FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Implementace Paillierova kryptosystému a útok injekcí chyb na procesoru CEC 1702
Student:	Lukáš Daněk
Vedoucí:	Dr.-Ing. Martin Novotný
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Implementujte Paillierův kryptosystém na procesoru Microchip CEC 1702. Použijte knihovnu „bigi" určenou pro implementaci kryptografických algoritmů na mikrokontrolérech a proveďte nezbytné úpravy. Při implementaci Paillierova kryptosystému vyjděte z jeho stávající implementace pro CEC 1302 a proveďte nezbytné úpravy. Vytvořte čistě softwarovou variantu firmware a variantu využívající vestavěný hardwarový akcelerátor RSA. Firmware má podporovat různé šířky klíčů.
Dále s pomocí knihovny „bigi" vytvořte pro procesor CEC 1702 čistě softwarovou variantu RSA-CRT a variantu RSA-CRT využívající jeho vestavěný hardwarový akcelerátor. S pomocí těchto variant prozkoumejte možnosti provedení útoku injekcí chyb (fault-injection attack) prostřednictvím soupravy ChipWhisperer. Porovnejte tento útok s útokem na procesor STM32.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 12. prosince 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Implementace Paillierova kryptosystému a útok injekcí chyb na procesoru CEC 1702

Lukáš Daněk

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

12. května 2022

Poděkování

Na prvním místě bych rád poděkoval Dr.-Ing. Martinu Novotnému za námět práce, za cenné rady a čas, který mi během práce věnoval. Mé poděkování patří také Ing. Jakubu Klemsovi za jeho odborné konzultace a rady a Terce Horníčkové, se kterou jsem při práci často spolupracoval. Dále bych chtěl poděkovat příteli Dominikovi za pomoc se stylistikou a psychickou podporu.

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 12. května 2022 . . . . . .

Odkaz na tuto práci

Daněk, Lukáš. *Implementace Paillierova kryptosystému a útok injekcí chyb na procesoru CEC 1702*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Bakalářská práce se zaměřuje na práci s kryptoprocesorem CEC1702. První část práce je orientovaná na zprovoznění programování kryptoprocesoru, úpravu již vzniklé implementace knihovny pro práci s velkými čísly a implementace Paillierova kryptosystému pro zmíněný kryptoprocesor. Dále je v rámci práce vytvořen firmware, který umožňuje použití šifrovacího algoritmu RSA-CRT a Paillierova kryptosystému s variabilní délkou klíče. Druhá část práce se zaměřuje na útoky pomocí injekce chyb na šifrovací algoritmus RSA-CRT. Je použito generování zákmitů na napájení a zákmitů na zdroji hodinového signálu, a to pomocí sady ChipWhisperer.

Útoky jsou nejprve úspěšně provedeny na mikrokontrolér STM32F3, kdy je RSA-CRT implementováno pomocí knihovny pro práci s velkými čísly z první části práce. S využitím znalostí z útoků na STM32F3 byla vytvořena další sada útoků pro kryptoprocesor CEC1702, a to pomocí zákmitů na zdroji napájení, jelikož CEC1702 nemá možnost využívat externí zdroj hodinového signálu. Útok byl proveden na dvě verze RSA-CRT. První verze odpovídá implementaci pro STM32F3. Druhá verze využívá hardwarový akcelerátor kryptografických operací. Útok byl na obě verze implementace úspěšný. V rámci práce je nastíněn možný postup, jakým směrem vzniklé implementace a realizované útoky injekcí chyb rozšířit. V příloze práce lze nalézt vytvořený návod pro programování kryptoprocesoru a dokumentaci k firmware.

Klíčová slova CEC1702, Paillierův kryptosystém, RSA-CRT, útok pomocí injekce chyb, útok postranními kanály, zabezpečení vestavných systémů, kryptoanalýza

Abstract

The bachelor thesis focuses on working with the CEC1702 cryptoprocessor. The first part of the thesis is oriented on programming the cryptoprocessor, modification of existing implementation of the library for operating on large numbers, and implementation of the Paillier cryptosystem for the mentioned cryptoprocessor. The thesis also includes development of firmware that allows the use of the RSA-CRT encryption algorithm and the Paillier cryptosystem with variable key length. The second part of the thesis focuses on fault injection attacks on the RSA-CRT encryption algorithm. Power supply glitches and clock source glitches used are generated using the ChipWhisperer suite.

The attacks are first successfully performed on the STM32F3 microcontroller, where RSA-CRT is implemented using the library for operating on large numbers from the first part of the thesis. Using the knowledge from the attacks on the STM32F3, a further set of attacks was designed for the CEC1702 cryptoprocessor, using only power supply glitches, as the CEC1702 does not have the ability to use an external clock source. The attack was performed on two versions of RSA-CRT. The first version corresponds to the implementation for the STM32F3. The second version uses a hardware accelerator for cryptographic operations. The attack was successful on both versions of the implementation. This thesis outlines a possible approach to extend the created implementations and the implemented fault injection attacks. Created manual for programming the cryptoprocessor and the firmware documentation can be found in the appendix of the thesis.

Keywords CEC1702, Paillier cryptosystem, RSA-CRT, fault injection attack, side-channel attacks, embedded security, cryptoanalysis

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je potřeba zabezpečit osobní data i data, která jsou zpracovávána méně výkonnými vestavnými zařízeními, jež se mohou nacházet například v chytré domácnosti nebo v průmyslu. Často tato zařízení, kvůli jejich nízkému výpočetnímu výkonu, posílají data k následnému zpracování třetím stranám, kdy je potřeba zajistit, aby nebylo možné data zneužít. Z tohoto důvodu používají tato zařízení různé šifrovací protokoly, aby data utajila. Procesory, které se v dnešní době používají ve vestavných systémech bývají vybaveny kryptografickým koprocesorem, který urychluje kryptografické operace (zejména operace využívané v asymetrické kryptografii).

Pro účely této práce byl vybrán CEC1702, protože to je procesor s kryptografickým koprocesorem, u kterého není nutné podepisovat NDA a je k dispozici i v jednotlivých kusech (a ne v objemech po desítkách kusů). Cíl práce je možné rozdělit na dvě na sebe navazující částí. Cílem první části je upravit implementaci Paillierova kryptosystému [1] spolu s podpůrnou knihovnou pro práci s velkými čísly [2] s možností využití hardwarového akcelerátoru kryptografických operací pro zmíněné zařízení. Cílem druhé části je prozkoumat, do jaké míry je CEC1702 odolný vůči útokům postranními kanály v porovnání s mikrokontrolérem STM32F3, a to za použití sady ChipWhisperer [3]. Útoky jsou prováděny pomocí injekce chyb na šifrovací algoritmus RSA-CRT [4], který využívá podobných matematických principů jako Paillierův kryptosystém. Útoky jsou realizovány na čistě softwarovou implementaci na STM32F3 i CEC1702 a dále na implementaci využívající hardwarový akcelerátor kryptografických operací na CEC1702.

Výsledná úprava implementace Paillierova kryptosystému by mohla být v budoucnu využita jako alternativa k jiným kryptoschématům, která již na vestavných zařízeních implementována jsou. Výsledky z útoků postranními kanály odhalují, jestli kryptoprocesor neobsahuje základní nedostatky v návrhu hardware a také tato část práce demonstruje univerzálnost sady ChipWhisperer pro realizaci samotných útoků.

Tato bakalářská práce navazuje na již implementované knihovny z diplomové práce *Implementace a vyhodnocení efektivity schématu VeraGreg na nízkonákladovém mikrokontroléru* studenta FIT ČVUT, Ing. Jana Říhy [5].

Text práce je rozdělen do šesti samostatných kapitol. První kapitola obsahuje teoretické základy asymetrické kryptografie, homomorfního šifrování, RSA, Paillierova kryptoschématu a také útoků postranními kanály. Následně na tuto část navazuje analýza možných vývojových přípravků obsahujících CEC1702, možností, jak na CEC1702 vyvíjet *firmware* a jaké jsou možnosti pro provádění útoků postranními kanály. Třetí kapitola se zabývá realizací úprav jednotlivých implementací pro CEC1702 a vytvoření *firmware*, který umožňuje jednodušší komunikaci sady ChipWhisperer s CEC1702. Na realizaci navazuje její testování, které je popsáno v kapitole čtvrté. Útoky postranními kanály jsou detailně popsány v páté kapitole. Jelikož se jedná o obsahově velkou část z celkové bakalářské práce, je téma strukturováno do samostatné kapitoly. V poslední kapitole jsou nastíněny další možnosti, na které by se dalo zaměřit v navazujících pracích.

Kapitola 1

Úvod do problematiky

Tato kapitola obsahuje popis kryptografie s veřejným klíčem (neboli také asymetrická kryptografie) a její zástupce – šifru RSA a modifikaci RSA-CRT. Také je zde popsáno homomorfní šifrování a s tím související Paillierův kryptosystém. V závěru kapitoly jsou sepsány informace o útocích postranními kanály.

1.1 Kryptografie s veřejným klíčem

V kryptografii existují dva principy šifrování. První staví na tom, že obě strany (odesílatel i příjemce) používají stejný – soukromý – klíč. Jedná se o symetrickou kryptografii a mezi zástupce symetrické kryptografie patří například šifra **AES** (Advanced Encryption Standard) [6]. Druhý princip využívá klíčů dvou – tajného a veřejného – kdy ten veřejný je využíván na šifrování a soukromý na dešifrování zpráv. V tomto případě se jedná o asymetrickou kryptografii neboli kryptografii s veřejným klíčem. Její zástupci jsou popsáni níže v kapitole.

Kryptografie s veřejným klíčem nám umožňuje nejen zajistit důvěrnost – utajení obsahu zprávy, ale je také využívána u elektronických podpisů. Poskytuje tedy zajištění autenticity – ověření identity podepsaného – a nepopiratelnost – nemožnost popření podpisu.

Výhoda asymetrické kryptografie je, že není třeba zařizovat výměnu tajného klíče mezi stranami, které mezi sebou chtějí komunikovat. Nevýhodou oproti symetrickým šifrám může být výpočetní náročnost operací šifrování. Na výpočetní náročnosti ale staví i zabezpečení asymetrické kryptografie, kdy v současné době neexistují algoritmy, které by byly schopny dopočítat soukromý klíč z veřejného v polynomiálním čase. Více detailů o asymetrické kryptografii lze nalézt v [7, 8], odkud byly čerpány i předchozí informace a také informace pro celou následující podsekci o RSA.

1.1.1 RSA

RSA (iniciály autorů šifry – Rivest, Shamir, Adleman) je asymetrická bloková (data se šifrují po blocích určité délky) šifra publikovaná v roce 1977 [4]. RSA neslouží jako náhrada symetrických šifer. Důvodem je, že šifrování zabírá řádově více času než u šifer blokových. Šifra se přesto stále hojně využívá například u elektronických podpisů či pro přenos malých částí dat, jakými může být výměna soukromých klíčů pro následné použití symetrického šifrování.

Bezpečnost RSA staví na problému faktorizace, kdy násobení dvou prvočísel je výpočetně jednoduché. Následné rozložení vzniklého čísla je ale nemožné v polynomiálním čase bez znalosti alespoň jednoho prvočísla, ze kterého číslo vzniklo. Pokud se však použijí malá čísla (desítky až stovky bitů), lze faktorizaci i tak provést. Proto se u RSA považuje za bezpečné, když jsou klíče dlouhé tisíce bitů – zejména 1024, 2048 a 4096 bitů. Tak dlouhé klíče jsou důvodem, proč trvá šifrování řádově déle, než je tomu například u AES, který pracuje s bloky dlouhými 128 bitů a nestaví na utajování soukromých klíčů díky výpočetní náročnosti.

1.1.1.1 Šifrování a dešifrování

RSA šifrování a dešifrování využívá modulární aritmetiku, zejména modulární mocnění. Před samotným šifrováním je nutno vygenerovat soukromý a veřejný klíč. Pro vytvoření klíče délky k je nejprve třeba zvolit dvě prvočísla – p a q, každé délky $k/2$ bitů. Číslo $n = p.q$, které se následně využívá jako modulo při výpočtech u šifrování a dešifrování, je součástí soukromého i veřejného klíče. Pro vytvoření dalších částí klíčů je nutné vypočítat Eulerovu funkci čísla n: $\Theta(n) = (p-1)(q-1)$. Následně je možné zvolit veřejný exponent $e \in \{1, 2.., \Theta(n)-1\}$, kdy musí platit $\gcd(e, \Theta(n)) = 1$. Je vhodné volit číslo s nízkým počtem bitů a malou Hammingovou váhou (co nejmenší počet „jedničkových“ bitů v binárním zápisu čísla), aby samotné šifrování bylo co nejrychlejší. Soukromý exponent d je vytvořen následovně: $d \equiv e^{-1} \bmod \Theta(n)$. Veřejný klíč K_V následně tvoří dvojice (e, n) a soukromý klíč K_T dvojice (d, n).

RSA s k-bitovým klíčem bere k-bitový text k zašifrování jako sekvenci bitů reprezentující číslo m. Následně je možné provést zašifrování $E_{K_V}(m)$ s využitím veřejného klíče:

$$E_{K_V}(m) = c = m^e \bmod n$$

Pro úspěšné šifrování musí být m tak dlouhé, aby se při mocnění e provedla redukce modulo n alespoň jednou. Dešifrování $D_{K_T}(c)$ lze provést obdobným způsobem s využitím klíče soukromého:

$$D_{K_T}(c) = m = c^d \bmod n$$

Při vytváření elektronických podpisů je využit soukromý klíč na podepisování a veřejný klíč na ověřování podpisu.

1.1.1.2 Modifikace RSA-CRT

Jak bylo popsáno v předchozí podsekci, RSA pracuje s obřími čísly a šifrování lze urychlit správnou volbou veřejného exponentu. Výpočty obsahující soukromý exponent jdou urychlit také. Protože se zde už nekladou požadavky na podobu samotného exponentu, které by oslabovaly bezpečnost šifry zmenšováním množiny použitelných klíčů, jsou kladeny na samotný algoritmus šifry. Šifrování i dešifrování se skládá hlavně z modulárního umocňování, které má složitost $O(k^3)$. Vznikla proto modifikace RSA využívající **CRT** (Chinese Remainder Theorem, neboli Čínská věta o zbytcích) [9], která se zaměřuje na zmíněnou slabinu.

U RSA-CRT zůstává veřejný klíč stejný jako u základního RSA, ale mění se dvojice soukromého klíče na pětici, která se skládá z nesledujících prvků:

- První zvolené prvočíslo p délky $k/2$ bitů,
- druhé zvolené prvočíslo q délky $k/2$ bitů,
- $d_p = d \bmod (p-1)$,
- $d_q = d \bmod (q-1)$,
- $q_{inv} = q^{-1} \bmod p$ nebo $p_{inv} = p^{-1} \bmod q$.

Použití veřejného klíče se také oproti základní verzi RSA nemění. Použití soukromého klíče je ale rozloženo do několika mezivýpočtů. Následující vztahy vyjadřují proces dešifrování c na zprávu m. Je také možné klíče u operací zaměnit a podepisovat zprávu soukromým klíčem, poté by se zaměnily i pozice c a m.

Nejprve se redukuje c na hodnoty $c_p = c \bmod p$ a $c_q = c \bmod q$. Následuje výpočet $m_1 = {c_p}^{d_p} \bmod p$ spolu s $m_2 = {c_q}^{d_q} \bmod q$. Poté se již výpočet liší, podle toho, zda je využito q_{inv} nebo p_{inv}.

1. Výpočet při použití q_{inv}:
   a) vypočte se $h = (q_{inv}(m_1 - m_2)) \bmod p$;
   b) vypočte se $m = m_2 + hq$.
2. Výpočet při použití p_{inv}:
   a) vypočte se $h = (p_{inv}(m_2 - m_1)) \bmod q$;
   b) vypočte se $m = m_1 + hp$.

Modulární mocnění se objevuje ve výpočtech m_1 a m_2. Při výpočtu se pracuje s čísly s poloviční délkou než je tomu u základního RSA (dp, dq, p, q mají délku $k/2$ bitů). Modulární mocnění má kubickou složitost, a tedy při počítání s čísly poloviční délky je až osmkrát rychlejší ($O(\frac{1}{2})^3 = O(\frac{1}{8})$). Jelikož se ale tato operace vyskytuje ve výpočtech dvakrát, zrychlení je tedy

čtyřnásobné při sekvenčním výpočtu. Protože jsou na sobě operace nezávislé, lze je paralelizovat a opět se přiblížit až k osminásobnému zrychlení oproti základnímu RSA. RSA-CRT sice obsahuje i výpočty další, ty jsou ale složitostí a časovou náročností oproti modulárnímu mocnění zanedbatelné.

1.2 Homomorfní šifrování

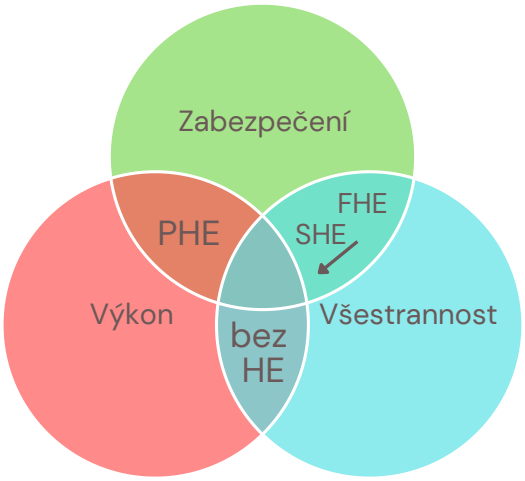
V dnešní době je kladen důraz na co největší soukromí, ale mnoho osobních dat uživatelů je vystaveno třetím stranám – nejčastěji *cloudovým* službám – kvůli potřebě jejich zpracování či uchování. Právě homomorfní šifrování umožňuje nechat si data zpracovávat třetími stranami, aniž by věděly, co tato data obsahují.

Homomorfní šifrování dovoluje provádět matematické operace nad zašifrovanými daty stejně jako nad nezašifrovanými. Výsledek matematických operací neprozrazuje před dešifrováním žádné tajné informace. Po dešifrování tajným klíčem je získán výsledek, který je roven hodnotě po provedení stejných operací nad daty nezašifrovanými.

PHE Pokud nad zašifrovanými daty může být prováděna pouze jedna operace – například jen sčítání, nebo jen násobení – jedná se o **PHE** (Partially Homomorphic Encryption – do češtiny přeloženo jako „Částečně homomorfní šifrování“). Podle podporované operace se mohou také používat názvy Aditivně homomorfní šifrování či Multiplikativně homomorfní šifrování. Schémata pro PHE již v dnešní době existuji a jsou použitelná. Jako příklad lze uvést algoritmus ElGamal [10] či Paillierův kryptosystém, který se řadí mezi aditivně homomorfní a je popsán v následují podsekci.

FHE Vzhledem k tomu, že často při zpracování dat není použita pouze jedna operace, ale kombinace operací a jejich inverzí, bývá PHE nedostatečné a přichází na řadu FHE (Fully Homomorphic Encryption – česky „Plně homomorfní šifrování“). Pokud se tedy nad zašifrovanými daty dají provádět operace sčítání a násobení zároveň, a to bez omezeného počtu, jedná se o FHE. Jako reálné příklady lze uvést BFV kryptosystém [11] a BGV kryptosystém [12].

SHE Poslední kategorií, která se všestranností spíše blíží FHE, je SHE (Somewhat Homomorphic Encryption – do češtiny volně přeloženo jako „Do jisté míry homomorfní šifrování“). Nad zašifrovanými daty je podporováno více typů operací, ale je limitován jejich počet opakování. Většinou je omezeno násobení na řádově jednotky až nízké desítky možných opakování. Jako příklad lze uvést CKKS [13].



Obrázek 1.1: Vennův diagram znázorňující provázanost vlastností a typů homomorfního šifrování. Jako zdroj pro tvorbu byla využita [14]

Porovnání variant V současné době jsou mezi PHE a SHE spolu s FHE rozdíly ve všestrannosti i ve výpočetním výkonu. Musí se tedy balancovat mezi těmito vlastnostmi spolu s mírou zabezpečení, a to v závislosti pro jakou aplikaci je homomorfní šifrování využito. Obrázek 1.1 znázorňuje pomocí Vennova diagramu, jak jsou tři zmíněné vlastnosti provázané s typy homomorfního šifrování. Jelikož se neustále toto odvětví posouvá dopředu a FHE či SHE systémy jsou stále efektivnější, lze tvrdit, že postupem času dosáhnou i středu diagramu.

Detailnější popis homomorfního šifrování a porovnání PHE a FHE lze nalézt například v 5. kapitole *The Cloud Security Ecosystem* [14], ze které byly čerpány informace pro tuto sekci. Kromě tohoto zdroje byl využit i [15], zejména pro sekci o SHE.

1.2.1 Paillierův kryptosystém

V roce 1999 zveřejnil Pascal Paillier popis pravděpodobnostního asymetrického šifrovacího algoritmu založeného na problému zbytkových tříd n-tého řádu (hodnota n popsána dále v textu). Algoritmus je nyní nazýván jako Paillierův kryptosystém (dále jen Paillier) [1] a má podobnou výpočetní náročnost jako RSA. Reálnou implementaci Pailliera například využívá diplomová práce *Implementace a vyhodnocení efektivity schématu VeraGreg na nízkonákladovém mikrokontroléru* [5] jako součást schématu VeraGreg [16].

Paillier existuje ve dvou verzích. První verze je ovlivněna volbou náhodného parametru g již při generování klíčů. Druhá verze volí $g = n + 1$ a generování náhodného parametru je využito až při šifrování. Tím se zjednodušuje výpočet parametrů klíče a následně i šifrovací a dešifrovací proces. Zde je

popsána pouze druhá verze, která je jednodušší a je využita v implementaci popsané v sekci 3.2. Detailní popis první varianty lze nalézt v dříve citovaném originálním článku autora schématu [1].

1.2.1.1 Šifrování a dešifrování

Před samotným šifrováním je opět nutné vygenerovat klíče. Tento proces se skládá z několika kroků, které se v určitých částech podobají generování klíčů pro RSA. Kromě toho se podobají i doporučené délky modula v řádech tisíců bitů – 1024, 2048, 4096, ... – aby bylo v polynomiálním čase výpočetně nemožné dopočítat tajné části klíče.

Nejdřív se zvolí dvě prvočísla p a q stejné délky $k/2$ a vypočítá se $n = p \cdot q$, kdy $|n| = k$. Následně se dopočítá λ, která je v jednodušší variantě rovna Eulerově funkci [17] čísla n, tedy $\lambda = \Theta(n) = (p-1)(q-1)$ a $\mu = \lambda^{-1} \bmod n$. Veřejným klíčem K_V se stává pouze n a tajným klíčem K_T se stává dvojice (λ, μ).

Pro následné šifrování $E_{K_V}(m, r)$, kdy $0 \leq m < n$, je třeba zvolit náhodně vygenerované číslo r, kdy $0 < r < m$. Šifrování je poté dáno vztahem:

$$E_{K_V}(m, r) = c = (1 + n \cdot m) r^n \bmod n^2$$

Dešifrování $D_{K_T}(c)$, kdy $c \in \mathbb{Z}^*_{n^2}$, je definováno vztahem:

$$D_{K_T}(c) = m = \frac{(c^\lambda \bmod n^2) - 1}{n} \cdot \mu \bmod n$$

1.2.1.2 Modifikace využívající CRT

Následují popis čerpá z [15, 1]. Pro urychlení výpočetně nejnáročnějších částí, kterou je u šifrování výpočet $r^n \bmod n^2$ a u dešifrování $c^\lambda \bmod n^2$, lze opět využít CRT (jako u modifikace RSA). U obou variant je ale nutné znát obě původní prvočísla p a q.

1. Urychlení šifrování $E_{K_V}(m)$:
   a) Vypočte se p^2, q^2 a $q^2_{inv} \equiv (q^2)^{-1} \bmod p^2$, kdy $|p^2| = |q^2| = \frac{|n^2|}{2}$.
   b) Vypočte[1] se $r_{p^2} = r^n \bmod p^2$ a $r_{q^2} = r^n \bmod q^2$.
   c) Využitím CRT se vypočte $h_r = (q^2_{inv} \cdot ((r_{p^2} - r_{q^2}) \bmod p^2))$ a následně $r_{n2} = r_{q^2} + h_r \cdot q^2$.
   d) Posledním krokem je vytvoření $c = (1 + n \cdot m) r_{n^2} \bmod n^2$.

[1] Náhodně vygenerované číslo r není třeba redukovat p^2 nebo q^2, jelikož platí následující vztahy: $|r| \leq |n|$ a $|p^2| = |q^2| = |n|$

2. Urychlení dešifrování $D_{K_T}(c)$:

   a) Vypočte se p^2, q^2 a $q_{inv}^2 \equiv (q^2)^{-1} \bmod p^2$, stejně jako u šifrování.

   b) Zredukuje se c, tedy $c_p = c \bmod p^2$ a $c_q = c \bmod q^2$, jelikož $|c| = 2 \cdot |p^2|$ a $|c| = 2 \cdot |q^2|$.

   c) Vypočte se $\lambda_p = \lambda \bmod (p \cdot (p-1))$ a $\lambda_q = \lambda \bmod (q \cdot (q-1))$.

   d) Vypočte se $c_{\lambda_p} = c_p^{\lambda_p} \bmod p^2$ a $c_{\lambda_q} = c_q^{\lambda_q} \bmod q^2$.

   e) Využitím CRT se vypočte $h_c = (q_{inv}^2 \cdot ((c_{\lambda_p} - c_{\lambda_q}) \bmod p^2))$ a následně $c_\lambda = c_{\lambda_q} + h_c \cdot q^2$.

   f) Posledním krokem je vytvoření $m = \frac{c_\lambda - 1}{n} \cdot \mu \bmod n$.

1.2.1.3 Aditivně homomorfní šifrování

Jak už bylo zmíněno dříve, Paillier spadá do kategorie Homomorfních šifrovacích systémů, přesněji jde o aditivně homomorfní kryptosystém. Při znalosti pouze veřejného klíče K_V a zašifrovaných zpráv lze vypočítat jejich součet nad tělesem $\mathbb{Z}_n$ a jeho opravdovou hodnotu získat až po dešifrování soukromým klíčem. Mějme tedy dvě zprávy m_1, m_2, dvě náhodně vygenerovaná čísla r_1, r_2 a $k \in \mathbb{N}$, potom platí následující vztahy:

$$D_{K_T}(E_{K_V}(m_1, r_1) \cdot E_{K_V}(m_2, r_2) \bmod n^2) = (m_1 + m_2) \bmod n$$

$$D_{K_T}(E_{K_V}(m_1, r_1)^k \bmod n^2) = (k \cdot m_1) \bmod n$$

$$D_{K_T}(E_{K_V}(m_1, r_1)^{m_2} \bmod n^2) = D_{K_T}(E_{K_V}(m_2, r_2)^{m_1} \bmod n^2) = (m_1 \cdot m_2) \bmod n$$

1.3 Útoky postranními kanály

Útoky postranními kanály lze definovat jako útoky, které se zaměřují přímo na systémy s implementací kryptografických algoritmů. Snaží se zneužít jakoukoliv výměnu informace systému s okolím, která nesouvisí s normální funkcí systému (například průběh spotřeby systému). Útoky tedy využívají slabin softwarové i hardwarové implementace kryptografických algoritmů. Zmíněným systémem může být čip vestavného zařízení, ve kterém je nahrán program šifrující zpracovávaná data před odesláním přes síť.

Útoky lze rozdělit na invazivní, semi-invazivní a neinvazivní. Invazivní útoky vyžadují odpouzdření čipu, odstranění pasivační vrstvy a připojení k vnitřním obvodům (např. pro přímý přístup ke sběrnici či zdroji hodinového signálu). Semi-invazivní útoky vyžadují odpouzdření čipu, ale nevyžadují připojení k vnitřním obvodům a je využíváno pouze vnějších jevů (např. elektromagnetického či rentgenového záření). Neinvazivní útoky využívají pouze vnějších jevů (např. elektromagnetického záření či průběhu spotřeby zařízení).

Dále lze útoky dělit na aktivní a pasivní. Aktivní útoky se konkrétním způsobem snaží ovlivnit vstup (příp. výstup) dat nebo chování celého systému

(např. pomocí injekce chyb na zdroj hodinového signálu). Tento typ útoku je detailněji popsán dále v následujících podsekcích. Pasivní útoky využívají pouze pozorování fyzikálních projevů konkrétního zařízení (např. odposlech zařízení, měření spotřeby). Jako příklad lze uvést **SPA** (*Simple Power Analysis* – jednoduchá odběrová analýza) [18] pro získání tajného klíče u základní implementace RSA či **CPA** (*Correlation Power Analysis* – diferenciální odběrová analýza) [19] pro získání tajného klíče u implementace šifry AES.

Zmíněná rozdělení jsou nezávislá. Útok postranním kanálem může být invazivní a zároveň pasivní a naopak neinvazivní a zároveň aktivní. Současně na sebe jednotlivé útoky mohou navazovat. Invazivní útok může být využit jako příprava pro následný neinvazivní útok, kdy pomůže odhalit architekturu čipu pro následné cílené připojení elektrod pro úspěšný odposlech.

Informace pro předchozí odstavce této sekce a tak i pro části následujících podsekcí byly čerpány z [20, 21].

1.3.1 Útok pomocí zákmitů na zdroji hodinového signálu

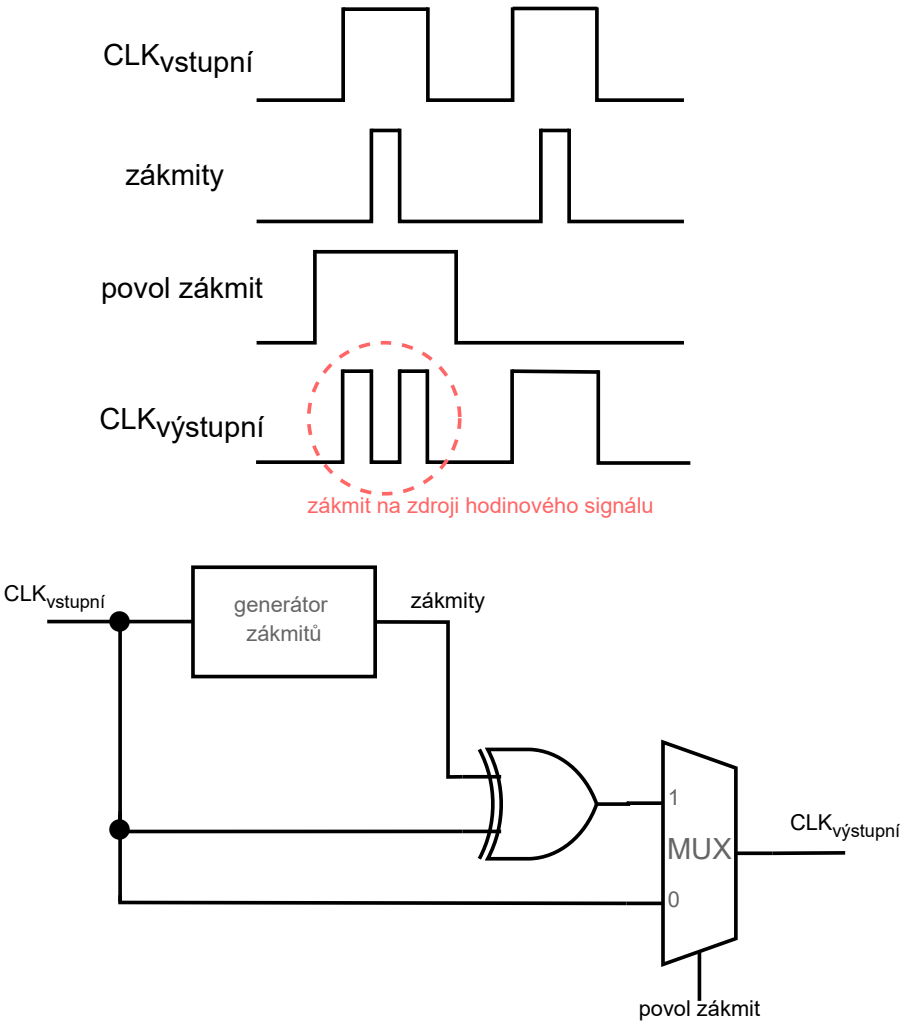
Útok pomocí zákmitů na zdroji hodinového signálu je zástupcem aktivních útoků postranními kanály. Pokud má čip externí zdroj hodin, není třeba provést jeho odpouzdření. V tomto případě se jedná o útok neinvazivní. Pokud je zdroj hodinového signálu součásti čipu, odpouzdření a případná analýza architektury čipu je nutná. V tomto případě tak jde o útok invazivní.

Číslicové systémy (čipy) téměř vždy spoléhají na stabilní hodinový signál. Pokud se však do zdroje hodinového signálu zanesou zákmity, systém může některé instrukce přeskočit či zpracovat chybně. To je způsobeno náhlým nedodržením časování jednotlivých operací, kdy je časování nutné vždy dodržet kvůli zpožděním průběhu signálu v obvodu. Právě chybného chování se snaží útoky pomocí zákmitů na zdroji hodinového signálu docílit, protože vzniklé chyby se následně dají využít například pro dopočítání tajných klíčů šifry, na jejíž implementaci se útočí.

Pro generování zákmitů se může využít původní zdroj hodinového signálu, od kterého se například fázovými posuny a logickými operacemi `xor` odvodí jednotlivé zákmity. Vzniklé zákmity jsou následně vloženy do hodinového signálu, který následně využije systém, na nějž se útočí [22]. Toto generování zákmitů je znázorněno na Obrázku 1.2

1.3.2 Útok pomocí zákmitů na zdroji napájení

Útok pomocí zákmitů na zdroji napájení je také zástupcem aktivních útoků postranními kanály. Jelikož je zdroj napájení přístupný ve většině případů bez nutného odpouzdření, útok proto bývá neinvazivní. Může být zvolen jako alternativa k útoku pomocí zákmitů na zdroji hodinového signálu, kdy má právě čip nepřístupný zdroj hodinového signálu.



Obrázek 1.2: Ukázka zákmitu na zdroji hodinového signálu

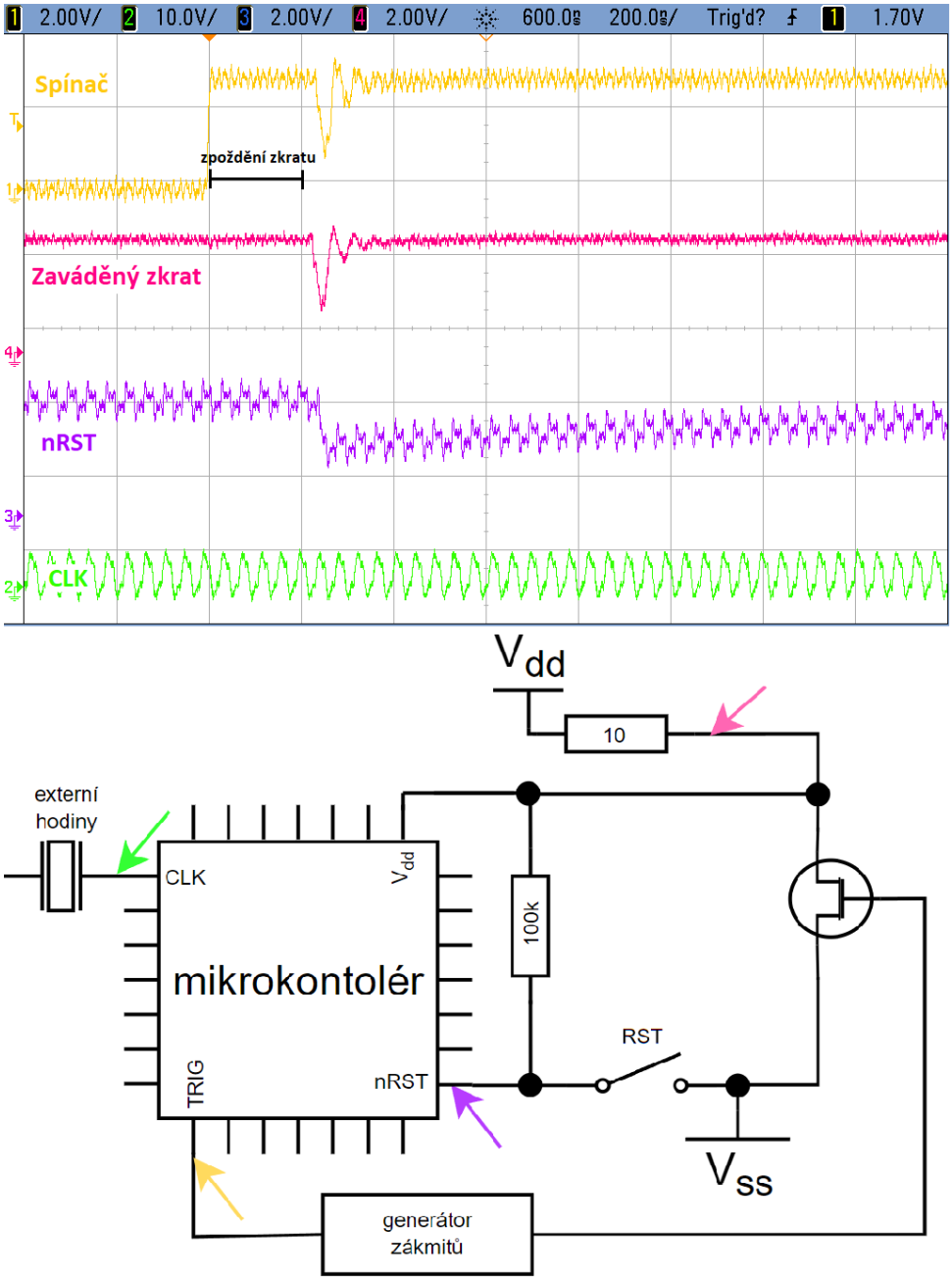
Útok spočívá ve zkratování zdroje napájení ve zvoleném okamžiku na určitou dobu, a to za účelem ovlivnit chování číslicového systému, který může díky sníženému napájení generovat chyby ve výpočtech. Příklad zákmitu na napájení lze pozorovat na Obrázku 1.3. Zmíněný obrázek zachycuje průběh napětí na několika vývodech zařízení, na které je útok prováděn. Obrázek 1.3 také obsahuje zjednodušené schéma možného provedení útoku.

Poměr doby, po jakou může být zákmit (neboli zkrat na napájení) aktivní, k době, kdy je napájení bez zkratu, závisí například na parametrech předřadného (anglicky „*shunt*“) rezistoru. Jakmile je napájení zkratováno, pak je předřadný rezistor jednou nožičkou připojený na napájecí napětí a druhou nožičkou na zem. V ten moment je na něm celé napájecí napětí V_{dd}, a tudíž je na něm ztrátový výkon $P = V_{dd} \cdot I = V_{dd} \cdot \frac{V_{dd}}{R} = \frac{V_{dd}^2}{R}$. Pokud by byly zvoleny hodnoty $V_{dd} = 3,3\ V$ a $R = 10\ \Omega$, potom je ztrátový výkon $P(t) = 3,3^2/10 \simeq 1$ W. Jedná se zároveň o okamžitý tepelný výkon. Rezistor se tedy začne zahřívat. Pokud překročí kritickou teplotu, spálí se a může způsobit zničení celého obvodu. Pokud se ale zahřívání včas vypne – skončí zkrat – začne se rezistor zase ochlazovat obyčejnou konvekcí (ochladí se do okolního vzduchu apod.). Jakmile je dostatečně ochlazen, může se zkrat zopakovat. V katalogu se uvádí zpravidla trvalý průměrný výkon, jaký rezistor snese (do vzduchu se stačí vyzářit více tepla, nežli kolik ho rezistor vyprodukuje). U SMD rezistorů to bývá typicky 0,12 W. Pokud tak budou pulsy dostatečně krátké (aby se během pulsu nedosáhlo kritické teploty), pak v případě, kdy je tepelný výkon 1 W (viz výpočet dříve), může být poměr časů zkratováno (topí) : nezkratováno (ochlazuje se) až 1 : 7 [23].

Překážkou pro použití této metody útoku můžou být obvody pro vyrovnávání napájecího napětí v čipu. Útok je také závislý na použitém hardware. Výsledný zkrat ovlivňuje pouhá změna délky kabelu, kterým je propojen generátor zákmitů a napájení čipu [24].

1.3.3 Útok na RSA-CRT

Pro získání tajného klíče (tedy tajného exponentu) u RSA-CRT je využíváno zanesení chyby do výpočtu jedné z hodnot m_1 a m_2. Toho se dá docílit například tím, že se použije útok pomocí zákmitů na zdroji napájení nebo na zdroji hodinového signálu. Následný výpočet tajného klíče je možný několika způsoby. Jedním z nich je Bellcore útok [25], který je popsán dále. Nadstavbou tohoto útoku je Lenstrův útok, který funguje na podobných matematických principech využívající pouze data dešifrovaná se zanesenou chybou. Tyto informace byly převzaty z [26]. Ze stejného zdroje bylo čerpáno i pro následující popis útoku.



Obrázek 1.3: Ukázka zákmitu na zdroji napájení

Prerekvizity pro úspěšný útok:

- útok je prováděn na implementaci RSA, která pro dešifrování či generování podpisů využívá CRT;
- je možné injektovat (zanášet) dočasné chyby do zařízení;
- je možné provést dešifrování či generování podpisů se zanesenou chybou a také bez ní;
- je znám veřejný klíč $K_V = (e, n)$.

Průběh útoku a výpočtu tajného klíče, kdy je CRT využito pro generování podpisu. Zpráva m se tedy šifruje tajným klíčem K_T:

1. zařízení vygeneruje podpis bez zanesené chyby:

$$s = ((q \cdot q_{inv}) \cdot s_1 + (p \cdot p_{inv}) \cdot s_2) \bmod n,$$

$$\text{kdy } s_1 = m^{d_p} \bmod p \text{ a } s_2 = m^{d_q} \bmod q$$

2. zařízení vygeneruje podpis znovu, tentokrát ale se zanesenou chybou při výpočtu s_1:

$$s^{'} = ((q \cdot q_{inv}) \cdot s_1^{'} + (p \cdot p_{inv}) \cdot s_2) \bmod n$$

3. následuje výpočet prvočísla q:

$$\begin{aligned} s - s^{'} &\equiv (q \cdot q_{inv}) \cdot s_1 - (q \cdot q_{inv}) \cdot s_1^{'} \bmod n \\ &\equiv (q \cdot q_{inv}) \cdot (s_1 - s_1^{'}) \bmod n \\ \gcd(s - s^{'}, n) &= \gcd((q \cdot q_{inv}) \cdot (s_1 - s_1^{'}), p \cdot q) \\ &= q \cdot \gcd((q \cdot q_{inv}) \cdot (s_1 - s_1^{'}), p) \\ &= q \cdot 1 = q \end{aligned}$$

4. jakmile je získáno q, lze snadno dopočítat $p = \frac{n}{q}$;
5. tajný exponent d lze vypočítat stejně jako při generování tajného klíče u RSA, využitím Eulerovy věty a modulární inverze, tedy $d = e^{-1} \bmod \Theta(n)$.

Kapitola 2

Analýza

Tato kapitola se v první části zaměřuje na kryptoprocesor CEC1702 (dále jen CEC1702) a možné přístupy k práci s tímto zařízením. Druhá část kapitoly se zabývá útoky postranními kanály, zejména jejich možností realizace. Třetí část se zabývá možnostmi programování CEC1702.

2.1 Kryptoprocesor CEC1702

Pro bakalářskou práci byl zvolen CEC1702 [27] zejména z následujících důvodů:

1. CEC1702 obsahuje zabezpečený hardwarový akcelerátor kryptografických operací. Zejména se jedná o RSA / RSA-CRT akcelerátor důležitý pro implementaci Pailliera, k urychlení modulárního mocnění.

2. Zařízení lze objednat i po jednotlivých kusech a není třeba vytvářet objednávky pro desítky kusů najednou.

3. Pro CEC1702 existuje detailnější dokumentace k hardwarovému (dále jen **HW**) akcelerátoru kryptografických operací bez nutnosti podepsat **NDA** (Dohoda o mlčenlivosti).

Další výhodou CEC1702, alespoň v době rozhodování, se zdála existence **IDE** (vývojové prostředí), které obsahuje **API** (Programové rozhraní) pro práci s HW akcelerátorem. To, že právě mikroC IDE od firmy MikroEletronika [28] bude při vývoji největší překážka, se ukázalo až po koupi HW. Tato problematika je popsána samostatně, zejména v kapitole 3.

2.1.1 Software pro vývoj firmware

Jelikož první kroky s mikroC IDE byly velmi bolestné, bylo rozhodnuto o krátkém průzkumu. Jeho cílem bylo zjistit, zda přece jen neexistuje konkurence,

která by obsahovala většinu funkcionalit mikroC IDE a zároveň by měla přívětivější uživatelské rozhraní. MikroC IDE bylo totiž naposledy aktualizováno výrobcem v roce 2019. CEC1702 se ale nemůže řadit mezi jiné mikrokontroléry a vestavné procesory co se známosti a rozšířenosti týče. Neexistuje proto mnoho IDE, které by obsahovaly kompilátor pro překlad do zdrojového kódu či dokonce API pro práci s periferiemi zařízení bez nutnosti jejich konfigurace přes registry zařízení.

V rámci průzkumu bylo nalezeno pouze jedno další IDE – MPLAB® X IDE [29]. Toto IDE obsahuje *plugin* podporující překlad kódu pro CEC1702 a autor se s IDE již setkal v předmětu BI-VES v rámci bakalářského studia.

Přesto bylo nakonec vybráno mikroC IDE, a to hned z několika důvodů:

- Přímá podpora přípravku *Clicker2 for CEC1702*,
- možnost „*debuggování*“ programu přímo za běhu na CEC1702 (viz podsekce 2.1.2),
- existence API pro práci s periferiemi jako je UART převodník nebo **PWM** (Pulzně Šířková Modulace) modul,
- existence API pro práci s HW akcelerátorem kryptografických operací,
- již koupená licence pro mikroC IDE,
- existující implementace Paillierova krypstosystému (viz Kapitola 3) již některé funkce z API v mikroC využívá.

MPLAB® X IDE dovoluje pouze přístup k registrům přes makra a jediná výhoda tedy je zmíněná uživatelská přívětivost a větší odladěnost prostředí. Po diskuzi s vedoucím práce bylo vyhodnoceno, že implementace kryptografických operací a práce s periferiemi by byla mnohem náročnější než snažení se zachovat klidnou hlavu při práci se zastaralým IDE. Vzniklé problémy, které vzešly na povrch při práci na úpravách implementací knihovny pro práci s velkými čísly a Paillierova kryptosystému a archaizmy IDE i kompilátoru, jsou následně popsány v sekcích 3.1 a 3.2.

2.1.2 Přípravky obsahující CEC1702

Existují dva přípravky obsahující CEC1702:

1. *Clicker2 for CEC1702* [30] (dále jen zkráceně *Clicker*) od stejné společnosti jako je mikroC IDE a
2. *NewAE CW308 CEC1702 Target* [31] (dále jen zkráceně *NewAE target*), jakožto cílový modul (target) pro CW308 UFO Board [32] ze sady ChipWhisperer [3].

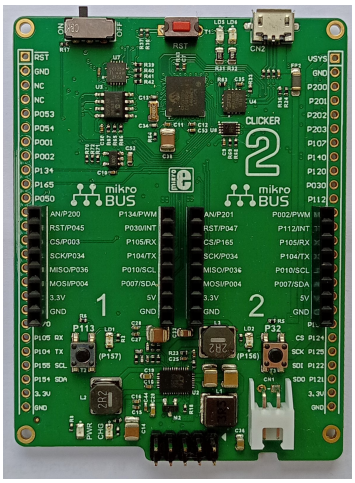
Pro rozhodnutí, která platforma se použije, bylo důležitých několik faktorů:

- Dostupnost platformy,
- možnost naprogramovat zařízení a program často měnit,
- možnost ladit program za běhu,
- funkční sériová linka pro komunikaci s CEC1702,
- vývod důležitých pinů pro možnou realizaci útoků.

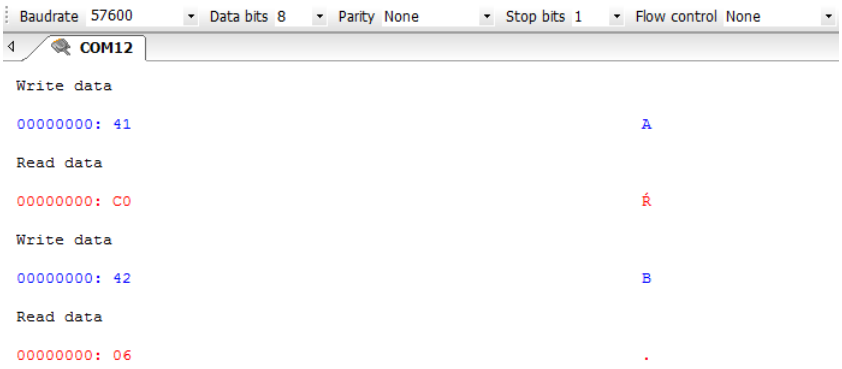
2.1.2.1 Clicker 2 for CEC1702

Clicker (viz Obrázek 2.1) byl pořízen spolu s *Clicker2 for CEC1302* [33], který byl využíván při předchozím vývoji implementací knihovny pro práci s velkými čísly a Paillierova kryptosystému [5]. Zároveň byl na půdě fakulty i HW programátor *mikroProg for CEC* [34] pro programování a ladění programů za běhu na *Clickeru*. Zvolené vývojové prostředí mikroC IDE obsahuje i sadu vzorových projektů pro práci s *Clickerem*. Avšak na rozdíl od *NewAE target* není *Clicker* přizpůsoben pro realizaci jakýchkoliv útoků.

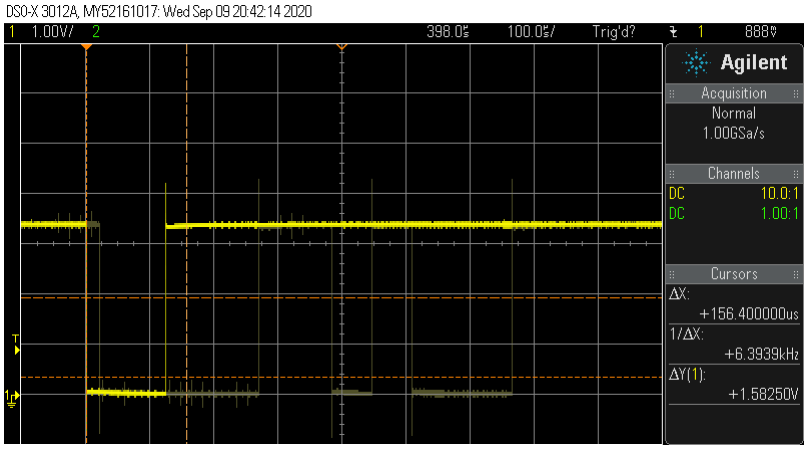
Jelikož byl *Clicker* již k dispozici, byl zvolen jako vývojová platforma. Kdyby nenastal dále popsaný problém, pro následnou realizaci útoků by se provedla analýza. Ta by vyhodnotila, zda by *Clicker* šel také využít se sadou ChipWhisperer, nebo zda by bylo nutné využít *NewAE target*. Avšak hned při zprovoznění sériové komunikace přes UART nastala při použití Clickeru komplikace. Potíž se sériovou komunikací u Clickeru a následné programování NewAE targetu byly řešeny ve spolupráci se studentkou Terezou Horníčkovou. V době vzniku této práce řešila taktéž výběr přípravku obsahující CEC1702 a jeho programování.



Obrázek 2.1: *Clicker2 for CEC1702*



Obrázek 2.2: Komunikace přes UART s přípravkem *Clicker2 for CEC1702*



Obrázek 2.3: Snímek z osciloskopu snímající odeslání znaku „A“ z přípravku *Clicker2 for CEC1702 přes UART*

Problém *Clickeru* s UART komunikací Pokud byl přes UART odeslán do *Clickeru* pouze jeden znak (například „A“ - `0x41`) při nastavení rychlosti na 57 600 Baud a následně *Clickerem* odeslán nazpět, druhá strana přijímala znak naprosto odlišný (odpovídající hodnotě `0xC0`), viz Obrázek 2.2. Byla tedy provedena analýza komunikace napojením osciloskopu na `RXD` pin USB-UART převodníku *CP2102 STC* [35]. Následným pozorováním průběhu signálu, viz Obrázek 2.3, byl vyvozen závěr, že nastavení Baudrate v programu na 57 600 odpovídá reálná rychlost cca 32 000 Baud (dále taktéž nastavených 9600 Baud odpovídá cca 4800 Baud).

Po komunikaci s produktovou podporou bylo zjištěno, že je třeba nastavit EFUSE bity, aby se správně nastavila frekvence hodinového signálu zařízení. Toto nastavení je popsáno v [36]. Po vyzkoušení tohoto postupu se ale problém nevyřešil. Byl tedy vytvořen testovací kód, ve kterém se rychlost sériové linky nastavuje manuálně pomocí registrů `TestUart.c` (na přiloženém médiu ve

Obrázek 2.4: *NewAE CW308 CEC1702 Target*

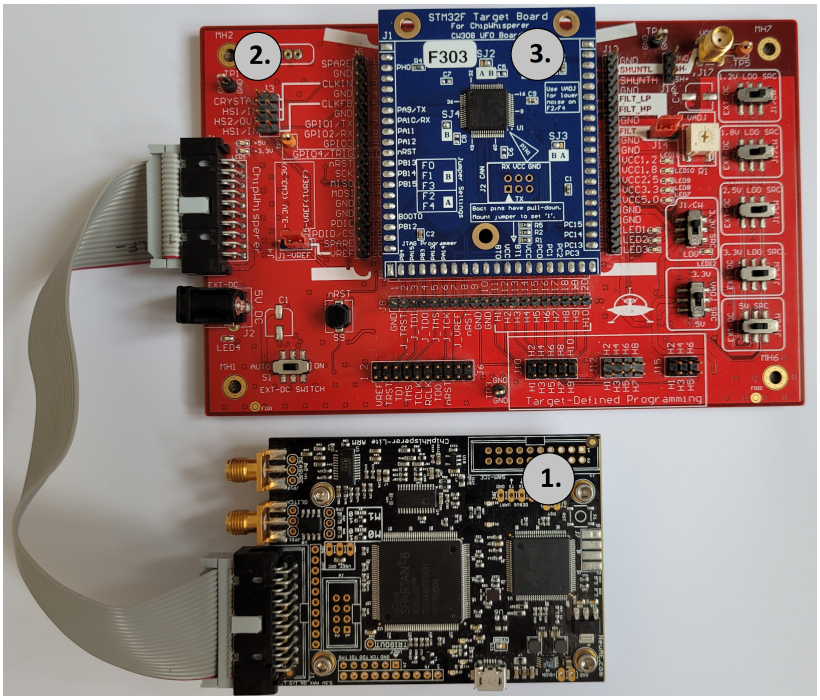
složce `0_Analýza`). Přesněji došlo k manuálnímu nastavení Baudrate přes systémové hodiny (24 MHz) a děličku. Pro nastavení Baudrate 9600 z 24MHz hodin by v děličce měl být nastaven dělitel na 156 ($24 \cdot 10^6/(156 \cdot 16) \simeq 9615$). Korektního chování UART bylo však dosaženo až po nastavení dělitele na 172 (zjištěno experimentálně). Hodnota 172 ale podle dokumentace čipu podporována není [27]. Z těchto hodnot se pak již potvrdila reálná frekvence hodin na cca 26,4 MHz. Byl tedy vyvozen závěr, že buďto byla udělána chyba v procesu nastavení pomocí EFUSE bitů, nebo byl přípravek již dodán s poškozeným zdrojem hodinového signálu.

Jelikož se používání *Clickeru* neosvědčilo z důvodu nutnosti zařízení prvně správně nakonfigurovat, což se zdá jako nadlidský úkon, byl tedy nahrazen *NewAE targetem*.

2.1.2.2 NewAE CW308 CEC1702 Target

Druhý přípravek (nebo spíše modul) *NewAE target* (viz Obrázek 2.4) byl spolu se soupravou ChipWhisperer na začátku práce objednán a nebyl ihned k dispozici. Dokumentace k *NewAE targetu* [31] obsahovala informace, že mikroC IDE jde využít k vývoji se vzorovými projekty. O kompatibilitě HW programátoru *mikroProg for CEC* se ale dokumentace nezmiňovala. Zmiňovala se pouze o využití programátorů Flash paměti. Velkou výhodou *NewAE targetu* je, že se jedná o modul do *CW308 UFO* desky [32] ze sady ChipWhisperer, která je přizpůsobena pro realizaci útoků postranními kanály, jejichž realizace je součástí zadání bakalářské práce. Detaily ohledně této sady jsou popsány v následující sekci 2.2.

Tento přípravek má správně přednastavené EFUSE bity již od výroby a není nutné další nastavování pro zprovoznění sériové komunikace přes UART. U *NewAE targetu* se ale nejprve musely analyzovat možnosti, jak do přípravku nahrát program. Tato část je popsána v sekci 2.3.



Obrázek 2.5: Sada ChipWhisperer: 1. ChipWhisperer-Lite, 2. CW308-UFO deska, 3. CW308 *target* s mikrokontrolérem STM32F3

2.2 Platforma pro realizaci útoků postranními kanály

„ChipWhisperer je kompletní sada nástrojů s otevřeným zdrojovým kódem, která slouží k poznávání útoků postranními kanály na vestavěná zařízení a zároveň k ověřování jejich odolnosti vůči těmto útokům. ChipWhisperer se zaměřuje zejména na analýzu napájení, která využívá informace uniklé ze spotřeby zařízení k provedení útoku, a také na útoky pomocí zákmitů na napájení a zdroji hodinového signálu, které krátkodobě přeruší napájení nebo hodiny zařízení a způsobí tak nechtěné chování (například vynechání kontroly hesla).“ [37]

Existuje několik verzí sady, které se liší v obsahu balení. Pro potřeby střídání modulů – *targetů* – s kryptoprocesory a mikrokotroléry na kterých se útoky provádí, byla zvolena sada *Level 1 Starter Kit (SCAPACK-L1)* [3] s dokoupeným modulem *NewAE targetu* obsahujícím CEC1702, již zmíněným ve volbě přípravků v podsekci 2.1.2. Všechny komponenty platformy ChipWhisperer, které jsou v rámci bakalářské práce využívány, jsou popsány v následujících odstavcích.

ChipWhisperer-Lite [38] Hlavní část sady (viz Obrázek 2.5) obsahující veškerý hardware potřebný pro generovaní zákmitů, vzorkování a zpracování spotřeby pozorovaného zařízení. Je nutné jej propojit s UFO deskou pomocí 10pinového konektoru (pro možnost programování zařízení) a koaxiálního kabelu (pro realizaci útoků).

CW308-UFO deska umožnující výměnu modulů (*targetů*) [32] Deska obsahující patici pro zasunutí jednoho z modulů (viz Obrázek 2.5), která disponuje mnoha vývody a konektory (např. pro snazší připojení osciloskopu či externích programátorů zařízení).

CW308 *target* s mikrokontrolérem STM32F3 [39] Základní *target* (viz Obrázek 2.5), který využívají všechny základní tutoriály usnadňující seznámení a realizaci jednotlivých typů útoků. Také je podporováno jeho programování přímo pomocí ChipWhisperer-Lite. STM32F3 je typu ARM podobně jako procesor CEC1702.

CW308 target s kryptoprocesorem CEC1702 [31] Dokoupený target, který obsahuje CEC1702 (viz Obrázek 2.4), jenž byl zvolen pro tuto bakalářskou práci. Usnadňuje realizaci útoků právě na zmíněný kryptoprocesor, kdy je lze provádět prostřednictvím ChipWhispereru-Lite. Je ale nutné využít externího programátoru pro nahrání programu přímo do zařízení.

Python knihovna s ChipWhisperer API Knihovna pro ovládání ChipWhispereru-Lite a snadné nastavení parametrů pro realizaci útoků. Také obsahuje komunikační protokol *SimpleSerial*, který umožňuje komunikaci s jednotlivými *targety* přes ChipWhisperer-Lite a UART. Celé API knihovny je detailně zdokumentováno a stále aktualizováno na [40].

Tutoriály s ukázkou jednotlivých útoků [41] Soubor *IPython* notebooků pro prostředí Jupyter Notebook, které uživatele nejdříve seznamují, jak s API a jak s ChipWhispererem-Lite komunikovat, nastavovat a jak programovat a komunikovat s jednotlivými *targety*. Tutoriály jsou rozděleny podle typu útoků a následně jsou stupňovány například od prolamování jednoduchých kontrol hesel až po útoky na reálné implementace šifer (jako je RSA nebo AES). V průběhu práce se sadou byly tutoriály v několikaměsíčních intervalech aktualizovány a rozšiřovány.

2.2.1 Zprovoznění prostředí

Jelikož autor bakalářské práce využívá operační systém Windows, zabývá se tato část zprovozněním prostředí pouze pro tento operační systém. Existují tři způsoby, jak s ChipWhisperer sadou pracovat [42]:

1. Stažení jednotlivých programů jako je Python, emulátor Unixového jádra, klient pro Git samostatně a následné naklonování repositáře ChipWhisperer. Výhodou je, že veškeré komponenty a jejich verze má uživatel pod kontrolou. Také pokud již většinu programů používá, není třeba stahovat mnoho dalšího.

2. Využití *image* pro *VirtualBox*, kdy se uživatel následně připojuje přes `localhost` k interaktivnímu webovému prostředí Jupyter Notebook, který zpřístupňuje vše pro práci s platformou ChipWhisperer.

3. Využití připraveného instalátoru pro Windows, který doinstaluje veškeré potřebné programy, emulátory a kompilátory a vytvoří spustitelný soubor `ChipWhisperer.exe`. Ten uživatele přesměruje také do prostředí Jupyter Notebook.

Jelikož autor bakalářské práce předtím programovací jazyk Python a prostředí Jupyter Notebook nepoužíval, byla zvolena třetí varianta (s instalátorem pro Windows). Problém nastal pouze se spouštěním *bash* příkazů, kdy buňky obsahující tyto příkazy musí obsahovat hlavičku `%%sh` místo `%%bash`, které využívá Ubuntu Subsystem pro Windows 10.

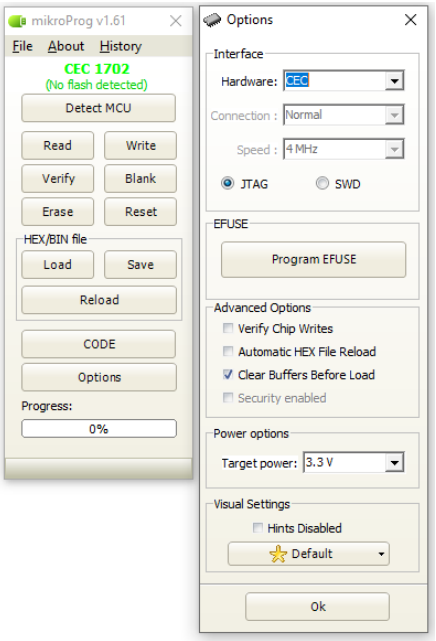
Volba instalátoru se ukázala jako moudrá, jelikož v průběhu bakalářské práce musel být operační systém osobního počítače autora bakalářské práce dvakrát přeinstalován a zprovoznění prostředí pomocí instalátoru se ukázalo jako velmi efektivní. Instalátor vytvoří celou adresářovou strukturu dostupnou z prostředí Jupyter Notebook přímo na disku. Ta je následně přístupnou přes průzkumník souborů a je tedy snadné využít vytváření záloh či verzovacího systému Git, aby se předešlo jakékoli ztrátě dat při nutné přeinstalaci operačního systému.

2.3 Nahrávání programů do CEC1702

Jelikož je přípravek *NewAE target* od jiného výrobce než *Clicker*, nemá přímou podporu pro programování RAM paměti pomocí HW programátoru. Deska UFO, do které je *NewAE target* zasazen, ale obsahuje 10pinový vývod, na který se dá již zakoupený programátor připojit. V dokumentaci k přípravku je však také popsán postup, jak naprogramovat přímo Flash paměť zařízení. Z té si čte CEC1702 program po restartu a nahrává jej do paměti RAM, ze které pak program běží.

2.3.1 Využití HW programátoru *mikroProg*

Jelikož je využití HW programátoru mikroProg [34] nutné i pro ladění programu, protože podporuje *debuggování* programu za běhu na zařízení, bylo rozhodnuto o zprovoznění právě této metody. Programátor zároveň využívá stejnojmenný SW – *mikroProg v1.61*, viz Obrázek 2.6 – který se spouští při



Obrázek 2.6: SW pro HW programátor mikroProg

nahrávání programu přes mikroC IDE a řídí HW programátor. Využitím schémat přípravku *Clicker* [30], *NewAE target* (sekce *Schematics* v [31]) a desky UFO (sekce *Schematic* v [32]) spolu s dokumentací programátoru [34] vzešlo najevo, že je propojení možné realizovat. Programátor využívá **JTAG** (Joint Test Action Group standard) [43] pro programování RAM paměti zařízení a přípravek má připravené vývody pro toto rozhraní přímo na desku UFO.

Po správném zadrátování 10pinu programátoru na desce UFO (viz Obrázek 2.8 a Obrázek 2.7) a také správném nastavení projektu je možné *NewAE target* úspěšně naprogramovat i *debuggovat*. V rámci spolupráce se studentkou Terezou Horníčkovou vznikl také návod, jak úspěšně pomocí HW programátoru *mikroProg for CEC* nahrát na *NewAE target* firmware s fungující UART komunikací. Návod je součástí datového média. Zda je návod kompletní, bylo testováno třetí osobou, která také používá mikroC IDE. Jelikož v návodu chyběla zmínka o nastavení projektu přímo v mikroC IDE, byl o tuto informaci návod doplněn.

Rozdílem oproti připojení ke *Clickeru* je, že HW programátor není schopen detekovat Flash paměť, která je totožná u obou přípravků. Byla provedena revize zapojení, ale pro konektory JTAG rozhraní – `J_TDI`, `J_TDO`, `J_TMS`, `J_TCK`, `3v3` – nebylo nalezeno jiné vhodné zapojení než to již vytvořené. Bylo tedy vyvozeno, že přístup HW programátoru k Flash paměti nejspíše zamezují piny `RST` (také označován `JTAG_nRST`) a `J_TRST`, jelikož jejich vývody je velmi snadné zaměnit při porovnávání schémat *Clickeru* [30] s *NewAE targetem* (sekce *Schematics* v [31]). Jako správné řešení se zdálo připojení `J_TRST`

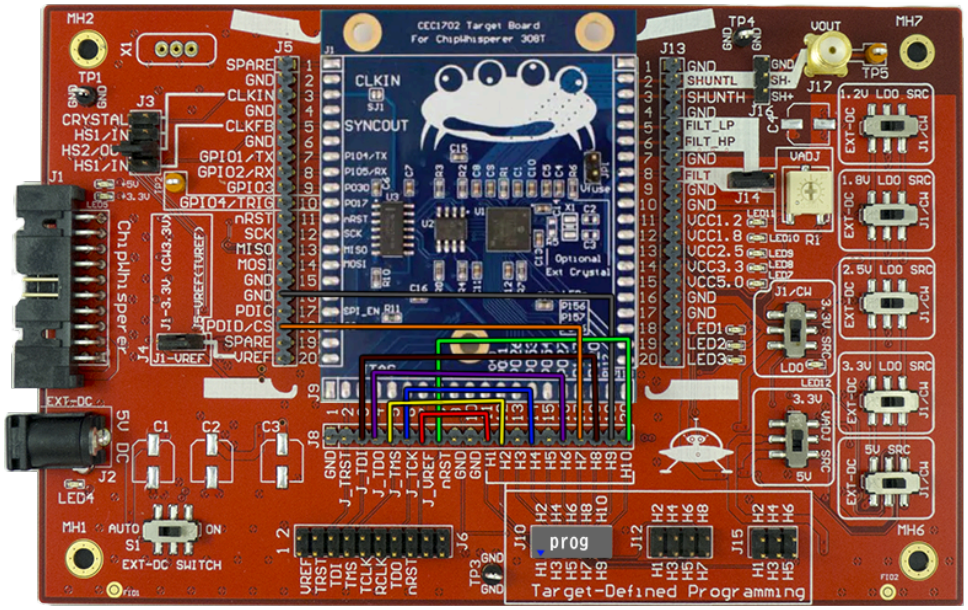
Obrázek 2.7: Reálné zapojení HW programátoru mikroProg k UFO desce obsahující CEC1702 *NewAE target*

k HW programátoru a pro umožnění přístupu programátoru k paměti připojit `nRST` k zemi. Tato kombinace však vyvolala zásek a spadnutí celého SW *mikroProg v1.61*. Když bylo propojení prohozeno, pády programu to nevyvolalo, ale paměť nebyla stále rozpoznána.

Pro realizaci úprav implementace Paillierova kryptosystému, knihovny pro práci s velkými čísly a vytvoření fungujícího firmware pro CEC1702 není třeba program nahrávat do paměti Flash. Na základě toho byla vyvozena implikace, že programováním Flash paměti a rozhraním JTAG se není třeba dále zabývat.

2.3.2 Programování Flash paměti zařízení

V začátcích práce na realizaci útoků postranními kanály si autor bakalářské práce uvědomil, že při snaze vyvolat změnu při výpočtu pomocí zákmitů na zdroji napájení se může CEC1702 často restartovat. A to z důvodu, že zákmitem může být i vyvolána porucha, kdy CEC1702 není schopen pokračo-



Obrázek 2.8: Detail zapojení HW programátoru mikroProg k UFO desce obsahující CEC1702 *NewAE target*, převzato z vytvořené dokumentace obsažené v příloze

vat v jakémkoliv výpočtu či komunikaci s okolím. Útok může probíhat i několik hodin, dokud se nenajde správné nastavení parametrů generátoru zákmitů pro úspěšné pozměnění výpočtu. Programování paměti RAM není vhodný způsob, jelikož se po restartu zařízení nemá odkud program nahrát a musí být nahrán znovu manuálně HW programátorem za využití mikroC IDE a útok by tedy nemohl být zcela automatizován. Byl tedy následně zvolen druhý způsob programování přímo paměti Flash v *NewAE targetu* [44], ze které se při každém restartu program opět do paměti RAM CE1702 nahrává. Jelikož využití HW programátoru s rozhraním JTAG se po předchozích zkušenostech nezdálo jako vhodné řešení, byla vzata v úvahu řešení jiná.

2.3.3 Možné varianty programování paměti

Existuje několik možností, jak Flash paměti naprogramovat, neboli změnit jejich obsah. *NewAE target* obsahuje paměť SST26VF016B [45], která podporuje pro její programování rozhraní **SPI** (Serial Peripheral Interface) či jeho modifikaci **SQI** (Serial Quad I/O) [46]. Jelikož autor bakalářské práce má již povědomí o SPI, byla provedena analýza možných řešení využívající programování za využití právě tohoto rozhraní.

Analýzou vyplula na povrch tři možná řešení. Prvním řešením je využít *Aardvark I2C/SPI Host Adapter* [47] s podpůrným SW *Total Phase Flash Center*, kdy je tento způsob zmíněn v dokumentaci k *NewAE targetu*. Druhou

možností je využití Arduina, kdy existuje několik projektů transformující platformu na jednoduchý SPI programátor Flash pamětí. Poslední možností, která byla vzata v úvahu, je využití USB-SPI převodníku *CH341A* [48] a jednoho z mnoha programů využívajícího převodníku pro programování Flash pamětí.

Využití *Aardvark I2C/SPI Host Adapter* s podpůrným SW *Total Phase Flash Center* se jeví jako nejjednodušší způsob. Problém je v ceně převodníku, která se pohybuje okolo 330 $ a je neúměrně vysoká v poměru s cenou přípravku, která činí přibližně 35 $ (700 Kč).

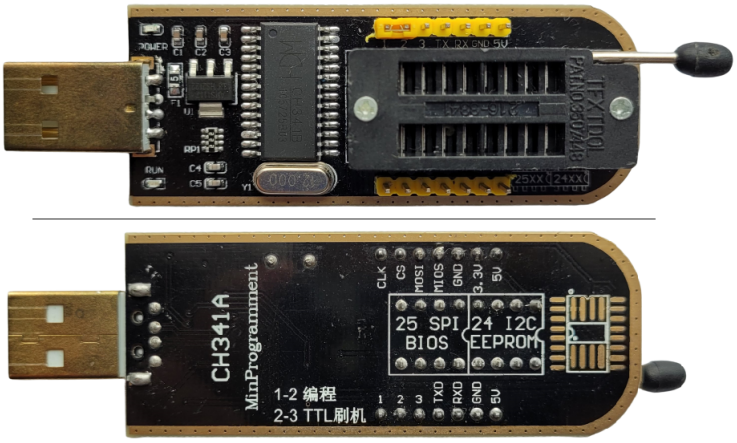
Úprava již existujících projektů pro Arduino, které z něj vytvoří jednoduchý SPI programátor, je pravým opakem první možnosti. Jelikož Arduino autor bakalářské práce vlastní, náklady by byly nulové. Problémem je nutný velký zásah do již existujících projektů, aby podporovaly jednotlivé příkazy pro mazání, čtení a zápis pro typ paměti obsažené v přípravku. Také by muselo být vyřešeno, jak binární soubor, který má několikanásobnou velikosti oproti paměti Arduina, z PC přes Arduino do paměti postupně nahrát.

V další sekci je popsána možnost poslední, která byla zároveň zvolena pro naprogramování Flash paměti.

2.3.4 Využití USB-SPI převodníku *CH341A*

Bylo zvoleno využití USB-SPI převodníku *CH341A* (viz Obrázek 2.9) a jednoho z mnoha programů, který využívá převodníku pro programování Flash pamětí. Převodník stojí pouze okolo 200 Kč, a jelikož je dostupný i na českých e-shopech, doručení autorovi bakalářské práce trvalo v řádu pár dnů. Pro převodník existuje mnoho *driverů*, které umožňují jeho jednoduché využití mnoha programy, které jsou většinou díly z východní polokoule, ale jsou dohledatelné i jejich zdrojové kódy pro případnou úpravu kompatibility s Flash pamětí přípravku *NewAE target*.

Ani jeden nalezený program využívající *CH341A* neobsahuje podporu paměti SST26VF016B [45], kterou obsahuje *NewAE target*. U této paměti je totiž nutné před každým zápisem či mazáním tyto operace povolit. Ovšem jeden z programů (*CH341A Programmer* [49]) obsahoval relativně moderní vzhled a také měl přístupnou celou implementaci využívající jazyk C#, kdy autorem sdílená implementace byla jednoduše spustitelná a editovatelná ve *Microsoft Visual Studio 2019*. Jedna zvláštnost spočívá v tom, že celý projekt není součástí Git repositáře a není verzován, ale jedna jediná verze je nahrána na Google disku autora [50]. Následná úprava kódu je popsána v sekci 3.4.



Obrázek 2.9: USB-SPI převodník *CH341A*, foto pořízené autorem bakalářské práce

Kapitola 3

Realizace

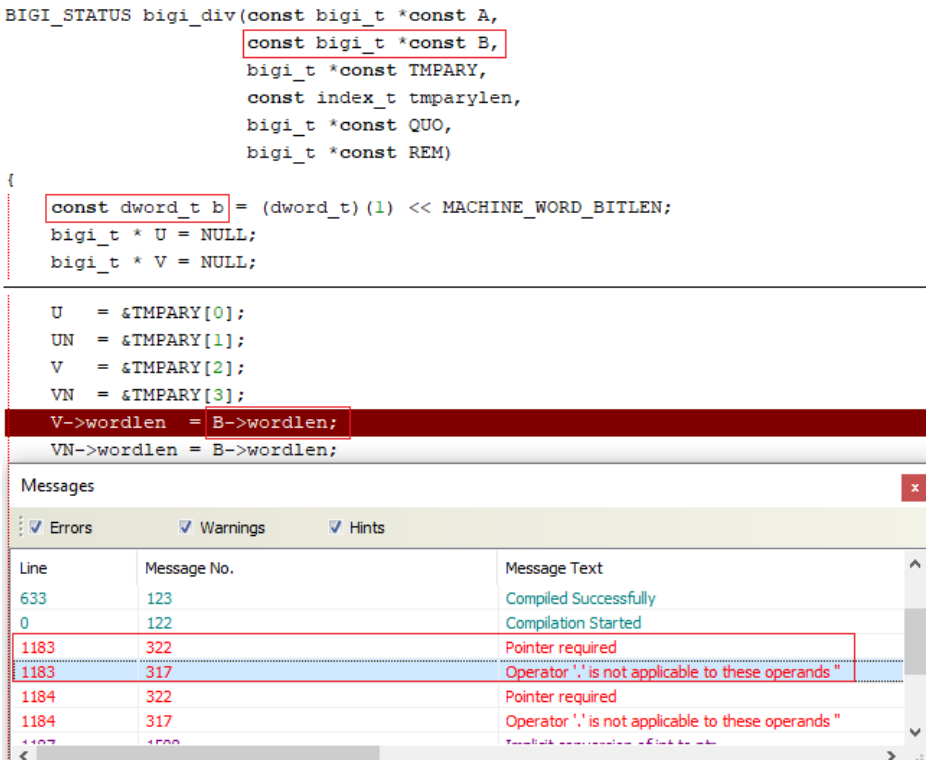
Tato kapitola obsahuje detailnější popis postupů při realizaci úprav u implementace knihovny pro práci s velkými čísly (dále jen *bigi*) a Paillierova kryptosystému. Dále je v této kapitole popsáno vytváření *firmware* pro CEC1702 s podporou šifrování pomocí RSA-CRT a Paillierova kryptosystému. *Firmware* CEC1702 také podporuje komunikaci s měřící sadou ChipWhisperer. Verifikací vyvinutého firmwaru se zabývá následující Kapitola 4. Na odladěný firmware byly následně aplikovány útoky postranními kanály, konkrétně útoky injekcí chyb. Útoky jsou popsány v Kapitole 5.

3.1 Úprava knihovny *bigi*

Knihovna pro práci s velkými čísly, pojmenovaná *bigi* [2], již existuje v podobě kompatibilní s Windows a Unix systémy a obsahuje zejména implementaci algoritmů pro modulární aritmetiku v naivních i efektivních variantách (např. využití Montgomeryho domény [51] u modulárního násobení a mocnění). Knihovna *bigi* je detailně popsána v článku „*Multiprecision ANSI C Library for Implementation of Cryptographic Algorithms on Microcontrollers*“ [2].

Autoři původní implementace již pracovali na její úpravě kvůli kompatibilitě s kryptoprocesorem CEC1302 [52]. Jedná se o zařízení ze stejné rodiny kryptoprocesorů jako CEC1702. Zdaleka ne všechny úpravy pro CEC1302 jsou kompatibilními s těmi pro CEC1702 a také nebyly některé z nich vhodně zvoleny. Potřebné úpravy jsou popsány v následujících odstavcích.

Názvy proměnných Po vytvoření projektu a snaze jej zkompilovat se vynořily chyby kompilace vskutku prazvláštní. Místo vstupní proměnné funkce `bigi_div ( ..., const bigi_t *const B ...)` si kompilátor vybíral proměnnou `const dword_t b` vytvořenou přímo v těle funkce. V tomto případě to tedy znamenalo fatální nesoulad datových typů a selhání při kompilaci, jak lze



Obrázek 3.1: Snímek obrazovky ilustrující problém se špatným rozpoznáním názvu proměnných

vidět na Obrázku 3.1. Zjistilo se, že kompilátor mikroC není *case-sensitive*[2], což je v rozporu se standardem jazyka C [53].

Po opravě kódu bylo o několik měsíců později zjištěno, že IDE má v nastavení schovaný přepínač, kterým lze *case-sensitivity* povolit. Tato změna ale následně znemožnila kompilaci konfigurace komunikace UART pomocí registrů, kdy jedno z maker přestal kompilátor poznávat.

Přetypování Dalším závažným nedostatkem mikroC IDE je nesprávné vyhodnocování přetypování. V několika funkcích se neukládala do proměnné přetypovaná hodnota, ale pouze nula. Kompilátor nedokáže zpracovat operaci s proměnnými, která je zabalená do přetypování na jednom řádku. Zároveň na tento nedostatek nebo nutnou úpravu kódu kompilátor neupozorní. Všechna přetypování obsahující jakoukoli operaci s proměnnými byla rozepsána na dva řádky s využitím pomocné proměnné v závislosti na typu, viz Obrázek 3.2.

[2]Case-sensitivity se dá do češtiny přeložit jako rozpoznatelnost malých a velkých písmen

```
...
t += k;
RES->ary[i] = (word_t)(t & MAX_VAL);
...
```

```
...
t += k;
/* IMPORTANT! Never merge the two following lines into single one
 * since MikroC compiler is a piece of [illegible]! And compiles it incorrectly!*/
tmp_loc = t & MAX_VAL;   /* (I.e., t & 0xFFFF). */
RES->ary[i] = (word_t)(tmp_loc);
...
```

Obrázek 3.2: Změny pro eliminaci problému s přetypováním v mikroC IDE

Licence mikroC IDE Základní verze mikroC IDE je omezena na kompilaci jednodušších projektů (do 2000 slov programu [54]). Aby se odemkla možnost kompilovat libovolně velký projekt, musela být pořízena pro mikroC IDE licence. Cena je 320 $, což vhledem k zastaralému vizuálu a neexistujícím aktualizacím od roku 2019 je cena přemrštěná. Nicméně vzhledem ke skutečnosti, že se na tento problém přišlo po dlouhých hodinách úprav kódu pro toto IDE, bylo rozhodnuto, že se licence obstará.

Omezení velikosti polí Aby bylo možné demonstrovat funkčnost knihovny, implementace již obsahuje soubor `bigi\examples\example.c`. Pro kompatibilitu CEC1702 bylo nutné zjistit, proč nelze v hlavní funkci programu `int main()` inicializovat dostatečné množství polí o velikosti závislé na délce čísel. Při překročení určitého limitu – kombinace délky zpracovávaných čísel a použitých operací – se běh programu na CEC1702 v určitém bodu výpočtu vždy zastavil. Bez snížení délky čísel či vypnutí některých operací program nikdy nepokračoval dále.

Problém byl simulován v projektu `Array_Test`, který je přiložen na médiu (`1_bigi\Array_Test`). V rámci projektu byla snaha vytvořit co největší pole v těle funkce, a to za využití metod pokus-omyl a binární půlení. Pokusy byla nalezena magická konstanta 5000 B – `uint_8t arr[5000]` nebo dvě pole `uint16_t arr[2500]` – kdy větší pole inicializovat uvnitř funkce kompilátor nepovolil. V těle funkce se ještě nacházel pomocný *buffer* pro výpis přes UART a pomocné proměnné na simulaci práce s polem. Při vytváření tří a více polí se již součet jejich velikostí v bajtech mírně snižoval oproti zmíněné konstantě 5000 (zřejmě z důvodu více interních proměnných pro správu paměti).

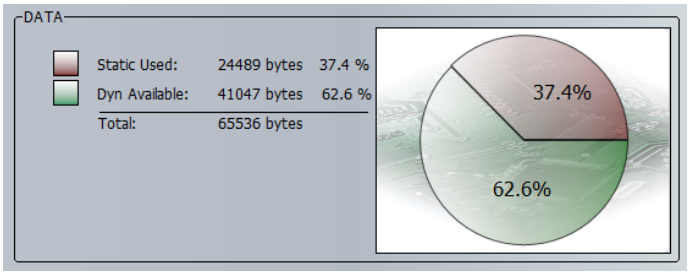
O dost lepší situace nastává, pokud se pole nacházejí v globálním prostoru, kde už není omezení velikosti pole určitou konstantou, ale pamětí RAM pro data. CEC1702 obsahuje 64kB data RAM, kdy se od této hodnoty musí odečíst již využité místo jinými globálními proměnnými. V souboru `example.c` se tedy musely převést všechny deklarace polí do globálního prostoru. Tento posun dovoluje nyní využít dostatečný počet pomocných polí (dvacet) pro správ-

nou funkčnost všech funkcí(včetně těch, které využívají Montgomeryho doménu). I při použití čísel dlouhých 4096 bitů a volání všech podporovaných operací v rámci jednoho běhu programu je paměť pro data využitá na 37,4 %, viz Obrázek 3.3. Při použití čísel l28 bitů dlouhých je využití RAM pro data pouze 5,6 %.

Úprava časovačů V rámci demo implementace v `example.c` je využíváno časovačů pro přesnější měření délky trvání jednotlivých operací. Implementace časovačů je platformě závislá, jelikož využívá interních registrů a přerušení. Ač by si člověk řekl, že by CEC1702, který je ze stejné řady co CEC1302, mohl mít i stejná makra pro přístup k jednotlivým registrům zařízení, opak je pravdou. Jelikož CEC1702 obsahuje stejné časovače a registry jako CEC1302 na totožných adresách, bylo nutné podle adres najít správná makra a ta všude v projektu upravit.

Funkčnost časovačů byla následně otestovaná s využitím zpožďovací funkce z mikroC IDE, kdy funkce umožňuje nastavit zpoždění v milisekundách. Výsledky časovačů ale přesně neodpovídaly hodnotám uvedeným ve funkci. Nepřesnost nebyla způsobena implementací časovačů, ale implementací zpožďovací funkce. Implementace funkce v mikroC IDE není přizpůsobena pro každé podporované zařízení zvlášť [55].

Zajímavost s volbou optimalizace kompilátoru Kompilátor dovoluje několik úrovní optimalizace – 0 až 5 – spolu s možností **SSA** (Single Static Assignment) [56] optimalizace. Problém nastává při volbě optimalizace 0, kdy následně přestane fungovat převod čísla v hexadecimálním zápisu do struktury `bigi_t` a s největší pravděpodobností i další funkcionalita. Ostatní stupně optimalizace tímto problémem netrpí. Závěrem jest doporučení nechat nastavení optimalizace, které je v základu v IDE nastaveno a zbytečně se jiným nastavením nezabývat. Jedná se o stupeň optimalizace 4 spolu s SSA optimalizací.



Obrázek 3.3: CEC1702 – využití paměti RAM pro data demo programem, který zpracovává 4096b čísla všemi *bigi* funkcemi

3.2 Úprava implementace Paillierova kryptosystému

Stejně jako u *bigi* i implementace Paillierova kryptosystému již existuje. Ta vznikla v rámci již několikrát zmiňované diplomové práce [5] ve funkční podobě kompatibilní s Windows a Unix systémy. Implementace byla také převzata s úpravami pro CEC1302.

Implementaci lze rozdělit na dvě části:

1. Naivní implementace Pailliera, která implementuje základní verzi Pailliera popsanou v podsekci 1.2.1.1,

2. implementace využívající modifikaci s CRT popsanou v podsekci 1.2.1.2.

První část je implementována čistě SW za využití *bigi*. Při začátcích práce byla funkční na platformách Unix, Windows a CEC1302. Druhá část implementace Pailliera využívající CRT existuje ve dvou variantách:

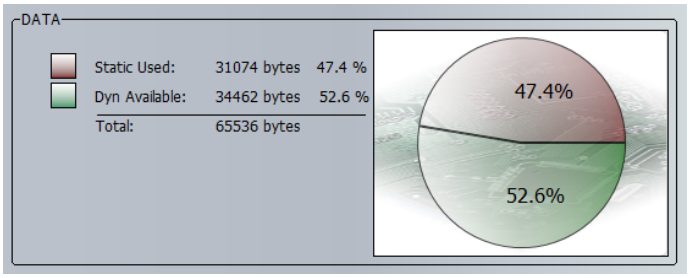
1. Pro urychlení modulárního mocnění je využita HW implementace RSA, přesněji funkce `rsa_encrypt()` z API pro HW RSA.

2. Modulární mocnění je implementováno voláním funkcí ze simulátoru HW akcelerátoru RSA, který je implementován pomocí *bigi*. Simulátor je využit na platformách, které neobsahují HW implementaci RSA.

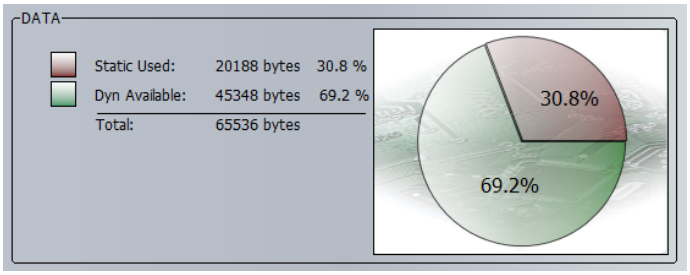
Volbu varianty Pailliera využívající CRT řídí makro `RSA_ACCELERATION`. Pokud je makro zadefinováno, je použita první varianta využívající HW implementace RSA. Program je poté možné zkompilovat pouze v mikroC IDE. Ukázka kódu s využitím obou variant je na Obrázku 3.4.

```
#ifndef __MIKROC_PRO_FOR_ARM__
    #ifdef RSA_ACCELERATION
        #error "RSA acceleration macro is on, but not supported by hardware."
    #endif
#endif
...
#if defined(__MIKROC_PRO_FOR_ARM__) && defined(RSA_ACCELERATION)
    ...
    rsa_hw_status = rsa_encrypt(RSA_BITLEN, &r_buff8_array, BIG_ENDIAN);
#else
    rsa_hw_status = simul_rsa_encrypt(RSA_BITLEN, &r_io_byte_ary[0],
                                      RSA_BYTELEN, RSA_SLOT_MOD_Q2);
#endif
```

Obrázek 3.4: Ukázka rozlišení variant v implementaci Pailliera využívající CRT



Obrázek 3.5: Využití RAM paměti pro data u CEC1702 při použití čistě SW implementace Pailliera pracujícího s 4096bitovými čísly



Obrázek 3.6: Využití RAM paměti pro data u CEC1702 při použití Pailliera s HW RSA akcelerátorem pracujícího s 4096bitovými čísly

Při začátcích práce byla implementace funkční na platformách Unix, Windows a CEC1302. Pro zprovoznění první části implementace a obou variant druhé části využívající CRT na CEC1702 bylo třeba provést několik úprav popsaných dále v této sekci.

Úpravy podle *bigi* Implementace obsahuje soubor pro demonstraci a otestování funkčnosti implementace – `paillier\examples\example.c`. Opět bylo nutné (obdobně jako u *bigi*) přesunout všechny inicializace polí do globálního prostoru. Dále `example.c` obsahuje použití časovačů pro přesnější měření času šifrování. Ty byly upraveny také po vzoru *bigi*. Problémy s přetypováním byly taktéž odhaleny a ihned napraveny. V názvech proměnných ale žádný konflikt při kompilaci nenastal. Využití RAM pro data, při použití dvou různých variant druhé části implementace Pailliera, lze pozorovat na Obrázku 3.5 a 3.6

Nápravy chyb z nedopatření Do souboru `paillier.c` byly přidány chybějící hlavičkové soubory (např. `#include <bigi_io.h>`). Následně kompilátor odhalil konflikt při pojmenování dvou pomocných polí struktur `bigi_t` v souborech `paillier.c` a `example.c`. Obě pole nesla stejný název `tmpary`. Pojmenovaní v `paillier.c` bylo tedy změněno na `tmpary_p`.

Další záludnou chybou bylo použití nesprávného portu pro sériovou komunikaci – označen jako `UART1`. Ten ovšem funguje pouze s vývojovou sadou *Clicker2 for CEC1702* a je nutné používat pouze port `UART0`.

Během spuštění demo programu `example.c` nebylo vypisováno náhodně generované číslo, které je použito při šifrování. Funkce `plain_paillier_encr()` a `paillier_crt_encr()` používají při každém volání nové náhodně generované číslo a bez jeho znalosti, nelze šifrování ověřit. Pro možnost výpisu náhodně generovaného čísla byla do implementace přidána možnost použití makra `RUNNING_EXAMPLE`. Pokud je makro zadefinované, výpis čísla se v rámci funkcí `plain_paillier_encr()` a `paillier_crt_encr()` provede. Pokud makro zadefinované není, náhodně generované číslo se nevypisuje.

Po provedení úprav začala implementace fungovat, pokud byla použita druhá varianta CRT modifikace Pailliera – využívající simulátor HW akcelerátoru pro RSA.

3.2.1 Využití hardwarového akcelerátoru kryptografických operací u CEC1702

Aby v implementaci bylo možné HW akcelerátor kryptografických operací CEC1702 použít, bylo nejdříve nutné zjistit, jaké volání funkcí API podporuje a jaký mají syntax.

V rámci mikroC IDE v sekci „*Library Manager*“ existuje výpis funkcí API zpřístupňující HW implementaci RSA. Výpis funkcí ale neobsahuje žádnou dokumentaci. V záložce „*mikroC PRO for ARM Help*“ sekce o RSA zcela chybí. Popis ostatních částí HW akcelerátoru kryptografických operací ve zmíněné záložce obsažen je, jmenovitě například popis API pro přístup k HW implementaci AES. Po chvíli hledání však byla nalezena vcelku podrobná dokumentace všech funkcí API pro HW akcelerátor kryptografických operací včetně RSA a RSA-CRT modulu [57].

Volba funkce z API pro urychlení CRT modifikace V dokumentaci API se objevují tři funkce, které by mohly být použity na urychlení modulárního mocnění u varianty implementace využívající CRT:

1. `rsa_encrypt()`, jelikož šifrování pomocí RSA je „pouhé“ modulární mocnění. Tato funkce podporuje až 4096bitová čísla. Stejná funkce je použita i při implementaci pro CEC1302. Funkce vyžaduje předem nahrát hodnoty do slotů HW akcelerátoru (jako parametry klíče).

2. `rsa_crt_decrypt()`, kterou by se dala CRT varianta také implementovat. Tato funkce podporuje maximálně 2048bitová čísla. Využití této funkce by způsobilo větší úpravy kódu, jelikož by při výpočtech nebylo nahrazeno pouze modulární mocnění.

3. `rsa_modular_exp()` podporující až 4096bitová čísla. Funkce nepotřebuje předem nahrát hodnoty do slotů HW akcelerátoru.

Druhá funkce – `rsa_crt_decrypt()` – byla odsunuta z výběru, jelikož využití této funkce by způsobilo nutné větší úpravy kódu. Po konzultaci s vedoucím práce bylo rozhodnuto, že v budoucích pracích může vzniknout třetí varianta implementace Pailliera s CRT modifikací využívající právě funkci `rsa_crt_decrypt()`.

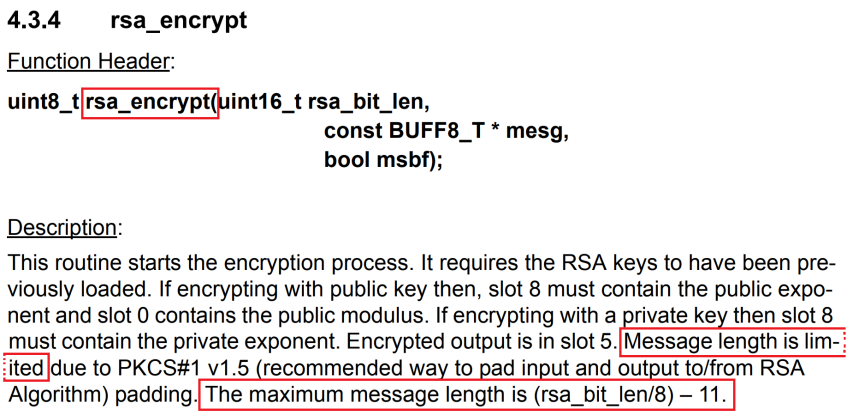
Pro porovnání rychlostí funkcí `rsa_modular_exp()` a `rsa_encrypt()` byl vytvořen jednoduchý projekt `RSA_Encrypt_vs_ModExp`, který je přiložen na médiu. Projekt využívá implementaci ze vzorového projektu mikroC IDE[3]. Modulární mocnění 1024bitového čísla s využitím `rsa_modular_exp()` trvá 2 ms. Stejná operace s využitím funkce `rsa_encrypt()` tvrá 3 ms. Jelikož je výpočet pomocí funkce `rsa_modular_exp()` rychlejší než pomocí `rsa_encrypt()`, vzniklo podezření, že by funkce `rsa_modular_exp()` nemusela být zabezpečená (doplněná o validaci výpočtu). Dokumentace `rsa_modular_exp()` toto tvrzení bohužel nepotvrzuje, jelikož je velice skoupá, jeden by řekl až nekompletní. Hypotézu by bylo možné zcela potvrdit útokem injekcí na funkce `rsa_encrypt()` a `rsa_modular_exp()` a porovnat, zda mají stejné chování. Zda při pokusu o ovlivnění výpočtu náhodou nevrací nesprávnou hodnotu. Útoky pomocí injekcí chyb se zabývá Kapitola 5. Vzhledem k časovým nárokům na tento pokus již nezbyl čas a je přesunut do potenciálních budoucích prací.

Pro urychlení modulárního mocnění byla nakonec zvolena první funkce – `rsa_encrypt()`. Není třeba velkých úprav kódu pro její použití, jelikož `rsa_encrypt()` byla využita pro urychlení modulárního mocnění na CEC1302. Autor předpokládá, že je funkce zabezpečená proti útokům postranními kanály (oproti funkci `rsa_modular_exp()`). Jelikož funkce `rsa_encrypt()` podporuje klíče do délky 4096 bitů, Paillier může být až 8192bitový. Dalším důvodem pro zvolení šifrovací funkce byla domněnka, že u ní lze využít více slotů pro paralelní šifrování. Zmíněná domněnka byla nakonec vyvrácena a je popsána v části „*Pouze jeden RSA modul*“.

Nesprávně zdokumentovaná potřeba *paddingu*. V dokumentaci API pro HW akcelerátor kryptografických operací [57] je v popisu `rsa_encrypt()` uvedeno, že by měla automaticky používat *padding*[4], viz Obrázek 3.7. Vstupní data pro šifrování by musela být kratší o 11 B, než je nastavena délka šifry. Bylo proto provedeno otestování, zda je *padding* vážně nutné použít.

[3] Jedná se o jednu ze vzorových implementaci demonstrující základní použití HW akcelerátoru kryptografických operací, které jsou součástí nainstalovaného mikroC IDE v záložce *Projects - Open Examples Folder...*

[4] Padding se dá do češtiny přeložit jako „výplň“, která se používá pro doplnění textu do délky bloku šifry, aby šifrování mohlo být provedeno korektně



4.3.4 rsa_encrypt

Function Header:

```
uint8_t rsa_encrypt(uint16_t rsa_bit_len,
                    const BUFF8_T * mesg,
                    bool msbf);
```

Description:

This routine starts the encryption process. It requires the RSA keys to have been previously loaded. If encrypting with public key then, slot 8 must contain the public exponent and slot 0 contains the public modulus. If encrypting with a private key then slot 8 must contain the private exponent. Encrypted output is in slot 5. Message length is limited due to PKCS#1 v1.5 (recommended way to pad input and output to/from RSA Algorithm) padding. The maximum message length is (rsa_bit_len/8) – 11.

Obrázek 3.7: Nesprávný popis funkce v dokumentaci API pro přístup k HW akcelerátoru kryptografických operací CEC1702, převzato z [57]

Nejdříve byl 1024b blok dat zašifrován pomocí funkce `rsa_encrypt()`. Výstup šifrování byl následně dešifrován využitím funkce `rsa_decrypt()`. Dešifrovaná data se rovnala datům před dešifrováním. Byl tedy vyvozen závěr, že v dokumentaci je nepravdivá informace a funkci `rsa_encrypt()` je možné využít pro urychlení modulárního mocnění.

Typy parametrů funkcí z API pro CEC1702 Použití funkcí z API pro CEC1702 se liší oproti použití funkcí z API pro CEC1302. Rozdíl lze pozorovat na definice funkce `rsa_encrypt()` pro CEC1702 (viz Obrázek 3.7) a pro CEC1302 (viz Obrázek 3.8). Nesedí počet parametrů funkcí a také má téměř každý parametr funkcí jiný význam nebo datový typ. Při použití funkce pro CEC1702 je blok dat pro šifrování třeba zaobalit do struktury `BUFF8`, kde je následně uložena i délka bloku. Jaké proměnné `BUFF8` struktura obsahuje, nikde zdokumentováno nebylo a „prokliknutí“ na deklaraci struktury v IDE také nefunguje. Inicializace a použití struktury `BUFF8` muselo být odpozorováno z existujících vzorových příkladů zahrnutých v IDE. V implementaci Pailliera byly tedy následně inicializovány další globální statické proměnné typu `BUFF8`, do kterých se data před zpracováním musí zaobalit, aby mohla být použita jako parametr pro `rsa_load_key()`, `rsa_encrypt()` a `rsa_decrypt()` (viz Obrázek 3.9).

Pouze jeden RSA modul HW akcelerátor kryptografických operací obsahuje 31 slotů o velikosti 512 B pro nahrání veřejných a tajných exponentů, modulů, dat k zašifrování… Jejich přiřazení je ale pevně dané. Když se tedy ve funkci `paillier_crt_encr()` s dvěma různými klíči šifruje (modulárně mocní), je nutné klíče vyměnit během šifrování. Není možné klíče nahrát do dvou různých slotů v rámci `paillier_crt_init()`, jak je tomu u simulátoru HW akcelerátoru.

CEC1302 Crypto API User's Guide

5.2.5 rsa_encrypt

Function Header:

```
uint8_t rsa_encrypt(uint16_t rsa_bit_len,
                    const uint8_t * mesg,
                    uint16_t mlen,
                    uint8_t flags);
```

Obrázek 3.8: Ilustrace naprosto odlišného popisu funkce v API pro přístup k HW akcelerátoru kryptografických operací CEC1302, převzato z [58]

```
static BUFF8 r_buff8_array;
static byte_t r_io_byte_ary [RSA_BYTELEN];
...
BIGI_STATUS paillier_crt_encr(...) {
  ...
  bigi_to_bytes(&r_io_byte_ary[0], RSA_BYTELEN, r);
  ...
  r_buff8_array.len = RSA_BYTELEN;
  r_buff8_array.pd  = (char*)&r_io_byte_ary[0];

  rsa_hw_status = rsa_encrypt(RSA_BITLEN, &r_buff8_array, BIG_ENDIAN);
  pke_start(0);
  ...
}
```

Obrázek 3.9: Příklad použití struktury `BUFF8` v implementaci Pailliera

Oprava kódu spočívá v tom, že se nyní inicializované CRT parametry uloží do `BUFF8` struktur ve funkci `paillier_crt_init()`. V `paillier_crt_encr()` se poté provede následující (viz podsekce 1.2.1.2):

1. Pomocí funkce `rsa_load_key()` se do slotů pro RSA nahrají data odpovídající p^2 a n.
2. Provede se zašifrování proměnné obsahující náhodně vygenerované číslo voláním funkce `rsa_encrypt()`. To odpovídá operaci $r^n \bmod p^2$.
3. Pomocí funkce `rsa_load_key()` se do slotů pro RSA přehrají data odpovídající q^2 a n.
4. Opět se voláním `rsa_encrypt()` provede zašifrování proměnné z 2. kroku. Tato operace odpovídá $r^n \bmod q^2$.

Nevylučuje se, že by použitím správných funkcí z API nebylo možné dva RSA sloty nakonfigurovat, ale po konzultaci s vedoucím práce bylo vyhodnoceno, že dosavadní řešení je pro naplnění cíle bakalářské práce dostačující a není třeba ho dále rozšiřovat.

```
for (i = 0; i < IN->wordlen; i++) {
    (word_t*)&out[i * MACHINE_WORD_BYTELEN] = IN->ary[IN->wordlen - i - 1];
}
return OK;
```

```
for (i = 0; i < IN->wordlen; i++) {
    tmp = IN->ary[IN->wordlen - i - 1];
    out[i * MACHINE_WORD_BYTELEN + 0] = (byte_t)((tmp >> 24) & 0xFF);
    out[i * MACHINE_WORD_BYTELEN + 1] = (byte_t)((tmp >> 16) & 0xFF);
    out[i * MACHINE_WORD_BYTELEN + 2] = (byte_t)((tmp >> 8) & 0xFF);
    out[i * MACHINE_WORD_BYTELEN + 3] = (byte_t)((tmp >> 0) & 0xFF);
}
return OK;
```

Obrázek 3.10: Úprava ukládání struktury `bigi_t` do pole bajtů v rámci funkce `bigi_to_bytes()`

Nutná úprava endianity Posledním problémem byla endianita - pořadí bajtů v paměti. Všechny funkce z API pro HW RSA obsahují parametr `msbf`, který určuje, jakou endianitu předávaná data mají. HW akcelerátor je zároveň řízen i **PKE** (kryptografie s veřejným klíčem) funkcemi z API. Například pro načtení zašifrovaných dat funkcí `rsa_encrypt()` je nutné využít funkci `pke_read_scm()`. Tato funkce obsahuje také parametr pro určení endianity ukládaných dat –`reverse_byte_order`. Přestože se parametry `msbf` a `reverse_byte_order` liší názvem a v dokumentaci API obsahují jiný popis, fungují identicky. Pro jednotné nastavení endianity bylo zadefinováno makro `#define BIG_ENDIAN 1`. Makro se poté předává všem funkcím, které vyžadují upřesnění endianity dat.

Po nastavení makra `BIG_ENDIAN` v celé implementaci Pailliera, byl ale výpis po provedeném šifrování stále nesprávný. Problém byl v *bigi* u funkcí `bigi_to_bytes()` a `bigi_from_bytes()`, kde se transformuje `bigi_t` struktura na pole bajtů a naopak. Struktura `bigi_t` obsahuje pole 4B slov. CEC1702 je little-endian a v případě použití `bigi_to_bytes()` se čtveřice bajtů přehrály do pole ve špatném pořadí, tzn. jednotlivá 4B slova byla ve správném pořadí (big-endian), ale v rámci slov se objevilo opačné pořadí bajtů (little-endian). Funkce `bigi_from_bytes()` načítala jednotlivá slova také nesprávně. Implementace funkcí se změnila, aby bylo postupně v rámci 4B slov upraveno pořadí bajtů bitovými posuny (viz Obrázek 3.10). Tato změna v implementaci *bigi* je zmíněna zde, jelikož tato chyba byla nalezena až při používání Pailliera. Po úpravě *bigi* funkcí začala celá implementace Pailliera fungovat správně.

3.3 Vytvoření *firmware* pro komunikaci s CEC1702

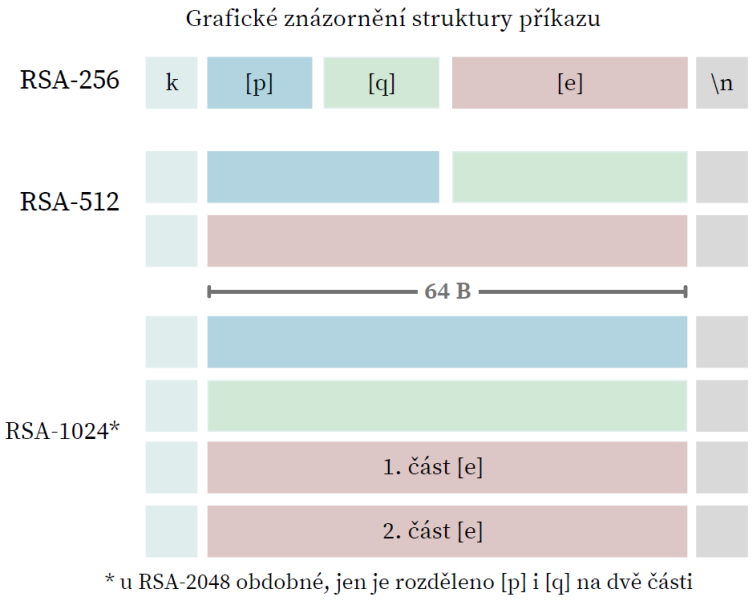
Funkčnost implementací z předchozích sekcích je demonstrována pouze pomocí `example.c` souborů. Nově vzniklý *firmware* podporuje:

1. Komunikaci s okolím pomocí UART, pro nastavení klíčů šifer a příjem dat k šifrování,
2. šifrování pomocí Pailliera, využitá implementace je popsána v sekci 3.2,
3. šifrování pomocí RSA-CRT implementovaného čistě SW využitím *bigi*,
4. šifrování pomocí RSA-CRT využívající HW akcelerátor kryptografických operací,
5. dynamicky měnit velikost klíče šifer za běhu programu,
6. dynamicky měnit používanou šifru za běhu programu.

Firmware nepodporuje dynamicky měnit implementaci šifer. Tedy jestli má být využívána pouze čistě SW implementace šifer, či implementace využívající HW akcelerátor kryptografických operací. Nově vytvářená implementace RSA-CRT by tuto možnost podporovat mohla. U Pailliera je použití HW akcelerátoru kryptografických operací voleno preprocesorovými makry, a tudíž by bylo třeba většího zásahu do kódu. Pro zachování jednotnosti implementace RSA-CRT a Pailliera je volba mezi čistě SW implementací šifer a implementací s HW akcelerátorem prováděna pomocí makra `HW_IMPLEMENTATION` při kompilaci *firmware*.

Podpora komunikace s ChipWhisperer-Lite *Firmware* je implementován na *NewAE targetu*, který je součástí sady ChipWhisperer. ChipWhisperer-Lite využívá pro komunikaci s moduly protokol *SimpleSerial* [59]. Pro podporu komunikace s ChipWhisperer-Lite byla v rámci *firmware* pro CEC1702 zvolena realizace právě protokolu *SimpleSerial*. V rámci ChipWhisperer repositáře již existovala základní implementace *SimpleSerial* protokolu pro CEC1702 [60] verze 1.1 a byla zvolena jako základ pro vytvoření celého *firmware*. *SimpleSerial* na CEC1702 má následující vlastnosti:

1. Pro příjem příkazů a odesílání odpovědí je využit UART,
2. přenosová rychlost (Baudrate) u UART je 38 400 Baud,
3. maximální délka dat v jednom příkazu je 64 B,
4. každý příkaz je identifikovatelný jedním znakem,
5. funkce v CEC1702, které jsou vyvolány příkazy, musí odesílat odpověď v jasně definovaném formátu `r[DATA - max. 64 B]`,
6. funkce v CEC1702 mohou informovat o úspěchu odesláním příkazu `z00`. V případě neúspěchu operace může být odeslán příkaz `z01`.



Obrázek 3.11: Grafické znázornění struktury příkazů pro nastavení klíče u RSA-CRT v rámci *firmware* pro CEC1702, výstřižek z dokumentace

SimpleSerial protokol má značná omezení a přesně definovanou strukturu, bez které ChipWhisperer API nedokáže příkazy a data zpracovávat. Z tohoto důvodu má posílání dat k šifrování a nastavení klíčů některá omezení. Protože lze z definice *SimpleSerial* posílat maximálně 64 B (512 b) v rámci jednoho příkazu, musí být odesílání dat pro šifrování či dešifrování u větších délek než 512 b rozděleno na několik dílů. Firmware tyto díly postupně spojuje dohromady a následně nad kompletními daty provede příslušnou operaci šifrování či dešifrování. Nastavení klíče je také zjednodušeno na odeslání pouze potřebných prvočísel p a q, případně e u RSA-CRT. Z přijatých parametrů *firmware* klíče šifer dopočítá a uloží. Odesílání parametrů pro klíč má pevně danou strukturu `k[p][q][e - jen u RSA]`, viz také ukázka grafiky z dokumentace k *firmware* na Obrázku 3.11. Celá dokumentace s detailním popisem *firmware* je obsažena v příloze C. Pokud je opět blok dat s parametry pro klíč delší než 64 B, je rozdělen do několika příkazů, které začínají `k`. Kvůli pevně dané délce příkazů se musí parametr e pro RSA-CRT odesílat vždy dlouhý tolik B, na kolik je nastavený klíč. Pokud je tedy zvoleno 1024b RSA-CRT, musí mít parametr e 128 B, i když je tvořen z velké části nulami (v důsledku konvencí pro volbu veřejného exponentu, viz podsekce 1.1.1.1).

Realizace RSA-CRT Pro *firmware* je kromě Paillierа implementováno RSA-CRT, jelikož právě RSA-CRT bylo vybráno pro prozkoumání možností provedení útoku injekcí chyb (viz Kapitola 5). V Paillierovi existuje naivní implementace RSA pomocí *bigi* v rámci simulátoru HW akcelerátoru (viz sekce 3.2).

Tato implementace je ale nedostatečná, jelikož pro *firmware* byla zvolena modifikace RSA-CRT. RSA-CRT je implementováno ve dvou variantách:

1. Čistě SW varianta využívající *bigi*, která umožňuje volbu klíče až 4096 bitů a

2. varianta využívající API pro HW akcelerátor RSA, která umožňuje délku klíče 1024 bitů nebo 2048 bitů.

Implementace čistě SW varianty se obešla bez větších potíží díky nabytým znalostem z úprav *bigi* knihovny (viz sekce 3.1) a Pailliera (viz sekce 3.2). Pro implementaci varianty s využitím HW akcelerátoru RSA byla zvolena funkce `rsa_crt_decrypt()`, která podle dokumentace API pro CEC1702 [57] využívá HW implementaci RSA-CRT a dovoluje maximální délku klíče 2048b. V dokumentaci API autor nalezl dvě funkce pro nastavení klíče pro RSA-CRT (viz sekce 1.1.1.2):

1. `rsa_crt_gen_params()`- z předaných prvočísel p, q a parametrů tajného klíče K_T funkce zbylé části klíče vygeneruje a uloží do slotů HW akcelerátoru.

2. `rsa_load_crt_params()` - funkci se předají již vypočítané parametry d_p, d_q, q_{inv}, které si uloží do slotů HW akcelerátoru.

Před voláním funkcí `rsa_crt_gen_params()` i `rsa_load_crt_params()` musí být podle dokumentace volána funkce `rsa_load_key()`, která nastavuje tajný i veřejný klíč v HW RSA akcelerátoru předáním parametrů n, e a d.

Byla zvolena první funkce – `rsa_crt_gen_params()` – jelikož dovoluje urychlení generování klíče. Jednotlivé části RSA-CRT klíče nemusí být předem vypočítány pomalejší SW implementací využívající *bigi*.

Při použití `rsa_crt_gen_params()` následné dešifrování pomocí funkce `rsa_crt_decrypt()` vracelo nesprávná data. Při nastavení 1024b klíče bylo vráceno 512 nulových bitů a zbylých 512 bitů sestávalo z náhodného čísla.

Následně byla otestována druhá funkce – `rsa_load_crt_params()` – pro nahrání všech CRT parametrů. To způsobilo zastavení chodu celého *firmware* po zavolání funkce `rsa_crt_decrypt()`. Čekání na dešifrovaná data je totiž blokující, viz Obrázek 3.12).

Jelikož funkce `rsa_crt_gen_params()` a `rsa_load_crt_params()` nenastavili klíč pro RSA-CRT samostatně správně, bylo otestováno nastavení klíče zavoláním obou funkcí za sebou. Dešifrování pomocí `rsa_crt_decrypt()` následně vrátilo validní data. Z této skutečnosti bylo vyvozeno, že je nutné zavolat `rsa_crt_gen_params()` i `rsa_load_crt_params()` pro správné nastavení klíče HW akcelerátoru RSA-CRT. Pořadí, ve kterém se `rsa_crt_gen_params()` a `rsa_load_crt_params()` volají, může být libovolné.

```
rsa_status =  PKE_RET_OK;
c_buff8_array.pd = (char*)&ciphertext_byte_array[0];

rsa_status = rsa_crt_decrypt(operation_length_in_bites,
                             &c_buff8_array, BIG_ENDIAN);
pke_start(0);

if (rsa_status != PKE_RET_OK) {
    return 0x01;
}
while(pke_busy() == 1) asm nop;

bytes_read = pke_read_scm(&plaintext_byte_array[0],
                          RSA_BITS, 5, BIG_ENDIAN);
```

Obrázek 3.12: Ukázka použití funkce `rsa_crt_decrypt()`, v červeném rámečku je označeno blokující čekání na dokončení dešifrování

Nutné přizpůsobení implementace Pailliera Aby bylo možné veškeré funkce Pailliera používat, vznikl soubor `paillier_wrapper.c`, který:

1. Obsahuje potřebné struktury a pomocné proměnné pro ukládání klíčů Pailliera,
2. umožňuje dynamicky měnit velikost struktur v závislosti na délce používaného klíče,
3. umožňuje dynamicky měnit použitou implementaci (naivní či s využitím CRT),
4. volá funkce Pailliera se správně nastavenými parametry.

Dále se v mikroC projektu nepodařilo nastavit vkládání hlavičkových souborů, aby si je kompilátor našel automaticky (např. `#include <bigi.h>`). V implementaci Pailliera bylo nutné upravit vkládání hlavičkových souborů s relativní cestu vůči implementaci (např. `#include "../bigi/bigi.h"`).

V implementaci Pailliera popsané v sekci 3.2 bylo rozšířeno použití makra `RUNNING_EXAMPLE`. Struktury pro uložení klíčů a výpočtů simulátoru HW RSA jsou obsaženy přímo v `paillier.c`. Pokud makro zadefinováno není, je možné měnit hodnoty proměnných `RSA_BITLEN` a `RSA_BYTELEN`, aby se mohla dynamicky měnit délka klíčů simulátoru HW RSA v Paillierovi. Pokud makro `RUNNING_EXAMPLE` zadefinováno je, simulátor HW RSA poté používá pevně danou délku bloku.

3.4 Programování Flash paměti *NewAE targetu*

V podsekci 2.3.4 byla diskutována volba USB-SPI převodníku *CH341A* spolu s volbou vhodné aplikace, která využívá převodník pro programování Flash pamětí. Zvolená aplikace [49] má přívětivé uživatelské rozhraní, viz Obrázek 3.14.

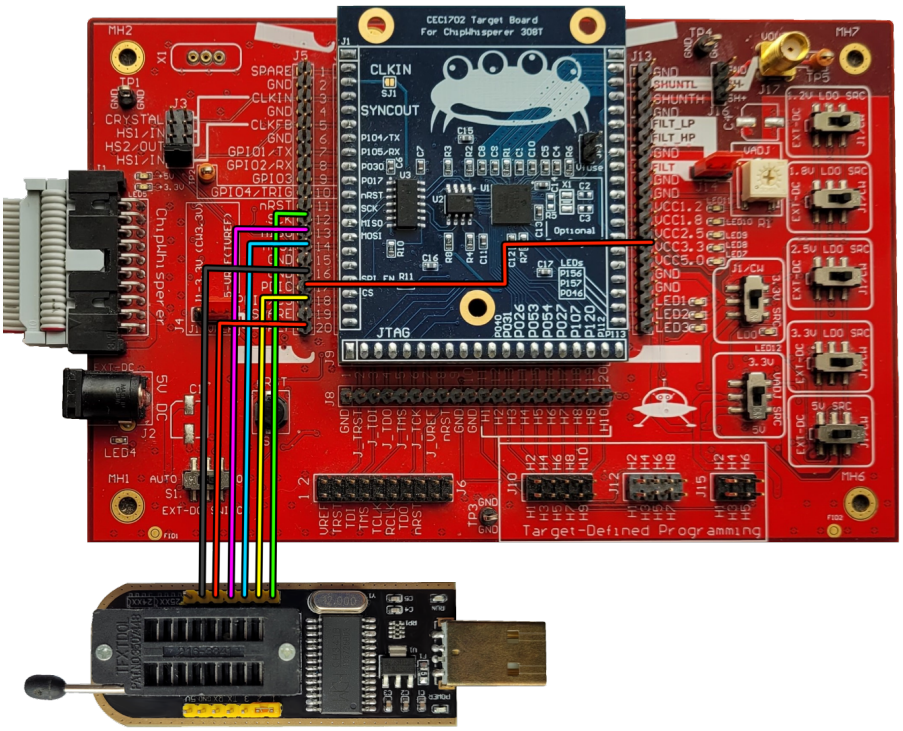
Aplikace má přístupné zdrojové kódy [50], kdy vrstva aplikace napsaná v C# řídí USB-SPI převodník pro programování Flash pamětí. Některé části kódu jsou dokumentovány v čínštině, což ale není překážkou. Metody mají srozumitelné anglické názvy, taktéž i jednotlivé proměnné a třídy jsou anglicky pojmenovány. Aplikace ale nepodporuje Flash paměť SST26VF016B [45], která se nachází v *NewAE target* s CEC1702. Paměť totiž obsahuje ochranu proti zápisu, kterou je nutné před každým programováním paměti vypnout. Podporu paměti SST26VF016B v aplikaci bylo nutné doimplementovat.

Zapojení převodníku *CH341* Aby aplikace mohla s pamětí přes SPI komunikovat, je nutné převodník *CH341* správně připojit k desce UFO obsahující NewAE target. Deska UFO zpřístupňuje rozhraní SPI pro programování Flash paměti *NewAE targetu*. Schéma zapojení je na Obrázku 3.13. Je využito 6 konektorů pro signály `MISO`, `MOSI`, `CS`, `CLK`, `GND` a `3V3`. V dokumentaci *NewAE targetu* pro CEC1702 [31] je informace, že musí být pin `nRST` připojen k zemi pro zpřístupnění programování Flash paměti. Toto ale neplatí a je nutné mít naopak pin `PDIC` připojen ke 3.3V zdroji napájení. UFO deska musí být také připojena 10pinovým konektorem k ChipWhispereru-Lite, jinak Flash paměť nereaguje. Důvod připojení k ChipWhispereru-Lite je přisuzován nedostatečnému napájecímu výkonu převodníku. Připojení ChipWhispereru-Lite může také odblokovat průchod signálu z převodníku do Flash paměti.

Úprava aplikace pro programování Flash pamětí První částí úprav bylo doplnění informací o paměti SST26VF016B do konfiguračního souboru `chiplist.xml`. Byl přidán typ příkazů `26XX`, který v aplikaci nastaví použití metod pro programování paměti SST26VF016B. Tyto úpravy dovolují vybrat paměť SST26VF016B v menu aplikace a nastavit automaticky velikost paměti a stránek paměti. Aplikace následně použije správné typy příkazů pro modifikaci obsahu paměti. Doplnění konfiguračního souboru zpřístupnilo automatické detekování paměti. Aplikace po stisku tlačítka „*Auto Detect*“ odešle do připojené paměti příkaz `JEDEC ID = 0x9F`. Paměť následně vrátí `ID` výrobce, typ paměti a své `ID`. Aplikace se poté automaticky nastaví pro použití správných příkazů pro modifikaci obsahu paměti.

Implementace [50] již obsahuje podporu pro paměť SST**25**VF016B [61], která využívá stejnou instrukční sadu jako SST**26**VF016B. SST**25**VF016B ale neobsahuje ochranu proti zápisu do paměti. Doplnění implementace v souboru `UsbManager.cs` o možnost programování paměti SST26VF016B spočívá v přidání následujících metod:

1. `DisableBlockProtection()` - Před každým zápisem do paměti odesílá příkaz `ULBPR = 0x98`, který povolí zápis do všech bloků paměti. Tento postup je zmíněn v dokumentaci k NewAE targetu CEC1702 [31].

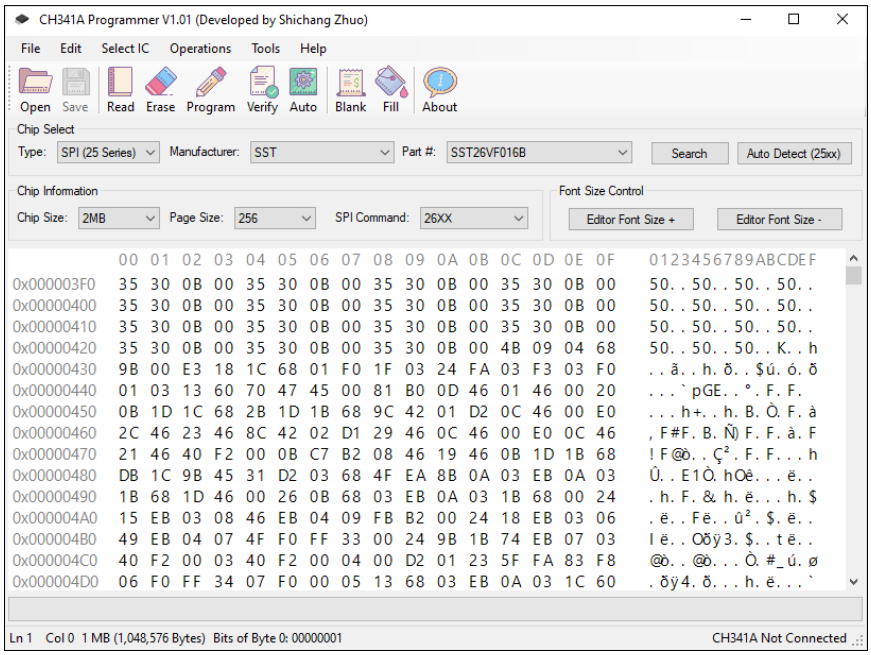


Obrázek 3.13: Připojení USB-SPI převodníku *CH341A* k desce UFO obsahující *CW308T-CEC1702 NewAE target*

2. `EraseSPIChip26()` - Využívá `DisableBlockProtection()` pro povolení modifikace paměti a následně je odeslán příkaz `CE = 0xC7` pro její celé vymazání.

3. `WriteDataSPI26()` - Před voláním této metody je vždy zavolána metoda `EraseSPIChip26()`, která povolí zápis do paměti a celou paměť vymaže. `WriteDataSPI26()` obstarává zápis do paměti a je doplněna o verifikaci každé stránky ihned po zápise. Také obsahuje možnost přeskočení zápisu stránky, pokud se mají zapsat pouze bajty s hodnotou `0xFF` (tuto hodnotu má každý bajt paměti po jejím vymazání).

Pro čtení paměti je využita existující metoda `ReadDataSPI25()`. Pro získání ID paměti je využita také již implementovaná metoda `ReadIdSpiMode25()`. Upravená implementace je přiložena na médiu spolu se spustitelným souborem ve složce `4_Flash_programmer`.



Obrázek 3.14: Uživatelské rozhraní programátoru Flash pamětí obsahující nastavení pro paměť SST26VF016B

Kapitola 4

Testování

Nejdříve byla otestována funkčnost jednotlivých upravených implementací *bigi* a Pailliera s využitím souborů `example.c` a jazyka Ruby. Následně byl otestován *firmware* pro CEC1702, kdy bylo využito jak ChipWhisperer API, tak jazyka Python. V poslední části kapitoly je popsáno testování Aplikace pro programování Flash paměti SST26VF016B.

4.1 Knihovna pro práci s velkými čísly – *bigi*

Pro *bigi* již existuje soubor `example.c`, který volá jednotlivé operace, které jsou v *bigi* implementovány. Makrem `EXAMPLE_BITLEN` je nastavena požadovaná délka čísel. Výstup programu je ve formátu jazyka Ruby (viz Obrázek 4.1) aby mohl být následně tento výstup zpracován a výsledky operací ověřeny právě využitím Ruby spolu s modulem *OpenSSL*. Nejjednodušší postup pro ověření je přesměrovat výstup sériové linky do souboru s příponou `.rb` a následně zavolat z příkazové řádky `ruby <nazev>.rb`. Jednoduchý výpis (viz Obrázek 4.2) následně informuje o správnosti operací i s údajem, jak dlouho jednotlivá operace trvala.

4.2 Paillierův kryptosystém

Stejný postup testování jako u *bigi* byl použit i u upravené implementace Paillierova kryptosystému. V souboru `example.c` se volí délka klíče Pailliera pomocí makra `EXAMPLE_BITLEN` a také se volí, jestli se provede šifrování naivní či využívající CRT.

Zda se testuje čistě SW implementace CRT varianty či s HW akcelerátorem, lze nastavit pomocí případného definování makra `RSA_ACCELERATION`. V neposlední řadě je nutné nastavit správnou délku RSA akcelerátoru – jak simulovaného, tak HW – pomocí makra `RSA_BITLEN` v souboru `paillier.h`. Posledním makrem, které je nutné zadefinovat, je `RUNNING_EXAMPLE`, které na-

```
...
puts "---   DEBUG / TEST mode ON                                    ---"
puts "      128-bit input values                                          "; puts
# Loading numbers ...

#  - loaded A
#  - loaded B
#  - loaded P
#  - loaded Q
#  - loaded N
# ------------------------------------------------------------------------------
print "Add / Subtr Fit:    "
print 0x00000000000000000000000004e8839e2876bcb217511f129c68265a6 +
0x00000000000000000000000000a892170d2e9809870923f82e150ac166 ==
0x00000000000000000000000000f71a50efb603d4a87e35e957db8d270c ? "PASS".green.bold :
"FAIL".red.bold
puts " (         0 ms)"
# -------
...
```

Obrázek 4.1: Ukázka zformátovaného výpisu CEC1702 přes UART pro následné zpracování výsledků *bigi* pomocí Ruby

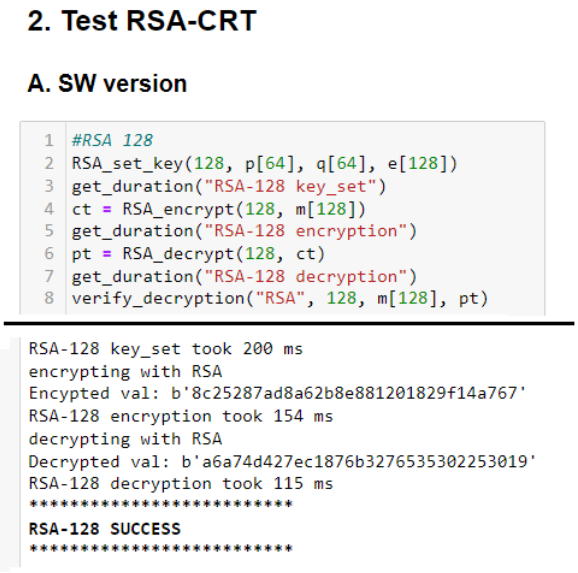
```
---   DEBUG / TEST mode OFF
      1024-bit input values

Add / Subtr Fit:    PASS (         2 ms)
                    PASS (         2 ms)
Mod red (plain):    PASS (         2 ms)
Mult Fit:           PASS (         2 ms)
Mod Mult (plain):   PASS (        91 ms)
Mod Mult (Mont):    PASS (         9 ms)
Mod Exp (plain):    PASS (    144872 ms)
Mod Exp (Mont):     PASS (     12072 ms)
Mod Inv:            PASS (     63799 ms)
Int Div:            PASS (         2 ms)
GCD:                PASS (        91 ms)
CLOCKS_PER_SEC = 48000
```

Obrázek 4.2: Ukázka zobrazení výsledků po testování *bigi* pomocí Ruby

staví vždy pevnou velikost polí při kompilaci. Makro `RUNNING_EXAMPLE` také umožní výpis náhodně generovaného čísla, které je použito ve funkcích a je potřeba následné zpracování pomocí Ruby. Následné ověření správnosti šifrování je totožné s testováním *bigi*.

4.3 *Firmware* s komunikačním protokolem

Pro ověření a zároveň demonstraci funkcionality *firmware* pro CEC1702 byl vytvořen Jupyter Notebook `BC_CEC1702_FW_TESTING.ipynb`, který využívá sadu ChipWhisperer a *NewAE target* s CEC1702. Při testování jsou provedeny následující kroky:



2. Test RSA-CRT

A. SW version

```
#RSA 128
RSA_set_key(128, p[64], q[64], e[128])
get_duration("RSA-128 key_set")
ct = RSA_encrypt(128, m[128])
get_duration("RSA-128 encryption")
pt = RSA_decrypt(128, ct)
get_duration("RSA-128 decryption")
verify_decryption("RSA", 128, m[128], pt)
```

```
RSA-128 key_set took 200 ms
encrypting with RSA
Encypted val: b'8c25287ad8a62b8e881201829f14a767'
RSA-128 encryption took 154 ms
decrypting with RSA
Decrypted val: b'a6a74d427ec1876b3276535302253019'
RSA-128 decryption took 115 ms
**************************
RSA-128 SUCCESS
**************************
```

Obrázek 4.3: Ukázka testování firmware pro CEC1702 s využitím IPython Notebooku a ChipWhisperer API [40]

1. Naprogramování CEC1702 s požadovanou verzí *firmware* – čistě SW či s využitím HW akcelerátoru. Programování CEC1702 je diskutováno v sekci 2.3.

2. Nastavení připojeného ChipWhispereru-Lite, který následně komunikuje s *NewAE targetem*.

3. Otestování *SimpleSerial* protokolu [59] ve verzi 1.1. je provedeno využitím ChipWhispereru-Lite, který *SimpleSerial* používá.

4. Definice funkcí pro obstarání komunikace s CEC1702 a definice hodnot použitých pro dešifrování. V Notebooku jde o sekci „*1. Connecting to ChipWhisperer and CEC1702*“.

5. Otestování podporovaných šifer ve *firmware* se všemi podporovanými délkami klíčů je provedeno pomocí spouštění buněk Notebooku v sekcích:

   a) „*2. Test RSA-CRT*“,

   b) „*3. Paillier - Plain testing (Only SW Implementation)*“,

   c) „*4. Paillier - CRT testing*“.

6. V rámci testování jsou použity i příkazy pro změnu používané šifry a změnu délky šifry. Jsou tedy otestovány všechny podporované příkazy *firmware*.

4.4 Aplikace pro programování Flash paměti

Aplikace ze sekce 3.4 byla otestována pouze pro programování Flash paměti *NEwAE targetu* s CEC1702 – SST26VF016B [45]. *NEwAE target* je vsazen do desky UFO ze sady ChipWhisperer. Po zapojení převodníku *CH341A* k desce UFO bylo otestováno zapisování do paměti, kdy se opakovaně provedlo:

1. Zápis binárního souboru
2. odpojení převodníku *CH341A* od desky UFO a vypnutí napájení desky
3. opětovné připojení převodníku i napájení desky
4. přečtení paměti
5. ověření, že soubor, který byl zapisován a soubor, který vznikl přečtením paměti, jsou binárně ekvivalentní

Otestování programování paměti bylo provedeno nahráním jednoduchého programu ovládajícího blikání LED. Ovládaná LED se nachází na UFO desce. LED se vždy rozblikala po odpojení převodníku *CH341A* a opakovaném restartu *NEwAE targetu*.

Kapitola 5

Útok postranními kanály

Tato kapitola obsahuje popis HW i SW výbavy pro úspěšné provedení útoků injekcí chyb na napájení a zdroji hodinového signálu. Dále jsou v této kapitole popsány nutné úpravy implementace RSA-CRT pro efektivnější provedení útoků. Následuje část s popisem útoků na mikrokontrolér STM32F3. Poslední částí kapitoly je popis realizace útoků na kryptoprocesor CEC1702.

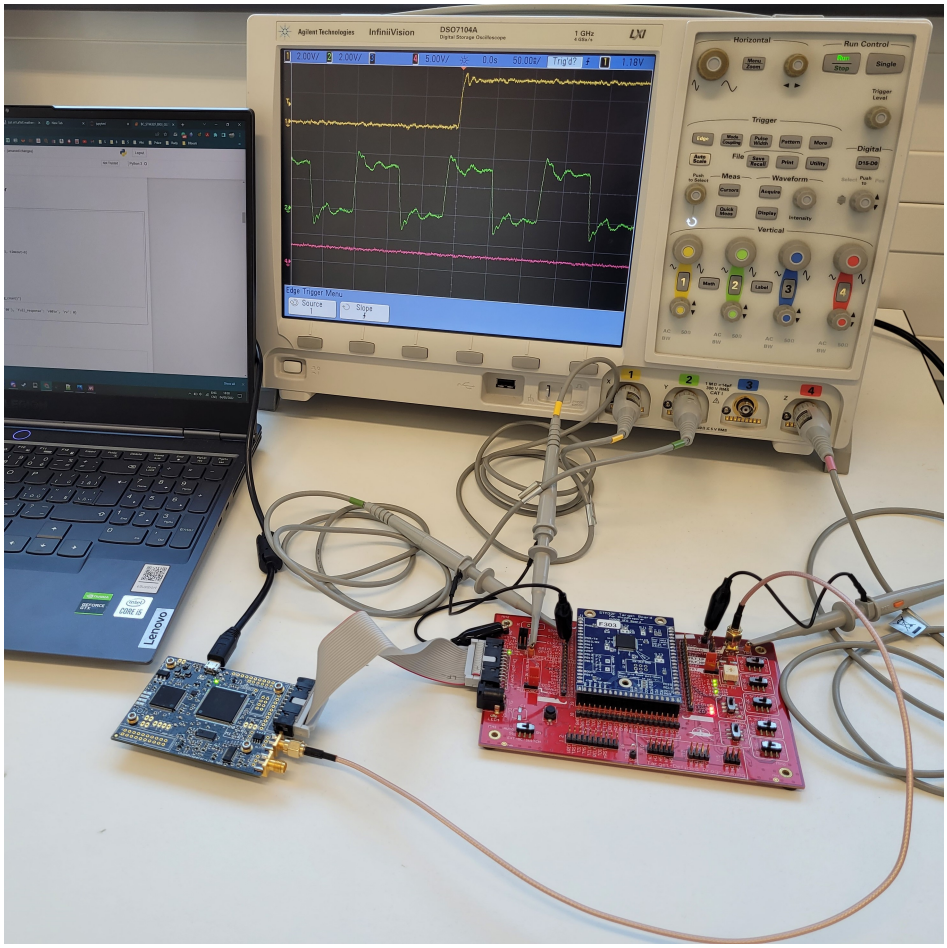
Veškerá implementace *firmware* spolu s IPython Notebooky pro realizaci útoků na STM32F3 je uložena na médiu ve složce `5_STM32F3_utoky`. Veškerá implementace *firmware* spolu s IPython Notebooky pro realizaci útoků na CEC1702 je uložena na médiu ve složce `6_CEC1702_utoky`.

5.1 Vybavení a použitý SW pro provedení útoků

Následuje výčet vybavení použitého pro realizaci útoků:

1. ChipWhisperer-Lite obsahující generátor zákmitů,
2. deska UFO pro zasazení modulů [32],
3. modul pro UFO *NewAE target* s STM32F3 [39] (dále jen STM332F3),
4. modul pro UFO *NewAE target* s CEC1702 [31] (dále jen CEC1702),
5. koaxiální kabel 50 cm s SMA-F konektory pro zavádění zákmitů (zkratů) na napájení ze sady ChipWhisperer *SCAPACK-L1* [3],
6. USB-SPI převodník *CH341A* [48] pro programování Flash paměti modulu s CEC1702,
7. osciloskop Agilent MSO 7104A [62] pro sledování průběhu signálu.

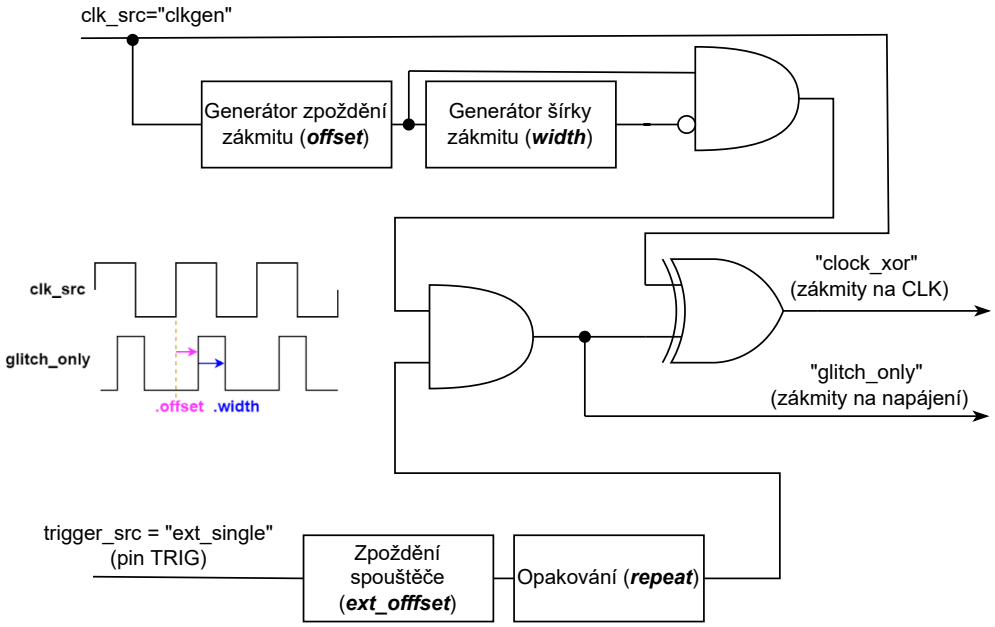
Útoky jsou realizovány pomocí IPython Notebooků verze 7.23.0, které jsou součástí prostředí ChipWhisperer verze 5.5. Programování Flash u STM32F3



Obrázek 5.1: Sestava pro realizaci útoků pomocí injekce chyb na STM32F3

je realizováno pomocí ChipWhisperer API, které obsahuje kompilátor pro ARM i implementaci programátoru Flash pamětí pro mikrokotroléry řady STM32F. Pro kompilaci programu pro CEC1702 je využito mikroC IDE a pro programování Flash paměti u CEC1702 je použita aplikace popsaná v sekci 3.4.

Původní útok na RSA-CRT v rámci lekce ChipWhisperer [63] cílí na podepisování *zahashované* zprávy m s využitím RSA-CRT. Útoky popsané v této práci jsou modifikací útoku ze zmíněné lekce a jsou cíleny na autorovy vlastní implementace RSA-CRT využívající *bigi*. Cílem útoků na STM32F3 i CEC1702 je získání soukromého exponentu d, který se využívá pro dešifrování c pomocí RSA-CRT (nikoliv při podepisování). Úspěšného útoku je docíleno pomocí modifikace první části dešifrování, kdy je využíváno prvočíslo p. Dešifrování poté vrátí zprávu m'. Hodnota m' je poté využita pro výpočet druhého prvočísla $q = \gcd((c - m')^e, n)$. Následně je možné získat úplný tajný klíč K_T. Jedná se o modifikaci Bellcore útoku popsaného v podsekci 1.3.3. Úprava výpočtu prvočísla q je nutná, jelikož původní verze Bellcore útoku pracuje s podepisováním



Obrázek 5.2: Zjednodušené schéma generátoru zákmitů obsaženého v ChipWhisperer-Lite, jako zdroj byl použit Obrázek [64]

pomocí RSA-CRT.

5.1.1 Generátor zákmitů v ChipWhisperer-Lite

Zjednodušené schéma generátoru je na Obrázku 5.2. Generátor zákmitů je spuštěn, pokud je na desce UFO zaktivován pin `TRIG` (objeví se náběžná hrana). Logická hodnota pinu `TRIG` je řízená voláním funkcí `trigger_high()` a `trigger_low()` v implementaci *firmware* pro STM32F3 a CEC1702. Pro generování zákmitů na zdroji hodinového signálu je využíván výstup generátoru označován `glitch_xor`, který je napojen na zdroj hodinového signálu zařízení pod útokem. Pro generování zákmitů na zdroji napájení je využíván výstup generátoru označován `glitch_only`, kdy je vygenerovaný zákmit převeden ve zkrat na napájení zařízení.

Pro úspěšné provedení útoku je třeba nastavit experimentálně následující parametry (podrobnější popis parametrů je uveden v [40]):

1. `offset` - Vzdálenost od náběžné hrany, která je dána poměrem k šířce jednoho hodinového taktu. Hodnota je zadávána v procentech v rozmezí -48,9 % až +48,9 %. Záporná hodnota značí posun před náběžnou hranu hodin.

```
#nastaveni ChipWhisperer, programování tagetu, ..

scope.glitch.clk_src = 'clkgen'
scope.glitch.trigger_src = 'ext_single'
scope.glitch.output = "clock_xor"
scope.io.hs2 = "glitch"

scope.glitch.width = 2
scope.glitch.offset = -14
scope.glitch.ext_offset = 120
scope.glitch.repeat = 1

print(scope.glitch)
```

```
clk_src     = clkgen
width       = 1.953125
width_fine  = 0
offset      = -14.0625
offset_fine = 0
trigger_src = ext_single
arm_timing  = after_scope
ext_offset  = 120
repeat      = 1
output      = clock_xor
```

Obrázek 5.3: Ukázka nastavení generování zákmitů na zdroji hodinového signálu v prostředí IPython Notebook s využitím API od ChipWhisperer

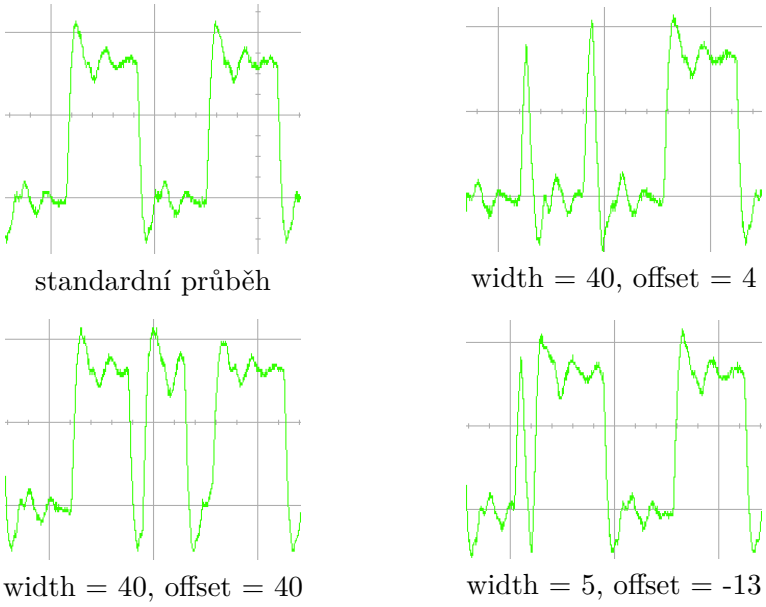
2. `width` - Šířka vygenerovaného zákmitu, která je dána poměrem k šířce jednoho hodinového taktu. Hodnota je zadávána v procentech v rozmezí -48,9 % až +48,9 %. Pokud je zadána záporná hodnota, je šířka zákmitu rovna šířce hodinového taktu zkrácená o poměr daný parametrem.

3. `ext_offset` - Zpoždění od vyvolání zákmitu, hodnota je udávána v hodinových taktech generátoru zákmitů nastavené.

4. `repeat` - Udává kolik taktů generátoru se bude vyvolaný zákmit opakovat od jeho vyvolání.

Reálná podoba modifikovaného hodinového signálu za použití různých kombinací parametrů `width` a `offset` je znázorněna na skupině Obrázků 5.4 .

Dále je třeba nastavit několik dalších parametrů generátoru (viz Obrázek 5.3). Nastavení všech parametrů generátoru zákmitů probíhá přes proměnné modulu z ChipWhisperer API – `chipwhisperer.scope.glitch`. Proměnné jsou podrobněji popsány v dokumentaci API [40]. Nastavení všech potřebných parametrů generátoru zákmitů v rámci IPython Notebooku je znázorněno na Obrázku 5.3.



Obrázek 5.4: Ukázky zákmitů na zdroji hodinového signálu

5.1.2 Použitý *firmware* pro efektivní realizaci útoků

Pro úspěšný útok injekcí chyb je třeba najít vhodné parametry generátoru zákmitů (`offset`, `width`, `ext_offset` a `repeat`). Je nevhodné útočit z počátku implementaci RSA-CRT s klíčem dlouhým 1024 bitů, a to z několika na sebe navazujících důvodů:

- Dešifrování pomocí čistě SW implementace RSA-CRT na STM32F3 i CEC1702 trvá několik vteřin a možných kombinací parametrů generátoru může být i několik set tisíc.
- Při útoku pomocí zákmitu na hodinovém signálu na ST32F3 trvá otestování jedné kombinace parametrů s kontrolou dešifrování v průměru 14,4 s.
- Není zaručeno, že se parametry generátoru zákmitů podaří najít při prvním průchodů jejich kombinací.
- Celková doba hledání parametrů generátoru by se tedy mohla vyšplhat i na několik týdnů.

Pro efektivnější hledání vhodných parametrů generátoru zákmitů bylo vytvořeno několik verzí redukovaného *firmware*. To umožňuje rychlejší nalezení jednotlivých parametrů generátoru, které vedou k modifikaci šifrování. Cílová implementace útoků je vždy RSA-CRT s 1024bitovým klíčem.

Dále je pro snadnější cílení útoku část kódu (jejíž provedení má být ovlivněno zákmity) ohraničena voláním funkcí `trigger_high()` a `trigger_low()`,

Tabulka 5.1: Počet taktů částí RSA-CRT na STM32F3 s frekvencí 7,386 MHz

Typ operace	Průměrný počet taktů	Čas trvání
celé dešifrování	104 505 999	14,14 s
výpočet m_1	49 779 313	6,74 s
z toho výpočet c_p	10 270	**1,39 ms**
výpočet m_2	53 840 771	7,29 s
dopočet h, h_q, m	817 797	110,7 ms

které ovládají pin `TRIG`. Pro otestování zranitelnosti implementace *bigi* [2] byly zvažovány funkce:

1. `bigi_mod_red()` implementující základní modulární redukci,
2. `bigi_mod_exp_mont()` implementující modulární mocnění za použití Montgomeryho domény,
3. `bigi_mod_exp()` implementující základní modulární mocnění.

Všechny zmíněné funkce jsou následně využity i v implementaci RSA-CRT (v částech, které je potřeba ovlivnit zákmitem pro následnou úspěšnou realizaci Bellcore útoku pro získání tajného exponentu d). Pro otestování zranitelnosti byla vybrána první funkce – `bigi_mod_red()`. Výpočet c_p pomocí funkce `bigi_mod_red()` není časově náročný (viz Tabulka 5.1), a tudíž dovoluje otestování zanesení zákmitu ve všech hodinových taktech výpočtu.

Jednotlivé verze redukovaných *firmware* jsou popsány níže a každá obsahuje pojmenování, které je následně využito k identifikaci v následných sekcích a tabulkách s výsledky jednotlivých útoků. Všechny verze využívají komunikační protokol *SimpleSerial v1.1* [59]. Na přiloženém médiu jsou uloženy projekty `BIGI_Glitching` a `RSA-CRT_Glitching` ve verzi pro STM32F3 a pro CEC1702, ze kterých jsou za použití příslušných preprocesorových maker (přepínačů) jednotlivé verze *firmware* vytvořeny:

1. `bigi_128_ret` - Základem upraveného původního *firmware* [65] je funkce s jednoduchou modulární redukcí 128bitového čísla na 64bitové využívající *bigi* funkci `bigi_mod_red()`. Výpočet je prováděn vždy nad stejnými zadefinovanými čísly a je vyvolán přijetím příkazu `b`. Funkce vrací pouze odpovědi `r00` (pokud výpočet nebyl nijak ovlivněn) a `r01` (v opačném případě) pro co největší redukci komunikace přes *UART*, která je v poměru se samotným výpočtem několikanásobně pomalejší. Implementace se nachází v `BIGI_Glitching` a nejsou použité žádné přepínače pro kompilaci.
2. `bigi_128_full` - Modifikace první verze, kdy výsledek výpočtu není kontrolován, ale je celý odeslán. Odpověď tedy je `r39...F4`. Implementace je v `BIGI_Glitching` a je použit přepínač `RETURN_BIGI_RESULT`.

3. `RSA_256_full` - Modifikace předchozí varianty, kdy je již prováděno celé dešifrování. Implementace je v `RSA-CRT_Glitching` a jsou použity přepínače `M1_GLITCH` a `RSA_BITS 256`.

4. `RSA_1024_ret` - Modifikace třetí varianty, kdy je využito klíče dlouhého 1024 bitů. Díky omezení pro přenesení jednoho bloku dat, jsou výsledná dešifrovaná data rozdělena na 3 bloky dlouhé 32, 48 a 48 B. Odeslání prvního – nultého – bloku je provedeno automaticky po dešifrování. Pro získání dalších bloků je třeba přijmout příkaz `1` a `2`. Jsou tedy použity přepínače `CP_GLITCH` a `RSA_BITS 1024`.

5. `RSA_1024_full` - Navazuje na předchozí varianty, kdy je opět využito klíče dlouhého 1024 bitů a je prováděno celé dešifrování. Pro kompilaci projektu `RSA-CRT_Glitching` jsou použity přepínače `M1_GLITCH` a `RSA_BITS 1024`.

6. `RSA_1024_HW` - Jako jediná varianta využívá i HW akcelerátor kryptografických operací CEC1702, a tudíž není kompatibilní s STM32F3. Generátor zákmitů je spouštěn souběžně se spuštěním dešifrování po nahrání všech potřebných hodnot (data k dešifrování, potřebné části klíče, ...) do jednotlivých slotů akcelerátoru. Firmware také obsahuje funkci pro dešifrování bez CRT pro následné porovnání času trvání, která je vyvolána přijetím příkazu `d`. Implementace se nachází v projektu `RSA-CRT_Glitching` a je použit přepínač `HW_IMPLEMENTATION`.

V `BIGI_Glitching` projektu je zaktivován pin `TRIG` vždy před voláním funkce `bigi_mod_red()`. Jakmile funkce dokončí výpočet je pin `TRIG` deaktivován. V `RSA-CRT_Glitching` projektu je zaktivován pin `TRIG` vždy před operací $c_p = c \bmod p$ (viz podsekce 1.1.1.2) a po konci operace je opět deaktivován.

Implementace *firmware* pro STM32F3 i CEC1702 používají stejné hodnoty klíčů pro RSA-CRT, které je útoky injekcí chyb snaha prolomit. Dešifrovaná jsou i stejná data. Použité hodnoty pro 256b RSA-CRT jsou uvedeny v Tabulce 5.2. Použité hodnoty pro 1024 RSA-CRT jsou uvedeny v Tabulce 5.3.

5.2 Realizace útoků na STM32F3

Jak již bylo zmíněno, útok na implementaci RSA-CRT na STM32F3 existuje v rámci lekcí, které jsou dostupné na [63] a využívá zákmitů na zdroji hodinového signálu. Pro větší jednotnost útoků na STM32F3 i CEC1702 byla vytvořena čistě SW implementace RSA-CRT pomocí *bigi* i na STM32F3 a následně i její redukované varianty (popsané v podsekci 5.1.2)Ty slouží pro usnadnění a urychlení postupu k úspěšnému útoku. V rámci průpravy na útok pomocí zákmitu na zdroji napájení, který je možný realizovat i na CEC1702, byl vybrán i útok pomocí zákmitu na zdroji hodinového signálu. Nesprávně

Tabulka 5.2: Hodnoty v hexadecimálním formátu použité u 256b RSA-CRT

p	CCE6F3803DED54B0779689B698365DC5
q	DEC874AB814EEEB747E98B515D8289DF
d_p	A990FAB4070B8FC955BE0B8B431DC2AD
d_q	CFC8FCF03D41A66A717B04C2D5C97BD7
q_{inv}	18FE4F16552854B630D74C272EB84173
e	10001
n	B250B8F95DFE81B752C93C92610CED1A 9DA77C8DD0FD22ECA9445B84DE941B9B
d	191F18404186C69D9E332DAE6DAFD583 6B2B892B9BD6B166D74C48A5F6645379
c	3E2F76BB13EADA988DE18A080063552F B92A3DF1609E8A22946339C0D46CAAFF
m	1980BF37E445B41C490A49CB7BA3658D A97D0EDF7E29C09D6F7723DC75D03E0D

Tabulka 5.3: Hodnoty v hexadecimálním formátu použité u 1024b RSA-CRT

p	C36D0EB7FCD285223CFB5AABA5BDA3D82C01CAD19EA484A87EA4377637E75500 FCB2005C5C7DD6EC4AC023CDA285D796C3D9E75E1EFC42488BB4F1D13AC30A57
q	C000DF51A7C77AE8D7C7370C1FF55B69E211C2B9E5DB1ED0BF61D0D9899620F4 910E4168387E3C30AA1E00C339A795088452DD96A9A5EA5D9DCA68DA636032AF
d_p	C1ACF567564274FB07A0BBAD5D26E2983C94D22288ACD763FD8E5600ED4A702D F84198A5F06C2E72236AE490C93F07F83CC559CD27BC2D1CA488811730BB5725
d_q	4959CBF6F8FEF750AEE6977C155579C7D8AAEA56749EA28623272E4F7D0592AF 7C1F1313CAC9471B5C523BFE592F517B407A1BD76C164B93DA2D32A383E58357
q_{inv}	9AE7FBC99546432DF71896FC239EADAEF38D18D2B2F0E2DD275AA977E2BF4411 F5A3B2A5D33605AEBBCCBA7FEB9F2D2FA74206CEC169D74BF5A8C50D6F48EA08
e	10001
n	9292758453063D803DD603D5E777D7888ED1D5BF35786190FA2F23EBC0848AEA DDA92CA6C3D80B32C4D109BE0F36D6AE7130B9CED7ACDF54CFC7555AC14EEBAB 93A89813FBF3C4F8066D2D800F7C38A81AE31942917403FF4946B0A83D3D3E05 EE57C6F5F5606FB5D4BC6CD34EE0801A5E94BB77B07507233A0BC7BAC8F90F79
d	C36D0EB7FCD285223CFB5AABA5BDA3D82C01CAD19EA484A87EA4377637E75500 FCB2005C5C7DD6EC4AC023CDA285D796C3D9E75E1EFC42488BB4F1D13AC30A57 C36D0EB7FCD285223CFB5AABA5BDA3D82C01CAD19EA484A87EA4377637E75500 FCB2005C5C7DD6EC4AC023CDA285D796C3D9E75E1EFC42488BB4F1D13AC30A57
c	1701E0C7306E79A3FBEF9C26B12C072B2F33B613941978404067E9484859EBA1 8669E1003A61E05B91F106D16F749B52AFDC5B66CCF2FAA75D630694DD2B425F 6BCD8D896BD1F7F15F65DD64DCAA17F1F3B826EC78D66D452132CAE92D35B642 146ACDD4B7CF218A42B57122149CD98A13C9018CD99ECDE00A699C5D6CF59666
m	1114444444444444444444444444444444444444444444444444444444444444 4444444444444444444444444444444444444444444444444444444444123456 1114444444444444444444444444444444444444444444444444444444444444 4444444444444444444444444444444444444444444444444444444444123456

nastavený útok pomocí zákmitu na napájení navíc může nenávratně poškodit ChipWhisperer-Lite i mikrokontrolér / kryptoprocesor samotný (viz podsekce 1.3.2). Nabyté zkušenosti z útoku pomocí zákmitu na zdroji hodinového signálu zvyšují pravděpodobnost, že se poškození zařízení předejde.

Pro útok na `bigi_128_ret` a následně na `bigi_128_full` byl vytvořen IPython Notebook `BC_STM32F_BIGI_GLITCHING`, který je založen na Noteboocích z ChipWhisperer lekcí [22, 66, 67]. Notebook je společný pro útok pomocí zákmitů na napájení i zdroji hodinového signálu a skládá se z jednotlivých částí:

1. Připojení ChipWhisperer-Lite,
2. naprogramování STM32F3 správnou verzí *firmware* podle hodnoty přepínače `FULL_RESPONSE`,
3. otestování funkčnosti *firmware* odesláním příkazu k výpočtu a kontrolou výsledku (v závislosti na použité verzi *firmware*),
4. nastavení generátoru zákmitů (v závislosti na použitém útoku), včetně parametru `repeat`,
5. smyčka pro otestování kombinací parametrů `offset`, `width`, `ext_offset` podle nastavených intervalů hodnot u jednotlivých parametrů (v závislosti na použitém útoku),
6. odpojení ChipWhispereru-Lite od Notebooku.

Pro útoky na `RSA_256_ret`, `RSA_256_full`, `RSA_1024_ret`, `RSA_1024_full` byly vytvořeny IPython Notebook `BC_STM32F_RSA_CLOCK_GLITCHING` spolu s Notebookem `BC_STM32F_RSA_VOLTAGE_GLITCHING`, které jsou založeny na Noteboocích z ChipWhisperer lekcí [63, 66, 67]. Notebooky mají stejnou strukturu, ale liší se typem použitého útoku. Skládají se z jednotlivých částí:

1. Připojení ChipWhisperer-Lite,
2. naprogramování STM32F3 správnou verzí *firmware* podle hodnot přepínačů `RSA_BITS` a `ONLY_CP`,
3. otestování funkčnosti *firmware* odesláním příkazu k šifrování a kontrolou výsledku (v závislosti na použité verzi *firmware*),
4. nastavení generátoru zákmitů (v závislosti na použitém útoku), včetně parametru `repeat`,
5. smyčka pro otestování kombinací parametrů `offset`, `width`, `ext_offset` podle nastavených intervalů hodnot u jednotlivých parametrů (v závislosti na použitém útoku),
6. je provedeno vypočítání tajného klíče v případě úspěšné modifikace dešifrování u `RSA_256_full` a `RSA_1024_full` ,
7. odpojení ChipWhispereru-Lite od Notebooku.

Tabulka 5.4: Parametry pro útok pomocí zákmitu na zdroji hodinového signálu na mikrokontroléru STM32F3

Typ	`width`	`offset`	`ext_offset`	`repeat`	Čas[5]
`bigi_128_ret`	2	- 14	771, 831	1	23,6 ms
`bigi_128_full`	2	- 14	196, 519	1	17 ms
`RSA_256_ret`	2	-12	624	1	27,1 ms
`RSA_256_full`	2	-12	254, 642	1	981 ms
`RSA_1024_ret`	2	-12	288, 560	1	90,6 ms
`RSA_1024_full`	2	-12	224	1	14,154 s

5.2.1 Útoky pomocí zákmitů na zdroji hodinového signálu

Pro útok je využit modul CW308T-STM32F3, který má nastaven externí zdroj hodinového signálu, jenž je generován v ChipWhisperer-Lite. Generovaná hodinová frekvence je nastavená 7,386 MHz a je umožněna její modifikace pomocí interního generátoru zákmitů.

Nejprve byl útok proveden na `bigi_128_ret` a následně na `bigi_128_full`. v rámci Notebooku `BC_STM32F_BIGI_GLITCHING` (sekce `CLOCK-GLITCHING`). Při otestování 100 612 kombinací, modifikovalo výpočet 1474 z nich (úspěšnost cca 1,47 %). Při nastavených hodnotách `width` = 2 a `offset` = -14 bylo zavedeno nejvíce chyb do výpočtů. Při testování parametrů generátoru byl zákmit zaváděn jen jednou za výpočet (`repeat` = 1). Výčet parametrů generátoru, které vedly k úspěšné modifikaci výpočtu, jsou vypsány v Tabulce 5.4.

Pro další útoky byl využit Notebook `BC_STM32F_RSA_CLOCK_GLITCHING`. Zúžené intervaly parametrů generátoru byly nejprve testovány s `RSA_256_ret` a `RSA_1024_ret`. Pokud byla hodnota `offset` = -14, byl výpočet modifikován častěji u `RSA_256_ret` i `RSA_1024_ret`.

U `RSA_256_full` a `RSA_1024_full` byl opět aplikován stejný postup, kdy se testovalo zavádění chyb s pevnými hodnotami `width` = 2 a `offset` = -12 a měnil se pouze ofset (`ext_offset`) zavádění zákmitu v průběhu výpočtu c_p. V případě `RSA_256_full` došlo k úspěšné modifikaci dešifrování (viz Obrázek 5.5), modifikovaná zpráva byla uložena a dále využita v poslední části Notebooku. Využitím upraveného Bellcore útoku bylo dopočítáno prvočíslo q a následně i tajný exponent d. U `RSA_1024_full` došlo také k získání klíče a to při nastaveném `ext_offset` = 224. Hodnoty generátoru zákmitů, které způsobily modifikaci výpočtu a tedy i úspěšný útok, jsou také obsaženy v Tabulce 5.4.

[5]Průměrný čas operace úspěšně modifikované zákmitem. Hodnota byla vypočtena z alespoň 10 měření s výjimkou použití `RSA_256_full` a `RSA_1024_full`, kdy bylo měření provedeno jednou

```
3%  55/2039 [01:01<39:52, 1.21s/it]

Glitched!

received_mgs = CWbytearray(b'86 0a ec 30 33 3c 91 c0 cc 3e 17 71 29 06 cb c5
c8 e3 da 21 f6 30 94 d5 f5 d1 97 e9 3a 91 1a cd')

correct_mgs  = CWbytearray(b'19 80 bf 37 e4 45 b4 1c 49 0a 49 cb 7b a3 65 8d
a9 7d 0e df 7e 29 c0 9d 6f 77 23 dc 75 d0 3e 0d')

clk_src     = clkgen
width       = 1.953125
width_fine  = 0
offset      = -12.109375
offset_fine = 0
trigger_src = ext_single
arm_timing  = after_scope
ext_offset  = 255
repeat      = 1
output      = clock_xor
```

Obrázek 5.5: Výpis o úspěšné modifikaci dešifrování pomocí RSA-CRT na mikrokontroléru STM32F3

5.2.2 Útoky pomocí zákmitů na zdroji napájení

Pro generování zákmitů na zdroji napájení musí být zvýšená hodinová frekvence pro STM32F3 na 24 MHz, a to kvůli omezeným schopnostem nastavení generátoru zákmitů. Zkrat je prováděn pomocí dvou MOSFET tranzistorů, které jsou součástí ChipWhisperer-Lite. Tranzistory jsou dostatečně předimenzovány, aby nedošlo k jejich zničení. Vzniklý zkrat musí být několik taktů zopakován, jinak nedojde k dostatečnému výkyvu napájení a chybám ve výpočtech. Pro zavádění zkratů na napájení byl zvolen koaxiální kabel s SMA-F konektory přímo ze sady ChipWhisperer s délkou 50 cm. Volba jiné délky kabelu by mohla způsobit nutnost hledat jinou konfiguraci generátoru zákmitů. Veškeré poznatky byly převzaty z [24]. Kabel je připojen k ChipWhisperer-Lite konektoru označeného `GLITCH` a druhá část je připojena k desce UFO s CW308T-STM32F3. Detailní zapojení je zdokumentováno na Obrázku 5.1.

Byl zvolen stejný postup útoků jako v podsekci 5.2.1. Pokud je útok prováděn na `bigi_128_ret` a `bigi_128_full`, úspěšné modifikování výpočtu se daří až při nastavení hodnot `width` a `offset` téměř na maximum. Také opakování zákmitu několik taktů za sebou je nutné, maximálně však lze použít `repeat` = 20. Při vyšších hodnotách `repeat` vzniklé zkraty již vyvolávají restart STM32F3 ve více než 95 % případů a úspěšné modifikace výpočtu již nelze docílit. Jednotlivé konfigurace parametrů jsou testovány desetkrát za sebou, jelikož vzniklé zákmity nejsou konzistentní (např. jedna konfigurace může modifikovat výpočet ve 3 z 10 pokusů). Výběr hodnot generátoru zákmitů, které úspěšně způsobily modifikaci výpočtu, jsou obsaženy v Tabulce 5.5.

Tabulka 5.5: Parametry pro útok pomocí zákmitu na zdroji napájení na mikrokontroléru STM32F3

Typ	`width`	`offset`	`ext_offset`	`repeat`	Čas[6]
`bigi_128_ret`	46	-35, -45	207, 266	7	22,7 ms
`bigi_128_full`	46	-35, -45	202, 268	7	22,6 ms
`RSA_256_ret`	46	-45	309, 343	7	22,7 ms
`RSA_256_full`	46	-45	342, 354	7	198,9 ms
`RSA_1024_ret`	46	-45	249, 341	10	46,8 ms
`RSA_1024_full`	46	-45	206, 236	10	4,35 s

Z průběhu testování parametrů generátoru bylo vypozorováno, že pokud je parametr `width` = 45, poté je výpočet modifikován, jestliže je hodnota `ext_offset` vyšší než 950 (interval `ext_offset` byl nastaven na $\langle 0, 1500 \rangle$). Při volbě `width` = 46 nastává úspěšná modifikace výpočtu již při hodnotě `ext_offset` vyšší než 200. V obou případech byl `offset` = -35.

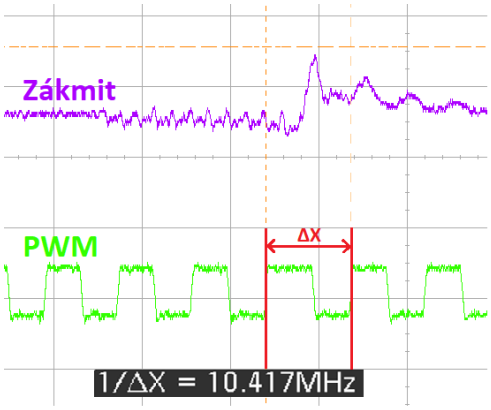
Po úspěšném zúžení parametrů generátoru zákmitů byl útok proveden na `RSA_256_ret` v rámci `BC_STM32F_RSA_VOLTAGE_GLITCHING` Notebooku. Útoky byly úspěšné, pokud byl parametr `offset` snížen na -45. To následně způsobilo úspěšný útok na `RSA_256_full`.

Jelikož se s parametry generátoru z útoku na `RSA_256_full` nedařilo prolomit i `RSA_1024_ret`, musela být zvýšena hodnota u `repeat` na 10. Pokud byla v případě `RSA_1024_full` dešifrovaná zpráva modifikována, tajný klíč K_T v některých případech dopočítat nešlo. Proto byla smyčka pro testování parametrů generátoru doplněna o funkci pro automatické získání tajného klíče `recover_key()` ihned po získání modifikované zprávy m'. Funkce obsahuje kód totožný s poslední částí Notebooku pro dopočet tajného klíče pro `RSA_1024_full`. Tabulka 5.5 opět obsahuje použité hodnoty parametrů, které způsobily úspěšné prolomení šifry.

5.3 Realizace útoků na CEC1702

Pro útok je využit modul CW308T-CEC1702 a zapojení je stejné jako to, které bylo popsáno v úvodu podsekce 5.2.2. CEC1702 má nastaven 48MHz zdroj hodinového signálu. Útoky na CEC1702 využívají pouze zákmitů na zdroji napájení, jelikož zařízení neumožňuje využití externího zdroje hodinového signálu. Flash paměť u CEC1702 též není možné programovat pomocí ChipWhisperer-Lite, proto je nutné použít postup popsaný v sekci 3.4 za využití USB-SPI převodníku *CH341A* a binárních souborů vytvořených kompilací v mikroC IDE.

[6]Jelikož se v Noteboocích používají pevné zpoždění v řádech 10–100 ms, nemůžou být časy ve stejném poměru jako hodinová frekvence v porovnání s útokem pomocí zákmitů na zdroji hodinového signálu



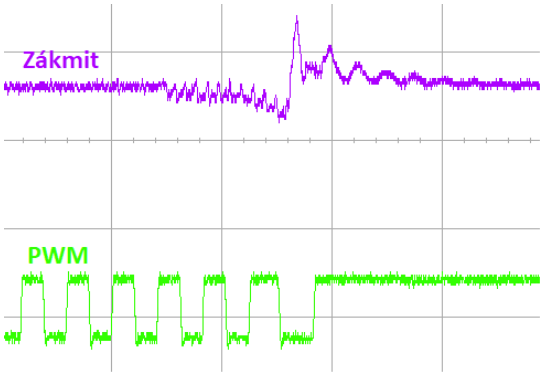
Obrázek 5.6: Snížení generované frekvence pomocí PWM při zákmitu na napájení na CEC1702

5.3.1 Útok na softwarovou implementaci RSA-CRT

Byl zvolen stejný postup jako u STM32F3. Nejprve byl proveden pokus o modifikaci modulární redukce použitím `bigi_128_ret` a následně `bigi_128_full`. Byl vytvořen Notebook `BC_CEC1702_BIGI_GLITCHING`, který se v několika částech liší oproti těm pro STM32F3. Programování zařízení již není v rámci Notebooku, ale je provedeno externími nástroji. Pro generování zákmitů je stále využito zdroje hodinového signálu z ChipWhisperer-Lite nastaveného na 48 MHz, aby hodinová frekvence odpovídala CEC1702.

CEC1702 generuje pomocí **PWM** (Pulzně Šířková Modulace) signál s frekvencí 12 MHz na pin `CLKFB` desky UFO, který lze přesměrovat pomocí přesunutí jumperu `J3` na spodní pozici – označena `HS1 / IN` – do ChipWhisperu-Lite. Tohoto se využívá zejména pro CPA útok, kdy je třeba mít frekvenci ChipWhispereru-Lite a CEC1702 co nejvíce synchronní. Pro generátor zákmitů bylo původně nastaveno totéž, kdy se přijímaná frekvence pomocí interní děličky vynásobila čtyřmi, aby odpovídala 48 MHz. Tento signál byl poté zvolen jako hodinová frekvence pro generování zákmitů. Autor si ale neuvědomil, že pokud se začne vytvářet na napájení zkrat, je ovlivněn i PWM modul (viz Obrázek 5.6 a 5.7). Chvilkové ovlivnění frekvence či neprosté vypnutí PWM způsobené zákmitem vyústilo v generování nedefinovaných stavů v ChipWhisperer-Lite (indikováno svícením dvou červených diod u micro USB portu). Následně musel být ChipWhisperer-Lite restartován a generátor zákmitů musel být nastaven znovu. Po překonfigurování na interní zdroj hodinového signálu se již problém s nedefinovaným stavem ChipWhisperer-Lite neobjevil.

Hodnoty pro nastavení generátoru byly převzaty z útoku na STM32F3, kdy bylo nutné zvýšit opakování zákmitu na dobu 11 taktů (`offset` = 11). Modifikace výpočtu se poté opět podařila a vybrané hodnoty generátoru zákmitů jsou v Tabulce 5.6.



Obrázek 5.7: Vypnutí PWM při zákmitu na napájení na CEC1702

Tabulka 5.6: Parametry pro útok pomocí zákmitu na zdroji napájení na SW implementaci RSA-CRT na CEC1702

Typ	`width`	`offset`	`ext_offset`	`repeat`	Čas
`bigi_128_ret`	46	-41	1378, 1606	11	26,2 ms
`bigi_128_full`	46	-41	331, 1466	11	27,4 ms
`RSA_256_ret`	46, 47	-41	742, 322	11	29,2 ms
`RSA_256_full`	46	-41	386, 442	11	386 ms
`RSA_1024_ret`	46	-41	252, 545	11	86 ms
`RSA_1024_full`	46	-41	1200	11	6.02 s

Pro realizaci útoku na varianty RSA-CRT byl po vzoru předchozích Notebooků vytvořen Notebook `BC_CEC1702_RSA_GLITCHING`. Opět ale neobsahuje část pro programování zařízení. Pro dopočet tajného klíče K_T z modifikované dešifrované zprávy m' je u 1024b RSA-CRT využito také funkce `recover_key()`, která je volána v rámci smyčky pro testování parametrů generátoru. Veškeré útoku byly úspěšné a oproti verzi na STM32F3 se pouze mírně změnily parametry generátoru. S největší pravděpodobností díky použití jiného kompilátoru, nejsou časy operací úměrné poměru hodinových frekvencí STM32F3 a CEC1702. I přes dvakrát vyšší hodinovou frekvenci CEC1702, je dešifrování v `RSA_1024_full` o cca 1,5 s pomalejší při využití stejné implementace.

5.3.2 Útok na implementaci RSA-CRT využívající HW akcelerátor

Pro útok byla využita verze firmware `RSA_1024_HW`, Notebook z útoku na čistě SW variantu – `BC_CEC1702_RSA_GLITCHING` – s částmi pro 1024b verzi RSA-CRT. Jelikož je dešifrování pomocí HW akcelerátoru spouštěno voláním funkce `pke_start(0)` a následně končí, jakmile podmínka `pke_busy() == 1` není pravdivá. Nelze se tedy již zaměřit pouze na určitou část dešifrování. HW akce-

Tabulka 5.7: Srovnání rychlostí dešifrování 1024b bloku dat na CEC1702 s využitím různých implementací RSA / RSA-CRT

Typ šifry	Implementace	Čas[7]
RSA-CRT	naivní, čistě SW	46,9 s
RSA-CRT	Montgomeryho doména, čistě SW	6,46 s
RSA	využití HW akcelerátoru	9,116 ms
RSA-CRT	využití HW akcelerátoru	3,228 ms

lerátor RSA používá 96MHz zdroj hodinového signálu, tudíž je šifrování mnohonásobně rychlejší než při využití čistě SW implementace (viz Tabulka 5.7). Z tohoto důvodu je možné s jedním nastavením parametrů otestovat vložení zákmitu v každém taktu dešifrování (celkem 154557 taktů).

Začalo se s testováním kombinace parametrů `repeat` = 11, `width` = 46, `offset` = $\langle -41, -40 \rangle$ (převzaty z předchozího útoku). Jelikož je snaha vyvolat modifikaci výpočtu pouze v první polovině procesu dešifrování (kdy by mělo být při výpočtech využito pouze jedno z prvočísel), bylo zvoleno rozpětí `ext_offset` = $\langle 100, 75000 \rangle$. Během zavádění zákmitů opět docházelo k restartování zařízení i k modifikaci výpočtu. Již při nastavené konfiguraci `repeat` = 11, `width` = 46, `offset` = -41 a `ext_offset` = 7710 provedla modifikace dešifrování vedoucí k úspěšnému získání d (viz Obrázek 5.8).

Jakmile se útok povedl, byla vytvořena verze *firmware* s novými hodnotami klíče i dat k šifrování a dešifrování (projekt `RSA-CRT_Glitching_New_Vals`). Nové hodnoty parametrů jsou uvedeny v Tabulce 5.8. Tento krok byl proveden pro ujištění, že je HW implementace RSA-CRT na CEC1702 vážně nezabezpečená proti útokům injekcí chyb na zdroj napájení. Byl vytvořen i další Notebook `BC_CEC1702_RSA_GLITCHING_2`, který obsahuje útok pouze na 1024b RSA-CRT za použití nových hodnot veřejné části klíče a šifrovaných dat. Útok byl spuštěn s upraveným intervalem `ext_offset` = $\langle 7500, 8000 \rangle$. Útok byl proveden pětkrát za sebou pro každou hodnotu `ext_offset`. Celkem bylo provedeno 2500 iterací útoku, z toho bylo provedeno 2482 standardních dešifrování, 15 restartů CEC1702, 15 modifikací dešifrování bez úspěšného získání tajného klíče. Tajný klíč se podařilo z modifikovaného dešifrování získat třikrát, a to při hodnotách `ext_offset` = 7948, 7969, 7974.

Otestován byl ještě interval `ext_offset` = $\langle 7500, 45000 \rangle$, kdy bylo z 37 500 útoků úspěšných 15 924 (42,5 %). Další testování parametrů nebylo prováděno, jelikož byly u HW implementace RSA-CRT prolomeny dva různé 1024b klíče. Je tedy prokázáno, že funkce `rsa_crt_decrypt()` z API pro HW RSA není nijak zabezpečena, a ani neprobíhá žádná verifikace dešifrování před vrácením dešifrovaných dat.

[7]Délka trvání měřena pomocí osciloskopu, snímky z měření jsou přiloženy na médiu ve složce *6_CEC1702utoky/oscillo_screenshots*

```
Glitched - W=46.09375, O=-41.015625, EO=7710
M'= 0x8c5dcaed9c0f2f69d74ca1980cbf1ea7876c6c333b2afee892b3d928a8d26de3c5c31aa95
cb11945db6d9ad92b847b5d1270ee4556c936f8fbd7335da79dcc41f6f4d7f1500e77097c26c708
b3ba01d03ea3291fa6dd7540018b648087ff3fa556f1085c172e5d030f2831657cd41a1a3b0de03
2a62110997051ff7bcab6afcd
----Acquired secret key parameters---
Q = 0xc000df51a7c77ae8d7c7370c1ff55b69e211c2b9e5db1ed0bf61d0d9899620f4910e41683
87e3c30aa1e00c339a795088452dd96a9a5ea5d9dca68da636032af
P = 0xc36d0eb7fcd285223cfb5aaba5bda3d82c01cad19ea484a87ea4377637e75500fcb2005c5
c7dd6ec4ac023cda285d796c3d9e75e1efc42488bb4f1d13ac30a57
D = 0x24bf6185468786fdd303083d25e64efc66ca472bc44d253102f8b4a9d3bfa75091386c007
7937fe33fa3252d28855837ae1b484a8a9a45f7ee8c0c634f99e8cddf79c5ce07ee72c7f1231421
98164234cabb724cf78b8173b9f880fc86322407af1fedfdde2beb674ca15f3e81a1521e071513a
1e85b5dfa031f21ecae91a34d
---- C^D mod N = M ?----
C^D mod N = 0x111444444444444444444444444444444444444444444444444444444444444
444444444444444444444444444444444444444444444444444123456111444444444444444
444444444444444444444444444444444444444444444444444444444444444444444444444
444444444444444444444444123456
        M = 0x111444444444444444444444444444444444444444444444444444444444444
444444444444444444444444444444444444444444444444444123456111444444444444444
444444444444444444444444444444444444444444444444444444444444444444444444444
444444444444444444444444123456

Glitching was successful! Both messages are the same
Average operation duration = 0.08808700000008685 s
```

Obrázek 5.8: Výpis při úspěšném útoku na HW implementaci RSA-CRT využívající 1024b klíč na CEC1702

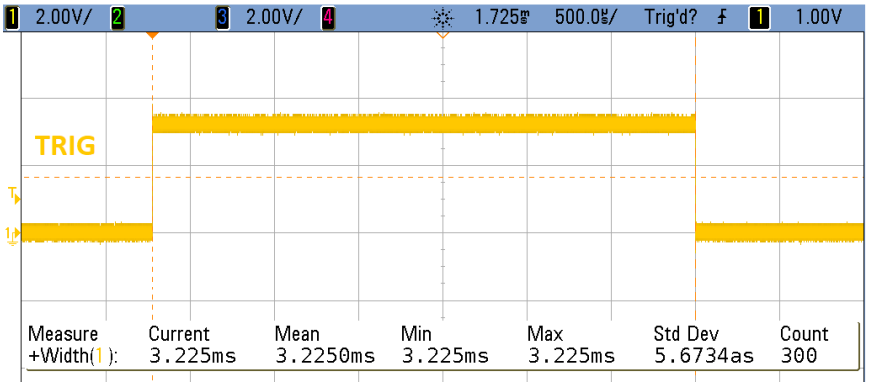
Po úspěšném útoku bylo ještě otestováno, zda by v budoucích pracích mohl být využit i časový útok (anglicky *Timing attack*) [68]. Byl proto využit firmware ze sekce 3.3. Po nastavení 1024b klíče bylo odesláno 300 náhodně vygenerovaných 1024b čísel k dešifrování pomocí HW RSA-CRT v rámci nově vytvořeného bloku v Notebooku `BC_CEC1702_FW_TESTING` označeného *Decryption testing*. Délka trvání dešifrování byla změřena pomocí osciloskopu. Na Obrázku 5.9 lze pozorovat, že odchylka je pouze 5,67 as (tedy 5,67 miliontiny biliontiny sekundy), jedná se tedy o zanedbatelnou hodnotu. Pro prolomení dešifrování by nebylo vhodné použití časového útoku a útok pomocí injekce chyb byla tedy správná volba.

5.4 Vyhodnocení útoků

Veškeré čistě SW implementace byly úspěšně prolomeny a tajný klíč K_T se podařilo získat na STM32F3 i na CEC1702. Variantu *firmware* využívající HW akcelerátor na CEC1702 se prolomit podařilo také. Použitá funkce `rsa_crt_decrypt()` z API k HW akcelerátoru RSA pro dešifrování není do-

Tabulka 5.8: Nové 1024b hodnoty použité pro ověření zranitelnosti HW implementace RSA-CRT v hexadecimálním zápise

p	C8C269B30E8A43A49EEB02C07E8ADFB6B98857530F6E174679884D7C59C7DDE9 A95070FFD570D71DB48FF86E41D27F61F29EDDC4C02AE5CFB25052C367F3EAA5
q	CA5445872192D7A962D15783274DEBF940C731F68EAE93C70EA54306D87D9CDB AD8FCA062409F33C2B1C5E077B32EB2A97D0AB21A1A12D8A9B3DDD09F2BF228F
d_p	72889F9EDCC3BAD5972882586999370A5B0CD2ACFB579685C95E1EBEBBC5CFEF B77DC209E2AE4EBB8EC0B83DC6E59D70B6E735A442F1F64C28F1B455F0DF7255
d_q	05C0658594DFBEBFB2EE6E856CED9B22A7C3C1D8D76F8105EC95A8151EBC5B64 2747EA1F4890FBD81F059C6AAC06E3867F70501D11A22A50415F7DD415917763
q_{inv}	37D955D22E6C3857394D5BAD78EA15BBCF5E72B7A1A22343BCC2C5B62D74C2DF ED8A6E31A4CE50B2CBA4AB9461B452E342F5DE625D52FAF2A7845458BB7DBBD9
e	10001
n	9EAB7DB8579116B680287DBB381EBD361767EEB2336A2B6FA03785918A816A29 9BB2CD43DEB3AD4EDFF4E6C4596D948936CD62D9D52EB247B56C2C0779B9797B 55BAC985266F4B9866314D547C46C201E71140BB15E573893D3702460AFCE8B1 931F63BF61548E45A7D395817ED84F6FDD5BA402BC48789CC760028E8184FC2B
d	1882CD6D08B9F514443FFD0C4AE314BFA265FFCDAF7B9B322741EC744B3D2AB5 3AC428496FC1B9E81158B7BD6543344AEE185448EA51C860A37A0E63293EA9B1 518C2033AC518DFC300090FBECE417C51819E4C8B68461D41A6757178E7AE794 32CA5EF42542D2AD3DC2324F12A60D8DC06590DA946FDB04A5E4EFF5BE9577A1
c	3B2288AF3257735F8124B95CD410FF8BEAFA8C3969DCF1DF8C92F6DB59356C46 21D6DC406DC17317657492A7E00D5D51EA81A4722DFE459B48315B8A7DC75A4C FFB9835E9FA2CEF6C903F2B294F486D5655DD35A407F8DD43DC1E3389FDCDF20 B850A794FC845B616113CA9B5F8DFF8DF22B4C4F90469476FE800DCFBBB111A1
m	13094FFD77443DD8EAC02FEF189C6BAFACB34EDFEAF44487199CE0C77640C074 94EEFB737D560ADEE3DA84F3E410CFE0AC13C6FFE3D76182ED677414BE832B14 5354477E9D6889243E902B979F48538524CC47FC356B9418173D8FA0DB6F10FA 04E65E33F0947368D8C393953577C38F57048A3981EE1C4CAD1236044D347509



Obrázek 5.9: Ověření času dešifrování na CEC1702 pomocí HW akcelerátoru kryptografických operací

statečně zabezpečená. Funkce nastaví hodnoty k dešifrování a celý proces dešifrování je obstarán HW akcelerátorem kryptografických operací. Byl vyvozen důsledek, že i samotná HW implementace RSA-CRT je nezabezpečená proti útoku injekcí chyb na zdroj napájení a následnému Bellcore útok pro dopočet tajného klíče.

Problémy vzniklé při útocích byly spjaty nejvíce s jazykem Python, kdy se s ním autor této práce seznamoval. S nynějšími znalostmi by Notebooky byly členěny a realizovány lépe, přehledněji a byly by více automatizované. V návazných pracích by se proto bylo vhodné zaměřit na úpravy Notebooků. Jelikož byly veškeré útoky injekcí chyb úspěšné (včetně útoku na HW implementaci RSA-CRT) bylo by vhodné zajistit zahrnutí útoků přímo do jedné z nových verzí prostředí ChipWhisperer (dosavadní útoky v rámci prostředí [69] rozšířit i na platformu CEC1702).

Varianta *firmware* `bigi_128_ret` oproti `bigi_128_full` neobsahuje téměř žádné urychlení, kdy kontrola výpočtu již v rámci implementace na mikrokontroléru / kryptoprocesoru je časově podobně náročná jako delší komunikace přes UART. Jelikož ale kontrola výpočtu proběhla vždy před odesláním výsledku, jednalo se o ujištění, že výpočet byl vážně modifikován a nenastala pouze chyba v přenosu dat přes UART.

Může se zdát, že ve verzích *firmware* mohly být větší rozdíly pro rychlejší otestování parametrů generátoru zákmitů a následného prolomení RSA-CRT, ale vždy při menších změnách byly ihned odladěny případné chyby ve vzniklých Noteboocích a redukovaném *firmware*.

Kapitola 6

Budoucí práce

V této kapitole jsou nastíněny další možnosti pro rozšíření jednotlivých implementací a taky útoků, na které by se dalo zaměřit v navazujících pracích.

6.1 Knihovna *bigi*

Výběr správné verze přípravku s CEC1702 a následné zprovoznění jeho programování zabralo řádově více času, než bylo plánováno. Po zprovoznění implementací a *firmware* již bylo třeba přejít na realizaci útoků postranními kanály. Nezbyl tedy čas na implementaci dalších užitečných operací a vyřešení veškerých `TODO`.

V navazujících pracích by bylo vhodné se tedy zaměřit na doplnění implementace o Barretovu modulární redukci [70]. Také by bylo vhodné vyřešit již zmíněné `TODO`. Jedná se zejména o dodatečné kontroly obsahů parametrů funkcí. Dále ve ve funkci `bigi_mod_mult_mont()` využít při výpočtu kruhový *buffer* místo bitových posunů nad `bigi_t` strukturou. Také by bylo vhodné otestovat funkčnost knihovny na zařízení s 16bitovými slovy. Žádné z `TODO` ale nebrání funkčnosti na CEC1702.

6.2 Implementace Paillierova kryptosystému

Ze stejného důvodu jako u *bigi* nebyla dořešena veškerá `TODO` v implementaci a bylo by vhodné je v navazující práci vyřešit. Jedná se zejména o nahrazení základní modulární redukce Barretovou modulární redukcí po implementování v *bigi*. Také by měly být vyřešeny veškeré kontroly parametrů funkcí. Opět žádné z `TODO` ale nebrání funkčnosti na CEC1702. Dále by mohla vzniknout třetí CRT varianta využívající funkci `rsa_crt_decrypt()` HW akcelerátoru kryptografických operací, na kterou by se poté mohl provést modifikovaný útok injekcí chyb. Zároveň by využití této funkce mohlo vést k ještě většímu urychlení průběhu šifrování a dešifrování.

6.3 Firmware pro CEC1702

V rámci *firmware* by mohly vzniknout funkce realizující podpis pomocí RSA-CRT a Pailliera spolu s verifikací podpisů. Dále by mohly být implementovány funkce pro získání aktuálně používaného klíče. Jelikož omezení velikosti přenášených zpráv mezi CEC1702 a okolím je nyní 64 B kvůli podpoře *SimpleSerial V1* protokolu, bylo by vhodné upravit implementaci pro podporu *SimpleSerial V2* [59]. Tato verze umožňuje délku dat do 192 B a to by umožnilo odesílání i 1024b čísel bez jejich rozděleni. Také rychlost by se zvýšila z 34 800 Baud na 230 400 Baud, což by umožnilo další urychlení komunikace s okolím. Dále by mohl být vzniklý *firmware* zahrnut do oficiálního ChipWhisperer repositáře GitHub, kde se nyní nachází pouze varianta obsahující pouze šifrování s použitím AES API z HW akcelerátoru [60].

6.4 Útoky postranními kanály

Útoky zatím cílily pouze na implementaci RSA-CRT. V budoucnu by se útoky na CEC1702 mohly rozšířit o útok injekcí chyb na implementaci Pailliera pomocí CRT a otestovat odolnost všech jeho implementovaných variant, včetně té využívající HW akcelerátor kryptografických operací. Dále by mohla být otestována bezpečnost funkcí `rsa_modular_exp()` a `rsa_encrypt()`. Použití těchto funkcí bylo diskutováno v podsekci 3.2.1.

Vytvořené IPython Notebooky, implementující jednotlivé útoky, by mohly být upraveny pro ještě větší automatizaci. V ideálním případě by po spuštění jednoho bloku byly provedeny automaticky veškeré kroky, až po testování zadaných intervalů parametrů generátoru zákmitů a následné vyhodnocení útoku. Po implementaci podepisovaní pomocí RSA-CRT, v rámci *firmware* pro CEC1702, by mohly být útoky přizpůsobeny i pro tento typ operace, aby odpovídaly již realizovaným útokům v rámci prostředí ChipWhisperer [63].

Závěr

Cílem práce bylo upravit implementaci Paillierova kryptosystému spolu s podpůrnou knihovnou pro práci s velkými čísly – *bigi* pro CEC1702. U varianty Pailliera, která využívá CRT pro urychlení výpočtu, byl využit hardwarový akcelerátor kryptografických operací. Pro splnění zadání bylo dále třeba prozkoumat, do jaké míry je již zmíněný kryptoprocesor odolný vůči útokům postranními kanály na šifrovací algoritmus RSA-CRT. Výsledky měly být porovnány s útokem na méně výkonném mikrokontroléru STM32F.

Prvním výsledkem práce je fungující a otestovaná implementace Paillierova kryptosystému a knihovny *bigi* na kryptoprocesoru CEC1702. Paillierův kryptosystém je implementován i ve verzi využívající HW akcelerátor kryptografických operací pro urychlení výpočtů obsahujících modulární mocnění. Většina problémů, která se při řešení objevila, byla spjata s vývojovým prostředím a kompilátorem firmy MikroElektronika. Nebylo ale nalezeno jiné prostředí obsahující API pro přístup k periferiím a HW akcelerátoru či kompilátor, který by toto API podporoval. Byl také vytvořen firmware, který podporuje komunikaci s CEC1702 přes UART. Dále podporuje dynamickou změnu délky klíče šifer, šifrování pomocí Pailliera a také obsahuje implementaci RSA-CRT ve dvou variantách podobně jako je tomu u Pailliera – využívající pouze *bigi* nebo HW akcelerátor kryptografických operací.

Při začátcích práce bylo třeba vybrat správný vývojový přípravek a zprovoznit nahrávání programu do CEC1702. Byl zvolen přípravek, který je zároveň i modulem do soupravy ChipWhisperer pro následné usnadnění realizace útoků postranními kanály – *CW308T CEC1702 NewAE target*. Programování zařízení bylo zprovozněno dvěma způsoby. První způsob využívá rozhraní JTAG a hardwarový programátor mikroProg od firmy MikroElektronika, který umožňuje ladění programu za běhu. Druhý způsob používá USB-SPI převodník *CH341A* a upravenou aplikaci pro programování Flash paměti zařízení. Tento způsob programování je nutný pro úspěšnou realizaci útoků injekcí chyb na napájení u CEC1702, kdy se zařízení během útoku často restartuje. Jelikož je program ihned po restartu nahrán automaticky z Flash

paměti, CEC1702 se nemusí pokaždé programovat manuálně znovu.

Útoky pomocí injekcí chyb byly realizovány pomocí soupravy ChipWhisperer. Vycházelo se z již realizovaného útoku pomocí zákmitu na zdroji hodinového signálu na mikrokontrolér STM32F3. Pro útok byla zvolena implementace RSA-CRT, která využívá *bigi*. Zákmity na zdroj hodinového signálu byly zaváděny, pokud byla při dešifrování prováděna modulární redukce. Útok byl úspěšný a podařilo se dopočítat tajný klíč u 256b i 1024b RSA-CRT s využitím modifikovaného Bellcore útoku. Za využití stejných postupů byl realizován útok pomocí zákmitů na zdroji napájení, kdy se podařilo opět úspěšně získat tajný klíč u 256b a 1024b RSA-CRT.

S nabytými znalostmi z útoku na STM32F3 byl proveden útok na CEC1702. Jelikož CEC1702 nepodporuje externí zdroj hodinového signálu, bylo možné využít pouze útoku injekcí chyb pomocí zákmitů na zdroji napájení. Při využití čistě SW implementace RSA-CRT byl útok úspěšný a došlo k prolomení 256b a 1024b klíče. Útok injekcí chyb pomocí zákmitů na zdroji napájení byl úspěšný i v případě implementace RSA-CRT využívající HW akcelerátor. V tomto případě došlo k prolomení 1024b klíče. HW implementace RSA-CRT na CEC1702 tedy neobsahuje žádné bezpečnostní opatření proti základním útokům injekcí chyb a nelze tedy HW akcelerátor označit za dostatečně zabezpečený.

V budoucnosti by bylo vhodné knihovnu *bigi* dál rozšířit o funkci Barrettova redukce. Útoky pomocí injekcí chyb by mohly být rozšířeny o útok na implementaci Paillierova kryptosystému. Tato bakalářská práce se totiž touto problematikou nezaobírá. Dále by bylo vhodné prozkoumat možnosti realizace jiných typů útoků a ověřit, zda i další části HW akcelerátoru kryptografických operací CEC1702 nejsou zabezpečené, jak je tomu u HW implementace RSA-CRT.

Literatura

[1] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, Springer, 1999, s. 223–238.

[2] Říha, J.; Klemsa, J.; Novotný, M.: Multiprecision ANSI C Library for Implementation of Cryptographic Algorithms on Microcontrollers. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, 2019, s. 1–4, doi:10.1109/MECO.2019.8760285.

[3] ChipWhisperer StarterKit SCAPACK-L1 [online], NewAE Technology Inc. [cit. 2022-02-25]. Dostupné z: `https://www.newae.com/products/NAE-SCAPACK-L1`

[4] Rivest, R. L.; Shamir, A.; Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, ročník 21, č. 2, feb 1978: str. 120–126, ISSN 0001-0782, doi:10.1145/359340.359342. Dostupné z: `https://doi.org/10.1145/359340.359342`

[5] Říha, J.: Implementation and Effectiveness Evaluation of the VeraGreg Scheme on a Low-Cost Microcontroller. Praha, 2019-06-07. Diplomová práce. České vysoké učení technické v Praze. Vypočetní a informační centrum.

[6] Dworkin, M.; Barker, E.; Nechvatal, J.; aj.: Advanced Encryption Standard (AES). 2001-11-26 2001, doi:https://doi.org/10.6028/NIST.FIPS.197.

[7] Paar, C.; Pelzl, J.: *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2010, ISBN 978-3-642-04100-6.

[8] Lórencz, R.; Kokeš, J.: 6. RSA, kryptografie s veřejným klíčem, DSA, El-Gamalův algoritmus [online]. Praha, České vysoké učení technické

v Praze, Fakulta informačních technologií, 2021. [cit. 2022-03-31], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/BI-BEZ/media/lectures/bez6.pdf

[9] Pei, D.; Salomaa, A.; Ding, C.: *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

[10] Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, ročník 31, č. 4, 1985: s. 469–472, doi:10.1109/TIT.1985.1057074.

[11] Fan, J.; Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012, https://ia.cr/2012/144.

[12] Brakerski, Z.; Gentry, C.; Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, New York, NY, USA: Association for Computing Machinery, 2012, ISBN 9781450311151, str. 309–325, doi:10.1145/2090236.2090262. Dostupné z: https://doi.org/10.1145/2090236.2090262

[13] Cheon, J. H.; Kim, A.; Kim, M.; aj.: Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, editace T. Takagi; T. Peyrin, Cham: Springer International Publishing, 2017, ISBN 978-3-319-70694-8, s. 409–437.

[14] Will, M. A.; Ko, R. K.: Chapter 5 - A guide to homomorphic encryption. In *The Cloud Security Ecosystem*, editace R. Ko; K.-K. R. Choo, Boston: Syngress, 2015, ISBN 978-0-12-801595-7, s. 101–127, doi:https://doi.org/10.1016/B978-0-12-801595-7.00005-7. Dostupné z: https://www.sciencedirect.com/science/article/pii/B9780128015957000057

[15] Klemsa, J.: [osobní sdělení přes platformu Microsoft Teams]. [2022-03-30].

[16] Klemsa, J.; Kencl, L.; Vaněk, T.: VeraGreg: A Framework for Verifiable Privacy-Preserving Data Aggregation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, s. 1820–1825, doi:10.1109/TrustCom/BigDataSE.2018.00275.

[17] Scholtzová, J.; Kleprlík, L.: Teorie čísel – vlastnosti prvočísel [online, slide 11 - 19]. [cit. 2022-03-31], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/BI-ZDM/lectures/ZDM-P11-numbt2-handout.pdf

[18] Novotný, M.: Implementation Attacks, slide 14–16 [online]. [cit. 2022-05-06], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/NI-BVS/media/lectures/attacks/NI-BVS-ImplementationAttacks.pdf

[19] Brier, E.; Clavier, C.; Olivier, F.: Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, editace M. Joye; J.-J. Quisquater, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN 978-3-540-28632-5, s. 16–29.

[20] Buček, J.: Útoky postranními kanály [online]. Praha, České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. [cit. 2022-05-06], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/BI-HWB/lectures/files/prednaska7.pdf

[21] Side channel attacks - State-of-the-art [online], Cryptography Research and Evaluation Committees. [cit. 2022-03-27]. Dostupné z: https://www.cryptrec.go.jp/exreport/cryptrec-ex-1047-2002.pdf

[22] Part 1, Topic 1: Introduction to Clock Glitching [online], NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://github.com/newaetech/chipwhisperer-jupyter/blob/e030e543eb2f1165672337bf8ff855e7ce23e8aa/courses/fault101/SOLN_Fault%201_1%20-%20Introduction%20to%20Clock%20Glitching.ipynb

[23] Novotný, M.: [osobní sdělení přes platformu Microsoft Teams]. [2022-04-16].

[24] Part 2, Topic 1: Introduction to Voltage Glitching [online], NewAE Technology Inc. [cit. 2022-04-16]. Dostupné z: https://github.com/newaetech/chipwhisperer-jupyter/blob/e030e543eb2f1165672337bf8ff855e7ce23e8aa/courses/fault101/SOLN_Fault%202_1%20-%20Introduction%20to%20Voltage%20Glitching.ipynb

[25] Boneh, D.; DeMillo, R. A.; Lipton, R. J.: On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97*, editace W. Fumy, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ISBN 978-3-540-69053-5, s. 37–51.

[26] Novotný, M.: Implementation Attacks, slide 30–38 [online]. Praha, České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. [cit. 2022-05-06], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/NI-BVS/media/lectures/attacks/NI-BVS-ImplementationAttacks.pdf

[27] CEC1702 Automotive Cryptographic Embedded Controller - Data Sheet [online], Microchip Technology Inc. [cit. 2022-02-25]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/CEC1702-Automotive-Data-Sheet-DS00003568B.pdf

[28] mikroC PRO for ARM Overview [online], MikroElektronika d.o.o. [cit. 2022-05-06]. Dostupné z: https://www.mikroe.com/mikroc-arm

[29] MPLAB X IDE Device Support List [online], Microchip Technology Inc. [cit. 2022-05-06]. Dostupné z: https://packs.download.microchip.com/DeviceDoc/Device_Support.pdf

[30] Clicker 2 for CEC1702 Datasheet [online], MikroElektronika d.o.o. [cit. 2022-05-06]. Dostupné z: https://download.mikroe.com/documents/starter-boards/clicker-2/cec1702/clicker-2-cec1702-manual-a.pdf

[31] NewAE Hardware Product Documentation - CW308T-CEC1702 [online], NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-CEC1702/

[32] NewAE Hardware Product Documentation - CW308 UFO [online], NewAE Technology Inc. [cit. 2022-03-27]. Dostupné z: https://rtfm.newae.com/Targets/CW308%20UFO/

[33] Clicker 2 for CEC1302 [online], MikroElektronika d.o.o. [cit. 2022-05-06]. Dostupné z: https://www.mikroe.com/clicker-2-cec1302

[34] mikroProg for CEC [online], MikroElektronika d.o.o. [cit. 2022-05-06]. Dostupné z: https://www.mikroe.com/mikroprog-cec

[35] CP2102 USB TTL převodník - dokumentace [online], ECLIPSERA s.r.o. [cit. 2022-05-06]. Dostupné z: https://dratek.cz/docs/produkty/0/747/eses1449940303.pdf

[36] CEC1702 Specifics [na přiloženém médiu], mikroElektronika ©2002-2019. [cit. 2022-05-06].

[37] ChipWhisperer Overview - What is ChipWhisperer? [online], ©2021, NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/getting-started.html

[38] NewAE Hardware Product Documentation - CW1173 ChipWhisperer-Lite [online], NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://rtfm.newae.com/Capture/ChipWhisperer-Lite/

[39] NewAE Hardware Product Documentation - CW308T-STM32F [online], NewAE Technology Inc. [cit. 2022-03-27]. Dostupné z: https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F/

[40] ChipWhisperer API [online], ©2021, NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/api.html#api

[41] ChipWhisperer Tutorials [online], ©2021, NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/tutorials.html

[42] ChipWhisperer Installation [online], ©2021, NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/installing.html

[43] Skrbek, M.: Vestavné systémy - Přednáška 13, slide 28–29[online]. Praha, České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. [cit. 2022-05-06], [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/BI-VES/files/bives_pred_13.pdf

[44] schéma CEC1702 Active Circuitry [online], 2018-04-05, © NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://rtfm.newae.com/Targets/UFO%20Targets/Images/CW308T-CEC1702-01_schematic_1.png

[45] Datasheet 20005262: SST26VF016B, 2.5V/3.0V 16-Mbit Serial Quad I/O™ (SQI™) Flash Memory [online], ©2014-2022 Microchip Technology Inc. and its subsidiaries. [cit. 2022-03-27]. Dostupné z: https://ww1.microchip.com/downloads/aemDocuments/documents/MPD/ProductDocuments/DataSheets/SST26VF016B-2.5V-3.0V-16-Mbit-Serial-Quad-IO-%28SQI%29-Flash-Memory-20005262G.pdf

[46] Serial Quad I/O (SQI) Flash Memory - SST26VF016 / SST26VF032 Data Sheet [online], ©2011 Silicon Storage Technology, Inc. [cit. 2022-05-06]. Dostupné z: http://ww1.microchip.com/downloads/en/devicedoc/25017a.pdf

[47] Aardvark I2C/SPI Host Adapter User Manual [online], ©2022 Total Phase, Inc. [cit. 2022-06-05]. Dostupné z: https://www.totalphase.com/support/articles/200468316

[48] EEPROM Flash BIOS USB programátor s čipem CH341A SPI [online], ©2014 - 2022 Neven. [cit. 2022-06-05]. Dostupné z: https://www.neven.cz/kategorie/elektronicke-soucastky/elektronicky-vyvoj/programatori-a-prislusenstvi-k-programovani/eeprom-flash-bios-usb-programator-s-cipem-ch341a-spi/

[49] Zhuo, S.: CH341A Programmer [online]. [cit. 2022-05-06]. Dostupné z: https://www.instructables.com/CH341A-Programmer/

[50] Zhuo, S.: CH341A Programmer [zdrojový kod přístupný online]. [cit. 2022-05-06]. Dostupné z: https://drive.google.com/file/d/1J9tumxOdq2S5d7htbNoL5gaxK37igyXn/view

[51] Montgomery, P. L.: Modular multiplication without trial division. 1985, doi:10.1090/s0025-5718-1985-0777282-x. Dostupné z: http://dx.doi.org/10.1090/S0025-5718-1985-0777282-X

[52] CEC1302 Low Power Crypto Embedded Controller - Data Sheet [online], Microchip Technology Inc. [cit. 2022-03-27]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/00002022B.pdf

[53] ANSI Technical Committee and ISO/IEC JTC 1 Working Group and others: Rationale for international standard, Programming languages. Technická zpráva, C. Technical Report 897, rev. 5.10, ANSI, ISO/IEC, April 2003, [strana 50, kapitola 6.4.2.1].

[54] MikroE fórum: Demo Limit [online], MikroElektronika d.o.o. [cit. 2022-05-06]. Dostupné z: https://forum.mikroe.com/viewtopic.php?t=8952

[55] MikroE fórum: Delay_ms() inaccuracy [online], MikroElektronika d.o.o. [cit. 2022-03-27]. Dostupné z: https://forum.mikroe.com/viewtopic.php?t=3376

[56] Single Static Assingment Optimization [online], MikroElektronika d.o.o. [cit. 2022-03-27]. Dostupné z: https://download.mikroe.com/documents/compilers/mikrobasic/arm/help/ssa_optimization.htm

[57] CEC/MEC family devices ROM API User's Guide [online], Microchip Technology Inc. [cit. 2022-03-27]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/50002157B.pdf

[58] CEC1302 crypto API user's guide [online], Microchip Technology Inc. [cit. 2022-03-27]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/50002504A.pdf

[59] ChipWhisperer Simpleserial Documentation - Simpleserial v1.1 [online], ©2021, NewAE Technology Inc. [cit. 2022-05-06]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/simpleserial.html

[60] Dewar, A.: Implementace firmware pro CEC1702 [zdrojový kód dostupný online]. [cit. 2022-05-06]. Dostupné z: https://github.com/newaetech/chipwhisperer/tree/develop/hardware/victims/firmware/CEC1702

[61] Datasheet 20005044: SST25VF016B Data Sheet - 16 Mbit SPI Serial Flash [online], ©2015 Microchip Technology Inc. [cit. 2022-03-27]. Dostupné z: https://ww1.microchip.com/downloads/aemDocuments/documents/MPD/ProductDocuments/DataSheets/SST26VF016B-

2.5V-3.0V-16-Mbit-Serial-Quad-IO-%28SQI%29-Flash-Memory-20005262G.pdf

[62] Agilent Technologies InfiniiVision 7000A Series Oscilloscopes Data Sheet, 2012-12-03 [online], Agilent Technologies, Inc. [cit. 2022-03-27]. Dostupné z: https://www.keysight.com/us/en/assets/7018-08233/data-sheets-archived/5989-7736.pdf

[63] Part 2, Topic 1: Fault Attack on RSA [online], NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://github.com/newaetech/chipwhisperer-jupyter/blob/e030e543eb2f1165672337bf8ff855e7ce23e8aa/courses/fault201/SOLN_Lab%202_1%20-%20Fault%20Attack%20on%20RSA.ipynb

[64] O'Flynn, C.: cwlitepro_glitch.png [online]. [cit. 2022-03-27]. Dostupné z: https://github.com/newaetech/chipwhisperer/blob/develop/docs/figures/cwlitepro_glitch.png

[65] Dewar, A.; O'Flynn, C.: Implementace simpleserial-glitch [zdrojový kód dostupný online]. [cit. 2022-05-06]. Dostupné z: https://github.com/newaetech/chipwhisperer/tree/develop/hardware/victims/firmware/simpleserial-glitch

[66] Part 1, Topic 2: Clock Glitching to Bypass Password [online], NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://github.com/newaetech/chipwhisperer-jupyter/blob/e030e543eb2f1165672337bf8ff855e7ce23e8aa/courses/fault101/SOLN_Fault%201_2%20-%20Clock%20Glitching%20to%20Bypass%20Password.ipynb

[67] Part 2, Topic 2: Voltage Glitching to Bypass Password [online], NewAE Technology Inc. [cit. 2022-06-05]. Dostupné z: https://github.com/newaetech/chipwhisperer-jupyter/blob/e030e543eb2f1165672337bf8ff855e7ce23e8aa/courses/fault101/SOLN_Fault%202_2%20-%20Voltage%20Glitching%20to%20Bypass%20Password.ipynb

[68] Kocher, P. C.: Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology| Crypto*, ročník 96, 1996, str. 104113.

[69] ChipWhisperer Tutorials [online], NewAE Technology Inc. [cit. 2022-03-27]. Dostupné z: https://chipwhisperer.readthedocs.io/en/latest/tutorials.html

[70] Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO' 86*, editace A. M. Odlyzko, Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, ISBN 978-3-540-47721-1, s. 311–323.

Příloha A

Seznam použitých zkratek

AES Advanced Encryption Standard

RSA Rivest, Shamir a Adleman – příjmení autorů a zároveň asymetrické šifry, již vymysleli

CRT Chinese Remainder Theorem, česky Čínská věta o zbytcích

IDE Integrated Development Environment, česky vývojové prostředí

NDA Non-Disclosure Agreement, česky dohoda o mlčenlivosti

API Application Programming Interface, česky rozhraní pro programování aplikací

UART Universal Asynchronous Receiver-Transmitter, doslova Univerzální asynchronní přijímač-vysílač

PWM Pulse Width Modulation, česky Pulzně šířková modulace

SSA Single Static Assignment

FHE Fully Homomorphic Encryption, česky Plně homomorfní šifrování

PHE Partially Homomorphic Encryption, česky Částečně homomorfní šifrování

SHE Somewhat Homomorphic Encryption

PKE Public Key Encryption, česky Kryptografie s veřejným klíčem

SPI Serial Peripheral Interface

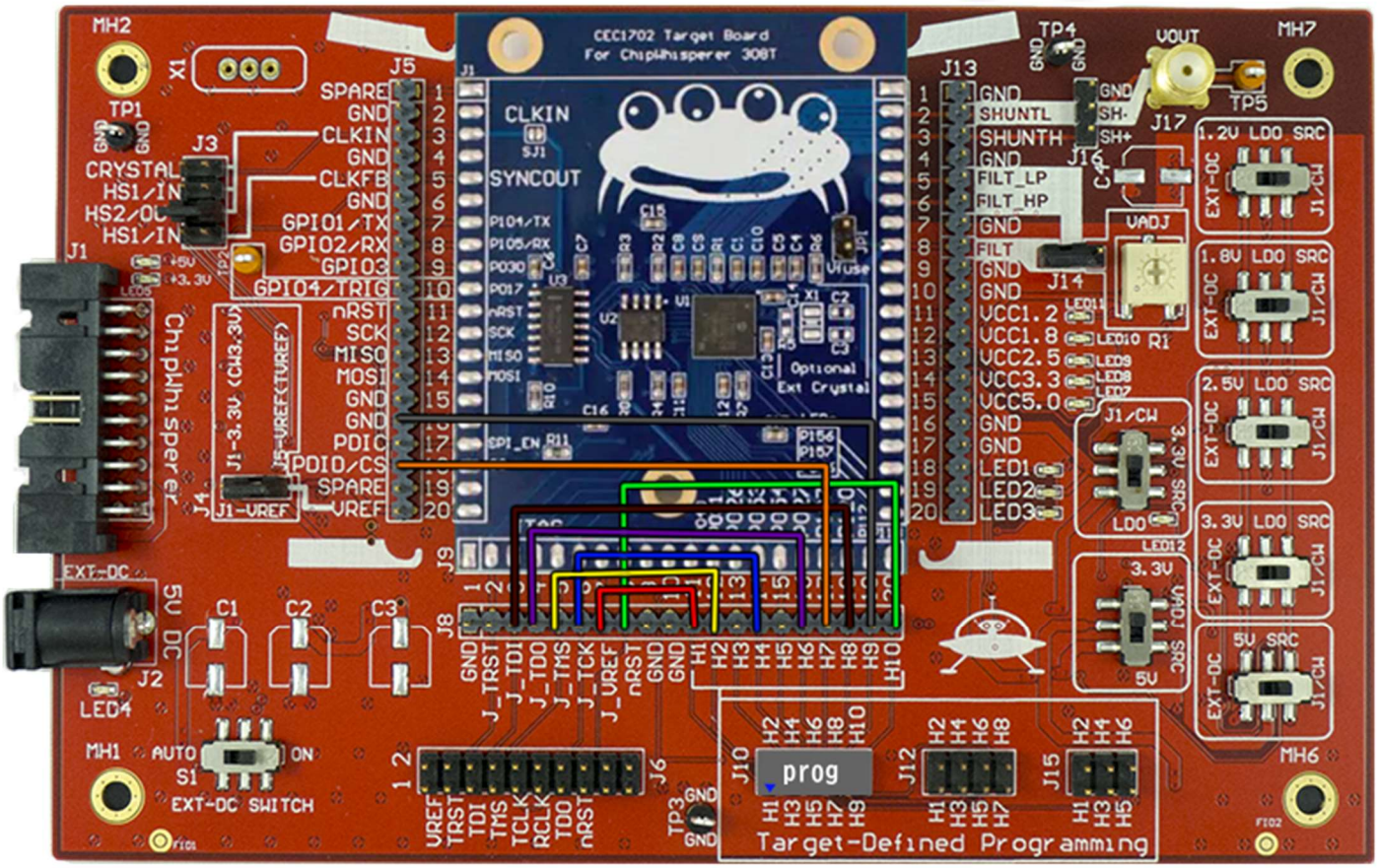
SQI Serial Quad I/O

JTAG Joint Test Action Group standard

Příloha B

Návod pro programování CEC1702

UFO CEC1702 zapojení



Zapojení HW programátoru mikroProg k UFO desce (Obrázek UFO desky převzat z https://rtfm.newae.com/Targets/Images/Cw308_top.png a Obrázek CEC1702 Target Board převzat z https://rtfm.newae.com/Targets/UFO%20Targets/Images/cec1702-picture.png)

Propojení podle schématu nahoře, kdy HW programátor *mikroProg for CEC* je připojen na J10 5x2 pinout (kdy je položen nad CW 308 UFO Board a pin H1 odpovídá pinu 1 programátoru, viz foto "Reálné zapojení")

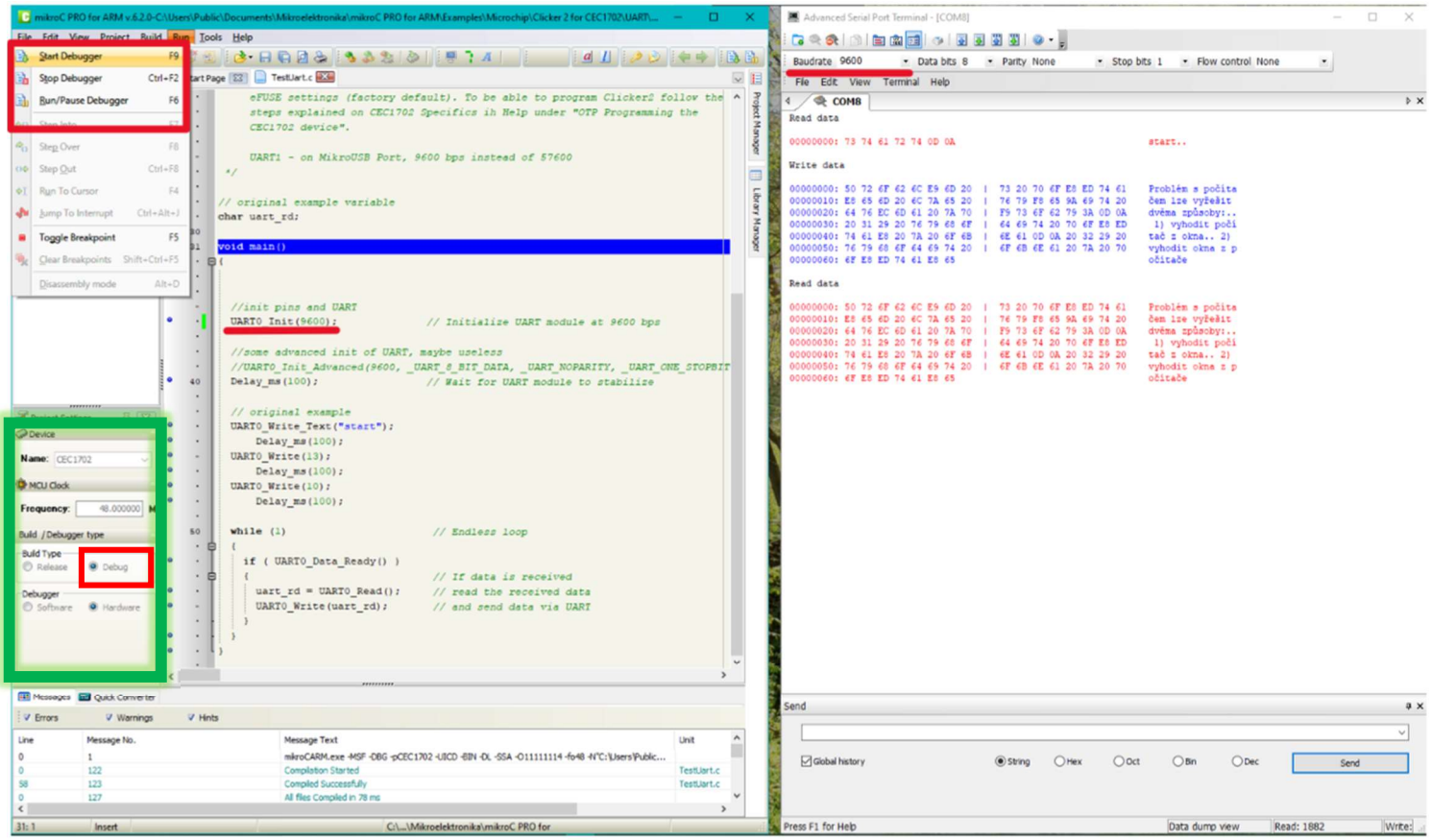
*nRST na CW 308 UFO Board odpovídá VCC_RST na Clicker2
*PDID na CW 308 UFO Board odpovídá SHD_CS0 na Clicker2
*J-TRST v Clicker2 připojen permanentně přes RC článek k napájecímu napětí 3,3V, na CW 308 UFO Board pin nepřiřazen, ale komunikaci s programmerem to nijak nebrání.

Pro spuštění UAR_ example je třeba:

1. Připojit USB-UART převodník na piny RX+TX+GND (odpovídající J7-8, J7-7, J7-6) na CW308 UFO Board
    - křížem, tj. RX-TX a TX-RX (křížem s převodníkem)
    - Použitý převodník: CP2102 STC
2. Spustit MikroC IDE a v něm otevřít UART example (příklad cesty v PC.: *C:\Users\Public\Documents\Mikroelektronika\mikroC PRO for ARM\Examples\Microchip\Clicker 2 for CEC1702\UART*)
    - Musí být nastaven UART0, jelikož UART1 nelze použít
    - V Project -> Edit Project... -> Data Type size by měla být velikost datového typu **int** 4 bytes (viz obr. "Úprava velikosti datového typu")
3. V mikroProg Suite je třeba detekovat zařízení (viz. obr. "mikroProg")
4. Program musí být spuštěn přes debug mode, (viz obr “komunikace” níže)
    - V podokně „Device“ je nutné mít zakliklé v „Build Type“ - Debug (viz zelený obdelníček)
    - Nejprve „Start Debugger“, poté klinutí na „Run Debugger“

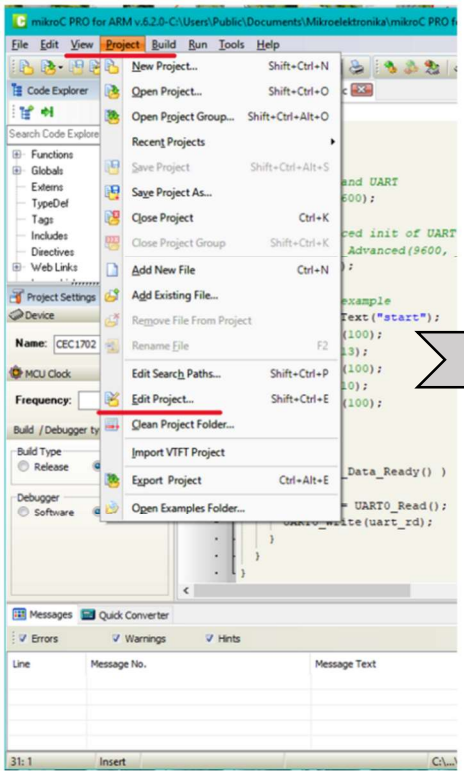
V terminálu pro komunikaci na sériové lince je následně vidět komunikace. (viz. obr "komunikace")

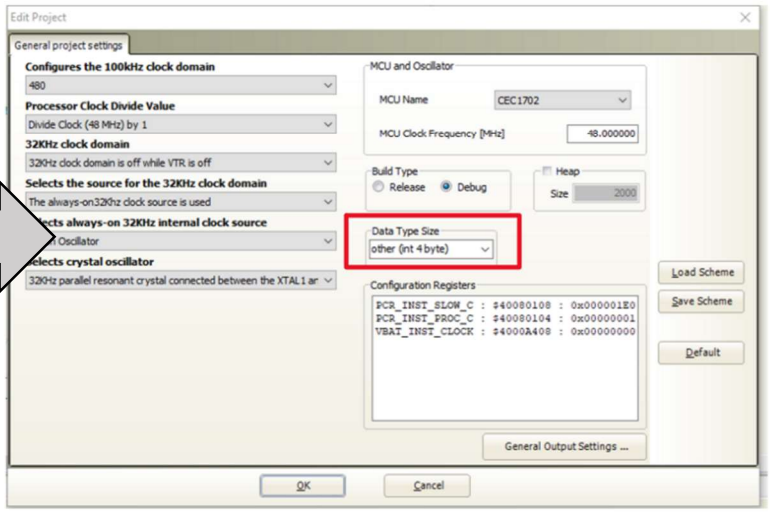
- Třeba nastavit správný baud-rate odpovídající tomu, co je nastaven v programu



komunikace

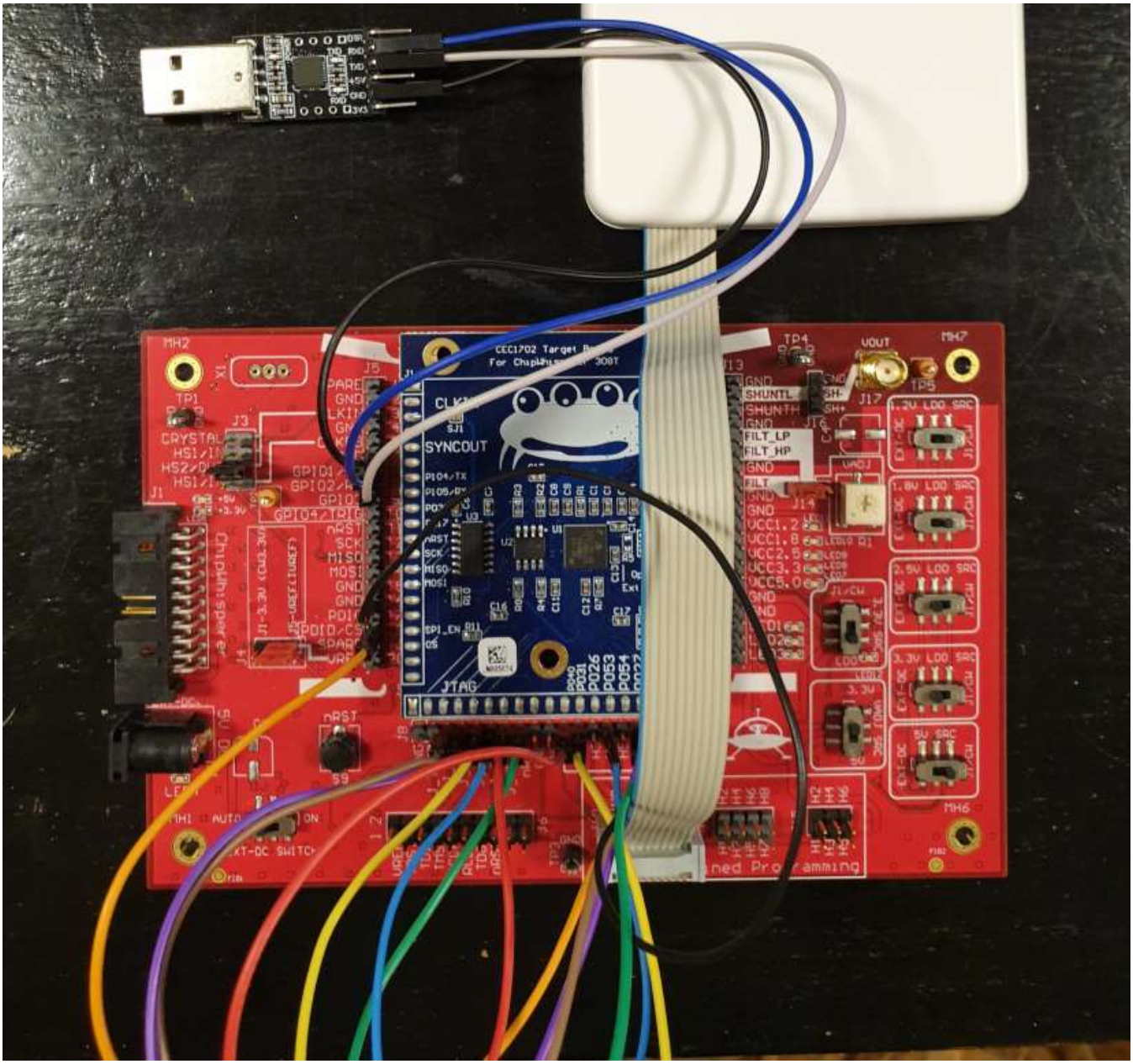
mikroProg





Úprava velikosti datového typu

Veškeré obrázky na této straně byly vytvořeny autory dokumentu (Lukáš Daněk a Tereza Horníčková)



1 - Reálné zpojení - foto pořízeno autory dokumentu

Příloha C

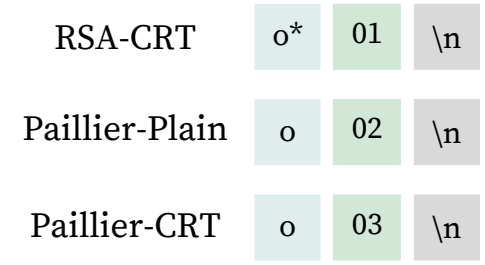
Dokumentace *firmware* pro CEC1702

Firmware CEC1702 umožňující šifrování pomocí RSA-CRT a Pailliera - přehled příkazů

VOLBA ŠIFRY

- příkaz musí být použit před nastavením délky šifry
- verze šifer [HW/SW] volit nelze, ale závisí na nahrané verzi firmware
- u všech šifer je vždy nastavená základní délka při použití příkazu pro volbu/změnu šifry
  - u čistě SW verze firmware je délka šifry nastavena vždy na 1024 bitů
  - u verze využívající krypto-HW je délka šifry nastavena vždy na 2048 bitů

Grafické znázornění struktury příkazu



*o jako opice

Příklad použití při sériové komunikaci

```
IN:   o01\n
OUT: z00

IN:   o04\n
OUT: z01
```

Příklad použití v ChipWhisperer API

```
RSA_cipher = bytearray.fromhex("01")
target.simpleserial_write('o', RSA_cipher)
target.simpleserial_wait_ack(timeout=0)
```

VOLBA DÉLKY ŠIFRY

- příkaz musí obsahovat vždy 4 číslice
- podporované bitové délky jsou:
  - 128, 256, 512, 1024, 2048 a 4096 bitů
- jednotlivé verze šifer mohou podporovat pouze podmnožinu bitových délek, viz. tabulka napravo

šifra/délka klíče	128	256	512	1024	2056	4096
RSA-CRT - HW verze	✗	✗	✗	✓	✓	✗
RSA-CRT - SW verze	✓	✓	✓	✓	✓	✓
Paillier-Plain	✓	✓	✓	✓	✓	✓
Paillier-CRT - HW ver.	✗	✗	✗	✗	✓	✓
Paillier-CRT - SW ver.	✓	✓	✓	✓	✓	✓

Grafické znázornění struktury příkazu

*l jako lanýž

Příklad použití při sériové komunikaci

```
IN:   l0512\n
OUT: z00\n
IN:   l1024\n
OUT: z00\n
IN:   l3192\n
OUT: z01\n
```

Příklad použití v ChipWhisperer API

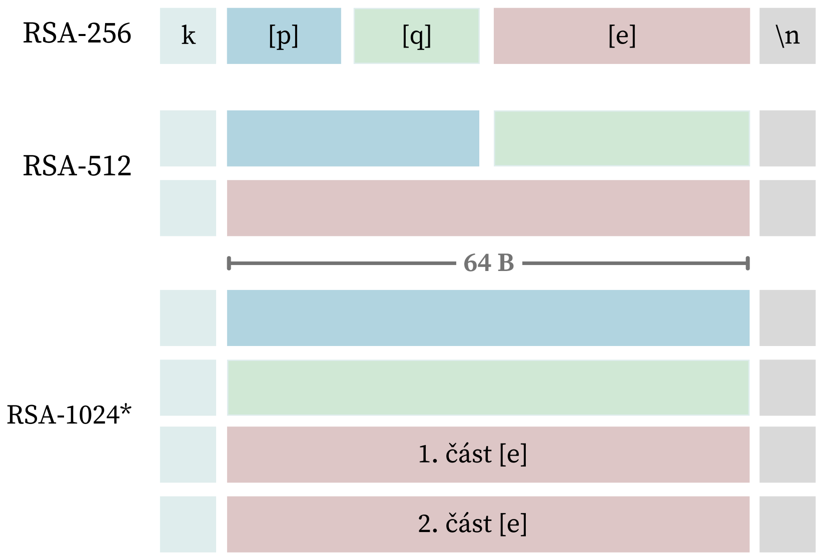
```
length = "0" + str(512)
lengt_bytes = bytearray.fromhex(length)
target.simpleserial_write('l', lengt_bytes)
target.simpleserial_wait_ack(timeout=0)
```

POUŽITÍ ŠIFRY RSA-CRT

NASTAVENÍ KLÍČE

- sekvence parametrů musí být vždy **p, q** následované **e**
    - **p, q** jsou prvočísla pro vytvoření modula n
    - **e** je veřejný exponent
- velikost **e** musí odpovídat nastavené bitové délce
- **p**, **q** musí být vždy přesně poloviční oproti nastavené bitové délce
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více po sobě jdoucích příkazů

Grafické znázornění struktury příkazu



* u RSA-2048 obdobné, jen je rozděleno [p] i [q] na dvě části

Příklad použití při sériové komunikaci (RSA-512)

IN(p,q):
kC11ABD937D8FD306166F62A8B7362CC4161D4AEC8569DB0CC9A8FDCD6CB32AC9D4CC057BEFFEC3C09FD18854BC684C17FF1F8725CD5D679C9932183FC41ED18F\n
OUT: z00\n

IN(e):
k00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100001\n
OUT: z00\n

Příklad použití v ChipWhisperer API (RSA-512)

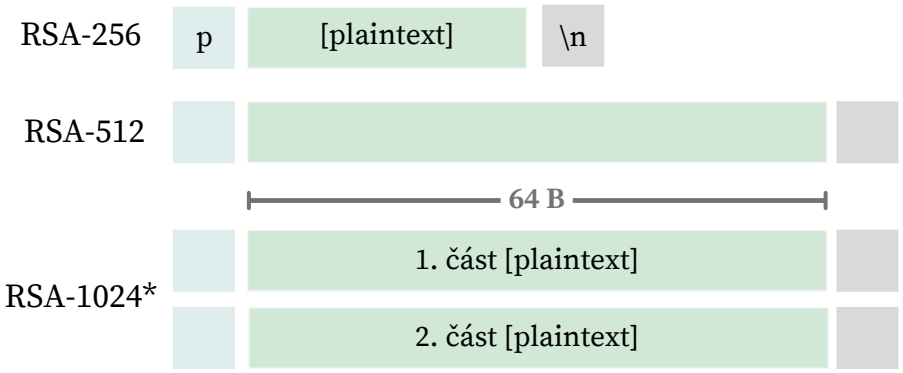
```
p = 'C11ABD937D8FD306166F62A8B7362CC4161D4AEC8569DB0CC9A8FDCD6CB32AC9'
q = 'D4CC057BEFFEC3C09FD18854BC684C17FF1F8725CD5D679C9932183FC41ED18F'
e = '00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100001'

p_q_bytes = bytearray.fromhex(p + q)
e_bytes   = bytearray.fromhex(e)

target.simpleserial_write('k', p_q_bytes)
target.simpleserial_wait_ack(timeout=0)

target.simpleserial_write('k', e)
target.simpleserial_wait_ack(timeout=0)
```

POUŽITÍ ŠIFRY RSA-CRT

ODESLÁNÍ DAT K ŠIFROVÁNÍ

- délka dat k zašifrování (plaintextu) musí vždy odpovídat nastavené bitové délce
  - pokud je nastavena délka klíče 1024, plaintext musí mít 1024 bitů - 128 B
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více po sobě jdoucích příkazů

Grafické znázornění struktury příkazu



* u RSA-2048 obdobné, jen je [plaintext] rozdělen na čtyři části

Příklad použití při sériové komunikaci (RSA-512)

IN(plaintext):
p978c1eb9198976b05a6a74d427ec1876b3276535301923ccbfbdd8f2f4acb4aed651e8037eac2d402ebd28d3d3ac442cc336169a6790643d22e6aaac7736a70f\n

OUT(ciphertext):
r465cf6b49884a2aeab9ded5c768aaf67a28caacda6d163623330f7cf9b456f41d3f9fea7d9043da0d2471cea4a3d442b8dde49192150d554e437ce995c280191\n
z00\n

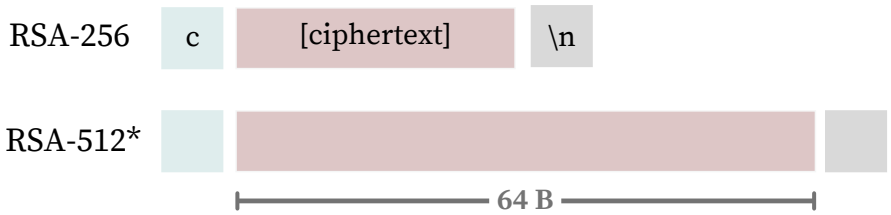
Příklad použití v ChipWhisperer API (RSA-512)

```
m = '978c1eb9198976b05a6a74d427ec1876b3276535301923ccbfbdd8f2f4acb4aed651e8037eac2d402ebd28d3d3ac442cc336169a6790643d22e6aaac7736a70f'
m_bytes = bytearray.fromhex(m)
target.simpleserial_write('p', m_bytes)
c_received = target.simpleserial_read('r', 64, timeout=0, ack=False)
```

ODESLÁNÍ DAT K DEŠIFROVÁNÍ

- délka dat k dešifrování (ciphertextu) musí vždy odpovídat nastavené bitové délce
  - pokud je nastavena délka klíče 1024, ciphertext musí mít 1024 bitů - 128 B
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více po sobě jdoucích příkazů

Grafické znázornění struktury příkazu



* u RSA-1024 a RSA-2048 stejné jako u šifrování pouze s jiným znakem příkazu - **c**

POUŽITÍ ŠIFRY RSA-CRT

ODESLÁNÍ DAT K DEŠIFROVÁNÍ

Příklad použití při sériové komunikaci (RSA-512)

IN(ciphertext):
c465cf6b49884a2aeab9ded5c768aaf67a28caacda6d163623330f7cf9b456f41d3f9fea7d9043da0d2471cea4a3d442b8dde49192150d554e437ce995c280191\n

OUT(plaintext):
r978c1eb9198976b05a6a74d427ec1876b3276535301923ccbfbdd8f2f4acb4aed651e8037eac2d402ebd28d3d3ac442cc336169a6790643d22e6aaac7736a70f\n
z00\n

Příklad použití v ChipWhisperer API (RSA-512)

```
c = '465cf6b49884a2aeab9ded5c768aaf67a28caacda6d163623330f7cf9b456f41d3f9fea7d9043da0d2471cea4a3d442b8dde49192150d554e437ce995c280191'
c_bytes = bytearray.fromhex(c)
target.simpleserial_write('c', c_bytes)
m_received = target.simpleserial_read('r', 64, timeout=0, ack=False)
```

Příklad použití v ChipWhisperer API (RSA-1024)

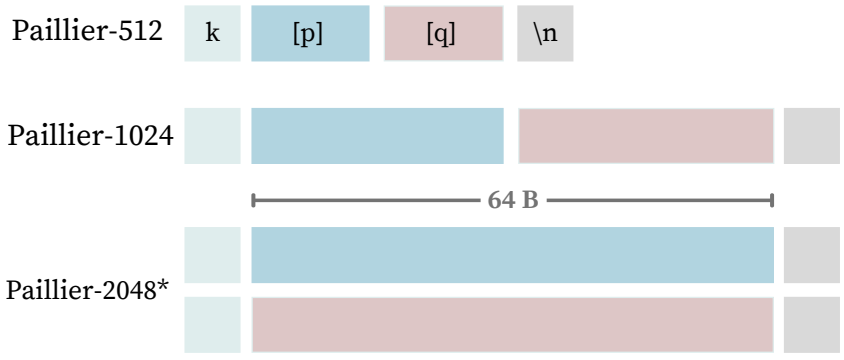
```
c = '6eee3987354e55d9d9dcb0b28a34b9f98dee2b551f64ebe14ec2eb310c74bf5e8b6e3cdd80426c6a5c6d8ffc469b9ded996e97f11ca20f1820882dba8ab0650b\
     990791da5b78ca1a1afdced4eac36b7965082dae2acb05af003a696a2c6fa898ac1476a0839cfff030d24d6e7795c695dbcc376f85f81b37e21477d5c88269b7'
c_bytes_1 = bytearray.fromhex(c[0:128]) #64 B == 128 hex chars
c_bytes_2 = bytearray.fromhex(c[128:256])

target.simpleserial_write('c', c_bytes_1)
target.simpleserial_wait_ack(timeout=0)
target.simpleserial_write('c', c_bytes_2)

m_received = target.simpleserial_read('r', 64, timeout=0, ack=False)
m_received = m_received + target.simpleserial_read('r', 64, timeout=0, ack=True)
```

POUŽITÍ ŠIFRY PAILLIER (NAIVNÍ I CRT)

NASTAVENÍ KLÍČE

- sekvence parametrů musí být vždy **p**, **q** - prvočísla pro vytvoření modula n
- **p**, **q** musí být vždy přesně čtyřikrát menší oproti nastavené bitové délce (Paillier-1024 a **p**, **q** dlouhé 256 bitů)
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více po sobě jdoucích příkazů

Grafické znázornění struktury příkazu



* u Paillier-4096 obdobné, jen je rozděleno [p] i [q] na dvě části

POUŽITÍ ŠIFRY PAILLIER (NAIVNÍ I CRT)

NASTAVENÍ KLÍČE

Příklad použití při sériové komunikaci (Paillier-512)

IN(p,q):
kC70DBF653E92AE474D92DCA059BD6FA7D1D8FD3CDE
CCB4F8C334DC0F240AA403\n
OUT: z00\n

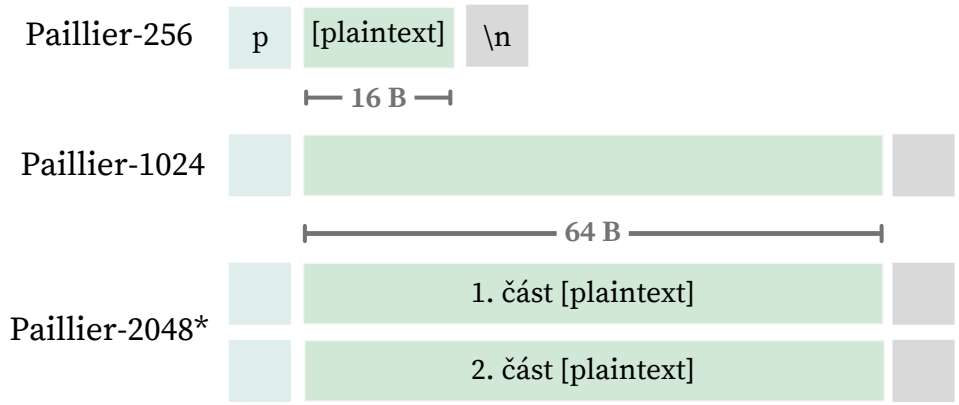
Příklad použití v ChipWhisperer API (Paillier-512)

```
p = 'C70DBF653E92AE474D92DCA059BD6FA7'
q = 'D1D8FD3CDECCB4F8C334DC0F240AA403'

p_q_bytes = bytearray.fromhex(p + q)

target.simpleserial_write('k', p_q_bytes)
target.simpleserial_wait_ack(timeout=0)
```

ODESLÁNÍ DAT K ŠIFROVÁNÍ

- data k zašifrování (plaintext) jsou poloviční délky než data zašifrovaná (ciphertext)
- plaintext musí být vždy přesně poloviční délky oproti nastavené bitové délce
  - například pro Paillier-1024 je plaintext dlouhý 512 bitů
- od délky plaintextu se odvíjí i celková délka příkazu
  - pro Paillier-256 je tedy délka příkazu 128 bitů - 16 B
  - pro Paillier-1024 je délka příkazu 512bitů - 64 B
  - pro Paillier-2048 je délka příkazu 512bitů - 64 B a plaintext je rozdělen do dvou příkazů
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více posobě jdoucích příkazů

Grafické znázornění struktury příkazu



* u Paillier-4096 obdobné, jen je [plaintext] rozdělen na čtyři části

Příklad použití při sériové komunikaci (Paillier-512)

IN(plaintext):
pd4cc057beffec3c09fd18854bc684c17ff1f8725cd5d679c9932183fc41ed18f\n
OUT(ciphertext):
r978c1eb9198976b05a6a74d427ec1876b3276535301923ccbfbdd8f2f4acb4ae\n
rd651e8037eac2d402ebd28d3d3ac442cc336169a6790643d22e6aaac7736a70f\n
z00\n

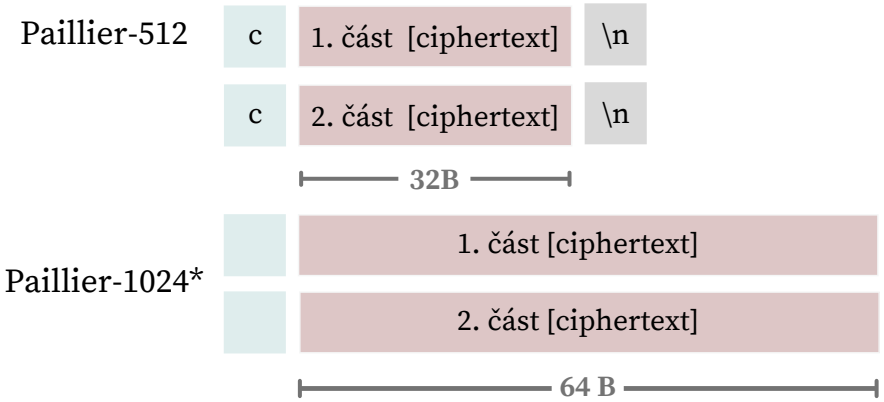
Příklad použití v ChipWhisperer API (Paillier-512)

```
m = 'D4CC057BEFFEC3C09FD18854BC684C17FF1F8725CD5D679C9932183FC41ED18F'
m_bytes = bytearray.fromhex(m)
target.simpleserial_write('p', m_bytes)
c_received = target.simpleserial_read('r', 64, timeout=0, ack=False)
c_received = c_received + target.simpleserial_read('r', 64, timeout=0, ack=True)
```

POUŽITÍ ŠIFRY PAILLIER (NAIVNÍ I CRT)
ODESLÁNÍ DAT K DEŠIFROVÁNÍ

- data k zašifrování (plaintext) jsou poloviční délky než data zašifrovaná (ciphertext)
- ciphertext musí být vždy přesně stejné délky jako je nastavená bitová délka
  - například pro Paillier-1024 je ciphertext dlouhý 1024 bitů - 128 B
- celková délka příkazu se odvíjí od délky plaintextu, tudíž je vždy ciphertext rozdělen na dva příkazy
  - pro Paillier-256 je tedy délka příkazu 128 bitů - 16 B a ciphertext se odešle ve dvou 16B příkazech
- pokud je celková délka příkazu větší než 64 B, musí se dělit na více posobě jdoucích příkazů
  - u Paillier-2048 je rozdělen cipher text na čtyři 64B příkazy

Grafické znázornění struktury příkazu



* u Paillier-2048/4096 obdobné, jen je [ciphertext] rozdělen na čtyři/ osm části

Příklad použití při sériové komunikaci (Paillier-256)

IN(ciphertext):
cD4CC057BEFFEC3C09FD18854BC684C17\n
cFF1F8725CD5D679C9932183FC41ED18F\n

OUT(plaintext):
rbfbdd8f2f4acb4aed651e8037eac2583\n
z00\n

Příklad použití v ChipWhisperer API (Paillier-512)

```
c = '6eee3987354e55d9d9dcb0b28a34b9f98dee2b551f64ebe14ec2eb310c74bf5e'
c_bytes_1 = bytearray.fromhex(c[0:32]) #16 B == 32 hex chars
c_bytes_2 = bytearray.fromhex(c[32:64])

target.simpleserial_write('c', c_bytes_1)
target.simpleserial_wait_ack(timeout=0)
target.simpleserial_write('c', c_bytes_2)

m_received = target.simpleserial_read('r', 64, timeout=0, ack=False)
```

ZÍSKÁNÍ DÉLKY TRVÁNÍ OPERACE

- příkaz neobsahuje žádná data, ale pouze znak reprezentující příkaz - **t**
- hodnota je vždy uváděna v ms
- příkaz vrací validní časový údaj po úspěšném provedení nastavení klíče, šifrování a dešifrování
- příkaz vrací 8 B reprezentujicí uint64_t, hodnota je little-endian
  - na první pozici je nejnižší bajt a na poslední pozici bajt nejvyšší
  - pokud operace trvala 8600 ms - 0x2198, příkaz vrátí pole bajtů (0x98, 0x21, 0x00, ..., 0x00)

Příklad použití při sériové komunikaci

(...úspěšné šifrování)
IN: t\n
OUT: r9821000000000000\n
z00\n

Příklad použití v ChipWhisperer API

```
target.simpleserial_write('t', bytearray([]))
#hodnota little-endian!
duration_in_bytes = target.simpleserial_read('r', 8, timeout=0)
duration_in_bytes.reverse()
duration_string = binascii.hexlify(duration_in_bytes)
duration_uint64 = int(duration_string, base=16)
print(f"{label} took {duration_uint64} ms")
```

Příloha D

Obsah přiloženého média

- `daneklu2_BP.pdf` ..... dokument obsahující bakalářskou práci
- `0_Analýza` ..... adresář se soubory zmíněnými v Analýze
- `1_bigi` ..... adresář s upravenou implementací knihovny *bigi*
  - `Array_Test` ..... projekt pro testování velikosti polí
  - `bigi` ..... aktuální implementace knihovny *bigi*
  - `bigi-16-12-2020` ..... původní implementace knihovny *bigi*
- `2_paillier` ..... adresář s upravenou implementací Pailliera
  - `paillier` ..... aktuální implementace Pailliera
  - `paillier-02-04-2021` ..... původní implementace Pailliera
  - `RSA_Encrypt_vs_ModExp` ... projekt testující funkce z API k HW RSA
- `3_CEC1702_firmware` ..... adresář s vytvořeným *firmware* pro CEC1702
  - `CEC1702_firmware` ..... *firmware* pro CEC1702
- `4_Flash_programmer` . adresář s upraveným programátorem Flash pamětí
  - `Application` ..... adresář se spustitelnou aplikací
  - `CH341A_Programmer` ..... implementace aplikace k programování Flash
  - `CH341A_Programmer_ORIGINAL` ..... původní implementace aplikace
- `5_STM32F3_utoky` ..... adresář se soubory k útoku na STM32F3
  - `BIGI_Glitching` ..... *firmware* pouze pro útok na *bigi*
  - `RSA-CRT_Glitching` ..... *firmware* pro útok na RSA-CRT
- `6_CEC1702_utoky` ..... adresář se soubory k útoku na CEC1702
  - `BIGI_Glitching` ..... *firmware* pouze pro útok na *bigi*
  - `binaries` ..... použité binární soubory k útokům
  - `oscillo_screenshots` ..... úplné snímky z osciloskopu
  - `RSA-CRT_Glitching` ..... *firmware* pro útok na RSA-CRT
  - `RSA-CRT_Glitching_New_Vals` ..... *firmware* pro útok na RSA-CRT