilní, protože zdvojnásobuje čas potřebný pro přenos dat a zvyšuje nároky na FPGA.
- Entita by měla být schopná odesílat i mezivýsledky při šifrování. U těchto implementací šifrovacích algoritmů není důležitá pouze správnost výsledku (šifrového textu), ale je nutné i ověřovat, zda i mezivýsledky jsou správné. Entitu by tedy mělo být možné spustit ve zvláštním režimu, ve kterém jsou v průběhu šifrování odesílány i mezivýsledky z každé rundy.

- Entita by měla zamaskovat otevřený text a odmaskovat šifrový text, aby do šifrovací entity vstupovaly již ochráněné vektory.

3.5.1 Rozhraní algoritmů

Na základě těchto požadavků jsem navrhl společné rozhraní všech šifrovacích entit (liší se jenom délky vektorů).

```
1  entity Present is
2    generic  (
3      Mode                    :   PresentModeEnum;
4      PresentProtectionMode   :   PresentProtectionModeEnum;
5      EnableRegPrecharge      :   std_logic;
6      WriteEnMode             :   WriteEnModeEnum
7    );
8    port     (
9      Clk           :  in   std_logic;
10     Rst           :  in   std_logic;
11     PlainText     :  in   std_logic_vector(63 downto 0);
12     CipherText    :  out  std_logic_vector(63 downto 0);
13     Key           :  in   std_logic_vector
14      (PresentGetKeyLength(Mode) - 1 downto 0);
15     OutputValid   :  out  std_logic;
16     Start         :  in   std_logic;
17     Ready         :  out  std_logic;
18     Reconfigure   :  in   std_logic;
19     ReloadConfig  :  in   std_logic;
20     RandomData    :  in   std_logic_vector
21      (PresentRandomDataLength(PresentProtectionMode) - 1 downto 0);
22     DebugHalt     :  in   std_logic;
23     InputMask     :  out  std_logic_vector(63 downto 0);
24     OutputMask    :  out  std_logic_vector(63 downto 0)
25 );
26 end entity Present;
```

V části `generic` jsou uvedeny konfigurovatelné parametry implementace, popisované již dříve.

- Port `Clk` je hodinový signál (již vydělený) a port `Rst` je synchronní reset.
- Vektory `PlainText` a `CipherText` přenáší otevřený text a šifrový text, přičemž oba jsou zamaskované maskami `InputMask` a `OutputMask`. Šířka těchto vektorů se u různých algoritmů liší. Port `CipherText` je použit i pro přenos mezivýsledků jednotlivých rund v debugovacím režimu.
- Vektor `Key` přenáší šifrovací klíč. Jeho šířka závisí na algoritmu a také na zvoleném režimu.
- Vektor `RandomData` jsou náhodná data použitá při rekonfiguraci. Jeho délka závisí na algoritmu a použitém způsobu ochrany. Nastavením určitých částí tohoto vektoru na dané konstantní hodnoty lze dosáhnout deaktivace jednotlivých ochran.

- Stavový signál `OutputValid` signalizuje dokončení šifrování, signál `Ready` signalizuje dokončení rekonfigurace.

- Řídící signál `Start` spouští šifrování. signál `Reconfigure` spouští výpočet nové konfigurace CFGLUTů, signál `ReloadConfig` pak spouští nahrávání konfigurace. Proces výpočtu a nahrávání je oddělen, protože výpočet neovlivňuje CFGLUTy, je možné jej spouštět i během šifrování. V aktuální implementaci se ale nikdy nespouští výpočet během šifrování, aby spotřeba tohoto procesu neovlivňovala měření. Jde o teoretickou možnost pro zvýšení propustnosti.

- Řídící signál `DebugHalt` se používá pouze v debugovacím režimu a slouží k pozastavení šifrování do doby, než se dokončí odesílání mezivýsledku.

3.5.2 Použití obálkové entity

Po resetu odešle obálková entita přes sériovou linku informace o aktuálně použitém šifrovacím algoritmu a režimu ochrany. Dále jsou uvedeny délky jednotlivých vektorů. Tento výpis je generován proto, aby nemohlo snadno dojít k záměně dvou bitstreamů s jinými ochranami.

Poté je již entita připravena k použití. Vektory se do entity posílají v šestnáctkovém zápisu (nerozlišuje se velikost písmen). Šifrovací klíč musí být vložen do špičatých závorek, náhodná data do složených závorek a otevřený text do kulatých závorek. Šifrový text je pak vložen do hranatých závorek.

Před prvním šifrováním je nutné provést rekonfiguraci, která se spouští zasláním náhodných dat. Není-li před prvním šifrováním spuštěna rekonfigurace, šifrování nevrátí správný výsledek. Není nutné mezi jednotlivými šifrováními spouštět rekonfiguraci (ale v takovém případě samozřejmě proběhnou všechna šifrování se stejnou konfigurací).

Záznam komunikace při šifrování jednoho bloku může vypadat například takto:

```
1  PRESENT
2  <Key>: 20 hex chars (80 bits)
3  {RandomData}: 304 hex chars (1216 bits)
4  (Plain): 16 hex chars (64 bits)
5  [Cipher]: 16 hex chars (64 bits)
6  Debug mode: '0'
7  Mode: presentprotection_basicpar
8  Reg precharge: '1'
9  Date: 2018-09-01 21:47:25
10
11 <00112233445566778899>
12 {f2A5ef8a9831efacff5b4e..a17d2bfc0bab44e4eb44e}
13 (0123456789ABCDEF)[1A6D783F0C184F4D]
```

Na řádcích 1-10 jsou úvodní informace odeslané obálkovou entitou po resetu. Řádkem 11 je odeslán šifrovací klíč, řádkem 12 náhodná data, která

spustí rekonfiguraci. V tomto výpise je pro přehlednost vektor náhodných dat zkrácen, vynechaná část je zastoupena dvěma tečkami. Na 13. řádku byl odeslán otevřený text. Odpovídající šifrový text přijatý z FPGA je na stejném řádku v hranatých závorkách.

Veškeré platné znaky odeslané z počítače do FPGA jsou odeslány i zpět, aby se zobrazily v terminálu (echo). Pokud je přijat znak, který není v aktuálním stavu očekáván, je zpět odeslán znak otazník, který signalizuje chybný vstup. Pokud tedy uživatel například odešle vektor špatné délky, je na to upozorněn.

Během rekonfigurace není obálková entita připravena přijímat další data a je nutné počkat na dokončení rekonfigurace. Protože nelze nijak snadno detekovat dokončení rekonfigurace, lze spojit zaslání náhodných dat do jednoho příkazu. V takovém případě jsou náhodná data odeslána společně s otevřeným textem ve složených závorkách, přičemž obě části jsou odděleny podtržítkem. Obálková entita nejprve spustí rekonfiguraci s přijatými náhodnými daty a bezprostředně po dokončení spustí samotné šifrování. Tento způsob je uveden v následujícím výpise, kde je opět pro přehlednost část vektoru nahrazena dvěma tečkami.

```
1 <00112233445566778899>
2 {f2A5ef8a9..44e4eb44e_0123456789ABCDEF}[1A6D783F0C184F4D]
```

Zasláním znaku * (hvězdička) lze vyvolat reset FPGA. Odesláním tečky se znovu zopakuje šifrování s minulým otevřeným textem.

3.5.3 Debugovací režim

Debugovací režim je zvláštní režim, ve kterém je možné obálkovou entitu spustit. Od normálního režimu se liší tím, že po každé rundě je šifrování pozastaveno a je odeslán výsledek této rundy. Po odeslání se pokračuje další rundou. Na závěr je opět odeslán šifrový text jako v klasickém režimu. V tomto režimu trvá jedno šifrování značně déle, protože samotná komunikace trvá tisíce hodinových taktů, a proto tento režim není určen pro měření. U všech algoritmů je odesílána hodnota po první části dekomponovaného S-Boxu, tedy hodnota, která je ukládána do stavového registru rundy.

3.5.4 Signály trigger

Tyto signály slouží jako trigger pro osciloskop, tedy jsou přivedeny do osciloskopu, který spustí záznam v okamžiku, kdy je detekována na tomto signálu pozitivní hrana. Je-li šifrování spuštěno příkazem v kulatých závorkách, je trigger aktivován před začátkem šifrování. Je-li šifrování spuštěno příkazem ve složených závorkách, je trigger aktivován před začátkem rekonfigurace, tj. osciloskop zaznamená i spotřebu rekonfigurace.

Mezi tímto impulzem a začátkem šifrování je krátká prodleva (jednotky taktů), aby samotný trigger příliš neovlivnil záznam spotřeby. Stejně tak je i prodleva mezi koncem šifrování a odesláním šifrového textu.

3.5.5 Konfigurovatelné parametry obálkové entity

V souboru `Common/PackageConfig.vhd` lze konfigurovat obálkovou metodu. Dostupné parametry jsou:

- `EncryptionAlgorithm`, kterým se vybírá šifrovací algoritmus. Možné hodnoty jsou `AlgPresent`, `AlgSerpent` a `AlgAES`.
- `DebugMode`, který zapíná či vypíná debugovací režim. Možné hodnoty jsou `0` a `1`.
- `ClockFreq`, kterým se definuje frekvence hodinového signálu vstupujícího do obálkové entity. Tato hodnota je použita pro správné nakonfigurování entity RS232.
- `ClockDivider`, kterým se nastavuje dělička hodinového signálu. Vydělený signálem jsou pak řízeny všechny komponenty (tedy šifrovací algoritmus i obálková entita). Tato hodnota je použita jako exponent dvojky, tedy pro vydělení hodinového signálu dvakrát musí být tento parametr nastaven na 1 ($2^1 = 2$). Pro vydělení signálu osmkrát musí být tento parametr nastaven na 3 ($2^3 = 8$) atd.
- `UartBaudRate`, kterým se nastavuje baudová rychlost komunikace

Tyto parametry jsou společné pro všechny algoritmy, jednotlivé implementace pak ještě mají další parametry (popsány výše), který se nastavují v souborech `PackagePresent.vhd`, `PackageSerpent.vhd`, a `PackageAES.vhd`.

3.5.6 Rozhraní obálkové entity

```
1  entity TopLevel is
2    port     (
3      Clk          :    in   std_logic;
4      ExtRst       :    in   std_logic;
5      Rx           :    in   std_logic;
6      Tx           :    out  std_logic;
7      Trigger1     :    out  std_logic;
8      Trigger2     :    out  std_logic;
9      Ready        :    out  std_logic
10 );
11 end entity TopLevel;
```

Význam jednotlivých portů je zřejmý již podle názvu. V aktuální verzi jsou oba signály `Trigger1` a `Trigger2` ekvivalentní. Stavový signál `Ready` indikuje,

zda je entita připravená přijímat data. Tento signál je momentálně využíván pouze testbenchem.

3.5.7 Testování

Ve složce `Common/Tests` je testbench, který umožňuje testovat obálkovou entitu (a potažmo i jednotlivé šifrovací algoritmy). Testbench dělá to, že do obálkové entity odešle (přes port `RX`) vstup uložený v souboru a zároveň zaznamenává výstup obálkové entity (port `TX`), který je zapisován opět do souboru.

Tento testbench je zásadní v tom, že umožňuje otestovat celý návrh bez nutnosti generovat bitstream a zkoušet jej na skutečném FPGA, což ušetří mnoho času. Pokud je režim zápisu do registru nastaven na `FreeRunning`, trvá tato simulace déle (při `FreeRunning` jsou data do stavového registru zapisována nepřetržitě a simulátor je tedy musí přepočítávat, i když šifrování ve skutečnosti neprobíhá).

3.5.8 Komponenta RS232

Tato komponenta realizuje komunikaci po sériové lince. Byla vytvořena Dr.-Ing. Martinem Novotným. V rámci této práce jsem jí částečně upravil, aby dosahovala vyšších přenosových rychlostí a umožnila tak rychlejší měření. Protože celé FPGA běží při měření na nižší frekvenci (okolo 10 MHz), při vyšších baudových rychlostech není možné přesně odměřit čas jednotlivých bitů a komunikace nefunguje nebo není spolehlivá. Upravená verze mění počet taktů pro každý baud tak, aby se během přenosu zaokrouhlovací chyba nenasčítávala, ale naopak eliminovala. Alternativním řešením by bylo řídit tuto komponentu jiným hodinovým signálem než ostatní komponenty a korektně vyřešit přechod mezi hodinovými doménami.

3.5.9 Shrnutí

Celkem byly implementovány šifrovací algoritmy PRESENT, SERPENT a AES s nastavitelnou délkou klíče, kterou je nutné zvolit při překladu. Pro jednotlivé implementace algoritmů existuje více režimů.

Pro PRESENT i SERPENT je k dispozici i paralelní verze. Paralelní verze potřebuje méně hodinových taktů pro rekonfiguraci CFGLUTů a navíc mohou být jednotlivé S-Boxy rekonfigurovány nezávisle podle vlastních náhodných dat.

Implementace AESu může běžet celkem v 6 režimech

- `Basic` - základní režim, ve které je S-Box rozdělen na 2 dílčí bijekce stejným způsobem jako v implementaci PRESENTu nebo SERPENTu.
- `BasicPar` - paralelní režim předchozího režimu.

- `CompositeField` - režim, při kterém je S-Box vypočítáván převedením bajtů do kompozitního tělesa a následném výpočtu inverze. Mezivýsledky při výpočtu S-Boxu jsou zabezpečeny pouze maskováním.
- `CompositeFieldPar` - paralelní režim předchozího režimu.
- `AesProtection_CompositeField_ADVANCED` - režim, při kterém se také používá kompozitní těleso pro výpočet S-Boxu, ale hodnoty při výpočtu jsou kromě maskování zabezpečeny i překladem hodnot.
- `AesProtection_CompositeField_ADVANCEDRegs` - stejné jako předchozí, ale do výpočtu S-Boxu jsou vloženy registry, které zamezují šíření hazardů, ale zvyšují počet hodinových taktů potřebných na jednu rundu.

Zdrojové kódy rekonfigurace pro režimy `AesProtection_CompositeField_ADVANCED` a `AesProtection_CompositeField_ADVANCEDRegs` byly vygenerovány nástrojem DynReconfGenV3, pro ostatní verze byl použit nástroj DynReconfGen. Pro ně by bylo možné použít také DynReconfGenV3, ale nemělo by to žádný přínos oproti aktuální verzi.

Konkrétní režim je nutné zvolit při překladu, nelze jej měnit za běhu.

3.6 Postupy při práci

V této podkapitole jsou popsány postupy a nástroje použité při práci. Je zde popsán celý postup od psaní VHDL kódu přes generování bitstreamu, měření spotřeby při šifrování až po analýzu naměřených výsledků.

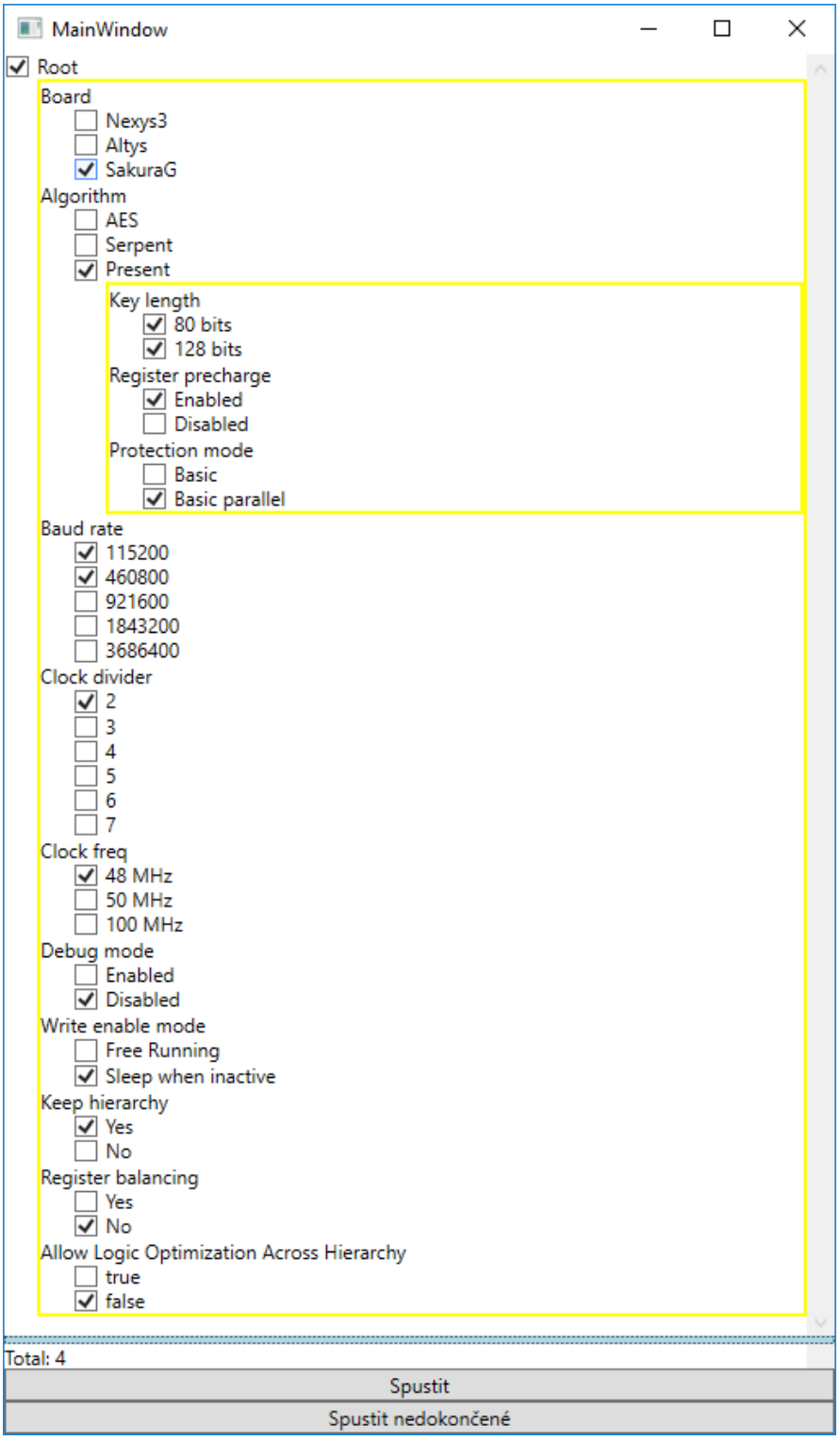
3.6.1 VHDL zdrojové kódy

V tomto projektu se hojně používají CFGLUTy, což je entita, která je rozpoznána při syntéze kódu, ale pro používaný simulátor ISim není známá. Pro efektivní testování je nutné mít možnost simulovat obvod, a proto jsem vytvořil entitu `FakeCFGLUT5`, která má stejné rozhraní a chování jako `CFGLUT5`, ale je napsána ve VHDL a tudíž spustitelná v simulátoru. Při syntéze však není tato entita rozpoznána jako CFGLUT.

Pro simulaci je tedy nutné používat jinou entitu než pro generování bitstreamu. Z toho důvodu není ve zdrojových kódech přímo uveden název entity, ale pouze zástupný symbol `__CFGLUT__`. Pro simulaci je pak nahrazen entitou `FakeCFGLUT5`, pro překlad do FPGA entitou `CFGLUT5`. Nahrazení je prováděno nástrojem GNU M4 (https://www.gnu.org/software/m4/)

3.6.2 Generování bitstreamu

Protože byly vytvořeny implementace několika algoritmů, u nichž lze navíc nastavovat další parametry (délku klíče, způsob ochrany atd.), vytvořil jsem



Obrázek 3.3: Hlavní okno nástroje Autobuild

jednoduchý program Autobuild, který se postará o automatické vygenerování bitstreamů požadovaných variant.

Z uživatelského pohledu jde o program s jedním oknem (obrázek 3.3), ve kterém lze označit požadované varianty. Ze zaškrtnutých parametrů program vytvoří všechny možné kombinace. Například, pokud uživatel zaškrtne obě možné délky klíče (80 a 128 bitů) a dvě baudové rychlosti (115200 B a 460800 B), budou vygenerovány celkem 4 bitstreamy, pro každou možnou variantu z vybraných možností jeden. Vzniklé bitstreamy jsou pojmenovány tak, aby v jejich názvu byly uvedeny všechny použité parametry a nemohlo tak snadno dojít k záměně.

Z vnitřního pohledu program funguje tak, že má v sobě definován TCL skript pro každou cílovou desku. Do něj program dosadí seznam všech zdrojových souborů a zvolené parametry pro překlad. Dále vygeneruje skript pro GNU M4 procesor, který zajistí, že při zpracovávání souborů tímto procesorem budou na příslušná místa do kódu dosazeny zbývající parametry. TCL skript je poté spuštěn programem *xtclsh.exe*. Všechny soubory vzniklého projektu včetně reportů jsou zkomprimovány do 7-Zip archivu pro případné pozdější zkoumání.

Hlavní výhodou tohoto nástroje je, že proces je automatizovaný, uživatel nemusí ručně nastavovat parametry ve zdrojových kódech, spouštět překlad jednotlivých variant, čekat na dokončení a správně pojmenovat vzniklý bitstream. To eliminuje riziko chyby, např. opomenutí nastavit nějaký parametr nebo záměnu bitstreamů.

3.6.3 Měření a analýza naměřených dat

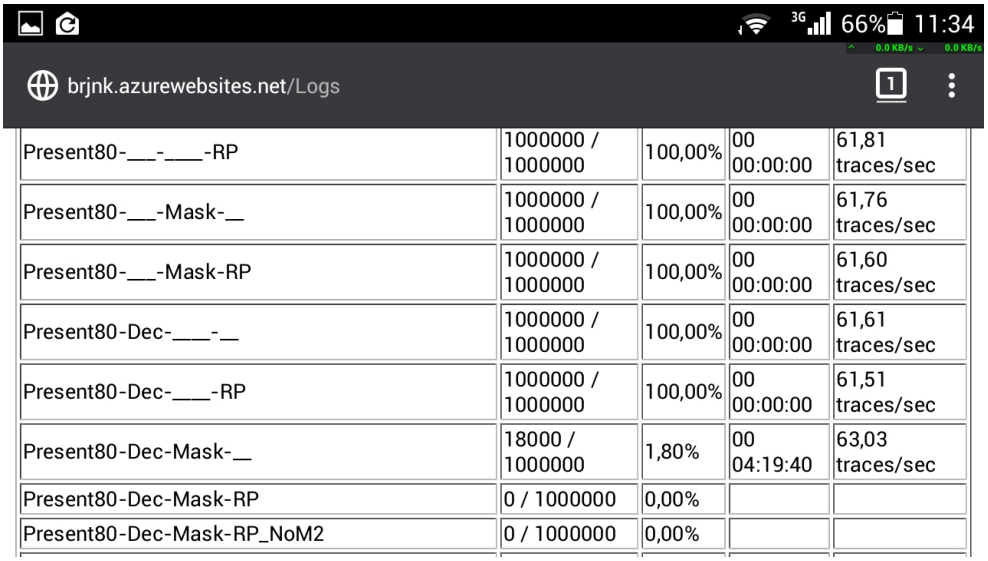
Protože potřebuji analyzovat spotřebu při šifrování několika různých implementací, přičemž chci navíc zkoumat výsledky pro různé kombinace použitých ochran, potýkám se zde s podobným problémem jak v předchozím odstavci, tedy s velkým množstvím různých variant. Proto jsem vytvořil program, který provádí celé měření včetně následné analýzy zcela samočinně, bez nutnosti uživatelského zásahu (vyjma počátečního nastavení).

Vstupem programu je jedna nebo více úloh, které mají být provedeny. Každá úloha je definována jedním XML souborem, který obsahuje všechny nezbytné parametry. Na následujícím výpise je uveden příklad jedné úlohy.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <MeasurementJob TracesCount="1000000"
3                  Name="Present80 Decomposition Masking RegPrecharge"
4                  IsEnabled="true"
5                  UniqueName="Present80-Dec-Mask-RP">
6    <Initializer
7        BitstreamFile="SakuraG_Present_80b_RP_BasicPar_921k6_48MHz_Div4.bit"
8        Type="XilinxImpact"/>
9    <Port Type="SerialPort"
10         TwoStopBits="false"
11         BaudRate="921600"
```

```
12          PortName="COM22"/>
13    <Transmitter Type="PresentDataTransmitter">
14        <KeyLength>80</KeyLength>
15        <Key>0123456789ABCDEF0123</Key>
16        <RegisterPrecharge>true</RegisterPrecharge>
17        <Mask1>true</Mask1>
18        <Mask2>true</Mask2>
19        <BijectionSwapsCount>8</BijectionSwapsCount>
20        <DependentSBoxesCount>1</DependentSBoxesCount>
21    </Transmitter>
22    <Receiver Type="PresentDataReceiver"/>
23    <Oscilloscope Type="Picoscope6000"
24                  VoltageRange="Range50mV"
25                  Offset="9300"
26                  NumberOfSamples="4300"/>
27    <Analyze Type="Present"
28             TrimEnd="4050"
29             TrimBegin="0"
30             ClockFreq="12000000"
31             Key="0123456789ABCDEF0123"
32             KeyLength="80"/>
33    <Uploader Type="OwnCloud"
34              Exe7Zip="C:\Program Files\7-Zip\7z.exe"
35              Url="https://owncloud.cesnet.cz/"
36              TargetDir="/MeasurementTool.Upload"
37              PasswdFile="c:\Measurement\p.txt"
38              UserName="brejnjan@cvut.cz"/>
39 </MeasurementJob>
```

- Kořenový element `<MeasurementJob>` obsahuje základní atributy měření, tedy počet měřených průběhů a název měření.
- Element `<Initializer>` na řádku 6 definuje způsob inicializace FPGA, tedy nahrání bitstreamu do FPGA. V tomto případě je uveden soubor s bitstreamem a pro nahrání bude použit nástroj Xilinx iMPACT.
- Element `<Port>` na řádku 9 nastavuje parametry sériového portu.
- Element `<Transmitter>` definuje formát odesílaných zpráv. Protože součástí dat posílaných do FPGA jsou i náhodná data použitá pro rekonfiguraci, lze nastavením určité části tohoto bloku na nějakou konstantu vypnout určitou ochranu. A právě elementy na řádcích 16 - 20 určují, které ochrany budou zapnuté. Element `<Receiver>` na řádku 22 určuje způsob dekódování příchozích zpráv (tedy šifrového textu).
- Element `<Oscilloscope>` na řádku 23 nastavuje osciloskop, např. počet zaznamenaných samplů nebo offset po signálu trigger.
- Element `<Analyze>` na řádku 27 obsahuje parametry nutné pro analýzu získaných dat.



Present80-___-____-RP	1000000 / 1000000	100,00%	00 00:00:00	61,81 traces/sec
Present80-___-Mask-__	1000000 / 1000000	100,00%	00 00:00:00	61,76 traces/sec
Present80-___-Mask-RP	1000000 / 1000000	100,00%	00 00:00:00	61,60 traces/sec
Present80-Dec-____-__	1000000 / 1000000	100,00%	00 00:00:00	61,61 traces/sec
Present80-Dec-____-RP	1000000 / 1000000	100,00%	00 00:00:00	61,51 traces/sec
Present80-Dec-Mask-__	18000 / 1000000	1,80%	00 04:19:40	63,03 traces/sec
Present80-Dec-Mask-RP	0 / 1000000	0,00%		
Present80-Dec-Mask-RP_NoM2	0 / 1000000	0,00%		

Obrázek 3.4: Aktuální stav měření lze sledovat vzdáleně přes internet

- Element `<Uploader>` na řádku 33 se pak postará o odeslání naměřených dat společně s výsledky analýzy do datového úložiště.

3.6.3.1 Spolehlivost programu

Hlavní předností je bezpochyby již zmíněná samostatnost programu, díky které je schopný automaticky bez zásahu člověka pracovat v kuse několik dní a provést všechna naplánovaná měření.

Aby byla spolehlivost programu maximální, je k němu vytvořen i watchdog, což je program, který běží v samostatném procesu a jeho úkolem je hlídat, zda měřící program pracuje. Měřící program s watchdogem spolupracuje a v pravidelném intervalu mu posílá zprávu, že je aktivní, podobně jako v případě hardwarového watchdogu. Pokud zamrzne, nebo předčasně skončí, watchdog se postará o ukončení předchozího procesu a nové spuštění měřícího programu.

Měřící program tedy musí perzistentně uchovávat informaci o tom, kolik průběhů již bylo naměřeno, aby při opětovném spuštění pokračoval tam, kde skončil. Pro ukládání těchto informací se používá databáze SQLite, která splňuje ACID vlastnosti, takže i po nečekaném přerušení (výpadek napájení) je možné obnovit datové soubory do konzistentního stavu a pokračovat v měření.

Měřící program navíc pravidelně odesílá přes internet informaci o aktuálním stavu, takže je možné vzdáleně sledovat průběh měření (obrázek 3.4).

Kapitola 4

Testování a měření

Tato podkapitola se zabývá testováním vytvořených implementací. Vzhledem k tématu této práce lze testování rozdělit do dvou rovin, na testování správnosti výsledku a na testování odolnosti proti útokům. Oba tyto aspekty jsou testovány při procesu měření. Ten spočívá v opakovaném šifrování dat v FPGA, při kterém se zaznamenává otevřený text odeslaný do FPGA, šifrový text přijatý z FPGA a průběh spotřeby během šifrování. Popisu procesu měření se věnuje první část této kapitoly.

Během prvotních měření se ukázalo, že odolnost proti útokům není v některých případech tak dobrá, jak se očekávalo. U některých problémů se podařilo najít příčinu a odstranit ji, což je popisováno ve zbylé části této kapitoly. Výsledky finálních měření pak nabízí následující kapitola.

Poznámka ke grafům: V této a další kapitole jsou uvedeny grafy, které znázorňují buď průběh spotřeby při jednom šifrování, nebo průběh t-hodnoty provedeného t-testu. V obou případech je vodorovná osa časová a jednotkou je počet vzorků (samplů) od začátku záznamu. Perioda vzorkování byla u všech provedených měření 1,6 ns. V případě grafu spotřeby je na svislé ose hodnota zaznamenaná osciloskopem. Jde o 16bitové znaménkové číslo. Použitý rozsah osciloskopu není uváděn, ale lze jej dohledat ve výsledcích měření na přiloženém DVD. V případě grafu t-hodnoty je na svislé ose samotná t-hodnota. Je-li uvedeno několik grafů nad sebou, sdílejí všechny stejnou vodorovnou osu.

4.1 Kontrola správnosti výsledku

Kontrola správnosti výsledku je prováděna při analýze naměřených dat. Součástí měřícího programu jsou softwarové implementace všech použitých šifrovacích algoritmů a jejich výstup je porovnáván s šifrovým textem přijatým z FPGA.

Tento typ kontroly je zde zahrnut proto, že nebyly vytvořeny tesbenche, které by testovaly správnost výsledku pro nějakou rozsáhlejší množinu vstupních vektorů. Testbenche jsou určené primárně pro ruční testování a obvykle

testují jeden až dva vstupy. Při měření jsou ale šifrována náhodná data a počet šifrování je v řádu statisíců, což už je dost velké číslo na to, aby mohla být vytvořená implementace považována za bezchybnou. Tato kontrola nikdy neodhalila žádný problém - na všechny chyby se přišlo již při testování v simulaci.

4.2 Časová lokalizace jednotlivých rund

Vstupem vnitřní rekonfigurace FPGA jsou náhodné bity. Ty nejsou vygenerovány uvnitř FPGA nějakým generátorem pseudonáhodných dat, ale jsou do něj zaslány z počítače. Program je tak může uložit pro pozdější využití. Z nich je totiž možné zpětně rekonstruovat vnitřní konfiguraci a podle ní potom i skutečné mezivýsledky při šifrování. V reálném použití by toto samozřejmě degradovalo použité ochrany, ale pro výzkumné účely to může být užitečné. Při analýze naměřených dat je možné pro každé šifrování modelovat velikost spotřeby při zápisu do stavového registru pro každou jednu rundu. Jako model spotřeby používám Hammingovu vzdálenost mezi po sobě jdoucími hodnotami. Poté je možné pro každý okamžik (sample) spočítat korelační koeficient mezi modelovanou spotřebou a skutečně naměřenou spotřebou. Tímto způsobem je možné určit, v kterém okamžiku byly jednotlivé rundy vyhodnocovány.

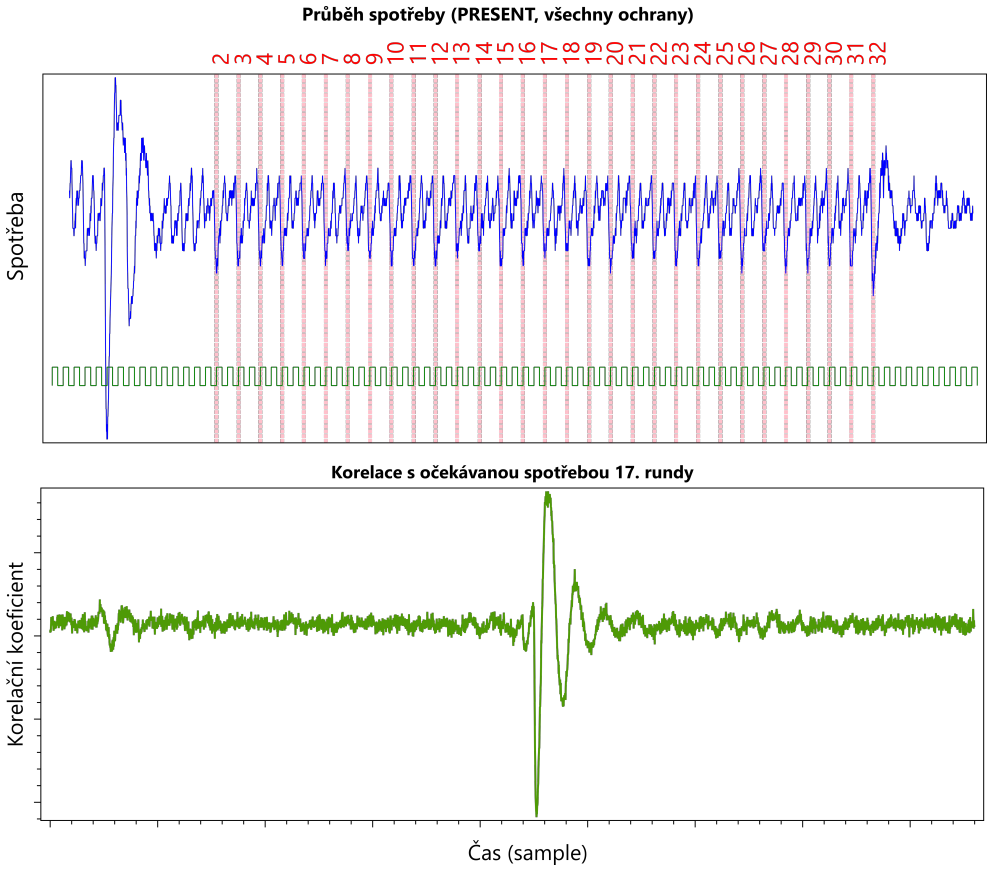
Lokalizaci rund znázorňují grafy na obrázku 4.1. Na horním grafu je průběh spotřeby jednoho šifrování. V něm jsou také vyznačeny pozice jednotlivých rund červenými svislými čárami. Zeleně je v něm naznačen hodinový signál. Jeho frekvence je známá (na základě známého hodinového zdroje a děličky), jeho posun je určen podle vzestupné hrany triggeru, který je nastaven současně s hranou hodinového signálu. Jde ale pouze o orientační údaj, protože signál trigger může být zpožděn. Na jednu rundu zde připadají dva hodinové takty (kvůli register precharge). Na dolním grafu je pak vyobrazen průběh korelačního koeficientu s očekávanou spotřebou sedmnácté rundy.

Správně by tedy červené svislé čáry měly být v grafu rozmístěny pravidelně a popořadě. Pokud tomu tak není, může to ukazovat na nějakou chybu:

- Měření je špatně provedené, například nebyl správně zapojen kabel do osciloskopu nebo je měřen časový úsek, ve kterém šifrování vůbec neprobíhá, ale probíhá v něm třeba přenos dat nebo rekonfigurace.

- Rekonfigurace neprobíhá tak, jak je očekáváno. V implementaci algoritmu může být nějaká chyba, která se neprojeví chybným šifrovým textem, ale chybnými mezivýsledky.

Časová lokalizace rund je tedy další kontrolní mechanismus, který může upozornit na některé chyby. Pokud by například byl měřen špatný časový úsek, ve kterém neprobíhá šifrování, test na odolnost proti útokům by dopadl velmi dobře, ačkoliv skutečnost by mohla být zcela jiná.



Obrázek 4.1: Časová lokalizace rund

Časová lokalizace rund je do jisté míry také pouze orientační údaj a nelze z něj s naprostou jistotou určit, že v daný okamžik skutečně proběhla daná runda. Je nutné vzít v úvahu, že v rundě jsou dva stavové registry za sebou, takže v obou z nich dochází ke stejné změně (se stejnou Hammingovou vzdáleností), ale v odlišný okamžik.

4.3 Testování odolnosti proti útoku

Pro ověřování odolnosti proti útokům byl použit postup uvedený v [16]. Ten využívá statistických metod a pomocí t-testu se snaží dokázat nebo vyvrátit, zda informace ze zařízení uniká. Pro test je nutné naměřit dvě sady průběhů (traců), přičemž v první sadě je pro každé šifrování použit náhodný otevřený text, zatímco v druhé sadě je vždy šifrován stále jeden stejný otevřený text, v mém případě jsem používal samé nuly.

Pro test je nulovou hypotézou výrok, že obě sady průběhů mají v každém časovém okamžiku stejný průměr a rozptyl. Alternativní hypotézou je, že prů-

měr závisí na mezivýsledcích při šifrování, tedy že tajná informace má vliv na mezivýsledky. Platnost alternativní hypotézy ještě ale neprokazuje, že lze zjistit část nebo celý šifrovací klíč, pouze prokazuje, že nějaká informace ze zařízení uniká [16].

Počet měřených by měl odpovídat počtů průběhů, které může útočník teoreticky získat [16].

V této práci byl pro provedení t-testu použit program, který vytvořil Ing. Vojtěch Miškovský a Bc. Petr Socha. Jeho vstupem jsou dvě sady průběhů ve dvou souborech. Výstupem je pak průběh t-hodnoty a stupňů volnosti. Z těchto údajů lze dopočítat p-hodnotu.

Aby bylo možné implementaci považovat za odolnou proti útoku na daném počtu průběhů, absolutní hodnota t-hodnoty nesmí převyšovat 4,5. [16]. Průběh t-hodnoty je možné vynést do grafu a určit tak, v který časový úsek informace uniká.

Příklad výsledků je vidět na obrázku 4.2, na kterém je graf t-hodnoty a p-hodnoty ochráněné verze algoritmu PRESENT. Z grafů je vidět, že během celého šifrování t-hodnota nepřesahuje 4,5. V oblasti poslední rundy je pak vidět špička, která značí nějaký únik. Na konci, kolem vzorku číslo 4100 je pak ještě jedna větší špička, která je způsobena zapsáním neochráněného šifrového textu do posuvného registru, ze kterého bude postupně odeslána do počítače.

Pro srovnání je uveden i obrázek 4.3, kde je naopak průběh t-hodnoty a p-hodnoty neochráněné verze algoritmu PRESENT.

(Pozn.: Na obrázku 4.3 lze pozorovat, že nejsou lokalizovány všechny rundy. Jak bylo uvedeno výše, lokalizace není zcela spolehlivá, zejména u neochráněných verzí se častěji vyskytují chyby.)

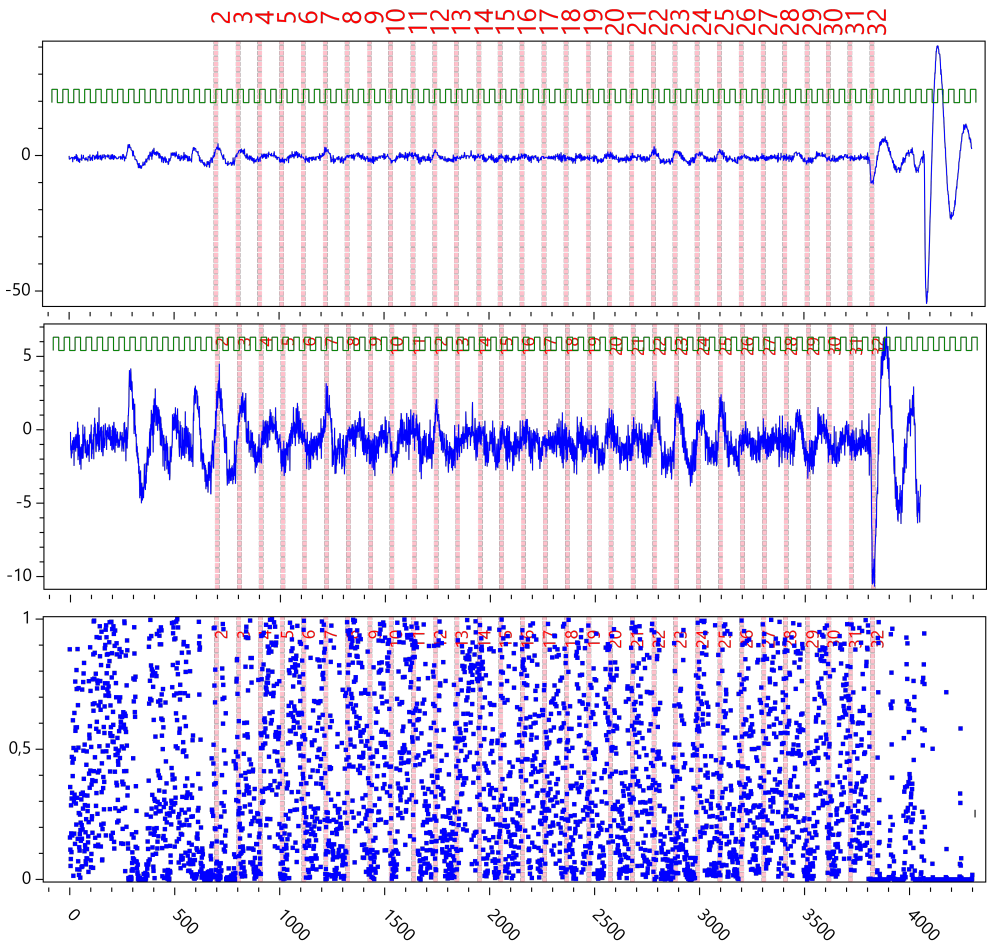
4.4 Použitý hardware

Měření probíhalo na desce SAKURA-G, která je osazena FPGA čipem Xilinx Spartan-6, XC6SLX75-2CSG484C. Velikost spotřeby byla měřena na konektoru J3, který je připojen k výstupu zesilovače. Jumper JP2 byl propojen, tedy měřící rezistor R12 byl přemostěn [17]. Propojením tohoto jumperu se zmenší rozsah měřeného signálu, na osciloskopu lze pak u některých verzí zvolit menší rozsah a naměřený průběh spotřeby je vizuálně plynulejší než bez použití jumperu. Při prvotních měřeních jsem měl jumper nepropojen, pro finální měření jsem jej propojil. Jde ale pouze o částečně subjektivní rozhodnutí, nedělal jsem žádné experimenty, které by porovnávaly tyto možnosti. Srovnání obou variant nabízí grafy na obrázku 4.4.

4.4.1 UCF soubor

Při překladu byl použit UCF soubor s tímto obsahem:

```
1 NET "Clk" LOC = J1;
2 NET "DividerMSB" TNM_NET = DividerMSB;
```

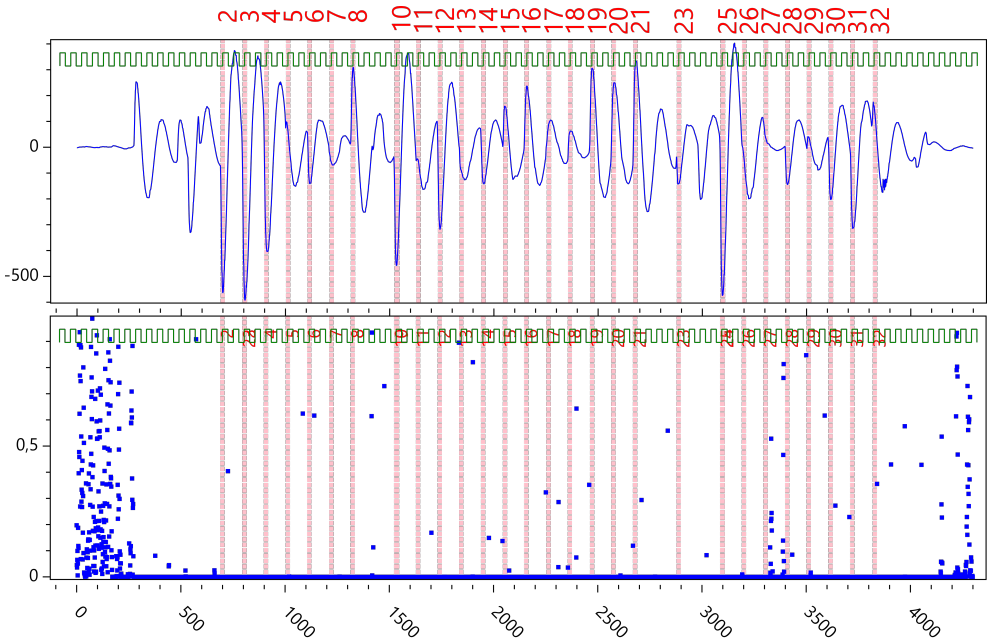


Obrázek 4.2: Příklad výsledků t-testu. Na prvním i druhém grafu je průběh t-hodnoty, oba grafy se liší pouze měřítkem svislé osy. Na třetím grafu jsou pak vyznačeny p-hodnoty. Jde o ochráněnou verzi.

```
3  TIMESPEC TS_DividerMSB = PERIOD "DividerMSB" 80 ns HIGH 50%;
4
5  NET "ExtRst" LOC = D3;
6  NET "Rx" LOC = A8;
7  NET "Tx" LOC = A9;
8  NET "Trigger1" LOC = A4;
9  NET "Trigger2" LOC = A5;
```

Jako hodinový signál je připojen výstup 48MHz oscilátoru, který je součástí desky. Resetovací signál je připojen k tlačítku SW3. Ostatní signály jsou vyvedeny na header CN3. Za zmínku ještě stojí řádek 3, kterým je nastavena maximální požadovaná frekvence vyděleného hodinového signálu.

Podle specifikace desky SAKURA-G [17] pak lze určit mapování signálů na konkrétní piny headeru CN3, které je uvedeno v tabulce 4.1.



Obrázek 4.3: Na prvním grafu je průběh t-hodnoty neochráněné verze a na druhém grafu odpovídající p-hodnoty

Signál	LOC	Číslo pinu na CN3	Směr
Rx	A8	7	Vstup do FPGA
Tx	A9	8	Výstup z FPGA
Trigger1	A4	1	Výstup z FPGA
Trigger2	A5	2	Výstup z FPGA

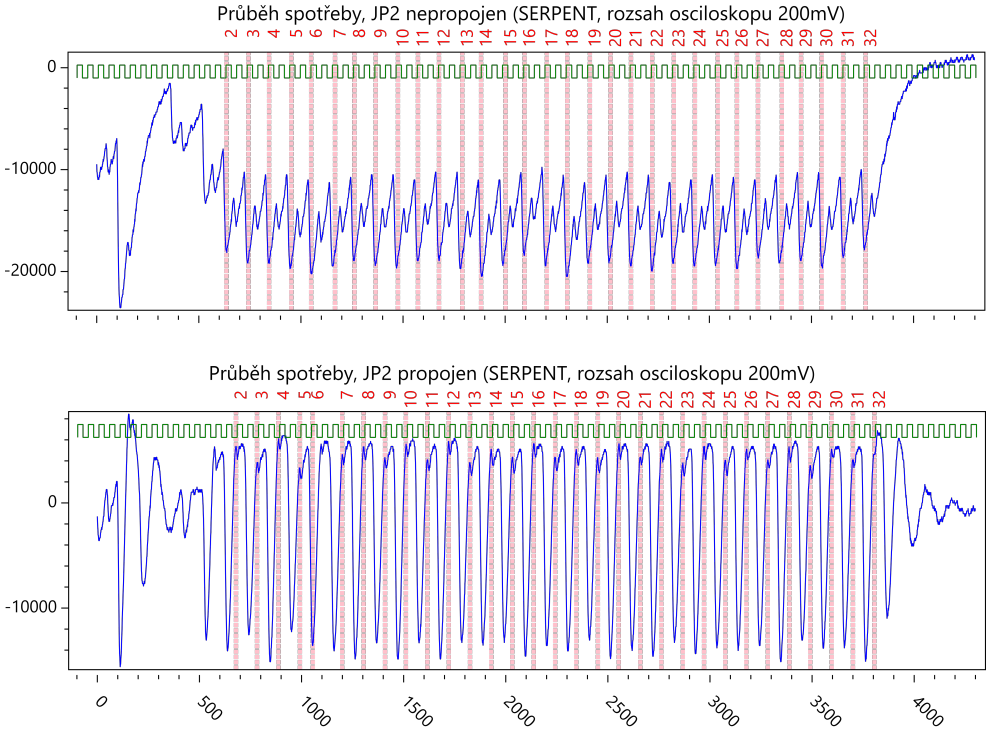
Tabulka 4.1: Mapování signálů na header CN3

Pro propojení zemí lze použít mimo jiné piny 9 nebo 10 na headeru CN3.

4.5 AES, základní verze

Tato část hovoří o základní implementaci AESu, která nepoužívá pro výpočet S-Boxu kompozitní těleso, ale pro realizaci osmibitového zobrazení používá kaskádu CFGLUTů.

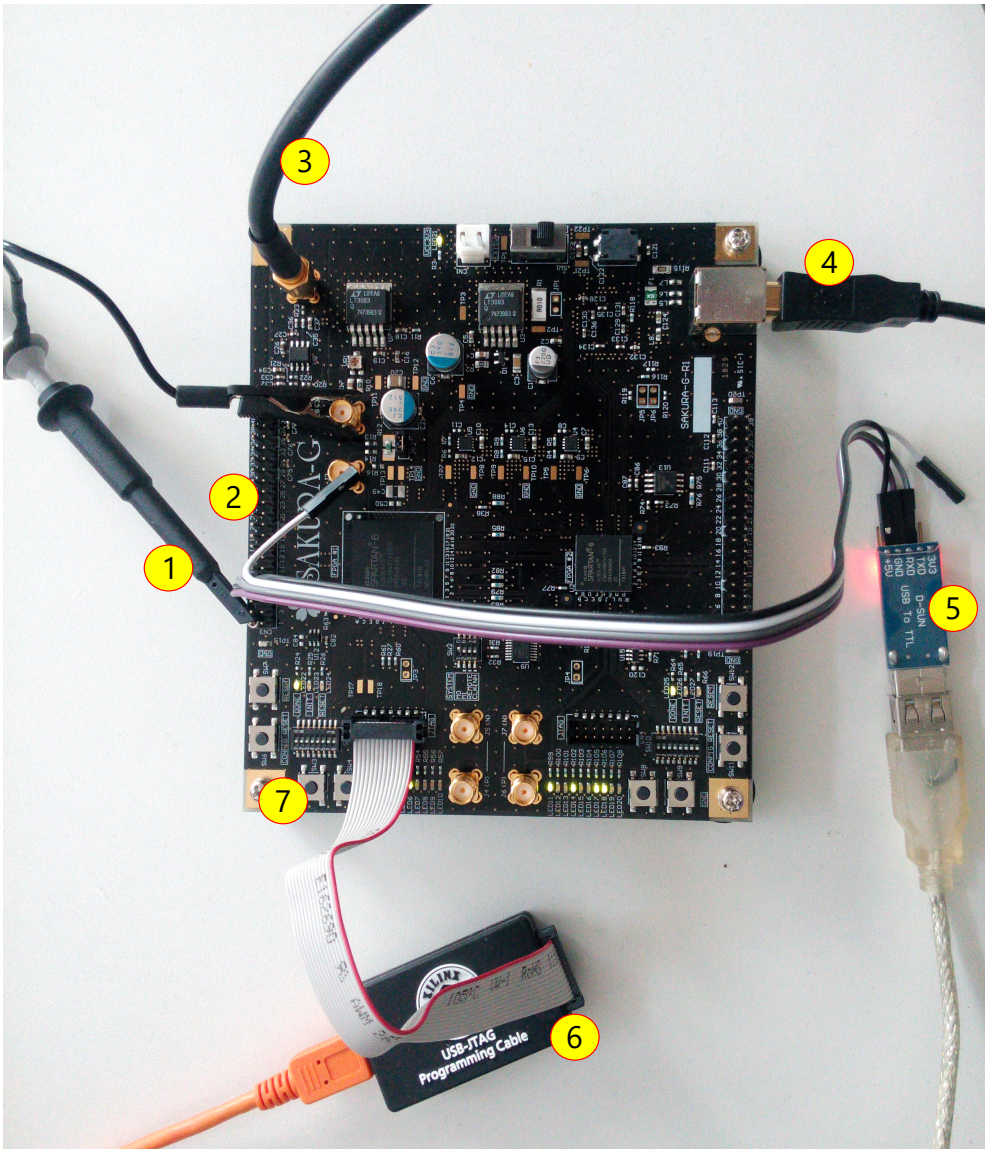
Při prvotním měření se ukázalo, že téměř celé šifrování je dobře ochráněno, vyjma úniku v první rundě. Na obrázku 4.6 je uveden průběh t-hodnot a pod ním odpovídající výřez ze simulace. Z obrázku je patrné, že k úniku informace dojde přesně před prvním zápisem do stavového registru, tedy ve chvíli, kdy je k neochráněnému otevřenému textu přičtena úvodní maska a následně počáteční rundovní klíč.



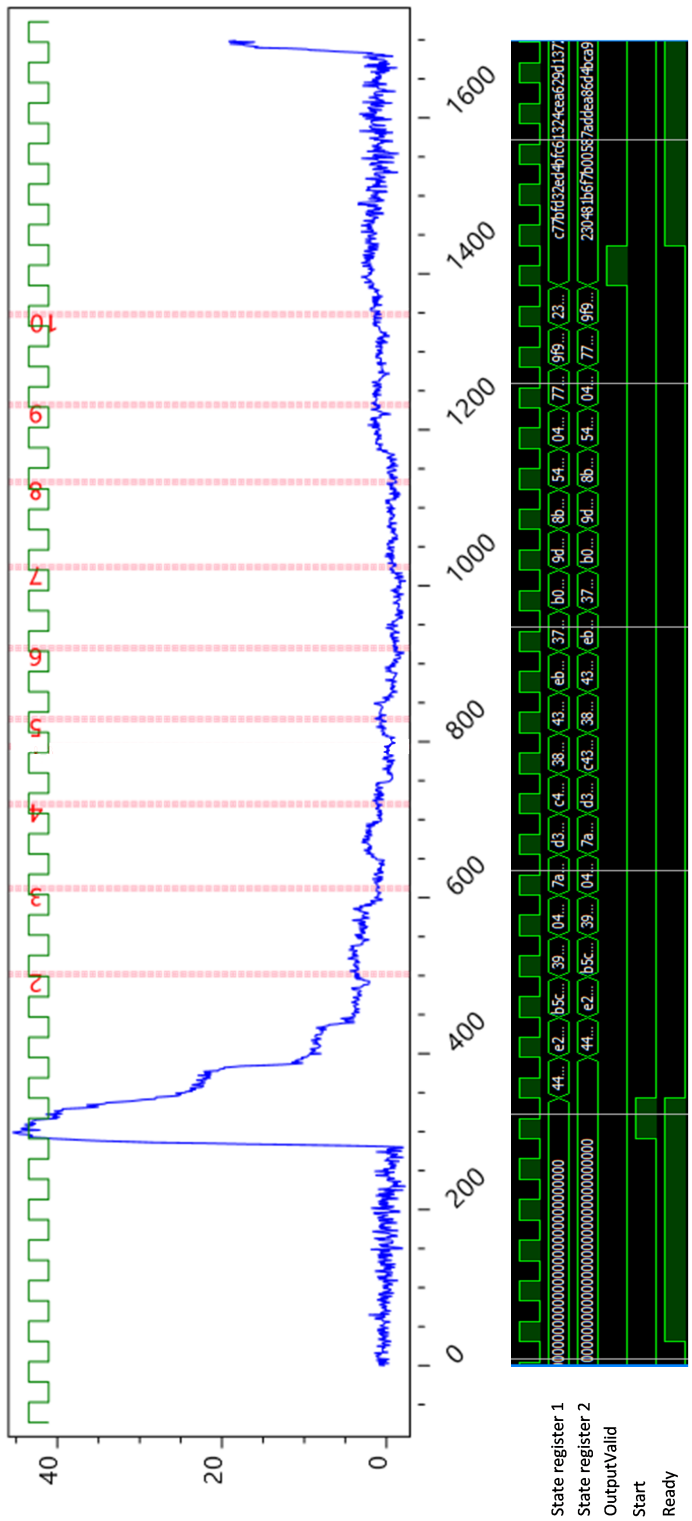
Obrázek 4.4: Porovnání naměřeného průběhu spotřeby s nepropojeným a propojeným jumperem JP2

Číslo	Popis
1	Sonda osciloskopu (kanál A), která snímá synchronizační signál spouštějící záznam (trigger)
2	Header CN3
3	Sonda osciloskopu (kanál C), snímá velikost spotřeby
4	USB kabel připojený k počítači. Slouží pouze pro napájení desky (ale bylo by možné jej použít i pro komunikaci místo USB-UART převodníku)
5	USB-UART převodník připojený přes USB k počítači a k desce přes UART vyvedený na header CN3
6	Programátor, slouží k nahrání bitstreamu do FPGA na začátku měření
7	Tlačítko SW3, reset obvodu

Tabulka 4.2: Vysvětlivky k obrázku 4.5



Obrázek 4.5: Fotografie zapojení desky SAKURA-G. Vysvětlivky jsou v tabulce 4.2



Obrázek 4.6: Na obrázku je průběh t-hodnoty při prvotním měření AESu a pod ním odpovídající výřez ze simulace zachycující hodnoty stavových registrů

4.5.1 Vylepšené maskování

Na základě tohoto pozorování jsem udělal úpravy v maskování. Doposud se přičítání úvodní masky a odečítání závěrečné masky řešilo uvnitř entity, která realizuje šifrování. V nové, upravené verzi se o přičítání masek stará obálková entita. Entita realizující šifrování má nově navíc dva výstupní porty přenášející hodnotu počáteční a koncové masky. Tyto hodnoty použije obálková entita na začátku pro ochránění plain textu a na konci pro získání správného šifrového textu.

Tím se nijak nezmění výsledek, ale práce s neochráněnými hodnotami by se měla posunout mimo úsek šifrování a z grafu t-hodnoty by tak mělo být patrné, zda je samotné šifrování dostatečně ochráněné. Aby se navíc zamezilo tomu, že v rámci různých optimalizací při generování bitstreamu dojde k přeuspořádání tohoto uspořádání, je samotné maskování implementováno ve vlastní entitě, přičemž je zapnuta volba Keep hierarchy a zakázány volby Allow logic optimalization across hierarchy a Register balancing.

Touto změnou se podařilo únik vyřešit, práce s neochráněnými daty po šifrování je sice v grafu patrná, ale je zřetelně oddělená od šifrování.

4.6 SERPENT

Horší výsledky ukázalo prvotní měření implementace SERPENTu, u kterého je vidět únik během celého šifrování, vyjma začátku. Graf společně s výřezem ze simulace je na obrázku 4.7.

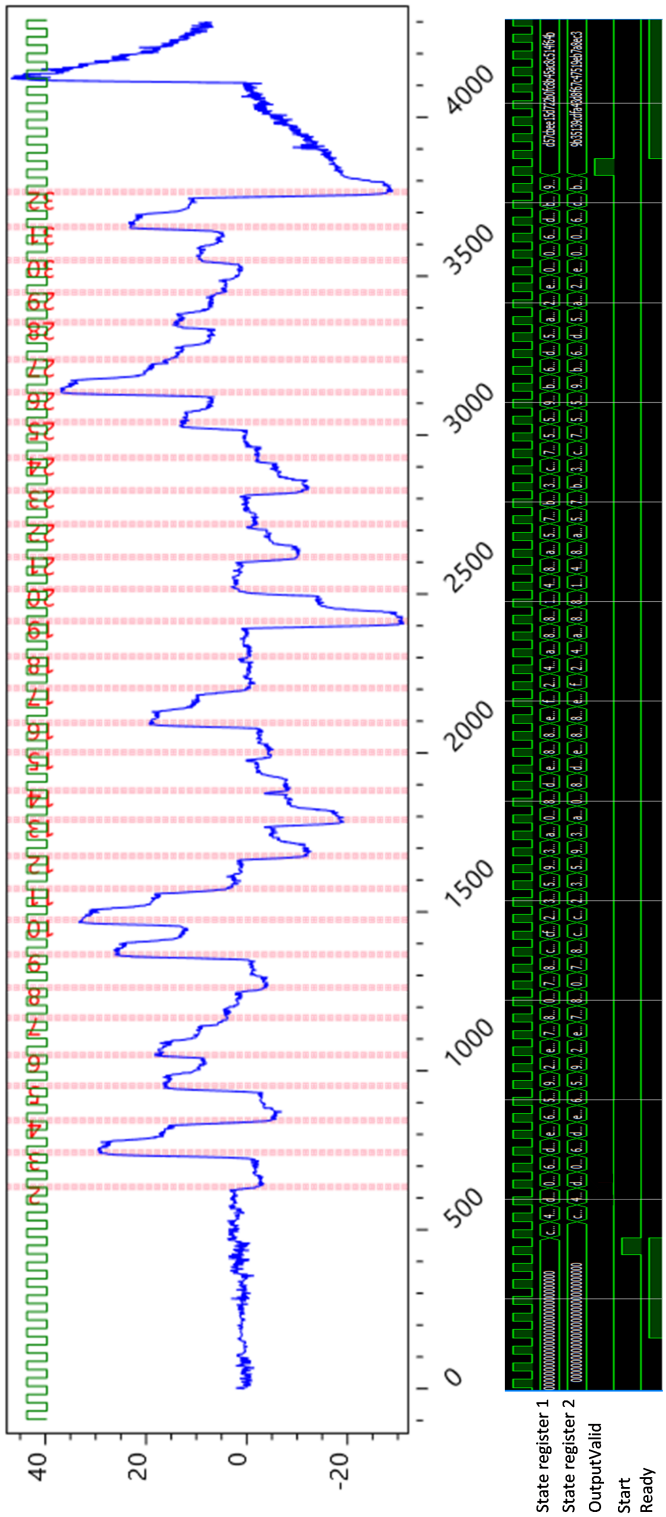
Diskutovali jsme tento problém s Nele Mentens (KU Leuven), která se domnívala, že únik může být způsoben hazardy v realizaci S-Boxu znázorněné na obrázku 1.10. Na základě její rady jsem kromě multiplexoru přidal i demultiplexor, která posílá vstup pouze do aktuálně zvoleného S-Boxu, do ostatních je posílán nulový vektor (obrázek 4.8). Tato úprava problém vyřešila.

Možné vysvětlení úniku je popsáno v následujícím textu, který se odkazuje na obrázek 1.10. Předpokládejme, že jde o první rundu, tedy `Round Index` má hodnotu `000` a jde o druhý takt této rundy (první takt pracoval s náhodnými daty - register precharge). Data x připojená na vstup tedy prochází přes bijekce R_1 a vstupují do všech bijekcí R_2^0, R_2^1 až R_2^7, přičemž je použit pouze výstup bijekce R_2^0. V dalším taktu dojde ke změně hodnoty `Round Index` a hodnota na vstupu se změní na y. Dále předpokládejme, že hodnota na `Round Index` se změní dříve, tj. že se šíří rychleji. V takovém případě se původní hodnota na výstupu `Output` změní z

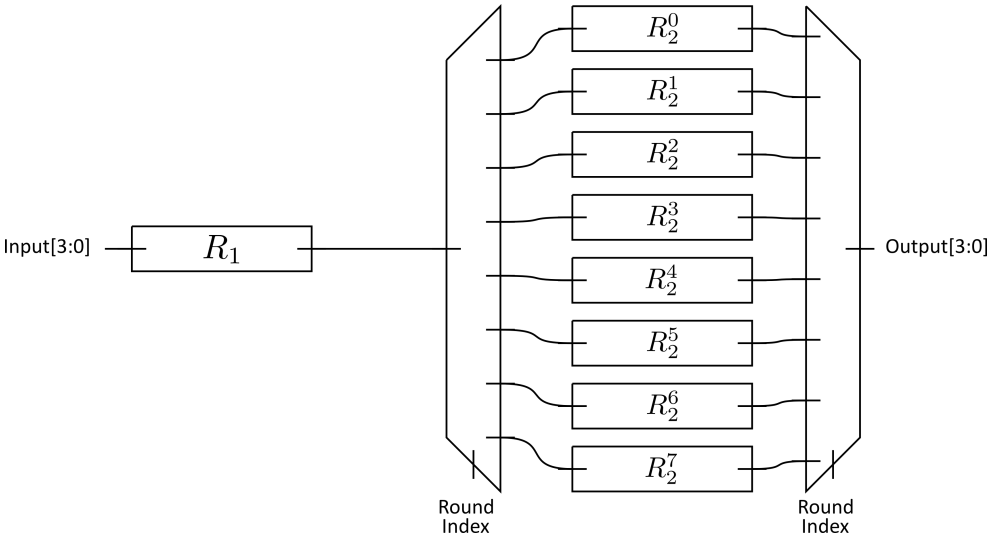
$$R_2^0(R_1(x)) \oplus Mask$$

vlivem změny signálu `Round Index` na hodnotu

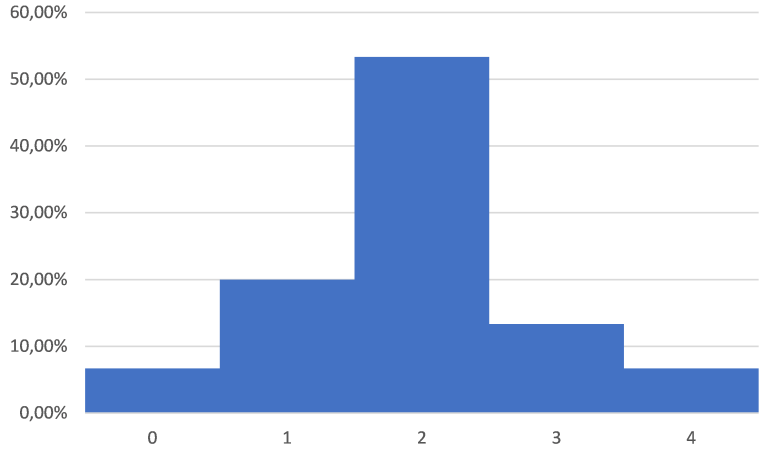
$$R_2^1(R_1(x)) \oplus Mask$$



Obrázek 4.7: Na obrázku je průběh t-hodnoty při prvotním měření SERPENTu a pod ním odpovídající výřez ze simulace zachycující hodnoty stavových registrů



Obrázek 4.8: Dekompozice S-Boxu s demultiplexorem (SERPENT)



Obrázek 4.9: Hustota Hammingových vzdáleností, bez demultiplexoru. Podrobnější popis v textu.

a až následně na konečnou hodnotu

$$R_2^1(R_1(y)) \oplus Mask$$

Zajímavá je první změna. Šifruje-li se opakovaně stejný otevřený text, je x vždy stejné a tudíž Hammingova vzdálenost této změny je vždy stejná (číslo z intervalu $< 0; 4 >$, závisí na otevřeném textu). Šifrují-li se náhodné otevřené texty, Hammingova vzdálenost se liší, lze na ní pohlížet jako na náhodnou veličinu, jejíž hustota je na obrázku 4.9 a střední hodnota je $1,875$.

Pokud naopak budeme uvažovat verzi s demultiplexorem (obrázek 4.8), dojde ke změně z

$$R_2^0(R_1(x)) \oplus Mask$$

na

$$R_2^1(0000) \oplus Mask$$

Nutno podotknout, že zobrazení R_2^1 očekává na vstupu hodnotu ochráněnou náhodnou masku. Pokud je na vstup přivedena nula, tak vzhledem k náhodně měnící se vstupní masce má výraz $R_2^1(0000)$ po každé rekonfiguraci jinou hodnotu, přičemž všechny hodnoty jsou stejně pravděpodobné.

Hammingova vzdálenost uvedené změny je tedy náhodná veličina, jejíž střední hodnota nezávisí na tom, zda se šifruje opakovaně konstantní otevřený text, nebo se šifrují náhodné otevřené texty.

Zatímco ve verzi bez demultiplexoru se střední hodnota Hammingovy vzdálenosti liší pro obě sady šifrování, u verze s demultiplexorem je střední hodnota pro obě sady rovna 2 a t-test tedy vyhodnotí, že informace neuniká.

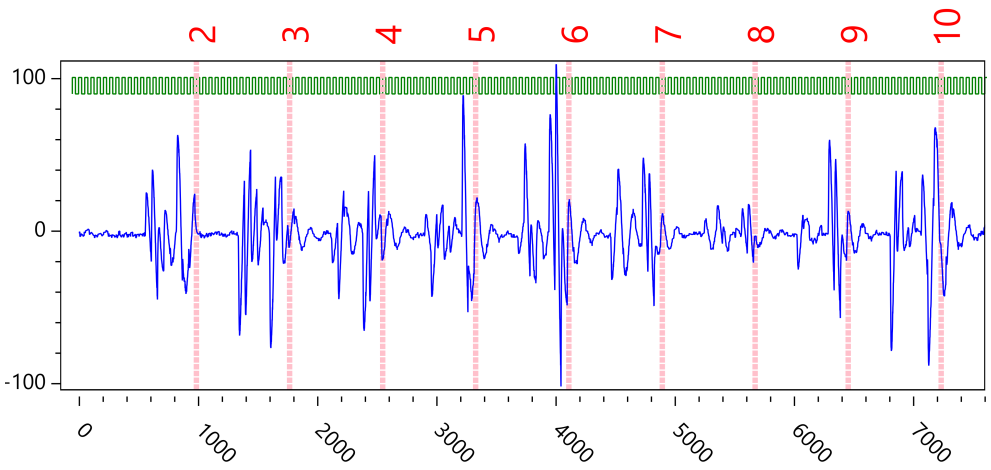
Toto vysvětlení je ale jenom hypotéza, nemám ničím prokázané, že skutečně nastává tento scénář. Odpovídá to ale grafu na obrázku 4.7, podle kterého únik informace začíná až okamžikem první změny signálu `Round Index`.

4.7 PRESENT

Implementace PRESENTu dávala dobré výsledky hned při prvotním měření. Po úpravě maskování se ale objevil malý únik na konci šifrování. Tento problém je dále rozebírán v části 6.2.

4.8 AES, S-Box realizovaný výpočtem v kompozitním tělese

Při prvotním měření se ukázalo, že u této implementace dochází k úniku informace během celého šifrování. Protože ve výpočtu inverze jsou rekonvergentní cesty (viz obrázek 3.1), přidal jsem do výpočtu inverze registry, které by měly zabránit hazardům. Vzhledem k těmto registrům trvá výpočet jedné rundy několik taktů (přibližně 13).



Obrázek 4.10: Průběh t-hodnoty, verze AESu používající kompozitní těleso a s vloženými registry proti hazardům

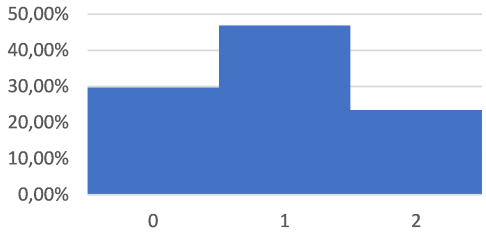
Registry byly umístěny tak, aby mezi dvěma registry byla vždy nejvýše jedna funkce. Mým cílem bylo umístit raději více registrů, než je nutné, a v případě dobrých výsledků pak zjišťovat, které registry jsou pro zlepšení nezbytné. Na to ale nakonec nedošlo, protože ani tato upravená verze nedává dobré výsledky. Umístění registrů je blíže popsáno v části 3.4.2.1 a graficky znázorněno na obrázku 3.1. Průběh t-hodnoty verze s registry je na obrázku 4.10.

4.8.1 Simulace spotřeby při šifrování

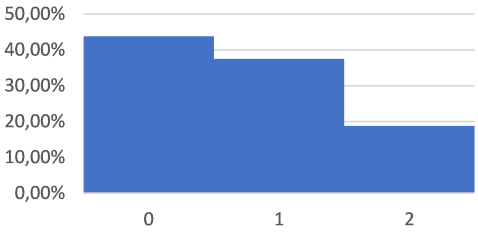
Samotný t-test prokazuje, že dochází k úniku informace, ale neposkytuje moc informací k přesnému určení příčiny. Z grafu na obrázku 4.10 lze sice vypozorovat, že k úniku informace dochází pouze v určité části výpočtu rundy, ale graf není natolik přesný, aby z něj bylo možné dělat věrohodné závěry.

Proto jsem se rozhodl udělat čistě počítačový experiment, který spočívá v provedení měření spotřeby a analýzy pomocí t-testu s tím rozdílem, že nebude měřena spotřeba na reálném FPGA, ale bude vypočítána na základě nějakého powermodelu. Až na tento rozdíl je postup stejný jako u klasického měření, tedy modeluje se spotřeba dvou sad šifrování, přičemž v jedné sadě se šifruje vždy stejný otevřený text, zatímco v druhé sadě jsou otevřené texty náhodné. Nevýhodou této simulace je, že powermodel těžko může dokonale odpovídat realitě. Výhodou je, že lze zkoumat jednotlivé signály nezávisle a odhalit tak, v které části výpočtu dochází k úniku.

Po praktické stránce jsem tento experiment realizoval tak, že jsem do VHDL kódu přidal procesy, jejichž úkolem bylo při změně hodnoty na nějakém ze sledovaných signálů zapsat novou hodnotu do souboru. Jako powermodel jsem použil Hammingovu vzdálenost po sobě jdoucích hodnot. Vzhle-



Obrázek 4.11: Hustota náhodné veličiny představující Hammingovu vzdálenost změny při šifrování náhodných dat. Střední hodnota je $0,93$ Podrobněji v textu



Obrázek 4.12: Hustota náhodné veličiny představující Hammingovu vzdálenost změny při šifrování konstantních dat. Střední hodnota je $0,75$. Podrobněji v textu

dem k tomu, že simulace je časově poměrně náročná, provedl jsem pouze 2000 šifrování. Jedno šifrování totiž trvá několik sekund (včetně nezbytné rekonfigurace).

Analýza těchto výsledků ukázala, že podle tohoto powermodelu zjevně uniká informace na signálech A, B, C, D uvnitř násobičky prvků tělesa $GF(2^4)$ (obrázek 1.13). Jde tedy o výstupy z násobičky prvků tělesa $GF(2^2)$.

4.8.2 Příčina úniku

V této části je vysvětlena příčina úniku, text se odkazuje na obrázek 1.13. Implementace obsahuje ochranu register precharge, tedy šifrování reálných dat je proložené šifrováním bloku náhodných dat. Předpokládejme, že na vstup je připojen blok náhodných dat, který je vybrán z uniformního rozdělení a tudíž jsou všechny hodnoty stejně pravděpodobné. V okamžiku další vzestupné hrany hodinového signálu dojde k připojení reálných dat na vstup, což způsobí změnu hodnoty mimo jiné na signálu A. Na Hammingovu vzdálenost této změny lze pohlížet jako na náhodnou veličinu. Jsou-li reálná data také z uniformního rozdělení (jedna ze sad t-testu), hustota této veličiny je na obrázku 4.11. Pokud se naopak šifruje vždy stejný konstantní text (druhá ze sad t-testu), může hustota této náhodné veličiny být například ta znázorněná na obrázku 4.12 (závisí na konkrétní konstantní hodnotě, v tomto případě byl předpokládán nulový vektor na vstupu násobičky). Rozdílná střední hodnota

těchto náhodných veličin způsobuje, že implementace neprojde t-testem. Mezivýsledky jsou sice zabezpečené maskováním, ale protože jsou pro obě hodnoty použity stejné masky, nemají na Hammingovu vzdálenost vliv.

Příčinou této nerovnoměrnosti je, že u operace násobení je výsledkem nulový prvek častější než kterýkoliv jiný. Pokud by místo násobení bylo jiné zobrazení s rovnoměrným rozdělením výstupních hodnot, byly by obě hustoty pravděpodobnosti stejné.

Opět jde pouze o hypotézu a není ničím prokázané, že skutečně dochází k tomuto scénáři.

4.8.3 Vylepšená verze s překladem hodnot

Na základě tohoto pozorování jsem přemýšlel nad řešením, které by zabránilo úniku. Inspiraci jsem hledal opět v článku [1], který prezentuje tři způsoby ochrany, z nichž jsou v mé implementaci použity všechny vyjma dekompozice S-Boxu. Tu zde nelze snadno provést, protože S-Box není implementován jako jedna operace, ale je složen z několika dílčích. Stejnou myšlenku lze ale aplikovat na dílčí operace. Konkrétní výstupní hodnoty operací mohou být překládány na jiné s tím, že operace, do kterých takto modifikovaná hodnota vstupuje, musí tuto změnu zohlednit.

Pro realizaci takové implementace byl vytvořen program DynReconfGenV3. Jeho bližší popis je v části 2.3, kde je také podrobněji popsán princip překladu hodnot.

Kromě toho jsem vytvořil i verzi s vloženými registry zabraňující hazardům, ale ani jedna z nich nepřinesla žádné zlepšení.

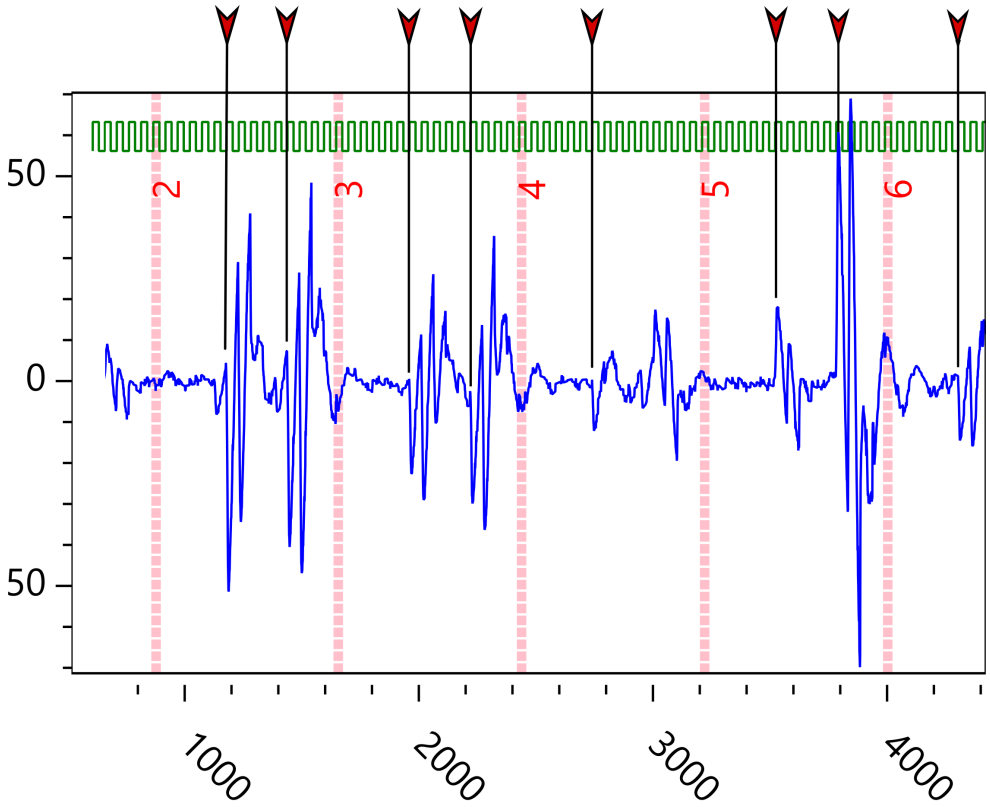
4.8.4 Určení okamžiků úniku informace pomocí grafu

Na obrázku 4.13 je část grafu t-hodnoty u implementace s vloženými registry. Z průběhu lze pohledem určit okamžiky, kdy dojde k náhlému zvýšení (nebo snížení) t-hodnoty. Tyto okamžiky jsou v grafu vyznačeny černou svislou čárou. Ne vždy se je podaří spolehlivě určit v každé rundě, protože ne vždy se t-hodnota ustálí po předešlém výkyvu a ne vždy je počátek výrazný a jednoznačný. Tam kde se to ale podařilo lze sledovat pozice těchto okamžiků.

První únik nastává 6. takt po předchozí červené svislé čáře, druhý pak 11. takt. To přesně odpovídá výstupům násobiček v tělese $GF(2^2)$ a podporuje to dříve popsanou teorii.

4.8.5 Simulace spotřeby při šifrování, verze s překladem hodnot

Abych potvrdil, že příčinou úniku je i u vylepšené verze výstup z násobičky v tělese $GF(2^2)$. Provedl jsem simulaci spotřeby podobným způsobem, jaký



Obrázek 4.13: Výřez grafu t-hodnoty s vyznačenými počátky úniku

byl dříve popisován, ale s jedním rozdílem. Nyní nepoužívám VHDL simulátor k zaznamenání mezivýsledků, ale použil jsem debugovací režim (popsán v 3.5.3). Ten odesílá do počítače nejen výsledný šifrový text, ale i jeden mezivýsledek z každé rundy. Tímto způsobem jsem získal výstupní hodnoty z násobiček. Výhodou tohoto přístupu je řádově větší rychlost. Zatímco simulování jednoho šifrování trvá jednotky sekund, tímto způsobem lze provést přibližně 100 šifrování za sekundu.

Ze získaných hodnot jsem stejně jako dříve vypočítal Hammingovu vzdálenost po sobě jdoucích hodnot a tento průběh podrobil t-testu podobně jako klasické měření. Ten potvrdil únik informace.

Problém úniku se tak nepodařilo vyřešit, ale v části 6.1 je nastíněno další vylepšení, které by mohlo být úspěšnější.

Kapitola 5

Výsledky měření

V rámci této práce bylo provedeno celkem 83 měření. V této kapitole jsou podrobněji popsány cíle a výsledky jednotlivých měření. Nejdůležitější jsou měření popsaná v části 5.1. Jejich cílem bylo prověřit odolnost implementací proti útokům odběrovou analýzou. V dalších částech je pak zkoumán vliv různých úprav.

Všechna měření byla prováděna tak, aby získaná data mohla být vyhodnocena dříve popsaným t-testem. V polovině šifrování se šifroval konstantní text (všechny bity nulové), ve zbylých šifrováních se šifroval náhodný text (pokaždé jiný). Volba mezi konstantním a náhodným textem probíhala náhodně, tj. před každým šifrováním si měřící program "hodil mincí".

Počet prováděných šifrování v jednom měření se liší pro jednotlivá měření, jde o 1 000 000 u důležitých měření a 300 000 u méně důležitých měření. Rychlost měření se pohybovala kolem 60 šifrování za sekundu.

Na přiloženém DVD je pro každé měření PDF soubor, ve kterém jsou uvedeny všechny parametry měření a výsledky.

5.1 Základní měření

Cílem základních měření je ověřit odolnost implementací pro útokům. U implementací

- AES (AesBasicPar)
- AES (AesCompFieldPar)
- PRESENT (PresentBasicPar)
- SERPENT (SerpentBasicPar)

lze zapnout tyto ochrany:

- Dekompozice S-Boxu (Dec)

- Maskování (Mask)
- Register precharge (RP)

V závorce je uvedena zkratka ochrany, používá se v názvech měření, v názvech PDF souborů na přiloženém DVD apod. Tedy například soubor `Report_Aes128-Dec-____-RP.pdf` obsahuje výsledky měření implementace AESu se zapnutou dekompozicí S-Boxu a se zapnutým register precharge. Bylo provedeno měření všech možných kombinací zapnutí/vypnutí těchto ochran, tedy celkem 8 měření pro každou implementaci, 32 měření pro zmíněné implementace. U implementace využívající kompozitní těleso byla navíc změřena varianta s vloženými registry (`Report_AesCompField128-Dec-Mask-RP_AllRegs.pdf`).

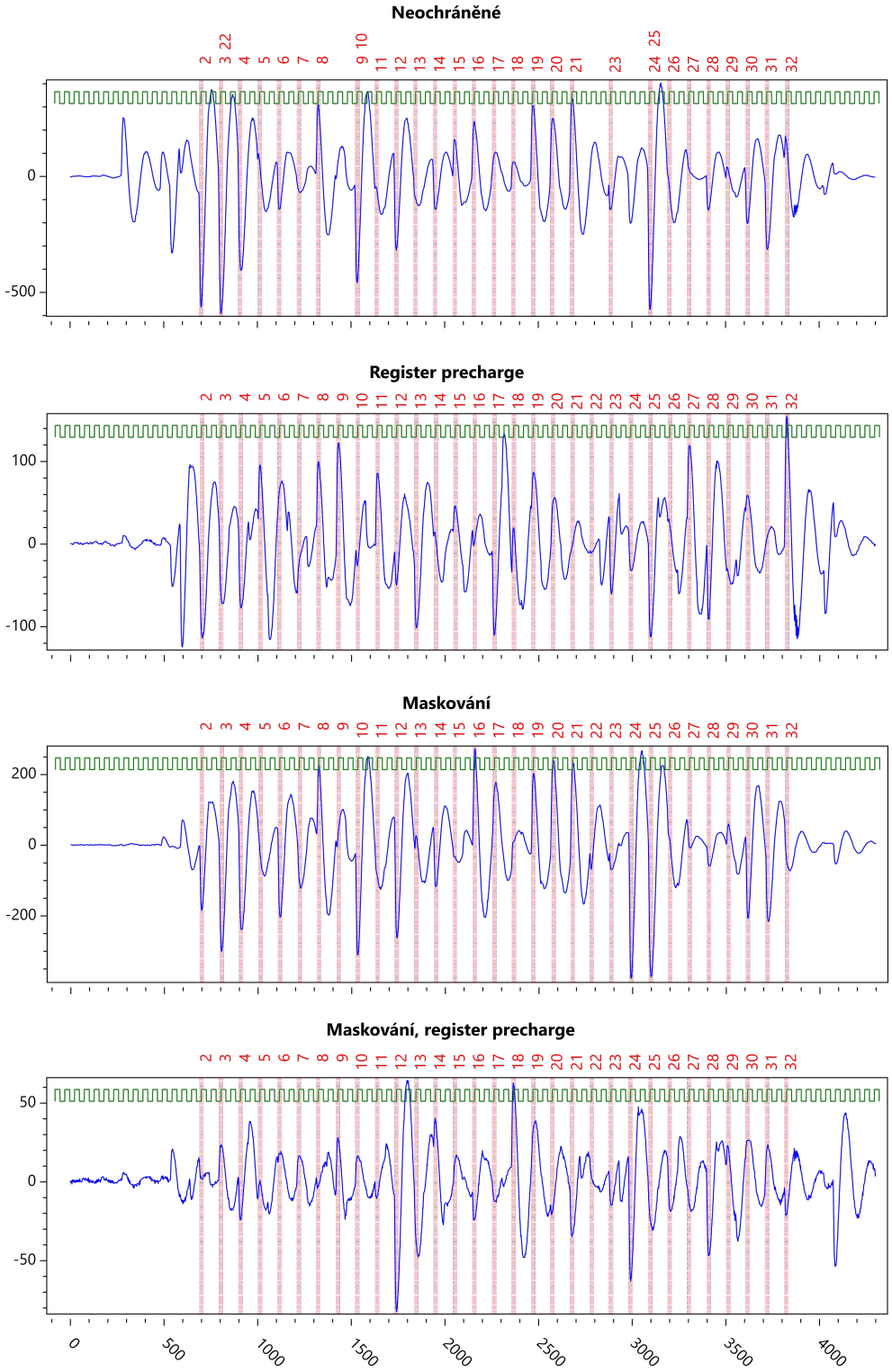
U pokročilé implementace AESu (AesCompFieldAdv) lze zapnout ochrany

- Maskování mezivýsledků v rundě (Mask)
- Maskování mezivýsledků uvnitř výpočtu S-Boxu (InMask)
- Překlad hodnot uvnitř výpočtu S-Boxu (InTr)

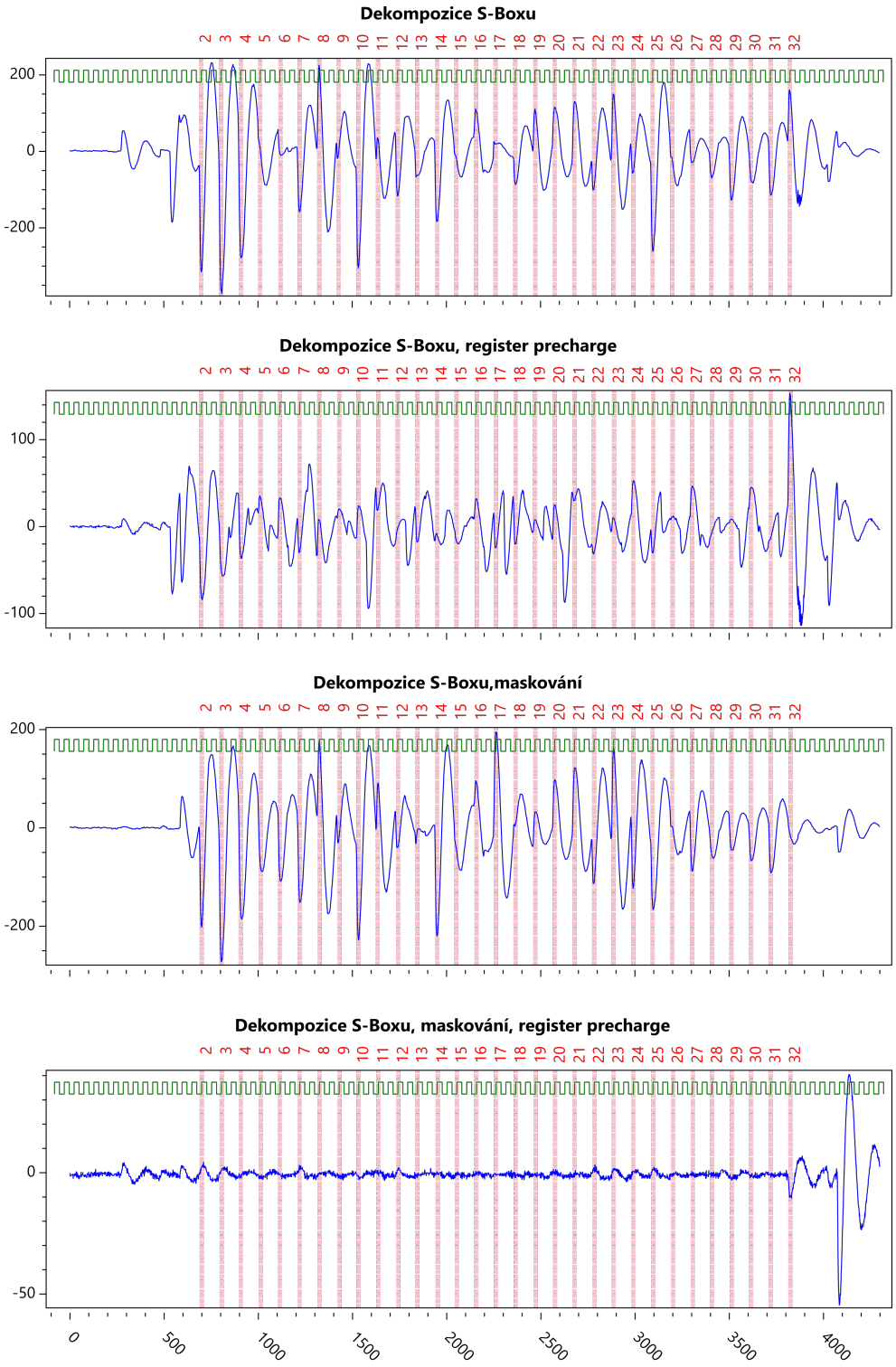
Opět bylo provedeno měření všech možných kombinací, tedy celkem 8 měření. Navíc ještě byla změřena varianta s vloženými registry, které by měly zamezovat hazardům.

Do této části tedy spadá celkem 42 měření, u všech se měřilo 1 000 000 průběhů (šifrování).

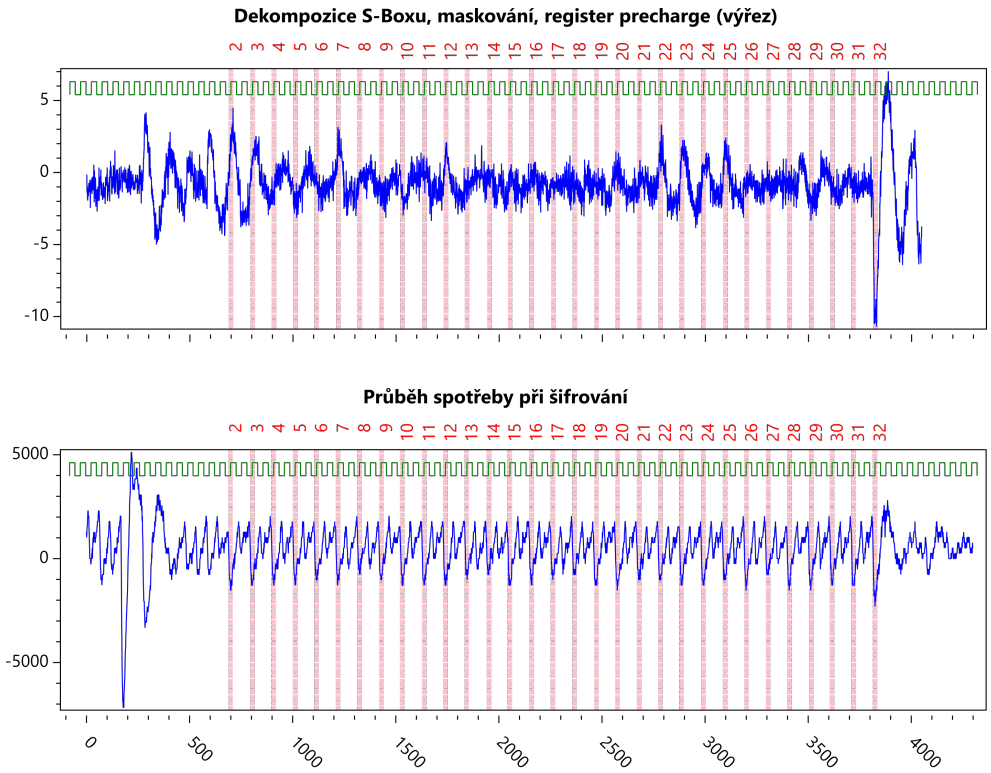
Na obrázcích 5.1 až 5.7 jsou uvedeny některé výsledky. Analýza získaných dat t-testem ukázala, že informace uniká u všech implementací používající kompozitní těleso a je patrný i velmi krátký únik v poslední rundě PRESENTu (který je již rozebírán v části 4.7 a 6.2), ten však nezpochybňuje účinnost ochran. Lze tedy říci, že úspěchu bylo dosaženo u všech implementací, vyjma implementací AESu používající kompozitní těleso.



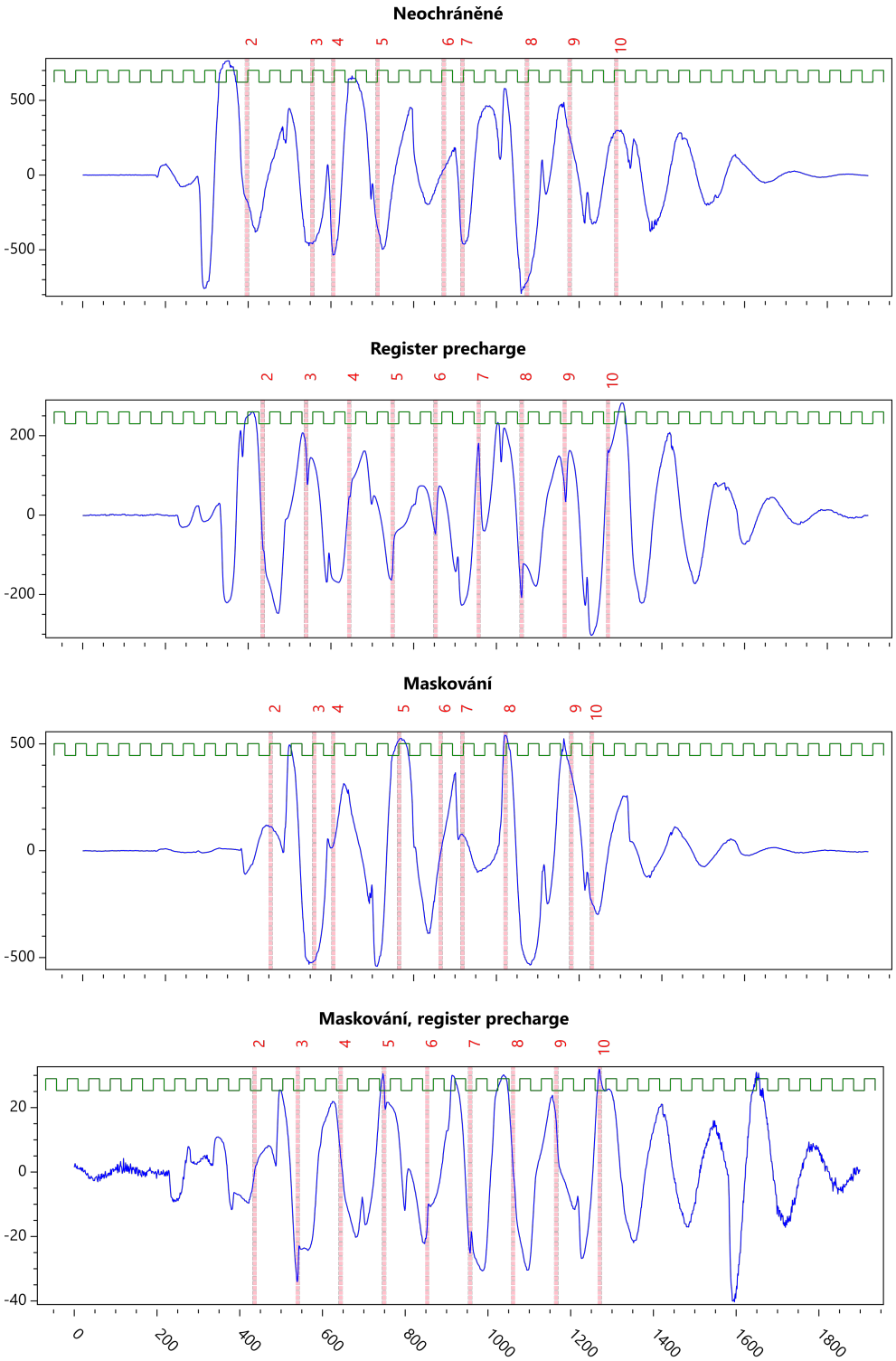
Obrázek 5.1: Výsledky PRESENT 1/3, t-hodnota



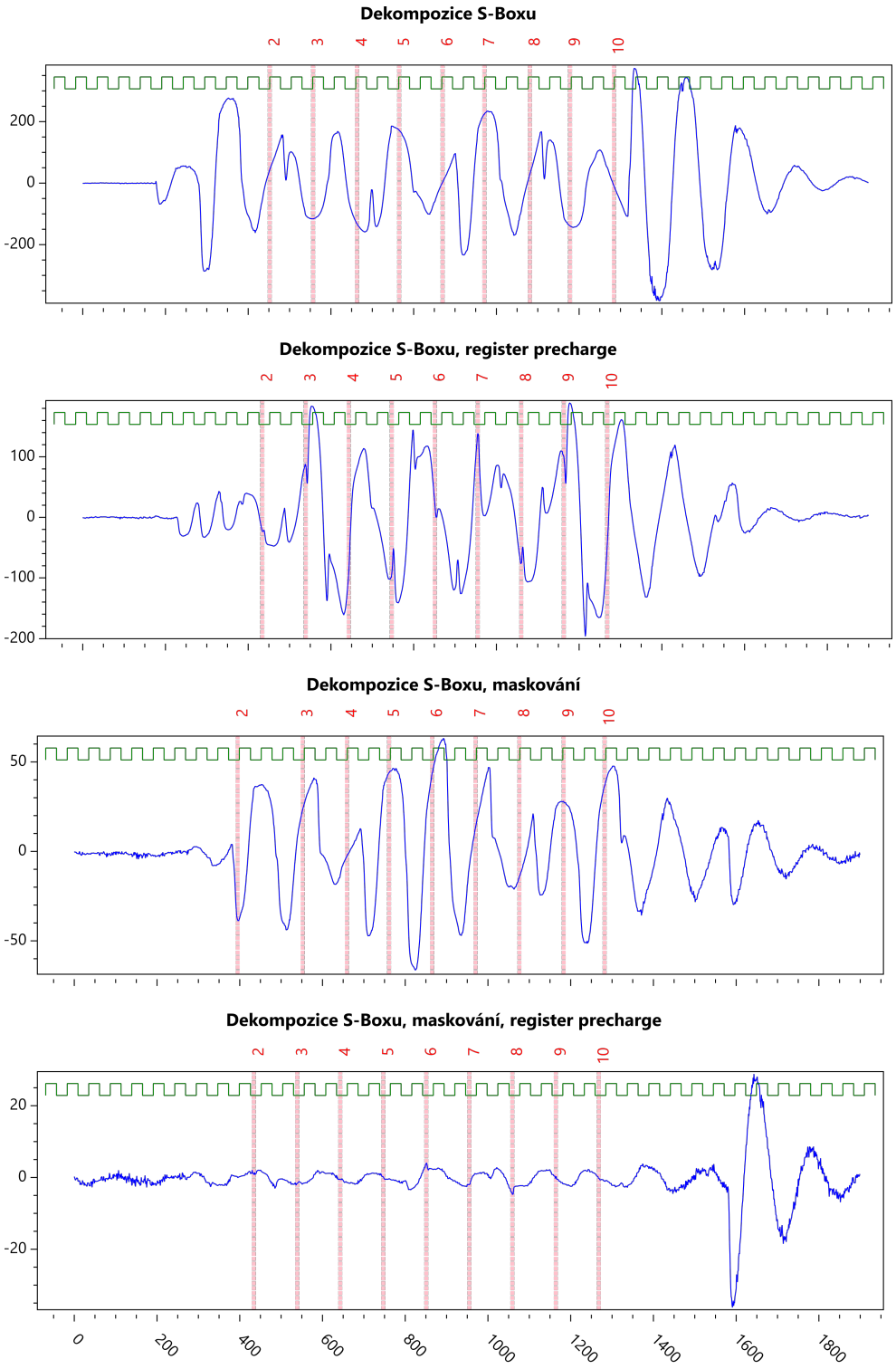
Obrázek 5.2: Výsledky PRESENT 2/3, t-hodnota



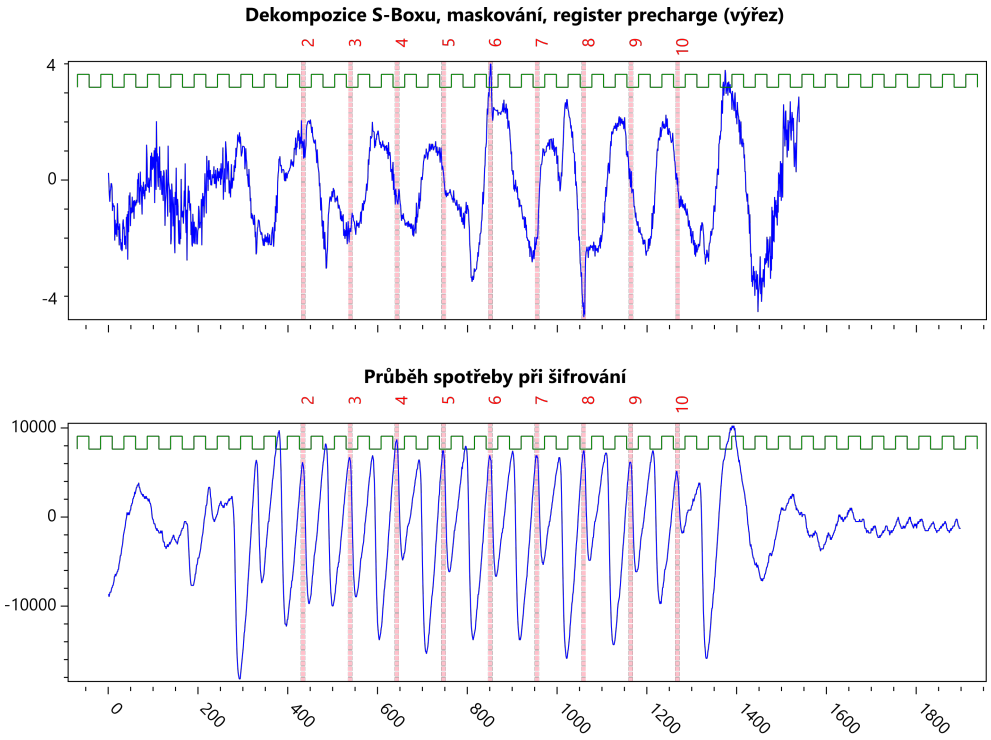
Obrázek 5.3: Výsledky PRESENT 3/3, t-hodnota a průběh spotřeby



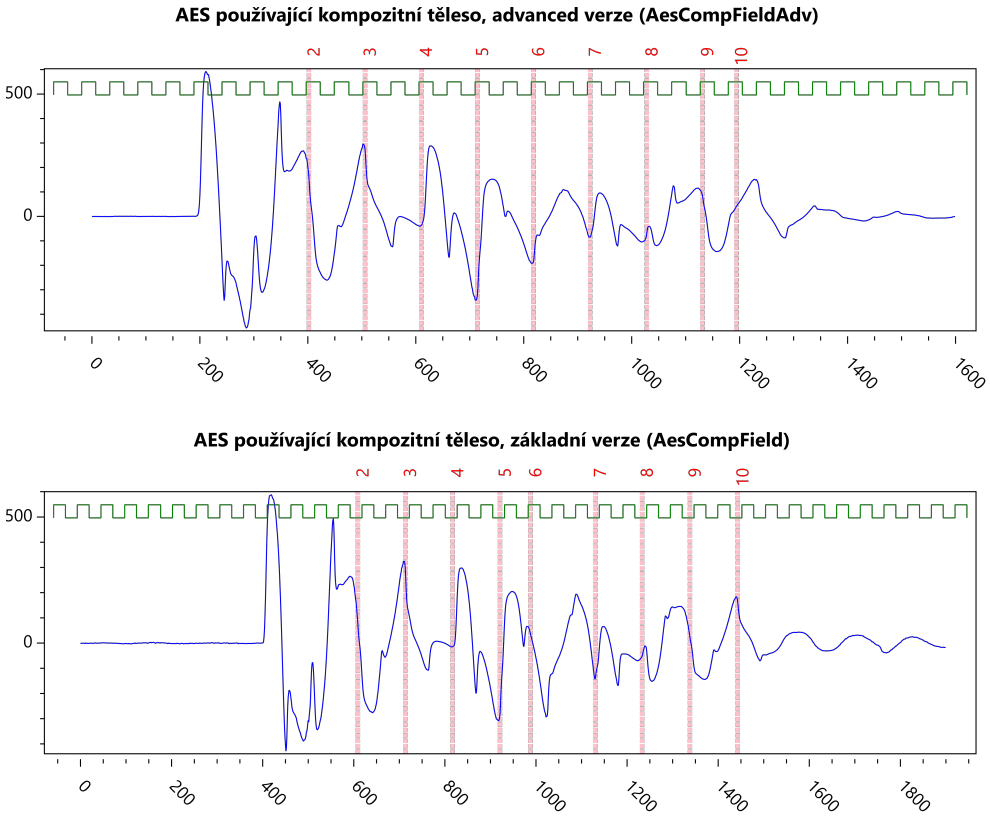
Obrázek 5.4: Výsledky AES 1/3, t-hodnota



Obrázek 5.5: Výsledky AES 2/3, t-hodnota



Obrázek 5.6: Výsledky AES 3/3, t-hodnota a průběh spotřeby



Obrázek 5.7: Výsledky, implementace AESu používající kompozitní těleso

5.2 Vliv absence druhé masky

Při dekompozici S-Boxu je hodnota mezi oběma částmi ochráněna dvěma způsoby - maskováním hodnoty a náhodnou modifikací obou bijekcí. Zajímalo mne, zda to není nadbytečné. Výsledky základního měření ukázaly, že při odstranění náhodné modifikace bijekcí dává implementace horší výsledky. Chtěl jsem tedy ověřit, zda naopak při vynechání maskování v této oblasti a ponechání náhodné modifikace bijekcí nebude stejně odolná, jako verze s oběma ochranami.

Pro implementace

- AES (AesBasicPar)
- AES (AesCompFieldPar)
- PRESENT (PresentBasicPar)
- SERPENT (SerpentBasicPar)

jsem provedl měření plně ochráněné verze s vynechaným maskováním mezi oběma bijekcemi (ostatní maskování zůstalo). V rámci každého měření bylo provedeno opět 1 000 000 šifrování.

Měření ukázalo, že obě verze dávají u implementací AESu a PRESENTu velmi podobné výsledky. Odebrání druhé masky způsobí zhoršení u implementace SERPENTu, kde absolutní hodnota t-hodnoty je nepatrně větší (až 9, oproti dosavadním 6). Porovnání obou verzí je na obrázku 5.8.

5.3 Vliv míry rekonfigurace

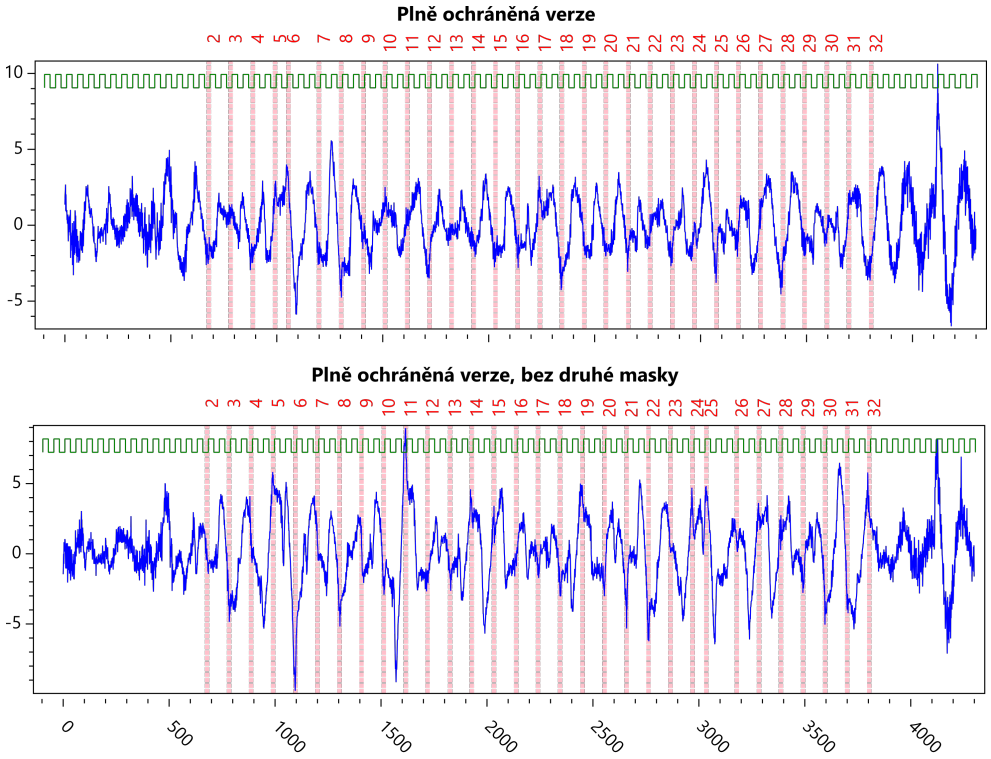
Při implementování ochran jsem zvažoval, zda je nutné modifikovat jednotlivé S-Boxy nezávisle podle různých náhodných dat, nebo stačí modifikovat je všechny stejným způsobem. První možnost je náročnější na FPGA. Zvolil jsem první možnost právě proto, abych mohl ověřit, zda má nějaký smysl, nebo není nezbytná a lze použít jednodušší řešení.

Obdobnou otázkou jsem se zabýval i u počtu náhodných modifikací bijekcí. Za náhodnou modifikaci považuji záměnu dvou řádků v pravdivostní tabulce. Provede-li se pouze jedna záměna, budou bijekce v po sobě jdoucích šifrováních zčásti stejné. Navržené implementace podporují až 8 modifikací při jedné rekonfiguraci.

Cílem experimentu je tedy porovnat tyto varianty.

V názvu příslušných měření je přídomek `Dep`, pokud jsou všechny S-Boxy modifikovány stejně, opakem je `Ind`. Počet modifikací je v názvu zapsán za zkratkou `Sw`, kde `Sw8` značí 8 modifikací a `Sw1` značí jednu modifikaci.

V každém z těchto měření bylo pro úsporu času provedeno pouze 300 000 šifrování, což je nutné zohlednit při vyvozování závěrů. Tato měření ukázala, že z hlediska prováděného způsobu testování jsou verze ekvivalentní.



Obrázek 5.8: Vliv absence druhé masky. Porovnání průběhu t-hodnot u implementace algoritmu SERPENT.

5.4 Vliv parametrů použitých při generování bitstreamu

Dále jsem se zabýval vlivem parametrů nastavených při generování bitstreamu. Je možné, že některé optimalizace probíhající při překladu VHDL kódu na bitstream způsobí změny nemající vliv na výsledek, ale mohou nějakým způsobem ovlivnit použité ochrany.

Odhadl jsem, že vliv by mohly mít tyto parametry:

- Keep hierarchy (KH)
- Register balancing (RB)
- Allow logic optimization across hierarchy (ALOAH)

Pro implementace

- AES (AesBasicPar)
- PRESENT (PresentBasicPar)

- SERPENT (SerpentBasicPar)

jsem tedy vygeneroval bitstreamy s použitím všech možných nastavení těchto parametrů, tedy 8 bitstreamů pro každou implementaci, 24 bitstreamů celkem. V každém měření jsem měřil opět pouze 300 000 šifrování.

Výsledky ukázaly, že nastavení parametrů může mít vliv. U všech implementací dopadla dobře tato varianta:

- Keep hierarchy (KH) = true
- Register balancing (RB) = false
- Allow logic optimization across hierarchy (ALOAH) = false

U jiných variant se objevují v grafu t-hodnoty špičky, a to buď výrazné špičky na začátku či konci šifrování, nebo méně výrazné špičky v průběhu šifrování.

Kapitola 6

Budoucí práce

Tato kapitole prezentuje některé poznatky pro další práci. Nejprve zmiňuje dvě vylepšení, která by mohla řešit zjištěné úniky informace při šifrování. Dále popisuje nápady, které by mohly zefektivnit budoucí práci.

6.1 Odlišné masky pro sudé a liché takty

Hlavním cílem tohoto nápadu je vyřešit problém s únikem informace u implementace AESu využívající kompozitní těleso. U ní uniká informace na výstupu násobičky kvůli nerovnoměrnému rozdělení výstupní hodnoty. Maskování problém neřeší, protože nemá žádný vliv na Hammingovu vzdálenost po sobě jdoucích hodnot. Překlad hodnot také únik neřeší, pravděpodobně z podobného důvodu - po sobě jdoucí hodnoty jsou opět přeloženy stejným způsobem.

Myšlenkou je docílit toho, aby po sobě jdoucí hodnoty byly ochráněny na základě jiných náhodných dat, tedy s použitím jiných masek a nejlépe i s jinak přeloženými hodnotami.

Mělo by být dostatečné používat různé masky pouze pro po sobě jdoucí takty, tedy v sudých taktech používat jednu masku a v lichých taktech druhou masku.

Vzhledem k použité ochraně register recharge to lze popsat i jinak: pro reálná data by se používaly jiné masky než pro blok náhodných dat. Dokonce je možné, že kdyby byly po sobě jdoucí hodnoty zamaskovány jinou maskou, nahradila by tato úprava ochranu register precharge, která je právě určena pro randomizaci Hammingových vzdáleností změn. Tato úprava by tak měla význam nejen pro zmíněnou implementaci AESu, ale i pro všechny ostatní, neboť by mohla snížit na polovinu počet taktů potřebných k zašifrování jednoho bloku. Je ale pravděpodobné, že by vznikaly hazardy, které by stejně bylo nutné eliminovat vložením dalších registrů.

6.1.1 Praktická realizace

Toto vylepšení je možné realizovat pomocí multiplexorů a demultiplexorů, které by vybíraly CFGLUTy, které jsou nakonfigurovány pro daný takt.

Tento způsob by se dal za určitých podmínek ještě trochu optimalizovat. Používané CFGLUTy mají 5 vstupních bitů, tedy nahrává se do nich pravdivostní tabulka o velikosti 32 řádků. Ve všech implementacích (kromě AESu v režimu Basic) se používá pouze pro realizace funkcí se 4bitovým vstupem, polovina CFGLUTu tedy zůstává nevyužitá. Pokud budeme uvažovat pouze dvě různé konfigurace, tedy různou pro sudé a liché takty, lze pátý vstupní bit využít pro přepínání mezi oběma konfiguracemi. Do jednoho CFGLUTu by tedy byly nahrány obě funkce, mezi kterými by se pouze přepínalo. Pravděpodobně by bylo nutné zabránit vzniku hazardů, skrz které by mohla unikat informace. Tento způsob je dle mého názoru nejvhodnější a použil bych ho pro případnou realizaci. Příklad takové konfigurace CFGLUTů je na obrázku 6.1.

Toto vylepšení by se dalo realizovat rozšířením programu DynReconfGenV3, který by musel umět nahrát do jednoho CFGLUTu dvě funkce, přičemž obě by byly ochráněné na základě jiných náhodných dat (různé masky a překlady hodnot). Musel by také zajistit, aby se za běhu správně přepínalo mezi oběma funkcemi.

6.2 Přenos maskovaného otevřeného a šifrového textu

U některých implementací jsem se potýkal s únikem informace na konci nebo na začátku šifrování. Konkrétně jde o implementaci AESu (bez kompozitního tělesa). Po úpravě maskování se podobný problém objevil zase u implementace PRESENTu, tentokrát na konci šifrování. Domnívám se, že tento únik způsobuje práce s nezamaskovanou hodnotou, tedy s otevřeným textem na začátku šifrování nebo s šifrovým textem na konci šifrování.

Tento únik by se dal eliminovat posíláním zamaskovaného otevřeného a šifrového textu po sériové lince. Do FPGA by se tedy posílal zamaskovaný otevřený text společně se zvolenou maskou a FPGA by naopak zpět posílalo zamaskovaný šifrový text a použitou masku. Jinak řečeno, přičtení masky na počátku šifrování by neprobíhalo v FPGA (obálkové entitě), ale již v počítači. FPGA by podle přijaté masky pouze muselo správně rekonfigurovat CFGLUTy.

6.3 Simulování spotřeby

V další práci bych zvážil zabývat se více simulováním spotřeby navrhované implementace. V této práci jsem se modelováním spotřeby zabýval až po vy-

x	$G_1(x)$
(I_4,I_3,I_2,I_1,I_0)	(O_3,O_2,O_1,O_0)
00 (00000)	B (1011)
01 (00001)	8 (1000)
02 (00010)	9 (1001)
03 (00011)	E (1110)
04 (00100)	F (1111)
05 (00101)	C (1100)
06 (00110)	D (1101)
07 (00111)	2 (0010)
08 (01000)	3 (0011)
09 (01001)	0 (0000)
0A (01010)	1 (0001)
0B (01011)	6 (0110)
0C (01100)	7 (0111)
0D (01101)	4 (0100)
0E (01110)	5 (0101)
0F (01111)	A (1010)
10 (10000)	4 (0100)
11 (10001)	7 (0111)
12 (10010)	6 (0110)
13 (10011)	1 (0001)
14 (10100)	0 (0000)
15 (10101)	3 (0011)
16 (10110)	2 (0010)
17 (10111)	D (1101)
18 (11000)	C (1100)
19 (11001)	F (1111)
1A (11010)	E (1110)
1B (11011)	9 (1001)
1C (11100)	8 (1000)
1D (11101)	B (1011)
1E (11110)	A (1010)
1F (11111)	5 (0101)

x	$G_2(x)$
(I_4,I_3,I_2,I_1,I_0)	(O_3,O_2,O_1,O_0)
00 (00000)	B (1011)
01 (00001)	C (1100)
02 (00010)	9 (1001)
03 (00011)	A (1010)
04 (00100)	F (1111)
05 (00101)	0 (0000)
06 (00110)	D (1101)
07 (00111)	E (1110)
08 (01000)	3 (0011)
09 (01001)	4 (0100)
0A (01010)	1 (0001)
0B (01011)	2 (0010)
0C (01100)	7 (0111)
0D (01101)	8 (1000)
0E (01110)	5 (0101)
0F (01111)	6 (0110)
10 (10000)	6 (0110)
11 (10001)	5 (0101)
12 (10010)	8 (1000)
13 (10011)	7 (0111)
14 (10100)	2 (0010)
15 (10101)	1 (0001)
16 (10110)	4 (0100)
17 (10111)	3 (0011)
18 (11000)	E (1110)
19 (11001)	D (1101)
1A (11010)	0 (0000)
1B (11011)	F (1111)
1C (11100)	A (1010)
1D (11101)	9 (1001)
1E (11110)	C (1100)
1F (11111)	B (1011)

Obrázek 6.1: Příklad, odlišné masky pro sudé a liché takty. Na obrázku jsou znázorněny dvě konfigurace CFGLUTů, G_1 a G_2. Každá z nich obsahuje dvě stejné funkce s tím rozdílem, že mezivýsledek mezi nimi je maskován jinou maskou. Maska výstupu horní funkce je 1010, maska výstupu dolní funkce je 0101. Mezi oběma funkcemi se přepíná bitem I_4. Všechny funkce realizují inkrementaci vstupu o jedna.

tvoření implementace, abych zjistil příčinu úniku informace. Kdybych si ověřil správnost nejprve simulací, ušetřil bych čas při vytváření, testování a měření nedostatečně ochráněné implementace. Těžko si to zpětně vyčítat, protože jsem neměl dostatek zkušeností a znalostí, které jsem získal právě tímto krokem vedle, ale pro další práci bych tuto možnost zvážil, pokud bych si nebyl jistý, že navrhovaný způsob ochrany bude fungovat.

6.4 Generování konfigurací CFGLUTů počítačem

Tento nápad se snaží řešit stejný problém jako výše uvedený, tedy efektivitu práce. V rámci této práce byly vytvořeny dvě odlišné implementace AESu, které používají kompozitní těleso k realizaci S-Boxu.

Obě si vyžádaly nemalé množství práce a ani jedna z nich nepřinesla kýžený výsledek. Mezi sebou se liší hlavně způsobem, jakým jsou generovány konfigurace CFGLUTů, samotné propojení je stejné a odpovídá obrázku 3.1.

Efektivnější by bylo nezabývat se generováním konfigurací v FPGA, ale jejich generování přenechat na měřící program. Implementovat tento úkol v počítači by bylo značně jednodušší a časově méně náročné než v FPGA. V FPGA by byly pouze propojené instance CFGLUTů, samotné konfigurace by se nahrály před šifrováním z počítače. Tímto způsobem by se dalo ověřit, zda způsob rekonfigurace přináší očekávané výsledky. Pokud ano, přistoupilo by se ke generování konfigurací v FPGA.

Je důležité zdůraznit, že tento nápad je určen pouze pro experimenty, pro důvěryhodná měření je nutné, aby FPGA fungovalo zcela samostatně.

Opět je třeba zvážit, zda práce potřebná pro tento krok navíc bude vykompenzována úsporou času, která vznikne nerealizováním nedostatečných implementací.

Závěr

Celá práce by se dala rozdělit na tři dílčí části. Do první části by patřilo zkoumání dynamické rekonfigurace a její snadné implementace. Šlo o hledání způsobu, jak snadno a rychle realizovat rekonfigurující se zapojení, přičemž tento způsob by měl být znovupoužitelný, aby se dal použít při implementaci algoritmů PRESENT, SERPENT, AES a nejlépe byl natolik obecný, aby se dal použít i pro implementaci jiných algoritmů v budoucnu. Konkrétně jde o vytvořené nástroje DynReconfGen a DynReconfGenV3. Tato část není explicitně zmíněna v zadání práce, ale je to část, která ušetřila mnoho času a umožnila tak věnovat se více samotné podstatě této práce.

Do druhé části by se dala zařadit implementace šifrovacích algoritmů AES, SERPENT a PRESENT s použitím ochran popsaných v článku [1]. V případě PRESENTu šlo čistě jenom o reimplementaci toho, co již bylo dokázáno a ověřeno v uvedeném článku. Zmíněná protiopatření poté byla použita u algoritmů SERPENT a AES. Bylo přitom nutné vypořádat se s některými odlišnosti od algoritmu PRESENT. Vzhledem k velké podobnosti všech algoritmů šlo ale pouze o malé odlišnosti, a proto bych tuto část označil jako čistě implementační.

Do poslední části by patřila implementace AESu, která používá kompozitní konečné těleso pro výpočet S-Boxu. Na rozdíl od druhé části nejde o použití již existujících poznatků, ale o hledání nového způsobu realizace AESu odolného proti odběrové analýze.

Pokud jde o část zabývající se implementací algoritmů, tak tu lze bezpochyby hodnotit jako úspěšnou, protože se podařilo vytvořit implementace všech tří algoritmů, které byly z hlediska použitého t-testu označeny jako odolné. U hledání nového způsobu implementace AESu (kompozitní těleso) se sice nepodařilo dosáhnout tohoto úspěchu, nicméně i u ní se podařilo prozkoumat příčiny úniku a realizovat ještě jednu vylepšenou verzi. Ani ta sice nepřinesla kýžený výsledek, ale bylo vymyšleno další možné vylepšení, které by v budoucnu jistě stálo za realizaci. Zatímco přínosem implementační části jsou dosažené výsledky, u zkoumání možného využití kompozitního tělesa považuji

za nejcennější zkušenosti a poznatky pro další práci.

Literatura

[1] Sasdrich, P.; Moradi, A.; Mischke, O.; aj.: Achieving Side-Channl Protection with Dynamic Logic Reconfiguration on Modern FPGAs. *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, McLean, VA, USA*, květen 2015. Dostupné z: `http://www.seceng.rub.de/research/publications/host15_present/`

[2] Bogdanov, A.; Knudsen, L. R.; Leander, G.; aj.: PRESENT: An Ultra-Lightweight Block Cipher. *Paillier P., Verbauwhede I. (eds) Cryptographic Hardware and Embedded Systems - CHES 2007. CHES 2007. Lecture Notes in Computer Science, vol 4727. Springer, Berlin, Heidelberg*, 2007. Dostupné z: `http://www.lightweightcrypto.org/present/present_ches2007.pdf`

[3] NIST: *Advanced Encryption Standard (AES)*. 2001. Dostupné z: `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf`

[4] Anderson, R.; Biham, E.; Knudsen, L.: Serpent: A Proposal for the Advanced Encryption Standard. 09 2000. Dostupné z: `https://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf`

[5] Kocher, P.; Jaffe, J.; Jun, B.: Differential Power Analysis. *Wiener M. (eds) Advances in Cryptology — CRYPTO' 99. CRYPTO 1999. Lecture Notes in Computer Science, vol 1666. Springer, Berlin, Heidelberg*, 1999. Dostupné z: `https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf`

[6] Kocher, P. C.: Timing attacks on Implementations of Diffie-Hellman, RSA, DSS, and other systems. *Koblitz N. (eds) Advances in Cryptology — CRYPTO '96. CRYPTO 1996. Lecture Notes in Computer Science, vol 1109. Springer, Berlin, Heidelberg*, 1996. Dostupné z: `https://www.paulkocher.com/doc/TimingAttacks.pdf`

[7] Genkin, D.; Shamir, A.; Tromer, E.: RSA Key Extraction via Low-Bandwidth Acoustic Cryptoanalysis. *Garay J.A., Gennaro R. (eds) Advances in Cryptology – CRYPTO 2014. CRYPTO 2014. Lecture Notes in Computer Science, vol 8616. Springer, Berlin, Heidelberg*, 2013. Dostupné z: http://www.cs.tau.ac.il/~tromer/papers/acoustic-20131218.pdf

[8] Prouff, E.; Rivain, M.: Masking against Side-Channel Attacks: A Formal Security Proof. *Johansson T., Nguyen P.Q. (eds) Advances in Cryptology – EUROCRYPT 2013. EUROCRYPT 2013. Lecture Notes in Computer Science, vol 7881. Springer, Berlin, Heidelberg*, 2013.

[9] Medwed, M.: Overview of Countermeasures against Implementation Attacks. Dostupné z: https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/marcel_medwed_countermeasures.pdf

[10] Mentens, N.; Batina, L.; Preneel, B.; aj.: A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box. *A.J. Menezes (Ed.): CT-RSA 2005, LNCS 3376, pp. 323–333, 2005. Springer-Verlag Berlin Heidelberg*, 2005. Dostupné z: https://www.researchgate.net/profile/Lejla_Batina/publication/221208406_A_Systematic_Evaluation_of_Compact_Hardware_Implementations_for_the_Rijndael_S-Box/links/0912f50cef031d51c9000000.pdf

[11] Xilinx Inc.: *Spartan-6 Libraries Guide for HDL Designs*. 2009. Dostupné z: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan6_hdl.pdf

[12] Shannon, C. E.: The Synthesis of Two-Terminal Switching Circuits. *The Bell System Technical Journal*, 1949. Dostupné z: https://ia802700.us.archive.org/6/items/bstj28-1-59/bstj28-1-59.pdf

[13] Olšák, R. P.: Lineární algebra. [Online, cit. 6. 1. 2019]. Dostupné z: http://petr.olsak.net/ftp/olsak/linal/linal2.pdf

[14] Paar, C.: *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. Dizertační práce, Institute for Experimental Mathematics, Universität Essen, 7 1994. Dostupné z: http://www.emsec.ruhr-uni-bochum.de/media/crypto/attachments/files/2010/04/paar_php_diss.pdf

[15] Doc. RNDr. Jiří Tůma, D.: Konečná tělesa, 2. přednáška. [Online, cit. 6. 1. 2019]. Dostupné z: http://www.karlin.mff.cuni.cz/~tuma/ffields.html

[16] Goodwill, G.; Jun, B.; Jaffe, J.; aj.: A testing methodology for side-channel resistance validation. *Non-Invasive Attack Testing Workshop*, 2011. Dostupné z: https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf

[17] MORITA TECH CO., LTD.: *SAKURA-G Specifications*. 2013. Dostupné z: http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G_Spec_Ver1.0_English.pdf

Příloha A

Útok na implementaci PRESENTu odběrovou analýzou

V rámci této práce byl proveden i experimentální útok odběrovou analýzou s cílem zjistit použitý šifrovací klíč. V tomto dodatku je ve stručnosti popsán způsob provedení útoku a následně dosažené výsledky.

A.1 Testované implementace

Útok byl vyzkoušen na dvou implementacích

- Na implementaci PRESENTu (80bitový klíč), která vznikla v rámci této práce. Měření bylo prováděno na desce Sakura-G.
- Na jiné implementaci PRESENTu (128bitový klíč), která je předchůdcem zde popisované implementace. Byla vytvořena Ing. Stanislavem Jeřábkem a autorem této práce. Tato verze se liší zejména tím, že pro realizaci rekonfigurace nebyl použit nástroj DynGenReconf ani způsob popisovaný v části 2.1 (V1). Samotná rekonfigurace byla napsána zcela ručně a jednoúčelově pro potřeby tohoto algoritmu. Měření bylo prováděno na desce DPA Card rev1.0.

A.2 Powermodel

Pro útok na šifrovací klíč je nezbytný powermodel, který predikuje spotřebu v závislosti na hodnotě otevřeného (nebo šifrového) textu pro každou možnou hodnotu nějaké části šifrovacího klíče.

Powermodel musí odpovídat implementaci, na kterou je útočeno. Vzhledem k tomu, že se jedná o FPGA, tak lze předpokládat, že spotřeba energie se odvíjí od toho, k jak velké změně dojde oproti předchozímu taktu. Jde tedy o Hammingovu vzdálenost mezi dvěma po sobě jdoucími hodnotami. Je

útočeno na poslední rundu, tedy je brána v úvahu Hammingova vzdálenost mezi poslední hodnotou ve stavovém registru v poslední rundě a hodnotou následující. Je tedy předpokládáno, že zápis do stavového registru je proveden i po skončení šifrování (což ale vůbec není nutné). Pokud by se tak nedělo, je možné útočit naopak na první rundu, protože hodnota ve stavovém registru před první rundou by zůstala nezměněná od poslední rundy předchozího šifrování a tedy známá pro útočníka. Díky tomu by bylo možné dopočítat Hammingovu vzdálenost i v první rundě.

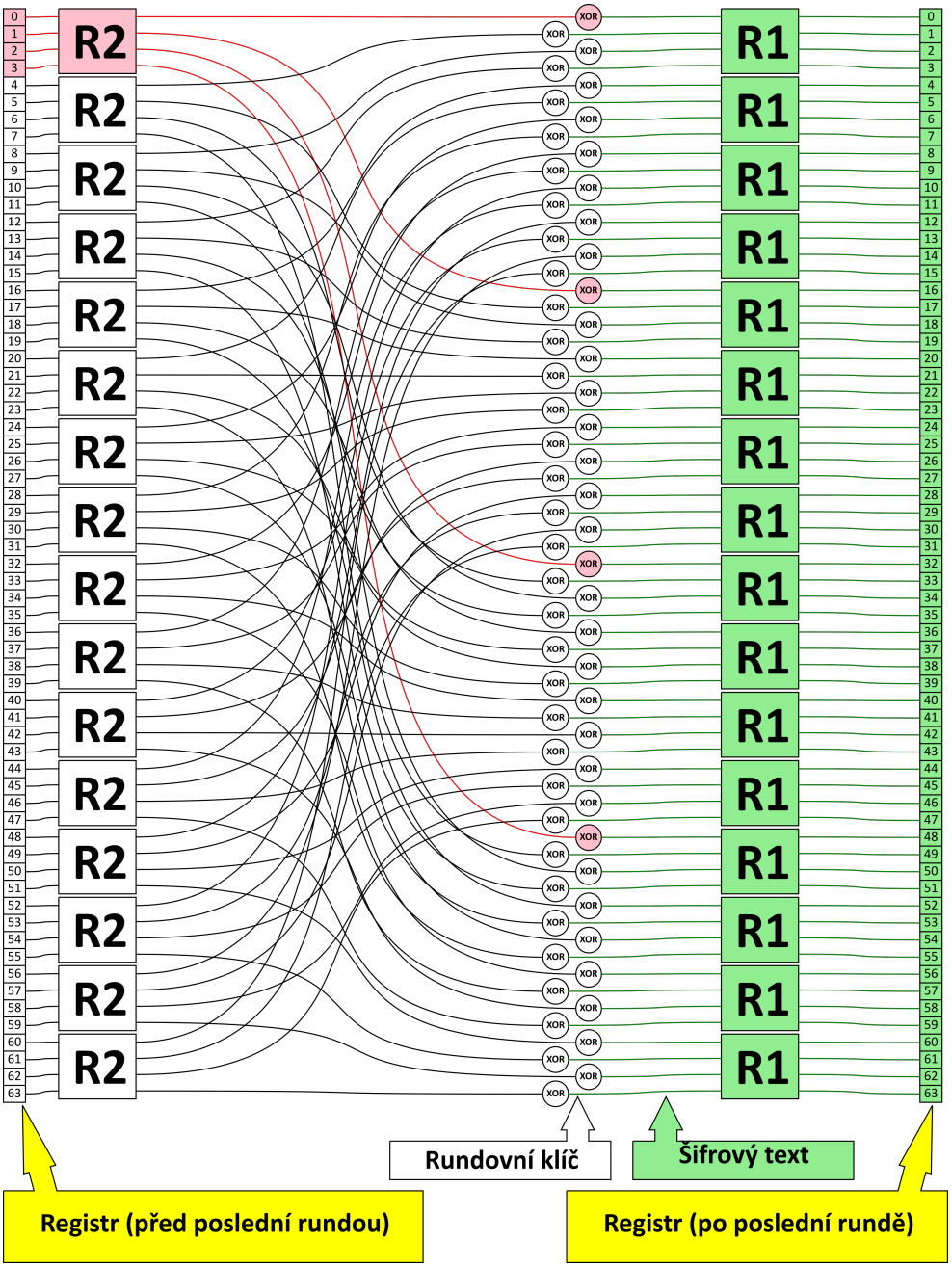
Vstupem powermodelu tedy bude šifrový text (protože útočím na poslední rundu) a nějaká část rundovního klíče. Z těchto hodnot powermodel určí hodnoty alespoň části bitů ve stavovém registru a vyhodnocením Hammingovy vzdálenosti pak získá očekávanou spotřebu pro danou hodnotu části rundovního klíče. Hodnota rundovního klíče, pro kterou modelovaná spotřeba nejvíce koreluje se skutečnou spotřebou, je považována za správnou. Tento postup je nutné opakovat pro všechny části rundovního klíče.

Útok na PRESENT je v porovnání s AESem výpočetně méně náročný. PRESENT používá 4bitový S-Box, takže by mělo být možné útočit na jednotlivé čtveřice bitů rundovního klíče (celkem tedy 16 možných hodnot), což je méně výpočetně náročné než útok na osmice bitů (256 možných hodnot) v případě AESu. Nelze obecně říct, že pokud šifra používá 4bitový S-Box, lze útočit na jednotlivé čtveřice. Příkladem může být SERPENT, který sice také má 4bitový S-Box, ale runda navíc obsahuje lineární transformaci, kvůli které ani 4 bity rundovního klíče nestačí na dopočítání alespoň části hodnoty ve stavovém registru v předchozím taktu.

Útok na jednu čtveřici bitů popisuje obrázek A.1. Na levém a pravém okraji obrázku je znázorněn stavový registr. V reálné implementaci jde o jeden a ten samý registr, ale zde je pro přehlednost nakreslen dvakrát. Pravý registr vlastně představuje následující hodinový takt. Úkolem powermodelu je tedy dopočítat alespoň část bitů v pravém registru a stejnou část bitů v levém registru, aby mohl určit Hammingovu vzdálenost, která by měla odpovídat spotřebě.

Na obrázku A.1 je S-Box dekomponován na dvě bijekce R1 a R2. Protože jsou předpokládány vypnuté ochrany, jedna z nich odpovídá S-Boxu a druhá je pouze identita. V dalším textu je odvozena očekávaná spotřeba za předpokladu, že R1 je identita a R2 je S-Box. To odpovídá situaci, kdy stavový registr je umístěn mezi přičítání rundovního klíče a S-Box.

Šifrovým textem je u algoritmu PRESENT hodnota po přičtení posledního rundovního klíče. Tedy jinak řečeno, v poslední rundě je vynechán S-Box a permutace. Na obrázku jsou zeleně vyznačeny bity, které jsou známé - jde vlastně o šifrový text. Pro každou z možných hodnot bitů klíče na indexech 0, 16, 32, 48 je možné vypočítat inverzní hodnotu S-Boxu a získat tak hodnoty bitů 0 až 3 ve stavovém registru vlevo (vyznačeny růžově). Hammingova vzdálenost mezi čtveřicí bitů v levém registru a pravém registru pak odpovídá očekávané spotřebě. Postup je nutné opakovat pro všechny části rundovního



Obrázek A.1: Znázornění výpočtu powermodelu pro bity 0, 16, 32, 48 rundovního klíče.

klíče (v dalším kroku lze útočit na bity 1, 17, 33, 49 atd.).

A.3 Výpočet původního klíče z rundovních klíčů

Proces generování rundovních klíčů se skládá z operací

- S-Box
- Cyklický bitový posun
- Přičtení (XOR) indexu aktuální rundy

Všechny tyto dílčí operace jsou invertovatelné, tedy je možné zpětně rekonstruovat původní klíč. Šifrovací klíč může být dlouhý 80 nebo 128 bitů. Protože délka rundovního klíče odpovídá délce bloku, tedy 64 bitů, je potřeba více rundovních klíčů ke zpětnému výpočtu původního klíče.

V případě 80bitové varianty jsou nutné dva po sobě jdoucí rundovní klíče. V případě 128bitové varianty je potřeba ještě o jeden rundovní klíč více. Ani ze dvou rundovních klíčů totiž nelze určit celý key register, stále zbývají 3 neznámé bity, jak znázorňuje obrázek A.2. Ty je možné určit buď útokem na další rundovní klíč, nebo dopočítat původní klíč pro všech $2^3 = 8$ možností a z nich pak vybrat ten správný. Druhý způsob by byl méně výpočetně náročný a proto lepší pro praktické použití. Cílem tohoto experimentu je ale ověřit funkčnost CPA útoku, a proto bude použit první způsob.

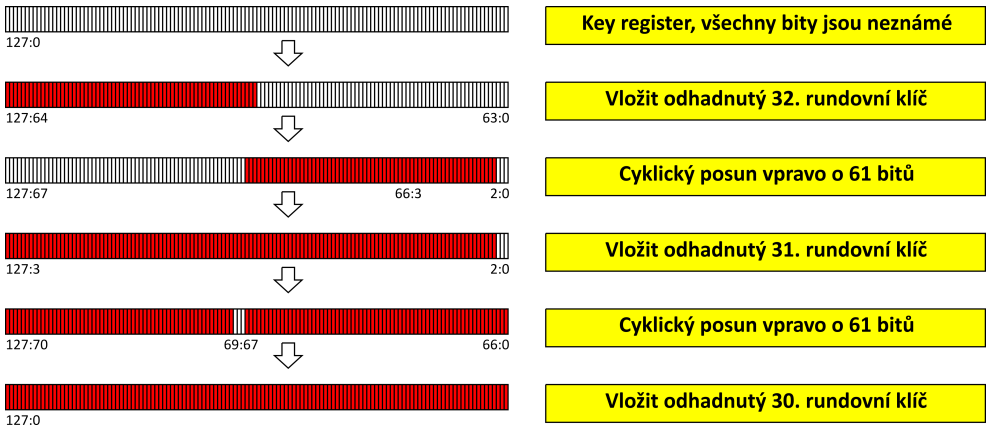
Útok na 31. resp. 30. rundovní klíč se provádí stejným způsobem jako na 32. rundovní klíč. Rozdíl je v tom, že powermodel potřebuje znát i následující rundovní klíče pro výpočet hodnoty ve stavovém registru. z tohoto důvodu je nutné útočit odzadu a postupně, tedy nejprve na 32. rundovní klíč. Jeho hodnota pak bude použita při útoku na 31. rundovní klíč atd.

A.4 Program PresentCPA

Pro realizaci útoku byl vytvořen program PresentCPA, který je přílohou této práce. Je napsán v jazyce C# (.NET Core 2.0).

Vstupem programu je

- Průběh spotřeby při šifrování. Binární soubor, ve kterém jsou průběhy při jednotlivých šifrování umístěny bezprostředně za sebou bez jakýchkoliv dalších formátovacích znaků. Sample může být typu `UInt8`, `Int8` (1 bajt na sample) nebo `Int16` (2 bajty na jeden sample).
- Soubor s šifrovými texty. Na každém řádku je jeden šifrový text (jako hexadecimální text), počet řádků tedy odpovídá celkovému počtu průběhů (tedy celkovému počtu provedených šifrování).



Obrázek A.2: Výpočet hodnoty key register ze 3 posledních rundovních klíčů, 128bitová verze.

- Soubor s parametry, který obsahuje informaci o počtu vzorků (samplů) v každém průběhu, celkovém počtů průběhů a podobně.

Soubor s parametry je ve formátu XML a má následující strukturu:

```
1  <InputArgsXML>
2    <SamplesCount>56000</SamplesCount>
3    <TracesCount>5219</TracesCount>
4    <TracesLimit>3000</TracesLimit>
5    <TracesFile>traces.bin</TracesFile>
6    <CipherTextFile>ciphertext2.txt</CipherTextFile>
7    <Algorithm>Present128</Algorithm>
8    <SampleType>Int8</SampleType>
9    <OutputName>Present128 (old implementation)</OutputName>
10   <Check>true</Check>
11 </InputArgsXML>
```

kde

- `SamplesCount` představuje počet samplů (vzorků) na jeden průběh (na jedno měření).
- `TracesCount` představuje celkový počet průběhů (počet měření).
- `TracesLimit` umožňuje omezit počet průběhů, které mají být zpracovány. Toto nastavení lze využít pro zrychlení výpočtu, ovšem za cenu toho, že nebudou brány v úvahu všechny průběhy a útok se tak nemusí podařit.
- `TracesFile` představuje název souboru s průběhy spotřeby.
- `CipherTextFile` představuje název souboru s šifrovými texty.

- `Algorithm` slouží pro výběr varianty algoritmu. Možné hodnoty jsou `Present80` a `Present128`.
- `SampleType` nastavuje typ vzorku (samplu). Možné hodnoty jsou `UInt8`, `Int8`, `Int16`.
- `OutputName` nastavuje název výstupního souboru.
- `Check` zapne nebo vypne kontrolu správnosti klíče. Pokud je kontrola zapnutá, program provede navíc ještě útok na 16. rudnovní klíč a jeho hodnotu porovná se skutečnou hodnotou podle získaného klíče. Jsou-li obě hodnoty stejné, je útok považován za úspěšný.

Výstupem programu je pak HTML soubor, který obsahuje výsledky dílčích útoků na rundovní klíče, grafy korelace a zpětný výpočet původního klíče z rundovních klíčů.

A.5 Výsledky

Na starší implementaci se podařilo tímto způsobem úspěšně zaútočit, přičemž k správnému odhadnutí všech tří potřebných rundovních klíčů stačí přibližně 1 500 průběhů.

Novou implementaci vytvořenou v rámci této práce se nepodařilo prolomit. Prolomení posledního rundovního klíče se podařilo až na 7 bitů, které byly odhadnuty opačně, než je jejich správná hodnota.

Odhadnutý rundovní klíč je 5D17 C6CE 211C DD7D, správná hodnota je 5D37 D6AE 211C DCF5. Liší se v bitech na pozicích 3, 7, 8, 37, 38, 44, 53.

Rozdíl mezi oběma implementacemi je v umístění S-Boxu. Zatímco v prvním případě se útočí na hodnotu uvnitř stavového registru, v druhém případě je stavový registr umístěn jinde a útočí se tak na hodnotu uvnitř kombinační logiky. Je tedy možné, že na větším počtu traců by byl útok úspěšný. Pro útok bylo použito 500 000 traců.

V obou případech je řeč o neochráněné verzi, tedy všechny ochrany byly vypnuté. Podrobnější výsledky obou útoků jsou na přiloženém DVD.

Příloha B

Přehled implementací

B.1 PRESENT

B.1.1 Základní informace

- Blokový šifrovací algoritmus, 64bitový blok
- 31 rund, po kterých následuje ještě jedno přičtení rundovního klíče (neboli 32 rund, přičemž v poslední rundě je vynechána substituční vrstva a permutace bitů)
- Runda se skládá z

  Přičtení rundovního klíče

  Substituční vrstvy (šířka S-Boxu je 4 bity)

  Permutace bitů v rámci bloku
- Délka klíče 80 nebo 128 bitů

B.1.2 Režimy implementace

- Basic (256 náhodných bitů pro rekonfiguraci)
- BasicPar (1 216 náhodných bitů pro rekonfiguraci)

B.2 SERPENT

B.2.1 Základní informace

- Blokový šifrovací algoritmus, 128bitový blok
- 32 rund, v poslední rundě je lineární transformace nahrazena přičtením dalšího rundovního klíče

- Runda se skládá z
  - Přičtení rundovního klíče
  - Substituční vrstvy (šířka S-Boxu je 4 bity)
  - Lineární transformace
- Délka klíče může být až 256 bitů (vytvořená implementace podporuje pouze délky dělitelné osmi)

B.2.2 Režimy implementace

- Basic (448 náhodných bitů pro rekonfiguraci)
- BasicPar (2 432 náhodných bitů pro rekonfiguraci)
- BasicParLite (1 408 náhodných bitů pro rekonfiguraci)

B.3 AES

B.3.1 Základní informace

- Blokový šifrovací algoritmus, 128bitový blok
- 10 rund (128bitový klíč), 12 rund (192bitový klíč) nebo 14 rund (256bitový klíč)
- Runda se skládá z
  - Substituční vrstvy (šířka S-Boxu je 8 bitů)
  - Operace ShiftRows (permutace)
  - Operace MixColumns (lineární transformace)
  - Přičtení rundovního klíče
- V poslední rundě se vynechává operace MixColumns.
- Před první rundou se navíc přičítá rundovní klíč
- Délka klíče 128, 192 nebo 256 bitů

B.3.2 Režimy implementace

- Basic (512 náhodných bitů pro rekonfiguraci)
- BasicPar (2 432 náhodných bitů pro rekonfiguraci)
- CompField (1 152 náhodných bitů pro rekonfiguraci)
- CompFieldPar (3 136 náhodných bitů pro rekonfiguraci)

- CompFieldAdv (3 424 náhodných bitů pro rekonfiguraci)
- CompFieldAdvRegs (3 424 náhodných bitů pro rekonfiguraci)

B.4 Nároky na FPGA

V následující tabulce jsou uvedeny nároky na FGPA při použití desky Sakura-G (Xilinx Spartan-6, XC6SLX75-2CSG484C)

Režim	Slice registers	Slice LUTs	Occupied slices
PresentBasic	1 124 (1 %)	1 322 (1 %)	517 (4 %)
PresentBasicPar	3 160 (3 %)	2 988 (6 %)	1 170 (10 %)
SerpentBasic	3 852 (4 %)	6 594 (14 %)	2 476 (21 %)
SerpentBasicParLite	6 308 (6 %)	9 408 (20 %)	3 588 (30 %)
AesBasic	2 044 (2 %)	5 257 (11 %)	1 832 (15 %)
AesBasicPar	6 121 (6 %)	9 928 (21 %)	3 459 (29 %)
AesCompField	4 596 (4 %)	6 396 (13 %)	2 566 (22 %)
AesCompField (s registry)	6 361 (6 %)	6 804 (14 %)	2 879 (24 %)
AesCompFieldPar	8 808 (9 %)	10 196 (21 %)	3 748 (32 %)
AesCompFieldPar (s registry)	10 573 (11 %)	10 231 (21 %)	4 352 (37 %)
AesCompFieldAdv	10 823 (11 %)	11 207 (24 %)	3 537 (30 %)
AesCompFieldAdv-Regs (s registry)	12 587 (13 %)	11 080 (23 %)	3 921 (33 %)
Celkem dostupné	93 296 (100 %)	46 648 (100 %)	11 662 (100 %)

Verze s registry jsou verze, do kterých jsou vložené registry pro eliminaci hazardů

B.4.1 Počet CFGLUTů

V následující tabulce jsou uvedeny počty CFGLUTů použitých v jednotlivých implementacích.

Implementace	Počet S-Boxů	CFGLUTů na S-Box	Celkem
PRESENT	16	8	128
SERPENT	32	8	256
AES	16	128	2048
AES (kompozitní těleso)	16	60	960
AES (kompozitní těleso, advanced)	16	74	1184

Příloha C

Seznam použitých zkratek

LUT Look-Up Table

CFGLUT Configurable Look-Up Table

AES Advanced Encryption Standard

FPGA Field Programmable Gate Array (programovatelné hradlové pole)

Příloha D

Obsah přiloženého DVD

- `Readme.txt` ..... stručný popis obsahu DVD
- `Bitstreams` ..... adresář obsahující některé předgenerované bitstreamy
- `GfReports` ..... informace o použitých tělesech a izomorfismech
- `Results` ..... výsledky měření ve formátu PDF
- `Tools` ..... podpůrné nástroje vyvinuté zjednodušení práce
  - `SwModel` ..... měřící program, SW implementace algoritmů a jiné
  - `AutoBuild` ..... nástroj AutoBuild
  - `GfCpp` ..... nástroj pro výpočty v kompozitním tělese
  - `DynReconfGen` ..... nástroj DynReconfGen
  - `DynReconfGenV3` ..... nástroj DynReconfGenV3
  - `PresentCPA` ..... nástroj PresentCPA
- `VhdlSrc` ..... VHDL zdrojové kódy
- `PresentV1` ..... implementace PRESENTu pomocí V1
- `Text` ..... Text práce ve formát PDF