# RISC-V implementation of the NaCl-library

van den Berg, S.H.M.

*Award date:*
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

# RISC-V implementation of the NaCl-library

*Master Thesis*

Stefan van den Berg
0898187

Supervisors:
Prof. Dr. Tanja Lange
Dr. Ir. Roel Jordans
Dr. Boris Škorić

version 1.06

Eindhoven, May 2020

# Abstract

In this thesis two native implementations of the Network and Cryptography library (NaCl), created by Bernstein, Lange and Schwabe, in RISC-V are presented. The implementations are created to make it easier to secure IoT devices using the RISC-V Instruction Set Architecture (ISA). RISC-V is an open-source ISA with extensions for different functionalities enabling companies to produce processors with only the necessary instructions. To provide NaCl to all RISC-V processors one implementation uses only the 32 bit base instruction set. The other implementation additionally uses the multiplication extensions, which is found on most processors. The implementation without multiplication extensions requires 121 247 cycles for a 1024 byte xSalsa20 stream, 42 325 262 cycles for a Curve25519 scalar multiplication and 492 454 cycles for a 1024 byte Poly1305 computation. The implementation with multiplication extension requires 121 910, 5 389 988 and 38 530 cycles respectively. The implementations do not have secret-data-dependent branch conditions or loads. This means that no timing-attacks from branch and load instructions are depending directly or indirectly on secret data.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A concerning development in the embedded systems field is found in the security report of Symantec [28], where it is concluded that the number of Internet of Things (IoT) attacks are increasing. This is worrying given the fact that IoT devices often do not have sufficient security in place because they need to ensure a low costs of development. Security is not easy to implement, however there are some libraries that do make it more manageable.

These libraries do not provide security out of the box. They only provide functionality that can improve security. The functionality provided is typically cryptographic. The cryptographic funtions can be used to encrypt data preventing attackers from reading personal data. Next to that the cryptographic functions can be used to authenticate messages that are received preventing commands from unauthorized sources. Currenlty most data being sent is unencrypted and more than half of all IoT devices are vulnerable to medium- or high-severity attacks [1].

One of the libraries providing cryptographic functionality is the NaCl library [5] (pronounced as 'salt" library). This library has been designed to be easier to implement and use correctly, since other libraries can be difficult to configure. Many different versions exist of the NaCl library. One of them is $\mu$NaCl [12], which is focused on providing the NaCl library to the AVR microcontrollers to show that the NaCl library does not need large CPUs to function, which were the target in the original NaCl implementation and paper [5].

Knowing that the NaCl library can be run on small CPUs and the fact that the NaCl library is created with ease of use in mind it makes sense to provide it for multiple microcontrollers and embedded processors. That way it becomes less of a burden to the developers of IoT devices to create a secure device. A couple of years ago providing software to multiple microcontrollers or embedded processors would cost a lot of effort, since most processors had, and still have, a different set of registers and instructions. RISC-V wants to improve this by providing a completely open Instruction Set Architecture (ISA) [18]. All processors using the RISC-V ISA will have the same registers and instructions, which makes it possible to provide one implementation that targets multiple processors. A few RISC-V processors are already commercially available and that number is increasing [15].

This thesis presents two implementations of NaCl for the 32-bit RISC-V architecture, one using the base instruction set and one using the multiplication extensions. The implementation without the multiplication extension performs the Message Authentication Code (MAC) computation Poly1305 over a message of 1024 bytes in 492 454 cycles, the key-exchange function Curve25519 in 42 325 262 cycles and the stream-cipher xSalsa20 of a 1024 byte message in 121 247 cycles. The implementation with the multiplication extensions performs the operations in 38 530, 5 389 988 and 121 910 cycles respectively. Next to the two implementations this thesis also provides a test-framework to fuzz-test the implementations. Finally the SMArTCAT tool [16] ( more information in section 5.2), is updated and adapted to simulate the Sifive Hifive1 rev-B board [23] and check that no timing attacks are possible due to branches or loads depending on secret data.

# Chapter 2

# Background

In this chapter background information is given for the components and techniques used in this thesis. This consists of RISC-V, NaCl and multiplication techniques.

## 2.1 RISC-V

There are many different types of processors that can be developed using RISC-V. This is due to the extendable nature of the RISC-V standard. RISC-V consists of three base instruction sets (32-bit, 64-bit and 128-bit) that can be extended depending on the use-case. The base instruction set only provides basic instructions i.e. `add`, load, store, `xor` (for the full instruction set see [29]).

For other functionality an extension can be added. An extension can add registers and/or instructions. The extensions that are currently available and standardized are integer multiplication and division, atomic instructions, floating point instructions and compressed instructions.

The 32-bit base instruction set gives access to 32 registers each of 32 bits. 27 of these registers are general-purpose. The general-purpose registers are split up into three groups: temporary (7 registers), argument (8 registers) and saved registers (12 registers). The argument registers contain, as the name suggests, the arguments to a function. These registers can be overwritten by the function that is called. The same holds for the temporary registers. The saved registers must be set to the original value when returning from a function. That means that they can be used by a function, but a few extra cycles are needed to store and restore the original values to and from memory. The saved registers are the only registers which are guaranteed to keep their value when a function is called, all other registers can be overwritten.

Given that the implementations from this thesis are focused on IoT devices the 32-bit base instruction set is used. The first implementation will not use any extensions. For the implementation with intrinsics, the second implementation, the assumption is made that the integer multiplication and division extension is available. Having the multiplication extension is valuable for implementing the NaCl library, since many cryptographic functions depend on multiplications of large integers.

A development board that provides these functionalities and more is the Sifive Hifive1 rev-B board [23]. The development board has 32 Mbit SPI-flash memory and uses the FE310-G002 chip. This chip supports the multiplication and division, atomic and compressed extensions for the 32 bit base instruction set. To support the multiplication extension it uses specialized hardware. The execution time of almost all instructions is one cycle. For some instructions the result is not ready after execution, due to result latency. The specific instructions that have extra latency can be found in section 3.3 of [24]. In [24] it is also stated that the multiplier blocks until the previous operation completes resulting in a 5 cycle execution.

The pipeline of the processor is a 5-stage in-order pipeline. All instructions are fully bypassed, such that the result can be used by the next instruction. The pipeline will interlock if there is a read-after-write and write-after-write hazard, note that this can only occur when result-latency is

in place. The pipeline can also be flushed due to several reasons. The most common reason is a mispredicted branch or jump instruction, which incurs a three-cycle penalty.

The board has 16KB L1-instruction cache and 16KB Data SRAM scratchpad. The scratchpad is memory that is comparable to cache, except that it does not write back to memory automatically. Write backs have to be explicitly instructed.

## 2.2 NaCl

Before looking into the NaCl library basic cryptology concepts are explained. Cryptography can be split up into two categories: symmetric-key and public-key cryptography.

### 2.2.1 Symmetric-key cryptography

In symmetric-key cryptography a single key $k$ is used to perform different operations. The key $k$ is known to all parties with which communication is desired. The key can be used by the sender to encrypt a message $m$ to cipher text $c$. The receiver can decrypt $c$ using $k$ to obtain $m$. For the encryption and decryption a block- or stream-cipher is used. In NaCl a stream-cipher is used. A stream-cipher creates a pseudo-random bitstream which is xored with the message. The encryption prevents an attacker from reading $m$, since only the cipher text $c$ is visible to the attacker. The attacker can alter $c$ after which the receiver will obtain a different message after decryption.

A MAC uses a shared secret-key $k$ and either $m$ or $c$ to used to protect against this. If the MAC is based on $c$ and $k$ the receiver uses the same algorithm on $c$ and compares the MAC values to detect if an attacker changed the cipher text. If the MAC is based on $m$ and $k$ the cipher text $c$ is decrypted before calculating and comparing the MAC values.

In NaCl the stream-cipher and MAC used are Salsa20 and Poly1305 respectively. A more detailed explanation on Poly1305 and Salsa20 is given in sections 3.1 and 3.3.

### 2.2.2 Public-key cryptography

In public-key cryptography all parties have a key set consisting out of a public-key $pk$ and a secret-key $sk$. To encrypt a message the public-key of the receiver is used. An encrypted message can only be decrypted using the corresponding secret-key.

Public-key cryptography can also authenticate a user using a signature. A signature uses the secret-key of the signer. If a party wants to verify a signed message the public key of the signer is used.

The public-key cryptography is often used in combination with symmetric-key cryptography due to the slow speeds of public-key cryptography in comparison to symmetric-key. The Diffie-Hellman key exchange [9] is a public-key system in which the secret-key $sk$ of the send and the public-key $pk$ of the receiver are used to derive a symmetric-key $k$. This key can be computed by the receiving party using its secret-key and the publick-key of the sender, $k$ is thus a shared secret-key. The key $k$ is then used in either a block- or stream-cipher together with a MAC in the symmetric-key cryptography.

In NaCl the Diffie-Hellman key exchange uses the elliptic-curve Curve25519 [2]. A scalar multiplication using the secret-key as scalar and public-key as the point on the curve results in the shared secret-key. In section 3.2 a detailed explanation is given of Curve25519.

### 2.2.3 Origins of the NaCl library

The NaCl library [5] is a cryptographic library, created by Bernstein, Lange and Schwabe, containing several basic functions on top of which complex cryptographic protocols can be built. The functionality provided by the NaCl library is both public-key and symmetric-key cryptography.

The library was created due to the many exploited vulnerabilities that occurred while using other libraries. These vulnerabilities were typically not in the libraries themselves, but due to incorrect usage of the complex Application Programming Interfaces (APIs). To combat this the NaCl library has a simple interface for the provided functionality. For example if the user wants to use authenticated encryption using public-key cryptography only a couple of functions are used. Firstly the key pairs need to be made using the `crypto_box_keypair` function. Then the message $m$, nonce[1] $n$, public key of the receiver $pk_r$ and the secret key of the sender $sk_s$ are used in the function `crypto_box`, which returns the cipher text. To decrypt the the cipher text $c$ the function `crypto_box_open` is called with $c$, the nonce $n$, the public key of the sender $pk_s$ and the secret key of the receiver $sk_r$. The function will either return the message $m$ or it marks that it has failed. This is a lot simpler than other libraries where more input or additional function calls are used to configure the encryption.

### 2.2.4 Symmetric-key cryptography in NaCl

To send a message in symmetric-key cryptography the authenticated encryption function is used. The function creates a cipher text $c$ is from a message $m$ of size mlen, a nonce $n$ and a 64-byte one-time secret key, consisting of two 16-byte variables $k$ and $s$ and a 32 byte variable $t$. Firstly, using $n$ and $t$ in xSalsa20, a stream of length mlen is created, which is then xored with $m$ to create $M$. Poly1305 computes a message authenticator $s$ over $M$ using key $k$ ant $t$. The resulting ciphertext is a combination of both results giving $c = (s, M)$.

### 2.2.5 Public-key cryptography in NaCl

To send a message $m$ two keys are used, the public-key of the receiver $pk_r$ and the secret-key of the sender $sk_s$. The keys are used in a scalar multiplication in Curve25519 [2] to create a temporary key $t$ which is then expanded into $k$, $t$ and $s$ used in the symmetric-key functions.

To generate the public key $pk$ a scalar multiplication in Curve25519 is used with a fixed base point and the secret-key $sk$ as scalar. This base point is chosen, since it creates a cyclic subgroup with a large prime as the order. In the implementation provided by this thesis the scalar multiplication with the fixed base point is the same as any other scalar multiplication on Curve25519. Therefore the key generation is explained separately in the analysis.

### 2.2.6 Core security features

Not only does the NaCl library provide the above mentioned functions. It also provides certain security features with them. These core security features are to prevent vulnerabilities in the library.

#### No data flow from secrets to load addresses

The first core security feature is no data flow from secrets to load addresses, which means that it is not allowed to load data from memory depending on a secret. The issue with data flow from secrets to load addresses is that an attacker with access to the cache can create a timing attack to learn this data. A simplified example of a cache-attack is as follows. Let program $p$ have a data flow from secret $s$ to $s'$ where $s'$ is used to load some data. When data is loaded it can either be inside the cache or not. If it is in the cache the data is fetched faster than if it was not inside the cache. The attacker can control the cache as stated before. The attacker makes a guess what $s'$ could be and evicts the corresponding cache line. The program will load the data from location $s'$. The attacker can measure how long this operation takes to figure out if the guess was correct. This can be repeated multiple times to get the value or a part of $s'$, giving away secret information. Therefore $p$ is vulnerable to cache-timing attacks.

[1] Nonce: a number used only once; this can be a message counter

#### No data flow from secrets to branch conditions

The second core security feature is no data flow from secrets to branch conditions. This guarantee tackles a similar problem as the first one in that a timing attack could be possible. Let program $p$ have a data flow from secret $s$ to $s'$ where $s'$ is used as branch condition. The branch $b$ depending on $s'$ splits into two different data flows $d$ and $d'$. The execution time of $d$ and $d'$ can be different. The attacker can measure how long the actual execution takes and determine whether $d$ or $d'$ was executed. From that information the attacker learns a bit of information of $s'$. Leaking this information makes $p$ vulnerable. It is not necessary for $d$ and $d'$ to have different execution times for the attack to succeed, but this simplifies the exposition. Next to that most processors have a branch predictor which could lead to the same type of attack as a cache-attack even though manipulating the branch predictor is more difficult than the cache.

#### No padding oracles

The third core security feature is that there are no padding oracles. A padding oracle gives information on whether a forged message is correctly padded. This can either be by error message, or the response time of the oracle. The NaCl library defends itself against this by not decrypting any message for which the authentication fails. A message is therefore always first encrypted and then authenticated. Another layer of defense is that a forged message follows the same execution path through the verification as a valid message. A partially correct forged message will therefore take the same amount of time as a incorrect forged message. Given the attack no information on the correctness of their forge.

#### Centralized randomness

The fourth core security feature is that all randomization is centralized. To prevent unnecessary complexity in the NaCl library it uses the operating system to obtain the random numbers. This moves the responsibility of creating the cryptographic random-number generator from the NaCl library to the operating system.

#### Avoiding unnecessary randomness

The fifth core security feature is to avoid unnecessary randomness. The only functions that contain a function call for new randomness are the functions that generate key pairs. All the other functions are deterministic, making it easier to test and preventing issues with random number generation.

## 2.3 SMArTCAT

Symbolically Modeled Architecture Timing Channel Analysis Tool (SMArTCAT) [16] is a tool, created by Krak, determining if and how secret-parameters influence program execution time. It was developed and tested for binaries of the ARM Cortex-A7 processor. It checks a binary for three different types of timing channels. Each type is correlated to a policy designed to prevent timing attacks. The first and second policy are already mentioned in section 2.2.6 namely no data flow from secrets to branch conditions and no data flow from secrets to load addresses respectively. The third policy states that parameters, of an instruction, depending on secrets should not influence the execution time of an instruction. When the first policy is violated it is referred as a type one violation and likewise for other violations.

SMArTCAT mainly relies on the symbolic execution of angr [20, 21, 27]. Angr is a framework for binary analysis. It takes in a binary file and lifts it to an intermediate representation (IR) called VEX. The IR is used for symbolic execution which creates a large expression. This expression can be used to solve different types of problems. In this case the expression is extended with timing information. This information comes from a timing model specifying the execution time of instructions taking into account latencies, cache misses and more.

The large expression with timing information is used to provide a self-composition proof. In the self-composition proof a program $p$ and corresponding expression $e$ are provided. A copy of $e$ called $e'$ is made. The solver tries to find concrete values such that the timing information of $e$ and $e'$ are different. If such a value is found it does not yet mean that a violation is found. The value also has to either be a parameter flagged as secret or be derived from it. If that is the case a violation is found, and SMArTCAT reports it.

## 2.4 Multiplication techniques

In this section different multiplication techniques to multiply $A$ and $B$ are explained. To be able to explain how these techniques work the representation of a number must be explained.

Assume that numbers are in base $2^{26}$ (in sections 3.1 and 3.2 it is explained why this base is used). Therefore value $A$ is represented by limbs as follows $A = \sum_{i=0}^{n-1} A_i \cdot 2^{i \cdot 26}$, where $A_i \in S = \{ a \in \mathbb{N} \mid 0 \leq a < 2^{26} \}$ and $n = \lceil \frac{\log_2 A}{26} \rceil$. If the number of limbs is different between $A$ and $B$ the value is zero-padded to the same number of limbs.

### 2.4.1 Schoolbook

Let $A$ and $B$ be the multiplier and multiplicand respectively consisting out of $n$ limbs. Each limb of $A$ needs to be multiplied by each limb of $B$ in schoolbook multiplication. The products of the limbs $A_i \cdot B_j$ need to summed over all $A_k \cdot B_l$ where $k + \ell = i + j$ for $0 \leq i, j, k, \ell < n$. Another way of stating this is given in equation 2.1. An example of a 3 limb multiplication is shown in table 2.1 with each cell representing a limb and the columns below the line summed up.

$$A \cdot B = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_i \cdot B_j \cdot 2^{26 \cdot (i+j)} \tag{2.1}$$

| | | | | |
|---|---|---|---|---|
| $A_0$ | $A_1$ | $A_2$ | | |
| $B_0$ | $B_1$ | $B_2$ | | |
| $A_0 \cdot B_0$ | $A_0 \cdot B_1$ | $A_0 \cdot B_2$ | | |
| | $A_1 \cdot B_0$ | $A_1 \cdot B_1$ | $A_1 \cdot B_2$ | |
| | | $A_2 \cdot B_0$ | $A_2 \cdot B_1$ | $A_2 \cdot B_2$ |

Table 2.1: Example schoolbook multiplication

.

### 2.4.2 Karatsuba

Karatsuba [14] is a multiplication algorithm which reduces the number of limb multiplications. The multiplier and multiplicand are split into a high and low part. Let $A_\ell$ and $A_h$ represent the low and high part respectively and let $\ell$ and $h$ be the number of limbs of the respective parts. In this thesis it is taken that $\ell \geq h$.

Instead of multiplying each pair of limbs two multiplications of $\ell$ limb values and one multiplication of $h$ limb values are used namely $x = A_\ell \cdot B_\ell$, $y = A_h \cdot B_h$ and $m = (A_\ell + A_h) \cdot (B_\ell + B_h)$. Before calculating the result $m' = m - x - y$ is obtained. The result is now calculated as follows $x + m' \cdot 2^{26 \cdot \ell} + y \cdot 2^{52 \cdot \ell}$. Note that the multiplications by a constant of a power of two are replaced

by shifts. The computation is also given as equation in 2.2.

$$\begin{aligned} x &= A_\ell \cdot B_\ell \\ y &= A_h \cdot B_h \\ m &= (A_\ell + A_h) \cdot (B_\ell + B_h) \\ m' &= m - x - y \\ A \cdot B &= x + m' \cdot 2^{26 \cdot \ell} + y \cdot 2^{52 \cdot \ell} \end{aligned} \tag{2.2}$$

**Refined Karatsuba**

Refined Karatsuba [13] is an improvement on the original Karatsuba by saving one addition of $l$ limbs. Next to refined Karatsuba another improvement is made to reduce the maximum size of the intermediate results. For this improvement $m$ is changed and the way that the result is calculated. The improvement calculates $m$ using the absolute values giving $\hat{m} = |A_\ell - A_h| \cdot |B_\ell - B_h|$. The value $t$ is introduced to determine if $\hat{m}$ is equal to $m$. If $\hat{m} = m$ then $t = 0$ otherwise 1. Which is used in $m' = x + y - \hat{m} \cdot (-1)^t$. The rest of the calculation remains the same.

Refined Karatsuba computes the intermediate result $z = x_\ell + y$; this result is used twice, saving the addition of $l$ limbs. The computation of $m'$ changes to $m' = z + x_\ell - \hat{m} \cdot (-1)^t$. To obtain the result $x_\ell + m' \cdot 2^{26 \cdot \ell} + z \cdot 2^{52 \cdot \ell}$ is computed. In equation 2.3 the system of equations is given.

$$\begin{aligned} x &= A_\ell \cdot B_\ell \\ y &= A_h \cdot B_h \\ \hat{m} &= |A_\ell - A_h| \cdot |B_\ell - B_h| \\ t &= 0 \text{ if } \hat{m} = (A_\ell - A_h) \cdot (B_\ell - B_h) \text{ else } 1 \\ z &= x_h + y \\ m' &= z + x_\ell - \hat{m} \cdot (-1)^t \\ A \cdot B &= x_\ell + m' \cdot 2^{26 \cdot \ell} + z \cdot 2^{52 \cdot \ell} \end{aligned} \tag{2.3}$$

# Chapter 3

# Implementation

The NaCl library consists of multiple building blocks. The building blocks are Poly1305 (2.2.4), Curve25519 (2.2.5) and xSalsa20. In this chapter the design decisions made during the implementation of these building-blocks and other major decisions are explained.

## 3.1 Poly1305

In section 2.2.4 Poly1305 is mentioned to create the authenticator using $M$, the result of xoring the message $m$ with the stream cipher output and keys $k$ and $t$. The first operation of Poly1305 is splitting $M$ into 16 byte words. Each word is appended with a 1 resulting in a 17 byte word. If the last word is not 16 bytes long then the 1 is appended and the results is zero padded to a 17-byte word. These words represent little-endian integers. For each word a multiplication and addition is performed in $\mathbb{F}_{2^{130}-5}$. Let $w_i$ be the $i^{\text{th}}$ 16-byte word of message $M$ and $n = \lceil \frac{\text{mlen}}{16} \rceil$, where mlen is the size of $M$ in bytes. The result of a Poly1305 computation is given in equation 3.1.

$$r = ((\sum_{i=1}^{n} w_i \cdot k^i) \mod 2^{130} - 5) + t \mod 2^{128} \tag{3.1}$$

## 3.2 Curve25519

A scalar multiplication on Curve25519 [2] uses the elliptic curve described by $y^2 = x^3 + 486662 \cdot x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. The result of the scalar multiplication $n$ and $q$ is as follows. Let $Q$ be a point on the curve such that the X-coordinate of $Q$ equals $q$ modulo $2^{255} - 19$, if $Q$ is the point at infinity 0 is used, and let $S$ be the point $n \cdot Q$ on the curve. The result of Curve25519($n$,$q$) is the X-coordinate of $S$ or 0 if the point is infinity.

To compute the scalar multiplication a Montgomery Ladder [17] is used, which constist of 255 steps for Curve25519. Each step reads a bit of the scalar and performs one conditional swap at the beginning and at the end of the step. The step itself consists of those two conditional swaps, four additions, subtractions and squaring, five multiplications and one multiplication by 121665. The constant is derived from $\frac{486662-2}{4} = 121665$.

## 3.3 xSalsa20

xSalsa20 is a stream cipher which given a nonce $n$, key $k$ and a length $\ell$ outputs a stream of pseudo-random bytes of length $\ell$. xSalsa20 starts of by creating a subkey $k'$ from $k$ and $n$ using hSalsa20.

The subkey $k'$ and variable $nc$, the nonce $n$ appended with a counter $c$, are used in Salsa20 [3] to create a block $b$ consisting of 16 words. A word is 4 bytes in little-endian representation of a constant, the subkey $k'$ or the variable $nc$.

The block $b$ goes through a process of 20 rounds. Each round consists of 4 so called quarter-rounds. In a quarter-round a row or column of length 4 is transformed. During the transformation two cells are added and then rotated by either 7, 9, 13, or 18. During a rotation a byte is shifted to the left and any overflow wraps around. After the rotation that value is xored with the original cell value. After the rounds the original block $b$ is added to the corresponding cell.

The stream consists of the blocks from Salsa20 where the counter $c$ corresponds to the position of the block.

The hSalsa20 function is very similar to Salsa20. The difference between the two is the addition of block $b$ at the end, which hSalsa20 does not do, and only 8 words are stored instead of the entire block.

In the reference implementation of Salsa20 one thing stood out. In addition to the operations needed to run Salsa20 all limbs were loaded from big-endian to little endian and stored from little-endian to big-endian. In section 2.1 it is mentioned that RISC-V is a little-endian system. These operations are therefore not needed and were removed from the computation.

The operations performed in Salsa20 are on 32 bit values and no carries need to be taken into account. Therefore considering different bases is not necessary.

## 3.4 Core security features

The implementations created will need to adhere to the core security features to remain secure. There is however a problem with the creation of randomness. The NaCl library uses the underlying operating system to provide the randomness. A lot of embedded devices will not have access to an operating system that provides random numbers. The easiest way to solve this is to let the user provide a random number when keys are generated. However it is very difficult to generate random numbers, which do not introduce vulnerabilities, making it more difficult to use the implementations.

Another option would be to obtain keys from the SRAM [11]. The SRAM has some randomness when it has not been initialized yet. The randomness can be stored and used for the key generation. The main issue with this technique is that it depends on the type of memory available on the board. The implementations presented in this report are for RISC-V in general where the specifications of the board and even the processor can differ. Therefore it is not possible to use this technique in a general case and thus the user has to provide the random numbers.

## 3.5 Base of values during operations

In the reference implementation of the NaCl library values are stored in a little-endian representation of 32 bit limbs. Each limb contains eight bits of the value. In other words a 130 bit value is stored base $2^8$. A lot of space is wasted, since only a quarter of a register is used. Increasing the base results in more bits used per limb. Not only does this decrease the memory footprint, since less space is wasted, it also increases the speed of the computation. The number of operations will stay almost the same per limb and the number of limbs is decreased leading to a faster implementation. This would suggest that base $2^{32}$ would be the best solution, but it is not. There is a trade-off that needs to be considered.

This trade-off is that the more space of a limb is used the less space there is to delay handling carries. When all 32 bits are used, then addition and multiplication will not fit inside one limb. Another limb is needed to store the carries, which then need to be added as well to the result. The carry would create a ripple effect, since adding it to another limb can also have carries and so on. The handling of the extra limb causes a lot more operations per limb. The extra space per limb can be used to reduce the ripple effect or even cancel it out completely.

| Base | Limbs | #values to sum |
|---|---|---|
| $2^{26}$ | 5 | 64 |
| $2^{22}$ | 6 | 1024 |
| $2^{19}$ | 7 | 8192 |

(a) A value of 130 bits

| Base | Limbs | #values to sum |
|---|---|---|
| $2^{32}$ | 8 | 1 |
| $2^{29}$ | 9 | 8 |
| $2^{26}$ | 10 | 64 |
| $2^{24}$ | 11 | 256 |

(b) A value of 255 bits

Table 3.1: Trade off between the number of values that can be summed and the number of limbs needed to represent a value.

The extra space is not only used for addition, but also for multiplications by a small constant. This type of multiplication is needed when the value is reduced. During the reduction the overflow is multiplied by a small constant. In the case of Poly1305 the main reduction is modulo $2^{130} - 5$ and thus the constant is 5. For Curve25519 the main reduction is modulo $2^{255} - 19$ with constant 19.

In table 3.1 the trade-off is shown for different sizes of values. The base column indicates the lowest possible base to contain the value in the corresponding number of limbs. The '#values to sum' column indicates how many limbs of that base can be summed up without checking for carries. For Poly1305 table 3.1a is relevant. In table 3.1a it is clear that reducing the base below $2^{26}$ provides a lot of extra space which cannot be used effectively by the multiplication of a small constant and the additions. Therefore base $2^{26}$ is used to represent the values of Poly1305.

The trade off between bases for Curve25519 is shown in table 3.1b. In table 3.1b it is shown that base $2^{32}$ has not enough space left to sum two limbs. This is obviously not enough. Base $2^{29}$ is the next option with the lowest number of limbs. A summation of only 8 values is possible without checking for carries, which should be enough in most cases. However in base $2^{29}$ it is not possible to multiply by the constant 19 without carries. Therefore the base for Curve25519 is also chosen to be $2^{26}$, since it has enough space for multiplication by the constant, and enough space for additions.

# Chapter 4

# Theoretical analysis

In this chapter a theoretical minimum is determined on the cycle count of Poly1305, Curve25519 scalar multiplication and xSalsa20 in the NaCl library. The minimum cycle count is given for both implementation with and without intrinsics. This theoretical minimum is only for implementations using the same design-decisions as in chapter 3. All other functions are not analysed due to them depending on Poly1305 or Curve25519, or having a small impact on the cycle count in comparison to the analyzed functions.

In section 3.1 and 3.2 it is stated that base $2^{26}$ is used to represent a value. The value $A$ is represented by limbs as follows $A = \sum_{i=0}^{n-1} A_i \cdot 2^{i \cdot 26}$, where $A_i \in S = \{ a \in \mathbb{N} \mid 0 \leq a < 2^{26} \}$ and $n = \lceil \frac{\log_2 A}{26} \rceil$. The multiplications analysed in this chapter are performed in a field. Therefore if the result consists of a value larger than the field a reduction will be applied, giving that $n$ can not become larger than the number limbs in the field. Next to that the analysis is focuses on terms involving $n$ and small additive constants are neglected.

## 4.1 Multiplication techniques

As stated before multiplication is a key operation in these cryptographic functions. In section 2.4 two methods of multiplication are explained. In this section each method is analyzed for multiplicand $A$ and multiplier $B$, using the representation explained above, to obtain an instruction count.

### 4.1.1 Schoolbook multiplication

In section 2.4.1 schoolbook multiplication is explained. It is stated that each limb of $A$ is multiplied by each limb of $B$ and all results are summed up. A multiplication of two limbs gives a high and a low limb, each costing one multiplication. The high and low limb need to be summed with other limbs costing an addition instruction, resulting in $2 \cdot n^2$ multiplication and addition instructions.

The additions of limbs can also result in a carry, since the intrinsic multiplication instruction returns the result in base $2^{32}$. A carry is detected if the value before the addition is higher than after the addition. The set-less-than instruction is used for this. The carry then needs to be added to the more significant limb and rippled through. The cost of the ripple depends on the position of the limb. Rippling through for every addition of two limbs is too expensive and thus does not make sense in base $2^{26}$.

As stated in section 3.5 the carry does not need to be handled if base $2^{26}$ is used for the first 64 additions, converting from the output to base $2^26$ is also costly. Therefore an intermediate state is used in which a low part in base $2^{26}$, a middle part of 6 bits and a high part of 20 bits are created. Instead of summing all the parts up a summation per type is first performed. After

this the high part is shifted 6 to the left and added with the middle parts giving the final result in base $2^{26}$. This results in $3 \cdot n^2 + n$ addition, $n^2$ `and`and $n^2 + n$ shift instructions.

The final instruction count for schoolbook multiplication becomes $n^2 + n$ shift, $3 \cdot n^2 + n$ addition, $2 \cdot n^2$ multiplication and $n^2$ `and` instructions if the base needs to be changed otherwise it will take $2 \cdot n^2$ addition and multiplication instructions.

### 4.1.2 Refined Karatsuba

In section 2.4.2 refined Karatsuba is explained, for the convenience of the reader the refined Karatsuba equation, see 2.3, is repeated as equation 4.1.

$$\begin{aligned} x &= A_\ell \cdot B_\ell \\ y &= A_h \cdot B_h \\ \hat{m} &= |A_\ell - A_h| \cdot |B_\ell - B_h| \\ t &= 0 \text{ if } \hat{m} = (A_\ell - A_h) \cdot (B_\ell - B_h) \text{ else } 1 \\ z &= x_h + y \\ m' &= z + x_\ell - \hat{m} \cdot (-1)^t \\ A \cdot B &= x_\ell + m' \cdot 2^{26 \cdot \ell} + z \cdot 2^{52 \cdot \ell} \end{aligned} \tag{4.1}$$

Refined Karatsuba uses three multiplications of which two multiplications are values of $\ell$ limbs and one of $h$ limb values.

To subtract the two values of $h$ limbs, used to compute $\hat{m}$, $h$ subtractions are needed. For each subtraction a borrow can occur except for the most significant limb. To borrow a value one set-less-than instruction, which checks a register against the zero-register, and one subtraction is used. The final borrow ripples through $\ell$ limbs resulting in $2 \cdot \ell - 1$ subtraction instructions and $2 \cdot \ell - 1$ set-less-than instructions.

To calculate an absolute value the sign, which is the final borrow, of the subtraction is needed. The sign needs to be extended to fill 26 bits, the full space of a limb, and xor'ed with the subtraction result taking one shift and $\ell$ `xor` instructions. The original sign is then added to the least significant limb. The carry of this addition needs to be rippled through the entire value. The set-less-than instruction is used to check for a carry. The carry is added to the next limb and rippled through. The addition with rippling of the carry results in $\ell$ additions and set-less-than instructions. Two absolute values result in 2 shift, $2 \cdot \ell$ `xor`, $2 \cdot \ell$ additions and set-less-than instructions.

To calculate $z$ an addition of $h$ limbs is needed given that $x$ and $y$ are in base $2^{26}$.

To calculate $\hat{m} \cdot (-1^t)$ the signs of $A_\ell - A_h$ and $B_\ell - B_h$ are extended to 32 bits and xor'ed with each other. The resulting value is added and xor'ed to each limb of $\hat{m}$, resulting in 2 shift, $2 \cdot \ell$ addition and $2 \cdot \ell + 1$ `xor` instructions.

Given $\hat{m} \cdot (-1)^t$ computing $m'$ only takes one addition of $\ell$ limbs and one subtraction of $2 \cdot \ell$ limbs. The subtraction of two limbs can underflow resulting in a borrow from a more significant limb. To check if a value is negative the set-less-than instruction is used. For a borrow one addition is used to add one to the value being subtracted from the more significant limb. The base, $2^{26}$, is then added to the underflowed value costing another addition. For this a mask is used and an `and` instruction creating the mask costs a negation instruction. The only exception is the last limb, which will be need one addition less, since there is no subtraction of a higher limb value and thus the borrow can be done directly. This results in $5 \cdot \ell - 1$ addition, $2 \cdot \ell$ set-less-than instructions $4 \cdot \ell$ subtraction, $2 \cdot \ell$ negation and $2 \cdot \ell$ `and` instructions.

To obtain the result $m'$ and $z$ need to be shifted by a power of the base which does not cost any instructions, since a shift by the base corresponds to storing in a different output limb. The addition of $x_\ell$ also does not need to be considered, since no other values end up in those limbs. The addition of $m'$ and $z$ is an $\ell$ limb addition, since that is the amount of overlap between $m'$ and $z$ in the output, resulting in $\ell$ additions.

The total instruction count for refined Karatsuba is 2 $\ell$ limb multiplications, 1 $h$ limb multiplication, 4 shift, $4 \cdot \ell + 1$ `xor`, $2 \cdot \ell$ negation, $2 \cdot \ell$ `and`, $10 \cdot \ell + h$ addition, $6 \cdot \ell - 1$ set-less-than and $6 \cdot \ell - 1$ subtraction instruction.

| Instr. | $x$ | $y$ | $\hat{m}$ | $z$ | $m'$ | $A \cdot B$ | Total |
|---|---|---|---|---|---|---|---|
| add. | 0 | 0 | $2 \cdot \ell$ | $h$ | $2 \cdot \ell +$ $5 \cdot \ell - 1$ | $\ell + 1$ | $10 \cdot \ell + h$ |
| shift | 0 | 0 | 2 | 0 | 2 | 0 | 4 |
| `and` | 0 | 0 | 0 | 0 | $2 \cdot \ell$ | 0 | $2 \cdot \ell$ |
| subtr. | 0 | 0 | $4 \cdot \ell - 2$ | 0 | $4 \cdot \ell$ | 0 | $8 \cdot \ell - 2$ |
| slt | 0 | 0 | $4 \cdot \ell -$ $2 + 2 \cdot \ell$ | 0 | $2 \cdot \ell$ | 0 | $8 \cdot \ell - 2$ |
| `xor` | 0 | 0 | $2 \cdot \ell$ | 0 | $2 \cdot \ell + 1$ | 0 | $4 \cdot \ell + 1$ |
| neg. | 0 | 0 | 0 | 0 | $2 \cdot \ell$ | 0 | $2 \cdot \ell$ |
| `or` | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\ell$ limb mult | 1 | 0 | 1 | 0 | 0 | 0 | 2 |
| $h$ limb mult | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

Table 4.1: Instruction count for refined Karatsuba

### 4.1.3 26 bit multiplication without intrinsics

#### Add and shift

In the implementation without instrinsics a 26 bit multiplication has to be done manually. The most common method is the add and shift method, where the multiplicand is added to the result if the multiplier's last bit is a one. Then the multiplicand is shifted to the left and the multiplier is shifted to the right. It is not possible to use a branch to check if a bit is one or not, since that would introduce a timing attack. Therefore a mask needs to be created by negating the last bit of the multiplier. The multiplicand is `and`'ed with the mask and added to the result. In total this mask needs to be created 26 times, since the input is 26 bits.

When the multiplicand is shifted one to the left each time it will at some point overflow the element. Instead of shifting the multiplicand by one each time it is shifted $i - 1$ in iteration $i$. When the shift results in a value that could overflow two shifts, one `and` and one `or` instruction are used to distribute the shift over two limbs. Due to the extra limb an extra `and` instruction is needed to apply the mask. To add this to the result two addition instructions are necessary. There is no need to check for carries if $n \leq 64$ as stated in section 3.5. Finally the overflow is reduced to prevent the issues described in section 3.5. The reduction in overflow needs one `and`, shift and addition instruction each.

Finally it should be noted that in the first iteration the multiplicand does not need to be shifted reducing the shift counts by two and the `and` count by one. Resulting in a total count of 102 `and`, 26 negation, 51 addition, 76 shift and 27 `or` instructions to multiply two 26 bit numbers.

#### Karatsuba

The Karatsuba method can also be applied here in an attempt to reduce the instruction counts. First the 26 bit elements need to be split in two 13 bit elements. To do this an `and` instruction is needed to get the low part and a shift to get the high part. There are two inputs resulting in two shift and two `and` instructions. To obtain the $x$ and $y$ result each will need a 13 bit multiplication. Creating the absolute values of the subtractions in $m$ will take two subtractions, additions and negations. Another 13 bit multiplication is applied to obtain $m$. To create $t$ the signs of the subtractions are xor'ed.

To calculate the result using Karatsuba the following equation is used $r = x + 2^{13} \cdot (x + y - (-1)^t \cdot m) + 2^{26} \cdot y$. Due to $m$ fitting in one word $(-1)^t \cdot m$ can be calculated using only

one addition and one `xor`. After that one addition and one subtraction is needed to calculate the entire $x + y - (-1)^t \cdot m$. One shift and one `and` instruction is used to obtain the first 13 bits. Those 13 bits are shifted and added to $x$ for the 26 least significant bits of the result. An additional shift and addition instruction are need to obtain the remaining part of the result. The least significant element of the result is not guaranteed to be 26 bits. Therefore an extra reduction is needed adding another `and`, shift and addition instruction.

A 13 bit shift and add is very similar to the 26 bit shift and add, except that the result fits inside one element. The cost of a single iteration is two `and`, one shift of the multiplier, one shift of the multiplicand, one negation and one addition instruction(s). This is the cost for all iterations except the first one in which the addition and shift instructions are not needed. The total cost is 26 `and`, 24 shift, 13 negation, 12 addition instructions.

The total cost of a 26 bit multiplication using Karatsuba will be 80 `and`, 76 shift, 42 addition, 41 negation, 3 subtraction and 2 `xor` instructions. In section 2.1 it is mentioned that the execution time of all instructions is the same, except for the multiplier. Therefore the sum can be taken of the instruction counts to compare the speed of methods. The sums of the instructions for add and shift vs Karatsuba are 282 vs 244 respectively, thus Karatsuba is used for the 26 bit multiplication without intrinsics.

## 4.2 Poly1305

In section 2.2.4 it is stated that for each 16-byte word of the message $M$ a multiplication and addition in $\mathbb{F}_{2^{130}-5}$ is used. Note that due to the operations being in $\mathbb{F}_{2^{130}-5}$ a maximum number of $\lceil \frac{\log_2(2^{130}-5)}{26} \rceil = 5$ limbs is used.

A reduction is needed when the result of a multiplication consists of 6 or more limbs. To reduce the result all limbs representing values larger than $2^{130} - 5$ need to be reduced first which costs one addition and one multiplication by a constant $c$. The constant $c$ depends on the base and the number of limbs the modulus uses. For Poly1305 $c = 2^{26 \cdot 5} \equiv 5 \mod 2^{130} - 5$. The result of a 5 limb multiplication consists of 10 limbs of which 5 need to be reduced resulting in 5 addition instructions and 5 multiplications by 5. The multiplications by 5 are replaced by one shift and one addition instruction, which is cheaper than a multiplication.

Finally the overflow in each limb needs to be handled. To handle the overflow one right shift and one addition is needed for each limb, except the most significant limb. The most significant limb also needs an extra multiplication with $c$. A single round of reducing the value is not sufficient to guarantee all values to fit in 26 bits, one limb could be 27 bits. The extra bit does not result in any problems with the next addition operation, since sufficient space is left to add without causing carries. The addition can result in 28 bit values, which will be handled by the multiplication. The reduction with a single round of overflow handling results in 6 addition, 5 `and` and 6 shift instructions.

### 4.2.1 With intrinsics

Before starting the computation of Poly1305 the key $k$ must be loaded in base $2^{26}$ which takes 16 load instructions and 40 other instructions (see appendix A.1 for the operations).

Before analysing the cost of a step the method of multiplication is determined. In the tables of 4.2 the instruction counts for one multiplication in Poly1305 with intrinsics are given. The multiplication of 130 bits also contains the count for the reduction used before the multiplication. The total cycle count is the summation of all instructions needed with the multiplication instruction counting for 5 cycles due to the blocking, described in 2.1. In table 4.2b L and H indicate the size of the low and high part of the refined Karatsuba computation. Note that table entries can have an addition as data. The first part of the addition is the count of the fastest multiplication method for multiplication, either refined Karatsuba or schoolbook multiplication, of two or more limbs. The second part of the addition consists of the instructions needed to perform refined Karatsuba. The fastest multiplication method is determined using the same tables. For example

the 130 bit multiplication in refined Karatsuba needs two $3 \cdot 26 = 78$ bit multiplications and one 52 bit multiplication. In both cases the cycle count is lower for schoolbook multiplication. The counts of one 52 multiplication and two 78 bit multiplications are added up and result in the first operand of the addition operator.

From tables 4.2a and 4.2b it can be concluded that the best option is to use schoolbook multiplication for the 130 bit multiplication.

| Instr. | 130 bit mult. | 78 bit mult. | 52 bit mult. |
|---|---|---|---|
| add. | 81 | 19 | 8 |
| shift | 45 | 14 | 7 |
| `and` | 30 | 9 | 4 |
| subtr. | 0 | 0 | 0 |
| slt | 0 | 0 | 0 |
| `xor` | 0 | 0 | 0 |
| neg. | 0 | 0 | 0 |
| `or` | 0 | 0 | 0 |
| mult. | 50 | 18 | 8 |
| Total | 406 | 132 | 59 |

(a) Schoolbook multiplication

| Limbs Instr. | 130 bit L=3 H=2 mult. | 78 bit L=2 H=1 mult. | 52 bit L=1 H=1 mult. |
|---|---|---|---|
| add. | 46+51 | 16+20 | 5 |
| shift | 35+17 | 14+5 | 5 |
| `and` | 22+20 | 8+10 | 6 |
| subtr. | 12 | 8 | 4 |
| slt | 9 | 5 | 1 |
| `xor` | 13 | 8 | 6 |
| neg. | 11 | 7 | 3 |
| `or` | 0 | 1 | 3 |
| mult. | 44 | 18 | 6 |
| Total | 456 | 192 | 63 |

(b) Refined Karatsuba with the most efficient method for L and H limb multiplication

Table 4.2: Instruction and cycle count for multiplication including the neccessary reduction in Poly1305 with intrinsics

The calculation of a Poly1305 step does not only consist of a multiplication, an addition is also needed. The addition takes 5 addition instructions per step, since the base is already $2^{26}$ and thus no carries occur. Next to the addition the values must also be loaded for each step. The input is mlen bytes. Each byte must be loaded and changed to base $2^{26}$. Changing to base $2^{26}$ takes 35 instructions[1] (see appendix A.2 for the operations). Giving a count of $406 + 5 + 35 + 16 = 462$ cycles per step.

After all the steps are taken the result needs to be frozen, i.e. fully reduced modulo $2^{130} - 5$, the value of $t$, a part of the key, needs to be added and the result must be stored. Freezing the result ensures that the result represents all numbers in a unique way. For a freeze a copy is made of the result, which costs 5 instructions. Next a constant is added to the result and each limb is reduced to $2^{26}$ giving $r'$. For the addition 5 instructions are needed, to handle the reduction another 5 shift and addition instructions are used. A shift and negation are used to create a mask. The $r$ is `xor`'ed with $r'$ and `and`'ed with the mask taking 5 `xor` and `and` instructions giving $r''$. Finally $r''$ is `xor`'ed with $r'$ costing another 5 `xor` instructions. This results in 10 addition, 10 `xor`, 6 shift, 1 negation, and 5 `and` instructions.

Before adding $t$ a conversion from base $2^{26}$ to $2^8$ is needed, since $t$ is in base $2^8$ and the result needs to be in base $2^8$. This conversion takes 37 instructions (see appendix A.3 for the operations). Loading and adding $t$ takes 16 addition and load instructions. Storing the result costs another 16 store instructions, giving a total cycle count of $141 + \lceil \frac{\text{mlen}}{16} \rceil \cdot 462$.

### 4.2.2 Without intrinsics

The count of Poly1305 only changes in the step, since in no other place a multiplication takes place. In section 4.1.3 a count is given for multiplication without intrinsics. However in Poly1305 there is an addition before the multiplication takes place. If only a 26 bit multiplication is possible then the value needs to be reduced first. A 28 bit multiplication only uses 12 extra instructions,

[1] This is less than loading the key $k$, since $k$ needs certain bits to be 0 using extra and instructions

in comparison to section 4.1.3, which is less than a reduction. In the tables 4.3 and 4.4 the counts for schoolbook multiplication and refined Karatsuba without intrinsics are given respectively. The multiplication by $c$ is replaced by a shift and an addition instruction. The addition operator in the table entries is used as in 4.2b. The lowest count for a multiplication without intrinsics in Poly1305 is 4615 giving a total step count of $4615 + 5 + 35 + 16 = 4671$. A complete Poly1305 operation will take $141 + \lceil \frac{\text{mlen}}{16} \rceil \cdot 4671$.

| | 130 bit | | 78 bit | | 52 bit | |
|---|---|---|---|---|---|---|
| Instr. | mult. | 28 bit mult. | mult. | 28 bit mult. | mult. | 28 bit mult. |
| add. | 66 | 2150 | 9 | 774 | 4 | 344 |
| shift | 11 | 3900 | 0 | 1404 | 0 | 624 |
| `and` | 5 | 4100 | 0 | 1476 | 0 | 656 |
| subtr. | 0 | 150 | 0 | 54 | 0 | 24 |
| slt | 0 | 0 | 0 | 0 | 0 | 0 |
| `xor` | 0 | 100 | 0 | 36 | 0 | 16 |
| neg. | 0 | 2100 | 0 | 756 | 0 | 336 |
| `or` | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 12582 | | 4509 | | 2004 | |

Table 4.3: Schoolbook multiplication instruction counts for Poly1305 without intrinsics

| Limbs | 130 bit L=3 H=2 | | 78 bit L=2 H=1 | | 52 bit L=1 H=1 | |
|---|---|---|---|---|---|---|
| Instr. | mult. | 27 bit mult. | mult. | 27 bit mult. | mult. | 27 bit mult. |
| add. | 786+51 | 0 | 263+20 | 43 | 5 | 129 |
| shift | 1344+17 | 0 | 472+4 | 78 | 2 | 234 |
| `and` | 1427+20 | 0 | 498+9 | 82 | 3 | 246 |
| subtr. | 87+12 | 0 | 26+8 | 3 | 4 | 9 |
| slt | 15+9 | 0 | 2+5 | 0 | 1 | 0 |
| `xor` | 80+13 | 0 | 24+8 | 2 | 6 | 6 |
| neg. | 743+11 | 0 | 258+7 | 42 | 3 | 126 |
| `or` | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 4615 | | 1854 | | 774 | |

Table 4.4: Refined Karatsuba instruction count for Poly1305 without intrinsics where the most efficient method is chosen for L and H limb multiplication

## 4.3 Curve25519

In section 2.2.5 it is stated that a scalar multiplication consists of 255 steps of two conditional swaps, four additions, subtractions and squarings, five multiplications and one multiplication by 121665. The squarings are replaced by multiplications to reduce the binary size. All of these operations are performed in $\mathbb{F}_{2^{255}-19}$ resulting in a maximum of $\lceil \frac{\log_2(2^{255}-19)}{26} \rceil = 10$ limbs.

As explained in section 4.2 a reduction is needed. In this case a value of 11 or more limbs need to be reduced, which takes one multiplication by constant $c$ and one addition per limb representing values larger than $2^{255} - 19$. For Curve25519 the constant $c = 2^{26 \cdot 10} \equiv 608 \mod 2^{255} - 19$. The multiplication of a value in base $2^{26}$ by 608 does not fit inside one limb. Therefore two multiplications are needed for the high and low part of the limb multiplication result. Resulting in 20 addition instructions and 10 multiplication by 608 operations.

To handle the overflow in each limb one right shift and one addition is used for each limb, except the most significant limb. The most significant limb needs to multiply $32 - 26 = 6$ bits by 608, which fits in one limb, adding one multiplication by 608 to the count, resulting in a total of 30 addition, 10 shift instructions and 11 multiplications by 608.

### 4.3.1 With intrinsics

Before starting the steps the 32 byte keys, $pk_r$ and $sk_s$, must first be loaded in base $2^{26}$. This takes $2 \cdot 32$ load instructions and $2 \cdot 72$ other operations (see appendix A.4 for the operations).

Before analyzing the instruction count of a step the multiplication method is chosen. The tables 4.5 and 4.6 are set up the same way as the previous tables, 4.2. A multiplication of a base $2^{26}$ number by 608 does not fit into one register resulting in 2 multiplication instructions per multiplication by 608. Next to that after a multiplication the base needs to be changed adding one shift, `or` and `and` instruction per multiplication by 608. The only exception is the multiplication of the overflow from the most significant limb. The overflow is at most 6 bits which does fit into one limb when multiplied by 608 thus only one multiplication instruction is needed and no base change.

From the tables, 4.5 and 4.6, it can be concluded that the best option is to use refined Karatsuba for 255 bit multiplication with schoolbook multiplication for the 130 bit multiplication used in the refined Karatsuba. For the multiplication by 121665 each limb will need two multiplications, because the result does not fit into one limb, and two addition instructions. The most significant bit also needs a multiplication by 608 and an extra addition for the high limb of the multiplication. These multiplications also need a base change costing 20 shift, 10 `or` and 10 `and` instructions. Resulting in a total of 21 multiplication and addition, 20 shift, 10 `or` and 10 `and` instructions.

| Instr. | 255 bit mult. | 130 bit mult. | 78 bit mult. | 52 bit mult. |
|---|---|---|---|---|
| add. | 411 | 65 | 19 | 8 |
| shift | 139 | 34 | 14 | 7 |
| `and` | 120 | 25 | 9 | 4 |
| subtr. | 0 | 0 | 0 | 0 |
| slt | 0 | 0 | 0 | 0 |
| `xor` | 0 | 0 | 0 | 0 |
| neg. | 0 | 0 | 0 | 0 |
| `or` | 10 | 0 | 0 | 0 |
| mult. | 221 | 50 | 18 | 8 |
| Total | 1785 | 374 | 132 | 59 |

Table 4.5: Schoolbook multiplication instruction counts for Curve25519 with intrinsics

Next the instruction count of the conditional swap is calculated. A conditional swap returns two 20 limb values, $r_1$ and $r_2$, and takes in two 20 limb values, $x_1$ and $x_2$, and a condition bit $b$. If the condition bit $b$ is 1 then $r_1 = x_2$ and $r_2 = x_1$ when $b$ is zero $r_1 = x_1$ and $r_2 = x_2$. For this operation one subtraction is needed to create a mask $m$ from $b$, since $0 - b$ wraps around to all ones if $b = 1$. Each limb of $x_1$ is `xor`'ed with the corresponding limb from $x_2$ giving the value $t$. The mask is applied to all limbs of $t$ giving $t'$. Eventually $r_1 = t' \oplus x_2$ and $r_2 = t' \oplus x_1$. This process is also given as an equation in 4.2. This results in in $3 \cdot 20$ `xor`, 20 `and` and one subtraction instruction. The two conditional swaps are not used in practive only in theory. In practice the number of conditional swap operations can be lowered to only one per step, except for the last step. Normally at the beginning and end of a step the conditional swap is applied. The conditional swap at the end can be combined with the conditional swap of the next step by combining the condition bits. Combining the conditional bits is done using an `xor` instruction. The number of conditional swaps to perform in total is now 256, since the last step cannot combine the last conditional swap with any other step.

| Limbs<br>Instr. | 255 bit<br>L=5 H=5<br>mult. | 130 bit<br>L=3 H=2<br>mult. | 78 bit<br>L=2 H=1<br>mult. | 52 bit<br>L=1 H=1<br>mult. |
|---|---|---|---|---|
| add. | 195+175 | 46+35 | 16+20 | 5 |
| shift | 102+40 | 35+6 | 14+5 | 5 |
| `and` | 75+87 | 22+15 | 8+10 | 6 |
| subtr. | 29 | 12 | 8 | 4 |
| slt | 27 | 9 | 5 | 1 |
| `xor` | 41 | 13 | 8 | 6 |
| neg. | 29 | 11 | 7 | 3 |
| `or` | 10 | 0 | 1 | 3 |
| mult. | 150+21 | 44 | 18 | 6 |
| Total | 1665 | 424 | 192 | 63 |

Table 4.6: Refined Karatsuba instruction count for Curve25519 with intrinsics where the most efficient method is chosen for L and H limb multiplication

$$\begin{aligned} m &= b - 1 \\ t' &= m \wedge (x_1 \oplus x_2) \\ r_1 &= t' \oplus x_2 \\ r_2 &= t' \oplus x_1 \end{aligned} \tag{4.2}$$

The subtraction operations in a step of Curve25519 are different from other subtractions in refined Karatsuba. The subtractions in refined Karatsuba are either known to give a positive result or its absolute value is taken. This subtraction can give a negative result, which in a field becomes positive by adding the modulus to it. The addition of the modulus will need to be checked for overflow and the overflow needs to be handled. The subtraction will take 10 subtraction. To handle the borrows 9 addition and 9 set-less-than instructions are needed. A mask is created to check if the value is negative or not taking one set-less-than and one negation instruction. To add the modulus and handle overflow 20 addition, 10 shift and 10 `and` instructions are needed and an extra multiplication by 608 is used for the overflow of the most significant limb. Adding the most significant limb can create a value larger than the base in the least significant limb. To handle this an extra 10 addition and shift instructions are needed, resulting in a total of 10 subtraction, 39 addition, 10 set-less-than, 10 `and`, 1 negation and 20 shift instructions and 1 multiplication by 608 operation.

The last operation to analyze in a step is the addition. The addition does not need to be checked for overflows thus only 10 addition operations are needed. The total instruction count for all step operations can be found in table 4.7. In the table the count is given per operation. Per operation the count is given for a single operation and the number of times the operation is used in a step. The trick to combine the conditional swap is not taken into account yet. The total cycle count of a complete step is the summation of all totals with the multiplication instruction costing 5 cycles giving 15733 cycles without combining the conditional swap and $15733 - 81 + 1 = 15653$ cycle with combining the conditional swaps.

After all the steps have been taken one value must be taken to the power $2^{255} - 21$. This operation takes 254 squarings and 11 multiplication operations, which in this implementation become 265 multiplications. Finally one more multiplication is used and a freeze operation. The freeze operation is performed as explained in section 4.2.1 except the number of limbs is doubled, resulting in 10 `and`, 1 negation, 11 shift, 20 addition and 20 `xor` instructions.

The final operation is changing the base to $2^8$ and storing the result. Storing the result takes 32 instructions and changing the base takes 74 instructions (see appendix A.5 for the instructions). The final count for Curve25519 scalar-multiplication is $15\,733 \cdot 254 + 15\,733 + 265 \cdot 1665 + 62 + 106 = 4\,432\,988$ cycles.

| | Multiplication | | Swap | | Addition | | Subtraction | | Mult. by 121665 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Instr. | 1 | 9 | 1 | 2 | 1 | 4 | 1 | 4 | 1 | |
| add. | 370 | 3330 | 0 | 0 | 10 | 40 | 39 | 156 | 21 | 3547 |
| shift | 142 | 1278 | 0 | 0 | 0 | 0 | 20 | 80 | 20 | 1378 |
| `and` | 162 | 1458 | 20 | 40 | 0 | 0 | 10 | 40 | 10 | 1548 |
| subtr. | 29 | 261 | 1 | 2 | 0 | 0 | 10 | 40 | 0 | 303 |
| slt | 27 | 243 | 0 | 0 | 0 | 0 | 10 | 40 | 0 | 283 |
| `xor` | 41 | 369 | 60 | 120 | 0 | 0 | 0 | 0 | 0 | 489 |
| neg. | 29 | 261 | 0 | 0 | 0 | 0 | 1 | 4 | 0 | 265 |
| `or` | 10 | 90 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 100 |
| mult. | 171 | 1539 | 0 | 0 | 0 | 0 | 1 | 4 | 21 | 1564 |

Table 4.7: Instruction count for a single step in Curve25519

### 4.3.2 Without intrinsics

As in section 4.2.2 a 27 bit multiplication is used to not need a reduction after a single addition or subtraction operation before the multiplication. The multiplication by 608 is performed in two steps. First a multiplication by 19 is performed taking 2 shift and 2 addition instructions. The second step is creating the low and high limb in base $2^{26}$ which takes 1 `and` and 2 shift instructions. From the tables it can be concluded that refined Karatsuba is the more efficient multiplication method.

| | 255 bit | | 130 bit | | 78 bit | | 52 bit | |
|---|---|---|---|---|---|---|---|---|
| Instr. | mult. | 28 bit mult. | mult. | 28 bit mult. | mult. | 28 bit mult. | mult. | 28 bit mult. |
| add. | 273 | 8600 | 50 | 2150 | 18 | 774 | 8 | 344 |
| shift | 73 | 15600 | 0 | 3900 | 0 | 1404 | 0 | 624 |
| `and` | 10 | 16400 | 0 | 4100 | 0 | 1476 | 0 | 656 |
| subtr. | 0 | 600 | 0 | 150 | 0 | 54 | 0 | 24 |
| slt | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| `xor` | 0 | 400 | 0 | 100 | 0 | 36 | 0 | 16 |
| neg. | 0 | 8400 | 0 | 2100 | 0 | 756 | 0 | 336 |
| `or` | 0 | 0 | 0 | | 0 | 0 | 0 | 0 |
| Total | 50356 | | 12550 | | 4518 | | 2008 | |

Table 4.8: Schoolbook multiplication Curve25519 without intrinsics

The count of a multiplication by 121665 is split into two parts the high and low limb. Before calculating any of the limbs it must be noted that 121665 in binary is $11101101101000001_2$. In the binary representation the pattern 11 occurs three times. It is more efficient to calculate that once and shift it to three different positions than shifting the value for each 1 in the binary representation. Calculating a multiplication by $11_2$ takes one shift and one addition instruction. To obtain $11011011_2$ two shift and two addition instructions are needed. Additionally 2 `and` instructions are used to obtain only the low limb value, without the `and` the summation overflows giving the wrong result. The summation becomes $11101101101_2$ after 2 shift, 2 addition and 2 `and` instructions. Finally one addition, shift and `and` instruction give the result for the low limb. The high limb is obtained in the same way except the shift direction and amount is different. To multiply a single limb by 121665 takes 13 shift, 14 `and` and 12 addition instructions. To add all the 20 limbs will take 10 addition instructions and one multiplication by 19. The multiplication by 19 is for the overflow of the most significant limb and the corresponding high limb result. Multiplication by 19 again takes 2 shift and 3 addition instructions. After this value has been added to the least significant limb the overflow most be handled once more, resulting in 10 `and`,

| Limbs | 255 bit L=5 H=5 | | 130 bit L=3 H=2 | | 78 bit L=2 H=1 | | 52 bit L=1 H=1 | |
|---|---|---|---|---|---|---|---|---|
| Instr. | mult. | 28 bit mult. | mult. | 28 bit mult. | mult. | 28 bit mult. | mult. | 28 bit mult. |
| add. | 2493+175 | 0 | 796+35 | 0 | 268+20 | 43 | 5 | 129 |
| shift | 4101+40 | 0 | 1361+6 | 0 | 478+5 | 78 | 5 | 234 |
| `and` | 4377+87 | 0 | 1444+15 | 0 | 504+10 | 82 | 6 | 246 |
| subtr. | 297+29 | 0 | 87+12 | 0 | 26+8 | 3 | 4 | 9 |
| slt | 72+27 | 0 | 15+9 | 0 | 2+5 | 0 | 1 | 0 |
| `xor` | 279+41 | 0 | 80+13 | 0 | 24+8 | 2 | 6 | 6 |
| neg. | 2262+29 | 0 | 743+11 | 0 | 258+7 | 42 | 3 | 126 |
| `or` | 51+10 | 0 | 17 | 0 | 6+1 | 0 | 3 | 0 |
| Total | 14370 | | 4644 | | 1880 | | 783 | |

Table 4.9: Refined Karatsuba with reduction instruction count for Curve25519 without intrinsics where the most efficient method is chosen for L and H limb multiplication

shift and addition instructions. The total instruction count for a multiplication by 121665 is 143 addition, 150 shift and 150 `and` instructions.

In table 4.10 the count for a step is given. The last difference between with and without intrinsics is that the freeze operation will use the refined Karatsuba method for multiplication. All other operations stay the same resulting in an instruction count of $254 \cdot 130255 + 130335 + 265 \cdot 14370 + 62 + 106 = 37\ 008\ 953$ cycles.

| | Multiplication | | Swap | | Addition | | Subtraction | | Mult. by 121665 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Instr. | 1 | 9 | 1 | 2 | 1 | 4 | 1 | 4 | 1 | |
| add. | 2668 | 24012 | 0 | 0 | 10 | 40 | 39 | 156 | 143 | 24351 |
| shift | 4141 | 37269 | 0 | 0 | 0 | 0 | 20 | 80 | 150 | 37499 |
| `and` | 4464 | 40176 | 20 | 40 | 0 | 0 | 10 | 40 | 150 | 40406 |
| subtr. | 326 | 2934 | 1 | 2 | 0 | 0 | 10 | 40 | 0 | 2976 |
| slt | 99 | 891 | 0 | 0 | 0 | 0 | 10 | 40 | 0 | 931 |
| `xor` | 320 | 2880 | 60 | 120 | 0 | 0 | 0 | 0 | 0 | 3000 |
| neg. | 2291 | 20619 | 0 | 0 | 0 | 0 | 1 | 4 | 0 | 20623 |
| `or` | 61 | 549 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 549 |

Table 4.10: Instruction count for a single step in Curve25519 without intrinsics

## 4.4 xSalsa20

xSalsa20, as described in section 3.3, is a stream cipher which given a nonce $n$, key $k$ and a length $\ell$ outputs a stream of pseudo-random bytes of length $\ell$. For convenience, the definition of xSalsa20 is repeated here. xSalsa20 starts off by creating a subkey $k'$ from $k$ and $n$ using hSalsa20. Using Salsa20 a block $b$ of the stream consisting of 16 words is created. A word is in little endian format and is loaded from 4 bytes of a constant, the subkey $k'$ or the nonce $n$. Loading a word costs 4 load, 3 `or` and 3 shift instructions.

The block $b$ goes through a process of 20 rounds. Each round consists of 4 so called quarter-rounds. In a quarter-round a row or column of length 4 is transformed. During the transformation two cells are added and then rotated by a constant. During a rotation a byte is shifted to the left and any overflow wraps around costing 2 shift and 1 `or` instruction. After the rotation that value is `xor`'ed with the original cell value, resulting in 8 shift, 4 `or`, 4 `xor` and 4 addition instructions for a quarter-round. The cycle count for a full round becomes $4 \cdot (8 + 4 + 4 + 4) = 80$ cycles.

After the rounds the original block $b$ is added to the corresponding cell using 16 addition instructions. Finally the result is stored in little endian format. The total cycle count to create 16 bytes of stream data is $20 \cdot 80 + 160 + 32 = 1792$ cycles.

The hSalsa20 function is very similar to Salsa20. The difference between the two is the addition of block $b$ at the end, which hSalsa20 does not do and it stores only 8 words. This results in a cycle count of $1792 - 16 - 56 = 1720$ cycles. The cycle count of xSalsa20 depends on $\ell$ and is $1720 + 1792 \cdot \lceil \frac{\ell}{16} \rceil$.

# Chapter 5

# Correctness

In this chapter a description is given on how the implementation is tested and how the absence of timing attacks is analyzed.

## 5.1 Test framework

To test if the implementations are correct a fuzz-test is performed. The fuzz-test is split into multiple test cases. A test case does multiple runs of one function with the same random input. The difference of the cycle counter is taken after each run of a function to obtain the cycle count. Each test case is flashed to the board. The output from the board is read by the framework. The framework stores the output to a file and compares the result with the expected result. The expected result for Poly1305, Curve25519 and Salsa20 is computed using libsodium [8] for all other functions microsalt [10] is used. These were used instead of the original NaCl, since those had bindings for the programming language used in the test framework. A test case also returns information on the instruction cache misses and the branch mispredictions, which are used to check the reliability of the cycle count. The core functions, Poly1305, Curve25519 and Salsa20, passed 500 test cases of 50 runs for both implementations. The remaining functions of the implementation without intrinsics use 200 test cases of 50 runs to keep a reasonable testing time. For the implementation with intrinsics all functionalities are tested using 500 test cases of 50 runs.

## 5.2 SMArTCAT

In section 2.3 SMArTCAT is introduced as a tool for ARM Cortex-A7. To be able to use SMArTCAT in this thesis multiple modifications have to be made. The first modification is updating the tool from Python2 to Python3, since Angr, a dependency of SMArTCAT, upgraded to Python3 as well. The second modification is enabling SMArTCAT to read RISC-V binaries. SMArTCAT relies on Angr to read the binary and symbolically execute it. Therefore I made an lifter extension for Angr. A lifter translates a binary into VEX, the interemediate respresentation Angr uses, enabling the symbolic execution. To create a lifter a translation is made from a set of bytes to an instruction. I modelled all possible instructions, in this case the basic RISC-V 32 bit instructions and the extensions multiplication and divide, compact and atomic, for Angr. In appendix A.7 the location of the source code and how to use it in Angr is given.

The final modification is changing the timing model used in SMArTCAT to fit the pipeline of the board. The pipeline of the board is described in [24]. The entire pipeline has been modeled for the board according to the manual. There can still be small differences between the manual and the actual test result due to the branch predictor not being modeled. The branch predictor is not modeled, since the algorithms used during the prediction are not public.

To improve the performance of SMArTCAT a skip functionality is added. The skip is used to ignore a part of the binary that is already shown to not contain timing vulnerabilities of type 1 and 2. Using this it is possible to first run the simulation on the multiplication and, when no timing vulnerabilities are found, the multiplication can be skipped during the simulation of a core function. Given that multiplication is performed often skipping the instructions saves not only time, but also memory usage. The reduction in memory usage made it possible to disable reduction steps. Normally the reduction steps are used to free up memory at the cost of longer computation times. Therefore the memory usage increases and the execution time decreases, which results in a large performance increase.

One issue encountered during the usage of SMArTCAT is related to the compact extension. SMArTCAT uses the Capstone engine [7] to determine which registers are used by an instruction. This information is needed to check for latency- and load-interlocking. The issue with the Capstone engine is the lack of compact instruction support. The compact instructions are used in the benchmarks, but have been disabled for the simulation in SMArTCAT. More information on why this is possible is given in section 6.2.

# Chapter 6

# Results

In this chapter benchmarks are presented. The benchmarks are obtained using the test framework from section 5.1. The benchmarks are obtained by measuring the cycle count on the Sifive Hifive1 rev B. Other information collected during the runs are the branch misprediction count and the instruction cache busy count, which are used to analyse the different cycle counts between runs.

## 6.1 Benchmarks

An advantage of the Sifive Hifive1 rev-B board are the two performance monitors which can measure certain metrics during execution. To be able to give more accurate results these monitors are used to measure events that impact the cycle count without being part of the computation. One monitor will measure the number of times that the instruction cache was busy, which could be due to loading the instructions from cache or other reasons. The second monitor counts the mispredictions made by the branch predictor. All benchmarks have mispredictions from the branch predictor. Each test case also has at least one run, for the definition see section 5.1, where the instruction cache is busy, since the instructions need to be loaded from flash at least once. In a test case there can be runs which do not have any instruction cache busy counts, which give the most reliable results. Instead of the instruction cache busy count the instruction cache misses count could also be used. However this is less accurate in representing the delay caused by a cache miss than the instruction cache busy count.

To compile the C-code the maximum optimization level is used; for the assembly code no optimization is used. The compact extension is also used during the benchmark to reduce the number of instruction cache misses leading to more reliable results. The Sifive Freedom metal library [22] is used to send data back over the TTY, by providing the 'stdio' library. The implementations do not use dynamic memory allocation and all RAM usage is from the stack. The size of the stack has been increased as a precaution to 0x1800 bytes to prevent running out of stack space. The size of the static library is 50 kB with intrinsics and 58 kB without intrinsics.

### 6.1.1 With intrinsics

The `crypto_box` function with intrinsics is the only function where the instruction cache was busy during execution for all runs. The large instruction cache busy count is expected, since it depends on Curve25519 scalar multiplication, Poly1305 and xSalsa20 functionality. It is not possible to keep all these instructions in the cache thus a count larger than 0 is expected. In figure 6.1 the `crypto_box` benchmarks with a message of 1024 bytes is given. A linear relation between the cycle count and the instruction busy count is visible indicating that a higher instruction cache busy count increases the cycle time. At the 900 000 instruction cache busy count a gap is visible. This gap is due to the runs with a larger count being the first run of a test case. The first run

loads all the instructions from flash while in the runs with a smaller count only the sections which have been replaced need to be loaded, which is less than loading all instructions.

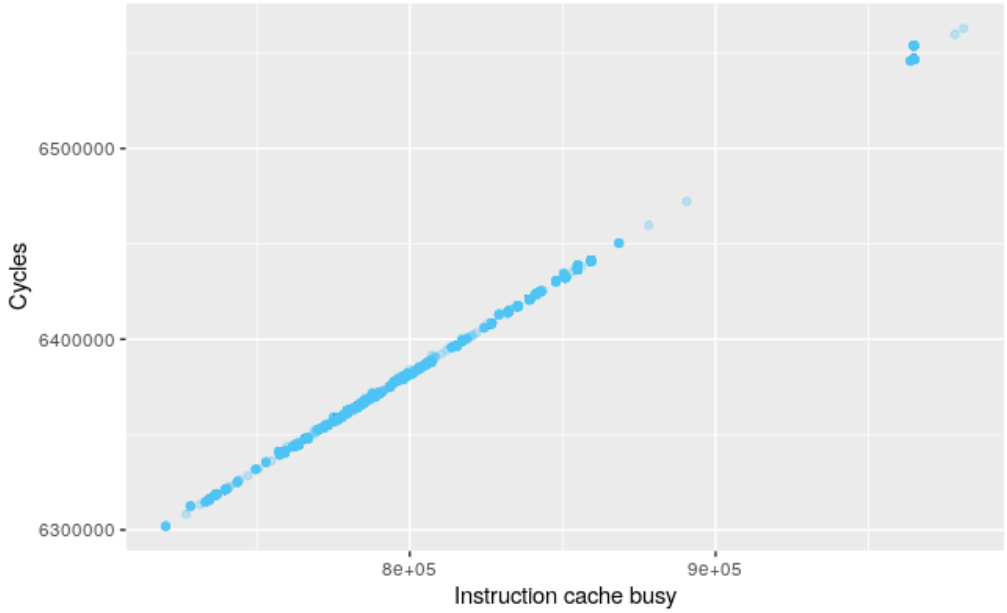


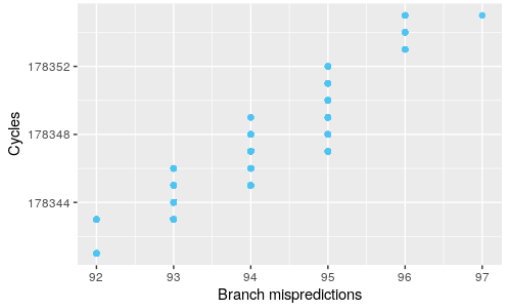

Figure 6.1: Cycle count relation to instruction cache busy for the `crypto_box` function

For all other functionality the instruction cache busy count was the same for all runs, except the first run where all instructions are loaded from the flash. For these functions the cycle count is plotted against the second metric, the branch mispredictions.

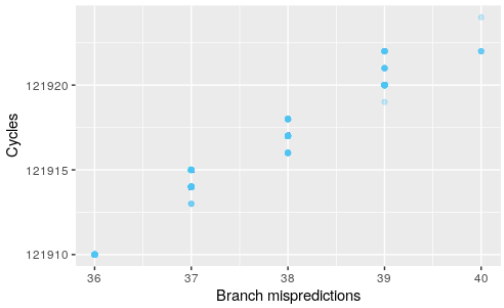
In figure 6.2 the graphs are given for the `secret_box` and xSalsa20 with a 1024 byte message and the scalar multiplication functionality. In the figures a linear relation is visible between the branch mispredictions and the cycle count indicating the same relation as `crypto_box` has with instruction cache busy count. One additional thing to note is that there are different cycle counts for the same branch misprediction count, clearly visible in figures 6.2a and 6.2b. The branch misprediction count is an aggregate of branch direction and branch target mispredictions. Each type of misprediction takes a different number of cycles to recover from. Aggregating the count of these types loses information, since multiple combinations give the same branch misprediction count. The composition of the number is however still visible in the cycle count, since the recovery of each type has a different cycle count, resulting in different cycle counts for the same branch misprediction count.

For the Poly1305 functionality different message sizes are benchmarked to see how it scales with larger messages. In figure 6.3 both the benchmark and the theoretical analysis is shown. It is visible that the two lines are diverging from each other. Indicating that extra cycles are used in the step in comparison to the theoretical analysis. The same type of divergence will also be in the `crypto_box` and `secret_box` functionality, since those depend on Poly1305 to digest the message. The slope of the benchmark in this graph is $\frac{147011-20451}{4096-512} = 35.125$ cycles per byte. Poly1305 consumes message blocks of 16 bytes per step therefore a single step takes $16 \cdot 35.125 = 562$ cycles.

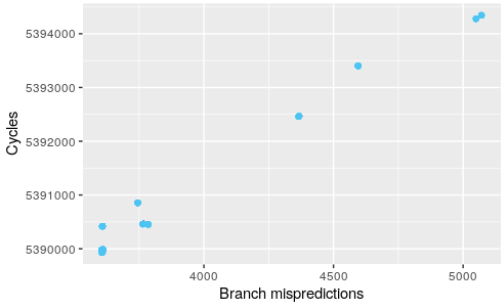
The median cycle counts are given for each function in table 6.1 with the respective theoretical count. If a function had a variable length input a 1024 byte input is used.




(a) `Secret_box` function with a 1024 byte message




(b) xSalsa20 function with a 1024 byte message




(c) Scalar multiplication on Curve25519

Figure 6.2: Relation between cycle count and the branch mispredictions

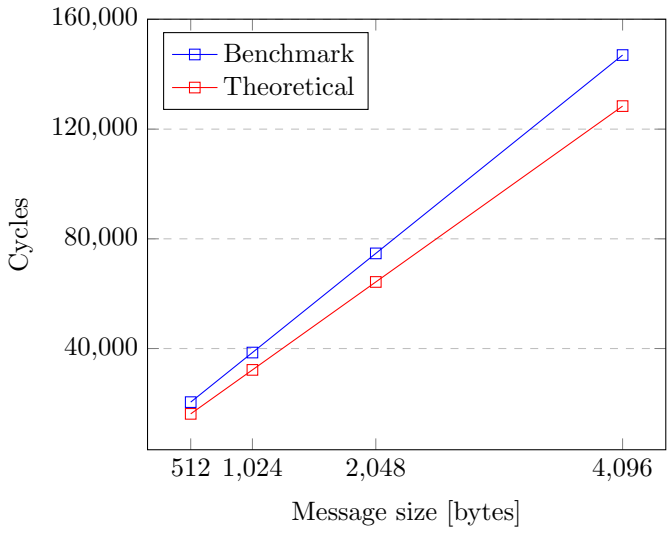



Figure 6.3: Poly1305 with intrinsics on different message sizes

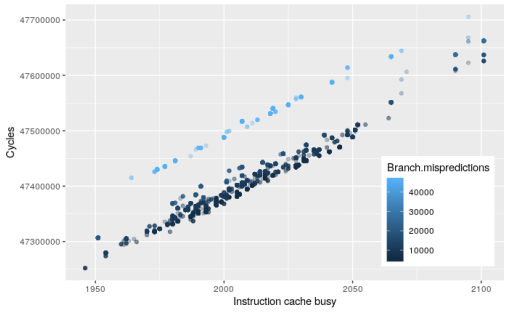
| Function | Poly1305 | Curve25519 | `Secret_box` | `Crypto_box` | xSalsa20 |
|---|---|---|---|---|---|
| Benchmark | 38 530 | 5 389 988 | 178 347 | 6 371 674 | 121 910 |
| Analysis | 29 709 | 4 432 988 | - | - | 116 408 |

Table 6.1: Cycle counts per function with a 1024 byte message for the implementation with intrinsics

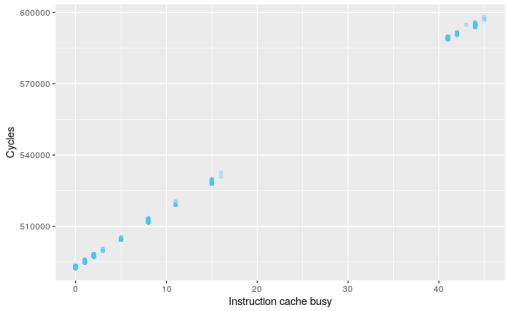
### 6.1.2 Without intrinsics

Due to the additional instructions needed to perform a multiplication all functionalities except the xSalsa20 stream have a non-constant instruction cache busy count. The figures in 6.4 show all functionalities having a non-constant instruction cache busy count. In these figures the first run of each test case has been ignored, since the gap, explained in section 6.1.1, between the data shown and the first run of a test case became too large reducing the visibility of the relevant data. Note that there is a large gap is visible in figure 6.4b and 6.4d. It is unclear why such an gap exists.

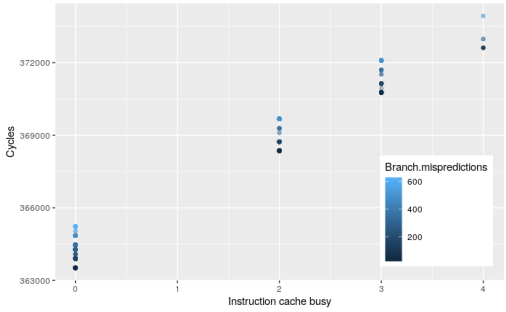
In figures 6.4a, 6.4c and 6.4d the color of the scatter plot also indicates the number of branch mispredictions. In these figures it can be seen that the lower the branch misprediction count the lower the cycle count.



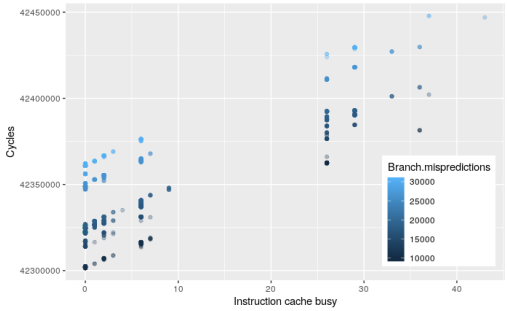

(a) `Crypto_box` function with a 1024 byte message




(b) `Secret_box` function with a 1024 byte message




(c) Poly1305 with a 1024 byte message



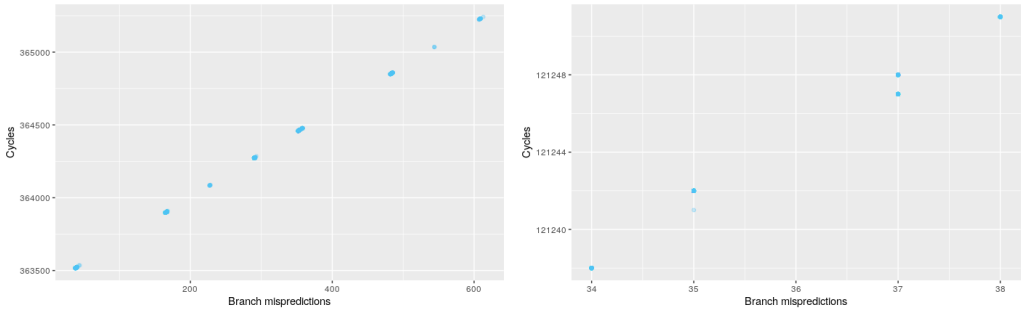

(d) Scalar multiplication on Curve25519

Figure 6.4: Relation between cycle count and the instruction cache busy count for functions with a non-constant instruction cache busy count

In figure 6.4c the instruction cache busy count is 0 for most cases (23988 out of 25000 runs). Therefore Poly1305 is filtered on cases where the instruction cache busy count is 0 and the cycle count is plotted against the branch misprediction count in 6.5a. In figure 6.5 the same relations as in 6.2 can be seen.

In figure 6.6 the median cycle count of different message sizes is shown in comparison to the theoretical limit. A similar divergence as 6.3 is visible indicating a difference in cycle count per step. The slope of the benchmark is $\frac{1\ 437\ 377 - 185\ 261}{4096 - 512} = 349.36$ cycles per byte. Poly1305 does not process a single byte but a block of 16 bytes therefore the cycles per step is approximately $16 \cdot 349.36 = 5589.76$ cycles.

The median cycle count for all functions of the implementation without intrinsics can be found in table 6.2. As in table 6.1 an input of 1024 bytes is used for all function with variable length input field.




(a) Poly1305 function with a 1024 byte message (b) xSalsa20 function with a 1024 byte message

Figure 6.5: Relation between cycle count and branch mispredictions

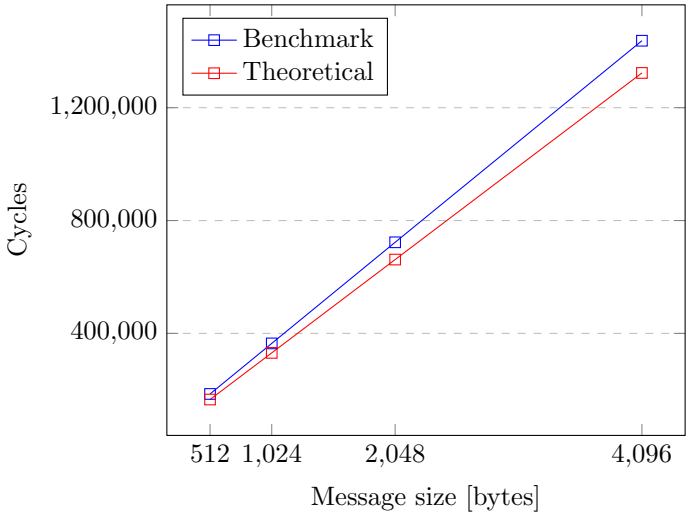



Figure 6.6: Poly1305 without intrinsics on different message sizes

| Function | Poly1305 | Curve25519 | `Secret_box` | `Crypto_box` | xSalsa20 |
|---|---|---|---|---|---|
| Benchmark | 364 461 | 42 325 262 | 492 454 | 47 405 549 | 121 247 |
| Analysis | 299 085 | 37 008 953 | - | - | 116 408 |

Table 6.2: Cycle counts per function with a 1024 byte message for the implementation without intrinsics

### 6.1.3 Comparison of theoretical analysis with related work

In [6] an implementation for ARM-based processor with the NEON vector instruction set is presented. The vector instructions provides 128 bit registers and can do computation on those. The authors present cycles counts of 5.30 cycles per byte for secret-key cryptography and 527 102 cycles for a Curve25519 scalar multiplication.

To be able to compare the result with the counts from [6] an upscale factor is used. As a rough estimate $\frac{128}{32} = 4$ is taken, since four 32 bit registers contain the same amount of information as one 128 bit register. This results in 21.20 cycles per byte for secret-key cryptography and 2 108 408 cycles for a Curve25519 scalar multiplication. To be able to compare the Poly1305 benchmark against the theoretical analysis a message of 1024 bytes is used resulting in $21.20 \cdot 1024 = 21\ 708.8$ cycles. The Poly1305 implementation takes 30 541 cycles and the Curve25519 scalar multiplication takes 3 962 798 cycles according to the theoretical analysis with intrinsics. Comparing the rough

estimates to the theoretical analysis it is found that they are close to the minimum given for Poly1305. For Curve25519 there is a larger difference. This can be explained by the fact that ARM-based processor can handle carries. Therefore removing all instructions that are used for carry handling, which in Curve25519 is significantly higher than in Poly1305. Confirming that the theoretical analysis is a reasonable minimum.

## 6.2 SMArTCAT results

To analyze the binary the compact extension, which is used in the benchmarks to ensure that no instruction cache misses are present, is disabled, since compact instructions are not supported as stated in section 5.2. This change does not alter the simulation, since the compact instruction have the same timings as the non-compact instructions. Only Poly1305, Curve25519 scalar multiplication and Salsa20, which are the building blocks for the other functions, are simulated in SMArTCAT. For the simulations with intrinsics only 1 distinct path is found, which means that no timing attacks are possible due to branches or loads depending on secret data. The simulation of Poly1305 and Salsa20 without intrinsics also found only 1 distinct path. The simulation of Curve25519 scalar multiplication did not finish in a reasonable time on an Intel Core i7-8750H. To speed up the simulation the refined Karatsuba computation is skipped and simulated separately. Both simulations, of refined Karatsuba and scalar multiplication without intrinsics, reported only 1 distinct path. The combination of these simulations shows that there is only 1 distinct path in the aggregate, since the simulation showed that the refined Karatsuba has only 1 distinct path for all possible inputs. Therefore the simulation of the aggregate would also show only 1 distinct path if it would finish in a reasonable time.

# Chapter 7

# Discussion

In this thesis two implementations in RISC-V are provided of the NaCl library. Both implementation have all functions except for the signature functionalities. The signature functionalities are similar to the Curve25519 scalar multiplication. To implement the functions the 32-bit base instruction set is used. One implementation also used the multiplication and division extension. Each implementation is checked using a modified version of SMArTCAT for the absence of timing channel attacks due to branch and load instructions depending on secret data. The self-composition proof did not find any such attacks on both implementations. The modifications made to SMArTCAT consist of writing the pipeline model of the Sifive Hifive1 rev-B board, the lifter of RISC-V binary code and an upgrade to Python3. All of the code written for this thesis is made available (for more details see appendix A.6 and A.7 or [25, 26]).

Comparing the Poly1305 and Curve22519 scalar multiplication benchmarks between the two implementations, see table 6.1 and 6.2 for the benchmarks, shows a factor of 9.5 and 7.9 decrease in cycle count respectively. For `secret_box` and `crypto_box` the decrease in cycle count is by a factor 2.76 and 7.44 respectively. The low factor of decrease in cycle counts of `secret_box` is due to the negligible difference in the xSalsa20 cycle counts, which is half of the computation in `secret_box` . The decrease in cycle count comes at a reasonably low cost, since most RISC-V processors will have the multiplication and division extension.

In the tables 6.1 and 6.2 a difference is found between the theoretical analysis and the benchmark. This difference is due to the extra load and store operations that have not been taken into account in the analysis. Most of these load and store operations occur when a function is called with more than 8 inputs. The arguments then need to be stored on the stack and loaded afterwards. Next to that when a function is called and information needs to be stored in a register a saved-register must be used. To use such a register an extra load and store operation is used. Additionally small differences exist between the process described in chapter 4 and the implementation to increase the readability and quality of the code. Finally other instructions used to load constants, do loops and change the stack size have not been taken into account in the analysis. To obtain a more accurate cycle count from theoretical analysis these factors can be taken into account and the implementation can be improved for a better comparison.

Next to improving the theoretical analysis the signature function can still be provided. During the implementation of the signature function the finite field arithmetic from the Curve25519 scalar multiplication can be reused. To improve the benchmarks the refined Karatsuba could be replaced by the technique from [4], which has a simpler data flow. Finally the floating point extension or the vector operations extension could be used to improve the performance of the overall solution.

# Bibliography


[1] Unit 42. 2020 Unit 42 IoT Threat Report, Mar 2020. https://unit42.paloaltonetworks.com/iot-threat-report-2020/, accessed at 23-03-2020. 1

[2] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. 4, 5, 9

[3] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008. 10

[4] Daniel J. Bernstein. Batch Binary Edwards. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009. 33

[5] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012. 1, 4

[6] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012. 31

[7] Capstone. The Ultimate Disassembly Framework, Jan 2019. https://www.capstone-engine.org/, accessed at 23-03-2020. 26

[8] Frank Denis. libsodium documentation, 2020. https://download.libsodium.org/doc/, accessed at 23-03-2020. 25

[9] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976. 4

[10] goldenMetteyya. Crate microsalt. https://docs.rs/microsalt/0.2.21/microsalt/, accessed at 23-03-2020. 25

[11] Anthony Van Herrewege, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede. Secure PRNG seeding on commercial off-the-shelf microcontrollers. In *TrustED@CCS*, pages 55–64. ACM, 2013. 10

[12] Michael Hutter and Peter Schwabe. $\mu$NaCl – The Networking and Cryptography library for microcontrollers, 2013. https://munacl.cryptojedi.org/index.shtml, accessed at 11-02-2020. 1

[13] Michael Hutter and Peter Schwabe. Multiprecision multiplication on AVR revisited. *J. Cryptographic Engineering*, 5(3):201–214, 2015. 8

[14] Anatoly Karatsuba and Yuri Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595–596, 1963. 7

[15] Camille Kokozaki. The Revolution Evolution Continues - SiFive RISC-V Technology Symposium - Part I, 18-03-2019. https://www.sifive.com/blog/the-revolution-evolution-continues---sifive-risc-v, accessed at 11-02-2020. 1

[16] Roeland Krak. Cycle-Accurate Timing Channel Analysis of Binary Code, May 2017. http://essay.utwente.nl/72321/, accessed at 03-05-2020. 1, 6

[17] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987. 9

[18] RISC-V Foundation. About the RISC-V Foundation. https://riscv.org/risc-v-foundation/, accessed at 13-02-2020. 1

[19] RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA, 2015. http://www.rstudio.com/, accessed at 23-04-2020. 41

[20] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015. 6

[21] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016. 6

[22] Sifive. Freedom Metal. https://sifive.github.io/freedom-metal-docs/, accessed at 23-03-2020. 27

[23] Sifive. HiFive1 Rev B, 2019. https://www.sifive.com/boards/hifive1-rev-b, accessed at 19-06-2019. 1, 3

[24] Sifive. SiFive FE310-G002 Manual, 2019. https://sifive.cdn.prismic.io/sifive%2F9ecbb623-7c7f-4acc-966f-9bb10ecdb62e_fe310-g002.pdf, accessed at 27-02-2020. 3, 25

[25] Stefan van den Berg. NaCl in RISC-V . https://github.com/stefanberg96/NaCl-RISC-V, accessed at 23-04-2020. 33, 41

[26] Stefan van den Berg. SMArTCAT. https://github.com/stefanberg96/SMArTCAT, accessed at 23-04-2020. 33, 41

[27] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016. 6

[28] Symantec. ISTR 2019: Internet of Things Cyber Attacks Grow More Diverse, 2019. https://www.symantec.com/blogs/expert-perspectives/istr-2019-internet-things-cyber-attacks-grow-more-diverse, accessed at 25-06-2019. 1

[29] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121, Dec 2019. https://riscv.org/specifications/isa-spec-pdf/, accessed at 27-02-2020. 3

# Appendix A

# Base transformations

## A.1 Poly1305 loading a key into base $2^{26}$

There are some additional `and` operations used to ensure certain requirements of the key $k$.

```
void loadKey( unsigned int* k, unsigned char* r){
  r[0] = k[0] + (k[1] << 8) + (k[2] << 16) + ((k[3] & 3) << 24);
  r[1] = ((k[3] >> 2) & 3) + ((k[4] & 252) << 6) + (k[5] << 14) +
         ((k[6] & 15) << 22);
  r[2] = (k[6] >> 4) + ((k[7] & 15) << 4) + ((k[8] & 252) << 12) +
         ((k[9] & 63) << 20);
  r[3] = (k[9] >> 6) + (k[10] << 2) + ((k[11] & 15) << 10) + ((k[12] & 252) << 18);
  r[4] = k[13] + (k[14] << 8) + ((k[15] & 15) << 16);
}
```

## A.2 Poly1305 loading a part of the message into base $2^{26}$

In the listing below the RISC-V assembly code is given to load 16 bytes in base $2^{26}$.

```
    lbu     t2, 3(s0)     # in[3]
    lbu     t1, 2(s0)     # in[2]
    lbu     t0, 1(s0)     # in[1]
    andi    a0, t2, 3     # c[0] = in[3] & 3
    slli    a0, a0, 8     #
    or      a0, a0, t1    # c[0] << 2 +in[2]
    lbu     t1, 0(s0)     # in[0]
    slli    a0, a0, 8     #
    lw      t4, 0(s2)     # h[0]
    or      a0, a0, t0    # c[0] << 8 + in[1]
    slli    a0, a0, 8     #
    or      a0, a0, t1    # c[0] << 8 + in[0]
    add     t4, t4, a0    # c[0] + h[0]
    sw      t4, 0(s2)     # store h[0] += c[0]

    lbu     t3, 6(s0)     # in[6]
    lbu     t1, 5(s0)     # in[5]
    lbu     t0, 4(s0)     # in[4]
    andi    a0, t3, 15    # c[1] = in[6] & 15
    slli    a0, a0, 8     # c[1] << 4
    or      a0, a0, t1    # c[1] += in[5]
    slli    a0, a0, 8     # c[1] << 8
    or      a0, a0, t0    # c[1] += in[4]
    lw      t4, 4(s2)     # load h[1]
    slli    a0, a0, 6     # c[1] << 6
    srli    t2, t2, 2     # in[3] >> 2
    or      a0, a0, t2    # c[1] += in[3]
    add     t4, t4, a0    #
    sw      t4, 4(s2)     # store h[1] += c[1]

    lbu     t2, 9(s0)     # in[9]
    lbu     t1, 8(s0)     # in[8]
    lbu     t0, 7(s0)     # in[7]
    andi    a0, t2, 63    # c[2] = in[9] & 63
    slli    a0, a0, 8     # c[2] <<= 2
    or      a0, a0, t1    # c[2] += in[8]
    slli    a0, a0, 8     # c[2] <<= 8
    lw      t4, 8(s2)     # load h[2]
    or      a0, a0, t0    # c[2] += in[7]
    slli    a0, a0, 4     # c[2] <<= 2
    srli    t3, t3, 4     # in[6] >> 4
    or      a0, a0, t3    # c[2] += in[6]
    add     t4, t4, a0    #
    sw      t4, 8(s2)     # store h[2] += c[2]

    lbu     a0, 12(s0)    # in[12]
    lbu     t1, 11(s0)    # in[11]
```

```
    lbu     t0, 10(s0)    # in[10]
    slli    a0, a0, 8     # c[3] <<= 8
    or      a0, a0, t1    # c[3] += in[11]
    slli    a0, a0, 8     # c[3] <<= 8
    or      a0, a0, t0    # c[3] += in[10]
    lw      t4, 12(s2)    # load h[3]
    slli    a0, a0, 2     # c[3] <<= 2
    srli    t2, t2, 6     # in[9] >> 6
    or      a0, a0, t2    # c[3] += in[9]
    add     t4, t4, a0    #
    sw      t4, 12(s2)    # store h[3] += c[3]

    lbu     a0, 15(s0)    # in[15]
    lbu     t1, 14(s0)    # in[14]
    lbu     t2, 13(s0)    # in[13]
    addi    a0, a0, 0x100 # append the 1 bit at the end
    slli    a0, a0, 8     # c[4] <<= 8
    lw      t4, 16(s2)    # load h[4]
    or      a0, a0, t1    # c[4] += in[14]
    slli    a0, a0, 8     # c[4] <<= 8
    or      a0, a0, t2    # c[4] += in[13]
    add     t4, t4, a0    #
    sw      t4, 16(s2)    # store h[4] += c[4]
```

## A.3 Poly1305 from base $2^{26}$ to base $2^8$

```
void toradix28_130(unsigned int h[17]) {
  h[16] = (h[4] >> 24);
  h[15] = (h[4] >> 16) & 0xFF;
  h[14] = (h[4] >> 8) & 0xFF;
  h[13] = h[4] & 0xFF;
  h[12] = (h[3] >> 18) & 0xFF;
  h[11] = (h[3] >> 10) & 0xFF;
  h[10] = (h[3] >> 2) & 0xFF;
  h[9] = (h[2] >> 20) + ((h[3] & 3) << 6);
  h[8] = (h[2] >> 12) & 0xFF;
  h[7] = (h[2] >> 4) & 0xFF;
  h[6] = (h[1] >> 22) + ((h[2] & 0x0F) << 4);
  h[5] = (h[1] >> 14) & 0xFF;
  h[4] = (h[1] >> 6) & 0xFF;
  h[3] = (h[0] >> 24) + ((h[1] & 0x3f) << 2);
  h[2] = (h[0] >> 16) & 0xFF;
  h[1] = (h[0] >> 8) & 0xFF;
  h[0] = h[0] & 0xFF;
}
```

## A.4 Load 32 bytes in base $2^{26}$

```
void convert_to_radix226_255(unsigned int *r, const unsigned char *k) {
  r[0] = k[0] + (k[1] << 8) + (k[2] << 16) + ((k[3] & 3) << 24);
  r[1] = (k[3] >> 2) + (k[4] << 6) + (k[5] << 14) + ((k[6] & 15) << 22);
  r[2] = (k[6] >> 4) + (k[7] << 4) + (k[8] << 12) + ((k[9] & 63) << 20);
  r[3] = (k[9] >> 6) + (k[10] << 2) + ((k[11]) << 10) + (k[12] << 18);
  r[4] = k[13] + (k[14] << 8) + (k[15] << 16) + ((k[16] & 3) << 24);
  r[5] = (k[16] >> 2) + (k[17] << 6) + (k[18] << 14) + ((k[19] & 15) << 22);
  r[6] = (k[19] >> 4) + (k[20] << 4) + (k[21] << 12) + ((k[22] & 63) << 20);
  r[7] = (k[22] >> 6) + (k[23] << 2) + ((k[24]) << 10) + (k[25] << 18);
  r[8] = k[26] + (k[27] << 8) + (k[28] << 16) + ((k[29] & 3) << 24);
  r[9] = (k[29] >> 2) + (k[30] << 6) + (k[31] << 14);
}
```

## A.5 Store a 260 bit value in base $2^8$

```
void toradix28_255(unsigned char out[32], unsigned int in[10]) {
  out[31] = (in[9] >> 14);
  out[30] = (in[9] >> 6) & 0xFF;
  out[29] = (in[8] >> 24) + ((in[9] & 0x3F) << 2);
  out[28] = (in[8] >> 16) & 0xFF;
  out[27] = (in[8] >> 8) & 0xFF;
  out[26] = in[8] & 0xFF;
  out[25] = (in[7] >> 18) & 0xFF;
  out[24] = (in[7] >> 10) & 0xFF;
  out[23] = (in[7] >> 2) & 0xFF;
  out[22] = (in[6] >> 20) + ((in[7] & 0x3) << 6);
  out[21] = (in[6] >> 12) & 0xFF;
  out[20] = (in[6] >> 4) & 0xFF;
  out[19] = (in[5] >> 22) + ((in[6] & 0x0F) << 4);
  out[18] = (in[5] >> 14) & 0xFF;
  out[17] = (in[5] >> 6) & 0xFF;
  out[16] = (in[4] >> 24) + ((in[5] & 0x3F) << 2);
  out[15] = (in[4] >> 16) & 0xFF;
  out[14] = (in[4] >> 8) & 0xFF;
  out[13] = in[4] & 0xFF;
  out[12] = (in[3] >> 18) & 0xFF;
  out[11] = (in[3] >> 10) & 0xFF;
  out[10] = (in[3] >> 2) & 0xFF;
  out[9] = (in[2] >> 20) + ((in[3] & 3) << 6);
  out[8] = (in[2] >> 12) & 0xFF;
  out[7] = (in[2] >> 4) & 0xFF;
  out[6] = (in[1] >> 22) + ((in[2] & 0x0F) << 4);
  out[5] = (in[1] >> 14) & 0xFF;
  out[4] = (in[1] >> 6) & 0xFF;
  out[3] = (in[0] >> 24) + ((in[1] & 0x3F) << 2);
  out[2] = (in[0] >> 16) & 0xFF;
  out[1] = (in[0] >> 8) & 0xFF;
  out[0] = in[0] & 0xFF;
}
```

## A.6 NaCl in RISC-V

In [25] the implementations of this thesis are available. Multiple folders are shown in the repository.

The `Parsing` folder contains the code used to parse the output from the test framework to work with RStudio [19] for analysis. In the `RStudio` the file used for the analysis and the figures used in chapter 6 can be found.

The `TestFramework` folder contains the test framework from section 5.1. For each function that is tested a generator is created, which creates a test-case, the expected result and a benchmark file. The benchmark file is included in the `make` command, which builds the new test case file and flashes it to the board. The reader will report back the results from the board and print it a file.

Finally in the `Programs` folder two sub-folders are found each containing one of the implementations. The folders for each implementation uses the same structure as the original NaCl library with a few additional folder to store the build files, the log files and the results. To create the library command `make hex` is used, which creates the `lib.a` and `program.elf` file.

## A.7 SMArTCAT

SMArTCAT is made available on [26]. To be able to run the tool the virtual environment `latestversion` needs to be activated. This virtual environment contains all the necessary dependencies. To be able to check the implementations provided the `elf`-files need to be provided, where to find these is described in appendix A.6. The paths to the hex file and the hex-address of the corresponding function needs to be changed in any of the `run_` python-files. The hex-address can be found by running `make objdump file=build/program.elf`, which creates an object dump of the elf file. In the object dump the hex address for all functions can be found. Afterwards the `run_` file can be executed in python. The result will be printed to the console. The original test results can be found in the result folder for comparison to check if the tool worked correctly.

The platforms folder contains the lifter from RISC-V binary to VEX, the IR of angr. The lifter used in this project is added to angr in pull-request 35 of the angr-platforms repository.