FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Ochrana šifry PRESENT prostřednictvím falešných a vícenásobnýcch rund na FPGA
Student:	Petr Moucha
Vedoucí:	Dr.-Ing. Martin Novotný
Studijní program:	Informatika
Studijní obor:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Naimplementujte šifru PRESENT na FPGA jako iterativní obvod, ve kterém se každá runda provádí v jednom hodinovém taktu (šifrování trvá 32 hodinových taktů). Po konzultaci s vedoucím práce následně navržený obvod upravte tak, aby v jednom hodinovém taktu mohla probíhat žádná, jedna, dvě nebo tři rundy (0, 1, 2 nebo 3 rundy). Data, která řídí počet rund v jednotlivých taktech, budou generována externě a zasílána do obvodu současně s otevřeným textem (plaintext).
Naimplementované protiopatření by mělo ztížit analýzu prostřednictvím postranních kanálů (side-channel analysis), proto proveďte měření a analýzu jak původního (nezabezpečeného), tak upraveného (zabezpečeného) obvodu a porovnejte výsledky. Pro měření a analýzu použijte toolkit SICAK vyvinutý na katedře číslicového návrhu.
Cílovou platformou je deska Sakura-G, případně obdobná FPGA deska upravená pro měření průběhů spotřeby.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

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

V Praze dne 5. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Ochrana šifry PRESENT prostřednictvím falešných a vícenásobných rund na FPGA

Petr Moucha

Katedra číslicového návrhu
Vedoucí práce: Dr. Ing. Martin Novotný

7. ledna 2020

Poděkování

Jako první bych rád poděkoval svému vedoucímu, Martinovi Novotnému, který se mnou měl neskutečnou trpělivost a bez jehož pomoci by tato práce nemohla vzniknout. Dále děkuji své partnerce Barboře Sedláčkové a kamarádce Evě Pospíšilové nejen za jejich češtinářské rady a korekturu. V neposlední řadě můj vděk patří mým prarodičům Jaroslavě a Josefovi Kochovým, protože jen díky jejich podpoře jsem se dostal v životě a studiu vysoké školy tak daleko.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, 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, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2020 . . . . . .

Odkaz na tuto práci

Moucha, Petr. *Ochrana šifry PRESENT prostřednictvím falešných a vícenásobných rund na FPGA*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Práce pojednává o zabezpečení blokových šifer před útoky zaměřujícími se na informace o spotřebě během šifrování. Testovaným protiopatřením je technika vícenásobných a falešných rund, jejíž efektivnost nebyla doposud prokazatelně dokázána. Za účelem ověření předchozích výsledků a nalezení případných chyb v návrhu byla vytvořena takto zabezpečená verze šifry PRESENT v jazyce VHDL. Na této verzi byla poté provedena série testů, která odhalila průsak informací během začátku šifrování. Další úpravy obvodu poté vedly k progresivně lepším, avšak stále neuspokojivým výsledkům. Hlavním vylepšením bylo především náhodně opožděné nahrání platných vstupních dat do obvodu. Alternativně bylo také prozkoumáno, jaký vliv bude mít přidání dalších registrů, určených pro uložení výsledků falešných rund. Součástí řešení jsou i užitečné nástroje, které by měly usnadnit budoucí testování vícenásobných a falešných rund.

Klíčová slova blokové šifry, FPGA, analýza postranního kanálu, opatření proti rozdílové odběrové analýze, PRESENT, falešné rundy, t-test, SICAK

Abstract

This thesis focuses on securing block ciphers against side-channel attacks that can deduce the secret key from a device's power consumption. Multiple-and dummy-rounds countermeasure was tested because its effectiveness is yet to be proven. To verify previous results and find possible errors in countermeasure design, I created VHDL implementation of PRESENT cipher secured by dummy rounds. This version underwent a series of tests that revealed a leakage at the beginning of the cryptographic operation. Further improvements led to progressively better but still unsatisfying results. The most prominent feature was the insertion of a random number of dummy clock cycles before the first valid operation. I also examined the influence of added dummy registers used as storage for outputs of dummy rounds. Furthermore, as part of my effort, I developed tools useful for future testing of multiple- and dummy-rounds countermeasure.

Keywords block cipher, FPGA, secure hardware, side channel analysis, DPA countermeasure, PRESENT, dummy rounds, t-test, SICAK

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Úvod

V dnešním světě je technologie mnohem větší součástí běžného života, než si mnoho lidí uvědomuje, a tento trend nemá v plánu v nejbližší době ustupovat. Čím dál banálnější přístroje disponují chytrými funkcemi a možností připojení k internetu. Bohužel bezpečnostní riziko, které představují, je velice často přehlíženo. K citlivým informacím o uživateli mají v současnosti přístup nejen mobilní telefony, ale i fitness náramky, hodinky nebo zubní kartáčky.

Naneštěstí existují dobře dokumentované techniky, které umožňují při fyzickém přístupu k přístroji relativně snadno prolomit některá běžně používaná bezpečnostní opatření. Například spotřeba přístroje může obsahovat cenné informace o zpracovávaných datech, a pokud se při vývoji s touto skutečností nepočítá, bude pro znalého útočníka mnohdy velice snadné ji využít. Proto existuje poptávka po technologiích, které berou tato rizika v potaz a dokáží je eliminovat.

Jednou z cest může být i princip vícenásobných a falešných rund (neboli *dummy rounds*) popsaný Ing. Stanislavem Jeřábkem. Ten se zaměřuje na odstranění vztahu mezi spotřebou a tajným klíčem v šifře PRESENT, realizované hardwarově na FPGA. V tomto textu na Jeřábka navazuji a věnuji se nejen jeho původnímu návrhu, ale pokusím se na základě získaných postřehů navrhnout i různá vylepšení, která by mohla efektivnost tohoto bezpečnostního protiopatření posunout správným směrem.

Práce je strukturovaná následovně. Druhá kapitola obsahuje stručný úvod do problematiky spolu s informacemi o šifře PRESENT, způsobu testování a technikách zabezpečení obvodů. Vícenásobné a falešné rundy jsou představeny v kapitole Analýza, stejně jako všechna použitá vylepšení. V kapitole Realizace je poté popsána struktura řešení a jeho součástí. Nutná testování pro ověření funkčnosti obvodu je možné nalézt o kapitolu dále. Předposlední kapitola Měření prezentuje výsledky testů, které posuzují bezpečnost všech implementovaných variant protiopatření a jejichž celkové zhodnocení nabídnu v samotném závěru práce.

Kapitola 1

Cíl práce

Cílem této práce je především vytvořit šifru PRESENT [1] s využitím vícenásobných a falešných rund na FPGA. Ta musí obsahovat obvod pro výpočet až třech rund v jednom taktu, jak navrhl Jeřábek et al. [2, 3]. Podle pokynů vedoucího práce bude průběh šifrování řízen konfigurací zasílanou do FPGA po sériové lince. Zdrojový kód šifry bude napsán v jazyce VHDL a posléze implementován na desce Sakura-G [4].

Na této implementaci bude poté provedena série měření, jejímž produktem budou průběhy spotřeby zachycené během šifrování. Na nich se za pomoci nástroje SICAK [5] provede t-test [6], který určí míru zabezpečení testovaného hardwaru. Tyto výsledky porovnám s výsledky publikovanými v práci Ing. Jeřábka, na kterou navazuji.

K naplnění cílů bude zároveň potřeba vytvořit SICAK plugin kompatibilní se šifrou PRESENT a softwarový generátor konfigurací, který umožní jednoduchou kontrolu šifrování a vytváření různých scénářů pro měření.

Druhotným cílem bude na základě úspěšnosti základní verze v různých scénářích odhalit případné nedostatky a následně se je pokusit odstranit v dalších verzích. Provedu samozřejmě dodatečná měření, která umožní srovnání s původní, nemodifikovanou variantou.

Kapitola 2

Úvod do problematiky

2.1 Bloková šifra PRESENT

PRESENT je hardwarově nenáročnou *(lightweight)* substitučně-permutační sítí [7] určenou pro zařízení, která vyžadují rozumný stupeň ochrany, ale hardwarová implementace jiných běžně používaných šifer by měla neadekvátní prostorové a následně i cenové nároky.

Otevřený i šifrový text má délku 64 bitů a klíč může být dlouhý buď 80, nebo 128 bitů. Pro výpočet šifrového textu je třeba provést 32 rund, kde výsledkem každé rundy jsou aktualizované hodnoty klíče a vektoru. Průběh šifrování lze shrnout následujícím pseudokódem:

```
rundovniKlice[32] = generujKlice(klic);
vektor = otevrenyText;
for (int i = 1; i <= 31; i++)
{
    vektor = xor(vektor, rundovniKlic[i]);
    vektor = substituce(vektor);
    vektor = permutace(vektor);
}
sifrovyText = xor(vektor, rundovniKlic[32]);
```

Ukázka kódu 1: Funkce šifry PRESENT (přeloženo z [1])

Z ukázky 1 je vidět, že prvním vektorem je otevřený text a poslední runda je odlišná tím, že se v ní provádí pouze úvodní xor, na jehož vstup se přivádí vektor a prvních 64 bitů klíče. V ostatních rundách je výsledek xorování ještě dále zpracován šestnácti 4bitovými S-boxy, jejichž výstupy jsou poté po jednotlivých bitech přeskupeny a použity jako následující vektor.

Současně je potřeba vygenerovat sérii 32 rundovních klíčů. Tento krok lze provést i před šifrováním, jelikož všechny hodnoty závisí pouze na vstupním klíči a čísle rundy. Konkrétní proces pro vygenerování celé série se liší podle zvolené délky klíče. V této práci je použita verze se 128bitovým klíčem, jejíž implementaci lze nalézt, stejně jako specifika S-boxů, v kapitole 4. Pro získání informací o 80bitové verzi a dalších detailů odkazuji na [1].

Zmíněný článek [1] také obsahuje popis hardwarové implementace, která kromě multiplexerů disponuje 2 registry a obvodem pro výpočet rundy a následujícího klíče. Na začátku se do registrů nahrají vstupní hodnoty a na konci dají ty samé registry po vzájemném xoru šifrový text. Bude se tedy šifrovat po jedné rundě a výpočet bude trvat 32 hodinových taktů.

2.2 Testování odolnosti proti útoku postranním kanálem

Jelikož většina dnes běžně používaných šifer je z matematického hlediska bezpečná a současně dostupné prostředky neumožňují jejich prolomení útoky hrubou silou, pozornost útočníka se spíše zaměřuje na konkrétní implementaci. Ten si může práci výrazně zjednodušit, pokud využije informací z postranního kanálu – čili jiných výstupů, než které chtěl vývojář zpřístupnit uživateli. Takovou informací může být mimo jiné spotřeba zařízení během šifrování. Nalézt tajný klíč pouze ze spotřeby a otevřených textů umí například rozdílová odběrová analýza (*Differential Power Analysis*, DPA) [8] nebo *Correlation Power Analysis* (CPA) [9].

Pro posouzení odolnosti vůči těmto útokům lze využít metodiku popsanou v [6]. Spotřeba je zachycena do dvou množin, rozlišených podle toho, jestli se šifrovacímu zařízení poslal konstantní, nebo náhodný otevřený text, což je náhodně rozhodnuto během měření. Na dvou roztříděných záznamech spotřeby se poté provede nespecifický Welchův t-test, jehož výsledkem bude průběh t-hodnot. Ten informuje o tom, jak moc se rozptyly množin liší – čím vyšší hodnota v daném okamžiku, tím větší pravděpodobnost, že jsme schopni odlišit použití konstantních a náhodných vstupů. Pokud maximální absolutní hodnota přesáhne hranici 4.5 a toto maximum se při vícero měření objeví vždy ve stejném čase, není již zařízení považováno za bezpečné [10].

Pro zajištění správnosti testování je nutné zařídit, aby byly záznamy spotřeby dobře zarovnané. To znamená, že pokud se začátek šifrovaní nachází na určitém místě v jednom záznamu, musí stejné místo odpovídat stejné operaci i ve všech ostatních záznamech. V praxi tedy existuje požadavek na synchronizační impulz, který se aktivuje na začátku šifrovaní a umožní osciloskopu spustit záznam spotřeby vždy ve stejnou chvíli [10].

2.3 Protiopatření proti útokům postranními kanály

Byla již představena spousta protiopatření, která by měla útoky postranním kanálem znemožnit, avšak detailněji zde budou zmíněny jen ty nejvíce relevantní pro tuto práci.

2.3.1 Skrývání

Spotřeba v nechráněném obvodu do velké míry závisí na zpracovávaných datech. Jde ji například odhadnout pomocí Hammingových vah, kdy se předpokládá, že spotřeba roste s počtem bitů, jejichž hodnota je 1. „*Aby pro útočníka nebylo možné tuto závislost najít, musí být pozměněna základní charakteristika spotřeby. Návrhář může tuto charakteristiku změnit tak, že zařízení vymyslí takovým způsobem, aby každá operace vyžadovala buď přibližně konstantní, nebo víceméně náhodné množství energie. V obou případech je závislost dat na spotřebě významně snížena. Je ale důležité podotknout, že implementace chráněné pomocí skrývání zpracovávají stejné mezivýsledky jako nechráněné implementace*“ [11](překlad autora).

To znamená, že při skrývání většinou do nechráněného obvodu něco přidáme, aniž bychom změnili jeho funkčnost. Během šifrování je například možné zpracovávat náhodná data a vytvořit tak méně předvídatelný průběh spotřeby. Systematičtější postupy jako *Dual Rail Precharge* upravují obvod na úrovni logických buněk, ze kterých se skládá. Každá hodnota je reprezentována dvěma signály, které jsou před před zápisem platných dat nejprve oba vynulovány. Na jeden z drátů je poté přivedena platná hodnota, zatímco na ten druhý přijde její negace. Díky tomu bude počet přechodů z 0 na 1 a obráceně v celém obvodu konstantní [12].

Skrývat se dá i v čase. V jednodušších případech se charakteristika spotřeby nemění, ale pomocí náhodných zpoždění se zařídí, aby nebylo možné předvídat čas vykonání jednotlivých operací a zachycené záznamy spotřeby nebyly správně zarovnané. Pokud se ale záznamy dále zpracují a podaří se je správně zarovnat, lze provést útok stejně jako u nechráněného obvodu [13]. Tudíž použití náhodných zpoždění bez kombinace s jinými protiopatřeními nelze považovat za bezpečné.

Vylepšit skrývání v čase lze, pokud momenty, při kterých se pouze čeká na vykonání další operace, nebudou na záznamech rozeznatelné. Je tedy nutné zároveň skrývat spotřebu, typicky šifrováním náhodných dat [14]. Případně je možné místo zpožďování využít nezávislosti některých operací a ty provádět v náhodném pořadí [15].

2.3.2 Maskování

Další relativně účinné protiopatření spočívá v zamaskování platných hodnot před šifrováním, uložení zamaskovaného výsledku a následném odmaskování. Maska je náhodně generovaná pro každé šifrování, a spotřebu operací se zamaskovanými daty je tak obtížné předvídat [14]. Pro svou jednoduchost je často voleno aditivní maskování, které probíhá přičtením (xor) masky m k hodnotě a:

$$a_m = a \oplus m$$

Pro získání původní hodnoty a by stačilo masku m opětovně přičíst k a_m. V praxi se ale bude maska určená pro odmaskování lišit od m, jelikož po dalším zpracování již zamaskované hodnoty a_m vznikne nová hodnota, která bude modifikována operacemi, ze kterých se šifra skládá. Tyto modifikace se samozřejmě musí promítnout i na původní masce m.

Maskování je efektivním protiopatřením proti rozdílové analýze prvního řádu, ale je stále zranitelné vůči útokům vyšších řádů [16]. Útočník si totiž může v čase vybrat dva body t_1 a t_2, při nichž se prováděly operace na hodnotách a_m a b_m zamaskovaných stejnou maskou. Poté je možné odhadnout Hammingovu váhu $a \oplus b$, jelikož bude stejná jako pro $a_m \oplus b_m$. Tím je odstraněn vliv náhodné masky, na kterém bezpečnost maskování stojí. Ačkoliv jsou tyto útoky typicky náročnější a je potřeba znát více informací než při útocích prvního řádu (například zmíněné body t_1 a t_2), jsou i tak reálnou hrozbou [17, 18]. Proto je maskování vhodné kombinovat například se skrýváním [14].

2.3.3 Register precharge

Za každý registr s mezivýsledky je přidán další registr, do kterého se v první fázi nahrají náhodná data (tzv. *precharge*), která jsou poté šifrována. V druhé fázi jsou použitá náhodná data přepsána platnými z druhého registru, do kterého se nahrají nová náhodná data. Po zašifrování platných dat se mezivýsledek zapíše do druhého registru s novými náhodnými daty, která se umístí zpět do prvního registru. Poté se celý proces opakuje.

Prokládání platných operací náhodnými zajistí, že změny signálů budou nepředvídatelné. Nevýhodou je snížení rychlosti, s jakou je možné šifrovat, jelikož rozdělení každé operace na *precharge* a evaluaci zabere dvakrát tolik času [19].

2.4 Softwarová sada SICAK

Pro usnadnění testování vytvořil Ing. Petr Socha komplexní sadu nástrojů [5], která umí nejen řídit měření spotřeby, ale i analyzovat výsledky pomocí několika statistických metod, včetně výše popsaného t-testu. Celý proces je rozdělen do následujících modulů:

- ***meas: modul pro měření**, užitečný např. pro kontrolování kryptografického zařízení a osciloskopu*
- ***prep: modul pro pre-processing**, užitečný např. pro pre-processing záznamů spotřeby nebo pro vytváření predikcí o spotřebě na základě otevřeného/šifrového textu*
- ***stan: modul pro statistickou analýzu**, užitečný např. pro korelační útoky (CPA) nebo t-test*
- ***correv: modul pro vyhodnocení korelace**, užitečný pro algoritmické vyhodnocení CPA útoku*
- ***visu: modul pro vizualizaci**, užitečný pro vykreslení záznamů spotřeby, korelačních záznamů nebo t-hodnot* [20](překlad autora)

Ze všech zmíněných se t-testu týkají moduly *meas*, *stan* a *visu*. Funkcionalita jednotlivých modulů lze pozměnit nebo rozšířit pomocí pluginů. Například pro modul *meas* jsou přiloženy pluginy, které se zaměřují na měření šifry AES, a pro podporu šifer s jiným komunikačním protokolem je nutné existující plugin upravit [20]. SICAK je napsaný v jazyce C++ a využívá knihovny Qt, díky čemuž jej lze zkompilovat a používat pod Linuxem i OS Windows.

Kapitola 3

Analýza

V této kapitole lze nalézt popis technik použitých k zabezpečení blokové šifry PRESENT, spolu s detailními schématy dále upřesňujícími jejich funkci. U vlastních vylepšení a zvolených postupů je zmíněn myšlenkový proces, který objasňuje jejich záměr a předpokládaný vliv na zabezpečení obvodu.

3.1 Původní dummy rounds bez modifikací

První verze vychází přímo z návrhu Ing. Stanislava Jeřábka a jejím cílem je ověřit výsledky originálních měření, popsaných v jeho článku [2]. Datová cesta sestává ne z jedné, ale tří rund, jež jsou zapojeny do série, jak je zobrazeno na obrázku 3.1. Šifrování probíhá vždy v 16 taktech a v jednom taktu lze spočítat jednu, dvě nebo tři rundy. Cílem tohoto přístupu ale nebylo zrychlení šifrování. Idea je totiž taková, že v aktuálním taktu bude počet zpracovaných rund variabilní a bude záviset na náhodné konfiguraci, která bude uložena v řadiči. V navazujícím článku [3] Jeřábek et al. navrhují modifikaci, která umožňuje také prázdné takty (0 rund) a variabilní počet taktů na jedno šifrování. V jednom taktu půjde tedy rozhodnout, jestli se spočte jedna, dvě, tři nebo žádná rudna, případně jestli se šifrování ukončí.

Je zřejmé, že v některých taktech nebude využit všechen hardware, který je k dispozici. Například pokud bude hodnota konfigurace v daném taktu 1, výstup posledních dvou rund se zahodí a uložen bude pouze výstup té první. Aby byla práce útočníkovi ještě více ztížena, do nevyužitých rund se místo platného výstupu předchozí rundy jako vstup použijí pseudonáhodná data. Tento krok je zásadní, jelikož bez prokládání náhodnými daty by na začátku šifrování (tj. v prvním taktu) byla pro stejný vstup spotřeba kombinační logiky vždy stejná, a to nezávisle na konfiguraci – ta by ovlivnila pouze spotřebu registrů na základě toho, která ze tří rund by byla uložena. Nadbytečné rundy, které zpracovávají pseudonáhodné hodnoty, budou v textu nadále označovány jako **falešné**.

Bezpečnost falešných a vícenásobných rund (neboli *dummy rounds*) stojí tedy na dvou principech - skrývání v čase a skrývání ve spotřebě. Útočník nejen že neví, kdy se jaká runda provede, ale zároveň nelze spotřebu odhadnout, jelikož závisí i na náhodných datech zpracovávaných ve falešných rundách.

3.2 Konfigurace

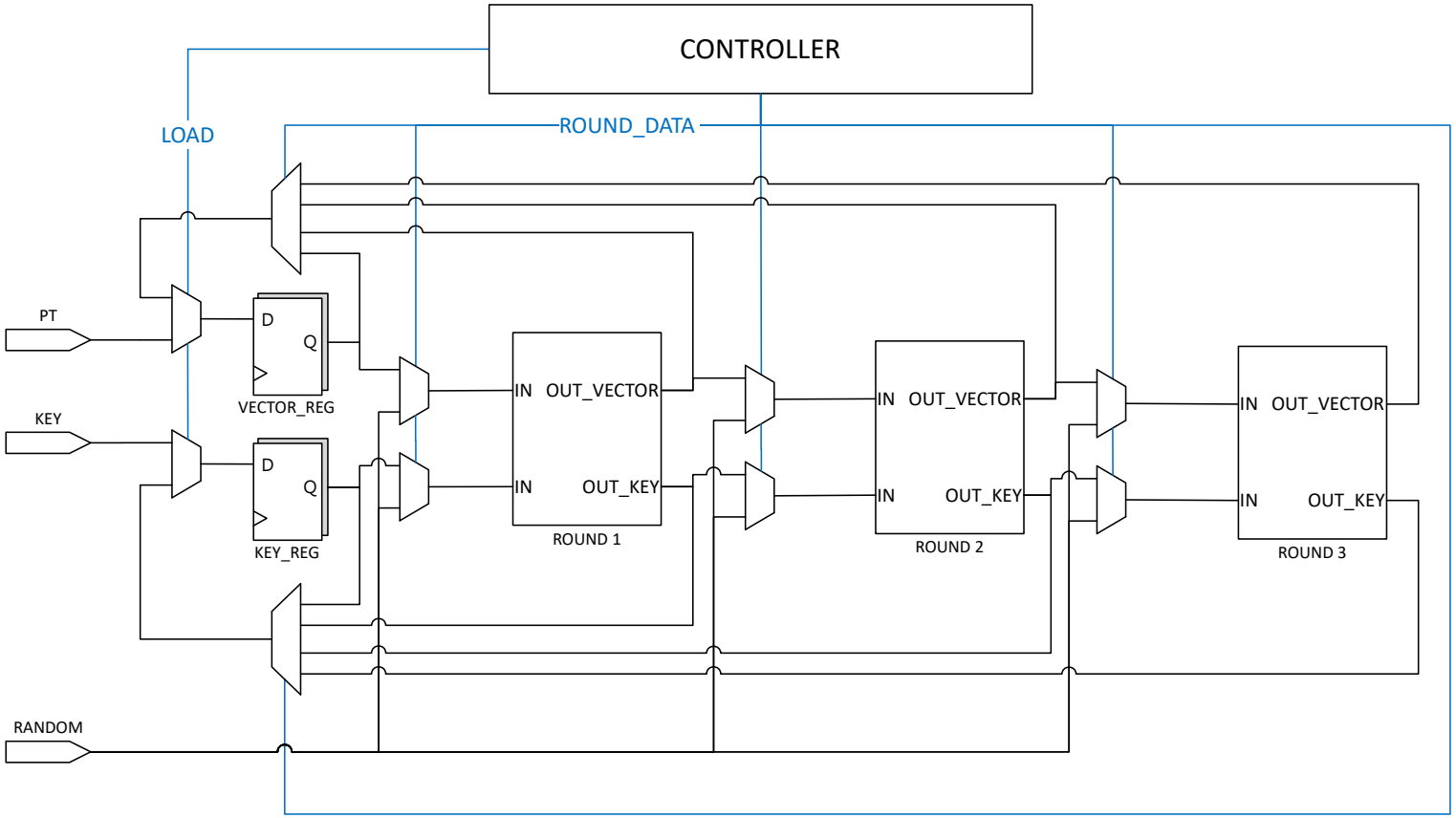
Průběh šifrování řídí 128bitová konfigurace, kde každý *nibble* (půlslabika, 4 bity) určuje počet platných rund v jednom ze 32 taktů. Hardware nijak konfiguraci nevytváří ani nekontroluje, pouze ji přijme z externího zdroje, například sériové linky. Platné hodnoty na jednotlivých 4bitových pozicích v konfiguraci definuje tabulka 3.1.

Hodnota	Nastavení
`0x1`	jedna platná runda
`0x2`	dvě platné rundy
`0x3`	tři platné rundy
`0xF`	**zarážka** ukončující šifrování

Tabulka 3.1: Platné hodnoty pozic v konfiguraci

Validní konfigurací je například `0x33332222211111111110F00000000000`. Pozice jsou čteny zleva doprava a šifrování v tom případě vypadá následovně:

1. V každém z prvních čtyř taktů se provedou tři platné rundy, celkem tedy 12 rund.
2. Poté se v pátém až devátém taktu provedou vždy dvě rundy a celkový počet zpracovaných rund se zvýší na 22.
3. Posledních 10 rund je po jedné rundě spočteno v taktech 10 až 19.
4. V dvacátém taktu se již žádná další runda neprovede, ale přesto není šifrování zatím ukončeno.
5. Ukončení nastane až o takt dál, po přečtení zarážky. Teprve tehdy je zaručeně dostupný platný výsledek.



Obrázek 3.1: Schéma šifry PRESENT s využitím *dummy rounds*

3.3 Generování pseudonáhodných dat

Pro generování pseudonáhodných dat, potřebných pro vstupy falešných rund, bude sloužit 64bitový lineární zpětnovazební registr [21]. Zvolil jsem Fibonaccioho způsob implementace, který nejen že se jevil nepatrně jednodušší na popsání ve VHDL, ale především testy naznačují, že se jedná o bezpečnější variantu z hlediska útoků postranním kanálem [22]. Alternativou je Galoisův LFSR, který je obecně považován za efektivnější, jelikož lze aritmetické operace provádět paralelně a nedochází k tak velkému zpoždění v kombinační cestě [23]. Protože ale budu operovat s malým kmitočtem a v návrhu budou ještě kritičtější cesty, není důvod k obavám.

Při délce LFSR 64 bitů budou pozice, ze kterých se počítá hodnota nového bitu po posunutí, určovány polynomem:

$$x^{64} + x^{63} + x^{61} + x^{60} + 1$$

Jeden takový registr bez problémů stačí na generování pseudonáhodných otevřených textů, které jsou také dlouhé 64 bitů. Problém nastává, pokud by se ve falešných rundách měl použít i náhodný klíč, protože pro něj nebude využití LFSR tak přímočaré. Zaprvé se délky registrů neshodují a zadruhé je potřeba zajistit, aby mezi otevřeným textem a klíčem neexistoval vztah, který by způsobil předvídatelnost ve spotřebě falešných rund. V t-testu se ale vliv klíče tolik nezkoumá a počítá se s tím, že bude konstantní po dobu celého měření. Přesto jsem se i touto variantou zabýval. I když se nabízelo jednoduše přidat druhý 128bitový LFSR, používal jsem ho zřídka a raději jsem se pro generování náhodných klíčů snažil efektivně využít výstupy falešných rund, pokud to bylo možné. V opačném případě určoval bity tento polynom:

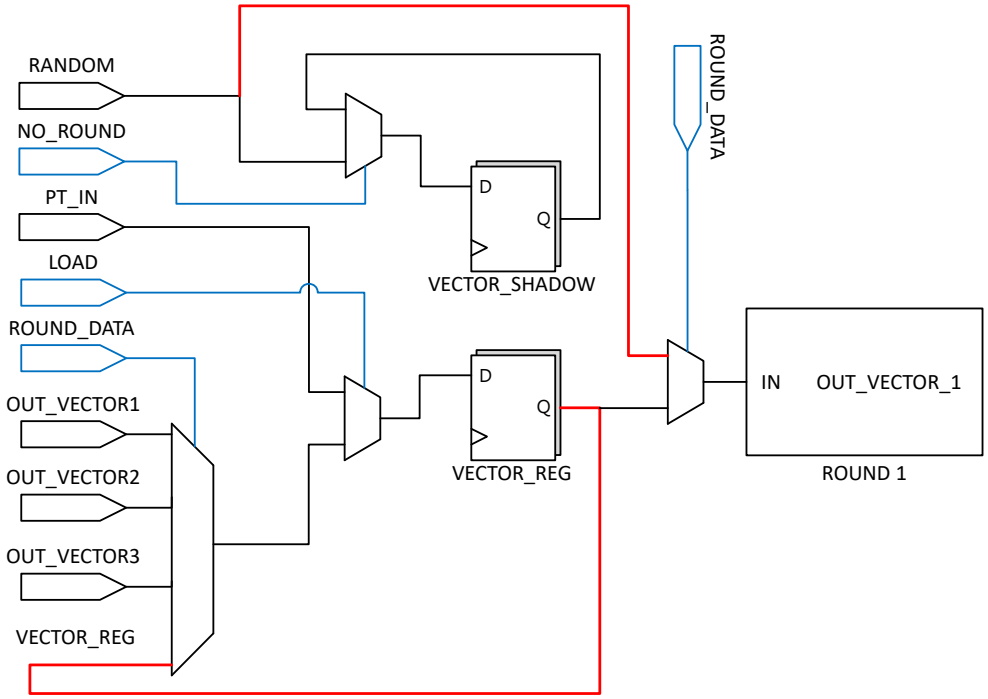
$$x^{128} + x^{126} + x^{101} + x^{99} + 1$$

3.4 Dummy rounds s falešnými registry

Verze testované mým předchůdcem v jeho článku [2] nepovolovaly **prázdné** takty – neboli momenty, kdy hodnota aktuální pozice v konfiguraci je 0 a všechny rundy jsou falešné. Datová cesta na to ani nebyla uzpůsobená a v každém taktu se musel zapsat výstup jedné z rund. Aby však toto protiopatření naplnilo svůj potenciál, je nutné prázdné takty povolit, jelikož bez nich jsou příliš krátká nebo dlouhá šifrování předvídatelná [3]. Například pokud bude šifrování probíhat 32 taktů, musí se v každém taktu provést přesně jedna runda. U prázdných taktů ale existuje riziko, že je útočník snadno rozpozná.

V práci [2] se předpokládá, že datová cesta tak, jak je popsaná na obrázku 3.1, by měla prázdné takty dostatečně skrývat. Předpoklad byl takový, že bude stačit povolit náhodný vstup už pro první rundu a v multiplexeru,

který řídí vstup registrů, povolit opětovné zapsání současné hodnoty. Na detailnějším obrázku 3.2 jsou zmíněné úpravy zvýrazněné červenou barvou.



Obrázek 3.2: Schéma ukládání vektorů s falešným registrem

Při měřeních se však ukázalo, že tento návrh je problematický. Jelikož v prázdných taktech nebudou k dispozici žádné platné mezivýsledky k uložení, bez zapsání nových hodnot do registrů nastane pokles ve spotřebě, který dokáží skrýt jen velké změny v kombinační logice. Bohužel pro skrytí prázdných taktů ne vždy stačí, aby byly všechny rundy falešné. Situaci je naštěstí možné zlepšit, pokud se kromě počítání falešných rund budou jejich výsledky i někam ukládat.

Pro tento účel jsem přidal dodatečné registry, do kterých se zapisují falešné výsledky jenom v případech, kdy zrovna nejsou ukládána žádná platná data. Na obrázku 3.2, který popisuje aktualizované ukládání vektorů (schéma bude vypadat obdobně i pro ukládání klíčů), je falešný registr označen jako VECTOR_SHADOW. Ten doplňuje platný registr VECTOR_REG. V každém taktu se zapíše jen a pouze do jednoho z dvojice registrů, aby se docílilo vyrovnání spotřeby v obvodu. V **aktivních** taktech, kdy je alespoň jedna runda platná, se falešné výsledky neukládají.

3.5 Dummy rounds s prohazováním

Samotné skrývání prázdných taktů nijak neovlivní míru zabezpečení v taktech aktivních. Problematickým případem na druhé straně spektra budou totiž takty se všemi rundami platnými. Náhoda se v nich vůbec nevyužije a i zapsaná data budou pro stejné vstupní hodnoty shodná. Tato situace je nejkritičtější na počátku šifrování, kdy útočník nepotřebuje mít informaci o celé konfiguraci a aktuálním vektorem je známý otevřený text. Bohužel ani při zbylých možnostech (1 nebo 2 platné rundy) nebudou falešné rundy samy o sobě poskytovat dostatečnou ochranu, pokud se vždy zapíší ty samé mezivýsledky.

Dvojnásobné množství registrů je tedy třeba využívat efektivněji a přidat prvek náhody do každého zápisu dat. Úmyslem je umožnit zapisování do obou registrů současně, přičemž jejich obsah se v každém taktu musí přepsat, a to aniž by se přemazala platná data, která je třeba během prázdných taktů uchovat.

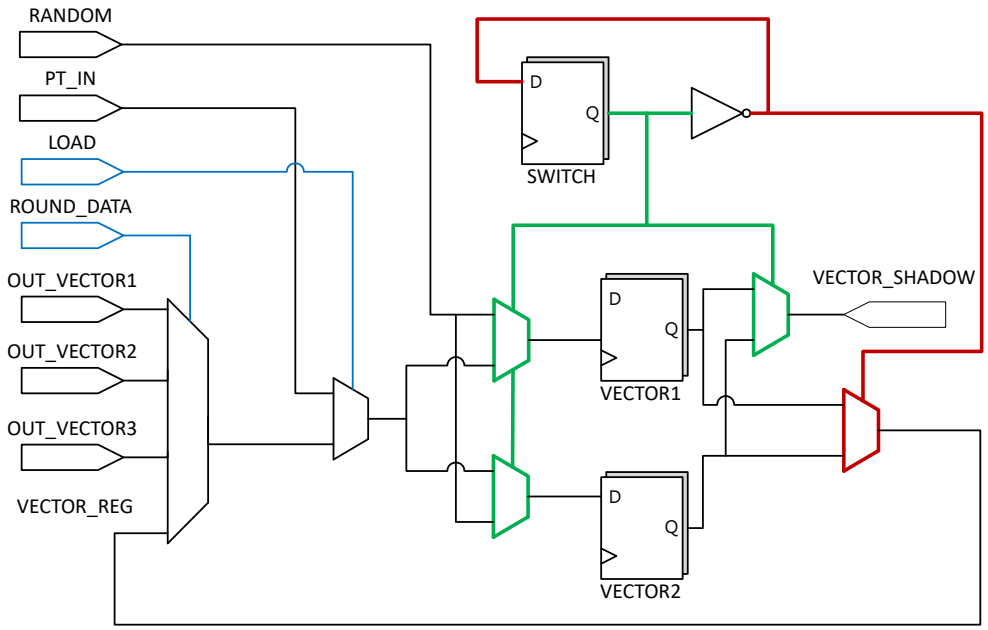
Všechny tyto požadavky splňují úpravy popsané na obrázku 3.3. Princip je jednoduchý – místo zapisování pouze platných nebo falešných výsledků zapisuji obojí, ale v každém taktu střídám cílové registry. Ty jsou si rovnocenné a nejsou nadále deterministicky označeny jako REG a SHADOW. Který je který, je při čtení i zápisu určeno alternujícím signálem SWITCH podle tabulky 3.2. V každém taktu se tak přepíší náhodná data platnými a platná náhodnými, což by mělo v ideálním případě kompletně eliminovat vliv registrů na zabezpečení. Oproti podobně fungujícímu *register precharge* nepočítám s *precharge* fází, a není tak nutné dvojnásobně prodlužovat dobu šifrování.

SWITCH	1	0
R1	SHADOW	REG
R2	REG	SHADOW

Tabulka 3.2: Umístění dat v závislosti na SWITCH

Tato verze je nejlepší, co se možností pro generování náhodných dat týče. V každém taktu se totiž objeví nová náhodná hodnota nejen v LFSR, ale i ve falešném registru. Jedná se o hodnotu, která je produktem S-boxů ve falešných rundách, a tím pádem jsou změny po jednom taktu výraznější než u LFSR. Z toho důvodu je také použita na vstupech falešných rund. To jinými slovy znamená, že v každém taktu pokračují falešné rundy šifrováním předchozího falešného výsledku.

Výjimkou a komplikací je však situace, kdy jsou všechny rundy platné a novou náhodnou hodnotu je nutné získat jiným způsobem. Tehdy není jiná možnost než při zápisu do falešného registru využít hodnoty v LFSR. Jelikož se ale v extrémních případech může stát, že nějakou falešnou rundu budu počítat vždy, tak hrozí malé nebezpečí, že celá posloupnost falešných rund bude vycházet z výchozí hodnoty falešného registru a nikdy nedojde k využití



Obrázek 3.3: Schéma ukládání vektorů s prohazováním registrů

LFSR, na kterém celá náhodnost stojí. Finální implementace by takové situaci ale měla jednoduše zabránit, například zapsáním hodnoty odvozené z LFSR do falešného registru před začátkem nebo po skončení každého šifrování.

3.6 Další možnosti optimalizace

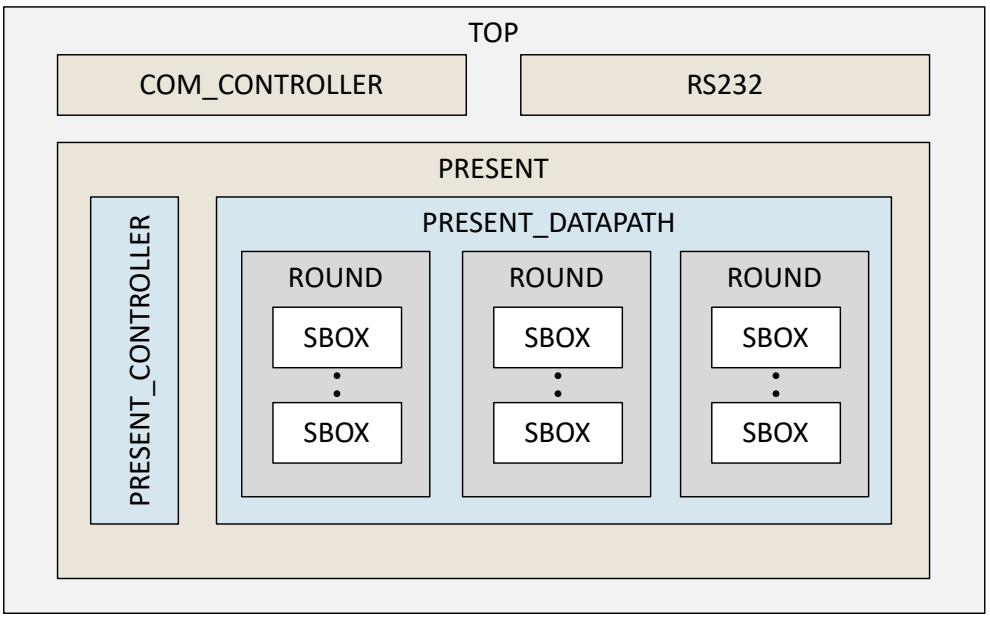
Protože útoky postranním kanálem se většinou zaměřují na začátek nebo konec šifrování, je důležité těmto místům věnovat zvýšenou pozornost. Zlepšit situaci na začátku šifrování je možné, pokud se do interních registrů nahraje otevřený text a klíč, až když budou potřeba. Konfigurace totiž může začínat sérií nul, a je tedy zbytečné ukládat platná data už během těchto úvodních prázdných taktů. Pokud bude zaručeno, že se data na vstupech v době mezi spuštěním šifrování a prvním aktivním taktem nezmění, nebude obtížné toto vylepšení implementovat [3].

Konec šifrování je neméně zajímavý. Zde se nabízí nechat šifrový text uložený v registrech pouze po dobu nutnou k jeho zpracování ve vnější části obvodu a poté jej přepsat náhodnými daty. Je ale nutné si uvědomit, že šifrování ukončuje zarážka v konfiguraci, a pokud bude mezi posledním aktivním taktem a zarážkou několik prázdných taktů, není možné jej smazat ihned. V každém případě ale další šifrování už bude na tom předchozím kompletně nezávislé a t-test by měl dopadnou lépe na obou koncích šifrování.

Kapitola 4

Realizace

V této kapitole je popsán způsob implementace celého testovaného systému v jazyce VHDL. Obecnou strukturu nabídne již obrázek 4.1. Jako první jsou detailně rozebrány všechny komponenty, ze kterých se skládá bloková šifra PRESENT. Následuje popis komunikačního protokolu a způsob jeho realizace. Zmíním i pomocný software, který jsem musel pro účely této práce vytvořit, a vysvětlím, jak jej použít.



Obrázek 4.1: Blokové schéma znázorňující hierarchii VHDL kódu

4.1 PRESENT

```
entity PRESENT is
    Port ( PT : in std_logic_vector(63 downto 0);
           KEY : in std_logic_vector(127 downto 0);
           CONF : in std_logic_vector(127 downto 0);
           START : in std_logic;
           DONE : out std_logic;
           SYNC : out std_logic;
           CLK : in std_logic;
           RESET : in std_logic;
           CT : out std_logic_vector(63 downto 0));
end PRESENT;
```

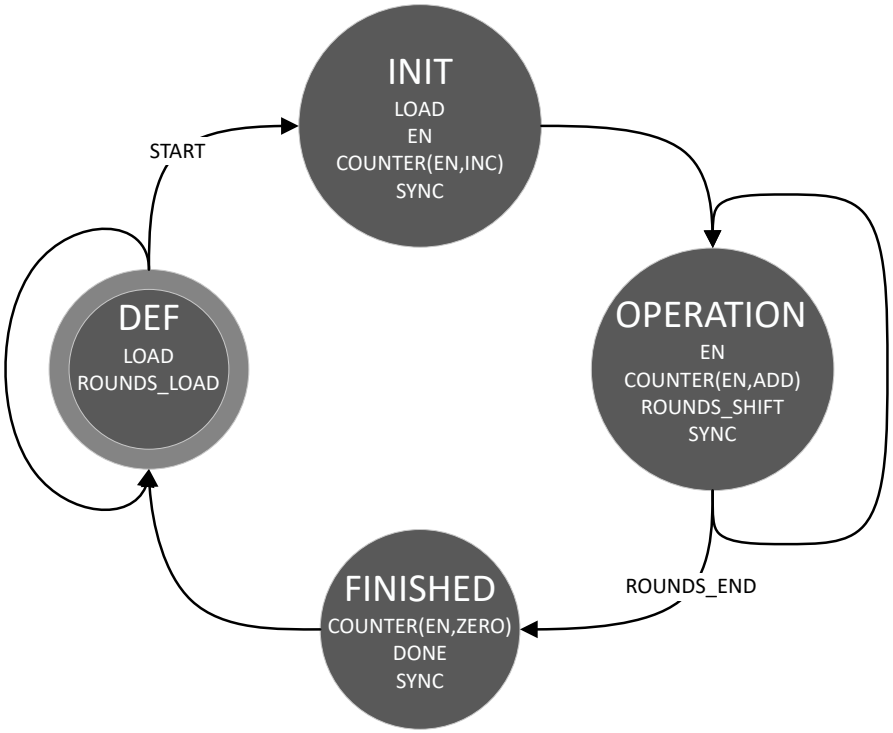
Ukázka kódu 2: Rozhraní šifry PRESENT

Implementace PRESENTu je rozdělena na dvě části – datovou a řídící. Ty poté spojuje rozhraní v ukázce kódu 2. Vstupní porty `PT`, `KEY`, `RNG` slouží k nastavení otevřeného textu, klíče a (náhodné) konfigurace. Šifrování se spustí nastavením signálu `START` a signál `DONE` po jeho dokončení informuje o dostupnosti šifrového textu na výstupním portu `CT`, který je zaručeně korektní po dobu jednoho hodinového taktu. `SYNC` je aktivní během šifrování a je zde kvůli synchronizaci měření jednotlivých průběhů spotřeby.

4.1.1 Řadič

Řadič má podobu konečného automatu typu Moore se čtyřmi stavy. Ve výchozím stavu `DEF` je pouze povolen zápis konfigurace ze vstupu do interního registru. Po spuštění šifrování přes vstupní signál `START` a přechodu do stavu `INIT` je povolen zápis do interních registrů, pro které se díky signálu `LOAD` jako zdroj vybere vstupní otevřený text a klíč. Zároveň je vypnuto načítání konfigurace a čítač, který zaznamenává počet zpracovaných rund, je inkrementováním nastaven na hodnotu 1. Následuje přechod do `OPERATION`, kdy se zapne posuv registru s konfigurací, ve kterém nejvýznamnější 4 bity určují počet platných rund v aktuálním taktu. Ve stavu řadič setrvá do doby, než se při posouvání konfigurace narazí na zarážku (viz. tabulka 4.2). Závěrečným stavem je `FINISHED`, jehož hlavním účelem je signalizovat pomocí `DONE` konec šifrování.

Registr konfigurace je součástí řadiče a pojme 132 bitů, neboli 33 *nibblů*. Uživatel jich však určuje pouze prvních 32, do posledního *nibblu* je vždy nahrána zarážka. Pokud ji tedy uživatel sám v konfiguraci nedefinuje, šifrování bude vždy implicitně ukončeno po 32 taktech.



Obrázek 4.2: Stavový diagram první verze řadiče

Pro ukončování by šlo také využít signálu `IS_LAST`, který je aktivní, pokud se v aktuálním taktu provádí poslední 32. runda. Šifrování by tak probíhalo vždy nejkratší potřebnou dobu. Pro účely této práce ale bylo přednější umožnit uživateli co největší kontrolu, kterou poskytuje ukončování pomocí zarážky. Přesto není `IS_LAST` zbytečný a je nutný pro správný výpočet poslední rundy, která se od těch předchozích liší a skládá se pouze z úvodního xoru, tj. nezahrnuje S-boxy (viz Ukázka kódu1). Pro získání výsledku je tedy potřeba provést pouze 31 plnohodnotných rund, a tak tento signál slouží ke korekci konfigurace, která v součtu obsahuje rund 32. Z toho také vyplývá, že je dosaženo platného výsledku, i pokud bude součet v konfiguraci 31.

Poslední runda bude nutně součástí posledního aktivního taktu. Ten lze poznat podle toho, že v taktu, který po něm nasleduje, vždy dojde k přetečení čítače `COUNTER` do počáteční hodnoty 1. Je však třeba vyloučit situace, kdy se na začátku šifrování zpracovávají prázdné takty a čítač se nijak v dalším taktu nezmění. V obou případech bude `NEXT_COUNT` nastaven na počáteční hodnotu. Poslední runda je ale specifická tím, že před přičtením `NEXT_COUNT` bude v čítači `COUNTER` jiná hodnota než ta počáteční. Proces, který na základě těchto skutečností správně nastavuje signál `IS_LAST`, je na ukázce 3. Zde je také vidět, že se `IS_LAST` jednoduše odečítá od hodnoty v konfiguraci, a tím je provedena požadovaná korekce.

```
alias CURRENT_ROUND is ROUNDS(131 downto 128);
NEXT_COUNT <= COUNTER + CURRENT_ROUND;
ROUNDS_END <= '1' when CURRENT_ROUND = x"F" else '0';
ROUND_DATA <= CURRENT_ROUND - IS_LAST;
ROUND_COUNT <= COUNTER;

IS_LAST_PROC : process (NEXT_COUNT, COUNTER)
begin
    if NEXT_COUNT = 1 and COUNTER /= 1 then
        IS_LAST <= '1';
    else
        IS_LAST <= '0';
    end if;
end process IS_LAST_PROC;
```

Ukázka kódu 3: Nastavení důležitých signálů v řadiči

Jelikož mým prvním cílem bylo vytvořit fungující PRESENT bez jakýchkoliv protiopatření a periferií, dbal jsem při vymýšlení řadiče na jiné priority než v pozdější fázi práce, kdy došlo na měření. Předně jsem se snažil o to, aby šlo hotovou šifru integrovat do jakéhokoliv systému a nebyla závislá na vnější části obvodu. Tato motivace se nejvíce projevila u stavu `INIT`, ve kterém pouze zbytečně duplikuji data ze vstupu. Tato činnost navíc negativně ovlivňuje bezpečnost všech testovaných datových cest.

Tudíž vznikla vylepšená verze, která nedostatky odstraňuje a přidává podporu pro optimalizace popsané v sekci 3.6. Stavový diagram vylepšeného řadiče je na obrázku 4.3. Změny jsou následující:

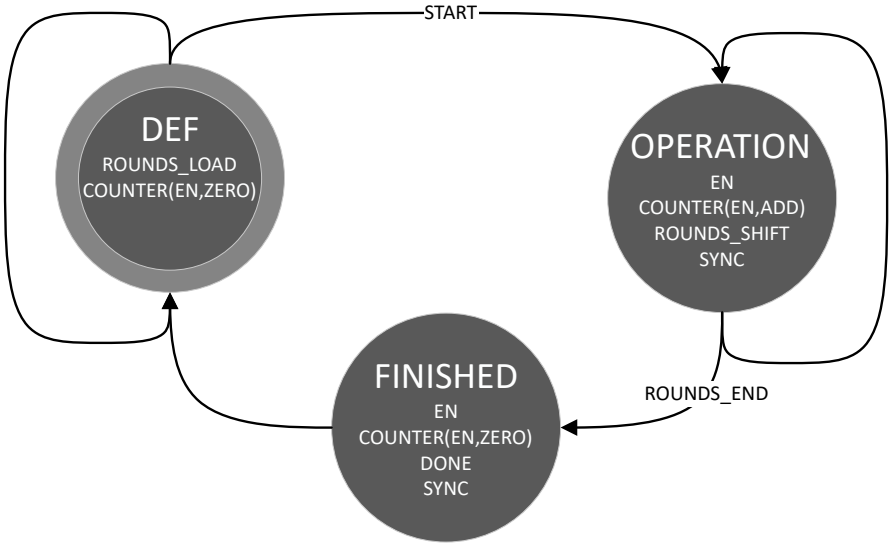
- Odstranění stavu `INIT`, z výchozího stavu nyní existuje přechod přímo do `OPERATION`.
- Do čítače se v módu `ZERO` nahraje 1 místo 0. Před přechodem do stavu `OPERATION` v něm tedy bude správná hodnota a není potřeba jej dále inkrementovat.
- Odstranění explicitního signálu `LOAD`. Načtení vstupů proběhne automaticky v prvním aktivním taktu. Aby toto fungovalo, bylo nutné upravit datovou cestu tak, aby se otevřený text s klíčem přivedly na vstup první rundy, místo na vstup interních registrů.
  - Aktivní takt lze triviálně vyčíst z `ROUND_DATA`. První takt je specifický tím, že zároveň platí `ROUND_COUNT="0b00001"`.
  - Jelikož je po dokončení všech 32 rund hodnota čítače opět 1, budou vstupní data opět načtena, pokud se po spočtení šifrového textu

EN	Povolení zápisu do registrů
LOAD	Pokud je signál aktivní, je povoleno nahrávání otevřeného textu a klíče. V opačném případě je na vstup registrů přiveden výstup jedné ze tří rund.
ROUNDS_LOAD	Pokud je signál aktivní, je povoleno nahrávání konfigurace. V opačném případě je konfigurace řízena signálem `ROUNDS_SHIFT`.
ROUNDS_SHIFT	Pokud je signál aktivní, registr konfigurace se cyklicky posune vlevo o 4 bity. Tím se nastaví počet platných rund pro příslušný takt.
ROUND_DATA	Signál, který řídí počet platných rund v aktuálním taktu. Povolené hodnoty jsou: `0x0, 0x1, 0x2, 0x3` — číslo označuje počet platných rund v aktuálním taktu, `0xF` — zarážka, která ukončuje šifrování.
ROUND_COUNT	Hodnota interního čítače, která indikuje počet doposud zpracovaných rund. Je předávána datové cestě kvůli výpočtu rundovních klíčů
COUNTER_EN	Povolení zápisu do čítače rund (`COUNTER`).
COUNTER_MOD	Pokud je do čítače povolen zápis, nová hodnota se získá na základě nastaveného módu. Ten může být: `ZERO` — čítač se vynuluje, `INC` — čítač se inkrementuje, `ADD_DATA` — do čítače se zapíše `NEXT_COUNT`, což je aktualizovaná hodnota po přičtení `ROUND_DATA`.
DONE	Signalizuje konec šifrování. Řadič, který ovládá komunikaci, na jeho základě spustí odesílání šifrového textu po sériové lince.
SYNC	Synchronizační impulz, použitý při měření spotřeby.

Tabulka 4.2: Seznam výstupních řídících signálů

provedou další aktivní takty. Taková konfigurace by ale nebyla validní ani v nevylepšené verzi.

- `DONE` slouží navíc k přemazání platného výsledku náhodnými daty a uložení nové náhodné hodnoty do falešného registru.
- Jelikož správná konfigurace musí být uložena v registru konfigurace o takt předtím, než se budou ukládat výsledky prvního aktivního taktu, je signál `ROUND_LOAD` nutně aktivní ve výchozím stavu. Ačkoliv se v tomto ohledu nic nezměnilo, v nevylepšené verzi šlo načítání provést až ve stavu `INIT`. Pokud by se zápis náhodné konfigurace provedl spolu se zápisem otevřeného textu a tajného klíče, byla by zranitelnost během `INIT` pravděpodobně nižší.



Obrázek 4.3: Stavový diagram vylepšeného řadiče

4.1.2 Datová cesta

Datová cesta se skládá celkem ze tří entit, kde tou hlavní je `PRESENT_DATAPATH`. Jelikož bylo vytvořeno a otestováno více revizí, jejichž funkcionalita odpovídá schématům popsaným v kapitole 3 a které se ještě dále rozlišují použitou verzí řadiče a dalšími detaily, není rozumné pro každou z nich uvádět v textu zdrojové kódy. Raději popíšu obecnou strukturu, kterou mají společnou.

Entita obsahuje v každé verzi alespoň dva registry pro vektor a klíč – `VECTOR_REG` a `KEY_REG`. Později přidané falešné registry jsou pojmenovány jako `VECTOR_SHADOW` a `KEY_SHADOW`. Funkci každého z registrů popisuje jeden proces. Situace je o něco komplikovanější ve verzích s prohazováním, kde jsou navíc pro vektory i klíče přidány další dva procesy. První rozhoduje o další

hodnotě uložené do příslušného platného registru. Druhý proces pak rozlišuje na základě hodnoty v registru `SWITCH`, který z registrů je platný a který falešný.

Dále je přítomný 64bitový `LFSR_REG`, který v každém taktu obsahuje novou pseudonáhodnou hodnotu. Případně se zde může nacházet i 128bitový `LFSR128_REG` se stejnou funkcí. Aktualizaci obou registrů provádí proces `LFSR`.

Samozřejmostí jsou také alespoň tři instance entity `ROUND`, jejichž vstupy řídí proces `RNG_INTERLACING`. Ten podle informací v `ROUND_DATA` přeruší řetězové zapojení rund a přivede na vstup jedné z nich náhodná data, namísto výstupu předchozí rundy. K přerušení nedojde pokud budou všechny rundy platné. Při použití vylepšeného řadiče se v tomto procesu také objeví načtení vstupních dat do první rundy.

Posledním zásadním prvkem je nastavení výstupního portu `CT`. Přiřazena je na něj hodnota `KEY_REG(127 downto 64) xor VECTOR_REG`. Pokud se v platných registrech nachází výsledky po spočtení 31 rund, tato operace vyprodukuje platný šifrový text (viz. Ukázka kódu 1).

4.1.2.1 ROUND

Úkolem entity je provést všechny potřebné operace pro výpočet jedné rundy. Ty se dají rozdělit do třech částí, které jsou detailně rozepsány v ukázce kódu 4:

- *aktualizace počtu rund* — hodnota vstupního portu `ROUND_COUNT` je inkrementována o jedna a nastavena na výstupu.

- *vytvoření následujícího rundovního klíče* — jako první je současný klíč cyklicky rotován o 67 bitů doprava. Nejvýznamnějších 8 bitů se poté nahradí pomocí dvou S-boxů a bity 66 až 62 se vyxorují s `ROUND_COUNT`.

- *vytvoření následujícího vektoru* — nejprve se vyxoruje nejvýznamnějších 64 bitů klíče s aktuálním vektorem. Výsledek je přiveden na vstup 16 S-boxů, které generuje `SBOX_GEN`. Jejich výstupy poté zpracuje proces `PERM_PROC` provedením rovnoměrné permutace, která lze jednoduše realizovat pomocí příkazu `for`. Permutovaná hodnota se rovnou nastavuje na výstupním portu `NEXT_VECTOR`.

4.1.2.2 SBOX

Činnost této entity je přímočará stejně jako její rozhraní, které se skládá ze dvou portů – čtyřbitového vstupu a výstupu. Výstupní hodnoty definuje na základě těch vstupních proces `SBOX_PROC`, který je možné vidět na ukázce 5.

```
architecture ROUND_BODY of ROUND is
    --deklarace signálů vynechány
begin
    --aktualizování počtu rund--------------------------
    NEXT_ROUND_COUNT <= ROUND_COUNT + 1;
    --rutina pro aktualizaci 128bitového klíče----------
    KEY_MANIP <= KEY(66 downto 0) & KEY(127 downto 67);
    SBOX_KEY1 : SBOX port map
                  (
                      INPUT => KEY_MANIP(127 downto 124),
                      OUTPUT => NEXT_KEY(127 downto 124)
                  );
    SBOX_KEY2 : SBOX port map
                  (
                      INPUT => KEY_MANIP(123 downto 120),
                      OUTPUT => NEXT_KEY(123 downto 120)
                  );
    NEXT_KEY(66 downto 62) <= KEY_MANIP(66 downto 62) xor ROUND_COUNT;
    NEXT_KEY(119 downto 67) <= KEY_MANIP(119 downto 67);
    NEXT_KEY(61 downto 0) <= KEY_MANIP(61 downto 0);
    --rutina pro aktualizaci vektoru--------------------
    SBOX_IN <= KEY(127 downto 64) xor VECTOR;

    SBOX_GEN:
        for I in 0 to 15 generate
           SBOX_INST : SBOX port map
             (
                 INPUT => SBOX_IN(I*4 + 3 downto I*4),
                 OUTPUT => SBOX_OUT(I*4 + 3 downto I*4)
             );
        end generate SBOX_GEN;

    PERM_PROC : process (SBOX_OUT)
    begin
        for I in 0 to 15 loop
            NEXT_VECTOR(I) <= SBOX_OUT(I*4);
            NEXT_VECTOR(I+16) <= SBOX_OUT((I*4) + 1);
            NEXT_VECTOR(I+32) <= SBOX_OUT((I*4) + 2);
            NEXT_VECTOR(I+48) <= SBOX_OUT((I*4) + 3);
        end loop;
    end process PERM_PROC;
end ROUND_BODY;
```

Ukázka kódu 4: Architektura entity `ROUND`

```
SBOX_PROC : process (INPUT)
begin
    case INPUT is
        when X"0" => OUTPUT <= X"C";
        when X"1" => OUTPUT <= X"5";
        when X"2" => OUTPUT <= X"6";
        when X"3" => OUTPUT <= X"B";
        when X"4" => OUTPUT <= X"9";
        when X"5" => OUTPUT <= X"0";
        when X"6" => OUTPUT <= X"A";
        when X"7" => OUTPUT <= X"D";
        when X"8" => OUTPUT <= X"3";
        when X"9" => OUTPUT <= X"E";
        when X"A" => OUTPUT <= X"F";
        when X"B" => OUTPUT <= X"8";
        when X"C" => OUTPUT <= X"4";
        when X"D" => OUTPUT <= X"7";
        when X"E" => OUTPUT <= X"1";
        when X"F" => OUTPUT <= X"2";
        when others => OUTPUT <= X"0";
    end case;
end process SBOX_PROC;
```

Ukázka kódu 5: Mapování hodnot v S-boxu

4.2 Komunikace

```
  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 52 13 13 21 00 10 13 00 32 30 00 00 00 33 11 00
00000010: 00 4B 00 11 22 33 44 55 66 77 88 99 AA BB CC DD
00000020: EE FF 50 00 11 22 33 44 55 66 77 00 00 00 00 00
```

Obrázek 4.4: Příklad komunikace

Systém se kompletně ovládá přes sériovou linku při rychlosti 115 200 baudů. Příkaz pro zašifrování textu může vypadat například jako na obrázku 4.4. V tomto příkladu je jako první poslána konfigurace, po které následuje klíč a otevřený text. Po přijetí celého otevřeného textu je po sériové lince nazpět odeslán výsledek šifrování. Konfiguraci a klíč lze zadávat v libovolném pořadí a po jejich nastavení je není třeba opětovně zasílat před každým dalším požadavkem na šifrování. Jejich hodnoty budou platné, dokud se je uživatel nerozhodne změnit.

Dále je vidět, že počáteční bajt každé operace odpovídá ASCII hodnotám písmen R, K a P. Písmeno R (`0x52`) uvozuje 16 bytů konfigurace *(Random)*, písmeno K (`0x4B`) uvozuje 16 bytů klíče *(Key)* a písmeno P (`0x50`) uvozuje 8 bytů otevřeného textu *(Plaintext)*. Po přijetí úvodního písmene se uloží následujících 16, resp. 8 bytů do příslušného registru, a čeká se na další příkaz. Na jiné hodnoty hardware nereaguje.

```
entity RS232 is
  port (
     RXD         : in  STD_LOGIC;
     RXD_DATA    : out STD_LOGIC_VECTOR (7 downto 0);
     RXD_STROBE  : out STD_LOGIC;

     TXD_DATA    : in  STD_LOGIC_VECTOR (7 downto 0);
     TXD_STROBE  : in  STD_LOGIC;
     TXD         : out STD_LOGIC;
     TXD_READY   : out STD_LOGIC;

     CLK         : in  STD_LOGIC;
     RESET       : in  STD_LOGIC
  );
end RS232;
```

Ukázka kódu 6: Rozhraní sériové linky

Pro zpracování signálu na portech `RXD` a `TXD` jsem ve své práci využil implementaci sériové linky používanou v předmětu MI-BHW, jejíž rozhraní popisuje ukázka 6. Signál `RXD_STROBE` informuje o načtení bajtu, který je dostupný po dobu jednoho taktu v `RXD_DATA`. Pro odeslání dat je nejprve nastaven jeden bajt na `TXD_DATA`. Ten je poté odeslán přivedením jedničky na `TXD_STROBE`. Možnost odeslat další bajt signalizuje `TXD_READY` – pokud není v jedničce, stále probíhá zpracování předchozího požadavku.

Toto rozhraní ovládá řadič `COM_CONTROLLER`, který se skládá ze dvou nezávislých automatů typu Moore – jednoho pro přijímání a druhého pro odesílání dat. Jejich činnost není nijak komplikovaná a je zřejmá při pohledu na způsob, jakým jsou data ze sériové linky přijímána v ukázce kódu 7).

V nejvýznamnějším bitu registru `PT_SEL` lze ve výchozím stavu nalézt jedničku a po přijetí odpovídajícího úvodního bajtu (zde to bude `0x50`) ji první automat pomocí signálu `SHIFT_PT` začne posouvat směrem doprava. Všechny pozice kromě té výchozí pak slouží jako signál povolující zápis z `RXD_DATA` do jednotlivých bajtů registru `PT_INPUT`. Poté, co se jednička vrátí zpět na výchozí pozici, je nahrávání dokončeno a řadič na tuto událost reaguje spuštěním šifrování v instanci entity `PRESENT` přes signál `START`. Obdobně funguje i přijímání klíče a konfigurace, avšak po jejich načtení se šifrování nespouští

```
PT_SEL_PROC : process (CLK)
    begin
        if CLK = '1' and CLK'event then
            if RESET = '1' then
                PT_SEL <= (8 => '1', others => '0');
            elsif SHIFT_PT = '1' then
                PT_SEL <= PT_SEL(0) & PT_SEL(8 downto 1);
            end if;
        end if;
    end process PT_SEL_PROC;

PT_INPUT_PROC : process (CLK)
    begin
        if CLK = '1' and CLK'event then
            if RESET = '1' then
                PT_INPUT <= (others => '0');
            else
                for I in 7 downto 0 loop
                    if PT_SEL(I) = '1' then
                        PT_INPUT(8*I+7 downto 8*I) <= RXD_DATA;
                    end if;
                end loop;
            end if;
        end if;
    end process PT_INPUT_PROC;
```

Ukázka kódu 7: Ukládání dat ze sériové linky v entitě `TOP`

a řadič rovnou přechází do výchozího stavu, ve kterém čeká na další platný úvodní bajt.

Druhý automat čeká na signál `DONE` oznamující konec šifrování. Na základě stejného signálu je také do registru `RESULT` zkopírován šifrový text. Odesílání výsledků probíhá tak, že nejvýznamnější bajt registru `RESULT` je připojen na `TXD_DATA` a následně odeslán nastavením signálu `TXD_STROBE` poté, co je obdrženo potvrzení `TXD_READY`. Obsah registru je v dalším taktu posunut o jeden bajt vlevo, díky čemuž je na `TXD_DATA` přiveden následující bajt šifrového textu. Odesílání pokračuje stejným způsobem a proces se opakuje. Počet odeslaných bajtů eviduje čítač, který také ve správný čas posouvání ukončí a vrátí automat do výchozího stavu.

Instance entit `COM_CONTROLLER`, `PRESENT` a `RS232` jsou propojeny v hlavní entitě `TOP`. Zde se také nachází všechny výše zmíněné registry přímo pracující se sériovou linkou stejně jako proces `CLK_SCALER_PROC`, který dělením frekvence oscilátoru vytváří hodinové impulzy.

4.3 Implementace a zapojení

Cílovou platformou je vývojová deska Sakura-G, jež disponuje FPGA Spartan-6 firmy Xilinx a umožňuje připojení osciloskopu a měření spotřeby [4]. Oscilátor zde běží na frekvenci 48 MHz, ale s ohledem na měření osciloskopem byl zpomalen na 1,5 MHz. Cílem je prodloužit periodu jednoho taktu a poskytnout tak více času na ustálení spotřeby. Spotřeba v následujícím taktu pak nebude tolik ovlivněna taktem předchozím. Pro testování jsem také vytvořil verzi kompatibilní s deskou Digilent Basys 3, která používá FPGA novější řady Artix-7 [24]. Jelikož novější syntézní nástroj Vivado 2019.1 a starší ISE 14.7 jsou každý vytvořen pro jednu generaci, musel jsem využít oba. Podstatná pro tuto práci je ale ISE verze pro Sakuru-G.

Signál	Pin	Umístění
SYNC	A5	CN3.2
CLK_48MHz	J1	M_CLK_OSC
RESET	E3	SW4
RX	A8	CN3.7
TX	A9	CN3.8

Tabulka 4.3: Mapování portů entity `TOP` na desce Sakura-G

Deska Sakura-G má sice na USB rozhraní převodník na sériovou linku, nepovedlo se mi pro něj ale zprovoznit ovladač ve Windows 10, a tak jsem byl nucen použít externí převodník. Z toho důvodu je rozhraní sériové linky spolu se synchronizačním impulsem napojeno na skupinu pinů CH3. Číslo za tečkou v tabulce 4.3 pak značí konkrétní pin. Resetování probíhá stiskem tlačítka. Sonda osciloskopu měřící průběh spotřeby je připojena na port J3, který poskytuje zesílený signál oproti alternativnímu portu J2 [4].

U verze pro Basys 3 je navíc ještě připojen sedmisegmentový displej, který je využit pro zobrazování posledních dvou bajtů přijatých po sériové lince. Displej ovládá komponenta `HEX2SEG` používaná při výuce předmětu BI-SAP. Zároveň nebylo u této desky nutné používat externí převodník na sériovou linku. Mapování všech portů pro Basys 3 je součástí zdrojových kódů v příloze.

4.4 Generátor konfigurace

Jak je patrné z popisu komunikačního protokolu, náhodná konfigurace určující počet zpracovávaných rund v každém taktu nebude generována přímo v hardware, ale bude se zadávat externě spolu s otevřeným textem a klíčem.

Hardwarová implementace by totiž neumožňovala testování komplikovanějších scénářů s takovou lehkostí jako implementace softwarová. Nejspíše by nezbylo nic jiného než pro každý případ navrhnout a syntetizovat odlišný design. Ten by navíc byl s největší pravděpodobností velmi komplikovaný,

jelikož konfigurace musí splnit spoustu problematických požadavků. Součet všech hodnot musí být přesně 32, aby se provedlo celé šifrování, konfigurace nemůže být delší než 16 bajtů a žádné platné rundy se nesmějí počítat za ukončující zarážkou.

Dalším argumentem pro zvolený postup je snadná podpora a nastavení speciálních požadavků na formát konfigurace (například na hodnoty prvních pozic), nemluvě o možnosti opakovat měření se stejnou posloupností konfigurací vícekrát. Zároveň nebude nutné kvůli kontrole vygenerovaných konfigurací jejich hodnoty zasílat spolu se šifrovým textem, aby bylo vůbec možné je kontrolovat.

Generátor konfigurací jsem tedy přesunul do programu napsaného v C++, kterému jsem mohl postupně rozšiřovat funkcionalitu a který ve své finální podobě umožňuje několik zajímavých nastavení. Jeho výstupem je soubor s volitelným počtem náhodných konfigurací, ve kterém prvních 32 bajtů obsahuje hlavičku určenou pro měření t-test scénáře nástrojem meas. Ta slouží pouze k nastavení konstantní konfigurace, jejíž použití lze zapnout pro konstantní, náhodné nebo všechny otevřené texty.

První příklad v ukázce 8 vygeneruje sto náhodných konfigurací náhodné délky, kde první takt bude se 60% pravděpodobností prázdný s tím, že ve zbytku případů bude zpracována jedna nebo dvě rundy. V následujícím taktu se poté nutně musí zpracovat 3 rundy. Druhý případ ilustruje použití konstantní konfigurace, která bude využita při šifrování konstantních otevřených textů, zatímco pro ty náhodné se použije 1000 náhodných konfigurací délky 16, kde žádný z taktů nebude prázdný. Detailnější popis všech parametrů nabídne tabulka 4.4.

```
$ gen.out id=01 n=100 prefix=[12000]3 len=random
$ gen.out id=02 const=20010320021001033000220023200111 pt=c
  n=1000 len=16 min=1
```

Ukázka kódu 8: Ukázka použití generátoru konfigurace

4.5 SICAK plugin

Pro měření spotřeby a výpočet t-hodnot ze zaznamenaných dat byl využit software SICAK. Z vícero komplexních nástrojů, které jsou v této sadě k dispozici, mě zajímal především modul meas. Ten posílá po sériové lince otevřený text s klíčem, zaznamená spotřebu během šifrování a přijme výsledek.

Tyto operace samozřejmě probíhají pro každé kryptografické zařízení jinak, a proto se využívá pluginů, které určují, v jakém formátu a v jakém pořadí se instrukce pošlou, stejně jako jaké informace budou výsledkem měření. Jako základní stavební kámen pro plugin kompatibilní s mojí implementací šifry

id	Konfigurace jsou ukládány do souboru `round-config-X.bin`, kde `X` je hodnota argumentu `id`
n	Počet náhodných konfigurací, které budou vygenerovány.
len	Délka šifrování, neboli pozice zarážky v konfiguraci. Platný rozsah 11-32, případně řetězec `random`. Výchozí hodnota je 32.
min	Minimální počet rund v jednom taktu (u náhodných konfigurací). Výchozí hodnota je 0.
max	Maximální počet rund v jednom taktu (u náhodných konfigurací). Výchozí hodnota je 3.
prefix	Povinný začátek všech náhodných konfigurací. Zadán jako série číslic 0-3. Pokud se na některé z pozic specifikovaných v prefixu může objevit více než jedna hodnota, je nutné všechny tyto možnosti uzavřít do hranatých závorek.
const	nastavení konstantní konfigurace *(pro meas plugin)*
pt	pokud není specifikováno, konstantní konfigurace bude použita pro konstantní i náhodné otevřené texty. V opačném případě lze její využití kontrolovat následovně *(pro meas plugin)*: • `c` - použití pouze při konstantním otevřeném textu • `r` - použití pouze při náhodných otevřených textech

Tabulka 4.4: Seznam parametrů generátoru konfigurace

PRESENT posloužil plugin ttest128co určený pro šifru AES. Ten zpočátku nastaví klíč a poté náhodně rozhodne, zda pošle konstantní, nebo náhodně generovaný otevřený text. Na základě tohoto rozhodnutí pak uloží průběh spotřeby do jednoho ze dvou souborů. Taktéž jsou ukládány náhodné otevřené texty spolu s výsledky šifrování, aby bylo později možné zkontrolovat integritu měření.

Hlavním přídavkem z mojí strany bylo zejména načítání konfigurací ze souboru. Plugin před začátkem měření vyhledá soubor `round-config-X.bin` s odpovídajícím identifikátorem na konci, který odpovídá identifikátoru měření, jež se zadává při spouštění modulu meas a je jedním z jeho vstupních argumentů. Po načtení prvních 32 bajtů, které reprezentují hlavičku a konstantní konfiguraci, se nastaví klíč a spustí měření. Před každým šifrováním je nejprve nastavena konfigurace, která se buď použije konstantní, nebo se získá načtením dalších 16 bajtů ze souboru. Využité konfigurace se stejně jako spotřeba uloží do dvou souborů dle typu otevřeného textu.

Kapitola 5

Testování

V této kapitole je popsán proces ověřování korektnosti návrhu a odhalování případných chyb.

5.1 Ověření správnosti návrhu

Před každým měřením a syntézou proběhla logická simulace, která zahrnovala pouze samotnou šifru bez sériové linky a zbytku obvodu. K testování slouží *testbench* `TB_PRESENT`, který vyhledá v adresáři simulace soubor `sim_in.txt` s referenčními daty získanými ze softwarové implementace PRESENTu, kterou poskytl Ing. Jeřábek. Pro každý řádek v souboru je provedeno jedno šifrování. Pokud celá simulace proběhne a všechny výstupy testovaného obvodu se shodují s referencí, je možné postoupit k syntéze. V případě chyby se simulace přeruší a je zobrazena varovná hláška. K neúspěchům zde docházelo zejména zpočátku, kdy jsem ladil funkci řadiče.

Správnost kompletního obvodu jsem testoval až po nahrání do FPGA. Množinu dat použitých při simulaci, upravenou v souladu s komunikačním protokolem a obohacenou o zasílání konfigurací, jsem odeslal po sériové lince a zachytával výsledky. Ty jsem poté porovnal linuxovým nástrojem diff, který nahlásil případné neshody s referencí. Tímto krokem jsem zároveň před zahájením měření ověřil funkčnost celého aparátu, včetně USB/UART převodníku.

Kromě technických problémů s programátorem nebo převodníkem jsem takto odhalil například chybu ve zpracovávání konfigurace. Jednoduše řečeno jsem ji rotoval obráceně, což způsobovalo nesprávné výsledky při použití zarážky. Při správné operaci by k ní řadič došel až po dokončení všech 32 rund, ale jelikož jsem posouval registr opačným směrem, k zarážce se došlo dříve než k jakékoliv jiné nenulové hodnotě a šifrování se ukončilo předčasně. Výsledkem byl tak pouze otevřený text vyxorovaný s klíčem.

5.2 Ověření správnosti měření

Nástroj meas ze sady SICAK při měření spotřeby zároveň zachycuje šifrové texty *(ciphertext)*, aby šlo ověřit, zda nedošlo během měření, která běžně trvají v řádu hodin, k nějaké chybě. Abych tuto možnost vyloučil, použil jsem softwarovou implementaci PRESENTu k vytvoření verifikátoru, který výsledky jednoduše zkontroluje. Každý otevřený text je softwarově zašifrován a výsledek je porovnán s hodnotou získanou během měření. Na konci nástroj vypíše případné chyby a jejich počet. Součástí přílohy jsou otevřené a šifrové texty získané při měřeních zmíněných v této práci a je tak možné jejich správnost překontrolovat.

Kapitola 6

Měření

Cílem měření bylo provést t-test (popsaný v sekci 2.2) na různých verzích protiopatření představených v kapitole 3. Pro měření všech verzí implementovaných na desce Sakura-G [4] byl použit osciloskop PicoScope 6404D [25]. Celý proces od měření až po analýzu byl realizován díky sadě SICAK [5, 20]. Desku i osciloskop ovládal nástroj meas spolu s pluginem popsaným v sekci 4.5. Tento plugin zařizoval šifrování konstantních a náhodných otevřených textů jedním konstantním vstupním klíčem ve všech scénářích popsaných v této kapitole. Třetím vstupním parametrem byla konfigurace popsaná v sekci 3.2, která řídila počet rund zpracovaných během jednoho taktu. Parametry použitých konfigurací jsou spolu s dalšími informacemi uvedeny v tabulkách scénářů. Při měření každého ze scénářů byly uvedené konfigurace použity pro všechna šifrování nezávisle na tom, jestli byl otevřený text náhodný nebo konstantní.

Spotřebu zachycenou osciloskopem plugin uložil do dvou souborů v závislosti na typu otevřeného textu (tyto soubory nejsou pro svoji velikost součástí přílohy). T-test na zachycených záznamech spotřeby provedl nástroj pro statistickou analýzu stan, jehož výstupem je binární soubor s t-hodnotami, které určují míru průsaku informací v čase. Obvod v t-testu neuspěl, pokud absolutní t-hodnota v některém okamžiku přesáhla hranici 4,5. Pro zakreslení t-hodnot do grafu byl využit vizualizační nástroj visu. Ten sloužil i k vykreslování průběhů spotřeby. Při měření jednoho šifrování bylo zachyceno osciloskopem 7813 vzorků spotřeby během necelých 38 hodinových taktů. Doba jednoho taktu odpovídá 208 vzorkům spotřeby a jednotlivé takty jsou ve všech grafech odděleny vertikální čarou.

6.1 Základní verze

Původní návrh (popsaný v [2] a shrnutý v sekci 3.1) jsem testoval ve čtyřech scénářích s konstantními konfiguracemi naměřením 100 000 vzorků. Jelikož bezpečnost stojí na tom, že pro každé šifrování bude konfigurace náhodná, tak bylo zřejmé, že i při tomto množství návrh testem neprojde a t-hodnoty budou vysoké, jak potvrzují výsledky shrnuté v prvních čtyřech řádcích tabulky 6.1. Nejlépe z nich – možná na první pohled nečekaně – dopadl scénář 1, standardní průběh šifrování, kdy se v každém taktu provede právě jedna runda. Nelze však říci, že nebyla použita žádná protiopatření, jelikož v každém taktu byly zároveň počítány dvě falešné rundy. Jejich převaha vysvětluje, proč jsou naměřená maxima t-hodnot oproti zbylým scénářům s konstantní konfigurací poloviční, neboť v případě konfigurací ve scénářích 2-4 je podíl falešných rund celkově nižší.

ID	Konfigurace	Maximální t-hodnota
1	`0x11111111111111111111111111111111`	142,32
2	`0x2222222222222222F000000000000000`	288,95
3	`0x3333333311111111F000000000000000`	353,03
4	`0x3131313131313131F000000000000000`	242,87
5	32 taktů, 0-3	27,16
6	16 taktů, 1-3	56,72
7	náhodný počet taktů, 0-3	10,93
8	náhodný počet taktů, 0-3, *náhodný klíč*=$LFSR\&LFSR$	61,90

Tabulka 6.1: Scénáře 1 až 8 (100 000 vzorků)

Scénáře 2-4 jsou totožné se scénáři použitými v [2] a je tedy možné porovnat jejich výsledky s výsledky popsanými v [2]. Zatímco u scénáře 2 jsou výsledky řádově srovnatelné, u scénáře 3 jsou maximální t-hodnoty v [2] mnohem vyšší, a ani při opakovaných měřeních se tato skutečnost nezměnila.

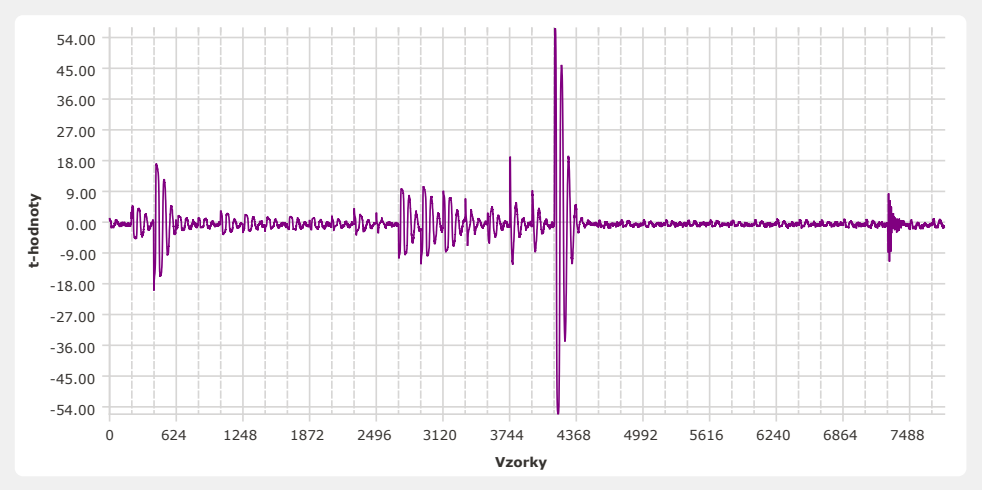
Příčinou může být například odlišné vytváření náhodných hodnot pro falešné rundy. Jako falešný vektor se přímo nabízí použít aktuální hodnotu celého LFSR, ale runda jako vstup potřebuje i nějaký falešný klíč. Ten byl v mém případě získán pomocí

$$(LFSR\&LFSR) \oplus predchoziKlic,$$

kde *predchoziKlic* je vlastně výstup poslední platné rundy (znak & značí zřetězení). Alternativně může být pro vytvoření falešného klíče využito pouze zřetězení hodnoty LFSR, jak je naznačeno ve scénáři 8. V tom případě se ale budou rovnat operandy první operace uvnitř falešné rundy, kterou je xor mezi vektorem a prvními 64 bity klíče. Jeho výsledkem bude vždy nulový vektor, čímž kompletně zbavíme falešné rundy jakékoliv náhodnosti. Při konfiguracích

shodných se scénářem 7 jsou výsledky horší a je tedy jasné, že nelze náhodnost falešných vektorů a klíčů opomíjet.

Scénáře 5-8 v tabulce 6.1 testují modifikaci navrženou v [3]. Při každém šifrování v rámci těchto scénářů je použita jiná náhodná konfigurace. Tyto scénáře dopadly podle očekávání mnohem lépe a scénář 7, který využívá celé spektrum možných konfigurací, se přiblížil hranici 4.5 ze všech nejvíce. Scénář 6 sice dopadl nejhůře, ale je třeba si uvědomit, že naměřená maximální hodnota se nachází až za koncem šifrování a tudíž nutně nevypovídá o bezpečnosti protiopatření. Protože každé šifrování trvá fixně 16 taktů, začne odesílání výsledku po sériové lince vždy ve stejný moment a právě tehdy jde velmi dobře ze spotřeby rozlišit, jestli byl při šifrování použit konstantní nebo náhodný otevřený text. Přesto je ale z obrázku 6.1 patrné, že t-test nebyl úspěšný už na začátku šifrování.



Obrázek 6.1: t-hodnoty scénáře 6

6.2 Falešné registry

V této sekci shrnuji výsledky detailnějších měření na modifikaci obvodu navržené v [3] a popsané v sekci 3.1 (scénáře 9-11). Dále zkoumám vliv falešných registrů, jak jsem navrhl v sekci 3.4 (scénář 12). Analyzováno bylo tentokrát 1 000 000 průběhů spotřeby, zatímco v předchozích měřeních bylo analyzováno pouze 100 000 průběhů spotřeby. Toto navýšení bude platit i pro všechna následující měření. Scénář 9 se od scénáře 7 liší pouze počtem analyzovaných průběhů spotřeby. V důsledku toho se maximální t-hodnota zvýšila.

Detailnější pohled na výsledky náhodných konfigurací nabídne obrázek 6.2, který porovnává první 4 takty scénářů z tabulky 6.2. Jako první je třeba si

všimnout toho, že ve všech spočtených průbězích t-hodnot se začnou hodnoty zvyšovat až u druhého taktu, zatímco v prvním zůstávají vždy nízké. To není chyba zarovnání záznamů, ale je to dáno tím, že ačkoliv se synchronizační impulz aktivuje se začátkem šifrování, první platná data se do registrů zapíší až při další náběžné hraně hodin.

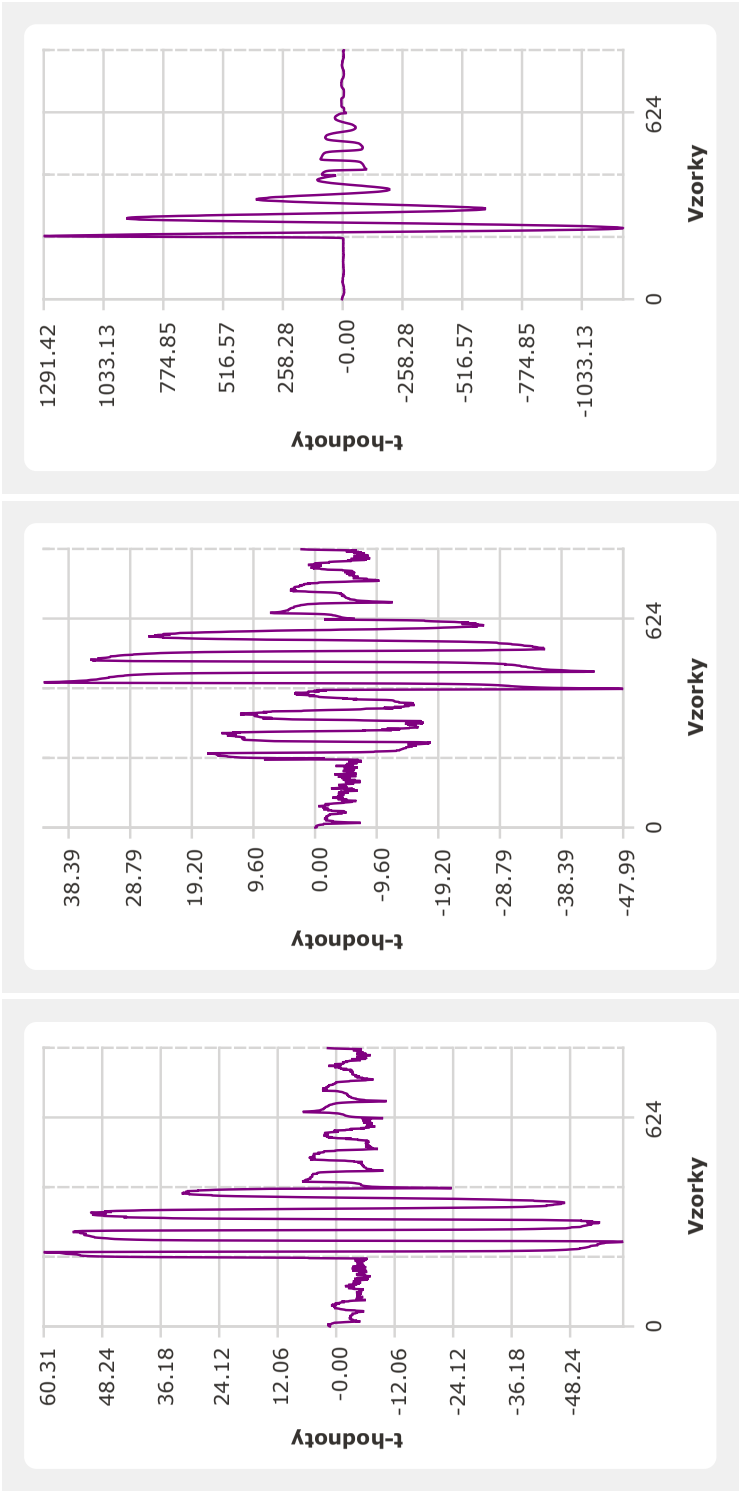
ID	Konfigurace	Maximální t-hodnota
9	náhodný počet taktů, 0-3	60,31
10	náhodný počet taktů, 0-3, první nikdy není 0	47,99
11	náhodný počet taktů, 0-3, první je vždy 0	1291,42
12	opakování scénáře 11 s falešnými registry	19,16

Tabulka 6.2: Scénáře 9 až 12

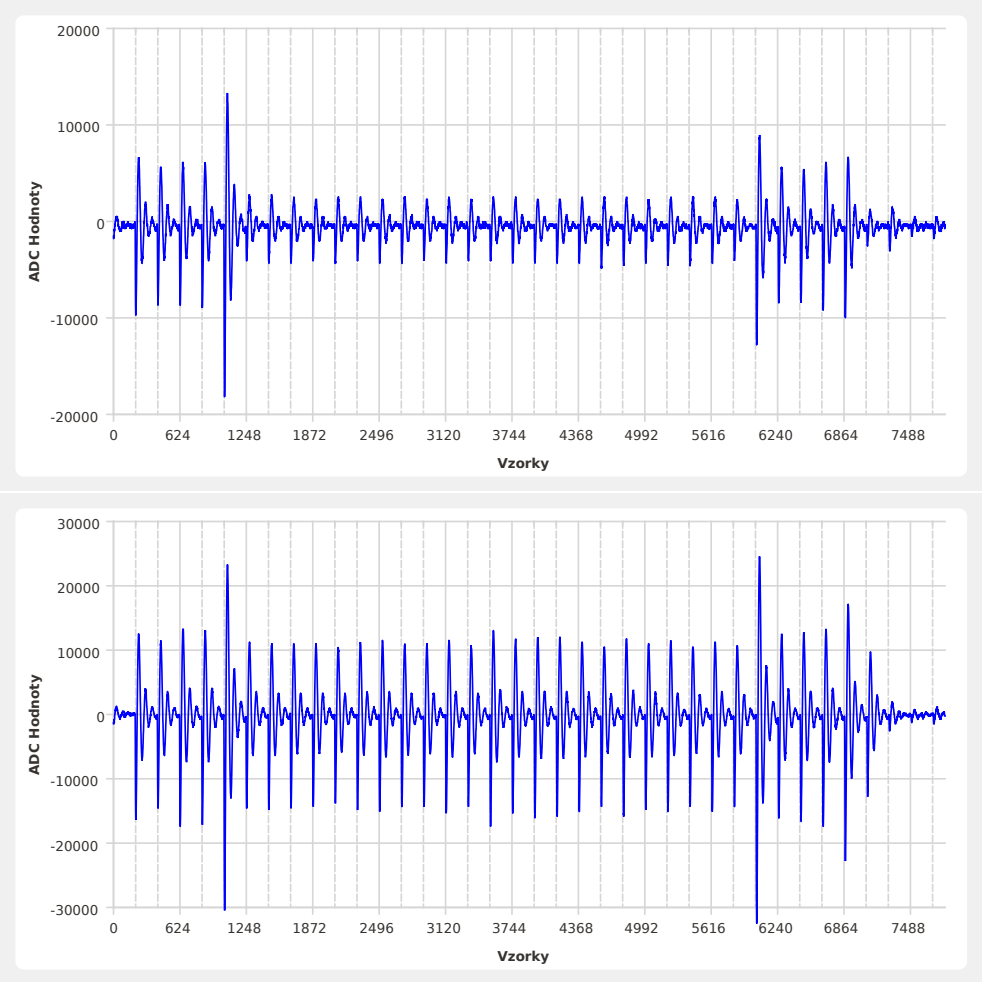
Konfigurace se zaručeně aktivním prvním taktem dopadly ze všech nejlépe, překvapivě i v konkurenci s plně náhodným scénářem 9. To spolu s extrémním maximem u scénáře 11 napovídá tomu, že použití prázdných taktů může způsobovat problémy. Pro potvrzení této domněnky jsem provedl krátké měření na konstantní konfiguraci, která uprostřed obsahovala sérii po sobě jdoucích nul. Z těchto a následně dalších průběhů spotřeby jsem vypozoroval, že dva a více po sobě jdoucích prázdných taktu způsobí výrazný pokles ve spotřebě. Podobná snížení se ale objevovala i během aktivních taktů a nelze tak s jistotou prohlásit, že jde pouze ze spotřeby prázdné takty vyčíst.

Přesto jsem tuto charakteristiku vnímal jako potenciální problém a dospěl jsem tak k první snaze o vylepšení. Do datové cesty jsem přidal falešné registry, do kterých se pouze v prázdných taktech ukládají výsledky falešných rund (viz sekce 3.4). Doufal jsem, že se mi podaří spotřebu vyrovnat a prázdné takty ji nebudou tak předvídatelně ovlivňovat. Jak je vidět na obrázku 6.3, zátěž je skutečně po této úpravě mnohem vyrovnanější. Jediné výraznější výkyvy nastanou v šestém a třicátém taktu, ve kterých nastane přechod mezi prázdným a aktivním taktem. Multiplexery na vstupu první rundy přepnou z platného na náhodný vstup (a obráceně), což má za následek navýšení spotřeby. Obdobné výkyvy se mají tendenci objevit, když se počet platných rund mezi takty změní. I to tedy může potenciálnímu útočníkovi práci usnadnit, avšak jedná se pouze o domněnku a je nutná detailnější analýza.

Scénář 11 jsem se kvůli abnormálně vysokému maximu rozhodl na verzi s falešnými registry zopakovat. Ukázalo se, že ukládání výsledků falešných rund skutečně pomohlo t-hodnoty snížit, avšak pouze v místě onoho maxima. Ve zbytku šifrování byl vliv prázdných taktů na t-hodnoty zanedbatelný. Příčinou zvýšených t-hodnot během druhého taktu může být použití první verze řadiče (viz Obrázek 4.2), která nejprve zkopíruje otevřený text a klíč do registrů pro mezivýsledky a teprve poté pokračuje šifrováním. Právě během tohoto kopírování se maxima objevila. Tento řadič byl ale použit i u scénáře 12 a lepší výsledky má tak na svědomí pouze přidání falešného registru.



Obrázek 6.2: Porovnání t-hodnot scénářů 9, 10 a 11



Obrázek 6.3: Porovnání průběhů spotřeby bez falešných registrů (nahoře) a s nimi (dole). [1]

[1]Spotřeby byly získány ze šifrování, při kterém byla použita konstantní konfigurace 0x33330000000000000000000000003333. Počet platných rund ale zjevně není 32 a tato konfigurace není validní. Účel však obrázek i tak splňuje.

6.3 Prohazování registrů

Navzdory tomu, že vliv falešných registrů na t-hodnoty nebyl ohromující, rozhodl jsem se myšlenku více prozkoumat. Problém jsem viděl v tom, že jejich využití v aktivních taktech je téměř nulové. Řešení, spočívající v prohazování platného a falešného registru, jsem popsal v sekci 3.5. Měřil jsem několik variant, které se lišily především ve způsobu, jakým byla použita hodnota v LFSR pro generování náhodných falešných vektorů a klíčů. U všech variant obvod řídila první verze řadiče (viz Obrázek 4.2).

ID	Konfigurace	Maximální t-hodnota
13	náhodný počet taktů, 0-3 – varianta 1	20,83
14	náhodný počet taktů, 0-3 – varianta 2	99,26
15	náhodný počet taktů, 0-3 – varianta 3 náhodná inicializace	30,73
16	náhodný počet taktů, 0-3 – varianta 3 inicializace na začátku šifrování	58,04
17	náhodný počet taktů, 0-3 – varianta 4	16,40

Tabulka 6.3: Scénáře 13 až 17 - prohazování registrů

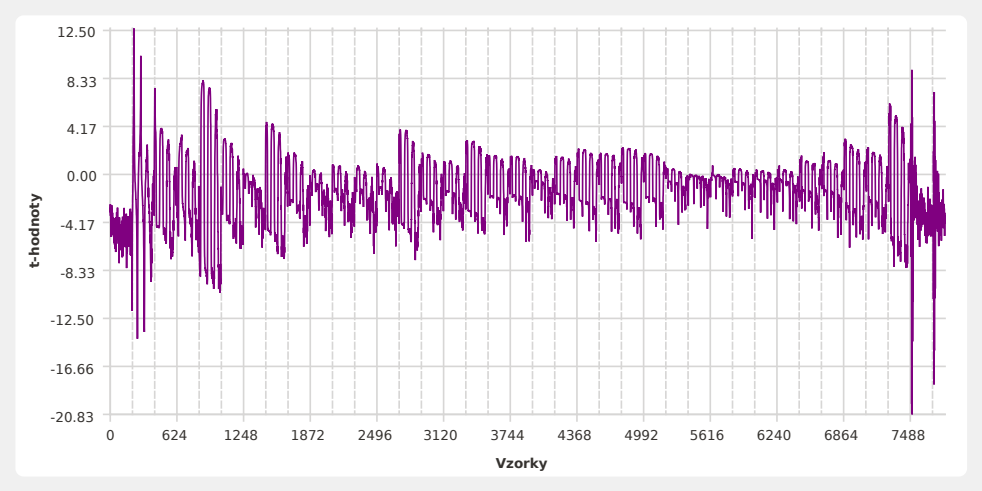
6.3.1 Varianta 1

Pilotní varianta, která měla za cíl pouze otestovat, jestli je možné bezpečnost prohazováním vylepšit. Nová falešná data se ve většině případů získala z výstupu poslední falešné rundy. Pokud ale poslední runda byla platná, falešný registr se aktualizoval následovně:

```
VECTOR1 <= VECTOR2 xor LFSR_REG;
KEY1 <= KEY1(66 downto 32) &
        (LFSR_REG xor (KEY1(31 downto 0)&KEY1(127 downto 96))) &
        KEY1(95 downto 67);
```

Na příkladu je situace, kdy se falešný vektor nachází v registru `VECTOR2` a falešný klíč v registru `KEY2`. Jelikož se falešná data po každém taktu přesouvají do druhého z dvojice rovnocenných registrů, při vytváření nové hodnoty falešného vektoru je správně čteno z jiného registru, než do kterého se zapisuje. Pokud by se zrovna falešný vektor nacházel v registru `VECTOR1`, pozice registrů by v prvním přiřazení byly obráceně.

V případě klíče tomu tak ale není, což znamená, že nový falešný klíč bude vycházet z platného mezivýsledku. Toto je neúmyslná chyba a při zápisu do `KEY1` se mělo číst z `KEY2`. Vliv tohoto pochybení na úspěšnost v t-testu je ale malý, jak je patrné v kontextu s ostatními variantami. Postup, jakým je klíč aktualizován, se může zdát zbytečně komplikovaný, ale snažil jsem se jej přiblížit operacím prováděným v entitě `ROUND` (viz Ukázka kódu 4).



Obrázek 6.4: t-hodnoty scénáře 13

Při pohledu na obrázek 6.4 si lze všimnout maximální hodnoty 14,27 na začátku šifrování, což se dá považovat za solidní výsledek. Absolutní maximum je dosaženo až po skončení šifrování.

6.3.2 Varianta 2

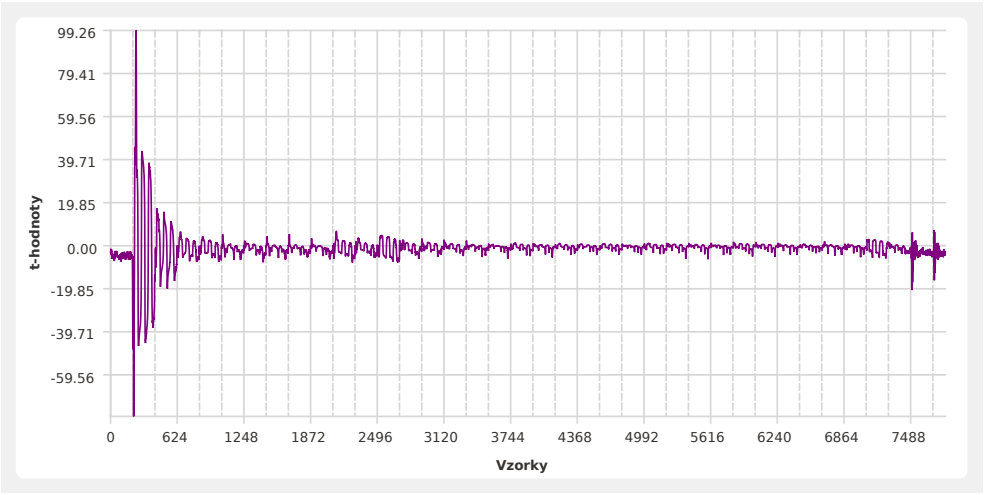
Ačkoliv první varianta přinesla výrazné zlepšení, zvýšené hodnoty na začátku šifrování stále vyčnívají a nejsou v doporučeném rozmezí. Rozhodl jsem se tedy řešit problematické generování falešných hodnot přidáním čtvrté rundy.

Tato přidaná runda bude v každém taktu falešná a nejen že zajistí konzistentnější generování náhodných dat, ale zároveň jsem předpokládal, že navýšení celkového počtu rund zlepší výsledky v průběhu celého šifrování. K mému překvapení se ale ani jeden z benefitů při měření neprojevil a výsledky byly horší než u předchozí verze.

Do falešných registrů totiž musí někdy přijít hodnota odvozená z LFSR a není tak možné využívat jen výstupu poslední rundy. Postupoval jsem ale neoptimálně a inicializaci falešných registrů jsem se rozhodl provádět na začátku šifrování. Jako zdroj náhodných dat sloužily tyto signály:

```
RND_VECTOR <= VECTOR_SHADOW;
RND_KEY <= KEY_SHADOW xor ((not LFSR_REG) & LFSR_REG);
```

Pokud se tedy měl na začátku šifrování přesunout například falešný klíč do registru `KEY1`, získala se nová hodnota ze signálu `RND_KEY`. Od varianty 1 se tento postup liší tím, že nový falešný klíč vytvářím jen jednou ze signálu `KEY_SHADOW`, který se v každém taktu aktualizuje podle toho, v jakém z registrů se falešný klíč momentálně nachází. Není tak nutné stejné xorování



Obrázek 6.5: t-hodnoty scénáře 14

s `LFSR_REG` provádět jednou pro `KEY1` a jednou pro `KEY2`. Analogicky probíhá i inicializace falešného vektoru.

Aby mezi falešným klíčem a vektorem nebyl žádný vztah, LFSR se využilo pouze pro vytvoření klíče, zatímco vektor se nijak při inicializaci neměnil a zůstala hodnota z předchozího šifrování. To vše nejspíš přispělo k o dost vyššímu maximu na obrázku 6.5.

6.3.3 Varianta 3

Signály `RND_VECTOR` a `RND_KEY` jsem se rozhodl zrušit a vrátil jsem se zpět k tomu, že do registru 1 zapisuji přímo obsah registru 2 a naopak. Výsledek sice bude stejný, ale je rozdíl v tom, jak ho syntézní nástroj dosáhne. Logika pro generování náhodných dat se totiž objeví v obvodu dvakrát pro každý z registrů. Pokud se ale při generování použijí hodnoty vycházející z `VECTOR_SHADOW` nebo `KEY_SHADOW`, bude vytvořena pouze jednou.

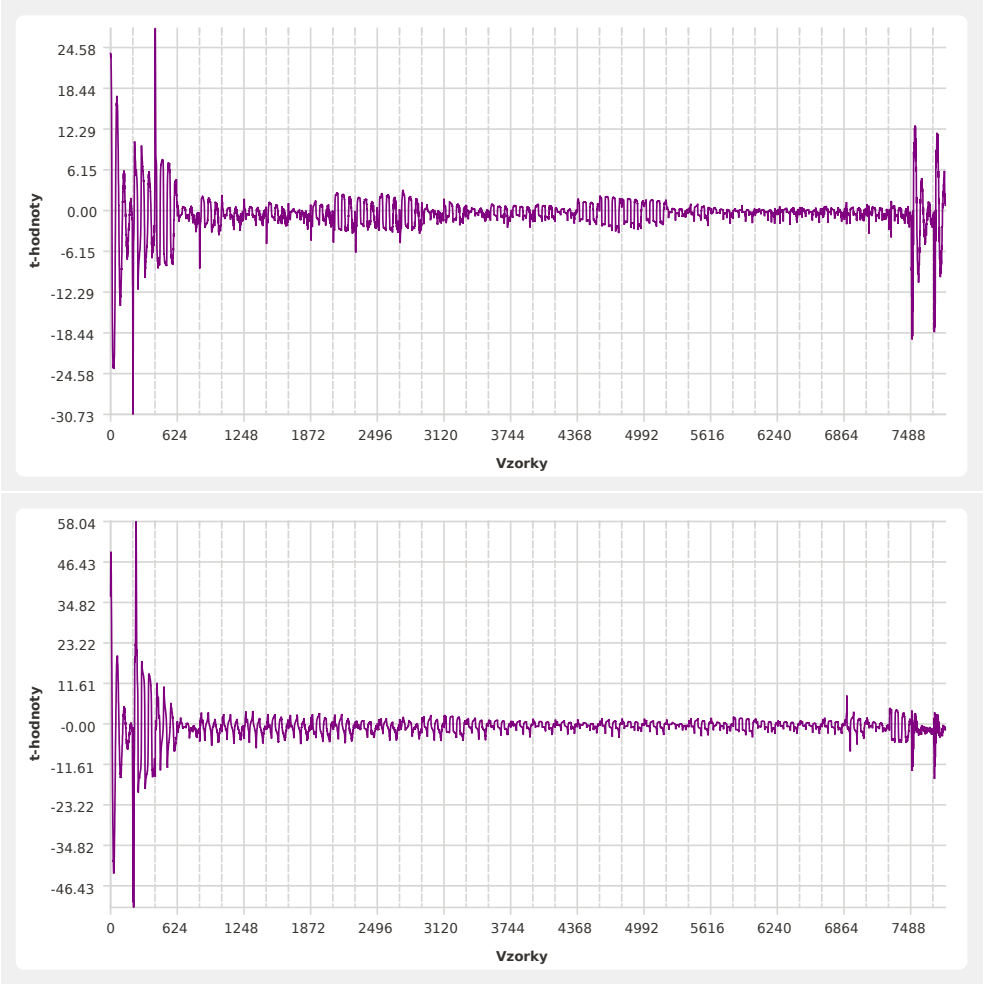
Zároveň jsem zkusil vymyslet inicializování falešného vektoru tak, aby se LFSR využilo, ale jeho hodnota se v něm přímo neobjevila. Spodní polovina LFSR tedy určovala, které bity se ve spodní polovině předchozího falešného vektoru prohodí se zrcadlovými protějšky v jeho horní polovině. Pokud měl například poslední bit LFSR hodnotu 1, prohodil se první a poslední bit falešného vektoru. Na pozicích, kde byla v LFSR hodnota 0, k prohození nedošlo. Generování falešného klíče probíhalo stejně jako u varianty 2.

```
for I in 31 downto 0 loop
    if LFSR_REG(I) = '1' then
        VECTOR1(I) <= VECTOR2(63 - I);
        VECTOR1(63 - I) <= VECTOR2(I);
    else
        VECTOR1(I) <= VECTOR2(I);
        VECTOR1(63 - I) <= VECTOR2(63 - I);
    end if;
end loop;

KEY1 <= KEY2 xor ((not LFSR_REG) & LFSR_REG);
```

Inicializaci jsem zkoušel provádět jak náhodně (hodnota dvou nejvýznamnějších bitů v LFSR je `0b11`), tak na začátku šifrování spolu s načítáním vstupních dat. V obou případech jsem dosáhl zlepšení oproti variantě 2, ale lépe si vedla verze s náhodnou inicializací. Kromě toho jsou výsledky zajímavé i v jiném ohledu. Na rozdíl od všech dosavadních měření se objevily zvýšené t-hodnoty na absolutním začátku šifrování, při aktivování synchronizačního signálu.

Během ladění jsem totiž zasáhl i do (stále první verze) řadiče a deaktivoval jsem signál `LOAD` ve výchozím stavu. Čili jedinou změnou v kombinační logice po spuštění šifrování bylo přivedení otevřeného textu a klíče na vstupy registrů, na což přirozeně t-test reagoval. U předchozích scénářů 1-14 byl multiplexer na vstupu registrů správně nastaven už před začátkem šifrování.



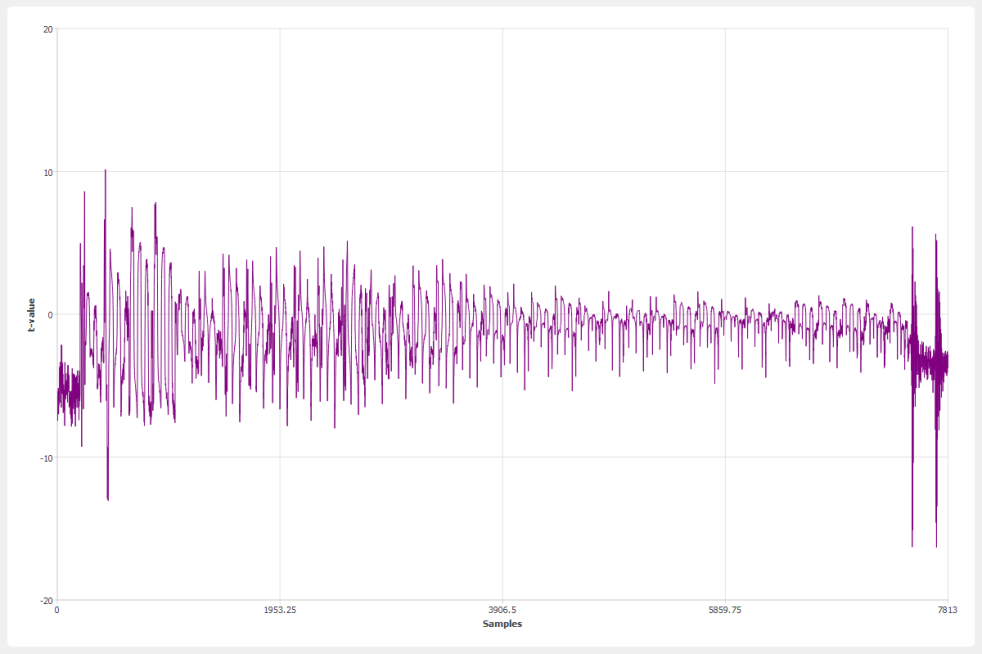
Obrázek 6.6: Porovnání t-hodnot scénářů 15 a 16

6.3.4 Varianta 4

Aby šlo definitivně rozhodnout, jestli má smysl přidávat další rundu, přidal jsem druhý 128 bitový LFSR pro inicializaci falešného klíče, který měl zajistit nejlepší možnou nezávislost falešných vektorů a klíčů. Signál `LOAD` byl opět povolen ve výchozím stavu řadiče (čili jeho činnost opět odpovídala obrázku 4.2) a inicializace se začala provádět po skončení šifrování, při aktivním signálu `DONE`. S inicializací falešných registrů se zároveň provedlo přemazání šifrového textu náhodným výsledkem čtvrté rundy.

Navzdory větším prostorovým nárokům se ale zdá, že přidání další rundy nevede k efektivnějšímu skrývání spotřeby. Naneštěstí jsem při manipulaci s výsledky zálohoval pouze exportovaný obrázek 6.7 s průběhem t-hodnot,

takže maximální hodnota je pouze odhad vyčtený z tohoto obrázku. V každém případě jsou ale výsledky t-test velice podobné variantě 1 a ke snížení maximální t-hodnoty došlo až na konci záznamu po dokončení šifrování.



Obrázek 6.7: Porovnání t-hodnot scénáře 17

6.4 Porovnání optimalizovaných verzí

Poslední měření zkoumala vliv optimalizací popsaných v sekci 3.6. Hlavní vylepšení spočívalo v aktualizaci řadiče (viz Obrázek 4.3), která především umožňuje nahrání vstupního otevřeného textu a klíče až v prvním aktivním taktu. Zároveň je platný výsledek po dokončení šifrování mazán. Vznikly dvě verze s takto optimalizovaným řadičem.

ID	Konfigurace	Maximální t-hodnota
18	náhodný počet taktů, 0-3 První optimalizovaná verze (bez prohazování)	11,83
19	náhodný počet taktů, 0-3 Druhá optimalizovaná verze (s prohazováním)	17,38

Tabulka 6.4: Scénáře 18 a 19

První optimalizovaná verze vychází ze základních *dummy rounds* bez falešných registrů, popsaných v sekci 3.1 a otestovaných v sekci 6.1. Kromě řadiče bylo pozměněno generování náhodného klíče pro falešné rundy. Připomínám, že v základní verzi není žádný falešný registr, jehož hodnoty by šly ve falešných rundách použít, proto je na příkladu níže vidět přiřazení náhodných hodnot přímo na vstup druhé rundy. Přiřazení probíhá analogicky i u ostatních rund – vždy se pro náhodný klíč využije výstup předchozí rundy, případně `KEY_REG`, pokud je již první runda falešná.

```
IN_VECTOR_2 <= LFSR_REG;
IN_KEY_2 <=
       (LFSR_REG(3 downto 0) & LFSR_REG & LFSR_REG(63 downto 4))
        xor OUT_KEY_1;
```

V neoptimalizované základní verzi (viz sekce 6.1) nebylo LFSR před xorováním s klíčem posunuto, a tak se po dalším xoru s náhodným vektorem, kterým bylo to stejné LFSR, na vstupech S-boxů objevila pouze známá hodnota klíče. Náhodnost falešných rund tak byla výrazně ovlivněna.

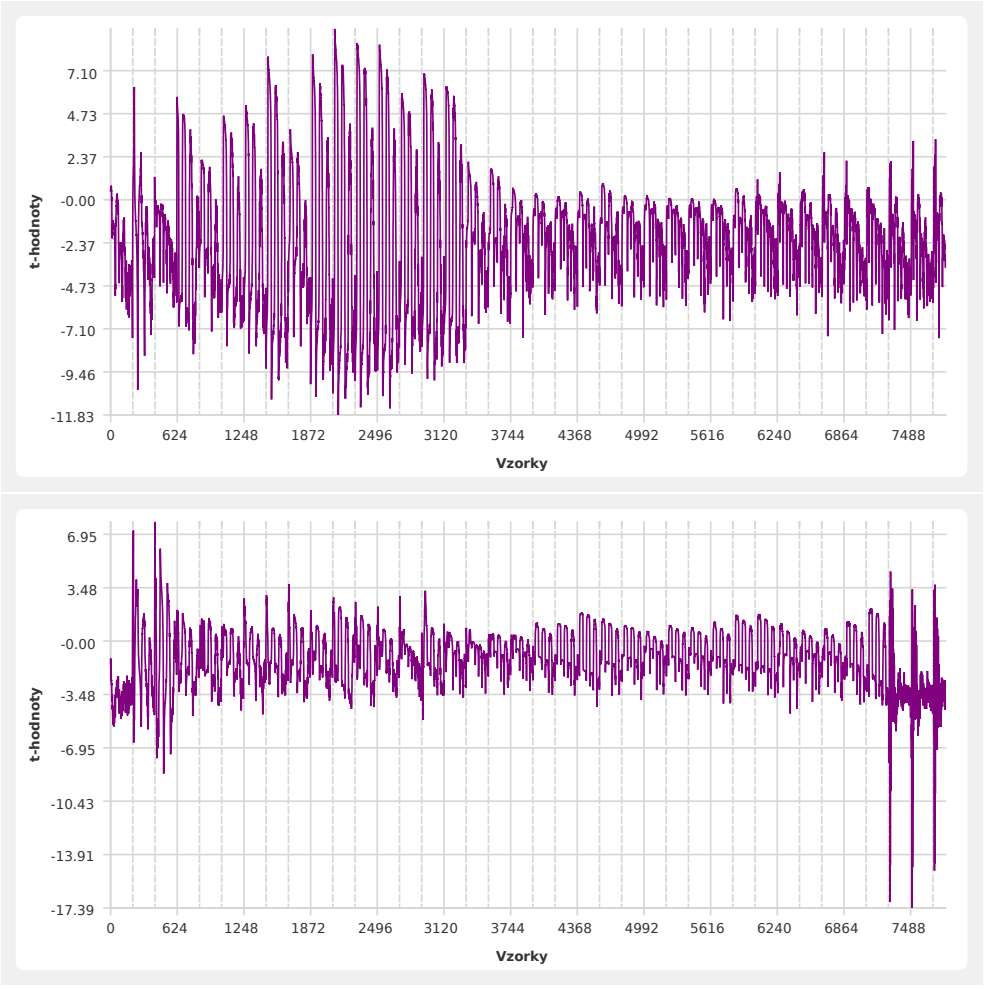
Druhá optimalizovaná verze vylepšuje prohazování registrů se třemi rundami popsané v sekci 6.3.1. I zde nastaly změny v generování náhodných dat. Ta jsou ale na rozdíl od první optimalizované verze nejprve uložena do falešných registrů, a to následovně:

```
if DONE = '1' then
    VECTOR1 <= LFSR_REG;
    KEY1 <= (OUT_VECTOR_1 & OUT_VECTOR_2) xor OUT_KEY_3 ;
elsif FULL_ROUND = '1' then
    VECTOR1 <= VECTOR2 xor LFSR_REG;
    KEY1 <= KEY2(3 downto 0) & KEY2(127 downto 4);
else
    VECTOR1 <= OUT_VECTOR_3;
    KEY1 <= OUT_KEY_3;
end if;
```

Je vidět, že inicializace falešných registrů je rozlišena podle toho, jestli nastal konec šifrování, nebo jen není v aktuálním taktu k dispozici žádná falešná runda, protože jsou všechny rundy platné. U inicializace klíče je důležité dodat, že při aktivním `DONE` jsou všechny rundy falešné a signály `OUT_VECTOR_1` a `OUT_VECTOR_2` je tak možné použít jako zdroj náhodných dat. Opět platí, že obdobný proces lze v entitě `DATAPATH` nalézt ještě jednou při zápisu do registrů `VECTOR2` a `KEY2`.

Z výsledků měření v tabulce 6.4 je zřejmé, že použití vylepšeného řadiče přineslo snížení maximálních t-hodnot pro obě verze. Zlepšení u první optimalizované verze je v porovnání s hodnotami v tabulce 6.2 enormní. Zdá se tedy, že předpoklady o optimalizaci vyslovené v [3] mají reálný základ a je opravdu možné bezpečnost *dummy rounds* zlepšit opožděným nahráváním platných dat.

Druhá optimalizovaná verze, tedy prohazování falešného registru, dopadla na první pohled o něco méně působivěji (obrázek 6.8 dole). Naměřené maximum se ale objevilo až po dokončení šifrování, u konce průběhu t-hodnot. Přesto jsem tuto nepříznivou část záznamu neignoroval, jelikož ve stejných taktech jsem dostal obdobné výsledky i u ostatních verzí s prohazováním (viz sekce 6.3). To značí, že verze s prohazováním mohou mít nějakou společnou chybu, díky které se ve spotřebě objevují informace o otevřeném textu i po dokončení šifrování. V optimalizované verzi je akorát o jeden výrazný „propad“ do záporných t-hodnot více, jelikož bez stavu `INIT` ve vylepšeném řadiči je šifrování o takt kratší. V průběhu šifrování je ale situace příznivější a maximální hodnota nepřesáhla **8,63**. Kromě toho lze jediné výraznější výkyvy pozorovat v druhém a třetím taktu, zatímco první optimalizovaná verze se v podobně vysokých číslech pohybovala mnohem častěji, nejvíce v první polovině šifrování (obrázek 6.8 nahoře). Také stále platí, že spotřeba základní verze i po optimalizaci nijak neskrývá charakteristiky jako například sníženou spotřebu při sérii prázdných taktů. Přesto jsou si ale t-hodnoty velice podobné a je tak otázkou, jestli benefity prohazování převáží zvýšení nároků na prostor na čipu.



Obrázek 6.8: Porovnání t-hodnot scénářů 18 a 19

Závěr

Cíle práce se povedlo naplnit. Implementace šifry PRESENT chráněné pomocí *dummy rounds* splňuje všechny požadavky, je jednoduše modifikovatelná a její fungování lze snadno ovládat externě bez zásahu do zdrojového kódu. Pro získání platných konfigurací je možné využít specializovaný nástroj pro příkazovou řádku, který umí vygenerovat řídící sekvence v závislosti na požadavcích uživatele. Tyto konfigurace poté mohou být využity nejen při měření pomocí nástroje SICAK. Neopominul jsem ani testování návrhu, které dostatečně zajistilo a ověřilo korektnost všech testovaných variant návrhu.

Na milionu záznamů spotřeby při různých scénářích a verzích obvodu byl proveden t-test, který měl za cíl odhalit, zdali je mezi spotřebou a zpracovávanými daty nějaký vztah. Míru pravděpodobnosti takového vztahu určuje průběh t-hodnot v čase. Ačkoliv se t-hodnoty všech testovaných variant mnohdy k hranici úspěšnosti 4,5 seshora přibližovaly, a často ji přesahovaly pouze o řády jednotek, žádná se pod ní nedokázala dostat. Proto prozatím nelze žádný z přístupů popsaných v této práci označit za prokazatelně bezpečný.

Pro naplnění cílů práce bylo nejdůležitější zaměřit se na základní verzi, vycházející z návrhu Ing. Stanislava Jeřábka. Měření odhalila zranitelnost především na začátku šifrování, což koresponduje s Jeřábkovými výsledky. Stejně tak se potvrdila jeho úvaha o tom, že je možné bezpečnost *dummy rounds* vylepšit, pokud se nahrání platných dat do obvodu provede až po uběhnutí náhodně dlouhého časového intervalu určeného konfigurací, během kterého se budou šifrovat pouze náhodně generované hodnoty.

Zároveň jsem ale zjistil, že spotřeba základní verze má určité charakteristiky, které potenciálně mohou vést k odhalení informací o použité konfiguraci. Proto jsem do obvodu v dalších verzích přidal falešný registr, který slouží k ukládání náhodných dat, získaných z výstupů falešných rund. Schéma ukládání jsem poté systematicky pozměnil tak, aby se obsah platného a falešného registru po každém taktu prohodil.

Tato modifikace vedla sice k mnohem rovnoměrnějšímu průběhu spotřeby, ale navzdory tomu se maximální hodnoty v t-testech až tak výrazně nezlepšily.

Přesto je zlepšení možné pozorovat a verze s prohazováním registrů si vede o něco lépe. Avšak je otázkou, jestli je reálná odolnost vůči útokům o tolik větší, aby bylo možné ospravedlnit její použití oproti úspornější a méně komplikované základní verzi. Nicméně se domnívám, že ono prohazování platných a náhodných dat může skrývat jistý potenciál a stojí za to se mu v budoucnu více věnovat.

Bibliografie

1. BOGDANOV, A.; KNUDSEN, L. R.; LEANDER, G.; PAAR, C.; POSCHMANN, A.; ROBSHAW, M. J. B.; SEURIN, Y.; VIKKELSOE, C. PRESENT: An Ultra-Lightweight Block Cipher. In: PAILLIER, Pascal; VERBAUWHEDE, Ingrid (ed.). *Cryptographic Hardware and Embedded Systems - CHES 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 450–466. ISBN 978-3-540-74735-2.
2. JEŘÁBEK, Stanislav; SCHMIDT, Jan; NOVOTNÝ, Martin; MIŠKOVSKÝ, Vojtěch. Dummy rounds as a DPA countermeasure in hardware. In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, s. 523–528.
3. JEŘÁBEK, S.; SCHMIDT, J. Analyzing and Optimizing the Dummy Rounds Scheme. In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2019, s. 1–4. ISSN 2334-3133. Dostupné z DOI: 10.1109/DDECS.2019.8724632.
4. GUNTUR, H.; ISHII, J.; SATOH, A. Side-channel AttacK User Reference Architecture board SAKURA-G. In: *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*. 2014, s. 271–274. ISSN 2378-8143. Dostupné z DOI: 10.1109/GCCE.2014.7031104.
5. SOCHA, Petr; MIŠKOVSKÝ, Vojtěch; NOVOTNÝ, Martin. SICAK: An open-source SIde-Channel Analysis toolKit. In: *8th Workshop on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE)*. 2019.
6. SCHNEIDER, Tobias; MORADI, Amir. Leakage assessment methodology. *Journal of Cryptographic Engineering*. 2016, roč. 6, č. 2, s. 85–99. ISSN 2190-8516. Dostupné z DOI: 10.1007/s13389-016-0120-y.
7. SHANNON, C. E. Communication theory of secrecy systems. *The Bell System Technical Journal*. 1949, roč. 28, č. 4, s. 656–715. ISSN 0005-8580. Dostupné z DOI: 10.1002/j.1538-7305.1949.tb00928.x.

8. KOCHER, Paul; JAFFE, Joshua; JUN, Benjamin. Differential Power Analysis. In: WIENER, Michael (ed.). *Advances in Cryptology – CRYPTO' 99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, s. 388–397. ISBN 978-3-540-48405-9.

9. BRIER, Eric; CLAVIER, Christophe; OLIVIER, Francis. Correlation power analysis with a leakage model. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2004, s. 16–29.

10. GILBERT GOODWILL, Benjamin Jun; JAFFE, Josh; ROHATGI, Pankaj et al. A testing methodology for side-channel resistance validation. In: *NIST non-invasive attack testing workshop*. 2011, sv. 7, s. 115–136.

11. POPP, Thomas; MANGARD, Stefan; OSWALD, Elisabeth. Power analysis attacks and countermeasures. *IEEE Design & test of Computers*. 2007, roč. 24, č. 6, s. 535–543.

12. DANGER, J. L.; GUILLEY, S.; BHASIN, S.; NASSAR, M. Overview of Dual rail with Precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors. In: *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*. 2009, s. 1–8. Dostupné z DOI: 10.1109/ICSCS.2009.5412599.

13. NAGASHIMA, S.; HOMMA, N.; IMAI, Y.; AOKI, T.; SATOH, A. DPA Using Phase-Based Waveform Matching against Random-Delay Countermeasure. In: *2007 IEEE International Symposium on Circuits and Systems*. 2007, s. 1807–1810. ISSN 2158-1525. Dostupné z DOI: 10.1109/ISCAS.2007.378024.

14. HERBST, Christoph; OSWALD, Elisabeth; MANGARD, Stefan. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In: ZHOU, Jianying; YUNG, Moti; BAO, Feng (ed.). *Applied Cryptography and Network Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 239–252. ISBN 978-3-540-34704-0.

15. VEYRAT-CHARVILLON, Nicolas; MEDWED, Marcel; KERCKHOF, Stéphanie; STANDAERT, François-Xavier. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In: WANG, Xiaoyun; SAKO, Kazue (ed.). *Advances in Cryptology – ASIACRYPT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 740–757. ISBN 978-3-642-34961-4.

16. SOCHA, P.; MIŠKOVSKÝ, V.; NOVOTNÝ, M. First-Order and Higher-Order Power Analysis: Computational Approaches and Aspects. In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. 2019, s. 1–5. ISSN 2377-5475. Dostupné z DOI: 10.1109/MECO.2019.8760033.

17. GIERLICHS, Benedikt; BATINA, Lejla; PRENEEL, Bart; VERBAUWHEDE, Ingrid. Revisiting Higher-Order DPA Attacks: in: PIEPRZYK, Josef (ed.). *Topics in Cryptology - CT-RSA 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 221–234. ISBN 978-3-642-11925-5.

18. OSWALD, Elisabeth; MANGARD, Stefan; HERBST, Christoph; TILLICH, Stefan. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In: POINTCHEVAL, David (ed.). *Topics in Cryptology – CT-RSA 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 192–207. ISBN 978-3-540-32648-9.

19. SOCHA, P.; BREJNÍK, J.; JEŘÁBEK, S.; NOVOTNÝ, M.; MENTENS, N. Dynamic Logic Reconfiguration Based Side-Channel Protection of AES and Serpent. In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 2019, s. 277–282. ISSN null. Dostupné z DOI: 10.1109/DSD.2019.00048.

20. SOCHA, Petr. *SICAK: SIde-Channel Analysis toolKit User's Guide* [online]. 2019. Verze 1.1 [cit. 2019-12-22]. Dostupné z: petrsocha.github.io/sicak/userguide.

21. PANDA, Amit Kumar; RAJPUT, Praveena; SHUKLA, Bhawna. FPGA implementation of 8, 16 and 32 bit LFSR with maximum length feedback polynomial using VHDL. In: *2012 International Conference on Communication Systems and Network Technologies*. 2012, s. 769–773.

22. CHAKRABORTY, Abhishek; MAZUMDAR, Bodhisatwa; MUKHOPADHYAY, Debdeep. Fibonacci lfsr vs. galois lfsr: Which is more vulnerable to power attacks? In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. 2014, s. 14–27.

23. GORESKY, Mark; KLAPPER, Andrew M. Fibonacci and Galois representations of feedback-with-carry shift registers. *IEEE Transactions on Information Theory*. 2002, roč. 48, č. 11, s. 2826–2836.

24. DIGILENT. *Basys 3™ FPGA Board Reference Manual* [online]. 2019. Verze C [cit. 2019-12-22]. Dostupné z: https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf.

25. PICO TECHNOLOGY LTD. *PicoScope® 6 Návod k použití* [online]. 2016. R41 [cit. 2019-12-22]. Dostupné z: https://www.picotech.com/download/manuals/PicoScope6UserGuideCS.pdf.

Příloha A

Seznam použitých zkratek

ASCII *American Standard Code for Information Interchange*

BI-SAP Struktura a architektura počítačů

CLK hodinový signál *(clock)*

CT Šifrový text *(ciphertext)*

FPGA Programovatelné hradlové pole *(Field-programmable gate array)*

IN vstup *(input)*

LFSR Lineární zpětnovazebný posuvný registr *(Linear-feedback shift register)*

MI-BHW Bezpečnost a technické prostředky

OS Operační systém

OUT výstup *(output)*

PT Otevřený text *(plaintext)*

UART *Universal asynchronous receiver-transmitter*

USB *Universal Serial Bus*

VHDL *Very High Speed Integrated Circuit Hardware Description Language*

Příloha **B**

Obsah přiloženého CD

- `src`
  - `common` ..... zdrojové kódy společné pro všechny verze
  - `dummy` ..... základní verze dummy rounds
  - `dummy_optimized` ..... optimalizovaná základní verze
  - `shadow` ..... verze s falešnými registry
  - `switch_optimized` ..... optimalizovaná verze s prohazováním
  - `switch_v1` ..... verze s prohazováním – varianta 1
  - `switch_v2` ..... verze s prohazováním – varianta 2
  - `switch_v3` . verze s prohazováním – varianta 3 s inicializací na začátku šifrování
  - `switch_v3_randomInit` verze s prohazováním – varianta 3 s náhodnou inicializací
  - `switch_v4` ..... verze s prohazováním – varianta 4
- `meas`
  - `configs` ..... konfigurační soubory pro meas plugin
  - `results` ..... výsledky šifrování, která proběhla během měření
  - `tvals` ..... t-hodnoty měřených scénářů
- `tools`
  - `sicak`
    - `ttest128present` ..... plugin pro nástroj meas
    - `config.json` ..... nastavení osciloskopu a sériové linky
    - `meas.bat` ..... skript pro spuštění měření
    - `ttest.bat` .. skript pro vyhodnocení t-hodnot ze záznamů spotřeby
  - `conf_gen.cpp` ..... zdrojový kód generátoru konfigurací
  - `conf_load.cpp` ..... zdrojový kód nástroje na čtení vygenerovaných konfigurací
- `test-data`. ..... referenční data pro testování
- `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `thesis.pdf` ..... text práce ve formátu PDF