THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Darius MERCADIER

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Usuba, Optimizing Bitslicing Compiler

soutenue le 20 novembre 2020

devant le jury composé de :

M. Gilles MULLER	Directeur de thèse
M. Pierre-Évariste DAGAND	Encadrant de thèse
M. Karthik BHARGAVAN	Rapporteur
Mme. Sandrine BLAZY	Rapporteur
Mme. Caroline COLLANGE	Examinateur
M. Xavier LEROY	Examinateur
M. Thomas PORNIN	Examinateur
M. Damien VERGNAUD	Examinateur

Abstract

Bitslicing is a technique commonly used in cryptography to implement high-throughput parallel and constant-time symmetric primitives. However, writing, optimizing and protecting bitsliced implementations by hand are tedious tasks, requiring knowledge in cryptography, CPU microarchitectures and side-channel attacks. The resulting programs tend to be hard to maintain due to their high complexity. To overcome those issues, we propose Usuba, a high-level domain-specific language to write symmetric cryptographic primitives. Usuba allows developers to write high-level specifications of ciphers without worrying about the actual parallelization: an Usuba program is a scalar description of a cipher, from which the Usuba compiler, Usubac, automatically produces vectorized bitsliced code.

When targeting high-end Intel CPUs, the Usubac applies several domain-specific optimizations, such as interleaving and custom instruction-scheduling algorithms. We are thus able to match the throughputs of hand-tuned assembly and C implementations of several widely used ciphers.

Futhermore, in order to protect cryptographic implementations on embedded devices against side-channel attacks, we extend our compiler in two ways. First, we integrate into Usubac state-of-the-art techniques in higher-order masking to generate implementations that are provably secure against power-analysis attacks. Second, we implement a backend for SKIVA, a custom 32-bit CPU enabling the combination of countermeasures against power-based and timing-based leakage, as well as fault injection.

Résumé

Le bitslicing est une technique utilisée pour implémenter des primitives cryptographiques efficaces et s'exécutant en temps constant. Cependant, écrire, optimiser, et sécuriser manuellement des programmes bitslicés est une tâche fastidieuse, nécessitant des connaissances en cryptographie, en microarchitecture des processeurs et en attaques par canaux cachés. Afin de remédier à ces difficultés, nous proposons Usuba, un langage dédié permettant d'implémenter des algorithmes de cryptographie symétrique. Usuba permet aux développeurs d'écrire des spécifications de haut niveau sans se soucier de leur parallélisation: un programme Usuba est une description scalaire d'une primitive, à partir de laquelle le compilateur Usuba, Usubac, produit automatiquement un code bitslicé et vectorisé.

Afin de produire du code efficace pour les processeurs haut de gamme, Usubac applique plusieurs optimisations spécialement conçues pour les primitives cryptographiques, telles que l'entrelacement et l'ordonnancement d'instructions. Ainsi, le code produit par notre compilateur offre des performances comparables à du code assembleur ou C optimisé à la main.

De plus, afin de générer des implémentations sécurisées contre des attaques par canaux cachés, nous proposons deux extensions de Usubac. Lorsque les attaques par analyse de courant sont un risque à considérer, Usubac est capable de protéger les implémentations qu'il produit à l'aide de masquage booléen. Si, additionellement, des attaques par injection de fautes doivent être prévenues, alors Usubac peut générer du code pour SKIVA, un processeur 32-bit offrant des instructions permettant de combiner des contre-mesures pour du code bitslicé.

Acknowledgments

Note: the names mentioned in this section are ordered using a non-disclosed algorithm. I recommend that you do not try to find a logic in the ordering; and especially that you do not wrongly assume that first is better than last.

First, I would like to thank Sandrine Blazy and Karthik Bhargavan for reviewing my manuscript. Having spent months trying to produce a good manuscript, I was very pleased that you enjoyed reading it, and your positive reviews really warmed my heart. I would also like to thank Damien Vergnaud, Xavier Leroy, Caroline Collange and Thomas Pornin for being part of the jury.

Of course, I am immensely grateful to Pierre-Évariste Dagand and Gilles Muller for supervising and guiding me throughout these 3 years. Pierre-Évariste, you always pushed me to strive for perfection, to leave no unturned stones, to try every possibilities, and to do the best work possible. Even though I probably wasn't the most docile PhD student, I enjoyed working with you, and I learned *a lot* during these 3 years, in large part thanks to you. Or, to put it otherwise (just for you), I won't say that I am unhappy to not have done this Ph.D. in another way.

A lot of work in this thesis would not have been possible without collaborations with great researchers. Patrick Schaumont, Pantea Kiaei, Raphaël Wintersdorff, Sonia Belaïd, Matthieu Rivain, Lionel Lacassagne, Karine Heydemann, I'm glad to have worked with you, and I hope this is neither the end of our collaborations, nor of our friendship. It was in particular really interesting to work with people from different sub-fields of cryptography: on one side, Patrick and Pantea, with your hardware expertise and low-level understanding of side-channel attacks, and on the other side, Raphaël, Sonia and Matthieu, with your theoretical approach to the problem. While not an official collaborator, I'm infinitely grateful to my brother Hector who helped me visualize and interpret quite a lot of data from my benchmarks.

I'd also like to thank some of my professors who inspired me to pursue this PhD and become a researcher: Jérémie Salvucci (although your last advice regarding the PhD was "run away"), Frédéric Peschanski, Antoine Genitrini, Emmanuel Chailloux, Julien Sopena, Quentin Meunier.

Three years in the same office working on the same topic would certainly have been too much for me, if it weren't for the amazing people surrounding me. In particular, Redha, Lucas, Antoine, Cédric and Pierre, whose eternal commitment to not being too serious made it a pleasure to work around them. And more generally, other PhD students, interns and post-doc from the Whisper, DELYS, MoVe and APR teams.

There is more to life than just research (hope I'm not offending too many people with that sentence). And one of the major reasons that those last 3 years were great is the time I spent with my friends (not really sorted): Pierre B, Julien P, Juliette, Camille, Théo, Guillaume, Ninon, Jordi, Rémy, Astrid and all the ones I'm forgetting. Thank you also to my cycling partners (Yacine, Jerome, Vicky and Antoine), and my Counter-Strike mates (Xili, Mando, John, Pasjordi, Arta, Thaur).

I'd like to thank my family (papa, maman, Hector, Briséïs, Kyra), for always being there and for being so great. While I might not show it, I'm glad to have you in my life.

Finally, I would like to thank the most important person to me, Marjorie, without whom life would be so grim. Beyond being the most amazing girlfriend ever, you are the nicest and most caring person I know, and living with you is an unending pleasure. Thank you for helping me get through the finish line of this PhD (I would probably have taken at least a few more months without you pushing me to work), and thank you for being in my life ♡

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography, from the Ancient Greek *kryptos* "hidden" and *graphein* "to write", is the practice of securing a communication by transforming its content (the *plaintext*) into an unintelligible text (the *ciphertext*), using an algorithm called a *cipher*, which often takes as additional input a secret *key* known only from the people encrypting and decrypting the communication. The first known use of cryptography dates back to ancient Egypt, in 1900 BCE. Almost 2000 years later, Julius Caesar was notoriously using cryptography to secure his orders to his generals, using what would later be known as a Caesar cipher, which consists in replacing each letter by another one such that the i^{th} letter of the alphabet is replaced by the $(n+i)^{th}$ one (for some fixed n between 1 and 25), wrapping around at the end of the alphabet. Throughout history, the military would continue to use cryptography to protect their communications, with the famous example of Enigma, used by Nazi Germany during World War II. Nowadays, in our increasingly digital and ever more connected world, cryptography is omnipresent, protecting sensitive data (*e.g.*, passwords, banking data) and securing data transfers over the Internet, using a multitude of ciphers.

The basic blocks of modern cryptography are called *cryptographic primitives*, and can be divided into three main categories:

- Asymmetric (or public-key) ciphers, which use two different keys for encryption and decryption: one is public and used for encryption, while the other one is private and used for decryption. Commonly used examples of asymmetric ciphers include RSA [265] and elliptic curves such as Curve25519 [59].

- Symmetric (or secret-key) ciphers, which use the same (secret) key for both encryption and decryption, and are subdivided into two categories:

  – Stream ciphers, which generate a pseudo-random bit-stream and `xor` it with the plaintext. The most widely used stream cipher in software is CHACHA20 [61].

  – Block ciphers, which only encrypt a single fixed-size block of data at a time. When the plaintext is longer than the block length, a block cipher is turned into a stream cipher using an algorithm called a *mode of operation*, which describes how to repeatedly call the block cipher until the whole plaintext is encrypted. The most used block cipher is the Advanced Encryption Standard [230] (AES), which replaces the now outdated Data Encryption Standard [229] (DES).

- Hash functions, which do not require a key, and are not reversible. They are typically used to provide data integrity or to irreversibly encrypt passwords. MD5 [264] and SHA-1 [283] are two well-known hash functions.

1.0.1 Secure Implementations

Cryptanalysis focuses on finding weaknesses in cryptographic primitives, either in their algorithms or in their implementations. For instance, the Caesar cipher, presented earlier, is easily broken by trying to shift all letters of the ciphertext by every possible n (between 1 and 25) until it produces a text that makes sense. Examples of more advanced attacks include related-key attack [79, 176, 54], which consists in observing similarities in the ciphertext produced by a cipher for a given plaintext with different keys, and differential cryptanalysis [73, 194], where several plaintexts are encrypted, and the attacker tries to find statistical patterns in the produced ciphertexts. A cipher is considered algorithmically secure if no practical attack exists, that is, no attack can be carried out in a reasonable amount of time or set up at a reasonable cost.

Even when cryptanalysis fails to break a cipher on an algorithmic level, its implementation might be vulnerable to *side-channel attacks*, which rely on physically monitoring a the execution of a cipher, in order to recover secret data. Side-channel attacks exploit physical vulnerabilities of the architecture a cipher is running on: executing a cipher takes time, consumes power, induces memory transfers, *etc.*, all of which could be attack vectors. A typical example is *timing attacks* [185, 95, 58], which exploit variations of the execution time of a cipher due to conditional branches depending on secret data. For instance, consider the following C code that checks if a provided password matches an expected password:

```
int check_password(char* provided, char* expected, int length) {
    for (int i = 0; i < length; i++) {
        if (provided[i] != expected[i]) {
            return 0;
        }
    }
    return 1;
}
```

If `provided` and `expected` start with different characters, then this function will quickly return 0. However, if `provided` and `expected` start with the same 10 characters, then this function will loop ten times (and therefore will take longer) before returning 0, thus informing an attacker monitoring the execution time of this function that they have the first characters right. This vulnerability could be fixed by decorrelating the execution time from secret inputs, making it *constant-time*:

```
int check_password(char* provided, char* expected, int length) {
    int flag = 1;
    for (int i = 0; i < length; i++) {
        if (provided[i] != expected[i]) {
            flag = 0;
        }
    }
    return flag;
}
```

Timing attacks can also be possible in the absence of conditional execution based on secret data: the time needed to read some data from memory on a modern computer depends heavily on whether those data are in cache. An attacker could therefore design a *cache-timing attack*, *i.e.,* a timing attack based on cache accesses pattern.

By monitoring the power consumption of a cipher, an attacker can carry a *power analysis* [186, 213] to gain knowledge of secret data: power consumption can be correlated with the number of transistor switching state, which may itself depend on the value of the secret data [187].

Rather than being passive (*i.e.*, observing the execution without tampering with it), an attacker can be active and inject faults in the computation [75, 32, 20] (using, for example, ionizing radiation, electromagnetic pulses, or lasers). Consider the following C code, which returns some secret data if it is provided with the correct pin code:

```
char* get_secret_data(int pin_code) {
    if (pin_code != expected_pin_code) {
        return NULL;
    }
    return secret_data;
}
```

An attacker could inject a fault during the execution of this code in order to skip the `return NULL` instruction, which would cause this function to return the secret data even when provided with the wrong pin code.

Protecting code against faults requires a deep understanding of both the hardware and existing attacks. For instance, let us consider a function called `lookup`, which takes as input a 2-bit integer `index`, and returns the 2-bit integer at the corresponding index in a private array `table`:

```
int lookup(int index) {
    int table[4] = { 3, 0, 1, 2 };
    return table[index];
}
```

This code is vulnerable to cache-timing attacks (or rather, would be, if executed on a CPU where a cache line is 4 bytes wide), since `table[index]` might hit or miss in the cache depending on the value of `index`. To make `lookup` resilient to such attacks, we can transform it to remove the table and only do constant-time bitwise operations instead:

```
int lookup_ct(int index) {
    // Extracting the index's 2 bits
    bool x0 = (index >> 1) & 1;
    bool x1 = index & 1;

    // Computing the lookup through bitwise operations
    bool r1 = ~x1;
    bool r0 = ~(x0 ^ x1);

    // Recombining the result's bits together
    return (r0 << 1) | r1;
}
```

`lookup_ct` is functionally equivalent to `lookup`. Its code does not perform any memory accesses depending on secret data, and is thus resilient to cache-timing attacks.

However, `lookup_ct` is still vulnerable to power analysis attacks: computing `~x1`, for instance, might consume a different amount of power depending on whether `x1` is `0` or `1`. To thwart power-based attacks, we can use *boolean masking*, which consists in representing each bit b of secret data by n random bits (called *shares*) such that their `xor` is equal to the original secret bit: `b = b1 ^ b2 ^ ... ^ bn` for each secret bit `b`. The idea is that an attacker needs to determine the value of n bits of data in order to know the value of a single secret bit, which increases exponentially (with n) the cost of an attack. Adding this protection to `lookup_ct` would produce the following code (assuming that `index` has already been masked and is therefore now an array of shares):

```
int* lookup_ct_masked(int index[NUMBER_OF_SHARES]) {
    // Extracting the index's 2 bits
    bool x0[NUMBER_OF_SHARES], x1[NUMBER_OF_SHARES];
    for (int i = 0; i < NUMBER_OF_SHARES; i++) {
        x0[i] = (index[i] >> 1) & 1;
        x1[i] = index[i] & 1;
    }

    // Computing the lookup
    bool r0[NUMBER_OF_SHARES], r1[NUMBER_OF_SHARES];
    // r1 = ~x1
    r1[0] = ~x1[0];
    for (int i = 1; i < NUMBER_OF_SHARES; i++) {
        r1[i] = x1[i];
    }
    // r0 = ~(x0 ^ x1)
    r0[0] = ~(x0[0] ^ x1[0]);
    for (int i = 1; i < NUMBER_OF_SHARES; i++) {
        r0[i] = x0[i] ^ x1[i];
    }

    // Recombining the result's bits together
    int result[NUMBER_OF_SHARES];
    for (int i = 0; i < NUMBER_OF_SHARES; i++) {
        result[i] = (r0[i] << 1) | r1[i]
    }
    // (pretending that we can return local arrays in C)
    return result;
}
```

Note that computing a masked `not` only requires negating one of the shares (we arbitrarily chose the first one): negating a bit `b` shared as `b0` and `b1` is indeed `(~b0) ^ b1` rather than `(~b0) ^ (~b1)`. Computing a masked `xor` between two masked values, on the other hand, requires `xor`ing all of their shares: `(a0, a1, ..., an) ^ (b0, b1, ..., bn) = (a0^b0, a1^b1, ..., an^bn)`.

Protecting this code against fault injection could be done by duplicating instructions, which would add yet another layer of complexity. However, one must be careful about the interactions between countermeasures: adding protections against faults could undo some of the protections against power analysis [256, 257, 112]. Conversely, an attacker could combine fault injection and power analysis [19, 267, 135], which needs to be taken into account when designing countermeasures [260, 125].

1.0.2 High-Throughput Implementations

A paramount requirement on cryptographic primitive is high throughput. Ideally, cryptography should be completely transparent from the point of view of the end users. However, the increasing complexity of CPU microarchitectures makes it hard to efficiently implement primitives. For instance, consider the following C code:

```
int a, b, c, d;
for (int i = 0; i < 2000000000; i++) {
  a = a + d;
  b = b + d;
  c = c + d;
}
```

An equivalent x86 assembly implementation is:

```
    movl  $2000000000, %esi
.loop:
    addl  %edi, %r14d
    addl  %edi, %ecx
    addl  %edi, %eax
    addl  %edi, %r14d
    addl  %edi, %ecx
    addl  %edi, %eax
    addq  $-2, %rsi
    jne .loop
```

The main loop of this code contains 7 additions and a jump. Since modern superscalar Intel CPUs can execute 4 additions per cycles, or 3 additions and a jump, one could expect the body of the loop to execute in 2 cycles. However, depending on the alignment of the jump instruction[1] (`jne .loop`), the loop will execute in 3 cycles rather than 2.

In order to achieve the best throughputs possible, cryptographers resort to various programming trick. One such trick, popularized by Matsui [206], is called interleaving. To illustrate its principle, consider for instance the following imaginary block cipher, written in C:

```c
void cipher(int plaintext, int key[2], int* ciphertext) {
    int t1 = plaintext ^ key[0];
    *ciphertext = t1 ^ key[1];
}

void encrypt_blocks(int* input, int key[2], int* output, int length) {
    for (int i = 0; i < length; i++) {
        cipher(input[i], key, &output[i]);
    }
}
```

The `cipher` function contains two instructions. Despite the fact that modern CPUs are superscalar and can execute several instructions per cycles, the second instruction (`t1 ^ key[1]`) uses the result of the first one (`t1 = plaintext ^ key[0]`) and thus cannot be computed in the same cycle. The execution time of `encrypt_blocks` can thus be expected to be `length` $*$ 2 cycles. Matsui's trick consists in unrolling once the main loop and interleaving two independent instances of `cipher`:

```c
void cipher_inter(int plaintext[2], int key[2], int ciphertext[2]) {
    int t1_0 = plaintext[0] ^ key[0];
    int t1_1 = plaintext[1] ^ key[0];
    ciphertext[0] = t1_0 ^ key[1];
    ciphertext[1] = t1_1 ^ key[1];
}

void encrypt_blocks_inter(int* input, int key[2], int* output, int length) {
    for (int i = 0; i < length; i+=2) {
        cipher_inter(&input[i], key, &output[i]);
    }
}
```

This new code is functionally equivalent to the previous one. `cipher_inter` computes the ciphertexts of 2 plaintexts at the same time, and the main loop of `encrypt_blocks_inter` thus performs twice less iterations. However, since the first two (resp., last two) instructions of `cipher_inter` have no data dependencies between them, they can be executed

[1] See https://stackoverflow.com/q/59883527/4990392

during the same cycle. `cipher_inter` thus takes as many cycles as `cycle` to be executed, but computes two ciphertexts instead of one. Overall, `encrypt_blocks_inter` is thus twice faster than `encrypt_blocks`.

1.0.3 The Case for Usuba

Implementing high-throughput cryptographic primitives, or securing primitives against side-channel attacks are complicated and tedious tasks. Both are hard to get right and tend to obfuscate the code, thus hindering code maintenance. Trying to achieve both at the same time, performance and side-channel protection, is a formidable task.

Instead, we propose Usuba [212, 211], a domain-specific programming language designed to write symmetric cryptographic primitives. Usuba is a high-level programming language, enjoying a straightforward formal semantics, allowing to easily reason on program correctness. Usuba programs are constant-time by construction, and thus protected against timing attacks. Furthermore, Usuba can automatically insert countermeasures, such as Boolean masking, to protect against power-based side-channels. Finally, Usuba compiles to high-performance C code, exploiting SIMD extensions of modern CPUs when available (SSE, AVX, AVX512 on Intel), thus performing on par with hand-tuned implementations.

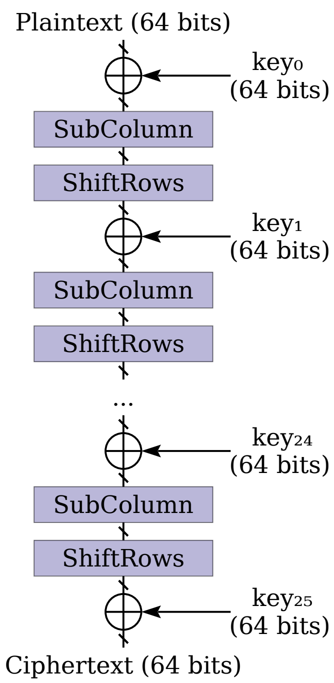
The design of Usuba is largely driven by the structure of block ciphers. A block cipher typically uses bit-permutations, bitwise operations, and sometimes arithmetic operations, and can therefore be seen as a stateless circuit. These basic operations are combined to form a *round*, and a block cipher is defined as *n* (identical) rounds, each of them taking the output of the previous round as well as a key as input. For instance, the RECTANGLE [309] block cipher takes as input a 64-bit plaintext and 25 64-bit keys, and produces the ciphertext through 24 rounds, each doing a `xor`, and calling two auxiliary functions: `SubColumn` (a lookup table), and `ShiftRows` (a permutation). Figure 1.1a represents RECTANGLE using a circuit.

Usuba aims at providing a way to write an implementation of a cipher which is as close to the specification (*i.e.,* the circuit) as possible. As such, RECTANGLE can be straightforwardly written in Usuba in just a few lines of code, as shown in Figure 1.1b. The code should be self-explanatory: the main function `Rectangle` takes as input the plaintext as a tuple of 4 16-bit elements, and the key as a 2D vector, and computes 25 rounds, each calling the functions `ShiftRows`, described as 3 left-rotations, and `SubColumn`, which computes a lookup in a table. This code is simple, and as close to the specification as can be. Yet, it compiles to a C code which is about 20% faster than the reference implementation (Section 5.1), while being much simpler and more portable: whereas the reference implementation explicitly uses SSE and AVX SIMD extensions, Usuba is not bound to a specific SIMD extension.

One of the other main benefits of Usuba is that the code it generates is constant-time *by construction*. To write constant-time code with traditional languages (*e.g.,* C) is to fight an uphill battle against compilers [27], which may silently rewrite one's carefully crafted program into one vulnerable to timing attacks, and against the underlying architecture itself [216], whose undocumented, proprietary micro-architectural features may leak secrets through timing or otherwise. For instance, the assembly generated by Clang 9.0 for the following C implementation of a multiplexer:

```
bool mux(bool x, bool y, bool z) {
    return (x & y) | (~x & z);
}
```

uses a `cmove` instruction, which is not specified to be constant-time [259]. Likewise, some integer multiplication instructions are known not to be constant-time, causing library



(a) RECTANGLE as a circuit

```
table SubColumn (in:v4) returns (out:v4) {
    6 , 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3 , 13, 8, 15, 4, 2
}

node ShiftRows (input:u16[4]) returns (out:u16[4])
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel

node Rectangle (plain:u16[4],key:u16[26][4])
       returns (cipher:u16[4])
vars state : u16[4]
let
    state = plain;
    forall i in [0,24] {
        state := ShiftRows(SubColumn(state ^ key[i]))
    }
    cipher = state ^ key[25]
tel
```

(b) RECTANGLE written in Usuba

Figure 1.1: The RECTANGLE cipher

developers to write their own software-level constant-time implementations of multiplication [247]. The issue is so far-reaching that tools traditionally applied to hardware evaluation are now used to analyze software implementations [259], treating the program and its execution environment as a single black-box and measuring whether its execution time is constant with a high enough probability. Most modern programming languages designed for cryptography have built-in mechanism to prevent non-constant-time operations. For instance, HACL* [310] has the notion of *secure integers* that cannot be branched on, and forbids the use of non-constant-time operations like division or modulo. FaCT [99], on the other hand, takes the stance that HACL* is too low-level, and that constant-timeness should be seen as a compilation problem: it provides high-level abstractions that are compiled down to constant-time idioms. Adopting yet another high-level approach, Usuba enforces constant-time by adopting (in a transparent manner from the developer's perspective) a programming model called *bitslicing*.

Bitslicing was first introduced by Biham [72] as an implementation trick to speed up software implementations of DES. Intuitively, the idea of bitslicing is to represent a n-bit value as 1 bit in n registers. In the case of 64-bit registers, each register therefore has 63-bit empty bit remaining, which can be filled in the same fashion by other independent values. To manipulate such data, the cipher must be reduced to bitwise logical operations (`and`, `or`, `xor`, `not`). On a 64-bit machine, a bitwise operation then effectively works like 64 parallel 1-bit operations. Throughput is thus achieved by parallelism: 64 instances of the cipher are computed in parallel. Consequently, bitslicing is especially good at exploiting vector extensions of modern CPUs, which offer large registers (*e.g.,* 128-bit SSE, 256-bit AVX and 512-bit AVX-512 on Intel). Bitsliced implementations are constant-time by design: no data-dependent conditionals nor memory accesses are made (or, in fact, possible at all). Many record-breaking software implementations of block ciphers exploit this technique [189, 207, 174, 33], and modern ciphers are now designed from the ground up with bitslicing in mind [78, 309]. However, bitslicing implies an increase in code complexity, making it hard to write efficient bitsliced code by hand, as can be demonstrated by the following few lines of C code that are part of a DES implementation written by Matthew Kwan [192]:

```
s1 (r31 ^ k[47], r0 ^ k[11], r1 ^ k[26], r2 ^ k[3], r3 ^ k[13],
  r4 ^ k[41], &l8, &l16, &l22, &l30);
s2 (r3 ^ k[27], r4 ^ k[6], r5 ^ k[54], r6 ^ k[48], r7 ^ k[39],
  r8 ^ k[19], &l12, &l27, &l1, &l17);
s3 (r7 ^ k[53], r8 ^ k[25], r9 ^ k[33], r10 ^ k[34], r11 ^ k[17],
  r12 ^ k[5], &l23, &l15, &l29, &l5);
s4 (r11 ^ k[4], r12 ^ k[55], r13 ^ k[24], r14 ^ k[32], r15 ^ k[40],
  r16 ^ k[20], &l25, &l19, &l9, &l0);
```

The full code goes on like this for almost 300 lines, while the Usuba equivalent is just a few lines of code, very similar to the RECTANGLE code shown in Figure 1.1b. The simplicity offered by Usuba does not come at any performance cost: both Kwan's and Usuba's implementations exhibit similar throughput.

The bitslicing model can sometimes be too restrictive as it forbids the use of arithmetic operations, and may fail to deliver optimal throughputs as it consumes a lot of registers. To overcome these issues, and drawing inspiration from Käsper & Schwabe's byte-sliced AES [174], we propose a generalization of bitslicing that we dub *m*slicing. *m*slicing preserves the constant-time property of bitslicing, while using less registers, and allowing to use SIMD packed arithmetic instructions (*e.g.,* `vpaddb`, `vmuldp`), as well as vector permutations (*e.g.,* `vpshufb`).

1.1 Contributions

We made the following contributions in this thesis:

- We designed Usuba, a domain-specific language for cryptography (Chapter 2). Usuba enables a high-level description of symmetric ciphers by providing abstractions tailored for cryptography, while generating high-throughput code thanks to its slicing model. We demonstrated the expressiveness and versatility of Usuba on 17 ciphers (Section 2.4). Furthermore, we formalized Usuba's semantics (Chapter 3), thus enabling formal reasoning on the language, and paving the way towards a verified compiler for Usuba.

- We developed Usubac, an optimizing compiler translating Usuba to C (Chapter 4). It involves a combination of folklore techniques—such as inlining and unrolling—tailored to the unique problems posed by sliced programs (Section 4.2) and introduces new techniques—such as interleaving (Section 4.2.6) and sliced scheduling (Section 4.2.5)—made possible by our programming model. We showed that on high-end Intel CPUs, Usuba exhibits similar throughputs as hand-tuned implementations (Chapter 5).

- We integrated side-channel countermeasures in Usuba, in order to generate side-channel resilient code for embedded devices. In particular, by leveraging recent progress in provable security [53], we are able to generate implementations that are provably secure against *probing* side-channel attacks (Chapter 6). We also implemented a backend for SKIVA, thus producing code resilient to both power-based side-channel attacks and fault injections (Chapter 7).

1.2 Background

Usuba has two main targets: high-end CPUs, for which it generates optimized code, and embedded CPUs, for which it generates side-channel resilient code. Several optimizations are specifically tailored for Intel superscalar CPUs (Section 4.2), and our main performance evaluation compares Usuba-generated ciphers against reference implementation on Intel CPUs (Chapter 5). In Section 1.2.1, we introduce the micro-architectural notions necessary to understand this platform, as well as our benchmarking methodology.

Section 1.2.2 and 1.2.3 present bitslicing and *m*slicing as data formats and their impact on code expressiveness, performance, and compilation.

Finally, Section 1.2.5 compares the throughputs of performing a bitsliced and a *m*sliced addition. We provide a detailed analysis of the performance of both implementations, which can be seen as a concrete example of the notions introduced in Section 1.2.1.

1.2.1 Skylake CPU

At the time of writing, the most recent Intel CPUs (Skylake, Kaby Lake, Coffee Lake) are derived from the Skylake microarchitecture. The Skylake CPU (Figure 1.2) is a deeply pipelined microarchitecture (*i.e.,* it can contain many instructions at the same time, all going through different execution phases). This pipeline consists of 2 main phases: the *frontend* retrieves and decodes instructions in-order from the L1 instruction cache, while the *out-of-order execution engine* actually executes the instructions. The instructions are finally removed in-order from the pipeline by the retiring unit.

Note that this is a simplified view of the Skylake microarchitecture, whose purpose is only to explain what matters to us, and to show which parts we will be focusing on when designing our optimizations, and when analyzing the performance of our programs.

Frontend

The L1 instruction cache contains x86 instructions represented as a sequence of bytes. Those instructions are decoded by the Legacy Decode Pipeline (MITE). The MITE operates in the following way:

- up to 16 bytes of instructions are fetched from the L1 instruction cache, and predecoded into macro-ops.
- up to 6 macro-ops per cycle are delivered to the instruction queue (IQ), which performs macro-fusion: some common patterns are identified and optimized by fusing macro-ops together. For instance, an increment followed by a conditional jump, often found at the end of a loop, may fuse together in a single macro-op.
- the IQ delivers up to 5 macro-ops per cycle to the decoders. The decoders convert each macro-ops into one or several μops, which are then sent to the Instruction Decode Queue (IDQ).

The MITE is limited by the fetcher to 16 bytes of instructions per cycle. This translates to 4 or 5 μops per cycle on programs manipulating integer registers, which is enough to maximize the bandwidth of the pre-decoder and decoders. However, SSE, AVX and AVX-512 instructions are often larger. For instance, an addition between two registers is encoded on 2 bytes for integer registers, 4 bytes on SSE and AVX, and 6 bytes on AVX-512. Therefore, on programs using SIMD extensions, the MITE tends to be limited by the fetcher. In order to overcome this, the Decoded Stream Buffer (DSB), a μop cache, can be used to bypass the whole MITE when dealing with sequences of instructions that have

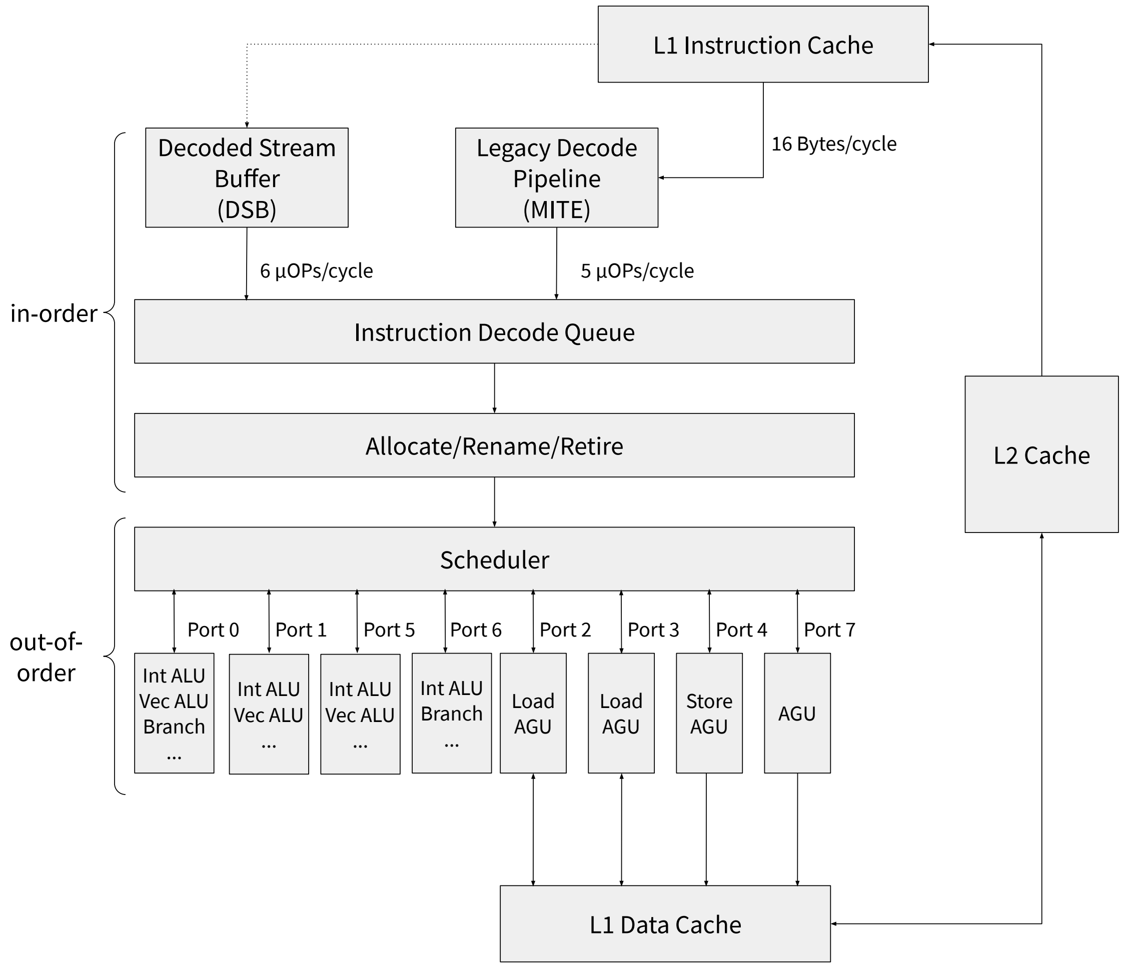


Figure 1.2: Skylake's pipeline

already been decoded (for instance, in a tight loop). The DSB delivers up to 6 μops per cycles, directly to the IDQ.

Execution Engine

The execution engine can be divided in 3 phases. The Allocate/Rename phase retrieves μops from the IDQ and sends them to the Scheduler, which dispatches μops to the execution core. Once μops are executed, they are retired: all resources allocated to them are freed, and the μops are effectively removed from the pipeline.

While assembly code can only manipulate 16 general purpose (GP) registers (*e.g.*, `rax`, `rdx`) and 16 SIMD registers (*e.g.*, `xmm0`, `xmm1`), the CPU has hundreds of registers available. The renamer takes care of renaming *architectural* registers (*i.e.*, registers manipulated by the assembly) into *micro-architectural* registers (the internal registers of the CPU, also known as *physical* registers). The renamer also determines the possible execution ports for each instruction, and allocates any additional resources they may need (*e.g.*, buffers for `load` and `store` instructions).

The execution core consists of several execution units, each able to execute some specific type of μops, accessed through 8 ports. For instance:

- Arithmetic and bitwise instructions can execute on ports 0, 1 and 5 (and port 6 for general-purpose registers).
- Branches can execute on ports 0 and 6.

- Memory loads can execute on ports 2 and 3.
- Memory reads can execute on port 4.

The scheduler dispatches μops out-of-order to the execution units of the execution core. When an instruction could be executed on several execution units, the scheduler chooses one (the algorithm making this choice is not specified).

The instructions are then removed from the pipeline in-order by the retiring unit. All resources allocated for them are freed, faults and exceptions are handled at that stage.

SIMD

SIMD (*single instruction, multiple data*) are CPU extensions that offer registers and instructions to compute the same operations on multiple data at once. Intel provides 4 main classes of SIMD extensions: MMX with 64-bit registers, SSE with 128-bit registers, AVX with 256-bit registers and AVX-512 with 512-bit registers. For the purposes of bitslicing, MMX offers little to no benefits compared to general purpose 64-bit registers, and we shall ignore them.

SSE, AVX and AVX-512 provide instructions to pack several 8-bit, 16-bit, 32-bit and 64-bit words inside a single register of 128 bits, 256 bits or 512 bits, thus producing a *vector*. Figure 1.3a illustrates the function `_mm256_set_epi64x`, which packs 4 64-bit integers in a 256-bit AVX register. The term *packed elements* is used to refer to the individual 64-bit integers in the AVX registers.

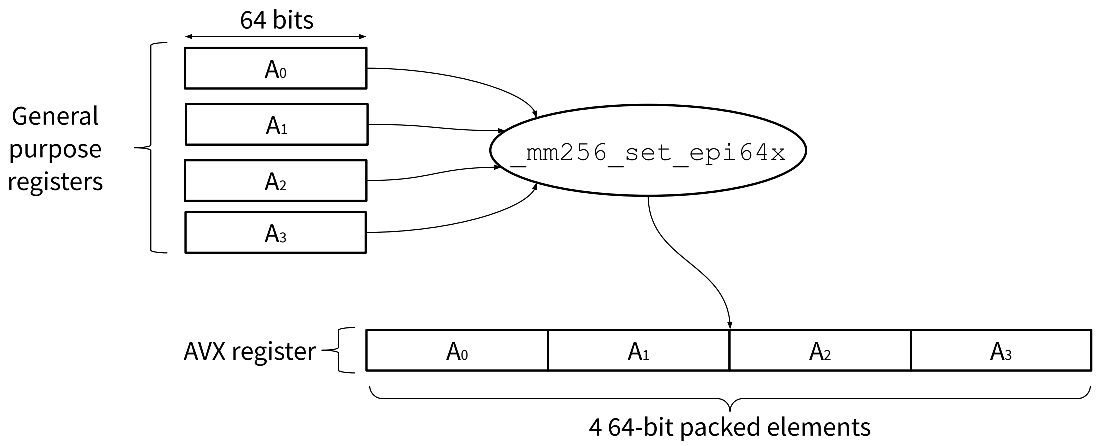
SIMD extensions then provide *packed instructions* to compute over vectors, which we divide in two categories. *vertical* m-bit instructions perform the same instruction over multiple packed elements in parallel. If we visually represent SIMD registers as aggregations of m-bit elements vertically stacked, vertical operations consist in element-wise computations along this vertical direction. For instance, the instruction `vpaddb` (Figure 1.3b) takes two 256-bit AVX registers as parameters, each containing 32 8-bit words, and computes 32 additions between them in a single CPU cycle. Similarly, the instruction `pslld` (Figure 1.3c) takes a 128-bit SSE register containing 4 32-bit words, and an 8-bit integer, and computes 4 left shifts in a single CPU cycle.

On the other hand, *horizontal* SIMD operations perform element-wise computations within a single register, *i.e.,* along the horizontal direction. For instance, the instruction `pshufd` (Figure 1.3d) permutes the 4 32-bit packed elements of a SSE register (according to a pattern specified by an 8-bit immediate).

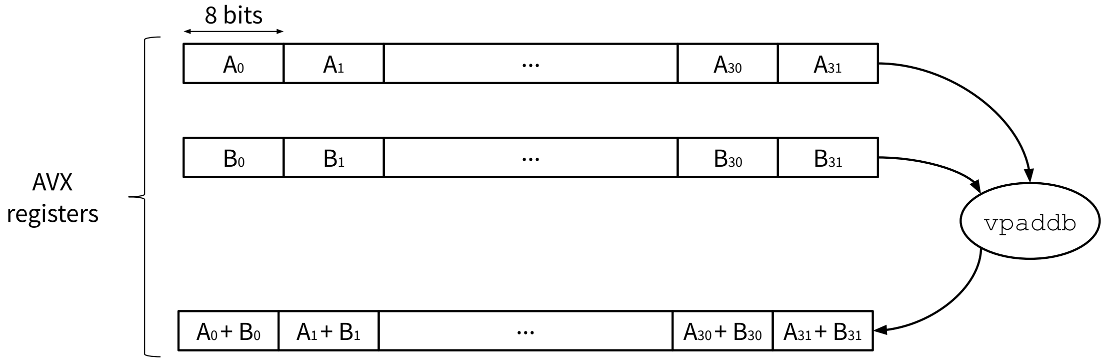
Intel provides *intrinsic* functions that allow C code to use SIMD instructions. For instance, the `_mm256_add_epi8` intrinsic is compile into a `vpaddb` assembly instruction, and `_mm_sll_epi32` is compiled into a `pslld` assembly instruction.

One straightforward application of SIMD extensions is to *vectorize* programs, that is, to transform a program manipulating scalars into a functionally equivalent but faster program manipulating vectors. For instance, consider the following C function, which computes 512 32-bit additions between two arrays:

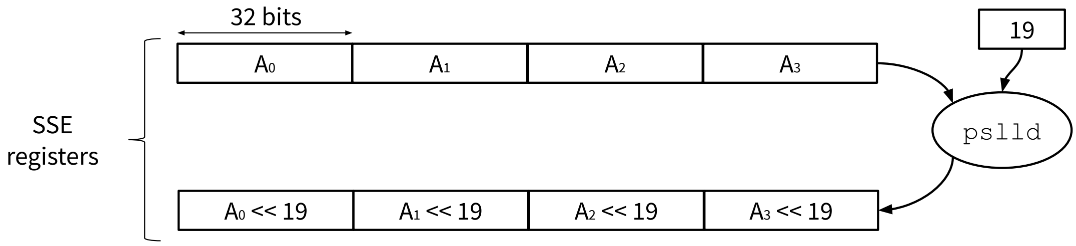
```
void add_512_integers(int* a, int* b, int* c) {
  for (int i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
  }
}
```



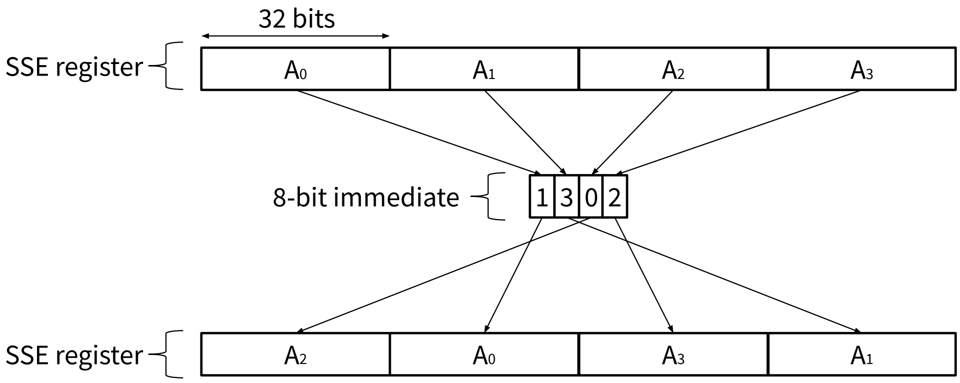
(a) AVX intrinsic `_mm256_set_epi64x`



(b) AVX instruction `vpaddb`



(c) SSE instruction `pslld`



(d) SSE instruction `pshufd`

Figure 1.3: Some SIMD instructions and intrinsics

This function can be transformed into the following equivalent one:

```
void add_512_integers_vec(__m256i* a, __m256i* b, __m256i* c) {
  for (int i = 0; i < 64; i++) {
    c[i] = _mm256_add_epi32(a[i],b[i]);
  }
}
```

The second one (`add_512_integers_vec`) manipulates 256-bit AVX registers (of type `__m256i`), and uses the `_mm256_add_epi32` intrinsic to perform 8 32-bit additions in parallel. The second one should divide by 8 the number of cycles required to perform 512 additions. Most C compilers (*e.g.*, GCC, Clang, ICC) automatically try to vectorize loops to improve performance. Some code cannot be vectorized however. For instance, consider the following snippet:

```
void all_fibonacci(int* res, int n) {
  res[0] = res[1] = 1;
  for (int i = 2; i < n; i++) {
    res[i] = res[i-1] + res[i-2];
  }
}
```

Since each iteration depends on the number calculated at the previous iteration, this code cannot be computed in parallel. Similarly, table lookups cannot be vectorized. Consider, for instance:

```
void n_lookups(int* table, int* indices, int* res, int n) {
  for (int i = 0; i < n; i++) {
    res[i] = table[indices[i]];
  }
}
```

Vectorizing this code would require a SIMD instruction to perform multiple memory lookups in parallel, but no such instruction exists.

As shown in Figure 1.2, ports 0, 1, 5 and 6 of the CPU can compute bitwise and arithmetic instructions, but only ports 0, 1 and 5 can compute SIMD instructions ("Vec ALU"). This limits the potential speedup offered by SIMD extensions: `add_512_integers_vec` should execute in $64/3 = 22$ cycles, whereas `add_512_integers` should execute in $512/4 = 128$ cycles, or 5.8 times more. AVX-512 restrict CPU usage even further: only 2 bitwise/arithmetic AVX-512 instructions can be executed per cycle.

New generations of SIMD offer more than simply twice larger registers. For instance, one of the additions of AVX extensions is 3-operand non-destructive instructions. SSE bitwise and arithmetic instructions take two registers as arguments, and override one of them with the result (*i.e.*, a SSE `xor` will compute `x ^= y`). On the other hand, AVX instructions set a third register with their output (*e.g.*, an AVX `xor` will compute `x = y ^ z`). Another example is AVX512 extensions, which provide 32 registers, whereas AVX and SSE extensions only provide 16.

Generations of SIMD. Several generations of SSE extensions have been released by Intel (SSE, SSE2, SSSE3, SSE4.1, SSE4.2), each introducing new instructions. Similarly, the AVX2 extensions improve upon AVX by providing many useful operations, including 8/16/32-bit additions and shifts. Whenever we mention SSE (resp., AVX) without specifying which version, we refer to SSE4.2 (resp., AVX2).

Micro-Benchmarks

Intel gives access to a Time Stamp Counter (TSC) through the `rdtscp` instruction. This counter is incremented at a fixed frequency, regardless of the frequency of the CPU core clock (and, in particular, is not affected by Turbo Boost, which increases the frequency of the core clock). On both Intel CPUs we used in this thesis (i5-6500 and Xeon W-2155), the TSC frequency is very close to the core frequency: on the i5-6500, the TSC frequency is 3192MHz and the core frequency is 3200Mhz, and on the W-2155, the TSC frequency is 3299MHz and the core frequency is 3300MHz. We can thus use this counter to approximate the number of clock cycles taken by a given code to execute. In order to keep the TSC frequency and core frequency correlated, we disabled Turbo Boost.

Figure 1.4a provides the C benchmark code we used throughout this thesis to benchmark some arbitrary function `run_bench`. Some components of the CPU go through a warm-up phase during which they are not operating at peak efficiency. For instance, according to Fog [138], parts of the SIMD execution units are turned off when unused. As a result, the first instruction using AVX registers takes between 150 and 200 cycles to execute, and the following instructions using AVX registers are 4.5 times slower than normal for about 56.000 clock cycles. 2.7 million clock cycles after the last AVX instruction, parts of the SIMD execution units will be turned back off. In order to prevent those factors from impacting our benchmarks, we include a warm-up phase, running the target code without recording its run time. In Figure 1.4a, the warm-up loop performs 100.000 iterations, which ensures that SIMD extensions are fully powered.

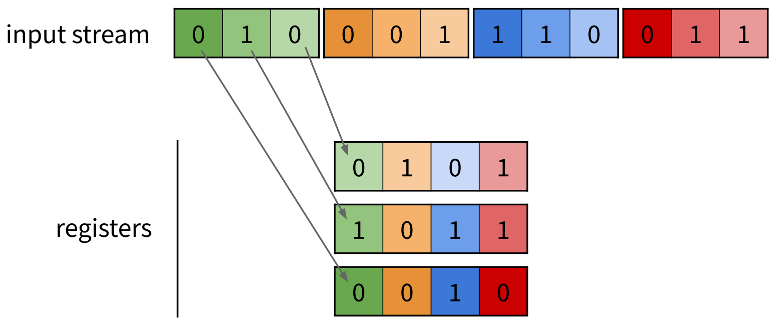
```
extern void run_bench();

int main() {
    // Warm-up
    for (int i = 0; i < 100000; i++) {
        run_bench();
    }

    // Actual benchmark
    unsigned int unused;
    uint64_t timer = __rdtscp(&unused);
    for (int i = 0; i < 1000000; i++) {
        run_bench();
    }
    timer = __rdtscp(&unused) - timer;

    // Outputting result
    printf("%.2f␣cycles/iteration\n", timer/1000000);
}
```

(a) Core of our generic benchmark code (C)

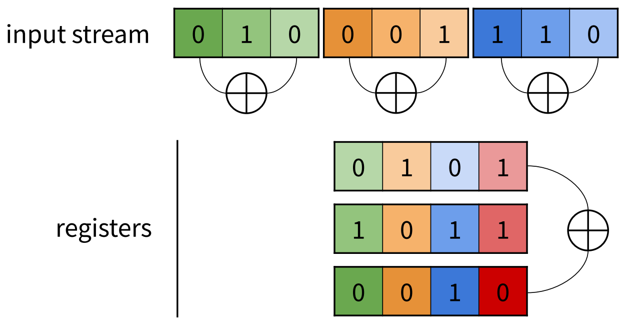
```
for my $i (0 .. 29) {
    $times_a[$i] = system "./bench_a";
    $times_b[$i] = system "./bench_b";
}
printf "bench_a:␣%.2f␣␣//␣bench_b:␣%.2f\n",
       average(@times_a), average(@times_b);
```

(b) Wrapper for our benchmarks (Perl)

Figure 1.4: Our benchmark code



(a) Bitslicing layout



(b) Computing on bitsliced data

Figure 1.5: Bitslicing example with 3-bit data and 4-bit registers

While the code from Figure 1.4a provides a fairly accurate cycle count for a given function, other programs running on the same machine may impact its run-time. In order to alleviate this issue, we repeated each measurement 30 times, and took the average of the measurements (Figure 1.4b).

To compare performance results, we use the Wilcoxon rank-sum test [300] (also called Mann-Whitney U test), which assesses whether two distributions are significantly distinct. Every speedup and slowdown mentioned in this thesis are thus statistically significant. For instance, Table A.4 (Page 196) shows some $\times 1.01$ speedups and $\times 0.99$ slowdowns, all of which are significant.

Intel provides hundreds of counters to monitor the execution of a program. For instance, the number of read and writes to the caches, the number of cache misses and hits, the number of cycles where 1, 2, 3 or 4 μops were executed, the number of cycles where the frontend did not dispatch μops, the number of instructions executed each cycles (IPC). To collect such counters, we use the `perf` utility [4], and `vtune` software [164].

1.2.2 Bitslicing

The general idea of bitslicing is to transpose m n-bit data into n m-bit registers (or variables). Then, standard m-bit bitwise operations of the CPU can be used and each acts as m parallel operation. Therefore, if a cipher can be expressed as a combination of bitwise operations, it can be ran m times in parallel using bitslicing. A bitsliced program can thus be seen as a combinational circuit (*i.e.*, a composition of logical operations) implemented in software.

Figure 1.5a illustrates bitslicing on 3-bit inputs, using 4-bit registers (for simplicity). 3 registers are required to bitslice 3-bit data: the first bit of each input goes into the first register; the second bit into the second register and the third bit into the third register. Once the data has this representation, doing a `xor` (or any other bitwise operation) between two registers actually computes 4 independent `xor`s (Figure 1.5b).

Scaling

SIMD instructions, presented in Section 1.2.1, offer large registers (up to 512 bits) and operations to perform data-parallel computation on those registers. Bitslicing being parallel by construction, it is always possible to make use of SIMD instructions sets to increase the throughput of a bitsliced program. As for general-purpose registers, a bitsliced program will only require SIMD bitwise instructions, such as `vpandd` that computes 512 bitwise 1-bit `and`s in parallel (Figure 1.6).

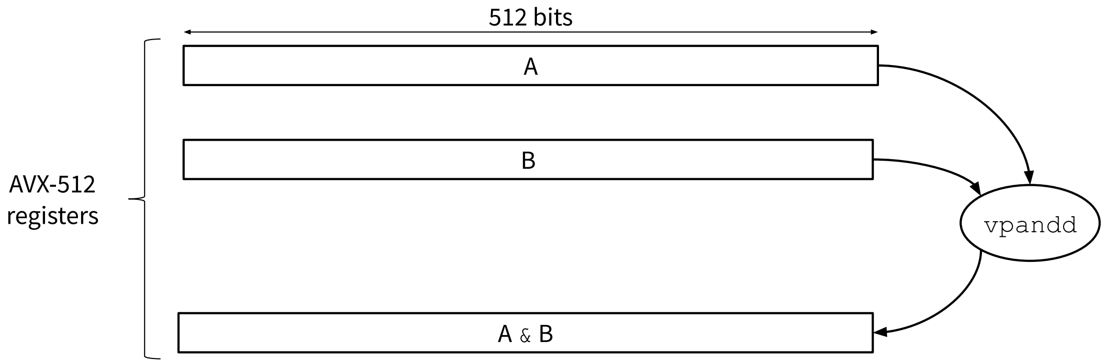


Figure 1.6: The AVX-512 `vpandd` instruction

Executing a bitsliced program on 512-bit registers will compute the same circuit on 512 independent data in parallel, thus dividing by 8 the number of cycles compared to 64-bit registers. On the other hand, the overall time needed to execute the circuit, and thus the latency, remains constant no matter the registers used: encrypting 64 inputs in parallel on 64-bit registers, or encrypting 512 inputs in parallel on 512-bit registers will take roughly the same amount of time.

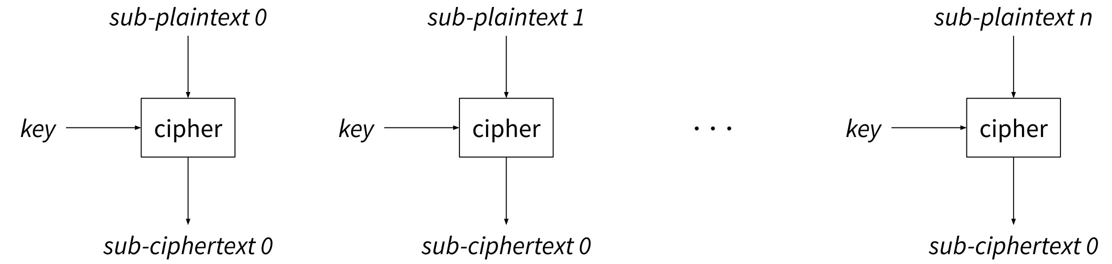
Finally, to make full use of SIMD extensions, hundreds of inputs must be available to be encrypted in parallel. For instance, on AES, which encrypts a 128-bit plaintext, 8 KB of data are required in order to fill 512-bit AVX-512 registers.

Modes of Operation

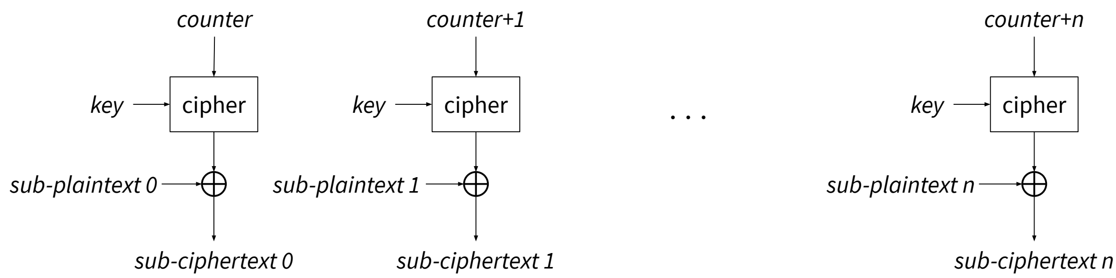
A block cipher can only encrypt a fixed amount of data (a block). The size of a block is generally between 64 and 128 bits (*e.g.*, 64 bits for DES and 128 bits for AES). When the plaintext is longer than the block size, the cipher must be repeatedly called until the whole plaintext is encrypted, using an algorithm called a *mode of operation*. The simplest mode of operation is Electronic Codebook (ECB, Figure 1.7a). It consists in dividing the plaintext into sub-plaintexts (of the size of a block), encrypting them separately, and concatenating the resulting sub-ciphertext to produce the full ciphertext.

This mode of operation is considered insecure because identical blocks will be encrypted into the same ciphertext. This can be exploited by an attacker to gain knowledge about the plaintext. Figure 1.8 illustrates the weaknesses of ECB: the initial image (Figure 1.8a) encrypted using AES in ECB mode results in Figure 1.8b, which clearly reveals information about the initial image.

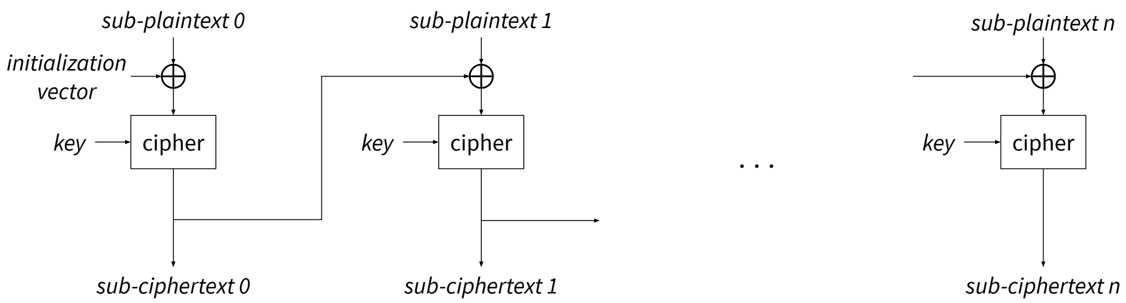
Counter mode (CTR) is a more secure parallel mode that works by encrypting a counter rather than the sub-plaintexts directly (Figure 1.7b). It then `xor`s the encrypted counter with the sub-plaintext. Encrypting the image from Figure 1.8a with this mode produces a seemingly random image (Figure 1.8c). Incrementing the counter can be done in parallel using SIMD instructions:



(a) Electronic codebook (ECB) mode



(b) Counter (CTR) mode



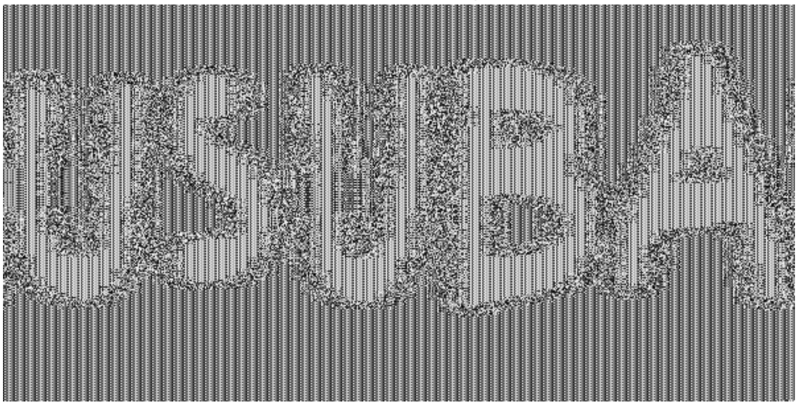
(c) Cipher block chaining (CBC) mode

Mode	**Encryption**	**Decryption**	**Secure**
ECB	parallel	parallel	✗
CTR	parallel	parallel	✓
CBC	sequential	parallel	✓

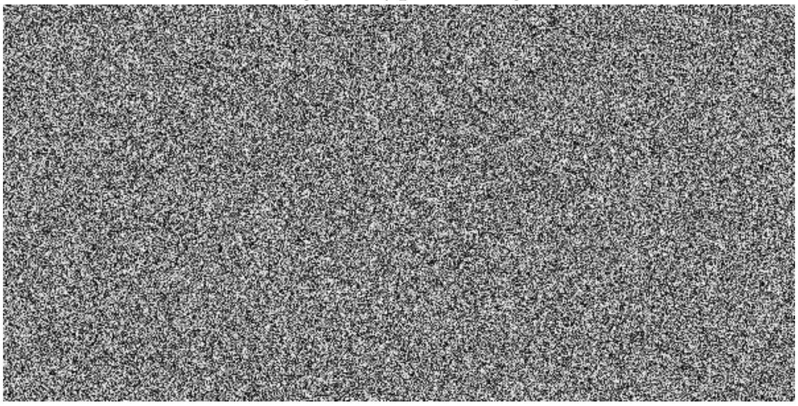
(d) Qualitative comparison of ECB, CBC and CTR

Figure 1.7: Some common modes of operation

(a) Source image



(b) Image encrypted using ECB



(c) Image encrypted using CTR

! The sources of this experiment are available at:
`https://github.com/DadaIsCrazy/usuba/tree/master/experimentations/ecb_vs_ctr`

Figure 1.8: ECB *vs.* CTR

```
// Load 4 times the initial counter in a 128-bit SSE register
__m128i counters   = _mm_set1_epi32(counter);
// Load a SSE register with integers from 1 to 4
__m128i increments = _mm_set_epi32(1, 2, 3, 4);
// Increment each element of the counters register in parallel
counters = _mm_add_epi32(counters, increments);
// |counters| can now be transposed and encrypted in parallel
```

CTR can thus be fully parallelized and therefore allows bitslicing to maximize parallelism and register usage.

Another commonly used mode is Cipher Block Chaining (CBC, Figure 1.7c), which solves the weakness of ECB by `xor`ing each sub-plaintext with the sub-ciphertext produced by the encryption of the previous sub-plaintext. This processed is bootstrapped by `xor`ing the first sub-plaintext with an additional secret data called an *initialization vector*.

However, because bitslicing encrypts many sub-plaintexts in parallel, it prevents the use of CBC, as well as any other mode that would rely on using a sub-ciphertext as an input for encrypting the next sub-plaintext (like Cipher Feedback and Output Feedback). To take advantage of bitslicing in such sequential modes, we can multiplex encryptions from multiple independent data streams, at the cost of some management overheads [148].

Compile-time Permutations

A desirable property of ciphers is *diffusion*: if two plaintexts differ by one bit, then statistically half of the bits of the corresponding ciphertexts should differ. In practice, this property is often achieved using permutations. For instance, Piccolo [277] uses an 8-bit permutation (Figure 1.9). A naive, non-bitsliced C implementation of this permutation would be:

```
char permut(char x) {
    return ((x & 128) >> 6) |
           ((x & 64)  >> 2) |
           ((x & 32)  << 2) |
           ((x & 16)  >> 2) |
           ((x & 8)   << 2) |
           ((x & 4)   >> 2) |
           ((x & 2)   << 2) |
           ((x & 1)   << 6);
}
```

A clever developer (or compiler) could notice that the three right-shifts by 2 can be merged together: `((x & 64) >> 2) | ((x & 16) >> 2) | ((x & 4) >> 2)` can be optimized to `(x & (64 | 16 | 4)) >> 2`. The same goes for the three left-shifts by 2, and the permutation can therefore be written as (with the masks written in binary for more simplicity):

```
char permut(char x) {
    return ((x & 0b10000000) >> 6) |
           ((x & 0b01010100) >> 2) |
           ((x & 0b00101010) << 2) |
           ((x & 0b00000001) << 6);
}
```

In bitslicing, each bit is stored in a different variable, so this permutation consists in 8 static assignments:

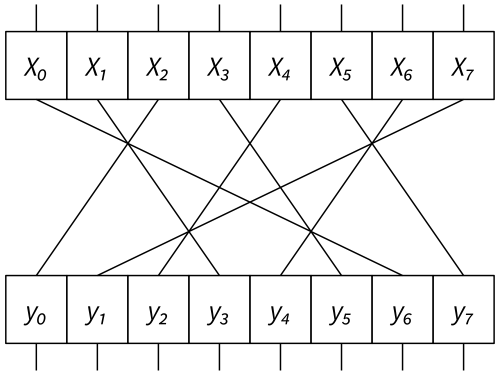


Figure 1.9: Piccolo's round permutation

```
void permut(bool x0, bool x1, bool x2, bool x3,
            bool x4, bool x5, bool x6, bool x7,
            bool* y0, bool* y1, bool* y2, bool* y3,
            bool* y4, bool* y5, bool* y6, bool* y7) {
    *y0 = x2;
    *y1 = x7;
    *y2 = x4;
    *y3 = x1;
    *y4 = x6;
    *y5 = x3;
    *y6 = x0;
    *y7 = x5;
}
```

The C compiler can inline this function, and get rid of the assignments by doing copy propagation, effectively performing the permutation at compile time. This technique can be applied to any bit-permutation, as well as shifts and rotations, thus reducing their runtime cost to zero.

Lookup Tables

Another important property of ciphers is *confusion*: each bit of a ciphertext should depend on several bits of the encryption key, so as to obscure the relationship between the two. Most ciphers achieve this property using functions called *S-boxes* (for substitution-boxes), often specified using lookup tables. For instance, RECTANGLE uses the following lookup table:

```
char table[16] = {
    6 , 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3 , 13, 8, 15, 4, 2
};
```

This lookup table is said to be a 4×4 table: it needs **4** bits to index its 16 elements, and returns integers on **4** bits (0 to 15). Such tables cannot be used in bitslicing, since each bit of the index would be in a different register. Instead, equivalent circuits can be used, as illustrated in Section 1.0.1. A circuit equivalent to the lookup table of RECTANGLE is:

```
void table(bool  a0, bool  a1, bool  a2, bool  a3,
           bool* b0, bool* b1, bool* b2, bool* b3) {
    bool t1  = ~a1;
    bool t2  = a0 & t1;
    bool t3  = a2 ^ a3;
    *b0      = t2 ^ t3;
    bool t5  = a3 | t1;
    bool t6  = a0 ^ t5;
    *b1      = a2 ^ t6;
    bool t8  = a1 ^ a2;
    bool t9  = t3 & t6;
    *b3      = t8 ^ t9;
    bool t11 = *b0 | t8;
    *b2      = t6 ^ t11;
}
```

Using 64-bit variables (`uint64_t` in C) instead of `bool` allows this code to compute the S-box 64 times in parallel. Bitslicing then becomes more efficient than direct code: accessing to a value in the original table will take about 1 cycle (assuming a cache hit), while doing the 12 instructions from the circuit above should take 12 cycles (or even less on a superscalar CPU) to compute 64 times the S-box (on 64-bit registers), thus costing at most 0.19 cycles per S-box.

Converting a lookup table into a circuit can be easily done using Karnaugh maps [172] or binary decision diagrams [197]. However, this tends to produce large circuits. Brute-forcing every possibility is unlikely to yield any results, as even a small 4×4 S-box usually requires a circuit of about 12 instructions, and hundreds of billions of such circuits exist. Heuristics can be added to the brute-force search in order to reduce the complexity of the search [234], but this does not scale well beyond 4×4 S-boxes. For large S-boxes, like the 8×8 AES S-box, cryptographers exploit the underlying mathematical structure of the S-boxes to optimize them [96, 91]. Such a task is hard to automate, yet it is essential to obtain good performance on bitsliced ciphers.

Constant-time

Bitslicing makes it impossible to (efficiently) branch on secret data, since within a single register, each bit represents a different input. In particular, the branch condition could be, at the same time, true for some inputs, and false for others. Thus, both branches of conditionals need to be computed, and combined by masking with the branch condition. For instance, the following branching code (assuming `x`, `a`, `b`, `c`, `d` and `e` are all Boolean, for the sake of simplicity):

```
if (x) {
    a = b;
} else {
    a = c ^ d ^ e;
}
```

would be implemented in a bitsliced form as:

```
a = (x & b) | (~x & (c ^ d ^ e));
```

If `x` is a secret data, the branching code is vulnerable to timing attacks: it runs faster if `x` is true than if it is false. An attacker could observe the execution time and deduce the value of `x`. On the other hand, the bitsliced code executes the same instructions independently of the value of `x` and is thus immune to timing attacks. For large conditionals, this would be expensive, since it requires computing both branches instead of one. However, this is not an issue when implementing symmetric cryptographic primitives, as they rarely—if ever—rely on conditionals.

The lack of branches is not sufficient to make a program constant-time: memory accesses based on secret indices could be vulnerable to cache-timing attacks (presented in Section 1.0.1). However, as shown in the previous section, bitslicing prevents such memory accesses by replacing lookup tables by constant-time circuits.

Since they prevent both conditional branches based on secret data, and memory accesses based on secret data, bitsliced programs are constant-time by construction.

Transposition

Transposing the data from a direct representation to a bitsliced one is expensive. Naively, this would be done bit by bit, using the following algorithm for a matrix of 64 64-bit registers (a similar algorithm can be used for any matrix size):

```
void naive_transposition(uint64_t data[64]) {
    // transposing |data| in a local array
    uint64_t transposed_data[64] = { 0 };
    for (int i = 0; i < 64; i++) {
        for (int j = 0; j < 64; j++) {
            transposed_data[j] |= ((data[i] >> j) & 1) << i;
        }
    }
    // copying the local array into |data|, thus transposing |data| in-place
    memcpy(data, transposed_data, 64 * sizeof(uint64_t));
}
```

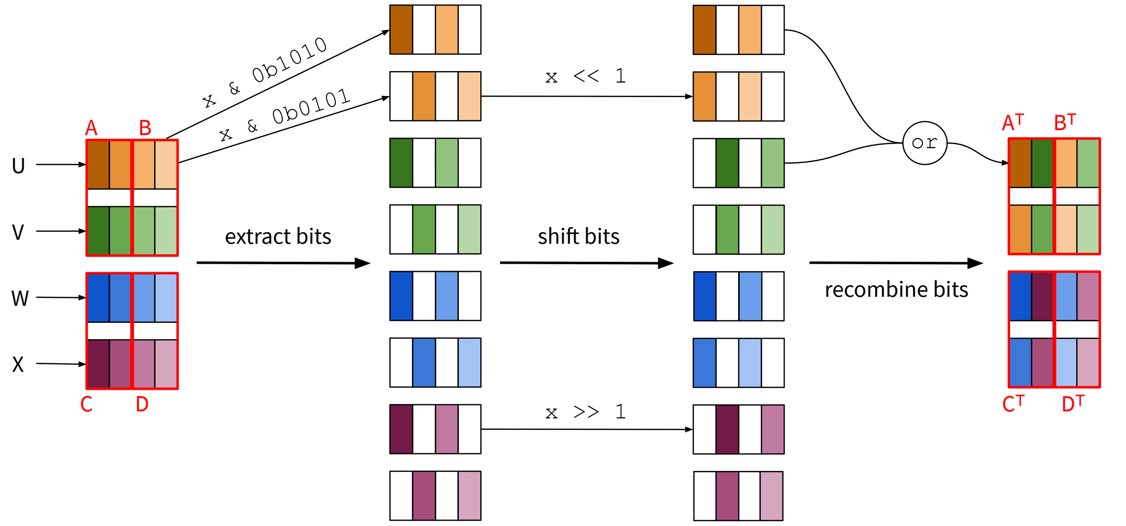
This algorithm does 4 operations per bit of data (a left-shift <<, a right-shift >>, a bitwise and `&` and a bitwise or `|`), thus having a cost in $\mathcal{O}(n)$ where n is the size in bit of the input. Given that modern ciphers can have a cost as low as half a cycle per byte (CHACHA20 on AVX-512, for instance), spending 1 cycle per bit (8 cycles per byte) transposing the data would make bitslicing too inefficient to be used in practice. However, this transposition algorithm can be improved (as shown by Knuth [184], and explained in detail by Pornin [248]) by observing that the transpose of a matrix can be recursively written as:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix}$$

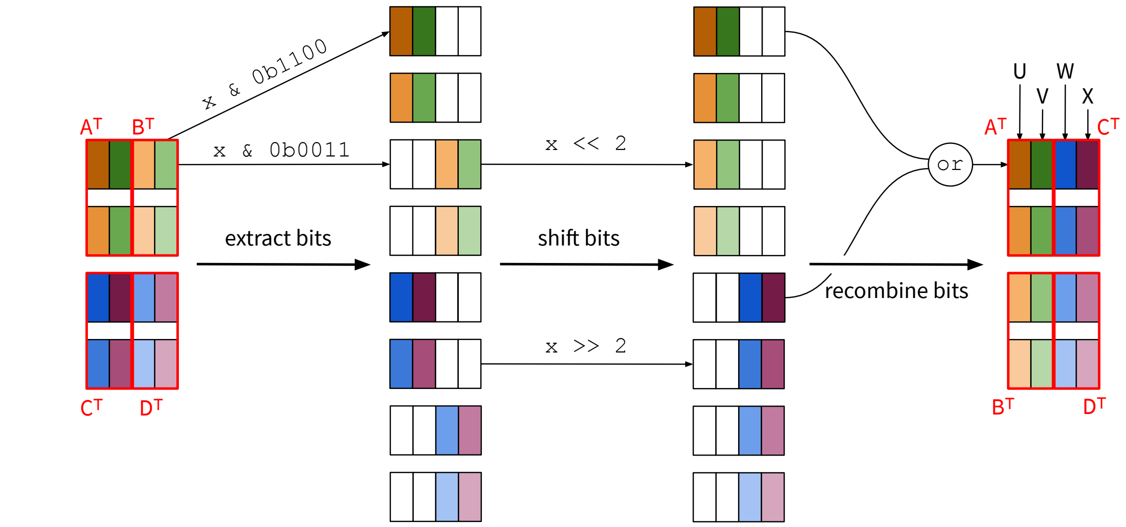
until we are left with matrices of size 2×2 (on a 64x64 matrix, this takes 6 iterations). Swapping B and C is done with shifts, `or`s, and `and`s. The key factor is that when doing this operation recursively, the same shifts/`and`s/`or`s are applied at the same time on A and B, and on C and D, thus saving a lot of operations compared to the naive algorithm.

Example 1.2.1. Let us consider a 4×4 matrix composed of 4 4-bit variables `U`, `V`, `W` and `X`. The first step is to transpose the 2×2 sub-matrices (corresponding to A, B, C and D above). Since those are 2×2 matrices, the transposition involves no recursion (Figure 1.10a). Transposing A and B is done with the same operations: `& 0b1010` isolates the leftmost bits in both A and B at the same time; `<< 1` shifts the rightmost bits of both A and B at the same time, and the final `or` recombines both A and B at the same time. The same goes for C and D. The individual transpose of B and C can then be swapped in order to finalize the whole transposition (Figure 1.10b).

When applied to a matrix of size $n \times n$, this algorithm performs $\log(n)$ recursive steps to get to 2×2 matrices, each of them doing n operations to swap sub-matrices B and C. The total cost is therefore $\mathcal{O}(n \log n)$ for a $n \times n$ matrix, or $\mathcal{O}(\sqrt{n} \log n)$ for n bits. On a



(a) First step



(b) Second step

Figure 1.10: Optimized transposition of a 4×4 matrix

modern Intel CPU (*e.g.*, Skylake), this amounts to 1.10 cycles per bits on a 16x16 matrix, down to 0.09 cycles per bits on a 512x512 matrix [212].

Furthermore, in a setting where both the encryption and decryption are bitsliced, transposing the data can be omitted altogether. Typically, this could be the case when encrypting a file system [248].

Arithmetic Operations

Bitslicing prevents from using CPU arithmetic instructions (addition, multiplication *etc.*), since each n-bit number is represented by 1 bit in n distinct registers. Instead, bitsliced programs must re-implement binary arithmetic, using solely bitwise instructions. As we will show in Section 1.2.5, using bitslicing to implement additions is 2 to 5 times slower than using native CPU `add` instructions, and multiplication would be much slower. We are not aware of any bitsliced implementation of a cipher that simulates arithmetic operations in this way.

1.2.3 *m*slicing

Bitslicing can produce high-throughput cipher implementations through data-parallelism. However, it suffers from a few limitations:

- bitslicing requires *a lot* of independent inputs to be efficient, since throughput is achieved by encrypting multiple inputs in parallel (m parallel inputs on m-bit registers).
- bitsliced code uses hundreds of variables, which puts a lot of pressure on the registers, which causes spilling (moving data back-and-forth between registers and memory), thus reducing performance.
- bitslicing cannot efficiently implement ciphers which rely heavily on arithmetic operations, like CHACHA20 [61] and THREEFISH [136].

To overcome the first two issues, Käsper and Schwabe [174] proposed a "byte-sliced" implementation of AES. Bitslicling would represent the 128-bit block of AES as 1 bit in 128 registers. Instead, Käsper and Schwabe proposed to represent the 128-bit block as 16 bits in 8 registers. Using only 8 registers greatly reduces register pressure and allows AES to be implemented without any spilling. Furthermore, this representation requires fewer inputs to fill the registers and thus maximize throughput: only 8 parallel AES inputs are required to fill up 128-bit SSE registers (against 128 with bitslicing).

We define *m*slicing as a generalization of bitslicing, where a n-bit input is split into k bits in m registers (such that $k \times m = n$). The special case $m = 1$ corresponds to bitslicing. When k (the number of bits in each register) is greater than one, there are two possible ways to arrange the bits within each register: they can either be stored contiguously, or they can be spread across each packed element. The first option, which we call *v*slicing, will enable the use of vertical instructions (*e.g.*, 16-bit addition), while the second option, which we call *h*slicing, will enable the use of permutation instructions (*e.g.*, `pshufb`).

Vertical Slicing

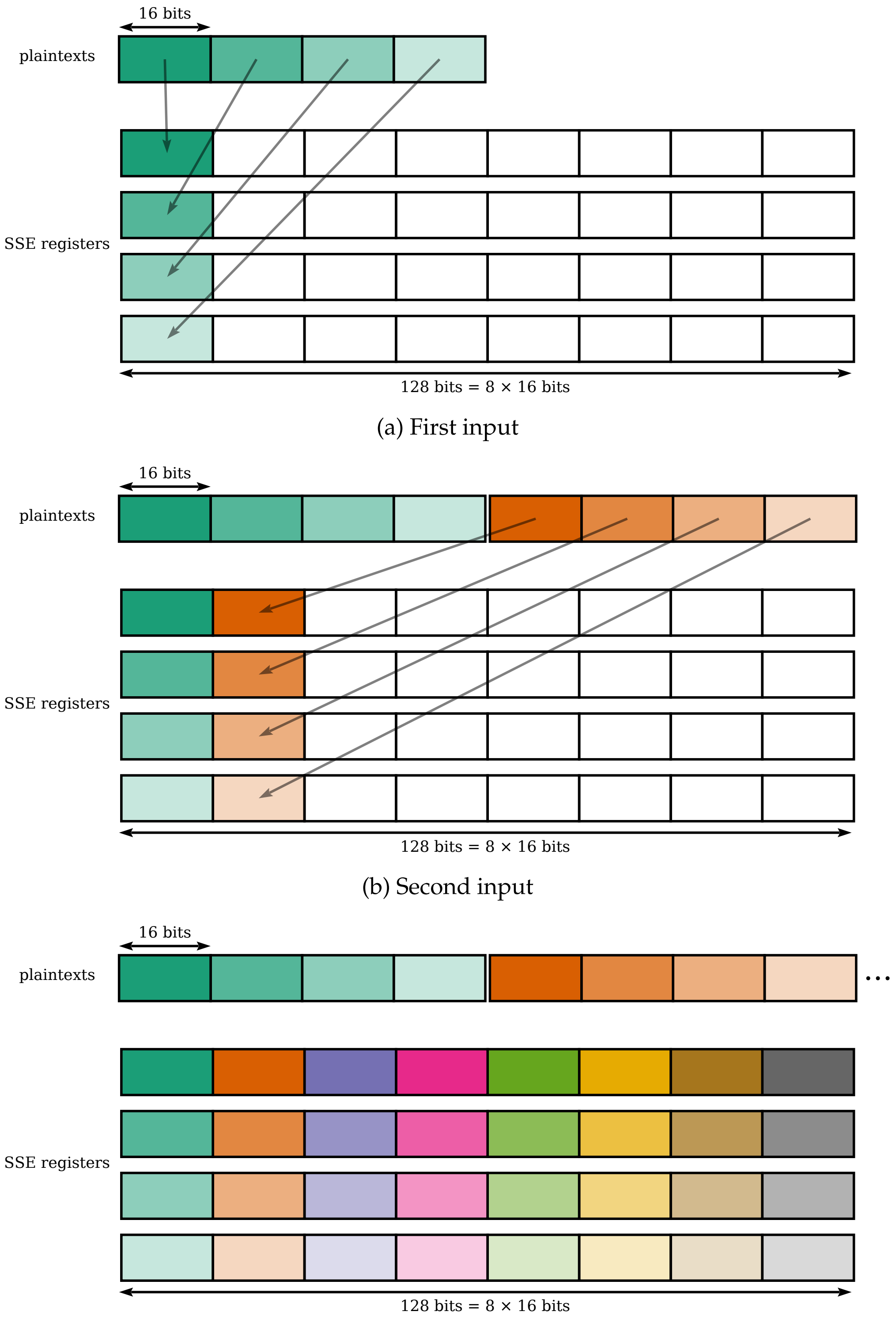
Rather than considering the bit as the atomic unit of computation, one may use m-bit words as such basis (m being a word size supported by the SIMD architecture). On Intel (SSE/AVX/AVX512), m can be 8, 16, 32, or 64. We can then exploit vertical m-bit SIMD instructions to perform logic as well as arithmetic in parallel.

This technique, which we call vertical slicing (or *v*slicing for short), is similar to the notion of *vectorization* in the compilation world. Compared to bitslicing, it puts less pressure on registers, and requires less parallel data to fill the registers. Furthermore, it can be used to implement arithmetic-based ciphers (like CHACHA20), which cannot be implemented efficiently in bitslicing.

However, permutations are costly in *v*slicing. Applying an arbitrary bit-permutation to the whole block requires using shifts and masks to extracts bits from the registers and recombine them, as shown in Section 1.2.2.

Example 1.2.2 (*v*sliced RECTANGLE)**.** The RECTANGLE cipher [309] (Figure 1.1, Page 19) encrypts a 64-bit input. We can split this input into 4 16-bit elements, and store them in the first 16-bit packed elements of 4 SSE registers (Figure 1.11a). Then, applying the same principle as bitslicing, we can fill the remaining empty elements of these SSE registers with independent inputs (Figures 1.11b and 1.11c), until the registers are full (in the case of RECTANGLE on 128-bit SSE registers, this is achieved with 8 inputs).

*v*sliced RECTANGLE can be efficiently computed in parallel. The S-box is a sequence of bitwise `and`, `xor`, `not` and `or` between each of these SSE registers, and is thus efficient

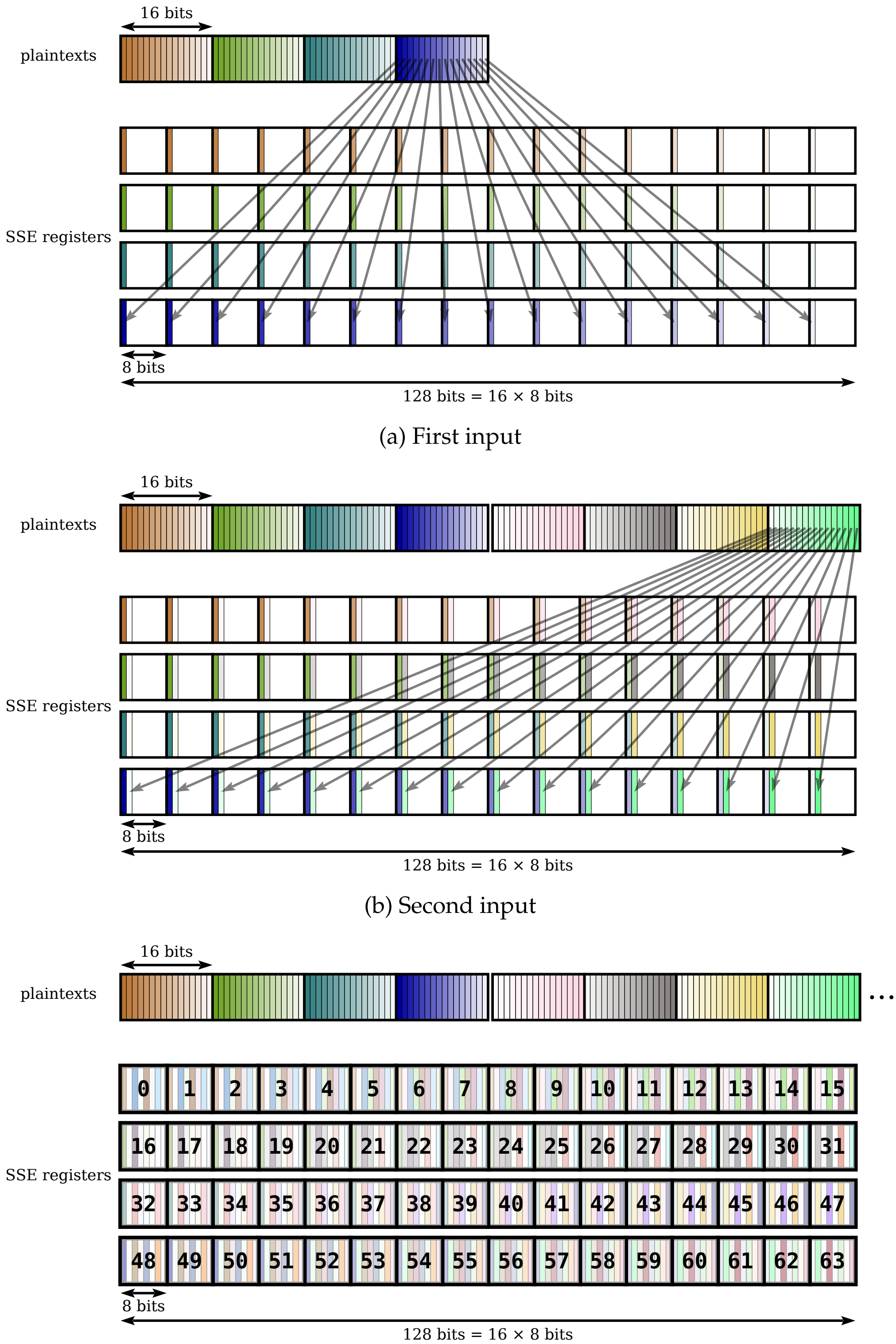


(a) First input

(b) Second input

(c) All height inputs

Figure 1.11: Data layout of *v*sliced RECTANGLE



(c) All 8 inputs

Figure 1.12: Data layout of *h*sliced RECTANGLE

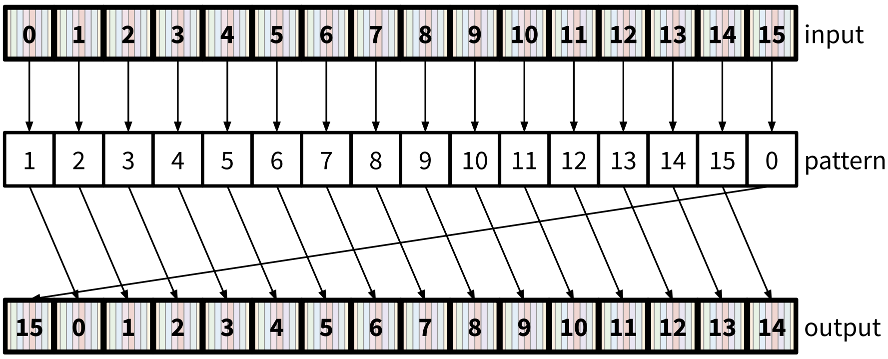


Figure 1.13: A left-rotation using a shuffle

in bitslicing and *v*slicing. The permutation can be written as three left-rotations, each one operating on a different SSE register. While the SSE extension does not offer a rotation instruction, it can be emulated using two shifts and an `or`. This is slightly more expensive than the bitsliced implementation of RECTANGLE, which can perform this permutation at compile time. However, the reduced register pressure more than compensate for this loss, as we shall demonstrate in our evaluation (Section 5.3).

Horizontal Slicing

Rather than considering a m-bit atom as a single packed element, we may also dispatch its m bits into m distinct packed elements, assuming that m is less or equal to the number of packed elements of the architecture. Using this representation, we lose the ability to perform arithmetic operations but gain the ability to perform arbitrary shuffles of our m-bit word with a single instruction (Figure 1.3d, Page 25). This style exploits on horizontal SIMD operations.

We call this technique *horizontal* slicing, or *h*slicing for short. Like *v*slicing, it lowers register pressure compared to bitslicing, and requires fewer inputs to fill the registers. While *v*slicing shines where arithmetic operations are needed, *h*slicing is especially useful to implement ciphers that rely on intra-register permutations, since those can be performed in a single shuffle instruction. This includes RECTANGLE's permutations, which can be seen as left-rotations, or Piccolo's permutation (Figure 1.9, Page 33).

Permutations mixing bits from distinct registers, however, will be expensive, as they will require manually extracting bits from different registers and recombining them.

Example 1.2.3 (*h*sliced RECTANGLE)**.** On RECTANGLE, the 64-bit input is again seen as 4 times 16 bits, but this time, each bit goes to a distinct packed element (Figure 1.12a). Once again, there are unused bits in the registers, which can be filled with subsequent inputs. The second input would go into the second bits of each packed element (Figure 1.12b). Like for *v*slicing, this is repeated until the registers are full, which in the case of RECTANGLE on SSE requires a total of 8 inputs. Numbering the bits from 0 (most significant) to 63 (least significant) can help visualize the *h*slicing data layout (Figure 1.12c): the 8-bit element labeled i contains the i-bit of each of the 8 inputs.

RECTANGLE's S-box, composed solely of bitwise instructions, can be computed using the same instructions as the *v*sliced implementation. The permutation, however, can be done even more efficiently than in *v*slicing, using shuffle instructions. For instance, Figure 1.13 shows how a shuffle can perform the first left-rotation of RECTANGLE's linear layer: given the right pattern, the `pshufb` instruction acts as a byte-level rotation of a SSE register.

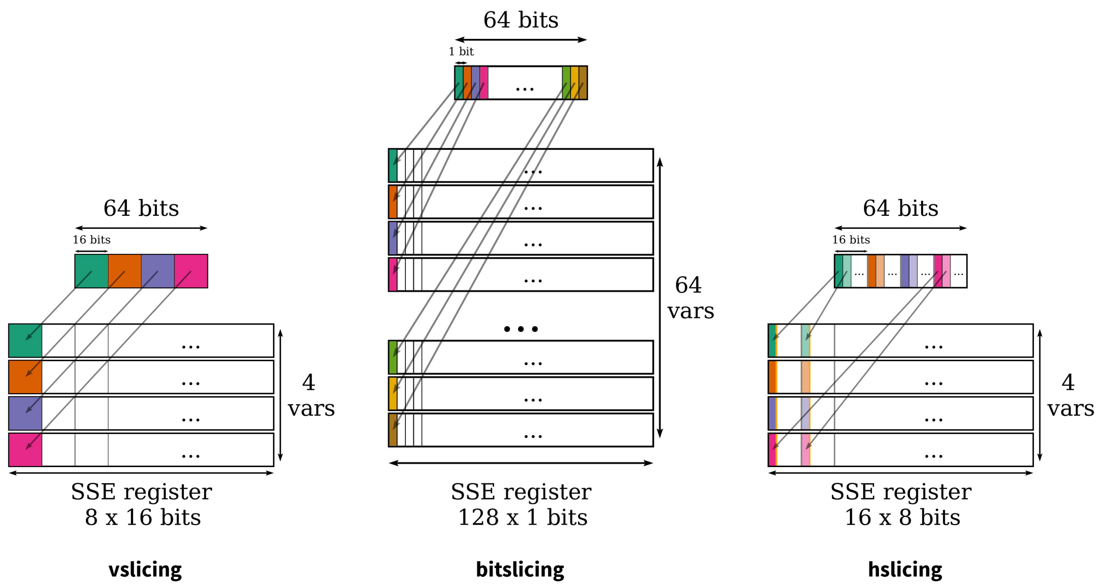


Figure 1.14: Slicing layouts

1.2.4 Bitslicing *vs.* *v*slicing *vs.* *h*slicing

Terminology. Whenever the slicing direction (*i.e.*, vertical or horizontal) is unimportant, we talk about *m*slicing (assuming $m > 1$) and we call *slicing* the technique encompassing both bitslicing and *m*slicing.

Figure 1.14 provides a visual summary of all slicing forms using RECTANGLE as an example. All these slicing forms share the same basic properties: they are constant-time, they rely on data-parallelism to increase throughput, and they use some sort of transposition to convert the data into a layout suitable to their model. However, each slicing form has its own strengths and weaknesses:

- transposing data to *h*slicing or *v*slicing usually has a cost almost negligible compared to a full cipher, while a bitslice transposition is much more expensive.
- bitslicing introduces a lot of spilling, thus reducing its potential performance, unlike *h*slicing and *v*slicing.
- only *v*slicing is a viable option on ciphers using arithmetic operations, since it can use SIMD arithmetic instructions. As shown in Section 1.2.5, trying to implement those instructions in bitslicing (or *h*slicing) is suboptimal.
- bitslicing provides zero-cost permutations, unlike *h*slicing and *v*slicing, since it allows to perform any permutation at compile-time. Intra-register permutations can still be done at run time with a single instruction in *h*slicing using SIMD shuffle instructions.
- bitslicing requires much more parallel data to reach full throughput than *h*slicing and *v*slicing. On RECTANGLE, for instance, 8 independent inputs are required to maximize the throughput on SSE registers using *m*slicing, while 128 inputs are required when using bitslicing.
- both *v*slicing and *h*slicing rely on vector instructions, and are thus only available on CPUs with SIMD extensions, while bitslicing does not require any hardware-specific instructions. On general-purpose registers, bitslicing is thus usually more efficient than *m*slicing.

The choice between bitslicing, *v*slicing and *h*slicing thus depends on both the cipher and the target architecture. For instance, on CPUs with SIMD extensions, the fastest implementations of DES are bitsliced, the fastest implementations of CHACHA20 are *v*sliced, while the fastest AES implementations are *h*sliced. Even for a given cipher, some slicings might be faster depending on the architecture: on x86-64, the fastest implementation of RECTANGLE is bitsliced, while on AVX, *v*slicing is ahead of both bitslicing and *h*slicing. If we exclude the cost of transposing the data, then *h*sliced and *v*sliced implementations of RECTANGLE have the same throughput.

Statically estimating which slicing mode is the most efficient for a given cipher is non-trivial, especially on complex CPU architectures. Instead, Usuba allows to write code that is polymorphic on the slicing types (when possible), making it easy to switch from a slicing form to another, and thus measure relative performance.

1.2.5 Example: Bitsliced adders

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/experimentations/add`

A bitsliced adder can be implemented using bitwise operations. The simplest adder is the ripple-carry adder, which works by chaining n full adders to add two n-bit numbers. Figure 1.15a illustrates a full adder: it takes two 1-bit inputs (A and B) and a carry (C_{in}), and returns the sum of A and B (s) as well as the new carry (C_{out}). A ripple-carry can then be implemented using a chain of full adders (Figure 1.15b).

Various techniques exist to build more efficient hardware adders than the carry-ripple adder (*e.g.,* carry-lookahead), but none applies to software implementations.

A software implementation of a n-bit carry-ripple adder thus contains n full adders, each doing 5 operations (3 `xor` and 2 `and`), for a total of $5n$ instructions. Since bitslicing still applies, such a code executes m additions at once when ran on m-bit registers (*e.g.,* 64 on 64-bit registers, 128 on SSE registers). The cost to do one n-bit addition is therefore $5n/m$ bitwise operations. On a high-end CPU, this adder is unlikely to execute in exactly $n * 5$ cycles. The number of registers, the superscalarity, and L1 data-cache latency will all impact performance.

In order to evaluate such a bitsliced adder, we propose to compare it with native packed addition instructions on SSE and AVX registers (*i.e.,* instructions that would be used for *v*slicing). SSE (resp., AVX) addition instructions do k n-bit additions with a single instruction: 16 (resp., 32) 8-bit additions, or 8 (resp., 16) 16-bit additions, or 4 (resp., 8) 32-bit additions, or 2 (resp., 4) 64-bit additions.

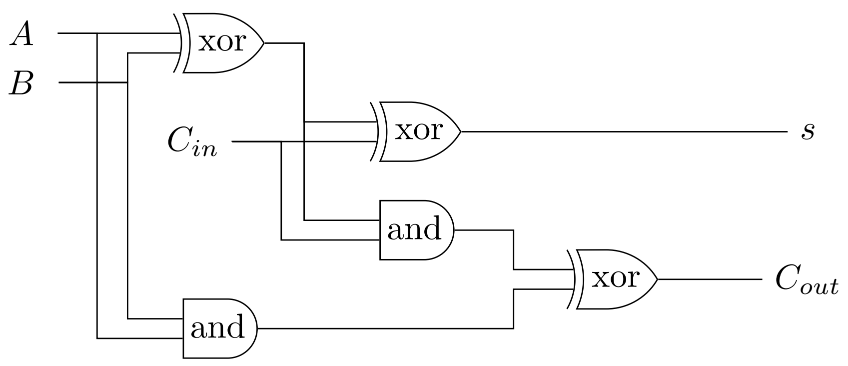
1.2.6 Setup

We consider 3 different additions for our evaluation:

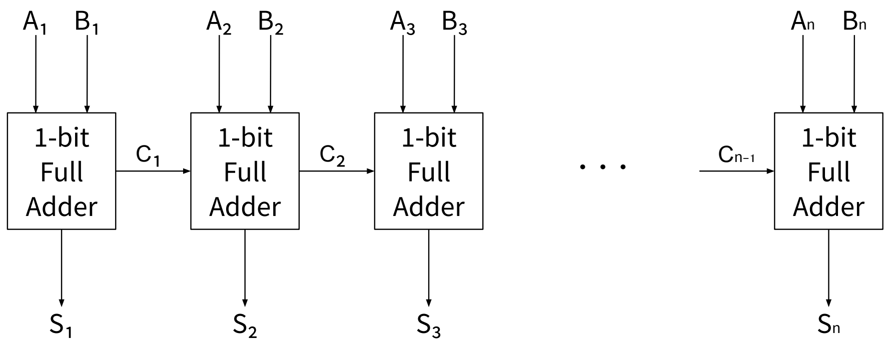
- a bitsliced ripple-carry adder. Three variants are used: 8-bit, 16-bit and 32-bit. Since they contain $5 * n$ instructions, we could expect the 32-bit adder to run in twice more cycles than the 16-bit adder, which itself would run in twice more cycle than 8-bit one. However, the larger the adder, the higher the register pressure. Our benchmark aims at quantifying this effect.
- a packed addition done using a single CPU instruction. For completeness, we consider the 8, 16 and 32-bit variants of the addition; all of which should have the same throughput and latency on general purpose (GP) registers. GP registers do not offer packed operations, and can therefore only do a single addition with an

add instruction. SSE (resp., AVX) on the other hand, can 4 (resp., 8) 32-bit, or 8 (resp., 16) 16-bit or 16 (resp., 32) 8-bit additions with a single packed instruction, which increases throughput without changing latency.

- 3 independent packed additions done with 3 CPU instructions operating on independent registers. Doing a single packed addition per cycle under-utilizes the superscalar capabilities of modern CPUs. Since up to 3 SSE or AVX additions (or 4 GP additions, since port 6 of the Skylake CPU can execute GP arithmetic and bitwise instructions but not SIMD instructions) can be done each cycle, this benchmark should show the maximum throughput achievable by native add instructions. Once again, we consider the 8, 16 and 32-bit variants, on SSE, AVX and GP.



(a) Full-adder



(b) Ripple-carry adder

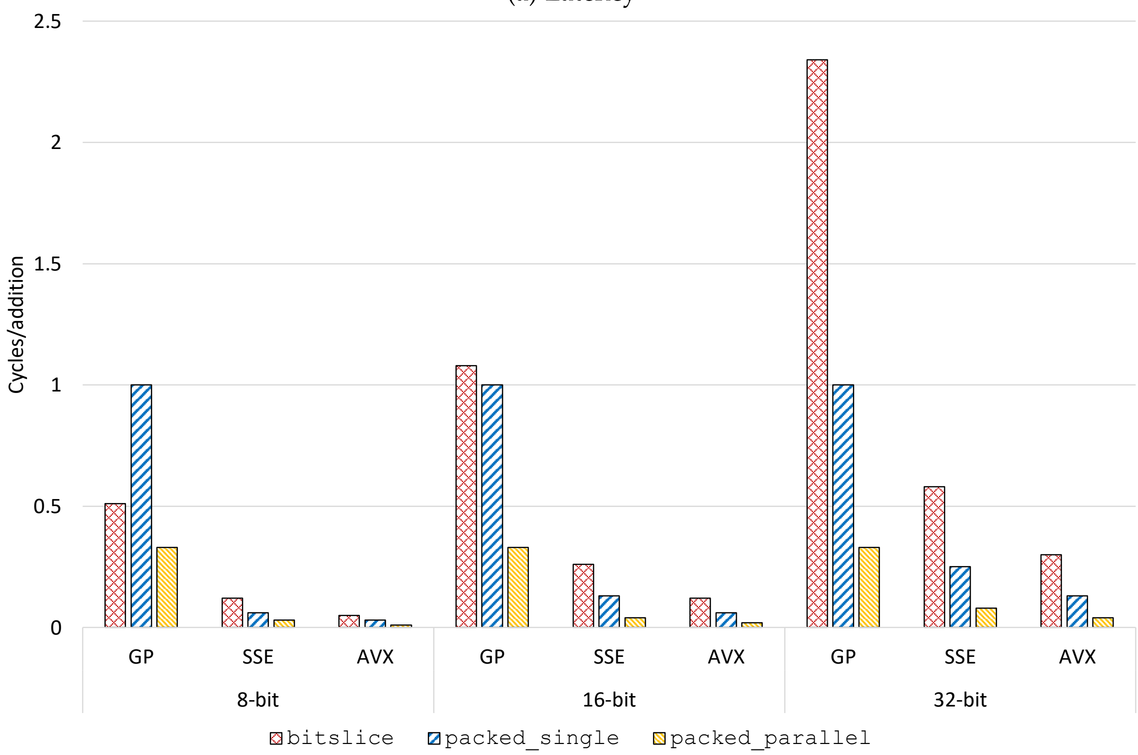
```
node full_adder(a,b,c_in:b1) returns (s,c_out:b1)
let
    s     = (a ^ b) ^ c_in;
    c_out = ((a ^ b) & c) ^ (a & b);
tel

node 32bit_ripple_carry(a,b:b32) returns (s:b32)
vars c:b1
let
    c = 0;
    forall i in [0, 31] {
        (s[i],c) := full_adder(a[i],b[i],c);
    }
tel
```

(c) 32-bit ripple-carry adder in Usuba

Figure 1.15: Circuits for some adders

Addition type	Cycles/iteration		
	AVX	SSE	GP
8-bit `bitslice`	12.19	14.78	16.30
8-bit `packed_single`	1.01	1.00	1.00
8-bit `packed_parallel`	1.00	1.00	1.00
16-bit `bitslice`	30.93	33.34	34.60
16-bit `packed_single`	1.00	1.00	1.00
16-bit `packed_parallel`	1.00	1.00	1.00
32-bit `bitslice`	76.17	74.78	74.93
32-bit `packed_single`	1.00	1.00	1.00
32-bit `packed_parallel`	1.00	1.00	1.00

(a) Latency



(b) Throughput

Figure 1.16: Comparison between bitsliced adder and native addition instructions

We compiled our C code using Clang 7.0.0. We tried using GCC 8.3.0, but it has a hard time with instruction scheduling and register allocations, especially within loops, and thus generates suboptimal code. We ran the benchmarks on a Intel Skylake i5-6500.

1.2.7 Results

We report in Figure 1.16 the latency in cycles per iteration (Figure 1.16a) and throughput cycles per addition (Figure 1.16b). `bitslice` corresponds to the bitsliced addition using a ripple-carry adder. `native_single` corresponds to the single addition done using a native packed instruction. `native_parallel` corresponds to the three independent native packed additions.

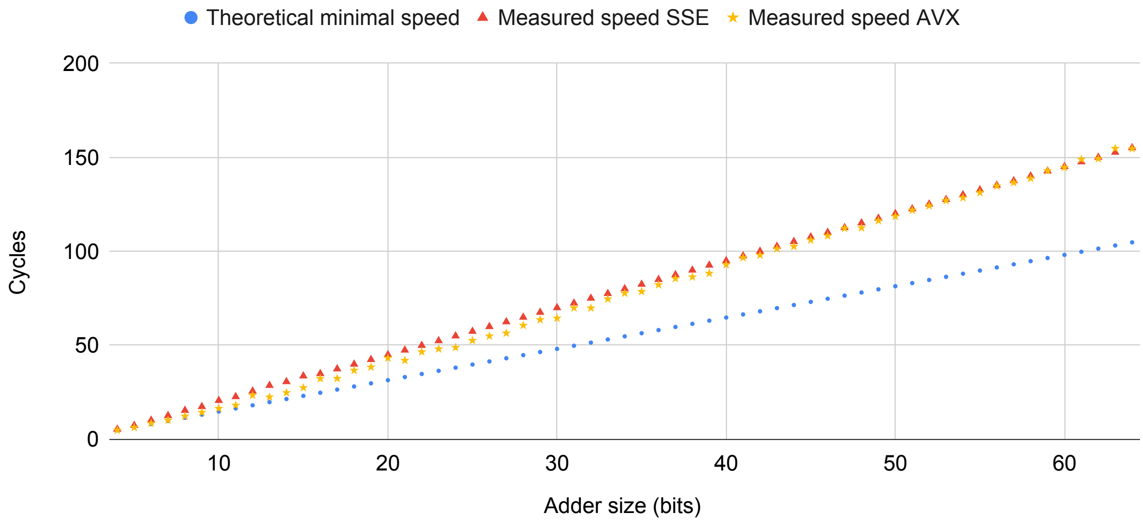


Figure 1.17: Scaling of bitsliced adders on SIMD

Single Native Packed Addition (`native_single`)

According to Intel's manual, additions (SSE, AVX and GP) should execute in one cycle. There should be no overhead from the loop: looping requires incrementing a counter and doing a conditional jump. Those two instructions get macro-fused together and execute on port 0 or 6. The SSE and AVX additions can execute on either port 0, 1 or 5, and can thus be done at the same time as the loop increment/jump.

Experimentally, we achieve a latency of 1 cycle/iteration, regardless of the size of the addition (8-bit, 16-bit or 32-bit). On GP registers, this means a throughput of 1 addition per cycle. On SSE (resp., AVX) registers, however, a single packed `padd` (resp., `vpadd`) instruction performs multiple additions, thus reducing the cost per addition: 0.25 (resp., 0.13) cycles for 32-bit, 0.13 (resp., 0.06) cycles for 16-bit and 0.06 (resp., 0.03) cycles for 8-bit.

Parallel Native Packed Parallel Additions (`native_parallel`)

Since SSE and AVX addition can execute on either port 0, 1 or 5, three of them could be done in a single cycle, provided that they do not suffer from data dependencies (*i.e.*, none uses the output of another one). The increment and jump of the loop would fuse and be executed on port 6, independently of the additions.

Once again, this corresponds to the numbers we observe experimentally: 1 cycle/iteration regardless of the size of the addition. On GP, SSE and AVX registers, 3 additions are executed per cycle, which translates into three times the throughput of `native_single`.

Bitsliced Addition (`bitslice`)

Recall that the n-bit ripple-carry adder contains n full-adders. In practice, 3 operations can be omitted from the first full-adder, since it always receives a 0 as input carry. Likewise, 3 more operations can be saved from the last full-adder, since its output carry is never used and does not need to be computed. We did not have to implement those optimizations by hand since Clang is able to find them by itself. The total numbers of operations of this n-bit ripple-carry adder is therefore $n \times 5 - 6$. Those operations are solely bitwise instructions, which can be executed on ports 0, 1 and 5 on SSE and AVX registers (and on port 6 as well on GP registers). This adder is therefore bound to run in at least $(n \times 5 - 6)/3$ cycles on SSE and AVX registers, and $(n \times 5 - 6)/4$ cycles on GP registers.

In practice, this number can never be reached because a full-adder contains inner data dependencies that prevent it from executing at a rate of 3 instructions per cycle. However, when computing consecutive full-adders, a full-adder can start executing before the previous one is done, thus bypassing this dependency issue. The limiting factor then becomes the CPU scheduler, which cannot hold arbitrarily many μops, and limits the out-of-order execution. The larger the adder, the more full-adders (and thus μops) they contain, and the more the scheduler will be saturated.

Figure 1.17 shows the minimal theoretical speed ($(n * 5 - 6)/3$ cycles) and the measured speed on SSE and AVX of ripple-carry adders depending on their sizes (from 4-bit to 64-bit). Small adders (around 4-bit) almost reach their theoretical minimal speeds. However, the larger the adders, the slower they get compared to the theoretical optimum. This is partly because of their inner dependencies, as mentioned above, but also due to spilling. Indeed, a bitsliced n-bit adder has $n * 2$ inputs, n outputs, and $n * 5$ temporary variables. Even though most of the temporary variables are short-lived, the register pressure remains high: there are only 16 SSE registers available and 14 GP registers (since one register is always kept for the stack pointer, and another one to store the loop counter). Without surprises, the larger the adders, the more registers they need, and the more they suffer from spilling.

Especially on small adders, AVX instructions are faster than SSE, because the latter use destructive 2-operand instructions, while the former use non-destructive 3-operand instructions. When computing a bitsliced SSE adder, some variables must therefore be saved (using a `mov`) before being overwritten with a new result. Even though `mov`es execute on ports 2, 3 and 4, whereas bitwise instructions use ports 0, 1 and 5, this introduces data dependencies, and is enough to slightly slow down the SSE adders compared to the AVX adders.

Since the CPU can do 4 GP bitwise instructions per cycle, and only 3 SSE and AVX, the GP adders should be faster than the SSE and AVX ones. However, the SSE and AVX adders use 16 registers, while the GP ones use only 14 registers (recall that two GP registers are reserved). This causes the GP adders to do more spilling than the SSE and AVX ones. Furthermore, the inner dependencies of the adders are the same regardless of the registers used. Since the SSE and AVX adders struggle to fully saturate 3 ports, the GP adders have an even harder time to saturate 4 ports. Overall, this causes AVX, SSE and GP adders to have similar execution time.

Conclusion On general-purpose registers, using bitslicing can improve the speed of small additions: an 8-bit bitsliced adder is twice faster than native `add` instructions. However, when SIMD extensions are available, bitslicing becomes 2 to 6 times slower than native instructions. The decision to emulate addition in bitslicing should therefore depend on both the size of the addition and the architecture.
Furthermore, the other parts of the ciphers will also influence this decision: a program mixing additions and permutations might be a good target for bitslicing, while a program containing solely additions is unlikely to be a good candidate for bitslicing.
On the other hand, it would be pointless to emulate multiplication this way: a multiplier requires at least n^2 instructions (whereas an adder only requires $n \times 5$), making the resulting program prohibitively expensive.

1.3 Conclusion

In this chapter, we motivated the need for domain-specific languages for cryptography, in order to automate the generation of efficient and secure cryptographic implementations. We showed how bitslicing can be used to achieve both: high throughput is achieved through data parallelization, and security against timing attack is achieved through constant time.
To overcome several limitations of bitslicing (*e.g.,* high latency, high register pressure), we proposed a generalized *m*slicing model, which often delivers higher throughputs that bitslicing, in addition to being able to support arithmetic-based ciphers (unlike bitslicing). Bitslicing and *m*slicing are able to fully exploit SIMD extensions of modern CPUs, which offer large registers (up to 512 bits), and can thus be used to provide high-throughput implementations of cryptographic primitives.

1.3.1 Related Work

Constant-time Cryptography

Sliced programs are constant-time *by construction*. However, not all cryptographic code can be sliced: slicing imposes data parallelization (to be efficient) and cannot efficiently compute conditionals (which are common in public-key cryptography). Several methods are used in cryptography to ensure constant-time in situations where slicing cannot be used.

Empirical Evaluation. Several tools allow to empirically check whether a piece of code runs in constant time. ctgrind [195] uses a modified version of Valgrind's Memcheck tool, which, instead of detecting (among other things) branches on uninitialized memory or accesses to uninitialized memory, detects branches on secret data and accesses to secret data (using a taint-tracking analysis where only secret inputs are initially tainted).

dudect [259] measures the execution times of a code with different inputs, and checks whether the timing distributions are statistically different: if so, then the code likely has a timing leak. Repeating this process many times allows to increase the confidence in the lack (or presence) of timing leaks.

ct-fuzz [160] relies on coverage-based greybox fuzzy testing to detect timing leaks. Their approach is based on *self-composition*: several copies of a program are executed in isolation with different inputs, and the traces (control flow, memory accesses) are recorded. If the traces are distinguishable, then a timing leak exists. Coverage-based greybox fuzzy testing is used to generate random inputs by mutating previously explored inputs. This allows ct-fuzz to achieve a similar result as dudect, albeit with fewer false positives, since ct-fuzz does not record timings but rather execution traces.

Static Analysis. Rather than empirically checking whether a code is *seemingly* constant-time, several approaches have been proposed to statically determine whether a piece of code is constant-time.

CacheAudit [130] uses abstract interpretation to detect cache-timing leaks. Similarly, tis-ct [163], inspired by ctgrind, leverages the Frama-C static analyzer to perform a dependency analysis in C code to detect timing leaks. FlowTracker [269], on the other hand, relies on an analysis of the SSA graph to track data dependencies and detect potential timing leaks. Several type systems have been proposed to ensure constant-time and detect timing leaks [42, 223, 297]. Finally, several approaches to detect timing leaks using self-composition (statically, unlike ct-fuzz) have also been proposed [16, 17].

More recently, Barthe et al. [45] proposed a modified version of CompCert [199] that preserves constant-time. They extended CompCert's semantics to include the notion and leakage, and modified the existing proofs of semantics preservation to cover timing leakages.

Source-to-source Transformation. Going further than simply detecting timing leaks, several tools have been developed to automatically transform arbitrary code into constant-time code.

Both Barthe et al. [40] and Molnar et al. [217] proposed generic transformations to remove control-flow based timing leaks based by computing both branches of conditionals. The former is a generic approach which relies on a transaction mechanism: conditionals are replaced by transactions, which are committed or aborted depending on the value of the condition. The latter is lower-level, and proposes the *program counter* security model, which considers an attacker that knows which instruction is being executed at any time. In this model, they propose program transformations to ensure that the information provided by the current value of the program counter does not reveal sensitive data. Coppens et al. [114] later improved on [217] by implementing a similar method directly as an LLVM backend in order to prevent the compiler from introducing timing leaks, and by dealing with conditional function calls and memory accesses.

SC-Eliminator [304], implemented as an LLVM pass, improves on the aforementioned techniques by removing cache-leaks as well as control-flow leaks. First, a static sensitivity analysis detects timing leaks by propagating sensitivity from secret inputs to other variables: a branch on a sensitive value, or a memory lookup on a sensitive index are considered to introduce a leak. A second pass then removes those leaks: both branches of conditionals are computed (and the result is selected in constant time), and preloading is used to mitigate cache-timing leaks.

Similarly, FaCT [100, 99] is a DSL providing high-level constructions and a dedicated compiler to remove any potential timing leaks. However, since FaCT generates LLVM bytecode, there is a risk that LLVM may break FaCT's constant-time guarantees. In the hope of detecting any timing leaks introduced by LLVM, FaCT uses dudect to ensure that the code thus generated is (empirically) constant-time.

Bitslicing Outside of Cryptography

The earliest use of the term bit slicing was to describe a technique to implement n-bit processors from components with smaller bit-width [215]. For instance, a 16-bit arithmetic logic unit (ALU) can be made from four 4-bit ALU (plus some additional machinery for carries). Although this technique is different from what we call bistlicing in this thesis, it is related: our bitslicing views a n-bit CPU as n 1-bit CPU, whereas this hardware bit slicing builds a n-bit CPU from n 1-bit components. This technique is, however, not used in large-scale manufactured processors (*e.g.*, Intel and ARM), even though some recent works suggest that it can be exploited to implement rapid single-flux-quantum (RSFQ) microprocessors [288].

Bitslicing has also been used as a data storage layout in order to speed up scans in databases, under the names "Bit-Slicing" [231] and "Vertical Bit-Parallel" [202]. SIMD vectorizations, similar to $v$slicing, have also been proposed to accelerate scans in compressed databases [301].

Maurer and Schilp [208] proposed to use of bitslicing to speed up software simulation of hardware: bitslicing allows to simulate a circuit on multiple inputs (*e.g.*, 32 on a 32-bit CPU) at once.

Xu and Gregg [305] developed a compiler to generate bitsliced code for custom precision floating-point arithmetic, whose use-cases include in particular image processing. Kiaei and Schaumont [178] proposed a technique to synthesize general-purpose bitsliced code in order to speed up time-sensitive embedded applications.

Chapter 2

Usuba, Informally

The design of Usuba is driven by a combination of algorithmic and hardware-specific constraints: implementing a block cipher requires some discipline to achieve high throughput and avoid timing attacks.

Usuba must be expressive enough to describe hardware-oriented ciphers, such as DES or TRIVIUM, which are specified in terms of Boolean operations. For instance, the first operation of DES is defined as a 64-bit bit-level permutation (Figure 2.1). To account for such ciphers, Usuba provides abstractions to manipulate bitvectors, such as extracting a single bit as well as splitting or combining vectors.

Usuba must also be able to describe software-oriented ciphers specified in terms of affine transformations, such as AES. For instance, `MixColumns`, one of the underlying functions of AES, is defined as a matrix multiplication in $GF(2^8)$ (Figure 2.2). To account for such ciphers, Usuba handles the matricial structure of each block of data, allowing bit-level operations to carry over such structured types while driving the compiler into generating efficient SIMD code. Altogether, Usuba provides a vector-based programming model, allowing us to work at the granularity of a single bit while providing static type-checking and compilation to efficient code.

Usuba programs are also subject to architecture-specific constraints. For instance, a cipher relying on 32-bit multiplication is but a poor candidate for bitslicing: this multiplication would turn into a prohibitively expensive multiplier circuit simulated in software. Similarly, a cipher relying on 6-bit arithmetic would be impossible to execute in vertical slicing: the SIMD instruction sets manipulate bytes as well as 16-bit, 32-bit and 64-bit words, leaving aside such an exotic word size. We are therefore in a rather peculiar situation where, on the one hand, we would like to implement a cipher once, while, on the other hand, the validity of our program depends on a combination of slicing mode and target architecture.

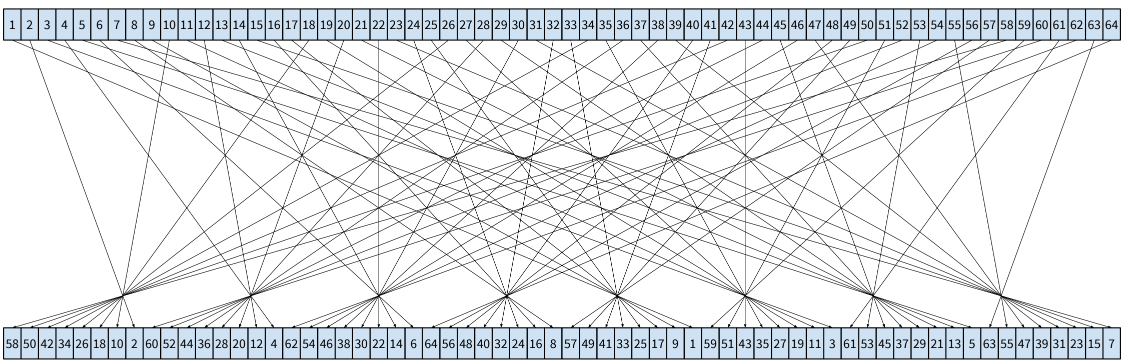


Figure 2.1: DES's initial permutation

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

Figure 2.2: AES's `MixColumns` matrix multiplication

How can we, at compile time, provide meaningful feedback to the Usuba programmer so as to (always) generate high-throughput code? We address this issue by introducing a type for structured blocks (Section 2.1) upon which we develop a language (Section 2.2) supporting parametric polymorphism (for genericity) and ad-hoc polymorphism (for architecture-specific code generation) as well as a type system (Section 2.3) ensuring that "well-typed programs do always vectorize". Overall, this chapter provides an intuitive introduction to Usuba, leaving the formalism to Chapter 3.

2.1 Data Layout

Our basic unit of computation is the block, *i.e.*, a bitvector of statically-known length. To account for its matricial structure and its intended parallelization, we introduce the type $\mathtt{u}_D m \times n$ (written `u<D>mxn` in Usuba source code) to denote $n \in \mathbb{N}^*$ registers of unsigned m-bit words ($m \in \mathbb{N}^*$ and "`u`" stands for "unsigned") that we intend to parallelize using vertical ($D = \mathsf{V}$) or horizontal ($D = \mathsf{H}$) SIMD instructions. A single m-bit value is thus typed $\mathtt{u}_D m \times 1$, abbreviated $\mathtt{u}_D m$. This notation allows us to unambiguously specify the data layout of the blocks processed by the cipher.
Note that directions collapse in the case of bitslicing, *i.e.*, $\mathtt{u}_{\mathsf{V}} 1 \times m \cong \mathtt{u}_{\mathsf{H}} 1 \times m$. Both cases amount to the same layout, or put otherwise: vertical and horizontal slicing are two orthogonal generalizations of bitslicing.

Example 2.1.1. Consider the 64-bit input block of the RECTANGLE cipher (Figure 2.3a). Bitsliced RECTANGLE manipulates blocks of type $\mathtt{u}_D 1 \times 64$, *i.e.*, each bit is dispatched to an individual register (Figure 2.3b). A 16-bit vertical slicing implementation of RECTANGLE manipulates blocks of type $\mathtt{u}_{\mathsf{V}} 16 \times 4$, *i.e.*, each one of the 4 sub-blocks is dispatched to the first 16-bit element of 4 registers (Figure 2.3c, where the double vertical lines represent the end of each 16-bit packed elements). Horizontally sliced RECTANGLE has type $\mathtt{u}_{\mathsf{H}} 16 \times 4$, *i.e.*, each 16-bit word is horizontally spread across each of the 16 packed elements of 4 SIMD registers (Figure 2.3d).

For a given data layout, throughput is maximized by filling the remaining bits of the registers with subsequent blocks of the input stream, following the same pattern. Thus, 16-bit SIMD addition (*e.g.*, `vpaddw` on AVX) in vertical slicing will amount to performing an addition on 8 blocks in parallel. We emphasize that the types merely describe the structure of the blocks, and do not enforce the architecture to be used. An Usuba program only specifies the treatment of a single slice, from which Usuba's compiler automatically generates code to maximize register usage. Transposing a sequence of input blocks in a form suitable for parallel processing (*i.e.*, *v*sliced, *h*sliced or bitsliced) is fully determined by its type.

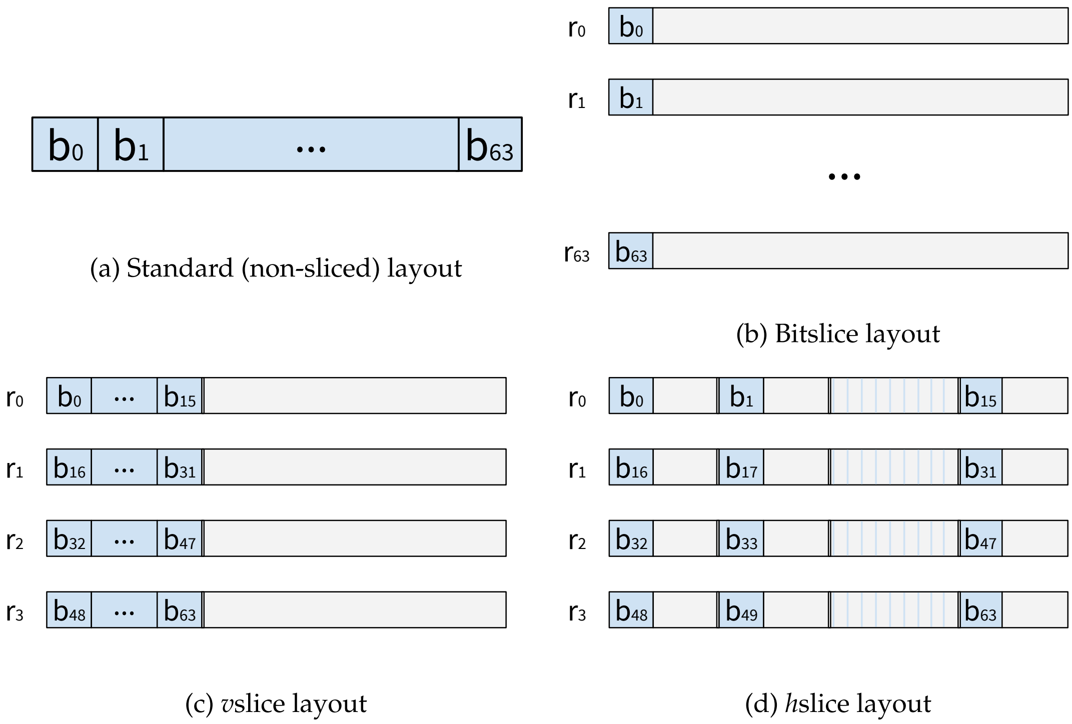


Figure 2.3: RECTANGLE's sliced data layouts

2.2 Syntax & Semantics, Informally

In and of itself, Usuba is an unsurprising dataflow language [98, 6]. Chapter 3 formally describes its semantics and type-system. In this section, in order to give some intuition of the language, we illustrate its syntax and semantics through examples, in particular RECTANGLE (Figure 1.1, Page 19). In the following, we shall leave types and typing aside, coming back to this point in the next section.

Nodes. An Usuba program is composed of a totally ordered set of nodes (`SubColumn`, `ShiftRows` and `Rectangle` in the case of RECTANGLE). The last node plays the role of the main entry point: it will be compiled to a C function. The compiler is free to compile internal nodes to functions or to inline them. A node consists of an unordered system of equations involving logic and arithmetic operators. The semantics is defined extensionally as a solution to the system of equations, *i.e.*, an assignment of variables to values such that all the equations hold.

Usuba also provides syntactic sugar for declaring lookup tables, useful for specifying S-boxes. RECTANGLE's S-box (`SubColumn`) can thus be written as:

```
table SubColumn (in:v4) returns (out:v4) {
    6 , 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3 , 13, 8, 15, 4, 2
}
```

Conceptually, a lookup table is an array: a n-bit input indexes into an array of 2^n possible output values. However, to maximize throughput and avoid cache timing attacks, the compiler expands lookup tables to Boolean circuits. For prototyping purposes, Usuba uses an elementary logic synthesis algorithm based on binary decision diagrams (BDD)—inspired by Pornin [248]—to perform this expansion. The circuits generated by this tool are hardly optimal: finding optimal representations of S-boxes is a full-time occupation for cryptographers, often involving months of exhaustive search [191, 96, 293, 234]. Usuba integrates these hard-won results into a database of known circuits, which is searched before trying to convert any lookup table to a circuit. For instance, RECTANGLE's S-box (`SubColumn`) is replaced by the compiler with the following node (provided by Zhang et al. [309]):

```
node SubColumn (a:v4) returns (b:v4)
  vars t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12 : v1
  let
    t1   = ~a[1];
    t2   = a[0] & t1;
    t3   = a[2] ^ a[3];
    b[0] = t2 ^ t3;
    t5   = a[3] | t1;
    t6   = a[0] ^ t5;
    b[1] = a[2] ^ t6;
    t8   = a[1] ^ a[2];
    t9   = t3 & t6;
    b[3] = t8 ^ t9;
    t11  = b[0] | t8;
    b[2] = t6 ^ t11
tel
```

Bit-permutations are commonly used in cryptographic algorithms to provide diffusion. Usuba offers syntactic support for declaring bit-permutations. For instance, the initial permutation of DES (Figure 2.1) amounts to the following declaration that specifies which bit of the input bitvector should be routed to the corresponding position of the output bitvector:

```
perm init_p (a:b64) returns (out:b64) {
     58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4,
     62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8,
     57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,
     61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7
}
```

It should be read as "the 1st bit of the output is the 58th bit of the input, the 2nd bit of the output is the 50th of the input, *etc.*". The direct bitsliced translation of this permutation is a function of 64 Boolean inputs and 64 Boolean outputs, which consists of simple assignments: `out[0] = a[58]; out[1] = a[50]; ...`. After Usuba's copy propagation pass (Section 4.2.2), a bit-permutation is thus no more than a (static) renaming of variables.

Finally, Usuba also offers a way to define arrays of nodes, tables or permutations. For instance, the SERPENT [78] cipher uses a different S-box for each round, all of which can be grouped into the same array of tables:

```
table[] sbox(input:v4) returns (out:v4) [
    { 3, 8,15, 1,10, 6, 5,11,14,13, 4, 2, 7, 0, 9,12 } ;
    {15,12, 2, 7, 9, 0, 5,10, 1,11,14, 8, 6,13, 3, 4 } ;
    { 8, 6, 7, 9, 3,12,10,15,13, 1,14, 4, 0,11, 5, 2 } ;
    { 0,15,11, 8,12, 9, 6, 3,13, 1, 2, 4,10, 7, 5,14 } ;
    { 1,15, 8, 3,12, 0,11, 6, 2, 5, 4,10, 9,14, 7,13 } ;
    {15, 5, 2,11, 4,10, 9,12, 0, 3,14, 8,13, 6, 7, 1 } ;
    { 7, 2,12, 5, 8, 4, 6,11,14, 9, 1,15,13, 3,10, 0 } ;
    { 1,13,15, 0,14, 8, 2,11, 7, 4,12,10, 9, 3, 5, 6 }
]
```

Calling, for instance, the 3rd S-box of this array can be done using the syntax `x = sbox<3>(y)`.

Vectors. Syntactically, our treatment of vectors stems from the work on hardware synthesis of synchronous dataflow programs [268]. Given a vector x of size n, we can obtain the element $x[k]$ at position $0 \leq k < n$, the consecutive elements $x[k..l]$ in the range $0 \leq k < l < n$ and a slice of elements $x[l]$ where l is a list of integers $k_0, ..., k_J$ respecting $0 \leq k_1 < n, ..., 0 \leq k_J < n$. This syntax is instrumental for writing concise bit-twiddling code. Indices must be known at compile-time, since variable indices could compromise the constant-time property of the code. The type-checker can therefore prevent out-of-bounds accesses. Noticeably, vectors are maintained in a flattened form. Given two vectors x and y of size, respectively, m and n, the variable $z = (x, y)$ is itself a vector of size $m + n$ (not a pair of vectors). Conversely, for a vector u of size $m + n$, the equation $(x, y) = u$ stands for $x = u[0..n]$ and $y = u[n..m + n]$.

Loops. To account for repetitive definitions (such as the wiring of the 25 rounds of RECTANGLE), the `forall` construct lets us declare a group of equations within static bounds. Its semantics is intuitively defined by macro-expansion: we can always translate it into a chunk of interdependent equations. In practice, Usuba's compiler preserves this structure in its pipeline (Section 4.1.3). SERPENT's main loop for instance is:

```
state[0] = plaintext;
forall i in [0,30] {
    state[i+1] = linear_layer(sbox<i%8>(state[i] ^ keys[i]))
}
```

Each of the 31 iteration computes a round of SERPENT: it `xor`s the result of the previous round (`state[i]`) with the key for this round (`keys[i]`), then calls the S-box for this round (`sbox<i%8>`), and finally calls the node `linear_layer` and stores the result in `state[i+1]` for the next round.

Imperative assignment. Some ciphers (*e.g.*, CHACHA20, SERPENT) are defined in an imperative manner, repetitively updating a local state. Writing those ciphers in a synchronous dataflow language can be tedious: it amounts to writing code in static single assignment (SSA) form. To simplify such programs, Usuba provides an assignment operator `x := e` (where `x` may appear free in `e`). It desugars into a standard equation with a fresh variable on the left-hand side that is substituted for the updated state in later equations. For instance, SERPENT's linear layer can be written as:

```
node linear_layer(x:u32x4) returns (out:u32x4)
let
    x[0] := x[0] <<< 13;
    x[2] := x[2] <<< 3;
    x[1] := x[1] ^ x[0] ^ x[2];
    x[3] := x[3] ^ x[2] ^ (x[0] << 3);
    x[1] := x[1] <<< 1;
    x[3] := x[3] <<< 7;
    x[0] := x[0] ^ x[1] ^ x[3];
    x[2] := x[2] ^ x[3] ^ (x[1] << 7);
    x[0] := x[0] <<< 5;
    x[2] := x[2] <<< 22;
    out = x
tel
```

which desugars to:

```
node linear_layer(x:u32x4) returns (out:u32x4)
vars t0, t1, t2, t3, t4, t5, t6, t7, t8, t9: u32
let
    t0 = x[0] <<< 13;
    t1 = x[2] <<< 3;
    t2 = x[1] ^ t0 ^ t1;
    t3 = x[3] ^ t1 ^ (t0 << 3);
    t4 = t2 <<< 1;
    t5 = t3 <<< 7;
    t6 = t0 ^ t4 ^ t5;
    t7 = t1 ^ t5 ^ (t4 << 7);
    t8 = t6 <<< 5;
    t9 = t7 <<< 22;
    out = (t8, t9, t4, t5);
tel
```

Operators. The constructs introduced so far deal with the wiring and structure of the dataflow graph. To compute, one must introduce operators. Usuba supports bitwise logical operators (conjunction `&`, disjunction `|`, exclusive-or `^` and negation `~`), arithmetic operators (addition `+`, multiplication `*` and subtraction `-`), shifts (left `<<` and right `>>`), and rotations (left `<<<` and right `>>>`). Additionally, a `shuffle` operator performs intra-register bit shuffling. A `pack` operator packs two values of type ui and uj into a value of type uk such that $i + j = k$. Finally, a `bitmask` extract a mask made of zeros or ones depending on the value of the n^{th} bit of an expression. For instance, `bitmask(2,1) = 0xffff`, and `bitmask(2,0) = 0` (for 16-bit values). The availability and exact implementation (especially, run-time cost) of the operators depend on the slicing mode and the target architecture.

2.3 Types

Base types. For the purpose of interacting with its cryptographic runtime, the interface of a block cipher is specified in terms of the matricial type $\mathtt{u}_D m \times n$, which documents the layout of blocks coming in and out of the cipher. A tuple of n elements of types $\tau_1, ..., \tau_n$ is of type $(\tau_1, ..., \tau_n)$. We also have general vectors whose types are $\tau[n]$, for any type τ and n a (static) natural number. Consider, for example, the subkeys used by $v$sliced RECTANGLE: they are presented as an object `key` of type $\mathtt{u}_V 16 \times 4[26]$, which is 26 quadruples of 16-bit words. The notation $\mathtt{u}_D m \times 4[26]$ indicates that accesses must be performed in column-major order, *i.e.*, as if we were accessing an object of type $\mathtt{u}_D m[26][4]$ (following C conventions). This apparent redundancy is explained by the fact that types serve two purposes. In the surface language, matricial types ($\mathtt{u}_D m \times 4$) document the data layout. In the target language, the matricial structure is irrelevant: an object of type $\mathtt{u}_D m \times 4[26]$ supports exactly the same operations as an object of type $\mathtt{u}_D m[26][4]$. Surface types are thus normalized, after type-checking, into *distilled* types. A value of type $\mathtt{u}_D m \times 1$ (for any D and m) is called an atom, and its type is said to be atomic.

Parametric polymorphism. A final addition to our language of types is the notion of parametric word size and parametric direction. A cipher like RECTANGLE can in fact be sliced horizontally or vertically: both modes of operation are compatible with the various SIMD architectures introduced after SSSE3. Similarly, the node `SubColumn` amounts to a Boolean circuit whose operations (`&`, `|`, `^` and `~`) are defined for any atomic word size (ranging from a single Boolean to a 512-bit AVX512 register): `SubColumn` thus applies to $\mathtt{u}_D 1$, $\mathtt{u}_D 8$, *etc.*

Nodes being first-order functions, a function type is (at most) rank-1 polymorphic: the polymorphic parameters it may depend on are universally quantified over the whole type (and node body). We use the abbreviation $\mathtt{b}n$ (n **b**its) and $\mathtt{u}m$ (**u**nsigned integer of size m) for the type $\mathtt{u}_{`D}1 \times n$ and, respectively, $\mathtt{u}_{`D}m$ where $`D$ is the direction parameter in the nearest scope. Similarly, we write $\mathtt{v}n$ for $\mathtt{u}_{`D}`m \times n$ when $`m$ is the nearest word size parameter.

When compiling a program whose main is direction polymorphic, the flag `-V` or `-H` must be passed to the compiler to instruct it to specialize the direction for *v*slicing or *h*slicing. Similarly, if the main is word size polymorphic, the flag `-m` *m* must be passed to the compiler to specialize the word size of the main to *m* bits.

Example 2.3.1 (Polymorphic direction)**.** This `ShiftRows` node of RECTANGLE is defined as follows:

```
node ShiftRows (input:u16[4]) returns (out:u16[4])
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

We leave its direction polymorphic (u16 is shorthand for $\mathtt{u}_{`D}16$), as the left-rotation on 16-bit words is defined for both *h*slicing and *v*slicing.

Example 2.3.2 (Polymorphic word size)**.** The CHACHA20 cipher relies on a node that computes additions, `xor`s, and left rotations:

```
node QR (a,b,c,d:u<V>m)
    returns (aR,bR,cR,dR:u<V>m)
let
    a := a + b;
    d := (d ^ a) <<< 16;
    c := c + d;
    b := (b ^ c) <<< 12;
    aR = a + b;
    dR = (d ^ aR) <<< 8;
    cR = c + dR;
    bR = (b ^ cR) <<< 7;
tel
```

Since additions can only be used in vertical slicing, we can explicitely type this node with the slicing direction V. However, the additions and left-rotations are defined for 8-bit, 16-bit, 32-bit and 64-bit integers, and `xor`s are defined for any word size. The word size can thus be left polymorphic. However, note that the CHACHA20 cipher itself is not word-size polymorphic: its specification explicitly uses 32-bit integers, and the Usuba implementation (Figure 2.10, Page 66) thus uses $\mathtt{u}_V32$ values.

Logic(τ) :	Arith(τ) :	Shift$_n[\tau]$:
$\&: \tau \to \tau \to \tau$	$+: \tau \to \tau \to \tau$	$\texttt{>>}_n: \tau \to \tau$
$\vert: \tau \to \tau \to \tau$	$\star: \tau \to \tau \to \tau$	$\texttt{<<}_n: \tau \to \tau$
$\hat{}: \tau \to \tau \to \tau$	$-: \tau \to \tau \to \tau$	$\texttt{>>>}_n: \tau \to \tau$
$\tilde{}: \tau \to \tau$		$\texttt{<<<}_n: \tau \to \tau$
Shuffle$_l[\tau]$:	Bitmask$_n[\tau]$:	Pack(τ_1, τ_2, τ_3) :
$\texttt{shuffle}_l : \tau \to \tau$	$\texttt{bitmask}_n : \tau \to \tau$	$\texttt{pack} : \tau_1 \to \tau_2 \to \tau_3$

Figure 2.4: Usuba's type-classes

Example 2.3.3 (Polymorphic word size and direction)**.** This `SubColumn` node of RECTANGLE is defined as follows:

```
node SubColumn (a:v4) returns (b:v4)
  vars t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12 : v1
  let
    t1   = ~a[1];
    t2   = a[0] & t1;
    t3   = a[2] ^ a[3];
    b[0] = t2 ^ t3;
    t5   = a[3] | t1;
    t6   = a[0] ^ t5;
    b[1] = a[2] ^ t6;
    t8   = a[1] ^ a[2];
    t9   = t3 & t6;
    b[3] = t8 ^ t9;
    t11  = b[0] | t8;
    b[2] = t6 ^ t11
tel
```

We leave both its word size and direction polymorphic (v4 is shorthand for $\mathtt{u}`\mathcal{D}`m \times 4$), as it contains only bitwise operations, which are defined for any word size and slicing direction.

Ad-hoc polymorphism. In reality, few programs are defined for any word size or any direction. Also, no program is absolutely parametric in the word size: we can only compute up to the register size of the underlying architecture.

To capture these invariants, we introduce a form of bounded polymorphism through type-classes [296]. Whether a given cipher can be implemented over a collection of word size and/or direction is determined by the availability of logical and arithmetic operators. We therefore introduce six type-classes for logical, arithmetic and shift/rotate operations, as well as shuffles, bitmasks and packs (Figure 2.4)

Shifts are parameterized by a natural number describing the amount to shift by. For instance, the operation `x << 4` is represented as $\texttt{<<}_4$ `x`. Similarly, bitmasks are parameterized by a natural number corresponding to the bit to extract from the argument, and shuffles are parameterized by a list of natural numbers representing the pattern to use for the shuffle.

The implementation of the type classes depends on the type considered and the target architecture. An example is arithmetic on 13-bit words, which is impossible, even in vertical mode. Another example is the shift operation: shifting a tuple amounts to renaming registers whereas shifting a scalar in horizontal mode requires an actual shuffle instruction. We chose to provide instances solely for operations that can be implemented statically or with a handful of instructions, with the exceptions of `shuffle`, which is

Table 2.1: Operator instances.

Class	Instances	Architecture	Compiled with
$\mathsf{Logic}(\tau)$	$\mathsf{Logic}(\tau) \Rightarrow \mathsf{Logic}(\tau[n])$ for $n \in \mathbb{N}$	all	homomorphic application (n instr.)
	$\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [1, 64]$ $\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [65, 128]$ $\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [129, 256]$ $\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [257, 512]$	$\geq$ x86-64 $\geq$ SSE $\geq$ AVX $\geq$ AVX512	`and`, `or`, *etc.* (1 instr.)
$\mathsf{Arith}(\tau)$	$\mathsf{Arith}(\tau) \Rightarrow \mathsf{Arith}(\tau[n])$ for $n \in \mathbb{N}$	all	homomorphic application (n instr.)
	$\mathsf{Arith}(\mathsf{u}_V 8)$ $\mathsf{Arith}(\mathsf{u}_V 16)$ $\mathsf{Arith}(\mathsf{u}_V 32)$ $\mathsf{Arith}(\mathsf{u}_V 64)$	all	`add`, `vpadd`, `vpsub`, *etc.* (1 instr.)
$\mathsf{Shift}_n[\tau]$	$\mathsf{Shift}_n[\tau[m]]$ for $m \in \mathbb{N}, 0 \leq n \leq m$	all	variable renaming (0 instr.)
	$\mathsf{Shift}_n[\mathsf{u}_V{'m}]$, $\mathsf{Shift}_n[\mathsf{u}_H{'m}]$ $\Rightarrow \mathsf{Shift}_n[\mathsf{u}_D{'m}]$	all	depends of instance
	$\mathsf{Shift}_n[\mathsf{u}_V 16]$ for $n \in [0, 15]$ $\mathsf{Shift}_n[\mathsf{u}_V 32]$ for $n \in [0, 31]$ $\mathsf{Shift}_n[\mathsf{u}_V 64]$ for $n \in [0, 63]$	all	`vpsrl`/`vpsll` ($\leq$ 3 instr.)
	$\mathsf{Shift}_n[\mathsf{u}_H 2]$ for $n \in [0, 1]$ $\mathsf{Shift}_n[\mathsf{u}_H 4]$ for $n \in [0, 3]$ $\mathsf{Shift}_n[\mathsf{u}_H 8]$ for $n \in [0, 7]$ $\mathsf{Shift}_n[\mathsf{u}_H 16]$ for $n \in [0, 15]$	$\geq$ SSE	`vpshuf` (1 instr.)
	$\mathsf{Shift}_n[\mathsf{u}_H 32]$ for $n \in [0, 31]$ $\mathsf{Shift}_n[\mathsf{u}_H 64]$ for $n \in [0, 63]$	$\geq$ AVX512	
$\mathsf{Shuffle}_l[\tau]$	$\forall D, m, l, \mathsf{Shuffle}_l[\mathsf{u}_D m] \implies \lvert l \rvert = m \wedge \forall n \in l, 0 \leq n < m$		
	$\mathsf{Shuffle}_l[\mathsf{u}_V 8]$	x86-64	`vpsrl`/`vpsll`/`vpand`/`vpxor` (many instr.)
	$\mathsf{Shuffle}_l[\mathsf{u}_V 16]$ $\mathsf{Shuffle}_l[\mathsf{u}_V 32]$ $\mathsf{Shuffle}_l[\mathsf{u}_V 64]$	all	
	$\mathsf{Shuffle}_l[\mathsf{u}_H 2]$ $\mathsf{Shuffle}_l[\mathsf{u}_H 4]$ $\mathsf{Shuffle}_l[\mathsf{u}_H 8]$ $\mathsf{Shuffle}_l[\mathsf{u}_H 16]$	$\geq$ SSE	`vpshuf` (1 instr.)
	$\mathsf{Shuffle}_l[\mathsf{u}_H 32]$ $\mathsf{Shuffle}_l[\mathsf{u}_H 64]$	$\geq$ AVX512	
$\mathsf{Bitmask}_n[\tau]$	$\mathsf{Bitmask}_n[\mathsf{u}_V 8]$ for $n \in [0, 7]$ $\mathsf{Bitmask}_n[\mathsf{u}_V 16]$ for $n \in [0, 15]$ $\mathsf{Bitmask}_n[\mathsf{u}_V 32]$ for $n \in [0, 31]$ $\mathsf{Bitmask}_n[\mathsf{u}_V 64]$ for $n \in [0, 63]$	all	`srl+and+sub` `vpsrl+vpand+vpsub` *etc.* (3 instr.)
$\mathsf{Pack}(\tau_1, \tau_2, \tau_3)$	$\mathsf{Pack}(\mathsf{u}_V i, \mathsf{u}_V j, \mathsf{u}_V k)$ $i + j = k \wedge k \leq 64$	all	`sll+or`/`vpsll+vpor` *etc.* (2 instr.)

made available in vertical mode despite its large cost. Our type-class mechanism is not user-extensible. The list of all the possible type-classes instances is fixed and given in Table 2.1. This set of instances is obviously non-overlapping so our overloading mechanism is coherent: if the type-checker succeeds in finding an instance for the target architecture, then that instance is unique.

Both logical and arithmetic operators can be applied to a tuple of size n, in which case they amount to n element-wise applications of the operator on each element of the tuple. Shifting a tuple, on the other hand, is performed at the granularity of the elements of the tuple, and amounts to statically renaming variables and shifting in zeros. For instance, if we consider a vector `x` of type `b1[4]`, `x << 2` is equivalent to `(x[0], x[1], x[2], x[3]) << 2`, which is reduced at compile time to `(x[2], x[3],0, 0)` (with the last two `0` being of type `b1`).

Logic instructions can be applied on any non-tuple type for any slicing, as long as the architecture offers large enough registers. Arithmetic instructions on non-tuple types are only valid for *v*slicing, and require some support from the underlying architecture. In practice, SSE, AVX, AVX2 and AVX512 offer instructions to compute 8-bit, 16-bit, 32-bit and 64-bit arithmetic operations. Vertical shifts of 16-bit, 32-bit and 64-bit values use CPU shift instructions (`vpsrl` and `vpsll` on AVX2, `shr` and `shl` on general purpose x86). Shifts in horizontal slicing are compiled using shuffle instructions (*e.g.*, `vpshufd`, `vpshufb`). For instance, consider a variable `x` of type `u16`: shifting right this variable by 2 is done using the shuffle pattern `[-1,-1,15,14,13,12,11,10,9,8,7,6,5,4,3,2]` (`-1` in the pattern causes a `0` to be introduced). SIMD registers older than AVX512 only offer shuffles for up to 16 elements, while AVX512 does also provide 32 and 64 elements shuffle, thus allowing us to compile 64-bit shifts. Shuffle instruction in horizontal mode map naturally to SIMD shuffle instructions. On the other hand, shuffle instructions in vertical mode are compiled to shifts moving each bit in a register. For instance, shuffling a variable `x` of type `u8` with the pattern `[5,2,0,1,7,6,4,3]` produces the following code:

```
((x << 5) & 128)  ^
((x << 1) & 64)   ^
((x >> 2) & 32)   ^
((x >> 2) & 16)   ^
((x >> 3) & 8)    ^
((x << 1) & 4)    ^
((x >> 2) & 2)    ^
((x >> 4) & 1)
```

Since SIMD instructions do not offer 8-bit shifts, 8-bit shuffles in vertical mode are only available on x86-64. Shuffles of 16, 32 and 64 bits can, on the other hand, be compiled for SIMD architectures. Shuffles are expensive in vertical mode, but we chose to support them nonetheless since some commonly used ciphers rely on them, such as AES.

An instruction `bitmask(x,n)` is compiled to `-((x >> n) & 1)`, thus leveraging two's complement arithmetic to build the mask: `(x >> n) & 1` is 0 if the n^{th} bit of `x` is 0, and 1 otherwise. Doing -0 returns 0 (*i.e.*, an empty mask), while doing -1 returns an integer full of ones (*i.e.*, `0xff` on 8 bits, `0xffff` on 16 bits *etc.*).

Finally, `pack` is simply compiled to a left-shift of its first operand and combines it with the second one using an `or`. Pack cannot be used to combined values whose total size would be more than 64 bits because the shift instructions provided by the architectures we target operate on at most 64-bit integers (even on SIMD extensions).

We now illustrate how type-classes enable ad-hoc polymorphism on a few examples.

Example 2.3.4.

```
node f(x:u16, y:u16) returns (z:u16)
let
    z = x + y;
tel
```

All parameters of `f` are direction polymorphic. The Arith type-class defines the addition for any type τ (Figure 2.4). During type-checking, a constraint will therefore be added on this node that the architecture must provide a instance of the addition of type $u_{`D}16 \rightarrow u_{`D}16 \rightarrow u_{`D}16$, for some universally-quantified direction $`D$. When specializing this node for a slicing direction (H or V), the type of the parameters becomes u_H16 or u_V16, and the corresponding instance of the addition is looked up in the Arith type-class (Table 2.1). Since Arith does not provide an instance of the addition for *h*slicing, this node cannot be *h*sliced. This node can, however, be *v*sliced, in which case the compilation of the addition will depend on the architecture. For instance, on AVX2, `vpaddw` will be used to compute the (vectorized) 16-bit addition.

Example 2.3.5.

```
node g(a:v1[5], b:u<V>32, c:u<H>8)
  returns (x:v1[5], y:u<V>32, z:u<H>8)
let
    x = a <<< 3;
    y = b <<< 3;
    z = c <<< 3;
tel
```

In this node, `a` and `x` are direction polymorphic and word size polymorphic vectors, while `b` and `y` (resp., `c` and `z`) are 32-bit *v*sliced (resp., 8-bit *h*sliced) words. The Shift type-class defines the left-rotation as having type $\tau \to \tau$, for any type τ (Figure 2.4). The type-checker will thus add as constraints on this node that the architecture must provide 3 instances of left-rotation: one of type $\mathtt{u}_{`D}`m[5] \to \mathtt{u}_{`D}`m[5]$ (for some universally-quantified $`D$ and $`m$), another one of type $\mathtt{u}_V32 \to \mathtt{u}_V32$, and a final one of type $\mathtt{u}_H8 \to \mathtt{u}_H8$. The compiler will then specialize each rotations with the instance provided by the Shift type-class (Table 2.1). The first one is turned into to a tuple assignment (`x = (a[3], a[4], a[0], a[1], a[2])`), regardless of the architecture and the slicing and word-size use to specialize $`D$ and $`m$. The compilation of the second and third one depends on the architecture. On AVX2, for instance, the *v*sliced left-rotation is compiled to a `vpslld`, and the *h*sliced left-rotation is compiled to a `vpshufb`.

2.4 Applications

We implemented 17 ciphers in Usuba: ACE [7], AES [230], ASCON [126], CAMELLIA [21], CLYDE [55], DES [230], GIFT [29], GIMLI [67], PHOTON [153], PRESENT [85], PYJAMASK [151], RECTANGLE [309], SERPENT [78], SKINNY [50], SPONGENT [86], SUBTERRANEAN [122] and XOODOO [120].

> **!** The Usuba implementations of these ciphers are available on GitHub at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/samples/usuba`

In this section, we comment the source code of 5 ciphers (AES, ASCON, ACE, CHACHA20 and SERPENT), taking this opportunity to explain their algorithm. Specifically, we chose these 5 ciphers for the following reasons. AES is by far the most widely used symmetric cipher. CHACHA20 is a commonly used stream cipher, designed to be efficient on SIMD architectures, and is one of the few ciphers relying on additions. SERPENT is one of the few ciphers to use a different S-boxes for each round. Finally, ACE and ASCON are two recent ciphers, candidates to the NIST Lightweight Cryptography (LWC) Standardization competition [227], showing that Usuba applies on 40-year-old ciphers (*e.g.,* DES) as well as on recent ones.

2.4.1 AES

The Advanced Encryption Standard (AES), also known as Rijndael [119], was chosen by the NIST in 2001 to replace DES as the standard for encrypting data. Its Usuba implementation is shown in Figure 2.5.

AES's state is 128-bit wide, represented as 4x4 matrix of 8-bit elements:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

with the original input being 16 bytes in the following order:

$[\ a_{0,0},\ a_{1,0},\ a_{2,0},\ a_{3,0},\ a_{0,1},\ a_{1,1},\ a_{2,1},\ \ldots,\ a_{1,3},\ a_{2,3},\ a_{3,3}\ ]$

However, our AES implementations use a different representation in order to be able to use a bitsliced S-box instead of the table. Thus, the type of the state in Usuba is `u16x8`: each bit of the 8-bit elements is in a different register, and each of the 8 register contains one bit from each of the 16 elements of the state.

AES's rounds are built from 4 basic operations: `SubBytes`, `ShiftRows`, `MixColumn` and `AddRoundKey`. `AddRoundKey` is a simple `xor` between the 128-bit subkey for the round and the state is thus written as `plain ^ key` in the Usuba code. It is expanded into 8 `xor`s during normalization, as per Table 2.1.

`SubBytes`, the S-box, is specified as a multiplicative inversion over the finite field $GF(2^8)$. However, it is traditionally implemented as a lookup table when cache-timing

```
table SubBytes(input:v8) returns (output:v8) {
    99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118,
    ...
    140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22 }

node ShiftRows(inputSR:u16x8) returns (out:u16x8)
let   forall i in [0,7] {
        out[i] = Shuffle(inputSR[i],[  0,  5, 10, 15,
                                       4,  9, 14,  3,
                                       8, 13,  2,  7,
                                      12,  1,  6, 11 ]) }   tel

node RL32(input:u16) returns (out:u16)
let   out = Shuffle(input,[  1,  2,  3,  0,
                             5,  6,  7,  4,
                             9, 10, 11,  8,
                            13, 14, 15, 12 ])   tel

node RL64(input:u16) returns (out:u16)
let   out = Shuffle(input,[  2,  3,  0,  1,
                             6,  7,  4,  5,
                            10, 11,  8,  9,
                            14, 15, 12, 13 ])   tel

node MixColumn(a:u16x8) returns (b:u16x8)
let
    b[7] = a[0] ^ RL32(a[0]) ^ RL32(a[7]) ^ RL64(a[7] ^ RL32(a[7]));
    b[6] = a[7] ^ RL32(a[7]) ^ a[0] ^ RL32(a[0]) ^ RL32(a[6]) ^ RL64(a[6] ^ RL32(a[6]));
    b[5] = a[6] ^ RL32(a[6]) ^ RL32(a[5]) ^ RL64(a[5] ^ RL32(a[5]));
    b[4] = a[5] ^ RL32(a[5]) ^ a[0] ^ RL32(a[0]) ^ RL32(a[4]) ^ RL64(a[4] ^ RL32(a[4]));
    b[3] = a[4] ^ RL32(a[4]) ^ a[0] ^ RL32(a[0]) ^ RL32(a[3]) ^ RL64(a[3] ^ RL32(a[3]));
    b[2] = a[3] ^ RL32(a[3]) ^ RL32(a[2]) ^ RL64(a[2] ^ RL32(a[2]));
    b[1] = a[2] ^ RL32(a[2]) ^ RL32(a[1]) ^ RL64(a[1] ^ RL32(a[1]));
    b[0] = a[1] ^ RL32(a[1]) ^ RL32(a[0]) ^ RL64(a[0] ^ RL32(a[0]))   tel

node AddRoundKey(plain,key:u16x8) returns (xored:u16x8)
let     xored = plain ^ key    tel


node AES_round(prev:u16x8,key:u16x8) returns (next:u16x8)
let   next = AddRoundKey(MixColumn(ShiftRows(SubBytes(prev))),key)   tel

node AES128(plain:u16x8,key:u16x8[11]) returns (cipher:u16x8)
vars state : u16x8
let
    state = AddRoundKey(plain, key[0]);

    forall i in [1,9] {
      state := AES_round(state, key[i])
    }

    cipher = AddRoundKey(ShiftRows(SubBytes(state)), key[10])
tel
```

Figure 2.5: Usuba implementation of AES

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \begin{matrix} \longrightarrow \lll 0 \longrightarrow \\ \longrightarrow \lll 1 \longrightarrow \\ \longrightarrow \lll 2 \longrightarrow \\ \longrightarrow \lll 3 \longrightarrow \end{matrix} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix}$$

(a) Reference description of `ShiftRows`

$$\begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix} \begin{matrix} \longrightarrow \lll 0 \longrightarrow \\ \longrightarrow \lll 1 \longrightarrow \\ \longrightarrow \lll 2 \longrightarrow \\ \longrightarrow \lll 3 \longrightarrow \end{matrix} \begin{pmatrix} 0 & 4 & 8 & 12 \\ 5 & 9 & 13 & 1 \\ 10 & 14 & 2 & 6 \\ 15 & 3 & 7 & 11 \end{pmatrix}$$

(b) `ShiftRows` with Usuba's *h*slice layout

Figure 2.6: AES's `ShiftRows` step

attacks are not a concern. A lot of work has been done to find efficient circuit implementations of this S-box [220, 302, 261, 96, 90, 91]. We incorporated the shortest known circuit (generated by Cagdas Calik [92]) into Usubac's circuit database. The `SubBytes` table is thus expanded to a node containing 113 Boolean operations.

`ShiftRows` left-rotates the second row of AES's state matrix by one, the third row by two and the fourth row by three (Figure 2.6a). However, with our representation of the state, each register contains one bit from each element of the matrix. `ShiftRows` can thus be done by shuffling each register. To find the pattern to use for the shuffle, we can number the bytes from the state from 0 to 15 and apply the rotations (Figure 2.6b). The pattern to use in `ShiftRows` is thus:

```
[  0,  5, 10, 15,
   4,  9, 14,  3,
   8, 13,  2,  7,
  12,  1,  6, 11 ]
```

Finally, `MixColumns` (Figure 2.2, Page 50), multiplies the state by a constant matrix. Käsper and Schwabe ([174], Appendix A) showed how to derive the equations to compute this multiplication on a sliced state, which we reused in our implementation. Since this multiplication mixes bytes from the same column, which are stored in the same registers, it requires left rotations by one and two in order for them to interact together. However, since each register contains bytes from 4 different columns, the rotations are applied on each of those 4 groups at once using `Shuffle`s (`RL32` to rotate by one and `RL64` to rotate by two).

2.4.2 ASCON

The ASCON cipher suite [126, 128] was designed for the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [63]. It was later submitted to the NIST LWC competition as well. It provides several ciphers (ASCON-128, ASCON-128a) and hash functions (ASCON-Hash) as well as other cryptographic functions (ASCON-Xof). All of those primitives rely on the same 320-bit permutation. We

```
node AddConstant(state:u64x5,c:u64) returns (stateR:u64x5)
let
    stateR = (state[0,1], state[2] ^ c, state[3,4]);
tel

table Sbox(x:v5) returns (y:v5) {
  0x4,  0xb,  0x1f, 0x14, 0x1a, 0x15, 0x9,  0x2,
  0x1b, 0x5,  0x8,  0x12, 0x1d, 0x3,  0x6,  0x1c,
  0x1e, 0x13, 0x7,  0xe,  0x0,  0xd,  0x11, 0x18,
  0x10, 0xc,  0x1,  0x19, 0x16, 0xa,  0xf,  0x17
}

node LinearLayer(state:u64x5) returns (stateR:u64x5)
let
  stateR[0] = state[0] ^ (state[0] >>> 19) ^ (state[0] >>> 28);
  stateR[1] = state[1] ^ (state[1] >>> 61) ^ (state[1] >>> 39);
  stateR[2] = state[2] ^ (state[2] >>> 1)  ^ (state[2] >>> 6);
  stateR[3] = state[3] ^ (state[3] >>> 10) ^ (state[3] >>> 17);
  stateR[4] = state[4] ^ (state[4] >>> 7)  ^ (state[4] >>> 41);
tel

node ascon12(input:u64x5) returns (output:u64x5)
vars
    consts:u64[12],
    state:u64x5
let
    consts = (0xf0, 0xe1, 0xd2, 0xc3, 0xb4, 0xa5,
              0x96, 0x87, 0x78, 0x69, 0x5a, 0x4b);

    state = input;
    forall i in [0, 11] {
        state := LinearLayer(Sbox(AddConstant(state,consts[i])))
    }
    output = state
tel
```

Figure 2.7: Usuba implementation of ASCON

focus on this permutation in Usuba, which—for simplicity—we shall call ASCON in the following, rather than "the ASCON permutation".

ASCON represents its 320-bit input as 5 64-bit registers (the *state*), on which it applies n rounds. For the NIST submission, the recommended number of rounds is 12. Each round consists of three steps: a constant addition, a substitution layer, and a linear layer. Figure 2.7 shows the Usuba implementation of ASCON.

The constant addition `AddConstant` consists in `xor`ing the third register of the state with a constant. The constant is different at each round.

The substitution layer applies 64 times a 5×5 `Sbox`. It is meant to be implemented in $v$slice form. ASCON's author provided an implementation using 22 bitwise instructions, into which the `table` above is expanded. In $v$slicing mode, its inputs and outputs are monomorphized from `v5` (which is shorthand for u_{D}‘$m \times 5$) to `u<V>64x5`, and it thus computes 64 S-boxes at once.

Finally, the linear layer rotates and `xor`s the 5 registers of the state, which can be very naturally written in Usuba using right-rotations `>>>` and exclusive `or`s `^`.

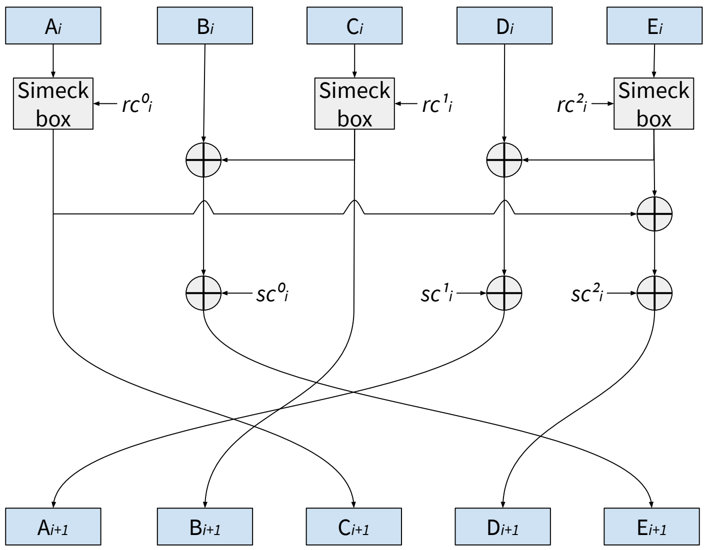


Figure 2.8: ACE's step function

2.4.3 ACE

ACE [7] is another 320-bit permutation submitted to the NIST LWC competition. An authenticated cipher and a hash algorithm are derived from this permutation. In Usuba, we shall focus on the permutation, whose implementation is shown in Figure 2.9.

ACE iterates a step function (`ACE_step`) 16 times on a state of 5 64-bit registers. The step function can be represented by the circuit from Figure 2.8. It performs 3 calls to the function `simeck_box`:

```
A := simeck_box(A,RC[0]);
C := simeck_box(C,RC[1]);
E := simeck_box(E,RC[2]);
```

Then `xor`s the 3 round constants (`SC`) with the state and the elements of the states between them:

```
B := B ^ C ^ (0,SC[0]) ^ (0xffffffff,0xffffff00);
D := D ^ E ^ (0,SC[1]) ^ (0xffffffff,0xffffff00);
E := E ^ A ^ (0,SC[2]) ^ (0xffffffff,0xffffff00);
```

And finally shuffles each of the five elements of the state:

```
(Ar, Br, Cr, Dr, Er) = (D, C, A, E, B);
```

The so-called Simeck Box is built by iterating 8 times the round function of the Simeck cipher [306] (`f` in the code) mixed with a constant addition. Since this function operates on 32-bit values rather than 64-bit ones, we used the type `u32x2` for the values manipulated by ACE rather than `u64`.

ACE uses two sets of constants: `RC` is used to provide constants to Simeck's round function, while `SC`'s values are directly `xor`ed with the state. `RC` and `SC` are both defined as two-dimensional vectors: they each contain three vectors corresponding to rc^0 (resp., sc^0), rc^1 (resp., sc^1) and rc^2 (resp., rc_2). For each round, the i^{th} value of each of the three sub-vectors are extracted using a slice `RC[0,1,2][i]` (resp., `SC[0,1,2][i]`). Those slices are expanded by Usubac into respectively `(RC[0][i], RC[1][i], RC[2][i])` and `(SC[0][i], SC[1][i], SC[2][i])`, but enable a more concise description of the algorithm.

In practice, `SC`'s values are not used as is, but are first padded with ones instead of

```
node f(x:u32) returns (y:u32)
let
    y = ((x <<< 5) & x) ^ (x <<< 1)
tel

node simeck_box(input:u32x2, rc:u32) returns (output:u32x2)
vars   state:u32x2
let
    state = input;
    forall i in [0, 7] {
      state := ( f(state[0]) ^ state[1] ^
                 0xfffffffe ^ ((rc >> i) & 1),
                 state[0] ) ;
    }
    output = state
tel


node ACE_step(A,B,C,D,E:u32x2,RC,SC:u32[3]) returns (Ar,Br,Cr,Dr,Er:u32x2)
let
    A := simeck_box(A,RC[0]);
    C := simeck_box(C,RC[1]);
    E := simeck_box(E,RC[2]);
    B := B ^ C ^ (0,SC[0]) ^ (0xffffffff,0xffffff00);
    D := D ^ E ^ (0,SC[1]) ^ (0xffffffff,0xffffff00);
    E := E ^ A ^ (0,SC[2]) ^ (0xffffffff,0xffffff00);
    (Ar, Br, Cr, Dr, Er) = (D, C, A, E, B);
tel


node ACE(input:u32x2[5]) returns (output:u32x2[5])
vars   SC:u32[3][16],
       RC:u32[3][16],
       state:u32x2[5]
let
    SC = (0x50,0x5c,0x91,0x8d,0x53,0x60,0x68,0xe1,0xf6,0x9d,0x40,0x4f,0xbe,0x5b,0xe9,0x7f,
          0x28,0xae,0x48,0xc6,0xa9,0x30,0x34,0x70,0x7b,0xce,0x20,0x27,0x5f,0xad,0x74,0x3f,
          0x14,0x57,0x24,0x63,0x54,0x18,0x9a,0x38,0xbd,0x67,0x10,0x13,0x2f,0xd6,0xba,0x1f);
    RC = (0x07,0x0a,0x9b,0xe0,0xd1,0x1a,0x22,0xf7,0x62,0x96,0x71,0xaa,0x2b,0xe9,0xcf,0xb7,
          0x53,0x5d,0x49,0x7f,0xbe,0x1d,0x28,0x6c,0x82,0x47,0x6b,0x88,0xdc,0x8b,0x59,0xc6,
          0x43,0xe4,0x5e,0xcc,0x32,0x4e,0x75,0x25,0xfd,0xf9,0x76,0xa0,0xb0,0x09,0x1e,0xad);
    state = input;

    forall i in [0, 15] {
        state = ACE_step(state, RC[0,1,2][i],SC[0,1,2][i]);
    }

    output = state;
tel
```

Figure 2.9: Usuba implementation of ACE

zeros. Put otherwise, SC should contain `0xffffffffffffff50`, `0xffffffffffffff5c` *etc.* instead of `0x50`, `0xff` *etc.* This trick is used in the reference implementation of ACE, which we followed in our Usuba implementation.

```
node QR (a,b,c,d:u<V>32)
     returns (aR,bR,cR,dR:u<V>32)
let
    a := a + b;
    d := (d ^ a) <<< 16;
    c := c + d;
    b := (b ^ c) <<< 12;
    aR = a + b;
    dR = (d ^ aR) <<< 8;
    cR = c + dR;
    bR = (b ^ cR) <<< 7;
tel

node DR (state:u<V>32x16) returns (stateR:u<V>32x16)
let
    state[0,4,8,12]  := QR(state[0,4,8,12]);
    state[1,5,9,13]  := QR(state[1,5,9,13]);
    state[2,6,10,14] := QR(state[2,6,10,14]);
    state[3,7,11,15] := QR(state[3,7,11,15]);

    stateR[0,5,10,15] = QR(state[0,5,10,15]);
    stateR[1,6,11,12] = QR(state[1,6,11,12]);
    stateR[2,7,8,13]  = QR(state[2,7,8,13]);
    stateR[3,4,9,14]  = QR(state[3,4,9,14]);
tel

node Chacha20 (plain:u<V>32x16) returns (cipher:u<V>32x16)
vars state : u<V>32x16
let
    state = plain;
    forall i in [1,10] {
      state := DR(state)
    }
    cipher = state + plain
tel
```

Figure 2.10: Usuba implementation of CHACHA20

2.4.4 CHACHA20

CHACHA [61] is a family of stream ciphers derived from Salsa [62]. Three variants are recommended by its author, D. J. Bernstein: CHACHA8, CHACHA12 and CHACHA20, depending on the security level required. Those ciphers differ only in their number of rounds: 8, 12 and 20. The Usuba implementation of CHACHA20 is shown in Figure 2.10.

CHACHA20's state is 64 bytes divided in 16 32-bit registers, which corresponds to the Usuba type `u<V>32x16`. Since CHACHA20 relies on 32-bit additions, it can only be sliced vertically. We document this specificity by annotating its types with the explicit direction V.

CHACHA20 is made of 10 "double rounds" (DR), each of them relying on "quarter rounds" (QR). The quarter round updates 4 elements of the state using additions, `xor`s and left rotations. Since CHACHA is specified in an imperative manner, by constantly updating its state, we use the imperative assignment operator `:=` to be as close to the specification as possible.

```
table[] sbox(input:v4) returns (out:v4) [
    { 3, 8,15, 1,10, 6, 5,11,14,13, 4, 2, 7, 0, 9,12 } ;
    {15,12, 2, 7, 9, 0, 5,10, 1,11,14, 8, 6,13, 3, 4 } ;
    { 8, 6, 7, 9, 3,12,10,15,13, 1,14, 4, 0,11, 5, 2 } ;
    { 0,15,11, 8,12, 9, 6, 3,13, 1, 2, 4,10, 7, 5,14 } ;
    { 1,15, 8, 3,12, 0,11, 6, 2, 5, 4,10, 9,14, 7,13 } ;
    {15, 5, 2,11, 4,10, 9,12, 0, 3,14, 8,13, 6, 7, 1 } ;
    { 7, 2,12, 5, 8, 4, 6,11,14, 9, 1,15,13, 3,10, 0 } ;
    { 1,13,15, 0,14, 8, 2,11, 7, 4,12,10, 9, 3, 5, 6 }
]

node linear_layer(x:u32x4) returns (out:u32x4)
let
    x[0] := x[0] <<< 13;
    x[2] := x[2] <<< 3;
    x[1] := x[1] ^ x[0] ^ x[2];
    x[3] := x[3] ^ x[2] ^ (x[0] << 3);
    x[1] := x[1] <<< 1;
    x[3] := x[3] <<< 7;
    x[0] := x[0] ^ x[1] ^ x[3];
    x[2] := x[2] ^ x[3] ^ (x[1] << 7);
    x[0] := x[0] <<< 5;
    x[2] := x[2] <<< 22;
    out = x
tel

node Serpent(plaintext:u32x4, keys:u32x4[33]) returns (ciphertext:u32x4)
vars  state : u32x4
let
    state = plaintext;

    forall i in [0,30] {
        state := linear_layer(sbox<i%8>(state ^ keys[i]))
    }

    ciphertext = sbox<7>(state ^ keys[31]) ^ keys[32]
tel
```

Figure 2.11: Usuba implementation of SERPENT

2.4.5 SERPENT

SERPENT [78] was another finalist of the AES competition, where it finished in second place behind Rijndael. It reuses parts of DES's S-boxes, which are known to be very resilient to differential cryptanalysis [74]. The Usuba implementation of SERPENT is shown in Figure 2.11.

SERPENT uses a 128-bit plaintext, represented as 4 32-bit registers (`u32x4` in Usuba). 32 rounds are applied to this plaintext, each one build from an S-box, a linear layer, and key addition (inlined in our implementation using a `xor`).

Each round uses one of 8 S-boxes in a round-robin way. We thus put the S-boxes in an array of tables (`table[] sbox`), and access them within the main loop using the syntax `sbox<index>`, where `index` is the round number modulo 8. Circuits to compute those S-boxes were provided in the AES submission, but better ones (requiring fewer registers) were later computed by Osvik [234]. We incorporated the latter in Usubac's tables database, after verifying that they are still faster on modern Intel CPUs.

The linear layer (`linear_layer`) is made of left rotations, left shifts and `xor`s. It repetitively updates the state's 4 registers, which is very naturally expressed in Usuba using the imperative assignment operator `:=`.

2.5 Conclusion

In this chapter, we described the Usuba programming language, which provides high-level constructions (*e.g.*, nodes, loops, vectors, tables) to specify symmetric block ciphers. Usuba abstracts away both slicing (through polymorphic types) and SIMD parallelization: an Usuba program can be compiled for any architecture that supports its operations.

Using several examples, we showed how Usuba can be used to specify a wide range of ciphers that were designed with slicing in mind (*e.g.*, SERPENT) or not (*e.g.*, AES), that rely on arithmetic operations (*e.g.*, CHACHA20) or not (*e.g.*, SERPENT), that are recent (*e.g.*, ASCON) or not (*e.g.*, AES).

2.5.1 Related Work

Languages for Cryptography

Cryptographic libraries have traditionally been implemented in low-level programming languages, such as C (*e.g.*, Linux Kernel [2], Sodium [1], OpenSSL [3], Crypto++ [123]). However, recently, some effort to use higher-level general-purpose languages have been undertaken in order to provide stronger guarantees (*e.g.*, memory safety). For instance, the TLS implementation nqsbTLS [170], written in OCaml, and MITLS [70], written in F# and F*, both benefit from the ability of OCaml and F# to prevent out-of-bound array indexing. Going even further, several languages have been designed to bring stronger guarantees, higher-level constructions and better performance to cryptographic implementations. We review the most relevant ones in the following, focusing on languages targeting cryptographic primitives, and excluding languages to write protocols (*e.g.*, CPPL [154]), which we consider out of scope.

Cryptol [201] started from the observation that due to the lack of standard for specifying cryptographic algorithms, papers described their ciphers using a combination of English (which Cryptol's authors deem "ambiguous" [201]) and pseudo-code (which tends to "obscure the underlying mathematics" [201]) while providing reference implementations in C ("far too low-level" [201]). Thus, Cryptol is a programming language for specifying cryptographic algorithms, from protocols and modes of operations down to primitives. It covers a broader range than Usuba, which focuses solely on primitives.

Usuba's design and abstractions were driven by existing CPU architectures, but still aims at providing a semantics abstract enough to allow reasoning on combinational circuits. Often, ciphers are described at this level of abstraction, but not always: AES, for instance, is specified in terms of operations in a finite field. Cryptol handles well this type of cipher, by providing high-level mathematical abstractions, even when they do not trivially map to hardware. As such, Cryptol natively supports polynomials and field arithmetic, allowing it to express naturally ciphers like AES. For instance, AES's multiplication in $GF(2^8)$ modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ can be written in Cryptol as:

```
irreducible = <| x^^8 + x^^4 + x^^3 + x + 1 |>

gf28Mult : (GF28, GF28) -> GF28
gf28Mult (x, y) = pmod(pmult x y) irreducible
```

Cryptol basic types are bitvectors, similar to Usuba's un types, which can be grouped in tuples, similar to what Usuba offers. However, where tuples are at the core of Usuba's programs, Cryptol's main construction is the sequence, which, unlike a tuple, only contains elements of the same type. Several operators allow to manipulate sequences: comprehensions, enumerations, infinite sequences, indexing and appending operators. Furthermore, bitvectors can be specified using boolean sequences. For instance, the expression `[True, False, True, False, True, False]` constructs the integer 42, and the indexing operation `@` can be used with the same effect on the literal 42 or on this sequence: both `42 @ 2` and `[True, False, True, False, True, False] @ 2` return `True`. Usuba does not allow bit-manipulation of words, which would not be efficient on most CPU architectures.

The Cryptol language is more expressive than Usuba, and includes features such as records, strings, user-defined types, predicates, modules, first-class type variables and lambda expressions. By focusing on symmetric cryptographic primitives in Usuba, we strove to keep the language as simple (yet expressive) as possible, and did not require those constructions.

Another aspect of Cryptol is that it allows programmers to write the specifications within their code, and assert their validity using SMT solvers (Z3 [124] by default). An example of such a property is found in Cryptol's reference AES implementation:

```
property AESCorrect msg key =
   aesDecrypt (aesEncrypt (msg, key), key) == msg
```

which states that decrypting a ciphertext encrypted with AES yields the original plaintext. When the SMT solver fails to prove properties in reasonable time, Cryptol falls back to random testing: it tests the property on a given number of random inputs and checks whether it holds. Usuba does not offer such features at this stage.

Overall, Cryptol is an expressive specification language for cryptography, but falls short on the performance aspect. With Usuba, we chose to focus only on symmetric primitives in order to provide performance on par with hand-tuned implementations. We thus adopted a bottom-up approach, offering only high-level constructions that we are able to compile to efficient code.

CAO [36, 38] is a domain-specific programming language (DSL) that focuses on cryptographic primitives. Like Usuba, CAO started from the observation that writing primitives in C either leads to poor performance because the C compiler is unable to optimize them well, or to unmaintainable code because optimizations are done by hand.

The level of abstraction provided by CAO is similar to Usuba: functions, `for` loops, standard C operators (+, −, ⋆, *etc.*), ranges to index multiple elements of a vector, concatenation of vectors. CAO also has a `map` operator, which specifies mapping from inputs to outputs of a permutation, in a style similar to Usuba's `perm` nodes.

However, whereas Usuba's main target is symmetric cryptography, CAO is more oriented toward asymmetric (public-key) cryptography, and thus offers many abstractions for finite field arithmetic. However, those constructions are also useful to specify some symmetric ciphers defined using mathematical operations (*e.g.*, AES). For instance, one can define a type `f` for AES's values as follows:

```
typedef f := gf[2 ** 8] over **8+ ** 4 + **3+ + 1
```

AES's Mixcolumn can then be written at a higher level in CAO than in Usuba, letting the programmer to directly appeal to operators in $GF(2^8)$, which Usuba does not provide.

To support public-key cryptography, CAO also provides conditionals in the language. However, to prevent timing attacks, variables can be annotated as being `secret`: the compiler will emit an error if a conditional depends on a `secret` value. Such a mechanism is not needed in Usuba, where conditionals on non-static values cannot be expressed at all.

Because of the exotic types introduced for dealing with public-key cryptography (*e.g.*, polynomials, very large integers), CAO applies several optimizations to its programs before generating C code. For instance, C compilers' strength reduction pass (replacing "strong" operations by "weaker" ones, such as replacing a multiplication by several additions) will not handle finite field arithmetic, but CAO's does.

CAO also offers a way for programs to be resilient against side-channel attacks, by providing an operator `?`, which introduces fresh randomness. However, introducing randomness throughout the computation is not sufficient to be secure [237]. Thus, CAO uses Hidden Markov Models to try and break the modified implementation. Usuba integrates recent progress in provable countermeasures against side-channel attacks [250], thus providing a stronger security (Chapter 6).

FaCT [100, 99] is a C-style DSL for cryptography, which generates provably constant-time LLVM Intermediate Representation (IR) code. FaCT allows developers to write cryptographic code without having to resort to programming "tricks" to make it constant-time, like masking conditionals, or using flags instead of early-return. Those tricks obfuscate the code, and an error can lead to devastating security issues [15].

Instead, FaCT allows developers to write C-style code, where secret values are annotated with the keyword `secret`. The compiler then takes care of transforming any non-constant-time idiom into constant-time ones. It thus automatically detects and secures unsafe early routine terminations, conditional branches, and memory accesses.

FaCT's transformations are proven to produce constant-time code. However, because LLVM could still introduce a vulnerability (for instance by optimizing branchless statements with conditional branches), they rely on dudect [259] to ensure that the final assembly is empirically constant-time. Moreover, FaCT has a notion of public safety to ensure the memory safety as well as the lack of buffer overflows/underflows and undefined behaviors. FaCT ensures the public safety of a program using the Z3 SMT solver [124].

FaCT and Usuba differ in their use-cases. FaCT targets protocols (*e.g.,* TLS) and asymmetric primitives (*e.g.,* RSA), whereas Usuba focuses on symmetric cryptography. Furthermore, Usuba is higher-level than FaCT: the latter can almost be straightforwardly compiled to C and requires developers to explicitly use SIMD instructions when they want them, while Usuba requires more normalization (especially when automatically bitslicing programs) and automatically generates vectorized code. Both languages, however, achieve similar performance as hand-tuned implementations.

dSCADL [254] is a **d**ata flow based **S**ymmetric **C**ryptographic **A**lgorithm **D**escription **L**anguage. The language is meant to allow developers to write symmetric primitives with a simple language, which should be intuitive to use, while still offering good performance.

dSCADL's variables are either scalars or cubes. Scalars are integers, either used in the control flow like loop indices, or to represent any mono-dimensional (potentially secret) data. Cubes are used for multidimensional data, for instance to represent AES's state as a matrix. To compare with Usuba's types, scalars resemble Usuba's $\mathtt{u}m$ atoms, while cubes are similar to Usuba's vectors. dSCADL provides operators to manipulate cubes, such as pointwise arithmetic and substitution, matrix multiplication, row and column concatenation. These cube operators allow dSCADL's AES implementation to be higher level that Usuba's, as `MixColumn` (the matrix multiplication) is done with a simple operator, expanded by the compiler to lower-level operations.

However, dSCADL compiles to OCaml code and then links with a runtime library, making it much slower than Usuba, which compiles directly to C with SIMD instructions.

Finally, dSCADL allows the use of secret values in conditionals, as well as lookup in tables at secret indices, making the produced code potentially vulnerable to timing attacks.

ADSLFC. Agosta and Pelosi [9] proposed a domain-specific language for cryptography. This DSL was not named, but we shall call it ADSLFC (**A** **D**omain **S**pecific **L**anguage **F**or **C**ryptography) in the following, for simplicity. ADSLFC is based on Python in the hope that developers will find it easy to use, and will easily assimilate the syntax (unlike Cryptol for instance, which they deem "much harder to understand for a non-specialized user [than C]" in [9], Section 4). Finally, ADSLFC is compiled to Python (but the authors mention as future work that they would like to compile to C as well), in order to allow for easy interoperability with C and C++.

The base type of ADSLFC is `int`, which represents a signed integer of unlimited precision. This type can then be refined by specifying its size (`int.32` for a 32-bit integer for instance), or made unsigned using the `u.` prefix. The TEA cipher for instance takes as input values of type `u.int.32`, similar to Usuba's `u32`. Vectors can be used to represent either arrays (possibly multidimensional) of integers or polynomials. To deal with finite field arithmetic, a `mod x` (where `x` is a polynomial) annotation can be added to a type, meaning that operations on this type are done modulo `x`.

Standard arithmetic operators are provided for integers (*e.g.,* addition, multiplication, exponentiation), as well as bitwise operators, and an operator to call a S-box (represented as a vector used as a lookup table). Additional operators are available to manipulate vectors: concatenation, indexing, replication, transposition, *etc.* The features of the language are thus similar to Usuba's, with added constructions to deal with finite field arithmetic and polynomials.

Finally, ADSLFC was designed to allow fast prototyping and development, with seemingly no regard for performance, unlike Usuba, for which speed is a crucial aspect. No performance numbers are provided in the original paper [9], but since ADSLFC compiles to Python, and no performance-related optimizations are mentioned, we can expect the generated code to be slower than Usuba's optimized C code.

BSC [248] is a direct ancestor of Usuba. The initial design of Usuba [212], which did not support *m*slicing, was largely inspired by BSC: lookup tables and permutations originated from BSC, and the types (Booleans and vectors of Booleans) were similar. Usuba's tuples are also inspired from BSC's vectors: in BSC, a vector of size n can be destructed into two vectors of size i and j (such that $i + j = n$) using the syntax `[a # b] = c` (where `a`, `b` and `c` are vectors of size i, j and n), which is equivalent to Usuba's `(a, b) = n`. Finally, Usuba borrows from BSC the algorithm to convert lookup tables into circuits.

> **!** The performance comparison between Usuba and BSC is available at: `https://github.com/DadaIsCrazy/usuba/tree/master/experimentations/usuba-vs-bsc`

Usuba improves upon BSC by expanding the range of optimizations beyond mere copy propagation. Furthermore, BSC did not offer loop constructs, and inlines every function, producing large amount of code. Benchmarking BSC against Usuba on DES (the only available example of BSC code) shows that Usuba is about 10% faster, mainly thanks to its scheduling algorithm.

Finally, *m*slicing was introduced almost a decade after BSC [189, 174], and most SIMD extensions postdate BSC: the first SSE instructions sets were introduced in 1999. Usuba shows that both *m*slicing and vectorization are nonetheless compatible with the programming model pioneered by BSC.

Low* [249] is a low-level language embedded in F* [286], which compiles to C and was used to implement the HACL* cryptographic library [310]. Being low-level, Low* offers high performance, almost on par with libsodium.

Low*, taking advantage of F* being a proof assistant as well as a programming language, can be used to prove the correctness of cryptographic implementations and enjoys a formally certified pipeline down to C (or assembly if CompCert is used to compile the produced C code). F*'s type-system also ensures the memory safety of Low* programs (*e.g.,* no out of bound accesses can be performed, uninitialized memory cannot be accessed). Furthermore, Low* ensures that branches and memory accesses do not depend on secret values, thus preventing any timing attack.

Low* is more versatile than Usuba, as the former can be used to write any cryptographic program (*e.g.,* protocols, public-key ciphers, block ciphers), whereas the latter is only designed for symmetric primitives. Usuba, however, provides higher-level abstractions: Low* offer similar abstractions as C and even requires programmers to explicitly manipulate the stack. Generating Low* code from Usuba would provide a high-level way to implement symmetric primitives, while still being able to benefit from the guarantees (*e.g.,* correctness, constant-time and memory safety for the whole protocol) provided by Low*.

Portable Assemblers

The cryptographic community tends to prefer assembly to C to write high-performance primitives, Daniel J. Bernstein even announcing "the death of optimizing compilers" [64]. Several custom assemblers have thus been developed (*e.g.,* `qhasm` [60] and `perlasm` [233]), aiming at writing high-performance code, while abstracting away some of the boilerplate and architecture-specific constraints.

The semantics of these assemblers remain largely undocumented, making for a poor foundation to carry verification work. Recent works have been trying to bridge that gap (*e.g.,* Jasmin [18] and Vale [87]) by introducing full-featured portable assembly languages, backed by a formal semantics and automated theorem proving facilities. The arguments put forward by these works resonate with our own: we have struggled to domesticate 3 C compilers (Clang, GCC, ICC), taming them into somewhat docile register allocators for SIMD intrinsics. While this effort automatically ripples through to all Usuba programs, it is still a wasteful process. Targeting one of these portable assemblers could allow us to perform more optimizations, and have more predictable performance, without needing to implement architecture-specific assembly backends.

Chapter 3

Usuba, Greek Edition

In this chapter, we give a formal treatment of Usuba, covering its type system and semantics. We take this opportunity to specify the invariants enforced by the compiler. Figure 3.1 illustrates the pipeline described in this chapter. Leaving aside type-inference and surface constructions (*e.g.*, vectors, loops, imperative assignments), we focus on Usuba$^{\text{core}}$, the core calculus of Usuba. After introducing the formal syntax of Usuba$^{\text{core}}$ and showing how it compares to Usuba (Section 3.1), we present its type system (Section 3.2). We then describe monomorphization of Usuba$^{\text{core}}$ programs into Usuba$^{\text{mono}}$ programs (Section 3.3): monomorphization is responsible for specializing a cipher to a given architecture and slicing form. The semantics of (well-typed) monomorphic programs is given in Section 3.4. We later introduce a normalization pass that lowers Usuba$^{\text{mono}}$ programs down to the Usuba0 language (Section 3.5). We can then translate Usuba0 programs into imperative programs in PseudoC, a C-like language (Section 3.6), focusing on the salient parts of the translation. Finally, we show that if our compiler is correct, compiling an Usuba$^{\text{core}}$ program pgm produces a PseudoC program $prog$, such that a single execution of $prog$ behaves as N sequential executions of pgm, or, put otherwise, $prog$ is a vectorized version of pgm.

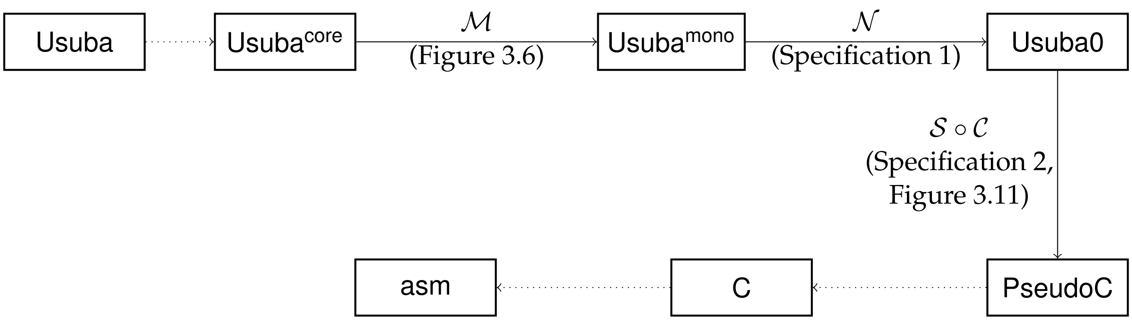


Figure 3.1: Usuba's formal pipeline

Notation: maps. We use the notation $\mu \mapsto \nu$ to denote the type of a map whose keys are of type μ and values are of type ν. Given a map $m : \mu \mapsto \nu$, and k and v two elements of type respectively μ and ν, the syntax $m(k) \leftarrow v$ denotes a map identical to m with an additional mapping from k to v, while the syntax $\{k \leftarrow v\}$ denotes a singleton from k to v. Conversely, the syntax $m(k)$ denotes the value associated with k in m. By construction, we never overwrite any key/value of a map: whenever we do $m(k) \leftarrow v$, there is never a previously existing binding for k in m.

Notation: collections. For conciseness, we use the symbol $\vec{t}$ to denote a tuple $(t_0, \dots, t_{n-1})$ of elements belonging to some syntactic category t. Given any tuple $\vec{t}$, we use the informal notation t_j to denote its j^{th} element.

3.1 Syntax

The syntax of Usuba^{core} is defined in Figure 3.2. A program *pgm* is a set of nodes *nd*. We impose that the node call graph is totally ordered. This forbids recursion (and mutual recursion) among nodes. Similarly, equations in a node must be totally ordered along variable definitions. This ensures that the equations can be sequentially scheduled and compiled to C. The node call graph must have a single root, which we conventionally call `main`. We assume that `main` is the last node of a program source.

Nodes (*nd*) are parameterized by polymorphic directions $\vec{\delta}$ and word sizes $\vec{\omega}$. An operator signature Θ provides the signature of the operators necessary for the node, thus restricting $\vec{\delta}$ and $\vec{\omega}$ to satisfy these signatures. Node calls explicitly pass the directions and word sizes of their argument to the callee.
Input type annotation $(i\ :_\flat\ \tau)$ are decorated so as to distinguish constants ($\flat = {}_\mathcal{K}$) from variables ($\flat = {}_i$). Similarly, output type annotations $(o\ :_\sharp\ \tau)$ are decorated so as to distinguish local variables ($\sharp = {}_\mathcal{L}$) from actual outputs ($\sharp = {}_o$)

Elaboration from Usuba to Usuba^{core} Notably, Usuba^{core} (Figure 3.2) does not provide vectors, loops and imperative assignments (:=). Their semantics is given by elaboration from Usuba to Usuba^{core} (Figure 3.3). We illustrate this elabortation on the following Usuba node:

```
node f(x:v1[4], y:v1) returns (z:v1[4])
vars t : v1
let
    t = 1;
    forall i in [0, 3] {
        z[i] = x[i] ^ y ^ t;
        t := ~t;
    }
tel

node g(a:u32x4, b:u32) returns (c:u32x4)
let
    c = f(a[3,1,0,2],b)
tel
```

This node is elaborated to:

$$
\begin{array}{l}
\textbf{node}\ f\ :\ \forall\delta\omega\ \{\texttt{\^{}}\ :\ \mathtt{u}_\delta\omega \rightarrow \mathtt{u}_\delta\omega \rightarrow \mathtt{u}_\delta\omega,\ \texttt{\~{}}\ :\ \mathtt{u}_\delta\omega \rightarrow \mathtt{u}_\delta\omega\}\ \Rightarrow \\
\quad (x_0\ :_i\ \mathtt{u}_\delta\omega,\ x_1\ :_i\ \mathtt{u}_\delta\omega,\ x_2\ :_i\ \mathtt{u}_\delta\omega,\ x_3\ :_i\ \mathtt{u}_\delta\omega,\ y\ :_i\ \mathtt{u}_\delta\omega) \\
\quad \rightarrow\ (t_0\ :_\mathcal{L}\ \mathtt{u}_\delta\omega,\ t_1\ :_\mathcal{L}\ \mathtt{u}_\delta\omega,\ t_2\ :_\mathcal{L}\ \mathtt{u}_\delta\omega,\ t_3\ :_\mathcal{L}\ \mathtt{u}_\delta\omega, \\
\qquad z_0\ :_o\ \mathtt{u}_\delta\omega,\ z_1\ :_o\ \mathtt{u}_\delta\omega,\ z_2\ :_o\ \mathtt{u}_\delta\omega,\ z_3\ :_o\ \mathtt{u}_\delta\omega)\ = \\
\quad t\ =\ 1; \\
\quad z_0\ =\ x_0\ \texttt{\^{}}\ y\ \texttt{\^{}}\ t; \\
\quad t_1\ =\ \texttt{\~{}}t; \\
\quad z_1\ =\ x_1\ \texttt{\^{}}\ y\ \texttt{\^{}}\ t_1; \\
\quad t_2\ =\ \texttt{\~{}}t_1; \\
\quad z_2\ =\ x_2\ \texttt{\^{}}\ y\ \texttt{\^{}}\ t_2; \\
\quad t_3\ =\ \texttt{\~{}}t_2; \\
\quad z_3\ =\ x_3\ \texttt{\^{}}\ y\ \texttt{\^{}}\ t_3; \\
\quad t_4\ =\ \texttt{\~{}}t_3;
\end{array}
$$

$$
\begin{array}{l}
\textbf{node}\ g\ :\ \forall\delta\ \{\}\ \Rightarrow \\
\quad (a_0\ :_i\ \mathtt{u}_\delta 32,\ a_1\ :_i\ \mathtt{u}_\delta 32,\ a_2\ :_i\ \mathtt{u}_\delta 32,\ a_3\ :_i\ \mathtt{u}_\delta 32,\ b\ :_i\ \mathtt{u}_\delta 32) \\
\quad \rightarrow\ (c_0\ :_o\ \mathtt{u}_\delta 32,\ c_1\ :_o\ \mathtt{u}_\delta 32,\ c_2\ :_o\ \mathtt{u}_\delta 32,\ c_3\ :_o\ \mathtt{u}_\delta 32)\ = \\
\quad (c_0,\ c_1,\ c_2,\ c_3)\ =\ f\ \delta\ 32\ (a_3,\ a_1,\ a_0,\ a_2,\ b)
\end{array}
$$

pgm	::=	$\vec{nd}$	program
nd	::=	$\textbf{node}\ f : \sigma = \vec{eqn}$	node
eqn	::=	$\vec{x} = e$	equation
e	::=		expressions
	$\mid$	n	(constant)
	$\mid$	x	(variable)
	$\mid$	$\vec{e}$	(tuple)
	$\mid$	$op\ e$	(overloaded call)
	$\mid$	$f\ \vec{D}\ \vec{w}\ e$	(node call)
σ	::=	$\forall \vec{\delta}\, \vec{\omega}.\Theta \Rightarrow (i \ \vec{:_\flat}\ \tau) \to (o \ \vec{:_\sharp}\ \tau)$	type scheme
Θ	::=		operator signature
	$\mid$	ϵ	(empty)
	$\mid$	$\Theta, op : \pi$	(declaration)
op	::=		overloaded operators
	$\mid$	$\texttt{\&} \mid \texttt{\|} \mid \texttt{\^{}} \mid \texttt{\~{}}$	(logical operators)
	$\mid$	$+ \mid \star \mid -$	(arithmetic operators)
	$\mid$	$\texttt{>>}_n \mid \texttt{<<}_n \mid \texttt{>>>}_n \mid \texttt{<<<}_n$	(shifts)
	$\mid$	$\texttt{shuffle}_{\vec{n}}$	(shuffles)
	$\mid$	$\texttt{bitmask}_n$	(bit masks)
	$\mid$	$\texttt{pack}$	(register packing)
$\flat$	::=		input annotation
	$\mid$	$\mathcal{K}$	(constant)
	$\mid$	i	(regular)
$\sharp$	::=		output annotation
	$\mid$	$\mathcal{L}$	(local)
	$\mid$	o	(regular)
π	::=	$\tau \to \tau$	function type
τ	::=		first-order types
	$\mid$	$\mathtt{u}_D w$	(atom)
	$\mid$	$\vec{\tau}$	(n-ary product)
D	::=		slicing direction
	$\mid$	dir	(static)
	$\mid$	δ	(slicing variable)
w	::=		word size
	$\mid$	m	(static)
	$\mid$	ω	(size variable)
m, n	::=	$0 \mid 1 \mid \ldots$	natural numbers
dir	::=		directions
	$\mid$	V	(vertical)
	$\mid$	H	(horizontal)

Figure 3.2: Syntax of Usuba$^{\text{core}}$

Loops:

$$(0) \quad \mathcal{T}(\texttt{forall}\ i\ \texttt{in}\ [e_i,\ e_f]\ \vec{eqs}) = \vec{eqs}[^i/_{e_i}]; \vec{eqs}[^i/_{e_i}+1] \dots \vec{eqs}[^i/_{e_f}-1]; \vec{eqs}[^i/_{e_f}]$$

Imperative assignment:

$$(1) \quad \mathcal{T}(x := e; \vec{eqs}) = x' = e; \vec{eqs}[^x/_{x'}]$$

Vectors:

$$\begin{array}{lrcl}
(2) & \mathcal{T}(x : \tau[n]) & = & (x_0 : \tau, x_1 : \tau \dots x_{n-1} : \tau) \\
(3) & \mathcal{T}(x^{\tau[n]}) & = & (x_0^\tau \dots x_{n-1}^\tau) \\
(4) & \mathcal{T}(x[i]) & = & x_i \\
(5) & \mathcal{T}(x[a..b]) & = & (x_a, x_{a+1} \dots x_{b-1}, x_b) \\
(6) & \mathcal{T}(x[l]) & = & (x_{l_0}, x_{l_1} \dots x_{l_{n-1}}, x_{l_n})
\end{array}$$

Figure 3.3: Elaboration of Usuba's syntactic sugar to Usuba$^{\text{core}}$

The `forall` and `:=` of f have been expanded (rules (0) and (1) of Figure 3.3). `f`'s vector parameters `x` and `z` declarations have been replaced by tuples (rule (2)). Accesses to elements of `x` and `z` have been replaced by scalars (rule (4)). In `g`, the two vectors `a` and `b` (recall from Section 2.3 that $um \times n$ is equivalent to $um[n]$) have also been expanded in the parameters (rule (2)), and have been replaced by tuples in the equations, including when used with a slice (rules (3) and (6)).
Additionally, polymorphic parameters δ and ω, which were implicit in the type `v1` of `f`, are now explicit. Similarly, the operator signature of the `f`, implied in the Usuba code by the use of `^` and `~` on values of type `v1`, is now explicitly lambda-lifted in a signature. `g`, on the other hand, is only direction-polymorphic, and does not use any operators. As a result, its operator signature is empty. The call to f in g now passes δ as slicing direction, and 32 as word size.

3.2 Type System

Our type system deliminates a first-order language featuring ad-hoc polymorphism [169, 296] over a fixed set of signatures.

Typing Validity Figure 3.4 defines the validity of the various contexts constructed during typing. Within a node, Γ is an environment containing polymorphic directions and word sizes as well as typed variables. Similarly, Δ contains function type-schemes.

Type system. Figure 3.5 gives the typing rules for Usuba$^{\text{core}}$. A program $\vec{nd}$ is well-typed with respect to an ambient operator signature $\Theta_{\mathcal{A}}$ if all of its nodes are well-typed in the program context $\Delta = \{f : \sigma \mid f : \sigma = \vec{eqn} \in \vec{nd}\}$ and the operator signature $\Theta_{\mathcal{A}}$ (rule PROG).

A node is well-typed with respect to an ambient operator signature $\Theta_{\mathcal{A}}$ and a program environment Δ if each of its equation is well-typed in the typing context Γ built from its type-scheme σ, and the operator signature formed by the union of $\Theta_{\mathcal{A}}$ and the node-local operator signature Θ (rule NODE).

The type of the application of an overloaded operator *op* to an expression e of type τ is τ' if the operator signature Θ contains the operator *op* with type $\tau \to \tau'$ (rule OP APP).

$$\begin{array}{rcl} \Gamma & ::= & \mathcal{E} \\ & | & \Gamma, \delta \\ & | & \Gamma, \omega \\ & | & \Gamma, x : \tau \end{array} \qquad \begin{array}{rcl} \Delta & ::= & \mathcal{E} \\ & | & \Delta, f : \sigma \end{array}$$

$$\boxed{\vdash \Gamma \text{ VALID}}$$

$$\frac{}{\vdash \mathcal{E} \text{ VALID}} \qquad \frac{\vdash \Gamma \text{ VALID}}{\vdash \Gamma, \delta \text{ VALID}} \qquad \frac{\vdash \Gamma \text{ VALID}}{\vdash \Gamma, \omega \text{ VALID}} \qquad \frac{\vdash \Gamma \text{ VALID} \qquad \Gamma \vdash \tau \text{ VALID}}{\vdash \Gamma, x : \tau \text{ VALID}}$$

$$\boxed{\Gamma \vdash \tau \text{ VALID}}$$

$$\frac{\Gamma \vdash \tau_j \text{ VALID}}{\Gamma \vdash \vec{\tau} \text{ VALID}} \qquad \frac{\delta \in \Gamma}{\Gamma \vdash \mathrm{u}_\delta m \text{ VALID}} \qquad \frac{\omega \in \Gamma}{\Gamma \vdash \mathrm{u}_{\mathrm{dir}} \omega \text{ VALID}}$$

$$\frac{}{\Gamma \vdash \mathrm{u}_{\mathrm{dir}} m \text{ VALID}} \qquad \frac{\delta \in \Gamma \qquad \omega \in \Gamma}{\Gamma \vdash \mathrm{u}_\delta \omega \text{ VALID}}$$

$$\boxed{\vdash \sigma \text{ VALID}}$$

$$\frac{\vec{\delta}, \vec{\omega} \vdash \Theta \text{ VALID} \qquad \vec{\delta}, \vec{\omega} \vdash \tau_j \text{ VALID} \qquad \vec{\delta}, \vec{\omega} \vdash \tau'_j \text{ VALID}}{\vdash \forall \vec{\delta}\, \vec{\omega}\, .\Theta \;\Rightarrow\; (\overrightarrow{i \;:_\sharp\; \tau}) \to (\overrightarrow{o \;:_\flat\; \tau'}) \text{ VALID}}$$

$$\boxed{\Gamma \vdash \Theta \text{ VALID}}$$

$$\frac{}{\Gamma \vdash \mathcal{E} \text{ VALID}} \qquad \frac{\Gamma \vdash \Theta \text{ VALID} \qquad \Gamma \vdash \pi \text{ VALID}}{\Gamma \vdash \Theta, op : \pi}$$

$$\boxed{\Gamma \vdash \pi \text{ VALID}}$$

$$\frac{\Gamma \vdash \tau \text{ VALID} \qquad \Gamma \vdash \tau' \text{ VALID}}{\Gamma \vdash \tau \to \tau' \text{ VALID}}$$

$$\boxed{\vdash \Delta \text{ VALID}}$$

$$\frac{}{\vdash \mathcal{E} \text{ VALID}} \qquad \frac{\vdash \Delta \text{ VALID} \qquad \vdash \sigma \text{ VALID}}{\vdash \Delta, f : \sigma \text{ VALID}}$$

$$\boxed{\vdash \Delta; \Gamma; \Theta \text{ VALID}}$$

$$\frac{\vdash \Delta \text{ VALID} \qquad \vdash \Gamma \text{ VALID} \qquad \Gamma \vdash \Theta \text{ VALID}}{\vdash \Delta; \Gamma; \Theta \text{ VALID}}$$

Figure 3.4: Validity of a typing context

$$\boxed{\Theta_{\mathcal{A}} \vdash pgm}$$

$$\frac{\overrightarrow{nd}; \Theta_{\mathcal{A}} \vdash nd_j}{\Theta_{\mathcal{A}} \vdash \overrightarrow{nd}} \text{ PROG}$$

$$\boxed{\Delta; \Theta_{\mathcal{A}} \vdash nd}$$

$$\frac{\Delta; \vec{\delta}, \vec{\sigma}, \overrightarrow{i : \tau}, \overrightarrow{o : \tau}; \Theta_{\mathcal{A}}, \Theta \vdash eqn_j}{\Delta; \Theta_{\mathcal{A}} \vdash \textbf{node}\ f : \forall\, \vec{\delta}\, \vec{\omega}.\Theta \Rightarrow (\overrightarrow{i :_\flat \tau}) \rightarrow (\overrightarrow{o :_\sharp \tau}) = \overrightarrow{eqn}} \text{ NODE}$$

$$\boxed{\Delta; \Gamma; \Theta \vdash \vec{x} = e}$$

$$\frac{\Delta; \Gamma; \Theta \vdash \vec{x} : \tau \qquad \Delta; \Gamma; \Theta \vdash e : \tau}{\Delta; \Gamma; \Theta \vdash \vec{x} = e} \text{ EQN}$$

$$\boxed{\Delta; \Gamma; \Theta \vdash e : \tau}$$

$$\text{CONST } \frac{\vdash \Delta; \Gamma; \Theta \text{ VALID}}{\Delta; \Gamma; \Theta \vdash n : \mathtt{u}_D w} \qquad \text{VAR } \frac{\vdash \Delta; \Gamma; \Theta \text{ VALID} \qquad x : \tau \in \Gamma}{\Delta; \Gamma; \Theta \vdash x : \tau}$$

$$\text{TUPLE } \frac{\Delta; \Gamma; \Theta \vdash e_j : \tau_j}{\Delta; \Gamma; \Theta \vdash \vec{e} : \vec{\tau}} \ (|\vec{e}| = |\vec{\tau}|) \qquad \text{OP APP } \frac{\Delta; \Gamma; \Theta \vdash e : \tau \qquad op : \tau \rightarrow \tau' \in \Theta}{\Delta; \Gamma; \Theta \vdash op\ e : \tau'}$$

$$\text{NODE APP } \frac{\begin{array}{c}\Delta; \Gamma; \Theta \vdash e : \tau \qquad f : \forall\, \vec{\delta}\, \vec{\omega}.\Theta' \Rightarrow (\overrightarrow{i :_\flat \tau'}) \rightarrow (\overrightarrow{o :_\sharp \tau''}) \in \Delta \\ \vec{\tau'}[\vec{\delta}/\vec{D}][\vec{\omega}/\vec{w}] = \tau \qquad \Theta'[\vec{\delta}/\vec{D}][\vec{\omega}/\vec{w}] \subseteq \Theta\end{array}}{\Delta; \Gamma; \Theta \vdash f\ \vec{D}\ \vec{w}\ e : \vec{\tau''}[\vec{\delta}/\vec{D}][\vec{\omega}/\vec{w}]} \ (|\vec{\omega}| = |\vec{w}|), (|\vec{\delta}| = |\vec{D}|)$$

Figure 3.5: Type system

Finally, node application (rule NODE APP) combines an explicit type application—for slicing and size parameters—followed by a full function application to the arguments (Usuba does not support partial application). We check that the local operator signature Θ' is compatible with the current Θ. That is, substituting f's slicing direction and word size variables for the call's own slicing direction and word size variables must produce an operator signature whose operators are provided by Θ, which ensures that all operations within the node are available on the architecture modelled by Θ.

3.3 Monomorphization

We do not implement ad-hoc polymorphism by dictionary passing [296], which would imply a run-time overhead, but resort to static monomorphization [169, 180]. Monomorphization (Figure 3.6) is a source-to-source transformation that converts a polymorphic Usuba$^{\text{core}}$ program into a monomorphic Usuba$^{\text{mono}}$ program. Directions and word sizes in an Usuba$^{\text{mono}}$ program are static, which means that all the type-schemes in the program are of the form $(\overrightarrow{i\ :_\flat\ \tau}) \rightarrow (\overrightarrow{o\ :_\sharp\ \tau})$ rather than $\forall \vec{\delta}\vec{\omega}.\Theta \Rightarrow (\overrightarrow{i\ :_\flat\ \tau}) \rightarrow (\overrightarrow{o\ :_\sharp\ \tau})$.

$$\boxed{\mathcal{M}^{\Theta_{\texttt{arch}}}(pgm) \overset{\vec{dir},\vec{m}}{\rightsquigarrow} pgm}$$

$$\frac{\mathcal{M}^{\Theta_{\texttt{arch}}}(\texttt{main})_{pgm} \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \vec{nd}; main'}{\mathcal{M}^{\Theta_{\texttt{arch}}}(pgm, \texttt{main}) \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \vec{nd}; main'} \text{ PROG}$$

$$\boxed{\mathcal{M}^{\Theta_{\texttt{arch}}}(nd)_{pgm} \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \vec{nd}; nd}$$

$$\text{NODE } \frac{\begin{array}{c} |\vec{\omega}| = |\vec{m}| \qquad |\vec{\delta}| = |\vec{dir}| \\ \Theta[\vec{\delta}/\vec{dir}][\vec{\omega}/\vec{m}] \subseteq \Theta_{\texttt{arch}} \\ \mathcal{M}^{\Theta_{\texttt{arch}}}(eqn_j)_{pgm} \rightsquigarrow \vec{nd_j}; eqn'_{\vec{j}} \\ nd = \textbf{node}\ f : \forall\, \vec{\delta}\, \vec{\omega}.\Theta' \Rightarrow (i \vec{:_\flat}\, \tau) \rightarrow (o :_\sharp \tau') = \vec{eqn} \\ nd' = \textbf{node}\ f^{\vec{dir}}_{\vec{m}} : (i \vec{:_\flat}\, \tau[\vec{\delta}/\vec{dir}][\vec{\omega}/\vec{m}]) \rightarrow (o \vec{:_\sharp}\, \tau'[\vec{\delta}/\vec{dir}][\vec{\omega}/\vec{m}]) = \vec{eqn'} \end{array}}{\mathcal{M}^{\Theta_{\texttt{arch}}}(nd)_{pgm} \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \bigcup \vec{nd_j}; nd'}$$

$$\boxed{\mathcal{M}^{\Theta_{\texttt{arch}}}(eqn)_{pgm} \rightsquigarrow \vec{nd}; eqn}$$

$$\frac{\mathcal{M}^{\Theta_{\texttt{arch}}}(e)_{pgm} \rightsquigarrow \vec{nd}; e'}{\mathcal{M}^{\Theta_{\texttt{arch}}}(\vec{x} = e)_{pgm} \rightsquigarrow \vec{nd}; \vec{x} = e'} \text{ EQN}$$

$$\boxed{\mathcal{M}^{\Theta_{\texttt{arch}}}(e)_{pgm} \rightsquigarrow \vec{nd}; e}$$

$$\frac{}{\mathcal{M}^{\Theta_{\texttt{arch}}}(n)_{pgm} \rightsquigarrow \varnothing; n} \text{ CONST} \qquad \frac{}{\mathcal{M}^{\Theta_{\texttt{arch}}}(x)_{pgm} \rightsquigarrow \varnothing; x} \text{ VAR}$$

$$\frac{\mathcal{M}^{\Theta_{\texttt{arch}}}(e_j)_{pgm} \rightsquigarrow \vec{nd_j}; e'_j}{\mathcal{M}^{\Theta_{\texttt{arch}}}(\vec{e})_{pgm} \rightsquigarrow \bigcup \vec{nd_j}; \vec{e'}} \text{ TUPLE} \qquad \frac{\mathcal{M}^{\Theta_{\texttt{arch}}}(e)_{pgm} \rightsquigarrow \vec{nd}; e'}{\mathcal{M}^{\Theta_{\texttt{arch}}}(op\ e)_{pgm} \rightsquigarrow \vec{nd}; op\ e'} \text{ OP APP}$$

$$\frac{\begin{array}{c} \mathcal{M}^{\Theta_{\texttt{arch}}}(pgm(f))_{pgm} \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \vec{nd'}; nd \\ \mathcal{M}^{\Theta_{\texttt{arch}}}(e)_{pgm} \rightsquigarrow \vec{nd''}; e' \end{array}}{\mathcal{M}^{\Theta_{\texttt{arch}}}(f\ \vec{dir}\ \vec{m}\ e)_{pgm} \rightsquigarrow nd, \bigcup \vec{nd'}, \vec{nd''}; f^{\vec{dir}}_{\vec{m}}\ e'} \text{ NODE APP}$$

Figure 3.6: Monomorphization of Usuba[core] programs

Monomorphization is defined with respect to a specific architecture that implements an operator signature $\Theta_{\texttt{arch}}$. The monomorphization of an expression e produces an expression *and* a monomorphic set of nodes corresponding to the monomorphization of the nodes potentially called by e. This means that in turn, the monomorphization of an equation (resp., node) produces an equation (resp., node) and a monomorphic set of nodes.

In order to monomorphize a program, we only need to monomorphize its `main` node, providing concrete directions $\vec{dir}$ and word sizes $\vec{m}$ as inputs to the transformation. The monomorphization of `main` then triggers the monomorphization of the nodes it calls, and so on. Any node that is not in the call graph of `main` will not be monomorphized and is discarded.

To monomorphize a node, its direction and word size parameters are instantiated with the concrete directions $\vec{dir}$ and word sizes $\vec{m}$ provided. We also check that the monomorphic signature $\Theta_{\texttt{arch}}$ is sufficient to implement the operator signature Θ specialized to $\vec{dir}$ and $\vec{m}$. This ensures that the operators used in the node exist on the architecture targeted by the compilation. We note $f_{\vec{m}}^{\vec{dir}}$ the *name* of the specialized node.

Monomorphization may only fail at the top level, when specializing `main`: we ask that directions $\vec{dir}$ and word sizes $\vec{m}$ are fully applied and that $\Theta_{\texttt{arch}}$ provides all operators of the signature of `main` specialized to $\vec{dir}$ and $\vec{m}$.

Each of the n equations of a node is also monomorphized, which produces n sets of nodes $\vec{nd}_1$, $\vec{nd}_2$ ... $\vec{nd}_n$. We absorb potential duplicates by taking their union. Note that two monomorphic nodes with the same name $f_{\vec{m}}^{\vec{dir}}$ have necessarily been monomorphized with the same concrete directions and word sizes and thus have the same semantics.

Monomorphization proceeds structurally over expressions (rules CONST, VAR, TUPLE, OP APP, NODE APP).

Monomorphizing an operator application $op\ e$ (rule OP APP) only requires monomorphizing its argument e. Typing ensures that op was in the node's operator signature Θ, while monomorphization of a node (rule NODE) ensures that Θ is compatible with the architectures operator signature $\Theta_{\texttt{arch}}$; we are guaranteed that $op : \pi \in \Theta_{\texttt{arch}}$.

Finally, monomorphizing a node application (rule NODE APP) $f\ \vec{dir}\ \vec{m}\ e$ triggers the monomorphization of f with the concrete direction $\vec{dir}$ and word size parameters $\vec{m}$, which produces a specialized version $f_{\vec{m}}^{\vec{dir}}$ of f.

Example 3.3.1. Consider the following program:

$$\begin{array}{l}
\textbf{node}\ f\ :\ \forall\,\delta\,\omega\ \{\hat{}\ : \mathrm{u}_\delta\omega \to \mathrm{u}_\delta\omega\}\ \Rightarrow\ (x :_i \mathrm{u}_\delta\omega,\ y :_i \mathrm{u}_\delta\omega) \to (z :_o \mathrm{u}_\delta\omega)\ =\\
\quad z = x\ \hat{}\ y\\
\\
\textbf{node}\ \texttt{main} : \{\} \Rightarrow (a :_i \mathrm{u}_V 32,\ b :_i \mathrm{u}_V 32,\ c :_i \mathrm{u}_H 16,\ d :_i \mathrm{u}_H 16,\ e :_i \mathrm{u}_V 32,\ f :_i \mathrm{u}_V 32)\\
\qquad\qquad\qquad\quad \to (u :_o \mathrm{u}_V 32,\ v :_o \mathrm{u}_H 16,\ w :_o \mathrm{u}_V 32)\ =\\
\quad u = f\ V\ 32\ (a, b);\\
\quad v = f\ H\ 16\ (c, d);\\
\quad w = f\ V\ 32\ (e, f)
\end{array}$$

Monomorphizing `main` (with respect to an ambient signature $\Theta_{\texttt{arch}} = \{\hat{}\ : \mathrm{u}_V 32 \to \mathrm{u}_V 32 \to \mathrm{u}_V 32, \hat{}\ : \mathrm{u}_H 16 \to \mathrm{u}_H 16 \to \mathrm{u}_H 16\}$ providing an instance of `xor` for $\mathrm{u}_V 32$ and another one for $\mathrm{u}_H 16$) produces the following program:

$$
\begin{array}{l}
\textbf{node } f^{V}_{32} \ : (x :_i \mathtt{u}_V 32,\ y :_i \mathtt{u}_V 32) \ \rightarrow\ (z :_o \mathtt{u}_V 32)\ = \\
\quad z = x \ \hat{}\ y
\end{array}
$$

$$
\begin{array}{l}
\textbf{node } f^{H}_{16} \ : (x :_i \mathtt{u}_H 16,\ y :_i \mathtt{u}_H 16) \rightarrow (z :_o \mathtt{u}_H 16)\ = \\
\quad z = x \ \hat{}\ y
\end{array}
$$

$$
\begin{array}{l}
\textbf{node } \mathtt{main} : (a :_i \mathtt{u}_V 32,\ b :_i \mathtt{u}_V 32,\ c :_i \mathtt{u}_H 16,\ d :_i \mathtt{u}_H 16,\ e :_i \mathtt{u}_V 32,\ f :_i \mathtt{u}_V 32) \\
\qquad\qquad\qquad \rightarrow (u :_o \mathtt{u}_V 32,\ v :_o \mathtt{u}_H 16, w :_o \mathtt{u}_V 32) = \\
\quad u = f^{V}_{32}\ (a, b); \\
\quad v = f^{H}_{16}\ (c, d); \\
\quad w = f^{V}_{32}\ (e, f)
\end{array}
$$

We conjecture that monomorphization is sound. That is, if a program type-checks, and the monomorphization of this program succeeds, then the resulting program type-checks as well:

Conjecture 1 (Soundness of monomorphization)**.** *Let* pgm *be a well-typed program in the empty context (*$\vdash$ pgm*). Let* $\Theta_{\mathtt{arch}}$ *be a valid operator signature provided by a given architecture. Let* $\vec{dir}$ *and* $\vec{m}$ *be concrete directions and word sizes. Let* pgm′ *be a monomorphic program such that* $\mathcal{M}^{\Theta_{\mathtt{arch}}}(\text{pgm}) \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \text{pgm}'$*. Then, we have* $\Theta_{\mathtt{arch}} \vdash \text{pgm}'$*.*

3.4 Semantics

Figure 3.7a gives an interpretation of types as set-theoretic objects. The semantics of a function type is a (mathematical) function mapping elements from its domain to its codomain. The semantics of a tuple is a Cartesian product of values. The semantics of an atom $\mathtt{u}_D m$ is a value that fits in m bits. Finally, the semantics of an operator signature is a map from operators to functions.

The semantics of programs (Figure 3.7b) is parameterized by ρ, a map from operators to mathematical functions, which assign their meaning to the operators:

$$\rho ::= ((op, \pi) \mapsto (\vec{c} \rightarrow c))*$$

An expression denotes a tuple of integers, provided an operator environment ρ, an equation environment $\vec{eqn}$ (that corresponds to the equations of the enclosing node) and the overall program pgm. The semantics of a variable x in the equation environment $\vec{eqn}$ is the value associated to x in the mapping created by $[\![\vec{eqn}]\!]$. This definition is well-founded since equations are totally ordered.

We assume that operators are typed, even though, for simplicity, we ignored this aspect so far. In practice, we can get the type of any expressions (and operators) using the rules introduced in Figure 3.5. The semantics of an operator application is defined through ρ, which provides an implementation of the operator at a given type on a given architecture. Note that the semantics is guided by types (as well as syntax): ρ may contain several implementations of a given operators at different types (*e.g.,* 16-bit and 32-bit addition, or *v*sliced and *h*sliced 16-bit rotation).

The semantics of an equation is then defined as a mapping from variables to values. Straightforwardly, the semantics of a set of equations $\vec{eqn}$ is defined as the union of the semantics of its equations eqn_j. The semantics of a node f is a function from a tuple of inputs to a tuple of outputs. To relate the formal inputs $\vec{i}$ to the actual inputs $\vec{n}$, we extend the equational system with $\vec{i} = \vec{n}$ and compute the resulting denotation. The outputs of the node are obtained by filtering out inputs and local variables. Finally, the semantics of a program pgm corresponds to the semantics of its $\mathtt{main}$ node.

$$
\begin{array}{lcl}
[\![\tau \to \tau']\!] & = & \{f \mid \forall v \in [\![\tau]\!], f(v) \in [\![\tau']\!]\} \\
[\![\vec{\tau}]\!] & = & \{(v_0, ..., v_N) \mid v_j \in [\![\tau_i]\!]\} \\
[\![\mathtt{u}_D m]\!] & = & \{n \mid 0 \le n < 2^m\} \\
[\![\Theta]\!] & = & \{(op^\pi \mapsto [\![\Theta(op^\pi)]\!]) \mid op^\pi \in \Theta\}
\end{array}
$$

(a) Semantics of types

$$\boxed{[\![e]\!] : env \to \vec{eqn} \to pgm \to eqn \to \vec{n}}$$

$$
\begin{array}{rl}
{}^{\rho}[\![n]\!]^{\vec{eqn}}_{pgm} & = n \\
{}^{\rho}[\![x]\!]^{\vec{eqn}}_{pgm} & = {}^{\rho}[\![\vec{eqn}]\!]^{\vec{eqn}}_{pgm}\Big|_{x} \\
{}^{\rho}[\![\vec{e}]\!]^{\vec{eqn}}_{pgm} & = \overrightarrow{{}^{\rho}[\![e]\!]^{\vec{eqn}}_{pgm}} \\
{}^{\rho}[\![op^\pi\ e]\!]^{\vec{eqn}}_{pgm} & = \rho(op^\pi)\ {}^{\rho}[\![e]\!]^{\vec{eqn}}_{pgm} \\
{}^{\rho}[\![f\ e]\!]^{\vec{eqn}}_{pgm} & = {}^{\rho}[\![pgm(f)]\!]_{pgm}\ {}^{\rho}[\![e]\!]^{\vec{eqn}}_{pgm}
\end{array}
$$

$$\boxed{[\![eqn]\!] : env \to pgm \to \vec{eqn} \to eqn \to \overrightarrow{var \to n}}$$

$$
{}^{\rho}[\![\vec{x} = e]\!]^{\vec{eqn}}_{pgm} = \textbf{let } \vec{e'} = {}^{\rho}[\![e]\!]^{\vec{eqn}}_{pgm} \textbf{ in } \{x_j \leftarrow e'_j \mid j \in [0, |\vec{x}|]\}
$$

$$\boxed{[\![\vec{eqn}]\!] : env \to pgm \to \vec{eqn} \to \overrightarrow{var \to n}}$$

$$
{}^{\rho}[\![\vec{eqn}]\!]_{pgm} = \bigcup {}^{\rho}[\![eqn_j]\!]^{\vec{eqn}}_{pgm}
$$

$$\boxed{[\![nd]\!] : env \to pgm \to nd \to \vec{n} \to \vec{n}}$$

$$
{}^{\rho}[\![\textbf{node } f : (\overrightarrow{i\ :_\flat\ \tau}) \to (\overrightarrow{o\ :_\sharp\ \tau'}) = \vec{eqn}]\!]_{pgm}\ \vec{n} = \textbf{let } \vec{eqn}' = (\vec{i} = \vec{n}, \vec{eqn}) \textbf{ in } {}^{\rho}[\![\vec{eqn}']\!]_{pgm}\Big|_{o_\sharp = o}
$$

$$\boxed{[\![pgm]\!] : env \to \vec{n} \to \vec{n}}$$

$$
{}^{\rho}[\![pgm, \mathtt{main}]\!]\ \vec{n} = {}^{\rho}[\![\mathtt{main}]\!]_{pgm} \vec{n}
$$

(b) Semantics of programs and expressions

ρ is a valid implementation of an operator signature $\Theta_{\texttt{arch}}$ if and only if ρ provides all the operators specified by $\Theta_{\texttt{arch}}$:

$$\rho \models \Theta \iff \forall op \in \Theta_{\texttt{arch}}, \rho(op) \in [\![\Theta_{\texttt{arch}}(op)]\!]$$

A program has a semantics with respect to an operator signature $\Theta_{\texttt{arch}}$ when, for all valid implementations ρ of $\Theta_{\texttt{arch}}$, the main node has a semantics:

$$\Theta_{\texttt{arch}} \models pgm, \texttt{main} \iff \forall \rho \models \Theta_{\texttt{arch}},\ {}^{\rho}[\![\texttt{main}]\!]_{pgm} \in [\![\texttt{typeof}(\texttt{main})]\!]$$

where $\texttt{typeof}(\textbf{node}\ f : \sigma = \vec{eqn}) = \sigma$.

Our type system is adequate with respect to this (monomorphic) semantics if it satisfies the following property:

Conjecture 2 (Fundamental theorem of monomorphic programs)**.** *Let* pgm *be a monomorphic program and* $\Theta_{\texttt{arch}}$ *a valid operator signature provided by a given architecture. If* pgm *type-checks in* $\Theta_{\texttt{arch}}$*, then* pgm *has a semantics with respect to* $\Theta_{\texttt{arch}}$*, i.e.,*

$$\Theta_{\texttt{arch}} \vdash \text{pgm} \implies \Theta_{\texttt{arch}} \models \text{pgm}$$

If both Conjecture 1 and Conjecture 2 are true, then a program that monomorphizes admits a semantics:

Corollary 1. *Let* pgm *be a program that type-checks in the empty context. Let* $\Theta_{\texttt{arch}}$ *be a valid operator signature provided by a given architecture. Let* $\vec{dir}$ *and* $\vec{m}$ *be concrete directions and word sizes. Let* pgm′ *be a monomorphic program such that* $\mathcal{M}^{\Theta_{\texttt{arch}}}(\text{pgm}) \overset{\vec{dir},\vec{m}}{\rightsquigarrow} \text{pgm}'$. *We have* $\Theta_{\texttt{arch}} \models \text{pgm}'$ *that defines the semantics of* pgm *through monomorphization.*

3.5 Usuba0

Usuba0 is a strict subset of Usuba$^{\text{mono}}$, closer to C, and thus easier to translate to C. Compared to Usuba$^{\text{mono}}$, expressions in Usuba0 are not nested (put otherwise, the definition of expressions is not recursive), and Usuba0 prevents the use of tuples in expressions as well as on the left-hand side of equations (except in the case of node calls):

$$\begin{array}{lcll}
eqn^0 & ::= & & \text{equation} \\
 & | & x = e^0 & \text{(assignment)} \\
 & | & x = op\ \vec{e^0} & \text{(operator call)} \\
 & | & \vec{x} = f\ \vec{e^0} & \text{(node call)} \\
e^0 & ::= & & \text{expressions} \\
 & | & n & \text{(constant)} \\
 & | & x & \text{(variable)}
\end{array}$$

Usuba$^{\text{mono}}$ is converted into Usuba0 by a semantics-preserving transformation called normalization (Specification 1). Normalization is applied to a monomorphic Usuba$^{\text{mono}}$ program before translating it to PseudoC. We describe its implementation in Section 4.1.

Specification 1 (Normalization)**.** Let *pgm* be a monomorphic Usuba$^{\text{mono}}$ program that type-checks with respect to an architecture operator signature $\Theta_{\texttt{arch}}$. Let $\rho \models \Theta_{\texttt{arch}}$ be an operator environment that implements $\Theta_{\texttt{arch}}$. Normalization, written $\mathcal{N}$, is a transformation that converts any monomorphic Usuba$^{\text{mono}}$ program *pgm* into an equivalent Usuba0 program, *i.e.*, ${}^{\rho}[\![pgm]\!] = {}^{\rho}[\![\mathcal{N}(pgm)]\!]$.

We also specify equation scheduling (Specification 2) as the process of sorting the equations of all nodes according to their dependencies, in anticipation of their translation to imperative assignments.

Specification 2 (Scheduling)**.** Let *pgm* be an Usuba0 program that that has a semantics with respect to an operator environment ρ. Scheduling, written $\mathcal{S}$, is a transformation that converts any Usuba0 program *pgm* into an equivalent Usuba0 program, such that ${}^{\rho}[\![pgm]\!]_{pgm} = {}^{\rho}[\![\mathcal{S}(pgm)]\!]_{pgm}$, and, that all equations of all nodes of $\mathcal{S}(pgm)$ are sorted according to their dependencies.

3.6 Translation to PseudoC

We formalize an imperative language close to C as our target, which we shall call PseudoC. PseudoC is inspired by the Clight [82, 81] language, a subset of C, used as source language for the CompCert compiler. However, PseudoC is a stripped down variant of Clight: we do not include `for` loops, pointer arithmetic, structures or arrays. The semantics of operators in PseudoC also differs from Clight since we do not fix the underlying architecture.

3.6.1 Syntax

Figure 3.8 describes the syntax of PseudoC. A PseudoC program is a list of function declarations. Functions are of type **void** (returned values are always passed back through pointers). Functions contain two kinds of typed variable declarations: parameters and local variables. Similarly to Clight, we prevent function calls from appearing within expressions. Expressions are thus free of side-effects (since no primitive performs any side-effect). Primitives are annotated with an integer n representing the size of their argument (*e.g.*, 128 for a 128-bit SSE vector, 256 for a 256-bit AVX vector), as well as an integer m containing the size of the atom they manipulate. For instance, the primitive $+_8^{128}$ represents a vector addition performing 16 8-bit additions.

3.6.2 Semantics

We model the memory using a collection of blocks. A memory M is a collection of blocks, each containing a single value (of varying size), and indexed using integers. This model is inspired by the one proposed by Leroy and Blazy [200] and used for the semantics of Clight [81]. However, Leroy and Blazy [200] allow for addresses to point within a block (an address is a pair *(address,offset)*), whereas thanks to the lack of arrays and structures in our language, blocks are always addressed with offset 0.

Statements are evaluated within a global environment Ξ containing all functions of the program, and an environment γ that assigns their meaning to primitives. Within a function, a local environment E maps local variables to references of memory blocks containing their values. Figure 3.9 defines the structure of the environments and memories.

The value of a local variable id can thus be retrieved by doing $M(E(id))$ (rule VAR in Figure 3.10). Similarly, talking the address of a variable id using the `&` operator is done with a simple $E(id)$ (rule REF). Dereferencing a variable, on the other hand, requires to get the value of the variable (which represents an address), and get the value stored at this address, or put otherwise, doing $M(M(E(i)))$ (rule DEREF).

Notably, thanks to the lack of side-effect in expressions, the memory M used to evaluate an expression e is not changed by the evaluation of e (rules CONST, DEREF, REF, VAR, PRIM). Conversely, statements update the memory.

$prog$	::=	$\vec{fd}$	program
fd	::=	**void** $id(\vec{decl})$ { $\vec{decl}; \vec{stmt}$ }	function definition
$decl$	::=	id	variable declaration
$stmt$	::=		statements
	|	$var \leftarrow expr$	(assignment)
	|	$f(\vec{expr})$	(function call)
$expr$	::=		expressions
	|	$prim(\vec{expr})$	(primitive call)
	|	c	(constant)
	|	& id	(reference)
	|	var	(variable)
var	::=	id	(simple variable)
	|	$\star id$	(dereference)
$prim$	::=		primitive
	|	$\texttt{\~{}}^n_m$ | $\&^n_m$ | $\texttt{\|}^n_m$ | $\texttt{\^{}}^n_m$ |	(bitwise operators)
	|	$+^n_m$ | $-^n_m$ | $\star^n_m$ |	(arithmetic operators)
	|	$\texttt{>>}^n_m$ | $\texttt{<<}^n_m$ | $\texttt{>>>}^n_m$ | $\texttt{<<<}^n_m$ |	(shifts)
	|	$\texttt{shuffle}^n_m$	(shuffle)
	|	$\texttt{bitmask}^n_m$	(bitmask)
	|	$\texttt{pack}^n_m$	(pack)
	|	$\texttt{broadcast}^n_m$	(broadcast)

Figure 3.8: Syntax of PseudoC

In order to evaluate statements, we introduce the function $\texttt{storeval}(M, E, var, c)$, which stores c at address $M(E(var))$ if var is a simple identifier, and at address $M(M(E(var)))$ if var is a pointer dereference (*i.e.*, of the form $*id$):

$$\texttt{storeval}(M, E, *id, c) = M(M(E(id))) \leftarrow c$$
$$\texttt{storeval}(M, E, id, c) = M(E(id)) \leftarrow c$$

Note that a variable on the left-hand side of an assignment cannot be a reference (*i.e.*, of the form $\&id$).

Evaluating an assignment $var \leftarrow e$ (rule ASGN) thus straightforwardly evaluates e and stores the result into in the memory using `storeval`.

When calling a function (rule FUNCALL), the *caller* allocates memory for the parameters and local variables using the $\texttt{alloc\_vars}(M, \vec{decl})$ function. This function allocates a block of type $\texttt{sizeof}(\tau_j)$ bits for each declaration $decl_j : \tau_j$ of $\vec{decl}$ and returns a new environment E' mapping the parameters' identifiers to indices of M', as well as all of the indices $\vec{idx}$ of all the new blocks allocated. The function $\texttt{bind\_params}(E, M, \vec{decl}, \vec{c})$ then binds the formal parameters to the values of the corresponding arguments by iterating the `storeval` function. The body of the function is then evaluated in the new environment E', thus producing a new memory M_3. Finally, the blocks allocated by `alloc_vars` are freed using the $\texttt{free}(M, \vec{idx})$ function.

c	$\in$	$\mathbb{N}$	values (representing both integers and addresses)
Ξ	$::=$	$(id \mapsto fd)*$	map from identifiers to functions
E	$::=$	$(id \mapsto c)*$	map from identifier to addresses
M	$::=$	$(c \mapsto c)*$	map from addresses to values
γ	$::=$	$((op, n, n) \mapsto (\vec{c} \to c))*$	map from primitives to functions

Figure 3.9: PseudoC's semantics elements

$$\boxed{\gamma, E \vdash e, M \Downarrow c}$$

$$\frac{}{\gamma, E \vdash c, M \Downarrow c}\ \textsc{Const} \qquad \frac{}{\gamma, E \vdash *id, M \Downarrow M(M(E(id)))}\ \textsc{Deref}$$

$$\frac{}{\gamma, E \vdash \&id, M \Downarrow E(id)}\ \textsc{Ref} \qquad \frac{}{\gamma, E \vdash id, M \Downarrow M(E(id))}\ \textsc{Var}$$

$$\frac{\gamma, E \vdash e_j, M \Downarrow c_j \qquad \vdash \gamma(prim)(\vec{c}) \Downarrow c}{\gamma, E \vdash prim(\vec{e}), M \Downarrow c}\ \textsc{Prim}$$

$$\boxed{\gamma, \Xi, E \vdash stmt, M \Downarrow M}$$

$$\frac{\gamma, E \vdash e, M \Downarrow c \qquad \texttt{storeval}(M, E, var, c) = M'}{\gamma, \Xi, E \vdash var \leftarrow e, M \Downarrow M'}\ \textsc{Asgn}$$

$$\frac{\begin{array}{c}\gamma, E \vdash e_j, M \Downarrow c_j \\ \texttt{void} f(decl1)\{decl2; \vec{stmt}\} \in \Xi \\ \texttt{alloc\_vars}(M, decl1 \cup decl2) = (E', M_1, \vec{idx}) \\ \texttt{bind\_params}(E', M_1, decl1, \vec{c}) = M_2 \\ \gamma, \Xi, E' \vdash \vec{stmt}, M_2 \Downarrow M_3 \qquad \texttt{free}(M_3, \vec{idx}) = M'\end{array}}{\gamma, \Xi, E \vdash f(\vec{e}), M \Downarrow M'}\ \textsc{FunCall}$$

$$\boxed{\gamma, \Xi, E \vdash \vec{stmt}, M \Downarrow M}$$

$$\frac{\begin{array}{c}\gamma, \Xi, E \vdash stmt_j, M_j \Downarrow M_{j+1} \\ |\vec{stmt}| = n \qquad M_0 = M \qquad M' = M_n\end{array}}{\gamma, \Xi, E \vdash \vec{stmt}, M \Downarrow M'}\ \textsc{Seq}$$

Figure 3.10: Semantics of PseudoC

3.6.3 Translation from Usuba0 to PseudoC

Figure 3.11 describes the translation from Usuba0 to PseudoC. The Usuba0 program is assumed to have been scheduled prior to this translation (using the transformation $\mathcal{S}$ specified in Specification 2). As a result, we simply process equations in textual order. Nodes are translated to procedures, returning values through pointers rather than using a `return` statement.

The return values $\sigma|_{\sharp=o}$ of a node $f \ : \ \sigma = \vec{eqn}$ are compiled to pointer parameters (declared with a $\star$). As a consequence, whenever a variable is used in the $\mathcal{L}$eft-hand side of an assignment, we use the rule $\mathcal{L}^\sigma$ that dereferences pointers (*i.e.*, variables that are in $\sigma|_{\sharp=o}$ of the Usuba0 node we are compiling) if needed. Similarly, the rule $\mathcal{R}^\sigma$ is used to compile expressions (on the $\mathcal{R}$ight-hand side of an assignment), and dereferences pointers.

Additionally, constants and constant variables (that is, variables in $\sigma|_{\sharp=\mathcal{L}}$) must be broadcasted to vectors using a `broadcast` primitive. Intuitively, a SIMD broadcast takes as input a scalar c and creates a vector containing multiple copies of c. The amount of copies is determined by the broadcast instruction (*e.g.*, `_mm256_set1_epi64` creates 4 copies in an AVX vector, `_mm_set1_epi8` creates 16 copies in a SSE vector). The exact semantics of the broadcast generated when translating from Usuba0 to PseudoC will be defined in γ when evaluating the PseudoC program.

Function calls must take pointers to variables to return values. We use the $\mathcal{P}^\sigma$ helper to compile the left-hand side of a function call to $\mathcal{P}$ointers: variables that are in σ_o of the current node are already pointers and we do not need to take their addresses.

Finally, Usuba0's scalar operators are lifted to the architecture `arch` used for the translation using the function $\textit{lift}^m_{\texttt{arch}}(e)$. We introduce the auxiliary functions $\textit{arch\_size}(\texttt{arch})$ to get the size of the vectors of an architecture `arch`:

$$\begin{array}{rcl} \textit{arch\_size}(SSE) & = & 128 \\ \textit{arch\_size}(AVX) & = & 256 \\ \textit{arch\_size}(AVX512) & = & 512 \\ & ... & \end{array}$$

We also introduce the function *arch_op*, which maps Usuba0 operators to the operators of an architecture `arch`, defined as $\textit{arch\_op}(\texttt{arch}, m, op) = op_m^{\textit{arch\_size}(\texttt{arch})}$. *arch_op* is defined for all operators of Usuba0 as well as `broadcast`.
$\textit{lift}^m_{\texttt{arch}}(e)$ is then defined as iterating structurally over e and replacing each operator op with $op_m^{\textit{arch\_size}(\texttt{arch})}$.

3.6.4 Compiler Correctness

We define the function $\textit{arch\_parallelism}(\texttt{arch}, m)$ to be equal to the number of elements of type um that can be packed in a register of the architecture `arch`. This function is defined as $\textit{arch\_parallelism}(\texttt{arch}, m) = \textit{arch\_size}(\texttt{arch})/m$. For instance:

$$\begin{array}{rcll} \textit{arch\_parallelism}(\text{SSE},m) & = & 128/m & \forall m \in \{1,8,16,32,64\} \\ \textit{arch\_parallelism}(\text{AVX},m) & = & 256/m & \forall m \in \{1,8,16,32,64\} \\ \textit{arch\_parallelism}(\text{AVX512},m) & = & 512/m & \forall m \in \{1,8,16,32,64\} \end{array}$$

We also define the function *transpose* which, for a given architecture `arch`, takes a tuple $\vec{c}$ of $k * N$ values of type um where $N = \textit{arch\_parallelism}(\texttt{arch}, m)$, and returns the transpose of $\vec{c}$ in the form of a tuple of k values of type $\textit{vector\_type}(\texttt{arch})$.

$$\boxed{\mathcal{C}_{\texttt{arch}}(\vec{nd}) = \vec{fd}}$$

$$\mathcal{C}_{\texttt{arch}}(\vec{nd}) = \overrightarrow{\mathcal{C}_{\texttt{arch}}(nd)}$$

$$\boxed{\mathcal{C}_{\texttt{arch}}(nd) = fd}$$

$$\mathcal{C}_{\texttt{arch}}(\textbf{node}\ f : \sigma = \vec{eqn}) = \textbf{void}\ f(\mathcal{C}_{\texttt{arch}}(\sigma|_{\flat}), \mathcal{C}_{\texttt{arch}}(\sigma|_{\sharp=o}))\ \{\mathcal{C}_{\texttt{arch}}(\sigma|_{\sharp=\mathcal{L}}); \overrightarrow{\mathcal{C}^{\sigma}_{\texttt{arch}}(eqn)}\}$$

$$\boxed{\mathcal{C}_{\texttt{arch}}(x : \tau) = id}$$

$$\mathcal{C}_{\texttt{arch}}(x \flat \tau) = x$$

$$\mathcal{C}_{\texttt{arch}}(x \sharp \tau) = \begin{cases} x & \text{if } \sharp = \mathcal{L} \\ *\,x & \text{otherwise} \end{cases}$$

$$\boxed{\mathcal{C}_{\texttt{arch}}(eqn) = stmt}$$

$$\mathcal{C}^{\sigma}_{\texttt{arch}}(\vec{x_o} = f(\vec{e})) = f(\mathcal{C}^{\sigma}_{\texttt{arch}}(\vec{e}), \mathcal{P}^{\sigma}(\vec{x_o}))$$

$$\mathcal{C}^{\sigma}_{\texttt{arch}}(x = e) = \mathcal{L}^{\sigma}(x) \leftarrow \mathcal{C}^{\sigma}_{\texttt{arch}}(e)$$

$$\mathcal{C}^{\sigma}_{\texttt{arch}}(x = op(\vec{e}) : \texttt{u}m) = \mathcal{L}^{\sigma}(x) \leftarrow \mathit{lift}^{m}_{\texttt{arch}}(op(\overrightarrow{\mathcal{C}_{\texttt{arch}}(e)}))$$

$$\boxed{\mathcal{C}_{\texttt{arch}}(e) = expr}$$

$$\mathcal{C}^{\sigma}_{\texttt{arch}}(c : \texttt{u}m) = \mathit{lift}^{m}_{\texttt{arch}}(\texttt{broadcast}(c))$$

$$\mathcal{C}^{\sigma}_{\texttt{arch}}(x) = \mathcal{R}^{\sigma}_{\texttt{arch}}(x)$$

with

$$\mathcal{L}^{\sigma}(x) = \begin{cases} *x & \text{if } (x_o :_o \texttt{u}_D m) \in \sigma \\ x & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\sigma}_{\texttt{arch}}(x) = \begin{cases} \mathit{lift}^{m}_{\texttt{arch}}(\texttt{broadcast}, x) & \text{if } (x :_{\mathcal{K}} \texttt{u}_D m) \in \sigma \\ *x & \text{if } (x :_o \tau) \in \sigma \\ x & \text{otherwise} \end{cases}$$

$$\mathcal{P}^{\sigma}(x) = \begin{cases} x & \text{if } (x\ :_o\ \tau) \in \sigma \\ \&x & \text{otherwise} \end{cases}$$

Figure 3.11: Translation rules from Usuba0 to PseudoC

Definition 1 (Valid vectorized support)**.** Let arch be an architecture providing an operator signature $\Theta_{\texttt{arch}}$, and ρ a valid implementation of $\Theta_{\texttt{arch}}$. γ is a valid vectorized support for ρ if and only if,

$$\forall op : \texttt{u}_D m \times ... \times \texttt{u}_D m \to \texttt{u}_D m \in \Theta_{\texttt{arch}}, \forall \vec{c_0}, ..., \vec{c_{N-1}} \in [\![\texttt{u}_D m]\!],$$
$$\gamma(op^N_m)(\textit{transpose}(\vec{c_0}, ..., \vec{c_{N-1}})) = \textit{transpose}(\rho(op)(\vec{c_0}), ..., \rho(op)(\vec{c_{N-1}})$$
$$\text{with } N = \textit{arch\_parallelism}(\texttt{arch}, m)$$

meaning that $\gamma(op)$ is a valid, semantic-preserving, vectorized variant of $\rho(op)$.

We can then formulate the correctness of our translation scheme from Usuba0 to PseudoC as follows:

Conjecture 3 (Translation correctness)**.** *Let* pgm *be a monomorphic* ***Usuba0*** *program that type-checks with respect to an operator signature* $\Theta_{\texttt{arch}}$ *provided by an architecture* arch*. Let* main *be the main node of* pgm*, and let* $\sigma = \texttt{typeof(main)}$ *be its type.*
Let $\Xi = \mathcal{C}_{\texttt{arch}}(\text{pgm})$ *be the compiled program. Let E and M be the initial execution environment defined as* $(E, M) = \texttt{alloc\_vars}(\varnothing, \vec{\sigma|_{\sharp=o}})$*. Assume, for simplicity, that* main*'s inputs are all of type* $\texttt{u}m$*, and let* $N = \textit{arch\_parallelism}(\texttt{arch}, m)$*.*
Let ρ *be a valid implementation of* $\Theta_{\texttt{arch}}$*, and let* γ *be a valid vectorized support of* ρ*.*
We have that, for all constants $\vec{c} \in [\![\sigma|_{\flat=\mathcal{K}}]\!]$*, and for all inputs* $(\vec{v^{in}_o}, ..., \vec{v^{in}_{N-1}}) \in [\![\sigma|_{\sharp=o}]\!]^N$*,*

$$\wedge . \quad \gamma, \Xi, E \vdash \Xi(\texttt{main})(\vec{c}, \textit{transpose}(\vec{v^{in}_o}, ..., \vec{v^{in}_{N-1}}), M \Downarrow M'$$
$$\wedge . \quad M'(E(\sigma|_{\sharp=o})) = \textit{transpose}({}^{\rho}[\![\text{pgm}]\!](\vec{c}, \vec{v^{in}_o}), ..., {}^{\rho}[\![\text{pgm}]\!](\vec{c}, \vec{v^{in}_{N-1}}))$$

meaning that executing the ***PseudoC*** *program processes N values at a time, producing a memory whose content agrees with the* ***Usuba0*** *semantics.*

One of the key elements of our programming model is that we assume that there is an arbitrarily large amount of input blocks to process. This ensures that there are enough input blocks for the compiled program to process N of them in parallel.

Conjecture 3 holds for any monomorphic program *pgm* that type-checks in an operator signature $\Theta_{\texttt{arch}}$. Remember from Conjecture 2 that this implies that *pgm* has a semantics with respect to $\Theta_{\texttt{arch}}$. Compiler correctness follows from Corollary 1 and Conjecture 3 as follows:

Corollary 2 (Compiler correctness)**.** *Let* pgm *be an* ***Usuba***$^{\textit{core}}$ *program that type-checks in the empty context. Let* $\Theta_{\texttt{arch}}$ *be an operator signature provided by an architecture* arch*, and* $\vec{dir}$ *and* $\vec{m}$ *concrete directions and word sizes such that*

$$\mathcal{M}^{\Theta_{\texttt{arch}}}(\text{pgm}) \overset{\vec{dir}, \vec{m}}{\rightsquigarrow} \text{pgm}'$$

Let $\text{pgm}'' = \mathcal{S}(\mathcal{N}(\text{pgm}'))$*.*
Let $\Xi = \mathcal{C}_{\texttt{arch}}(\text{pgm}'')$ *be the compiled program. Then, the semantics of* Ξ *agrees with the semantics of* pgm″ *for any valid implementation* ρ *of* $\Theta_{\texttt{arch}}$*, any valid vectorized support of* ρ*, and any well-formed input.*

Type-classes. Section 2.3 gives an informal semantics of Usuba using type-classes. These type-classes can be understood as the concrete implementations of $\Theta_{\texttt{arch}}$, ρ and γ used by the compiler: for a given architecture `arch`, a type-class provides a collection of operators with a given signature ($\simeq \Theta_{\texttt{arch}}$), a semantics ($\simeq \rho_{\texttt{arch}}$) and an implementation ($\simeq \gamma_{\texttt{arch}}$). Consider, for instance, the Shift type-class for AVX2. It provides the following operator signature $\Theta_{\texttt{AVX2}}$:

$$\begin{array}{lcll} >>_n & : & \mathtt{u}_V16 \rightarrow \mathtt{u}_V16 & \forall n \in [0,15] \\ >>_n & : & \mathtt{u}_V32 \rightarrow \mathtt{u}_V32 & \forall n \in [0,31] \\ >>_n & : & \mathtt{u}_V64 \rightarrow \mathtt{u}_V64 & \forall n \in [0,63] \\ & & \dots & \end{array}$$

It also gives a semantics to each operator ($\rho_{\texttt{AVX2}}$):

$$\begin{array}{lcll} >>_n^{\mathtt{u}_V16 \rightarrow \mathtt{u}_V16} & \rightarrow & \text{16-bit right-shift by } n \text{ places} & \forall n \in [0,15] \\ >>_n^{\mathtt{u}_V32 \rightarrow \mathtt{u}_V32} & \rightarrow & \text{32-bit right-shift by } n \text{ places} & \forall n \in [0,31] \\ >>_n^{\mathtt{u}_V64 \rightarrow \mathtt{u}_V64} & \rightarrow & \text{64-bit right-shift by } n \text{ places} & \forall n \in [0,63] \\ & & \dots & \end{array}$$

Finally, it provides a vectorized semantics of the operators for several architectures (*i.e.*, a vectorized support $\gamma_{\texttt{AVX2}}$ of $\rho_{\texttt{AVX2}}$):

$$\begin{array}{lcll} >>_{16}^{256} & \rightarrow & \text{16 16-bit right-shifts} & (\texttt{vpsllw}) \\ >>_{32}^{256} & \rightarrow & \text{8 32-bit right-shifts} & (\texttt{vpslld}) \\ >>_{64}^{256} & \rightarrow & \text{4 64-bit right-shifts} & (\texttt{vpsllq}) \\ & & \dots & \end{array}$$

Note that type-classes are coarser than strictly necessary: an Usuba code relying on 32-bit right-shifts only requires $\Theta_{\texttt{arch}}$ to provide a 32-bit right-shift, rather than 16-bit, 32-bit and 64-bit left and right shifts and rotations (as the Shift type-class provides). Nonetheless, we chose to group similar operations in type-classes since they are provided together by an architecture: the availability of a 32-bit left-shift implies the availability of a 32-bit right-shift, as well as 32-bit left and right rotations.

3.7 Conclusion

In this chapter, we formally described the Usuba language, from its type system, monomorphization and semantics, to its compilation to an imperative language.

Notably, we provided a semantics of Usuba$^{\text{mono}}$ rather than Usuba, which is not fully satisfying. Several constructions of Usuba are thus specified by elaboration to Usuba$^{\text{core}}$ (*e.g.*, loops, vectors, imperative assignments, lookup tables), and the semantics of polymorphic Usuba$^{\text{core}}$ programs is not defined for programs that do not monomorphize. Furthermore, as we will show in Chapter 6, loops and vectors are essential to produce efficient code on embedded devices. In practice, we thus keep loop and vectors in the pipeline, leaving the compiler free to expand vectors and loops as needed.

Chapter 4

Usubac

The Usubac compiler consists of two phases: the frontend (Section 4.1), whose role is to distill the high-level constructs of Usuba into a low-level minimal subset of Usuba, called Usuba0 (already introduced in Section 3.5, Page 83), and the backend (Section 4.2), which performs optimizations over the core language. In its final pass, the backend translates Usuba0 to C code with intrinsics. The whole pipeline is shown in Figure 4.1. The "Masking" and "tightPROVE$^+$" passes add countermeasures against power-based side-channel attacks in the code generated by Usubac. They are both optional, and are presented in Chapter 6.

4.1 Frontend

The frontend extends the usual compilation pipeline of synchronous dataflow languages [71, 268], namely normalization (Specification 1, Page 83) and scheduling, with domain-specific transformations. Normalization enforces that, aside from nodes, equations are

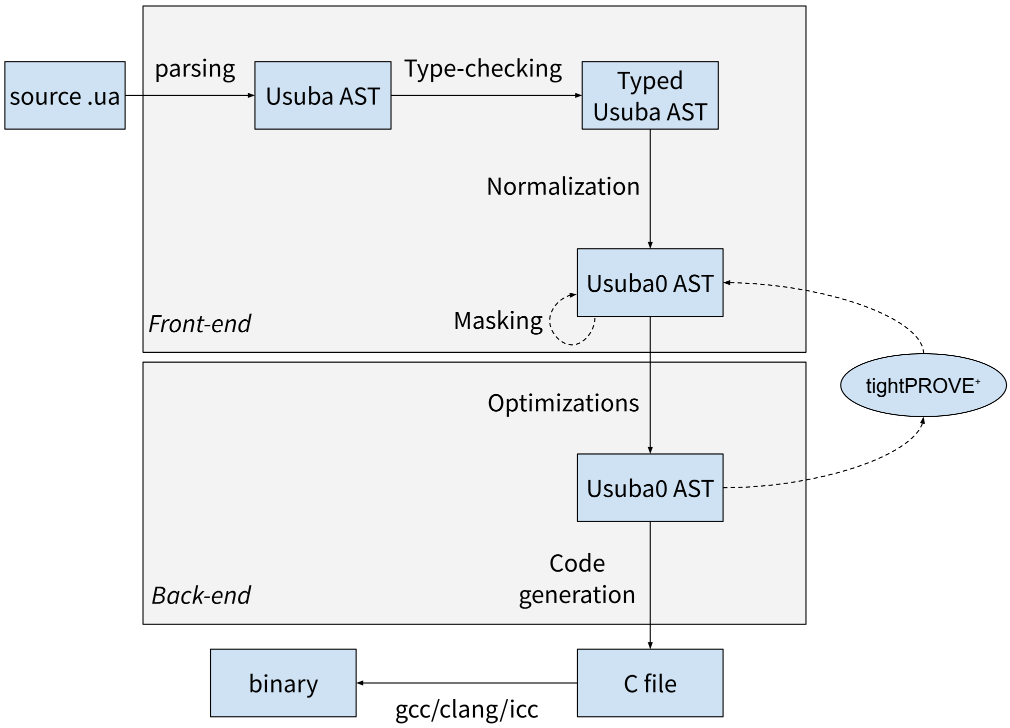


Figure 4.1: Usubac's pipeline

restricted to defining variables at integer type. Scheduling checks that a given set of equations is well-founded and constructively provides an ordering of equations for which a sequential execution yields a valid result. In our setting, scheduling has to be further refined to produce high-performance code: scheduling belongs to the compiler's backend (Section 4.2.5).

Several features of Usuba require specific treatment by the frontend. The language offers domain-specific syntax for specifying cryptographic constructs such as lookup tables or permutations, which boil these constructs down to Boolean circuits (Section 2.2, Page 51). Tuples, vectors, loops and node calls also require some special transformations in order to facilitate both optimizations and generation of C, as we show in the following.

4.1.1 Vectors

Section 3.1 (Page 74) defines vectors through elaboration to Usuba^{core}. However, to achieve good performance on embedded platforms, vectors are essential. Such platforms offer only a few kilobytes of memory, and fully unrolling loops and nodes would produce too large programs. Keeping nodes throughout the pipeline without keeping vectors would produce inefficient code. Consider for instance, consider RECTANGLE's `ShiftRows` node, whose bitslice signature is

```
node ShiftRows (input:b1[4][16]) returns (out:b1[4][16])
```

If `input` and `out` were to be expanded, `ShiftRows` would take 64 parameters as input, which would require 64 instructions to push on the stack, and 64 to pop. Instead, by keep its input as a vector throughout the pipeline, and eventually producing a C array when generating C code, the overhead of calling this function is almost negligible.

Even when targeting Intel CPUs, some ciphers perform better when they are not fully inlined and unrolled, as shown in Section 4.2.4.

Consequently, we keep loops and vectors throughout the Usubac pipeline. Usubac is still free to unroll and expand any array it sees fit, but is under no obligation to do this for all of them. Semantically, Usubac proceeds over vectors as it does over tuples, the difference between the two being only how they are compiled: a vector becomes a C array, while a tuple becomes multiple C variables or expressions.

Node calls. When keeping vectors non-expanded, we need to make sure that the types of node calls arguments are identical to the types of the node parameters. Indeed, from Usuba's standpoint, given a variable `x` of type `b1[2]`, both node calls `f(x[0],x[1])` and `f(x)` are equivalent. Similarly, a variable `a` of type `b1[2][4]` is equivalent to a variable of type `b1[8]`. However, when generating C, those expressions are different, and only one of them can be valid: passing two arguments to a C function that expects only one is not allowed.

Usubac thus ensures that arguments of a node call have the same syntactic type as expected by the node, and that node calls have the same amount of arguments as expected by the node. If not, Usubac expand array until the types are correct. Consider, for instance:

```
node g(a,b:b1[2]) returns (r:b1[2][2])
let
    r[0][0] = a[0];
    r[0][1] = a[1];
    r[1][0] = b[0];
    r[1][1] = b[1];
tel

node f(a:b1[4]) returns (r:b1[4])
let
    r = g(a)
tel
```

Such a program will be normalized by Usubac to:

```
node g(a0,a1,b0,b1:b1) returns (r:b1[4])
let
    r[0] = a0;
    r[1] = a1;
    r[2] = b0;
    r[3] = b1;
tel

node f(a:b1[4]) returns (r:b1[4])
let
    r = g(a[0],a[1],a[2],a[3])
tel
```

4.1.2 Tuples

Operations on Tuples

As shown in Section 2.3 (Page 54), logical and arithmetic operations applied to tuples are unfolded. For instance, the last instruction of RECTANGLE is:

```
cipher = round[25] ^ key[25]
```

where `cipher`, `round[25]` and `keys[25]` are all of type `u16[4]`. Thus, this operation is expanded by the frontend to:

```
cipher[0] = round[25][0] ^ key[25][0]
cipher[1] = round[25][1] ^ key[25][1]
cipher[2] = round[25][2] ^ key[25][2]
cipher[3] = round[25][3] ^ key[25][3]
```

Shifts and rotations involving tuples are performed by the frontend. For instance, the `ShiftRows` node of RECTANGLE contains the following rotation:

```
out[1] = input[1] <<< 1;
```

In bitslice mode, `out[1]` and `input[1]` are both tuples of type `b1[16]`. Usubac thus expands the rotation and this equation becomes:

```
out[1] = (input[1][1],  input[1][2],  input[1][3],  input[1][4],
          input[1][5],  input[1][6],  input[1][7],  input[1][8],
          input[1][9],  input[1][10], input[1][11], input[1][12],
          input[1][13], input[1][14], input[1][15], input[1][0])
```

Tuple Assignments

Assignments between tuples are expanded into assignments from atoms to atoms. This serves three purposes: first, Usubac will eventually generate C code, which does not support tuple assignment. Second, this makes optimizations more potent; in particular copy propagation and common subexpression elimination. Third, doing this before the scheduling pass (Section 4.2.5) allows a tuple assignment to be scheduled non-contiguously. The previous example is thus turned into:

```
(out[1][0],  out[1][1],  out[1][2],  out[1][3],
 out[1][4],  out[1][5],  out[1][6],  out[1][7],
 out[1][8],  out[1][9],  out[1][10], out[1][11],
 out[1][12], out[1][13], out[1][14], out[1][15]) =
             (input[1][1],  input[1][2],  input[1][3],  input[1][4],
              input[1][5],  input[1][6],  input[1][7],  input[1][8],
              input[1][9],  input[1][10], input[1][11], input[1][12],
              input[1][13], input[1][14], input[1][15], input[1][0])
```

which is then expanded into:

```
out[1][0]  = input[1][1];
out[1][1]  = input[1][2];
out[1][2]  = input[1][3];
out[1][3]  = input[1][4];
out[1][4]  = input[1][5];
out[1][5]  = input[1][6];
out[1][6]  = input[1][7];
out[1][7]  = input[1][8];
out[1][8]  = input[1][9];
out[1][9]  = input[1][10];
out[1][10] = input[1][11];
out[1][11] = input[1][12];
out[1][12] = input[1][13];
out[1][13] = input[1][14];
out[1][14] = input[1][15];
out[1][15] = input[1][0];
```

4.1.3 Loop Unrolling

As mentioned in Section 3.7 (Page 90), Usubac is free to keep loops throughout its pipeline. The decision to inline loops or not is made in the backend, and will be discussed in Section 4.2.3. However, in two cases, unrolling is necessary to normalize Usuba down to Usuba0, and is thus performed by the frontend. The first case corresponds to shifts and rotations on tuples that depends on loop variables. For instance,

```
forall i in [1, 2] {
    (x0,x1,x2) := (x0,x1,x2) <<< i;
}
```

Since rotations on tuples are resolved at compile time, this rotation requires the loop to be unrolled into

```
(x0,x1,x2) := (x0,x1,x2) <<< 1;
(x0,x1,x2) := (x0,x1,x2) <<< 2;
```

which is then simplified to

```
(x0,x1,x2) := (x1,x2,x0);
(x0,x1,x2) := (x2,x0,x1);
```

which will be optimized away by copy propagation in the backend.

Unrolling is also needed to normalize calls to nodes from arrays of nodes within loops. This is for instance the case with SERPENT, which uses a different S-box for each round, and whose main loop is thus:

```
forall i in [0,30] {
    state[i+1] = linear_layer(sbox<i%8>(state[i] ^ keys[i]))
}
```

which, after unrolling, becomes:

```
state[1] = linear_layer(sbox0(state[0] ^ keys[0]))
state[2] = linear_layer(sbox1(state[1] ^ keys[1]))
state[3] = linear_layer(sbox2(state[2] ^ keys[2]))
...
```

Note that we chose to exclude both shifts on tuples and arrays of nodes from Usuba0 because they would generate suboptimal C code, which would rely on the C compilers to be willing to optimize away those constructs. For instance, we could have introduced conditionals in Usuba in order to normalize the first example (tuple rotation) to

```
forall i in [1,2] {
    if (i == 1) {
        (x0,x1,x2) := (x1,x2,x0);
    } elsif (i == 2) {
        (x0,x1,x2) := (x2,x0,x1);
    }
}
```

And the second example (array of nodes) to

```
forall i in [0,30] {
    if (i % 8 == 0) {
        state[i+1] = linear_layer(sbox0(state[i] ^ keys[i]))
    } elsif (i % 8 == 1) {
        state[i+1] = linear_layer(sbox1(state[i] ^ keys[i]))
    } elsif
        ...
    }
}
```

This Usuba0 code would then have been compiled to C loops. However, to be efficient, it would have relied on the C compiler unrolling the loop in order to remove the conditionals and optimize (*e.g.,* with copy propagation) the resulting code. In practice, C compilers avoid unrolling large loop, and performing the unrolling in Usuba leads to better and more predictable performance.

4.1.4 Flattening From *m*slicing to Bitslicing

We may also want to flatten a *m*sliced cipher to a purely bitsliced model. Performance-wise, it is rarely (if ever) interesting: the higher register pressure imposed by bitslicing is too detrimental. However, some architectures (such as 8-bit microcontrollers) do not offer vectorized instruction sets at all. Also, bitsliced algorithms can serve as the basis for hardening software implementations against fault attacks [193, 240]. To account for these use-cases, Usubac can automatically flatten a cipher into bitsliced form. Flattening is a whole-program transformation (triggered by passing the flag `-B` to Usubac) that globally rewrites all instances of vectorial types $\texttt{u}_D m \times n$ into the type $\texttt{b}m[n]$ (shorthand for $\texttt{u}_D 1 \times m[n]$, equivalent to $\texttt{u}_D 1[n][m]$). Note that the vectorial direction of the source is irrelevant: it collapses after flattening. The rest of the source code is processed as-is: we rely solely on ad-hoc polymorphism to drive the elaboration of the operators at the rewritten types. Either type checking and monomorphization succeeds, in which case we have obtained the desired bitsliced implementation, or monomorphization fails, meaning that the given program exploits operators that are not available in bitsliced form. For instance, short of providing an arithmetic instance on `b32`, we will not be able to bitslice an algorithm relying on addition on u32 (such as CHACHA20).

4.2 Backend

The backend of Usubac takes care of optimizing the Usuba0 code and ultimately generating C code.
Targeting C code allows us to rest on the optimizers of C compilers to produce efficient code. However, that alone would not be sufficient to achieve throughputs on par with carefully hand-tuned assembly code.

We divide our optimizations in two categories. The simple ones (common subexpression elimination, inlining, unrolling) are already done by most C compilers, but we still perform them in Usuba, mainly in order to improve the potency of the more advanced optimizations, but also to not rely on the heuristic of C compilers, which were not tailored for sliced code. The more advanced optimizations include two scheduling algorithms (Section 4.2.5), and an interleaving pass (Section 4.2.6). They are key to generate efficient code on SIMD architectures.

Those advanced optimizations can only be done thanks to the invariants offered by the Usuba programming model. One of our scheduling algorithms is thus tailored to reduce spilling in linear layers of bitsliced ciphers, while the other one improves instruction-level parallelism by mixing linear layers and S-boxes in *m*sliced ciphers.

Usuba's dataflow programming model is also key for our interleaving optimization: since node inputs are taken to be (infinite) streams rather than scalars, we are able to increase instruction-level parallelism by processing several elements of the input stream simultaneously (Section 4.2.6).

4.2.1 Autotuning

The impact of some optimizations is hard to predict. For example, inlining reduces the overhead of calling functions, but increases register pressure and produces code that does not fully exploit the μop cache (DSB). Our interleaving algorithm improves instruction-level parallelism, at the expense of register pressure. Our scheduling algorithm for bitsliced code tries to reduce register pressure, but sometimes increases register pressure instead. Our scheduling algorithm for *m*sliced code increases instruction-level parallelism, but this sometimes either increase register pressure or simply produces code that the C compiler is less keen to optimize.

Fortunately, with Usuba, we are in a setting where:

- The compilation time is not as important as with traditional C compilers. The generated ciphers will likely be running for a long period of time, and need to be optimized for performance in order not to bottleneck applications. Also, ciphers are made of at most dozens of thousands of lines of code, are can thus be executed in a few milliseconds at most.
- The execution time is independent of the inputs, by construction. While benchmarking a generic C program requires a representative workload (at the risk of missing a frequent case), every run of a Usuba program will have the same execution time, regardless of its inputs.

Given the complexity of today's microarchitectures, statically selecting heuristics for the optimizations is hard. We minimize choices a priori by using an autotuner [299, 140], which enables or disables optimizations based on accurate data. The autotuner thus benchmarks heuristic optimizations (inlining, unrolling, scheduling, interleaving), and keeps only those that improve the execution time (for a given cipher).

4.2.2 Common Subexpression Elimination, Copy Propagation, Constant Folding

! The benchmarks of this section are available at:
`https://github.com/DadaIsCrazy/usuba/tree/master/experimentations/code-size-CSECPCF`

Common subexpression elimination (CSE) is a classical optimization that aims at preventing identical expressions from being computed multiple times. For instance,

```
x = a + b;
y = (a + b) * c;
```

is transformed by CSE into

```
x = a + b;
y = x * c;
```

Copy propagation is also a traditional optimization that removes assignments of a variable into another variable. For instance,

```
x = a + b;
y = x;
z = y * c;
```

is transformed by copy propagation into

```
x = a + b;
z = x * c;
```

Finally, constant folding consists in computing constant expressions at compile time. The expressions that Usuba simplifies using constant folding are either arithmetic or bitwise operations between constants (*e.g.*, replacing `x = 20 + 22` by `x = 42`) or bitwise operations whose operand is either `0` or `0xffffffff` (*e.g.*, replacing `x = a | 0` by `x = a`).

CSE, copy propagation and constant folding are already done by C compilers. However, performing them in Usubac has two main benefits:

- It produces smaller and more readable C code. A non-negligible part of the copies removed by copy propagation comes from temporary variables introduced by the compiler itself. Removing such assignments improves readability. This matters particularly in bitsliced code, which can contain tens of thousands lines of C code after these optimizations: on average, these optimizations reduce by 67% the number of C instructions of the bitsliced ciphers generated by Usuba. For instance, bitsliced ACE goes from 200.000 instructions without these optimizations to only 50.000 with them.

- It makes the scheduling optimizations (Section 4.2.5) more potent, since needless assignments and redundant expressions increase the number of live variables, which may throw off the schedulers.

4.2.3 Loop Unrolling

Section 4.1.3 (Page 94) showed that Usubac unrolls loops in some cases as part of the normalization. The backend of Usubac automatically unrolls all loops by default. Experimentally, this produce the most efficient code. The user can still use the flag `-no-unroll` to disable nonessential unrolling (*i.e.,* unrolling that is not required by the normalization). In the following, we explain the reasoning behind the aggressive unrolling performed by Usubac.

Loop unrolling is clearly beneficial for bitsliced ciphers as almost all ciphers use either permutations, or shifts or rotations to implement their linear layers. After unrolling (and only in bitslice mode), those operations can be optimized away by copy propagation at compile time.

For *m*sliced code, the rational for unrolling is more subtle. Very small loops are always more efficient when unrolled since the overhead of looping negatively impacts execution time.

Furthermore, unrolling small and medium-sized loops is often beneficial as well because it allows our scheduling algorithm to be more efficient. Most loops contain data dependencies from one iteration to the next one. Unrolling enables the scheduler to perform interleaving (Section 4.2.6) to improve performance. Section 4.2.5 also shows the example of ACE, which is bottlenecked by a loop and that our scheduling algorithm is able to optimize by interleaving 3 loops (provided that they are unrolled).

We may want to keep large loops in the final code in order to reduce code size and maximize the usage of the μop cache (DSB). For instance, in CHACHA20, unrolling the main loop (*i.e.,* the one that calls the round function) does not offer any performance improvement (nor does it decrease performance for that matter). However, it is hard to choose an upper bound above which unrolling should not be done. For instance, the PYJAMASK cipher contains a matrix multiplication in a loop:

```
forall i in [0, 3] {
    output[i] = mat_mult(M[i], input[i]);
}
```

After inlining, `mat_mult` becomes 160 instructions, which is more than the number of instructions in RECTANGLE's round (15), or in ASCON's (32) or in CHACHA20's (96). However, those instructions are plagued by data dependencies, and unrolling the loop in Usuba (Clang chooses not to unroll it on its own) speeds up PYJAMASK by a factor 1.68.

4.2.4 Inlining

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/inlining`

The decision of inlining nodes is partly justified by the usual reasoning applied in general-purpose programming languages: a function call implies a significant overhead that, for very frequently called functions (such as S-boxes, in our case) compensates for the increase in code size. Furthermore, Usuba's *m*sliced code scheduling algorithm (Section 4.2.5) tries to interleave nodes to increase instruction-level parallelism, and requires them to be inlined. We thus perform some inlining in Usuba. Figure 4.2 shows the speedups gained by inlining all nodes on some *m*sliced ciphers, compared to inlining none, on general purpose x86 registers and AVX2 registers.

The impact of inlining depends on which C compiler is used, and the architecture targeted. For instance, inlining every node of XOODOO speeds it up by a factor 1.61 on

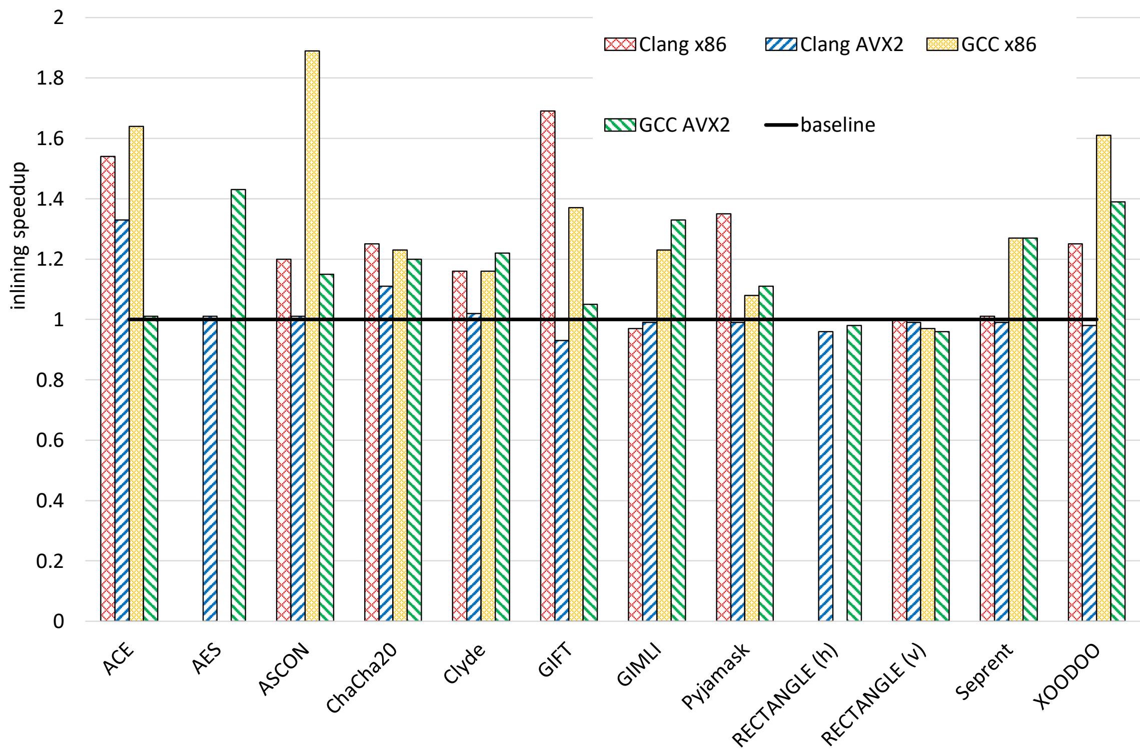


Figure 4.2: Impact on throughput of fully inlining *m*sliced ciphers
(raw numbers in Table A.1, Page 195)

general purpose x86 register when compiling with GCC, but slows it down by a factor 0.98 on AVX2 registers when compiling with Clang. Overall, inlining every node is generally beneficial for performance, delivering speedups of up to $\times 1.89$ (ASCON on general-purpose x86 registers with GCC), but can sometimes be detrimental and reduce performance by a few percents.

On AVX2 registers, when compiling with Clang, inlining tends to be slightly detrimental, as can be seen from the lower half of the third column. Those ciphers are the ones that benefit the less from our scheduling optimization, as can be seen from Figure 4.6 (Page 109). One of the reasons for this performance regression is the fact that fully inlined AVX code does not use the μop cache (DSB), but falls back to the MITE. On GIFT compiled with Clang, for instance, when all nodes are inlined, almost no μops are issued by the DSB, while when no node is inlined, 85% of the μops come from the DSB. This translates directly into a reduced number of instructions per cycle (IPC): the inlined version has an IPC of 2.58, while the non-inlined version is at 3.25. The translates into a mere 0.93 slowdown however, because the fully inlined code still contains 15% fewer instructions. This is less impactful on general-purpose registers than on AVX because their instructions are smaller, and the MITE can thus decode more of them each cycle.

Inlining also improves performance of several ciphers that do not benefit from our scheduling algorithms. For instance, scheduling improves the performance of GIFT, CLYDE, XOODOO and CHACHA20 on general-purpose registers by merely $\times 1.01$, $\times 1.02$, $\times 1.03$ and $\times 1.05$ (Figure 4.6, Page 109), and yet, fully inlining these ciphers speeds them up by $\times 1.69$, $\times 1.16$, $\times 1.25$ and $\times 1.25$ (with Clang). In these cases, both Clang and GCC chose not to be too aggressive on inlining, probably in order not to increase code size too much, but this came at the expense of performance.

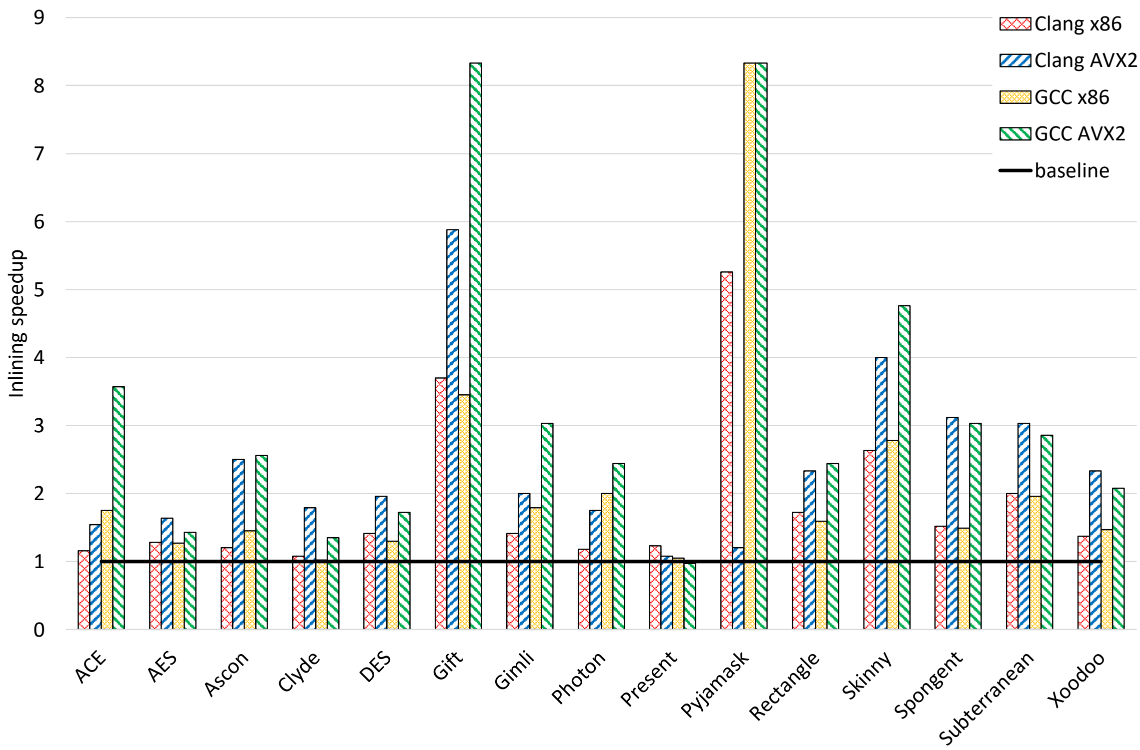


Figure 4.3: Impact on throughput of fully inlining bitsliced ciphers (raw numbers in Table A.3, Page 196)

Bitslicing, however, definitely confuses the inlining heuristics of C compilers. A bitsliced Usuba node is compiled to a C function taking hundreds of variables as inputs and outputs (depending on Usubac's frontend ability to keep vectors in parameters or not). For instance, the round function of DES takes 120 arguments once bitsliced. Calling such a function requires the caller to push hundreds of variables onto the stack while the callee has to go through the stack to retrieve them, leading to a significant execution overhead and code size increase. Similarly, a permutation may be compiled into a function that takes hundreds of arguments and just does assignments, while once inlined, it is virtually optimized away by copy propagation. C compilers avoid inlining such functions because their code is quite large, missing the fact that they would be later optimized away. The performance impact of inlining in bitsliced ciphers is shown in Figure 4.3.

Inlining improves performance in all cases, reaching an impressive $\times 8$ speedup for PYJAMASK on general-purpose registers with GCC. While some of the improvements are explained by the scheduling opportunities enabled by inlining, most of them are due to the overhead saved by not calling functions, and by copy propagation being able to remove unnecessary assignments. One of the takeaways from our (*m*slice and bitslice) inlining benchmarks is that heuristics of C compilers for inlining are not suited to Usuba-generated sliced code.

4.2.5 Scheduling

The C programs generated by Usubac are translated to assembly using C compilers. While a (virtually) unlimited amount of variables can be used in C, CPUs only offer a few registers to work with (between 8 and 32 for commonly used CPUs). C variables are thus mapped to assembly registers using a *register allocation* algorithm. When more variables are alive than there are available registers, some variables must be *spilled*: the stack is used to temporarily store them.

Register allocation is tightly coupled with *instruction scheduling*, which consists in ordering instructions so as to take advantage of a CPU's superscalar microarchitecture [144, 57]. To illustrate the importance of instruction scheduling, consider the following sequence of instructions:

```
u = a + b;
v = u + 2;
w = c + d;
x = e + f;
```

Consider a hypothetical in-order CPU that can compute two additions per cycle. Such a CPU would execute `u = a + b` in its first cycle, and would be unable to compute `v = u + 2` in the same cycle since `u` has not been computed yet. In a second cycle, it would compute `v = u + 2` as well as `w = c + d`, and, finally, in a third cycle, it would compute `x = e + f`. On the other hand, if the code had been scheduled as:

```
u = a + b;
w = c + d;
v = u + 2;
x = e + f;
```

it could have been executed in only 2 cycles. While out-of-order CPUs reduce the impact of such data hazards at run-time, these phenomenons still occur and need to be taken into account when statically scheduling instructions.

The interaction between register allocation and instruction scheduling is partially due to spilling, which introduces memory operations. The cost of those additional memory operations can sometimes be reduced by using an efficient instruction scheduling. Combining register allocation and instruction scheduling can produce more efficient code that performing them separately [291, 147, 243, 222]. However, determining the optimal register allocation and instruction scheduling for a given program is NP-complete [275], and heuristic methods are thus used. The most classical algorithm to allocate registers are based on graph coloring [102, 102, 93, 281], and bound to be approximate since graph coloring itself is NP-complete.

Furthermore, C compilers strive to still offer small compilation times, sometimes at the expense of the quality of the generated code. Both GCC and LLVM thus perform instruction scheduling and register allocation separately, starting instruction scheduling, followed by register allocation, followed by some additional scheduling.

Despite reusing traditional graph coloring algorithms, GCC and LLVM's instruction schedulers and register allocators are meticulously tuned to produce highly efficient code, and are the product of decades of tweaking. GCC's register allocator and instruction scheduler thus amount to more than 50.000 lines of C code, and LLVM's to more than 30.000 lines of C++ code.

We cash out on both these works by providing two pre-schedulers, in order to produce C code that will be better optimized by C compilers' schedulers and register allocators. The first scheduler reduces register pressure in bitsliced ciphers, while the second increases instruction-level parallelism in *m*sliced ciphers.

Bitsliced Scheduler

! The benchmarks of this section are available at:
`https://github.com/DadaIsCrazy/usuba/tree/master/bench/scheduling-bs` and at:
`https://github.com/DadaIsCrazy/usuba/tree/master/experimentations/spilling-bs`

In bitsliced code, the major bottleneck is register pressure: a significant portion of the execution time is spent spilling registers to and from the stack. Given that hundreds of variables can be alive at the same time in a bitsliced cipher, it is not surprising that C compilers have a hard time keeping register pressure down. For instance, it is common for bitsliced ciphers to have up to 60% of their instructions being loads and stores for spilling, as shown in Figure 4.4. We thus designed a scheduling algorithm that aims at reducing register pressure (and improve performance) in bitsliced code.

Let us take the example of RECTANGLE to illustrate why bitsliced code has so much spilling and how this can be improved. RECTANGLE's S-box can be written in Usuba as:

```
node sbox (a0, a1, a2, a3 : u32)
     returns  (b0, b1, b2, b3 : u32)
vars
    t1, t2, t3,  t4,  t5,  t6,
    t7, t8, t9, t10, t11, t12 : u32
let
    t1 = ~a1;
    t2 = a0 & t1;
    t3 = a2 ^ a3;
    b0 = t2 ^ t3;
    t5 = a3 | t1;
    t6 = a0 ^ t5;
    b1 = a2 ^ t6;
    t8 = a1 ^ a2;
    t9 = t3 & t6;
    b3 = t8 ^ t9;
    t11 = b0 | t8;
    b2 = t6 ^ t11
tel
```

Usubac's naive automatic bitslicing (Section 4.1.4, Page 95) would replace all `u32` with `b32`, which would cause the variables to become vectors of 32 boolean elements, thus causing the operators to be unfolded as follows:

```
node sbox (a0, a1, a2, a3 : b32)
     returns  (b0, b1, b2, b3 : b32)
vars ...
let
    t1[0] = ~a1[0];
    t1[1] = ~a1[1];
    t1[2] = ~a1[2];
    ...
    t1[31] = ~a1[31];
    t2[0] = a0[0] & t1[0];
    t2[1] = a0[1] & t1[1];
    ...
    t2[31] = a0[31] & t1[31];
    ...
```

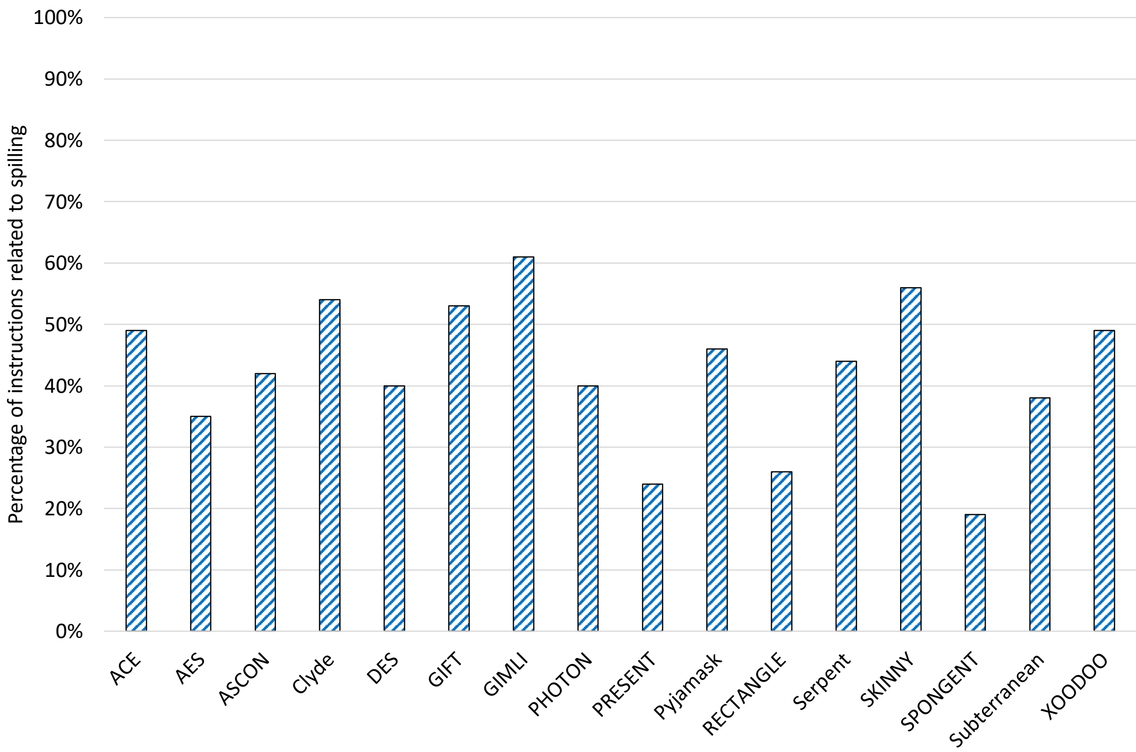


Figure 4.4: Spilling in Usuba-generated bitsliced ciphers, without scheduling (compiled with Clang 7.0.0)

Optimizing the automatic bitslicing While the initial S-box contained fewer than 10 variables simultaneously alive, the bitsliced S-box contains 32 times more variables alive at any point. A first optimization to reduce register pressure thus happens during automatic bitslicing: Usubac detects nodes computing only bitwise instructions and calls to nodes with the same property. These nodes are specialized to take `b1` variables as input, and their call sites are replaced with several calls instead of one. In the case of RECTANGLE, the shorter S-box is thus called 32 times rather than calling the large S-box once. If the initial pre-bitslicing S-box call was:

```
(x0, x1, x2, x3) = sbox(y0, y1, y2, y3)
```

with `x0, x1, x2, x3, y0, y1, y2, y3` of type `u32`. Then, the naive bitsliced call would remain the same except that variables would be typed `b32`, and `sbox` would be the modified version that takes `b32` as inputs and repeats each operation 32 times. The optimized bitsliced version, however, is:

```
(x0[0], x1[0], x2[0], x3[0]) = sbox1(y0[0], y1[0], y2[0], y3[0]);
(x0[1], x1[1], x2[1], x3[1]) = sbox1(y0[1], y1[1], y2[1], y3[1]);
...
(x0[31], x1[31], x2[31], x3[31]) = sbox1(y0[31], y1[31], y2[31], y3[31]);
```

where `sbox1` calls the initial `sbox` node specialized with input types `b1`.

Concretely, this optimization reduces register pressure at the expense of performing more function calls. The improvements heavily depend on the ciphers and C compilers used. On AVX, when using Clang, this optimization yields a 34% speedup on ASCON, 12% on GIFT and 6% on CLYDE.

The same optimization cannot be applied to linear layers, however, as they almost always contain rotations, shifts or permutations, which manipulate large portions of the cipher state. Consider, for instance, RECTANGLE's linear layer:

```
node ShiftRows (input:u16x4) returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
    tel
```

Using a `u1x4` input instead of a `u16x4` is not possible due to the left rotations, which requires all 16 bits of each input. Instead, this function will be bitsliced and become:

```
node ShiftRows (input:b1[4][16]) returns (out:b1[4][16])
let
    out[0][0] = input[0][0];        // out[0] = input[0]
    out[0][1] = input[0][1];
    ...
    out[0][15] = input[0][15];
    out[1][0] = input[1][1];        // out[1] = input[1] <<< 1
    out[1][1] = input[1][2];
    ...
    out[1][15] = input[1][0];
    out[2][0] = input[2][12];       // out[2] = input[2] <<< 12
    out[2][1] = input[2][13];
    ...
    out[2][15] = input[2][11];
    out[3][0] = input[3][13];       // out[3] = input[3] <<< 13
    out[3][1] = input[3][14];
    ...
    out[3][15] = input[3][12];
tel
```

In the case of RECTANGLE's `ShiftRows` node, it can be inlined and entirely removed using copy propagation since it only contains assignments. However, for some other ciphers such as ASCON, SERPENT, CLYDE or SKINNY, the linear layers contains `xor`s, which cannot be optimized away. Similarly, the `AddRoundKey` step of most ciphers introduces a large number of consecutive `xor`s.

Overall, after bitslicing, the main function of Rectangle thus becomes:

```
state := AddRoundKey(state, key[0]);    // <--- lots of spilling in this node
state[0..3] := Sbox(state[0..3]);
state[4..7] := Sbox(state[4..7]);
...
state[60..63] := Sbox(state[60..63]);
state := LinearLayer(state);            // <--- lots of spilling in this node
state := AddRoundKey(state, key[1]);    // <--- lots of spilling in this node
state[0..3] := Sbox(state[0..3]);
state[4..7] := Sbox(state[4..7]);
...
state[60..63] := Sbox(state[60..63]);
...
```

Bitslice code scheduling We designed a scheduling algorithm that aims at interleaving the linear layers (and key additions) with S-box calls in order to reduce the live ranges of some variables and thus reduce the need for spilling.

This algorithm requires a first inlining step that inlines linear nodes (that is, nodes made of only linear operations: `xor`s, assignments, and calls to other linear nodes). After this inlining step, RECTANGLE's code would thus be:

```
state[0] := state[0] ^ key[0][0];         // AddRoundKey 1
state[1] := state[1] ^ key[0][1];
state[2] := state[2] ^ key[0][2];
state[3] := state[3] ^ key[0][3];
state[4] := state[4] ^ key[0][4];
state[5] := state[5] ^ key[0][5];
state[6] := state[6] ^ key[0][6];
state[7] := state[7] ^ key[0][7];
state[8] := state[8] ^ key[0][8];
state[9] := state[9] ^ key[0][9];
...
state[63] := state[63] ^ key[0][63];
state[0..3] := Sbox(state[0..3]);         // S-boxes 1
state[4..7] := Sbox(state[4..7]);
state[8..11] := Sbox(state[8..11]);
...
state[60..63] := Sbox(state[60..63]);
state[0] := state[0];                     // Linear layer 1
state[1] := state[1];
state[2] := state[2];
state[3] := state[3];
state[4] := state[4];
state[5] := state[5];
state[6] := state[6];
...
state[63] := state[60];
state[0] := state[0] ^ key[1][0];         // AddRoundKey 2
state[1] := state[1] ^ key[1][1];
state[2] := state[2] ^ key[1][2];
state[3] := state[3] ^ key[1][3];
state[4] := state[4] ^ key[1][4];
state[5] := state[5] ^ key[1][5];
...
state[63] := state[63] ^ key[1][63];
state[0..3] := Sbox(state[0..3]);         // S-boxes 2
state[4..7] := Sbox(state[4..7]);
...
```

The inlined linear and key addition introduce a lot of spilling since they use a lot of variables a single time. After our scheduling algorithm, RECTANGLE's code will be:

```
state[0] := state[0] ^ key[0][0];         // Part of AddRoundKey 1
state[1] := state[1] ^ key[0][1];
state[2] := state[2] ^ key[0][2];
state[3] := state[3] ^ key[0][3];
state[0..3] := Sbox(state[0..3]);         // S-box 1
state[4] := state[4] ^ key[0][4];         // Part of AddRoundKey 1
state[5] := state[5] ^ key[0][5];
state[6] := state[6] ^ key[0][6];
state[7] := state[7] ^ key[0][7];
state[4..7] := Sbox(state[4..7]);         // S-box 1
...
...
state[0] := state[0];                     // Part of Linear layer 1
state[1] := state[1];
state[2] := state[2];
state[3] := state[3];
state[0] := state[0] ^ key[1][0];         // Part of AddRoundKey 2
state[1] := state[1] ^ key[1][1];
```

```
state[2] := state[2] ^ key[1][2];
state[3] := state[3] ^ key[1][3];
state[0..3] := Sbox(state[0..3]);        // S-box 2
state[4] := state[4];                    // Part of Linear layer 1
state[5] := state[5];
state[6] := state[6];
state[7] := state[7];
state[4] := state[4] ^ key[1][4];        // Part of AddRoundKey 2
state[5] := state[5] ^ key[1][5];
state[6] := state[6] ^ key[1][6];
state[7] := state[7] ^ key[1][7];
state[4..7] := Sbox(state[0..3]);        // S-box 2
...
...
```

Our scheduling algorithm (Algorithm 1) aims at reducing the lifespan of variables computed in the linear layers by interleaving linear layer with S-boxes calls: the parameters of the S-boxes are computed as late as possible, thus removing the need to spill them.

Algorithm 1 Bitsliced code scheduling algorithm

```
1: procedure SCHEDULE(prog)
2:     for each function call funCall of prog do
3:         for each variable v in funCall's arguments do
4:             if v's definition is not scheduled yet then
5:                 schedule v's definition (and dependencies) next
6:         schedule funCall next
```

The optimization of automatic bitslicing is crucial for this scheduling algorithm to work: without it, the S-box is a single large function rather than several calls to small S-boxes, and no interleaving can happen. Similarly, the inlining pass that happens before scheduling itself is essential to ensure that the instructions of the linear layer are inlined and ready to be interleaved with calls to the S-boxes.

Performance We evaluated this algorithm on 11 ciphers containing a distinct S-box and linear layer, where the S-box only performs bitwise instructions. We compiled the generated C code with GCC 8.3.0 and Clang 7.0.0, and ran the benchmarks on a Intel i5 6500. The results are shown in Figure 4.5.

Our scheduling algorithm is clearly beneficial, yielding speedups of up to $\times 1.35$. Some ciphers (*e.g.,* ACE, GIMLI, SPONGENT, SUBTERRANEAN, XOODOO) do not have a clearly distinct S-box and linear layer, and we thus do not evaluate our scheduling algorithm on them. In practice, Usubac will try to apply the scheduling algorithm, leaving the autotuner to judge whether the resulting code is more or less efficient.

The exact benefits of this scheduling algorithm significantly vary from one cipher to the other, from one architecture to the other, and from one compiler to the other. For instance, while on general purpose register our algorithm improves AES's performance by a factor 1.04, it actually reduces it by 0.99 on AVX2 registers. On PRESENT, our algorithm is clearly more efficient on general-purpose registers, while on PYJAMASK, it is clearly better on AVX2 registers.

There does not seem to be a single root cause explaining the performance improvement or degradation observed in our benchmarks. Once again, we rely on the autotuner to make the final decision.

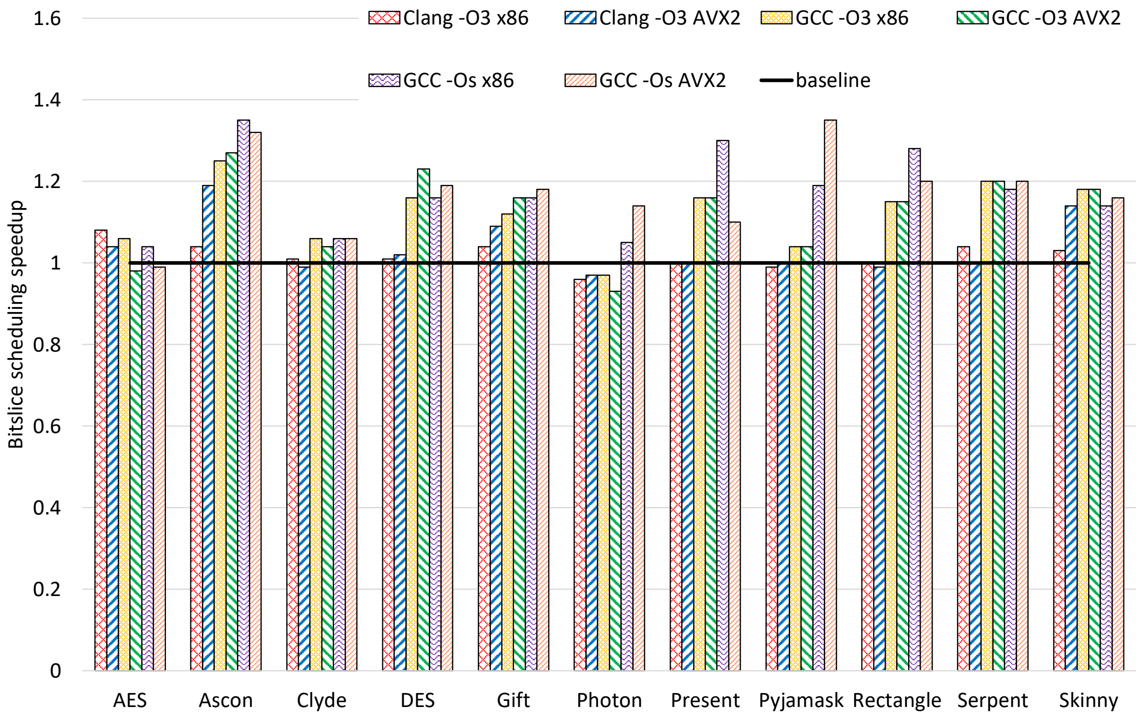


Figure 4.5: Performance of the bitsliced code scheduling algorithm (raw numbers in Table A.4, Page 196)

*m*slice Scheduler

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/scheduling`

*m*sliced programs have much lower register pressure than bitsliced programs. When targeting deeply-pipelined superscalar architectures (like high-end Intel CPUs), spilling is less of an issue, the latency of the few resulting load and store operations being hidden by the CPU out-of-order execution pipeline. Instead, the challenge consists in being able to saturate the CPU execution units. To do so, one must increase instruction-level parallelism (ILP), taking into account data hazards. Consider, for instance, CLYDE's linear layer:

```
node lbox(x,y:u32) returns (xr,yr:u32)
vars
    a, b, c, d: u32
let
    a  = x ^ (x >>> 12);
    b  = y ^ (y >>> 12);
    a := a ^ (a >>> 3);
    b := b ^ (b >>> 3);
    a := a ^ (x >>> 17);
    b := b ^ (y >>> 17);
    c  = a ^ (a >>> 31);
    d  = b ^ (b >>> 31);
    a := a ^ (d >>> 26);
    b := b ^ (c >>> 25);
    a := a ^ (c >>> 15);
    b := b ^ (d >>> 15);
    (xr, yr) = (a,b)
tel
```

Only 2 instructions per cycles can ever be computed because of data dependencies: `x >>> 12` and `y >>> 12` during the first cycle, then `x ^ (x >>> 12)` and `y ^ (y >>> 12)` in the second cycle, `a >>> 3` and `b >>> 3` in the third one, *etc.*However, after inlining, it may be possible to partially interleave this function with other parts of the cipher (for instance the S-box or the key addition).

One may hope for the out-of-order nature of modern CPU to alleviate this issue, allowing the processor to execute independent instructions ahead in the execution streams. However, due to the bulky nature of sliced code, we empirically observe that the reservation station is quickly saturated, preventing actual out-of-order execution. Unfortunately, C compilers have their own heuristic to schedule the instructions, and they often fail to generate code that is optimal with regard to data hazards.

Algorithm 2 *m*slice code scheduling algorithm

```
procedure SCHEDULE(node)
    lb_window = make_fifo(size = 10)
    remaining = node.equations
    scheduled = empty list
    while remaining is not empty do
        found = false
        for each equation eqn of remaining do
            if eqn has no dependency with any equation of lb_window then
                add eqn to scheduled
                add eqn to lb_window
                remove eqn from remaining
                found = true
                break
        if found == false then remove the oldest instruction from lb_window
    return scheduled
```

Our *m*slice scheduling algorithm (Algorithm 2) statically increases ILP by maintaining a look-behind window of the previous 10 instructions. To schedule an instruction, we pick one with no data hazard with the instructions in the look-behind window. If no such instruction can be found, we reduce the size of the look-behind window, and, ultimately just schedule any instruction that is ready. While C compilers may undo this optimization when they perform their own scheduling, we observe significant performance improvements by explicitly performing this first scheduling pass ourselves, as shown in Figure 4.6.

This algorithm is particularly useful with ciphers that involve functions with heavy data dependencies. For instance, consider ACE's `f` node:

```
node f(x:u32) returns (y:u32)
let
    y = ((x <<< 5) & x) ^ (x <<< 1)
tel
```

It contains only 4 instructions, but cannot execute in fewer than 3 cycles due to data dependencies. However, looking at the bigger picture of ACE, this function is wrapped inside the so-called `Simeck-box`:

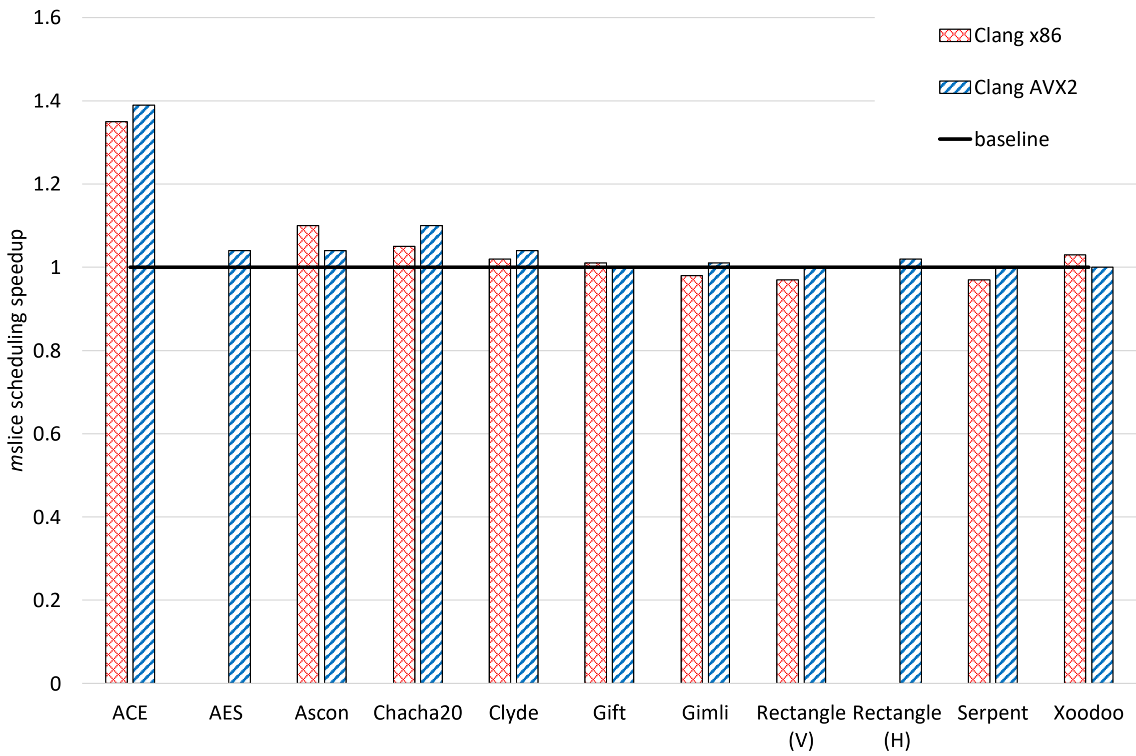


Figure 4.6: Performance of the *m*sliced code scheduling algorithm
(raw numbers in Table A.2, Page 195)

```
node simeck_box(input:u32x2, rc:u32) returns (output:u32x2)
vars round:u32x2[9]
let
    round[0] = input;
    forall i in [0, 7] {
      round[i+1] = ( f(round[i][0]) ^ round[i][1] ^
                     0xfffffffe ^ ((rc >> i) & 1),
                    round[i][0] ) ;
    }
    output = round[8]
tel
```

This function is also a bottleneck because of data dependencies, but it is actually called 3 times consecutively each round, with independent inputs:

```
node ACE_step(A,B,C,D,E:u32x2,RC,SC:u32[3]) returns (Ar,Br,Cr,Dr,Er:u32x2)
let
    A := simeck_box(A,RC[0]);
    C := simeck_box(C,RC[1]);
    E := simeck_box(E,RC[2]);
    ...
```

After inlining and unrolling `simeck_box`, the effect of our scheduling algorithm will be to interleave those 3 `simeck_box`, thus removing any pipeline stalls from data hazard. Our *m*slice scheduling algorithm brings up the number of instructions executed per cycle (IPC) from 1.93 to 2.67 on AVX2 registers, translating into a $\times 1.35$ speedup.

Similarly, CHACHA20's round relies on the following node:

```
node QuarterRound (a, b, c, d : u32)
     returns (aR, bR, cR, dR : u32)
let
    a := a + b;
    d := (d ^ a) <<< 16;
    c := c + d;
    b := (b ^ c) <<< 12;
    aR = a + b;
    dR = (d ^ aR) <<< 8;
    cR = c + dR;
    bR = (b ^ cR) <<< 7;
tel
```

This function, despite containing only 12 instructions, cannot be executed in fewer than 12 cycles: none of the instructions of this function can be computed in the same cycle, since each of them has a dependency with the previous one. However, this function is called consecutively 4 times on independent inputs:

```
node DoubleRound (state:u32x16) returns (stateR:u32x16)
let
    state[0,4,8,12]  := QR(state[0,4,8,12]);
    state[1,5,9,13]  := QR(state[1,5,9,13]);
    state[2,6,10,14] := QR(state[2,6,10,14]);
    state[3,7,11,15] := QR(state[3,7,11,15]);
```

Our scheduling algorithm is able to interleave each of those calls, allowing them to be executed simultaneously, removing stalls from data dependencies. We observe that, on general-purpose registers, using our scheduling algorithm increases the IPC of CHACHA20 from 2.79 up to 3.44, which translates into a $\times 1.05$ speedup.

For other ciphers likes AES, ASCON and CLYDE, the speedup is lesser but still a significant at $\times 1.04$. AES's `ShiftRows` contains 8 shuffle instructions and nothing else, and thus cannot be executed in fewer than 8 cycles. This algorithm allows this function to be run simultaneously with either the S-box (`SubBytes`) or the `MixColumn` step. Both CLYDE's and ASCON's linear layer are bottlenecked by data dependencies. This algorithm allows those linear layers to be interleaved with the S-boxes, reducing data hazards.

Look-behind window size The look-behind window needs to be at least $n - 1$ on an architecture that can execute n bitwise/arithmetic instructions per cycle. However, experimentally, increasing the size of the look-behind window beyond this number leads to better performance.

To illustrate the impact of the look-behind window size, let us assume that we want to run the C code below on a CPU that can compute two `xor`s each cycle:

```
a1 = x1 ^ x2;
a2 = x3 ^ x4;
b1 = a1 ^ x5;
b2 = a2 ^ x6;
c1 = x7 ^ x8;
c2 = c1 ^ x9;
```

As is, this code would be executed in 4 cycles: during the first cycle, `a1` and `a2` would be computed, and during the second cycle `b1` and `b2` would be computed. However, `c2` depends on `c1` and cannot be computed during the same cycle. Two cycles cycles would thus be necessary to compute the last two instructions.

If we were to reschedule this code using our algorithm with a look-behind window of 1 instructions, the final scheduling would not change (assuming that ties are broken using the initial order of the source, *i.e.,* that when 2 instructions could be scheduled without dependencies with the previous ones, the one that came first in the initial code is selected). Using a look-behind window of 2 instructions, however, would produce the following scheduling:

```
a1 = x1 ^ x2;
a2 = x3 ^ x4;
c1 = x7 ^ x8;
b1 = a1 ^ x5;
b2 = a2 ^ x6;
c2 = c1 ^ x9;
```

This code can now run in 3 cycles rather than 4. This illustrates why using a look-behind window of 2 instructions for SSE/AVX architectures is not always optimal, despite the fact that only 3 bitwise/arithmetic instructions can be executed per cycle. In practice, the complexity of the algorithm depends on the size of the look-behind window, and using arbitrarily large look-behind windows would be prohibitively expensive. Experimentally, there is very little to gain by using more than 10 instructions in the look-behind window, and this size allows the algorithm to still be sufficiently fast.

4.2.6 Interleaving

Scheduling is not always enough to remove all data hazards. For instance, consider RECTANGLE (Figure 1.1, Page 19). Its S-box is bottlenecked by data dependencies, and its linear layer only contains 3 instructions, which can be partially scheduled alongside the S-box, but are not enough to saturate the CPU execution units.

Another optimization to maximize CPU usage (in terms of IPC) consists in interleaving several executions of the program. Usuba allows us to systematize this folklore cryptographer's trick, popularized by Matsui [206]: for a cipher using a small number of registers (for example, strictly below 8 general-purpose registers on Intel), we can increase its ILP by interleaving several copies of a single cipher, each manipulating its own independent set of variables.

This can be understood as a static form of hyper-threading, by which we (statically) interleave the instruction stream of several parallel execution of a single cipher. By increasing ILP, we reduce the impact of data hazards in the deeply pipelined CPU architecture we are targeting. Note that this technique is orthogonal from slicing (which exploits spatial parallelism, in the registers) by exploiting temporal parallelism, *i.e.,* the fact that a modern CPU can dispatch multiple, independent instructions to its parallel execution units. This technique naturally fits within our programming model: we can implement it by a straightforward Usuba0-to-Usuba0 translation.

Example

RECTANGLE's S-box for instance can be written in C using the following macro:

```
#define sbox_circuit(a0,a1,a2,a3) {            \
  int t0, t1, t2;                              \
  t0 = ~a1;                                    \
  t1 = a2 ^ a3;                                \
  t2 = a3 | t0;                                \
  t2 = a0 ^ t2;                                \
  a0 = a0 & t0;                                \
  a0 = a0 ^ t1;                                \
```

```
  t0 = a1 ^ a2;                             \
  a1 = a2 ^ t2;                             \
  a3 = t1 & t2;                             \
  a3 = t0 ^ a3;                             \
  t0 = a0 | t0;                             \
  a2 = t2 ^ t0;                             \
  }
```

This implementation modifies its input (`a0`, `a1`, `a2` and `a3`) in-place, and uses 3 temporary variables `t0`, `t1` and `t2`. It contains 12 instructions, and we might therefore expect it to execute in 3 cycles on a Skylake, saturating the 4 bitwise ports of this CPU. However, it contains a lot of data dependencies: `t2 = a3 | t0` needs to wait for `t0 = ~a1` to be computed; `t2 = a0 ^ t2` needs to wait for `t2 = a3 | t0`; `a0 = a0 ^ t1` needs to wait for `a0 = a0 & t0`, and so on. Compiled with Clang 7.0.0, this code executes in 5.24 cycles on average. By interleaving two instances of this code, the impact of data hazards is reduced:

```
#define sbox_circuit_interleaved(a0,a1,a2,a3,a0_2,a1_2,a2_2,a3_2) {   \
    int t0, t1, t2;         int t0_2, t1_2, t2_2;            \
    t0 = ~a1;               t0_2 = ~a1_2;                    \
    t1 = a2 ^ a3;           t1_2 = a2_2 ^ a3_2;              \
    t2 = a3 | t0;           t2_2 = a3_2 | t0_2;              \
    t2 = a0 ^ t2;           t2_2 = a0_2 ^ t2_2;              \
    a0 = a0 & t0;           a0_2 = a0_2 & t0_2;              \
    a0 = a0 ^ t1;           a0_2 = a0_2 ^ t1_2;              \
    t0 = a1 ^ a2;           t0_2 = a1_2 ^ a2_2;              \
    a1 = a2 ^ t2;           a1_2 = a2_2 ^ t2_2;              \
    a3 = t1 & t2;           a3_2 = t1_2 & t2_2;              \
    a3 = t0 ^ a3;           a3_2 = t0_2 ^ a3_2;              \
    t0 = a0 | t0;           t0_2 = a0_2 | t0_2;              \
    a2 = t2 ^ t0;           a2_2 = t2_2 ^ t0_2;              \
  }
```

This code contains twice the S-box code (visually separated to make it clearer): one instance computes the S-box on `a0`, `a1`, `a2` and `a3`, while the other computes it on a second input, `a0_2`, `a1_2`, `a2_2` and `a3_2`. This code runs in 7 cycles; or 3.5 cycles per S-box, which is much closer to the ideal 3 cycles/S-box. We say that this code is 2-interleaved.

Despite interleaving, some data dependencies remain, and it may be tempting to interleave a third execution of the S-box. However, since each S-box requires 7 registers (4 for the input, and 3 temporaries), this would require 21 registers, and only 16 general-purpose registers are available on Intel. Still, we benchmarked this 3-interleaved S-box, and it executes 12.92 cycles, or 4.3 cycles/S-box; which is slower than the 2-interleaved one, but still faster than without interleaving at all. Inspecting the assembly code reveals that in the 3-interleaved S-box, 4 values are spilled, thus requiring 8 additional `move` operations (4 stores and 4 loads). The 2-interleaved S-box does not contain any spilling.

Remark. In order to benchmark the S-box, we put it in a loop that we unrolled 10 times (using Clang's unroll pragma). This unrolling is required because of the expected port saturation: if the S-box without interleaving were to use ports 0, 1, 5 and 6 of the CPU for 3 cycles, this would leave no free port to perform the jump at the end of the loop. In practice, since the non-interleaved S-box does not saturate the ports, this does not impact execution time. However, unrolling improves performance of the 2-interleaved S-box—which puts more pressure on the execution ports—by 14%.

Cipher	Instruction per cycle		Speedup
	without interleaving	with 2-interleaving	
XOODOO	3.45	3.47	0.81
PYJAMASK	3.45	3.26	0.93
GIMLI	3.33	3.11	0.83
GIFT	3.31	3.36	0.96
CLYDE	2.96	3.67	1.24
CHACHA20	2.85	2.92	1.00
RECTANGLE	2.79	3.49	1.19
ASCON	2.73	3.33	1.22
SERPENT	2.32	3.39	1.40
ACE	2.20	2.75	1.22

Table 4.1: Impact of interleaving on some ciphers

Initial Benchmark

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/interleaving`

To benchmark our interleaving optimization, we considered 10 ciphers with low register pressure, making them good candidates for interleaving. We started by generating non-interleaved and 2-interleaved implementations on general-purpose registers for Intel. We compiled the generated code with Clang 7.0.0. The results are reported in Table 4.1 (sorted by IPC without interleaving). Interleaving is more beneficial on ciphers with low IPC. In all cases, the reason for the low IPC is data hazards. The 2-interleaved implementations reduce the impact of those data hazards, and bring the IPC up, improving throughput. We will examine each case one by one in the next section. One exception is CHACHA20, which, despite its low IPC, does not benefit from interleaving. CHACHA20, as shown in Section 4.2.5 (Page 107), is bottlenecked by data dependencies, and two interleaved instances are not enough to alleviate this bottleneck.

Factor and Granularity

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/interleaving-params`

We showed that interleaving can increase throughput of ciphers suffering from tight dependencies. Our interleaving algorithm is parameterized by a factor and a granularity. The factor determines the number of implementations to be interleaved, while the granularity describes how the implementations are interleaved: 1 instruction from an implementation followed by 1 from another one would be a granularity of 1, while 5 instructions from an implementation followed by 5 from another one would be a granularity of 5. For instance, the RECTANGLE 2-interleaved S-box above has a factor of 2 and a granularity of 1 since instructions from the first and the second S-box are interleaved one by one. A granularity of 4 would correspond to the following code:

```
    t0 = ~a1;                                                 \
    t1 = a2 ^ a3;                                             \
    t2 = a3 | t0;                                             \
    t2 = a0 ^ t2;                                             \
                        t0_2 = ~a1_2;                         \
                        t1_2 = a2_2 ^ a3_2;                   \
                        t2_2 = a3_2 | t0_2;                   \
                        t2_2 = a0_2 ^ t2_2;                   \
    a0 = a0 & t0;                                             \
    a0 = a0 ^ t1;                                             \
    t0 = a1 ^ a2;                                             \
    a1 = a2 ^ t2;                                             \
                        a0_2 = a0_2 & t0_2;                   \
                        a0_2 = a0_2 ^ t1_2;                   \
                        t0_2 = a1_2 ^ a2_2;                   \
                        a1_2 = a2_2 ^ t2_2;
```

The user can use the flags `-interleaving-factor <n>` and `-interleaving-granularity <n>` to instruct Usubac to generate a code using a given interleaving factor and granularity.

The granularity is only up to a function call or loop, since we do not duplicate function calls but rather the arguments in a function call. The rationale being that interleaving is supposed to reduce pipeline stalls within functions (resp., loops), and duplicating function calls (resp., loops) would fail to achieve this. For instance, the following Usuba code (extracted from our ChaCha20 implementation):

```
state[0,4,8,12]  := QR(state[0,4,8,12]);
state[1,5,9,13]  := QR(state[1,5,9,13]);
state[2,6,10,14] := QR(state[2,6,10,14]);
```

is 2-interleaved as (with `QR`'s definition being modified accordingly):

```
(state[0,4,8,12],state_2[0,4,8,12])   :=
                QR_x2(state[0,4,8,12],state_2[0,4,8,12]);
(state[1,5,9,13],state_2[1,5,9,13])   :=
                QR_x2(state[1,5,9,13],state_2[1,5,9,13]);
(state[2,6,10,14],state_2[2,6,10,14]) :=
                QR_x2(state[2,6,10,14],state_2[2,6,10,14]);
```

rather than:

```
state[0,4,8,12]  := QR(state[0,4,8,12]);
state_2[0,4,8,12]  := QR(state_2[0,4,8,12]);
state[1,5,9,13]  := QR(state[1,5,9,13]);
state_2[0,4,8,12]  := QR(state_2[0,4,8,12]);
state[2,6,10,14] := QR(state[2,6,10,14]);
state_2[0,4,8,12]  := QR(state_2[0,4,8,12]);
```

Note that interleaving is done after inlining and unrolling in the pipeline, which means that any node call still present in the Usuba0 code will not be inlined.

To evaluate how factor and granularity impact throughput, we generated implementations of 10 ciphers with different factors (0 (without interleaving), 2, 3, 4 and 5), and different granularities (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 30, 50, 100, 200). We used Usubac's `-unroll` and `-inline-all` flags to fully unroll and inline the code in order to eliminate the impact of loops and function calls from our experiment. Overall, we generated 19.5 millions of lines of C code in 700 files for this benchmark.

We benchmarked our interleaved implementations on general-purpose registers rather than SIMD, and verified that the frontend was not a bottleneck. Since most programs (*e.g.,* HPC applications) on SIMD are made of loops and function calls, decoded instructions are stored in the μop cache (DSB). However, since we fully inlined and unrolled the code, the legacy pipeline (MITE) is used to decode the instructions, and can only process up to 16 bytes of instruction per cycle. SIMD assembly instructions are larger than general-purpose ones: an AVX instruction can easily take 7 bytes (*e.g.,* an addition where one of the operand is a memory address), and 16 bytes often correspond to only 2 instructions; not enough to fill the pipeline. By using general-purpose registers rather than SIMD registers, we remove the risk of underutilizing the DSB in our benchmark.

We report the results in the form of graphs showing the throughput (in cycles per bytes; lower is better) of each cipher depending on the interleaving granularity for each interleaving factor.

Remark. When benchmarking interleaving on general-purpose registers, we were careful to disable Clang's auto-vectorization (using `-fno-slp-vectorize -fno-vectorize`). Failing to do so produces inconsistent results since Clang erratically and partially vectorizes some implementations and not others. Furthermore, we would not be benchmarking general-purpose registers anymore since the vectorized code would use SSE or AVX registers.

RECTANGLE (Figure 4.7a)

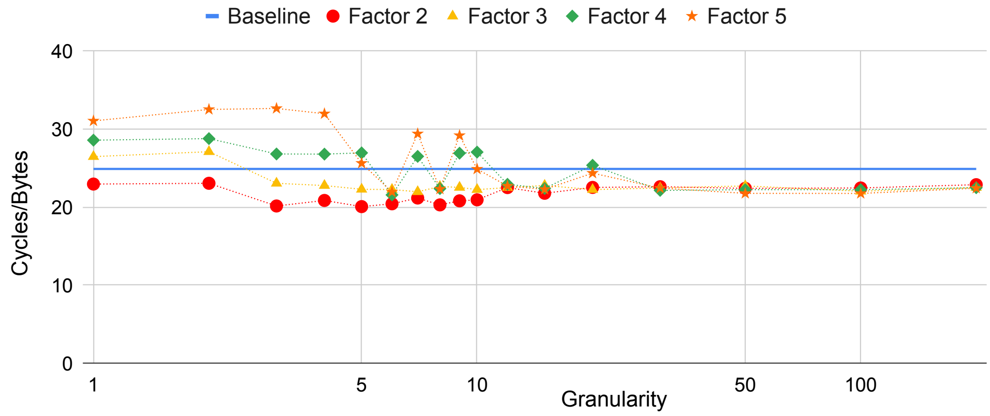
RECTANGLE's S-box contains 12 instructions, and could be executed in 4 cycles if it were not slowed down by data dependencies: its run time is actually 5.24 cycles on average. Interleaving 2 (resp., 3) times RECTANGLE yields a speedup of $\times$1.26 (resp., $\times$1.13), regardless of the granularity. The number of loads and stores in the 2-interleaved RECTANGLE implementation is exactly twice the number of loads and stores in the non-interleaved implementation: interleaving introduced no spilling at all. On the other hand, the 3-interleaved implementation suffers from spilling and thus contains 4 times more memory operations than the non-interleaved one, which results in a slower throughput than the 2-interleaved version.

Interleaving 4 or 5 instances with a small granularity (less than 10) introduces too much spilling, which reduces throughput. Increasing granularity reduces spilling while still allowing the C compiler to schedule the instructions in a way to reduce the impact of data hazards. For instance, 5-interleaved RECTANGLE with a granularity of 50 contains twice fewer memory loads and stores than with a granularity of 2: temporary variables from all 5 implementations tend not to be alive at the same time when the granularity is large enough, while they are when the granularity is small.

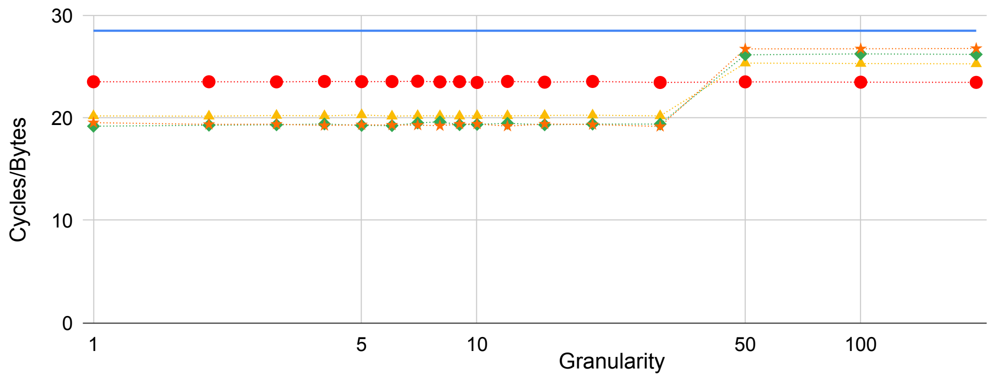
ACE (Figure 4.7b)

One of ACE's basic block is a function that computes `((x <<< 5) & x) ^ (x <<< 1)`. It contains 4 instructions, and yet cannot be executed in fewer than 3 cycles, because of its inner dependencies. Interleaving this function twice means that it contains 8 instructions that cannot execute in fewer than 3 cycles, still wasting CPU resources. Interleaving it 3 times or more allows it to fully saturate its ports (*i.e.,* run its 12 instructions in 3 cycles), and it is indeed faster than the 2-interleaved implementation despite containing more spilling, and is 1.5 times faster than the non-interleaved implementation.

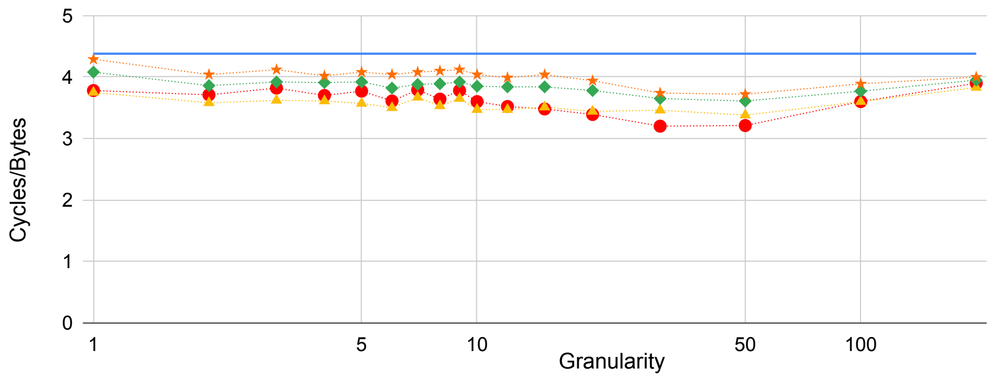
The 3-interleaved code contains already 9 times more memory operations (*i.e.,* reads and stores) than the non-interleaved implementation, as a direct cause of additional spilling. The increase in ILP that the 3-interleaved code brings is, however, more than enough to compensate for this additional spilling. The 5-interleaved code only contains twice more memory operations than the 3-interleaved one, showing that this additional



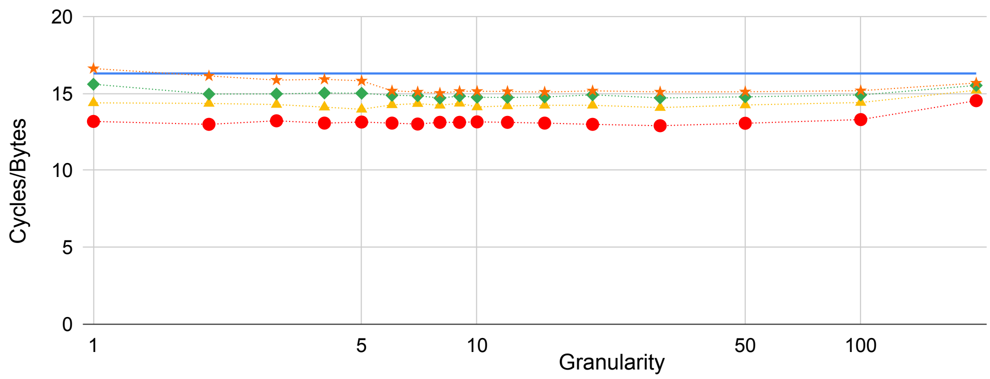
(a) RECTANGLE



(b) ACE



(c) ASCON



(d) CLYDE

Figure 4.7: Impact of interleaving on general-purpose registers

interleaving caused little extra spilling, which explains why it exhibits a similar throughput as the 3-interleaved code.

ACE also contains other idioms that can limit CPU utilization, such as `0xfffffffe ^ ((rc >> i) & 1)`, which cannot execute in fewer than 3 cycles despite containing only 3 instructions. These idioms benefit from interleaving as well.

In all cases, the granularity has a low impact. The reordering done by Clang, as well as the out-of-order nature of the CPU are able to schedule the instructions in a way to reduce the impact of data hazards. Interestingly, when the granularity is larger than 50 instructions, the speed of the 3, 4 and 5-interleaved code falls down, as a result of Clang not being able to properly schedule ACE's instructions.

ASCON (Figure 4.7c)

ASCON's S-box contains 22 instructions with few enough data dependencies to allow it to run in about 6 cycles, which is very close to saturate the CPU. Its linear layer, however, is the following:

```
node LinearLayer(state:u64x5) returns (stateR:u64x5)
let
  stateR[0] = state[0] ^ (state[0] >>> 19) ^ (state[0] >>> 28);
  stateR[1] = state[1] ^ (state[1] >>> 61) ^ (state[1] >>> 39);
  stateR[2] = state[2] ^ (state[2] >>> 1)  ^ (state[2] >>> 6);
  stateR[3] = state[3] ^ (state[3] >>> 10) ^ (state[3] >>> 17);
  stateR[4] = state[4] ^ (state[4] >>> 7)  ^ (state[4] >>> 41);
tel
```

This linear layer is not bottlenecked by data dependencies but on rotations: general-purpose rotations can only be executed on ports 0 and 6. Only two rotations can be executed each cycle, and this code can therefore not be executed in fewer than 7 cycles: 5 cycles for all the rotations, followed by two additional cycles to finish computing `stateR[4]` from the result of the rotations, each computing a single `xor`.

There are two ways that ASCON can benefit from interleaving. First, if the linear layer is executed two (resp., three) times simultaneously, the last 2 extra cycles computing a single `xor` now compute two (resp., three) `xor`s each, thus saving one (resp., two) cycle. Second, out-of-order execution can allow the S-box of one instance of ASCON to execute while the linear layer of the other instance runs, thus allowing a full saturation of the CPU despite the rotations being only able to use ports 0 and 6.

Table 4.2 shows the IPC, number of memory operations (loads/stores), and number of cycles when 1, 2, 3, or 4 instructions are executed on several interleaved version of ASCON (with a granularity of 10). The non-interleaved code has a fairly low IPC of 2.68, and uses fewer than 3 ports on half the cycles. Interleaving twice ASCON increases the IPC to 3.31, and doubles the number of memory operations which indicates that no spilling has been introduced. However, only 72% of the cycles execute 3 μops or more, which still shows an underutilization of the CPU. 3-interleaved ASCON suffers from a lot more spilling: it contains about 10 times more memory operations than without interleaving. However, it also brings the IPC up to 3.80, and uses 3 ports or more on 91% of the cycles. This translates into a 2.5% performance improvement over the 2-interleaved implementation at small granularities.

However, interleaving a fourth or a fifth instance of ASCON does not increase throughput further, which is not surprising since 3-interleaved ASCON already almost saturates the CPU ports.

Factor	Cycles/ Bytes	IPC	#Memory ops (normalized)	% of cycles when at least n μopt are executed			
				1	2	3	4
0	4.89	2.68	1	97.5	90.0	49.2	14.4
x2	3.91	3.31	1.98	98.9	98.9	72.7	36.1
x3	3.77	3.79	10.76	97.5	96.3	88.2	63.3
x4	4.22	3.79	23.23	99.8	97.8	90.6	64.3
x5	4.45	3.74	34.48	99.0	97.1	88.5	64.9

Table 4.2: Impact of interleaving on ASCON

Factor	Cycles/Bytes	IPC	#Memory ops (normalized)
0	16.11	2.95	1
x2	13.23	3.68	2.66
x3	14.07	3.43	4.39
x4	14.59	3.30	5.79
x5	15.07	3.21	6.84

Table 4.3: Impact of interleaving on CLYDE

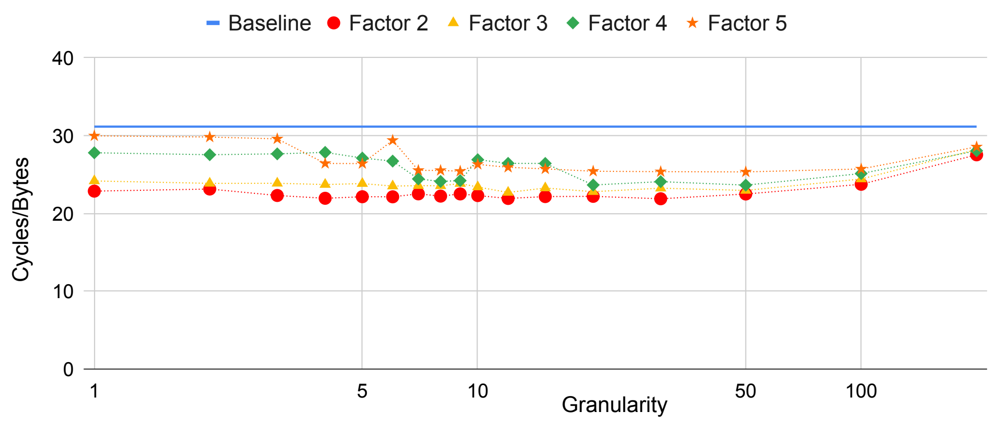
CLYDE (Figure 4.7d)

CLYDE's linear layer consists in two calls to the following Usuba node:

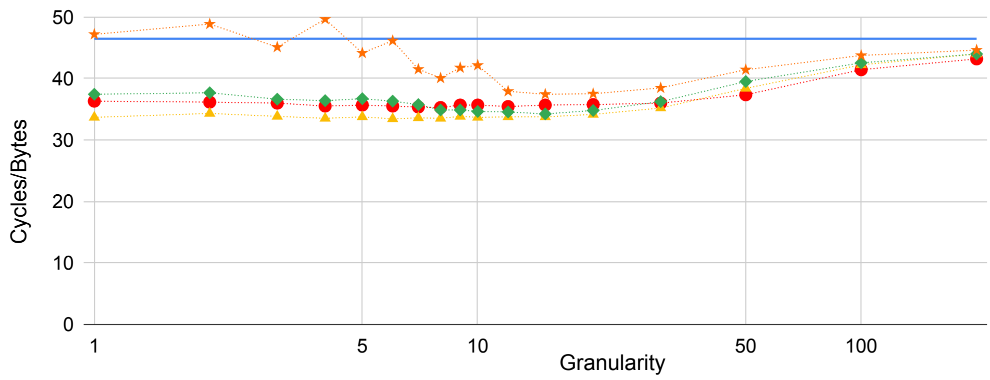
```
node lbox(x,y:u32) returns (xr,yr:u32)
vars
    a, b, c, d: u32
let
    a  = x ^ (x >>> 12);
    b  = y ^ (y >>> 12);
    a := a ^ (a >>> 3);
    b := b ^ (b >>> 3);
    a := a ^ (x >>> 17);
    b := b ^ (y >>> 17);
    c  = a ^ (a >>> 31);
    d  = b ^ (b >>> 31);
    a := a ^ (d >>> 26);
    b := b ^ (c >>> 25);
    a := a ^ (c >>> 15);
    b := b ^ (d >>> 15);
    (xr, yr) = (a,b)
tel
```

We already showed in Section 4.2.5 (Page 107) that this node bottlenecks CLYDE due to its inner data dependencies. Scheduling was able to alleviate the impact of those data hazards by scheduling the two calls to this node simultaneously. Similarly, interleaving two instances of CLYDE removes data hazards, thus improving throughput by 18%.

Table 4.3 the IPC and number of memory operations of CLYDE depending of the interleaving factor. The higher the interleaving factor, the more spilling an implementation contains. Furthermore, 2-interleaving is already enough to bring the IPC up to 3.68. As a result, interleaving beyond a factor of 2 reduces throughput.



(a) General-purpose registers



(b) AVX2

Figure 4.8: Impact of interleaving on Serpent

SERPENT (Figure 4.8a)

SERPENT only uses 5 registers once compiled to assembly (4 for the state and 1 temporary register used in the S-box). Its linear layer contains 28 `xor`s and shifts, yet executes in 14 cycles due to data dependencies. Likewise, the S-boxes were optimized by Osvik [234] to put a very low pressure on the registers, and to exploit a superscalar CPU that is able to execute only two bitwise instructions per cycle (whereas modern Intel CPUs can do 3 or 4 bitwise instructions per cycles). For instance, the first S-box contains 17 instructions, but executes in 8 cycles. This choice made sense back when there were only 8 general-purpose registers available, among which one was reserved for the stack pointer, and one was used to keep a pointer to the key, leaving only 6 registers available.

Now that we have 16 registers available, interleaving several implementations of SERPENT makes sense and does greatly improve throughput as can be seen from Figure 4.8a. Once again, interleaving more than two instances introduces spilling, and does not improve throughput.

On AVX, however, all 16 registers are available for use: the stack pointer and the pointer to the key are kept in general-purpose registers rather than AVX ones. 3-interleaving therefore introduces less spilling than on GP registers, making it 1.05 times faster than 2-interleaving, as shown in Figure 4.8b.

The current best AVX2 implementation of Serpent is—to the best of our knowledge—the one used in the Linux kernel, written by Kivilinna [183], which uses 2-interleaving. Kivilinna mentions that he experimentally came to the conclusion that interleaving at a granularity of 8 is optimal for the S-boxes and that 10 or 11 is optimal for the linear layer.

Using Usuba, we are able to systematize this experimental approach and we observe that, with Clang, granularity has a very low impact on throughput, and a factor of 3 is optimal.

Other Ciphers (Figure 4.9)

On the other ciphers we benchmarked (CHACHA20, GIFT, GIMLI, PYJAMASK and XOODOO), interleaving made throughput worse, from a couple of percents (*e.g.,* 2-interleaving on CHACHA20) up to 50% (*e.g.,* 5-interleaving on XOODOO). As shown in Table 4.1, all of those ciphers have very high IPC, above 3.3. None of those cipher is bottlenecked by data dependencies (except for CHACHA20, which also has a high register pressure), and the main impact of interleaving is to introduce spilling.

Overall Impact of Factor and Granularity

All of the examples we considered show that choosing a coarse granularity (50 instructions or more) reduces the impact of interleaving, regardless of whether it was positive or negative. Most of the time, the granularity has little to no impact on throughput, as long as it is below 20. There are a few exceptions, which can be attributed to either compiler or CPU effects, and can be witnessed on SERPENT (Figure 4.8), RECTANGLE (Figure 4.7a), and ACE (Figure 4.7b). For instance, on RECTANGLE 5-interleaved, granularities of 6 and 8 are 1.45 times better than granularities of 3 and 4 and 1.33 times better than granularities of 7 and 9. When monitoring the programs, we observe that implementations with a granularity of 7 and 9 do 3 times more memory accesses than the ones with a granularity of 6 and 8, thus showing that our optimizations always risk to be nullified by poor choices from the C compiler's heuristics.

Determining the ideal interleaving factor for a given cipher is a complex task because it depends both on register pressure and data dependencies. In some cases, like SERPENT, introducing some spilling in order to reduce the amount of data hazards is worth it, while in other cases, like RECTANGLE, spilling deteriorates throughput. Furthermore, while we can compute the maximum number of live variables in an Usuba0 program, the C compiler is free to schedule the generated C code how it sees fit, potentially reducing or increasing the number of live variables.

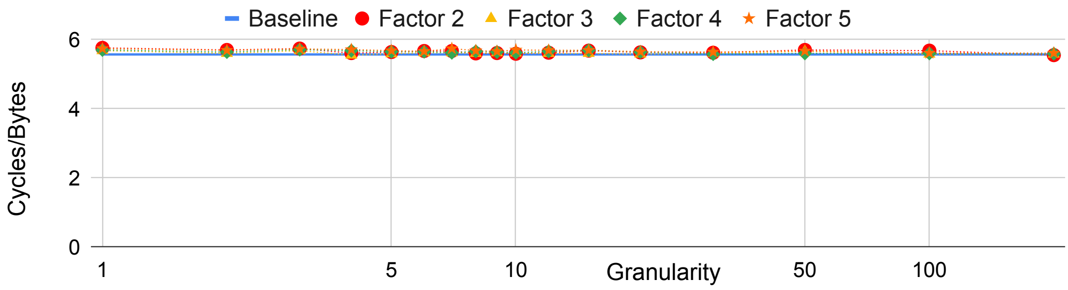
Fortunately, Usubac's autotuner fully automates the interleaving transformation, thus relieving programmers from the burden of determining by hand the optimal interleaving parameters. When compiling with the autotuner enabled, Usubac automatically benchmarks several interleaving factors (0, 2 and 3) in order to find out which is best.

The autotuner does not benchmark the granularity when selecting the ideal interleaving setup, since its impact on throughput is minimal. Furthermore, our *m*slice scheduling algorithm (Section 4.2.5, Page 4.2.5) strives to minimize the impact of data dependencies and thus reduces even further the influence of granularity.

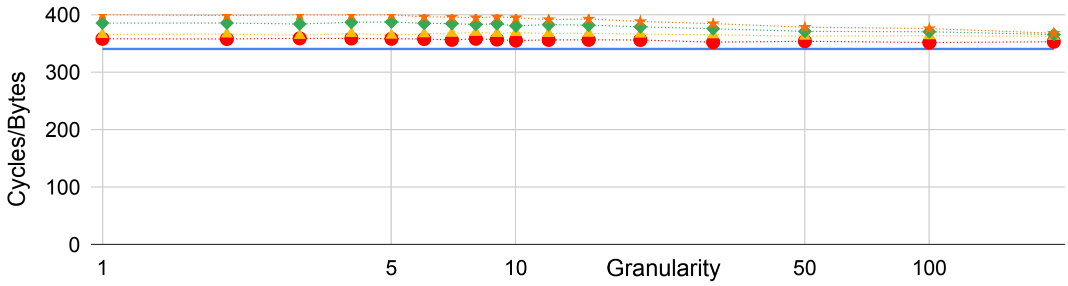
4.2.7 Code Generation

Compiling Usuba0 to C is straightforward, as shown in Section 3.6.3. Nodes are translated to function definitions, and node calls to function calls. All expressions in Usuba0 have a natural equivalent in C, with the exception of Usuba's `shuffle`, `bitmask` and `pack` operators, whose compilation has already been discussed in Section 2.3.

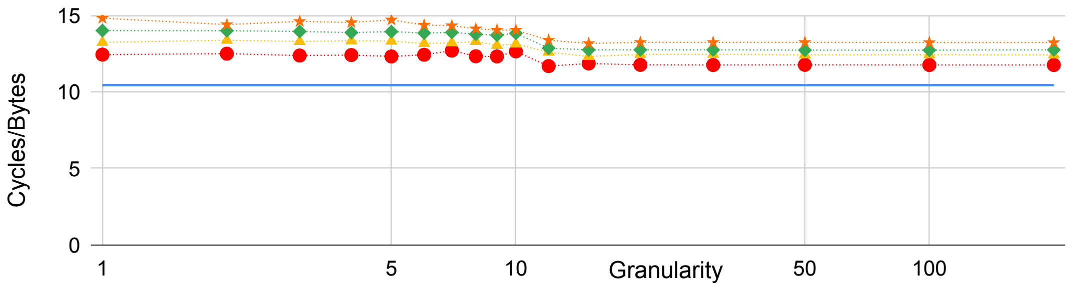
The generated C code relies on macros rather than inline operators. For instance, compiling the following Usuba0 nodes to C:



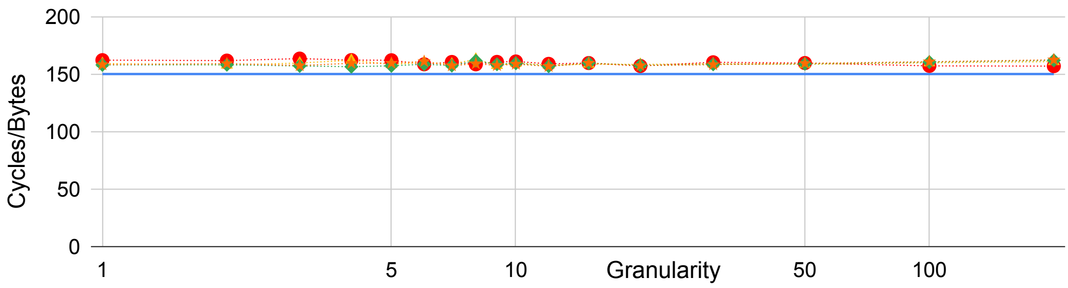
(a) CHACHA20



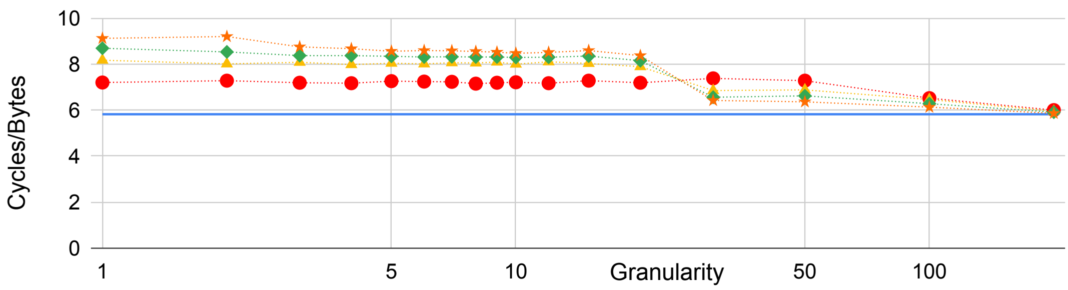
(b) GIFT



(c) GIMLI



(d) PYJAMASK



(e) XOODOO

Figure 4.9: Impact of interleaving on general-purpose registers

```
node sbox (i0:u32, i1:u32)
     returns (r0:u32, r1:u32)
vars t1 : u32
let
    t1 = ~i0;
    r0 = t1 & i1;
    r1 = t1 | i1;
tel
```

produces

```
void sbox__ (/*inputs*/ DATATYPE i0__,DATATYPE i1__,
             /*outputs*/ DATATYPE* r0__,DATATYPE* r1__) {
  // Variables declaration
  DATATYPE t1__;

  // Instructions (body)
  t1__ = NOT(i0__);
  *r0__ = AND(t1__,i1__);
  *r1__ = OR(t1__,i1__);
}
```

Where `DATATYPE` is `unsigned int` on 32-bit registers, `__m128i` on SSE, `__m256i` on AVX2, etc., and `NOT`, `AND` and `OR` are defined to use the architecture's instructions. Using macros allows us to change the architecture of the generated code by simply changing a header. The new header must provide the same instructions, which means that, for instance, a code compiled for AVX2 and using `shuffle`s cannot be run on general-purpose registers by simply changing the header since no shuffle would be available.

Usubac performs no architecture-specific optimizations, beyond its scheduling and interleaving, which target superscalar architectures. Put otherwise, we do not compile any differently a code for SSE or AVX2, except that Usubac's autotuner may select different optimizations for each architecture. For general-purpose registers, SSE, AVX and AVX2, the instructions used for cryptographic primitives are fairly similar, and we felt that there was no need to optimize differently on each architecture. In most cases where architecture-specific instructions can be used to speed up computation, Clang and GCC are able to detect it and perform the optimization for us.

The AVX512 instruction set is, however, much richer, and opens the door for more optimizations. For instance, it offers the `vpternlogd` instruction, which can compute any Boolean function with 3 inputs. It can be useful, in particular, to speed up S-boxes. For instance,

```
t1 = a ^ b;
t2 = t1 & c;
```

can be written with a single `vpternlog` as

```
t2 = _mm512_ternarylogic_epi64(a,b,c,0b00010100);
```

Thus requiring one instruction rather than two. ICC is able to automatically perform this optimization in some cases, but Clang and GCC are not. We leave it to future work to evaluate whether Usubac could produce more optimized code by performing such optimizations.

4.3 Conclusion

We showed in this chapter how Usubac compiles Usuba programs down to C. The frontend first normalizes (and monomorphizes) Usuba down to Usuba0, a monomorphic low-level subset of Usuba that can be straightforwardly compiled to C. The backend applies several optimizations on the Usuba0 representation before generating C code.
We demonstrated that C compilers are unable to properly optimize the sliced cryptographic code generated by Usubac. To overcome this limitation of C compilers, we implemented several domain-specific optimizations ourselves (*e.g.*, scheduling and interleaving) to produce efficient code.

4.3.1 Related Work

Fisher and Dietz [137] and Ren [258] proposed architecture-agnostic languages and compilers to enable high-level generic SIMD programming. By programming within these languages, programmers do not need to worry about the availability of particular SIMD instructions on target architectures: when an instruction is not realized in hardware, the compiler provides "emulation code". Our use-case takes us to the opposite direction: it forbids us from turning to emulation since our interest is precisely to expose as much of the (relevant) details of the target SIMD architecture to the programmer.

Clang, GCC and ICC all have passes to automatically (and heuristically) vectorize loops. They also provide pragma annotations that can be used to manually drive the heuristics to vectorize specific loops. OpenMP [232], initially designed to provide abstractions for thread-oriented parallelization, now (since version 4.0) provides a `#pragma omp simd` directive to instruct the compiler to vectorize a specific loop. The C++-based language Cilk Plus [266] offers a `#pragma simd` directive that behaves similarly.

Several projects have also been developed to give programmers a more explicit control over vectorization. Cebrian et al. [101] for instance developed a set of generic macros to replace SIMD-specific instructions, similar to Usuba's operators. Several C++ libraries, such as Vc [190] or Boost.SIMD [134], provide type abstractions and overloaded operators to manipulate generic vectors, which are then compiled down to SIMD instructions.

ispc [242] ports the concept of Single Program Multiple Data (SPMD)—traditionally describing multithreaded code—to SIMD. The ispc language provides C-like syntax and constructions, and let the programmer write a scalar program which is automatically vectorized in its entirety (as opposed to partially vectorized) by the compiler. Usuba takes a similar stance: to take full advantage of hardware-level parallelism, we must be able to describe the parallelism of our programs. However, our scope is much narrower: we are concerned with high-throughput realization of straight-line, bit-twiddling programs whose functional and observational correctness is of paramount importance. This rules out the use of a variant of C or C++ as a source language.

Chapter 5

Performance Evaluation

Publication

This work was done in collaboration with Pierre-Évariste Dagand (LIP6). It led to the following publication:

D. Mercadier and P. Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 157–173. ACM, 2019. doi: 10.1145/3314221.3314636

Usuba has two targets: high-end CPUs (*e.g.*, Intel), for which speed is the main concern, and low-end embedded devices (*e.g.*, ARM), for side-channel resistance is the main concern. In this chapter, we evaluate the throughput of ciphers written in Usuba on Intel CPUs. First, we demonstrate that our implementations perform on par with reference, hand-tuned implementations (Section 5.1): the high-level features of Usuba do not prevent us from generating efficient C code. We then analyze the scaling of our implementations on Intel SIMD extensions (SSE, AVX, AVX2, AVX512), showing that Usuba is able to fully exploit these extensions (Section 5.2). Finally, we illustrate the throughput differences between bitslicing, *v*slicing and *h*slicing on RECTANGLE, and show that no slicing mode is strictly better than the others, thus illustrating the benefits of Usuba's polymorphism, as it allows to easily switch between slicing modes (Section 5.3).

5.1 Usuba *vs.* Reference Implementations

! The benchmarks of this section are available at:
`https://github.com/DadaIsCrazy/usuba/tree/master/bench/ua-vs-human`

In the following, we benchmark our Usuba implementations against hand-written C or assembly implementations. When possible, we benchmark only the cryptographic primitive, ignoring key schedule, transposition, and mode of operation. To do so, we selected open-source, reference implementations that were self-identified as optimized for high-end CPUs. The ciphers we considered are DES, AES (Figure 2.5, Page 61), CHACHA20 (Figure 2.10, Page 66), SERPENT (Figure 2.11, Page 67), RECTANGLE (Figure 1.1, Page 19), GIMLI, ASCON (Figure 2.7, Page 63), ACE (Figure 2.9, Page 65) and CLYDE. We also evaluate the throughput of Usuba on PYJAMASK, XOODOO and GIFT, although we did not find reference implementations optimized for high-end CPUs for these ciphers.

Several reference implementations (AES, CHACHA20, SERPENT) are written in assembly, without a clear separation between the primitive and the mode of operations, and only provide an API to encrypt bytes in CTR mode. To compare ourselves with those implementations, we implemented the same mode of operation in C, following their code as closely as possible. This puts us at a slight disadvantage, because assembly implementations tend to fuse the cryptographic runtime (*i.e.,* mode and transposition) into the primitive, thus enabling further optimizations. We used the Supercop [65] framework to benchmark these implementations (since the references were already designed to interface with it), and the performance we report below includes key schedule, transposition (when a transposition is needed) and management of the counter (following the CTR mode of operation).

For implementations that were not designed to interface with Supercop (DES, RECTANGLE, GIMLI, ASCON, ACE, CLYDE), we wrote our own benchmarking code as described in Section 1.2.1 (Page 27). For these ciphers, the cost of transposing data is omitted from our results, since transposition is done outside of Usuba. Transposition costs vary depending on the cipher, the slicing mode and the architecture: transposing bitsliced DES's inputs and outputs costs about 3 cycles per bytes (on general-purpose registers); while transposing *v*sliced SERPENT's inputs and outputs costs about 0.38 cycles/bytes on SSE and AVX, and 0.19 cycles/bytes on AVX2.

We have conducted our benchmarks on a Intel Core i5-6500 CPU @ 3.20GHz machine running Ubuntu 16.04.5 LTS (Xenial Xerus) with Clang 7.0.0, GCC 8.4.0 and ICC 19.1.0. Our experiments showed that no C compiler is strictly better than the others. ICC is for instance better to compile our CHACHA20 implementation on AVX2, while Clang is better to compile our SERPENT implementation on AVX. We thus compiled both the reference implementations and ours with Clang, ICC and GCC, and selected the best-performing ones.

We report in Table 5.1 the benchmarks of our Usuba implementations of the 9 aforementioned ciphers against the most efficient publicly available implementations. Reference implementations for the ciphers submitted to the NIST lightweight ciphers competition (GIMLI, ASCON, ACE, CLYDE) are taken from their NIST submission. For other ciphers (DES, AES, CHACHA20, SERPENT, RECTANGLE), we provide in Table 5.1 the sources of the reference implementations. In all cases we instructed Usubac to generate code using the same SIMD extensions as the reference. We also provide the SLOC (source lines of code) count of the cipher primitive (*i.e.,* excluding key schedule and counter management) for every implementation. Usuba programs are almost always shorter than the reference ones, as well as more portable: for each cipher, a single Usuba code is used to generate every specialized SIMD code. In the following, we comment on the performance of each cipher.

DES is not secure and should not be used in practice. However, the simplicity and longevity of DES makes for an interesting benchmark: bitslicing was born out of the desire to provide fast software implementations of DES. There exists several publicly-available, optimized C implementations in bitsliced mode for 64-bit general-purpose registers. The execution time of the reference implementation, and even more, of Usuba's implementation are reliant on the C compiler's ability to find an efficient register allocation: Clang produces a code that is 11.5% faster than GCC for the reference implementation, and 12% faster than ICC for the Usuba implementation.

Slicing mode	Cipher	Instr. set	Code size (SLOC)		Throughput (cycles/byte)		Speedup
			Ref.	Usuba	Ref.	Usuba	
bitslicing	DES [191, 192]	x86-64	1053	655	11.31	10.63	+6.01%
16-*h*slicing	AES [174, 175]	SSSE3	272	218	5.73	5.93	-3.49%
16-*h*slicing	AES [183, 181]	AVX	339	218	5.53	5.81	-5.06%
32-*v*slicing	CHACHA20 [129]	AVX2	20	24	1.00	0.99	+1%
32-*v*slicing	CHACHA20 [219]	AVX	134	24	2.00	1.98	+1%
32-*v*slicing	CHACHA20 [219]	SSSE3	134	24	2.05	2.08	-1.46%
32-*v*slicing	CHACHA20 [61]	x86-64	26	24	5.58	5.20	+6.81%
32-*v*slicing	SERPENT [156]	AVX2	300	214	4.17	4.15	+0.48%
32-*v*slicing	SERPENT [155]	AVX	300	214	8.15	8.12	+0.37%
32-*v*slicing	SERPENT [182]	SSE2	300	214	8.88	9.22	-3.83%
32-*v*slicing	SERPENT [235]	x86-64	300	214	30.95	22.37	+27.72%
16-*v*slicing	RECTANGLE [309]	AVX2	115	31	2.28	1.79	+21.49%
16-*v*slicing	RECTANGLE [309]	AVX	115	31	4.55	3.58	+21.32%
16-*v*slicing	RECTANGLE [309]	SSE4.2	115	31	5.80	3.71	+36.03%
16-*v*slicing	RECTANGLE [309]	x86-64	115	31	26.49	21.77	+17.82%
32-*v*slicing	GIMLI	SSE4.2	70	52	3.68	3.11	+15.49%
32-*v*slicing	GIMLI	AVX2	117	52	1.50	1.57	-4.67%
64-*v*slicing	ASCON	x86-64	68	55	4.96	3.73	+24.80%
32-*v*slicing	ACE	SSE4.2	110	43	18.06	10.35	+42.69%
32-*v*slicing	ACE	AVX2	132	43	9.24	4.55	+50.86%
64-*v*slicing	CLYDE	x86-64	75	71	15.22	10.79	+29.11%

Table 5.1: Comparison between Usuba code & optimized reference implementations

AES fastest hand-tuned SSSE3 and AVX implementations are *h*sliced. The two reference implementations are given in hand-tuned assembly. On AVX, one can take advantage of 3-operand, non-destructive instructions, which significantly reduces register pressure. Thanks to Usubac, we have compiled our (single) implementation of AES to both targets, our AVX implementation taking full advantage of the extended instruction set thanks to the C compiler. Our generated code lags behind hand-tuned implementations for two reasons. First, the latter fuses the implementation of the counter-mode (CTR) run-time into the cipher itself. In particular, they locally violate x86 calling conventions so that they can return multiple values within registers instead of returning a pointer to a structure. Second, the register pressure is high because the S-box requires more temporaries than there are available registers. While hand-written implementations were written in a way to minimize spilling, we rely on the C compiler to allocate registers, and in the case of AES, it is unable to find the optimal allocation.

CHACHA20 has been designed to be efficiently implementable in software, with an eye toward SIMD. The fastest implementations to date are *v*sliced, although some very efficient implementations use a hybrid mix of *v*slicing and *h*slicing (discussed in Section 8.1.3, Page 165). As shown in Section 4.2.5 (Page 100), CHACHA20 contains four calls to a quarter round (`QR`) function. This function is bottlenecked by data dependencies, and the key to efficiently implement CHACHA20 is to interleave the execution of these four functions in order to remove any data hazards. Usubac's scheduler is able to do so automatically, thus allowing us to perform on-par with hand-tuned implementations (whose authors manually implemented this optimization).

SERPENT was designed with bitslicing in mind. It can be implemented with less than 8 64-bit registers. Its fastest implementation is written in assembly, exploiting the AVX2 instruction set. The reference AVX2, AVX and SSE implementations manually interleave 2 parallel instances of the cipher. Usubac's auto-tuner, however, is able to detect that on AVX2 and AVX, interleaving 3 implementations is slightly more efficient, thus yielding similar throughput as the reference implementations (despite generating C code). When interleaving only 2 implementations, Usubac is a couple of percents slower than hand-tuned implementations. On general-purpose registers, the reference implementation is not interleaved, while Usuba's implementation is, hence the 27% speedup.

RECTANGLE is a lightweight cipher, designed for relatively fast execution on micro-controllers. It only consumes a handful of registers. We found no high-throughput implementation online. However, the authors were kind enough to send us their SIMD-optimized implementations. These implementations are manually *v*sliced in C++ but fail to take advantage of interleaving: as a result, our straightforward Usuba implementation easily outperforms the reference one.

GIMLI is a lightweight permutation submitted to the NIST lightweight ciphers competition (LWC). The reference implementations rely on a hybrid *m*slicing technique, mixing aspects of *v*slicing and *h*slicing (discussed in Section 8.1.3, Page 165). This hybrid *m*slicing requires fewer registers than pure *v*slicing, at the expense of additional shuffles. The AVX2 implementation takes advantage of the reduced register pressure to enable interleaving, which would not be efficient in purely *v*sliced implementation. However, the authors chose not to interleave their SSE implementations, allowing Usuba to be 15% faster. Note that another benefit of the hybrid *m*slicing used in the reference implementations is that they require less independent inputs to be efficient. The reference implementation thus has a lower latency if less than 4 (on SSE) or 8 (on AVX2) blocks need to be encrypted.

ASCON is another candidate to the LWC. The authors provided an optimized implementation, written in a low-level C: loops have been manually unrolled and functions manually inlined. In addition to unrolling and inlining nodes, Usubac interleaves ASCON twice, thus resulting in a 25% speedup over this reference implementation. When disabling interleaving and scheduling, the Usuba-generated implementation has indeed similar performance as the reference one.

ACE provides two vectorized implementations in its LWC submission: one for SSE and one for AVX. As shown in Section 4.2.5 (Page 100), ACE's `simeck_box` function is bottlenecked by its data dependencies. By fully inlining and unrolling the code, Usubac is able to better schedule the code, thus removing any data hazards, which leads to a 42% speedup on SSE and a 50% speedup on AVX2. Alternatively, if code size matters, the developer can ask Usubac to interleave ACE twice (using the `-interleaving-factor 2` flag), thus removing the need for unrolling and inlining, which produces a smaller code, while remaining more than 30% faster than the reference implementation.

CLYDE is a primitive used in the Spook submission to the LWC. A fast implementation for x86 CPUs is provided by the authors. However, because of data hazards in its linear layer, its IPC is only 2.87, while Usuba's 3-interleaved optimization reaches an IPC of 3.59, which translates into a 29% speedup.

Slicing mode	Cipher	Code size (SLOC)		Throughput (cycles/byte)		Speedup
		Ref.	Usuba	Ref.	Usuba	
32-*v*slicing	PYJAMASK	60	40	268.94	136.70	+49.17%
32-*v*slicing	XOODOO	51	53	6.30	5.77	+8.41%
32-*v*slicing	GIFT	52	65	523.90	327.13	+37.56%

Table 5.2: Comparison between Usuba code & unoptimized reference implementations

We also compared the Usuba implementations on general-purpose registers of 3 candidates of the LWC that only provided naive implementations for x86. These implementations were chosen because they were among the benchmarks of Tornado (Chapter 6), and their reference implementations are sliced (unlike, for instance, PHOTON and SPONGENT: both of them rely on lookup tables), and therefore comparable with Usuba's implementations. In all 3 cases, Usubac-generated implementations are faster than the reference. While we do not pride ourselves in beating unoptimized implementations, this still hints that Usuba could be used by cryptographers to provide faster reference implementations with minimal effort. Our results are presented in Table 5.2.

PYJAMASK is slow when unmasked because of its expensive linear layer. After unrolling and inlining several nodes, Usubac's *m*sliced scheduler is able to interleave several parts of this linear layer, which were bottlenecked by data dependencies when isolated. The IPC of the Usuba-generated implementation is thus 3.92, while the reference one is only at 3.23.

XOODOO leaves little room for optimization: its register pressure is low enough to prevent spilling, but too high to allow interleaving. Some loops are bottlenecked by data dependencies, but C compilers automatically unroll them, thus alleviating the issue. As a result, Usuba is only 8% faster than the reference implementation. Furthermore, manually removing unnecessary copies from the reference implementation makes it perform on-par with Usuba. Still, Usuba can automatically generate SIMD code, which would easily outperform this x86-64 reference implementation.

GIFT suffers from an expensive linear layer, similarly to PYJAMASK. By inlining it, Usubac is able to perform some computations at compile time, and once again, improves the IPC from 3.37 (reference implementation) to 3.93. Note that a recent technique called *fixslicing* [8] allows for a much more efficient implementation of GIFT: Usuba's fixsliced implementation performs at 38.5 cycles/bytes. However, no reference fixsliced implementation of GIFT is available on x86 to compare ourselves with: the only fixsliced implementation available is written in ARM assembly. Also, note that despite its name, there is no relationship between bitslicing/*m*slicing and fixslicing. For instance, our fixsliced implementation of GIFT is in fact *v*sliced.

5.2 Scalability

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/scaling-avx512`

The ideal speedup along SIMD extensions grows linearly with the size of registers. In practice, however, spilling wider registers puts more pressure on the L1 data-cache, leading to more frequent misses. Also, AVX and AVX512 registers need tens of thousands warm-up cycles before being used, since they are not powered when no instruction uses them (this is not taken into account in our benchmarks because we include a warm-up phase). SSE instructions take two operands and overwrite one to store the result, while AVX offer 3-operand non-destructive instructions, thus reducing register spilling. 32 AVX512 registers are available against only 16 SSE/AVX ones, thus reducing the need for spilling. The latency and throughput of most instructions differ from one microarchitecture to another and from one SIMD to another. For instance, up to 4 general purpose bitwise operations can be computed per cycle, but only 3 SSE/AVX, and 2 AVX512. Finally, newer SIMD extensions tend to have more powerful instructions than older ones. For instance, AVX512 offers, among many useful features, 32-bit and 64-bit rotations (`vprold`).

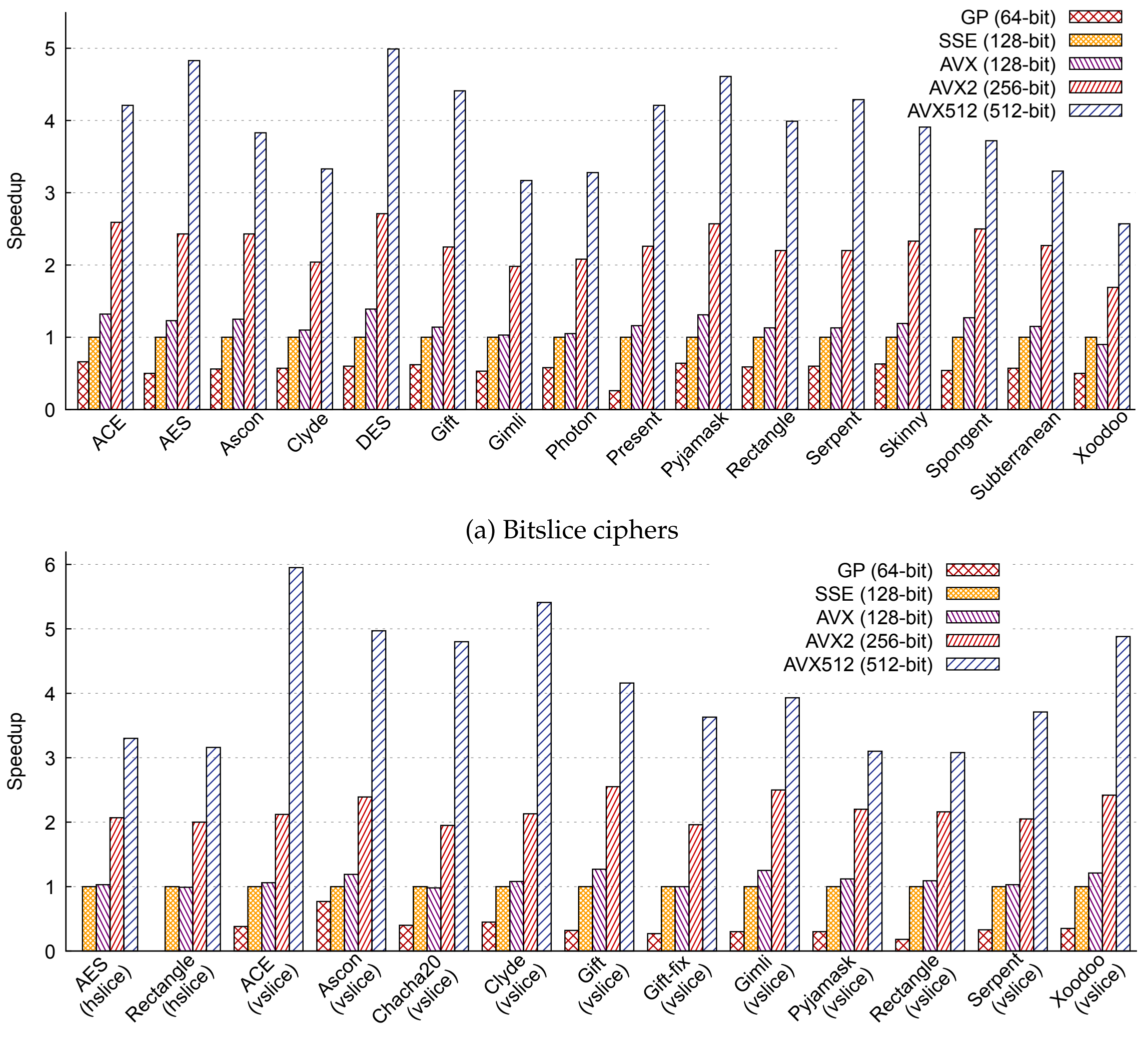
In the following, we thus analyze the scaling of our implementations on the main SIMD extensions available on Intel: SSE (SSE4.2), AVX, AVX2 and AVX512. These benchmarks were compiled using Clang 9.0.1, and executed on a Intel Xeon W-2155 CPU @ 3.30GHz running a Linux 5.4.0. We distinguish AVX from AVX2 because the latter introduced shifts, n-bit integer arithmetic, and byte-shuffle on 256 bits, thus making it more suitable for slicing on 256 bits. Our results are shown in Figure 5.1.

We omitted the cost of transposition in this benchmark to focus solely on the cryptographic primitives. The cost of transposition depends on the data layout and the target instruction set. For example, the transposition of $\mathtt{u}_V 16 \times 4$ can cost as little as 0.09 cycles/byte on AVX512 while the transposition of $\mathtt{u}_H 16 \times 4$ costs up to 2.36 cycles/byte on SSE.

Using AVX instructions instead of SSE (still filled with 128 bits of data though) increases performance from 1% (*e.g.*, RECTANGLE *v*sliced) up to 31% (*e.g.*, ACE bitslice). AVX can offer better performance than SSE mainly because they provide 3-operand non-destructive instructions, whereas SSE only provides 2-operand instructions that overwrite one of their operands to store the results. Using AVX instructions reduces register pressure, which is especially beneficial for bitsliced implementations: DES, for instance, contains 3 times less memory operations on AVX than on SSE. Some *v*sliced ciphers are also faster thanks to AVX instructions, which is once again due to the reduced register pressure, in particular when interleaving—which puts a lot of pressure on the registers—is involved. *v*sliced ASCON, GIMLI and XOODOO contains respectively 6, 2 and 2.5 times less memory operations on AVX than on SSE.

We observe across all ciphers and slicing types that using AVX2 rather than AVX registers doubles performance.

Doubling the size of the register once more by using AVX512 has a very different impact depending on the ciphers, and is more complex to analyze. First, while 3 arithmetic or bitwise AVX or SSE instructions can be executed each cycle, it is limited to 2 on AVX512. Indeed, our *m*sliced AVX2 implementations have IPC above 3, while their AVX512 counterparts are closer to 2. This means that if Clang chooses to use equivalent instructions for AVX2 and AVX512 instructions (*e.g.*, `_mm256_xor_si256` to perform a `xor` on AVX2, and



(b) *m*slice ciphers

Figure 5.1: Scaling of Usuba's implementations on Intel SIMD

`_mm512_xor_si512` on AVX512), the throughput on AVX512 should only be 1.33 times greater than on AVX2 ($\times 2$ because the registers are twice larger, but $\times 2/3$ because only 2 instructions can be executed each cycle instead of 3). However, only 1 shuffle can be executed each cycle regardless of the SIMD instruction set (SSE, AVX2, AVX512), which mitigates this effect on *h*sliced ciphers. Thus, *h*sliced RECTANGLE is 1.6 times faster on AVX512 than on AVX2: one eighth of its instructions are shuffles.

Besides providing registers twice as large as AVX2, AVX512 also offer twice more registers (32 instead of 16), thus reducing spilling. These 16 additional registers reduce spilling in AVX512 implementations compared to AVX2. Bitslice DES, GIFT, RECTANGLE and AES thus contain respectively 18%, 20%, 32% and 55% less spilling on AVX512 than on AVX2, and are 4 to 5 times faster than their SSE counterparts.

Similarly, *h*sliced AES suffers from some spilling on AVX2, but none on AVX512, which translates into 13% fewer instructions on AVX512. Furthermore, one eighth of its instructions are shuffles (which have the same throughput on AVX2 and AVX512). This translates into a $\times 1.59$ speedup on AVX512 compared to AVX2.

Using AVX512 rather than AVX2, however, tends to increase the rate of cache-misses. For instance, on GIMLI (resp., XOODOO and SUBTERRANEAN), 14% (resp., 18% and 19%) of memory loads are misses on AVX512, against less than 2% (resp., less than 3% and

1%) on AVX2. We thus observe that bitslice implementations of ciphers with large states, such as GIMLI (384 bits), SUBTERRANEAN (257 bits), XOODOO (384 bits), PHOTON (256 bits), generally scale worse than ciphers with small states, like DES (64 bits), AES (128 bits), PYJAMASK (128 bits), RECTANGLE (64 bits) and SERPENT (128 bits). An exception to this rule is CLYDE, whose state is only 128 bits, but is only 1.63 faster on AVX512 than on AVX2: 21% of its memory accesses are misses on AVX512, against only 2% on AVX2. Another exception is ACE, whose state is 320 bits, and is 1.62 times faster on AVX512: only 4% of its memory accesses are misses on AVX512. Those effects are hard to predict since they depend both on the cipher, the compiler, and the hardware prefetcher.

The *m*sliced ciphers that exhibit the best scaling on AVX512 (CHACHA20, ASCON, CLYDE, ACE, XOODOO) all heavily benefit from the AVX512 rotate instructions. On older SIMD instruction sets (*e.g.*, AVX2/SSE), rotations had to be emulated using 3 instructions. For instance, left-rotating a 32-bit integer `x` by n places can be done using `(x << n) | (x >> (32-n))`.
Rotations amount to more than a third of CLYDE's and CHACHA20's instructions, and a fourth of ACE, ASCON, XOODOO's instructions. For those 5 ciphers, AVX512 thus considerably reduce the number of instructions compared to AVX2 and SSE, which translates into speedups of $\times 4$ to $\times 6$ compared to SSE.

Let us focus on SERPENT to provide a detailed explanation of scaling from AVX2 to AVX512. About 1 in 6 instructions of *v*sliced SERPENT are rotations. SERPENT contains only 13 spill-related moves on AVX2, which, while absent from the AVX512 implementation, have little impact on performance. SERPENT contains 2037 bitwise and shift instructions, and 372 rotations. On AVX512, this corresponds to $2037 + 372 = 2409$ instructions. On AVX2, rotations are emulated with 3 instructions, which causes the total of instructions to rise to $2037 + (372 \times 3) = 3153$. Since two AVX512 or three AVX2 instructions are executed each cycle, $2409/2 = 1205$ cycles are required to compute the AVX512 version, and only $3153/3 = 1051$ cycles for the AVX2 version. Since the AVX512 implementation computes twice many instances in parallel than on AVX2, the speedup of the AVX512 should thus be $1051/1205 \times 2 = 1.74$. In practice, we observe a speedup of $\times 1.80$. The few additional percents of speedup are partially due to a small approximation in the explanations above: we overlooked the fact that the round keys of SERPENT are stored in memory, and that as many loads per cycles can be performed on AVX512 and AVX2.

5.3 Polymorphic Cryptographic Implementations

> **!** The benchmarks of this section are available at:
> `https://github.com/DadaIsCrazy/usuba/tree/master/bench/rectangle`

The relative performance of *h*slicing, *v*slicing and bitslicing varies from a cipher to another, and from an architecture to another. For instance, on PYJAMASK, bitslicing is about 2.3 times faster than *v*slicing on SSE registers, 1.5 times faster on AVX, and as efficient on AVX512. On SERPENT however, bitslicing is 1.7 times slower than *v*slicing on SSE, 2.8 times slower on AVX, and up to 4 times slower on AVX512.

As explained in Chapter 2, we can specialize our polymorphic implementations to different slicing types and SIMD instruction sets. Usuba thus allows to easily compare the throughputs of all possible slicing modes of a cipher. In practice, few ciphers can be bitsliced, *v*sliced and *h*sliced: *h*slicing (except on AVX512) requires the cipher to rely on 16-bit (or smaller) values; arithmetic operations constraint ciphers to be only *v*sliced; bit-permutations prevent efficient *v*slicing, and often *h*slicing as well.

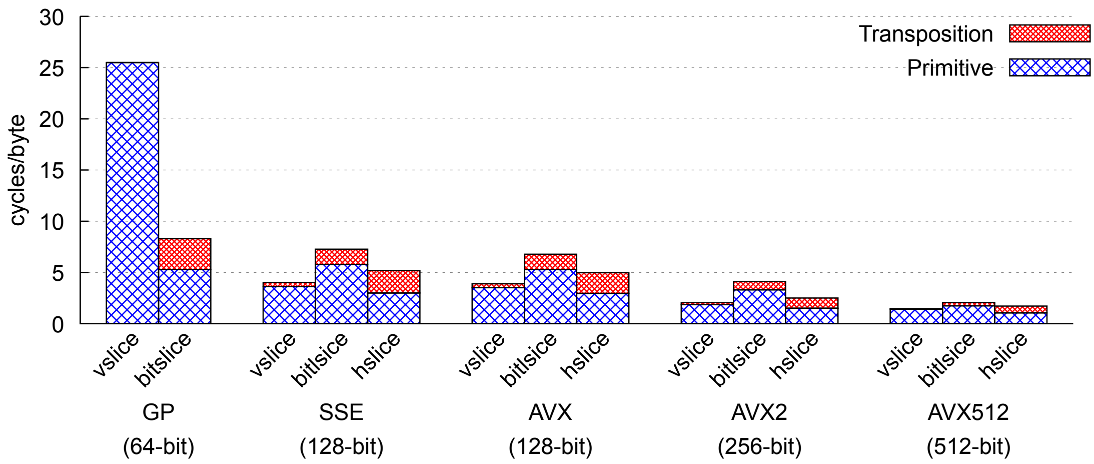


Figure 5.2: Comparison of RECTANGLE's monomorphizations

The only two ciphers compatible with all slicing types and all instruction sets are AES and RECTANGLE. The type of RECTANGLE in Usuba is:

```
node Rectangle(plain:u16x4, key:u16x4) returns (cipher:u16x4)
```

Compiling RECTANGLE for AVX2 in *v*sliced mode produces a C function whose type is:

```
void Rectangle (__m256i plain[4], __m256i key[26][4], __m256i cipher[4])
```

and that computes $256/16 = 16$ instances of RECTANGLE at once. Targeting instead SSE registers in bitsliced mode produces a C function whose type is:

```
void Rectangle (__m128i plain[64], __m128i key[26][64], __m128i cipher[64])
```

and that computes 128 (the size of the registers) instances of RECTANGLE at once.

Figure 5.2 shows the throughput of *v*sliced, *h*sliced and bitsliced RECTANGLE on general-purpose registers, SSE, AVX, AVX2 and AVX512; all of which were automatically generated from a single Usuba source of 31 lines. We ran this comparison on a Intel Xeon W-2155 CPU @ 3.30GHz, running Linux 5.4.0, and compiled the C code with Clang 9.0.1.

Slicing modes and transpositions have different relative performance depending on the architectures. On general-purpose registers, *v*sliced RECTANGLE processes inputs one by one, due to the lack of instruction to perform rotations/shifts on packed data in a 64-bit general-purpose registers. Bitslicing, however, does not require any architecture-specific instructions and can thus process 64 inputs in parallel on 64-bit registers, easily outperforming *v*slicing.

On SIMD extensions, the transposition plays a large part in the overall performance of the implementations. The *h*sliced primitive is faster than the *v*sliced one. However, transposing data for *h*slicing is more expensive, which results in lower performance for *h*slicing compared to *v*slicing. As mentioned Section 1.2.2 (Page 35), transposition can be omitted if both the algorithms used to encrypt and decrypt are using the same slicing. In such a case, *h*slicing should be preferred over *v*slicing for RECTANGLE.

Bitslicing is always slower than *m*slicing on SIMD instruction sets for RECTANGLE, as a result of spilling introduced by the huge pressure on the registers caused by bitslicing. *m*slicing on the other hand requires a few more instructions (to compute the rotations of the linear layer of RECTANGLE), but the reduced register pressure largely offset this overhead.

The results presented here on RECTANGLE do not constitute an absolute comparison of bitslicing, *v*slicing and *h*slicing. On AES for instance, *h*slice transposition is cheaper than on RECTANGLE, and *v*slicing requires much more instructions that *h*slicing to compute the `ShiftRows` step. As a result, *h*slicing is the obvious choice for AES, whereas *v*slicing seems more interesting for RECTANGLE.

Selecting a slicing mode thus requires forethought: the best mode depends on the ciphers, the available architectures, and the use-cases (*i.e.,* can the transposition be omitted?).

5.4 Conclusion

In this chapter, we evaluated the throughput of the implementations generated by Usubac on high-end Intel CPUs.
AES is the only cipher on which Usuba is significantly (3 to 5%) slower than hand-tuned assembly implementations. On the other implementations we considered, Usuba either performs on-par with hand-optimized implementations (*e.g.,* CHACHA20, SERPENT), or significantly better (*e.g.,* RECTANGLE, ACE). In all cases, the domain-specific optimizations (*i.e.,* scheduling and interleaving) carried by Usubac are key to achieve high throughputs with Usuba.

We also showed that, as claimed in the introduction, bitslicing and *m*slicing are able to scale well on SIMD extensions: AVX512 implementations reach throughputs up to 6 times greater than SSE implementations Usuba is able to automatically target those extensions, thus combining portability with high throughput.

Finally, we demonstrated that no slicing is strictly better than the others: bitslicing, *v*slicing and *h*slicing can all be good candidates depending on the cipher, and the targeted SIMD architecture. Usuba is thus a clear asset, since it allows to easily change slicing (through its types) and architecture.

Chapter 6

Protecting Against Power-based Side-channel Attacks

Publication

This work was done in collaboration with Sonia Belaïd (CryptoExperts), Matthieu Rivain (CryptoExperts), Raphaël Wintersdorff (CryptoExperts) and Pierre-Évariste Dagand (LIP6). It led to the following publication:

S. Belaïd, P. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, pages 311–341, 2020. doi: 10.1007/978-3-030-45727-3_11

CPUs leak information through timing [185], caches accesses [177, 238, 58], power consumption [77, 185, 213], electromagnetic radiation [251, 142, 255] and other compromising emanations [294, 303]. A way to recover this leaked information is to design a Simple Power Analysis (SPA) or a Differential Power Analysis (DPA) attack [186], which consist in analyzing power or EM traces from a cipher's execution in order to recover the key and plaintext.

Several approaches to mitigate SPA and DPA have been proposed. This includes inserting dummy instructions, working with both data and their complements, using specific hardware modules to randomize power consumption [118], or designing hardware gates whose power consumption is independent of their inputs [290, 245]. The most popular countermeasure against SPA and DPA, called *masking*, is, however, based on *secret sharing* [80, 276]. Masking, unlike some other countermeasures mentioned above, does not require custom hardware, and offers *provable security*: given an attacker model, one can prove that a masked code does not leak any secret data.

In this chapter, we integrate recent advances in secure masking [52] with Usuba to build a tool called Tornado, which generates masked implementations that are provably secure against SPA and DPA. These implementations are meant to be run on low-end embedded devices, featuring a few kilobytes of RAM, scalar CPU (*i.e.*, not superscalar), and lacking any SIMD extension. The primary objective here is to generate secure code, performance being only secondary. Still, we implemented several optimizations (Section 6.2.3 and 6.2.4) to speedup our implementations. We then demonstrate the benefits of Tornado by comparing the performances of 11 ciphers from the ongoing NIST lightweight cryptography competition (LWC) (Section 6.3).

6.1 Masking

The first masking schemes were proposed by Chari et al. [105] and Goubin and Patarin [150]. The principle of masking is to split each intermediate value b of a cipher into k random values $r_1 ... r_k$ (called *shares*) such that $b = r_1 \star r_2 \star ... \star r_k$ for some group operation $\star$. The most commonly used operation is the exclusive or (*Boolean masking*), but modular addition or modular multiplication can also be used (*arithmetic masking*) [214, 14].

A masked implementation using 2 shares is said to be *first-order* masked. Early masked implementations were only first-order masked [214, 14, 83, 146], which is sufficient to protect against first-order DPA. However, *higher order DPA* (HO-DPA) can be designed by combining multiple sources of leakage to reconstruct the secrets, and can break first-order masked implementations [186, 213, 295, 168]. Several masked implementations using more than two shares (*higher-order masking*) where thus proposed to resist HO-DPA [165, 262, 273, 263, 97]. The term *masking order* is used to refer to the number of shares minus one. Similarly, an attack is said to be of order t if it combines t sources of leakage.

The *probing-model* is widely used to analyze the security of masked software implementations against side-channel attacks. This model was introduced by Ishai et al. [165] to construct circuits resistant to hardware probing attacks. It was latter shown [262, 97, 117] that this model and the underlying constructions were instrumental to the design of efficient secure masked cryptographic implementations. A masking scheme secure against a t-probing adversary, *i.e.,* an adversary who can probe t arbitrary variables during the computation, is indeed secure by design against the class of side-channel attacks of order t [116].

Most masking schemes consider the cipher as a Boolean or arithmetic circuit. Individual gates of the circuit are replaced by gadgets that process masked variables. One of the important contributions of Ishai et al. [165] was to propose a multiplication gadget secure against t-probing attacks for any t, based on a Boolean masking of order $n = 2t+1$. This was reduced to the tight order $n = t + 1$ by Rivain et al. [263] by constraining the two input sharings to be independent, which could be ensured by the application of a mask-refreshing gadget when necessary. The design of secure refresh gadgets and, more generally, the secure composition of gadgets were subsequently subject to many works [117, 44, 46]. Of particular interest, the notions of Non-Interference (NI) and Strong Non-Interference (SNI) introduced by Barthe et al. [44] provide a practical framework for the secure composition of gadgets that yields tight probing-secure implementations. In a nutshell, such implementations are composed of ISW multiplication and refresh gadgets (from the names of their inventors Ishai, Sahai, and Wagner [165]) achieving the SNI property, and of share-wise addition gadgets. The main difficulty consists in identifying the minimal number of refresh gadgets and their (optimal) placing in the implementation to obtain a provable t-probing security.

Terminology. In the context of a Boolean circuit, multiplications refer to `and` and `or` gates, while additions refer to `xor` gates. Multiplications (and therefore `and` and `or`) are said to be *non-linear*, while additions (and therefore `xor`) are said to be *linear*.

6.2 Tornado

Belaïd et al. [52] introduced tightPROVE, a tool that verifies the t-probing security of a masked implementation at any order t. This verification is done in the bit probing model, meaning that each probe only retrieves a single bit of data. It is thus limited to verify the security of programs manipulating 1-bit variables. Raphaël Wintersdorff, Sonia Belaïd, and Matthieu Rivain recently developed tightPROVE$^+$, an extension of tightPROVE to

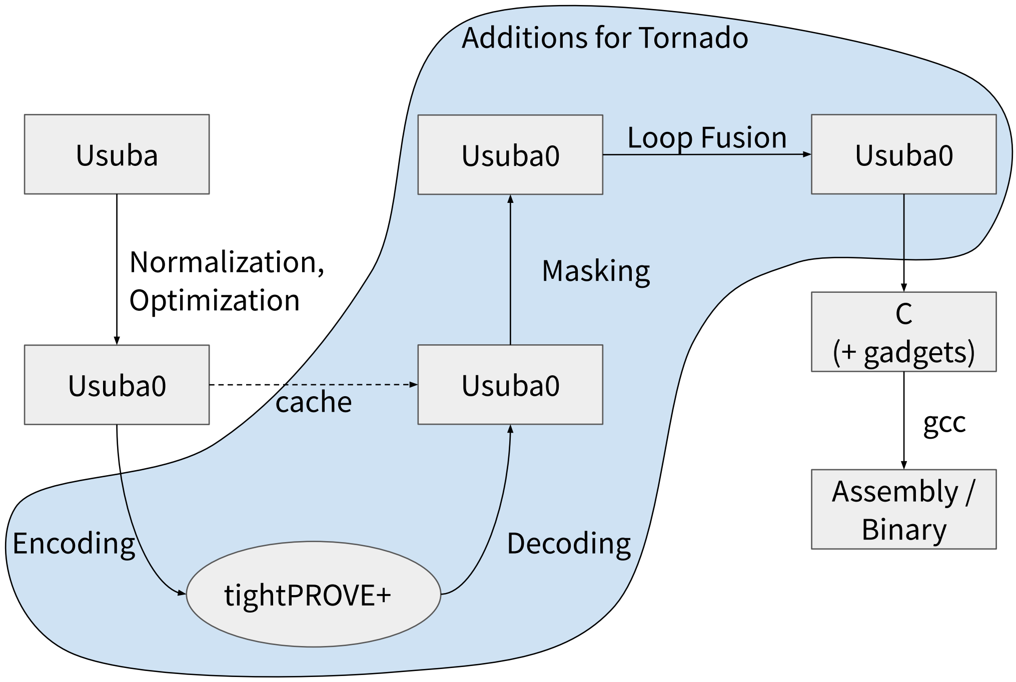


Figure 6.1: Tornado's pipeline

the register probing model where variables can hold an arbitrary number of bits, all of which leak simultaneously. Furthermore, tightPROVE$^+$ automatically insert refreshes when needed, whereas tightPROVE was only able to detect that some refreshes were missing. We worked with them to integrate tightPROVE$^+$ with Usuba [53]. The resulting tool is called Tornado. Tornado generates provably t-probing secure masked implementations in both the bit-probing and the register-probing model.

Given a high-level description of a cryptographic primitive (*i.e.,* an Usuba implementation), Tornado synthesizes a masked implementation using ISW-based multiplication and refresh gadgets [165], and share-wise addition gadgets. The key role of Usuba is to automate the generation of a sliced implementation, upon which tightPROVE$^+$ is then able to verify either the bit probing or register probing security, or suggest the necessary refreshes. By integrating both tools, we derive a probing-secure masked implementation from a high-level description of a cipher.

The overall architecture of Tornado is shown in Figure 6.1. After normalization and optimization, the Usuba0 program is sent to tightPROVE$^+$, which adds refreshes if necessary (Section 6.2.1). The output of tightPROVE$^+$ is translated back to Usuba0, which Usubac then masks (Section 6.2.2): variables are replaced with arrays of shares, linear operators and non-linear operators involving constants are applied share-wise, and other non-linear operators and refreshes are left as is to be replaced by masked gadgets during C code generation. Before generating C code, a pass of *loop fusion* merges share-wise applications of linear operators when possible (Section 6.2.4).

6.2.1 Encoding and Decoding Usuba0 for tightPROVE$^+$

Tornado targets low-end embedded devices with drastic memory constraints. It is therefore essential to produce a C code that uses loops and functions, in order to reduce code size, especially when masking at high orders. This is one of the main motivations to keep loops and nodes in the Usuba pipeline.

```c
static void isw_mult(uint32_t *res,
                     const uint32_t *op1,
                     const uint32_t *op2) {
  for (int i=0; i<=MASKING_ORDER; i++)
    res[i] = 0;

  for (int i=0; i<=MASKING_ORDER; i++) {
    res[i] ^= op1[i] & op2[i];

    for (int j=i+1; j<=MASKING_ORDER; j++) {
      uint32_t rnd = get_random();
      res[i] ^= rnd;
      res[j] ^= (rnd ^ (op1[i] & op2[j]))
                ^ (op1[j] & op2[i]);
    }
  }
}
```

(a) Multiplication

```c
static void isw_refresh(uint32_t *res,
                        const uint32_t *in) {
  for (int i=0; i<=MASKING_ORDER; i++)
    res[i] = in[i];

  for (int i=0; i<=MASKING_ORDER; i++) {
    for (int j=i+1; j<=MASKING_ORDER; j++) {
      uint32_t rnd = get_random();
      res[i] ^= rnd;
      res[j] ^= rnd;
    }
  }
}
```

(b) Refresh

Figure 6.2: ISW gadgets

However, tightPROVE$^+$ only supports straight-line programs, providing no support for loops and functions. Usubac thus fully unrolls and inlines the Usuba0 code before interacting with tightPROVE$^+$. The refreshes inserted by tightPROVE$^+$ must then be propagated back into the Usuba0 program, featuring loops, nodes and node calls.

To do so, when inlining and unrolling, we keep track of the origin of each instruction: which node they come from, and which instruction within that node. For each refresh inserted by tightPROVE$^+$, we therefore know where (*i.e.*, which node and which loop, if any) the refreshed variable comes from, and are therefore able to insert the refresh directly in the right node.

In order to help tightPROVE$^+$ find where to insert refreshes, a first pass sends each node (before inlining) to tightPROVE$^+$ for verification. Those nodes are smaller than the fully inlined program, and tightPROVE$^+$ is thus faster to detect the required refreshes. Only then are all nodes inlined and the whole program is sent to tightPROVE$^+$. This is useful for instance on ACE, which contains the following node:

```
node f(x:u32) returns (y:u32)
let
    y = ((x <<< 5) & x) ^ (x <<< 1)
tel
```

This node is obviously vulnerable to probing attacks, and tightPROVE$^+$ is able to insert a refresh on one of the `x` to make it probing-secure. Once the secure version of `f` is inlined in ACE, no other refreshes are needed.

6.2.2 Masking

To mask an Usuba0 program, Usubac replaces each variable with an array of shares, and each operator with a masked gadget. This is a source-to-source, typed transformation, which produces a typeable Usuba0 program. The gadget to mask a linear operator (`xor`) is simply a loop applying the operator on each share, written directly in Usuba. To mask a negation, we only negate the first share (since `~(r0 ^ r1 ^ ... ^ rk) == ~r0 ^ r1 ^ ... ^ rk`), still in Usuba. To mask non-linear operators (`and` and `or`), however, we introduce the operators `m&` and `m|` in Usuba, which are transformed into calls to masked C gadgets when generating C code. In particular, for the multiplication (`and`), we use the so-called ISW gadget, introduced by Ishai et al. [165], shown in Figure 6.2a. Note that this algorithm has a quadratic complexity in the number of shares.

```
node mat_mult(col:u32,vec:u32) returns (res:u32)
vars
    mask:u32
let
    res  = 0;

    forall i in [0, 31] {
        mask := bitmask(vec,i);
        res  := res ^ (mask & col);
        col  := col >>> 1;
    }
tel
```

Figure 6.3: PYJAMASK matrix multiplication node

`or`s are transformed into `not`s and `and`s, since `a | b == ~(~a & ~b)`. Similarly, refreshes, either inserted manually by the developer or automatically by tightPROVE$^+$, are compiled into calls to the ISW refresh routine (Figure 6.2b) when generating C code.

6.2.3 Constants

Constants are not secret values and thus do not need to be masked. Furthermore, when multiplying a constant with a secret value, there is no need to use the ISW multiplication gadgets: we can simply multiply each share of the secret value with the constant. The cost of masking a multiplication by a constant is thus linear in the number of shares, rather than quadratic. Our benchmarks (Section 6.3) show that indeed, the more masked multiplication a cipher has, the slower it is. Avoiding unnecessary masked multiplications is thus essential.

For instance, PYJAMASK's linear layer contains a matrix multiplication, implemented by calling 4 times the `mat_mult` node (Figure 6.3), which multiplies two 32-bit vectors `col` and `vec`. When this node is called in PYJAMASK, `col` is always constant, and the multiplication (`mask & col`) does not need to be masked. Usubac uses a simple static analysis to track which variables are constant and which are not, and is thus able to identify that `col` does not need to be shared, and that the multiplication `mask & col` does not need to be masked. This optimization both reduces stack usage (since constant variables are kept as a single share rather than an array of shares) and increases performance (since multiplying by a constant becomes linear rather than quadratic).

6.2.4 Loop Fusion

Since each linear operation is replaced with a loop applying the operation on each share, the masked code contains a lot of loops. The overhead of those loops is detrimental to performance. Consider for instance the following Usuba0 snippet:

```
x = a ^ b;
y = c ^ d;
z = x ^ y;
```

After masking, it becomes:

```
forall i in [0, MASKING_ORDER-1] {
    x[i] = a[i] ^ b[i];
}
forall i in [0, MASKING_ORDER-1] {
    y[i] = c[i] ^ d[i];
}
forall i in [0, MASKING_ORDER-1] {
    z[i] = x[i] ^ y[i];
}
```

In order to reduce the overhead of looping over each share for linear operations, we aggressively perform loop fusion (also called *loop jamming*) in Usubac, which consists in replacing multiple loops with a single one. The above snippet is thus optimized to:

```
forall i in [0, MASKING_ORDER-1] {
    x[i] = a[i] ^ b[i];
    y[i] = c[i] ^ d[i];
    z[i] = x[i] ^ y[i];
}
```

Loop fusion is also able to reduce the amount of stack used by allowing Usubac to replace temporary arrays with temporary variables. In the example above, Usubac would thus convert `x` and `y` into scalars rather than arrays and produce the following code:

```
forall i in [0, MASKING_ORDER-1] {
    x = a[i] ^ b[i];
    y = c[i] ^ d[i];
    z[i] = x ^ y;
}
```

On embedded devices (which Tornado targets), this is especially beneficial since the amount of stack available is very limited (*e.g.,* a few dozens of kilobytes).

C compilers also perform loop fusion, but experimentally, they are less aggressive than Usubac. We thus obtain better performance by fusing loops directly in Usubac. On the 11 ciphers of the NIST LWC we implemented in Usuba and compiled with Tornado, performing loop fusion in Usubac allows us to reduce stack usage of our bitsliced implementations by 11kB on average whereas this saves us, on average, 3kB of stack for our *m*sliced implementations (note that our platform offers a measly 96kB of SRAM). It also positively impacts throughput, with a 16% average speedup for bitslicing and a 21% average speedup for *m*slicing.

6.3 Evaluation

We used Tornado to compare 11 cryptographic primitives from the second round of the ongoing NIST LWC. The choice of cryptographic primitives was made on the basis that they were self-identified as being amenable to masking. We stress that we do not focus on the full authenticated encryption, message authentication, or hash protocols but only on the underlying primitives, mostly block ciphers and permutations. The list of primitives and the LWC submissions they appear in is given in Table 6.1.

Whenever possible, we generate both a bitsliced and an *m*sliced implementation for each primitive, which allows us to exercise the bit-probing and the register-probing models of tightPROVE$^+$. However, 4 primitives do not admit straightforward *m*sliced implementations. The SUBTERRANEAN permutation involves a significant amount of bit-twiddling across its 257-bit state, which makes it a resolutely bitsliced primitive (as confirmed by its reference implementation). PHOTON, SKINNY, SPONGENT rely on lookup

Submission	Primitive	*m*sliceable	Slice size	State size (bits)
block ciphers				
GIFT-COFB [31], HYENA [103], SUNDAE-GIFT [30]	GIFT-128	✓	32	128
PYJAMASK [151]	PYJAMASK-128	✓	32	128
SKINNY [51], ROMULUS [166]	SKINNY-128-256	✗	-	128
Spook [55]	CLYDE-128	✓	32	128
permutations				
ACE [7]	ACE	✓	32	320
ASCON [128]	p^{12}	✓	64	320
Elephant [69]	SPONGENT-π[160]	✗	-	160
GIMLI [68]	GIMLI-36	✓	32	384
ORANGE [104], PHOTON-BEETLE [34]	PHOTON-256	✗	-	256
Xoodyak [121]	XOODOO	✓	32	384
others				
SUBTERRANEAN [122]	blank(8)	✗	-	257

Table 6.1: Overview of the primitives selected to evaluate Tornado

tables that would be too expensive to emulate in *m*slice mode. In bitslicing, these tables are simply implemented by their Boolean circuit, either provided by the authors (PHOTON, SKINNY) or generated through SAT [284] with the objective of minimizing multiplicative complexity (SPONGENT, with 4 `and`s and 28 `xor`s).

Note that the *m*sliced implementations, when they exist, are either 32-sliced or 64-sliced. This means in particular that, unlike bitslicing that allows processing multiple blocks in parallel, these implementations process a single block at once on our 32-bit Cortex M4. Note also that, in practice, all *m*sliced implementations are *v*sliced, since the Cortex M4 we consider does not provide SIMD extensions, which are required for *h*slicing.

Running tightPROVE$^+$ on our Usuba implementations showed that all of them were t-probing secure in the bit-probing model, but 3 were not secure in the register-probing model. ACE required 384 additional refreshes, CLYDE-128 required 6 refreshes, and GIMLI required 120 refreshes. In the case of ACE and CLYDE, the number of refreshes inserted by tightPROVE$^+$ is known to be minimal, while for GIMLI, it is only an upper bound [53].

6.3.1 Baseline Performance Evaluation

! The benchmarks of this section are available at:
`https://github.com/DadaIsCrazy/usuba/blob/master/bench_nist.pl`

In the following, we benchmark our implementations—written in Usuba and compiled by Tornado—of the NIST LWC submissions against the reference implementation provided by the contestants. This allows us to establish a performance baseline (without masking), thus providing a common frame of reference for the performance of these

primitive	Throughput (cycles/bytes) *(lower is better)*		
	Usuba *m*slice	Usuba bitslice	reference
ACE	34.25	55.89	273.53
ASCON	9.84	4.94	5.18
CLYDE	33.72	21.99	37.69
GIMLI	15.77	5.80	44.35
GIFT	565.30	45.51	517.27
PHOTON	-	44.88	214.47
PYJAMASK	246.72	131.33	267.35
SKINNY	-	46.87	207.82
SPONGENT	-	146.93	4824.97
SUBTERRANEAN	-	17.64	355.38
XOODOO	14.93	6.47	10.14

Table 6.2: Comparison of Usuba *vs.* reference implementations on Intel

primitives based on their implementation synthesized from Usuba. In doing so, we have to bear in mind that the reference implementations provided by the NIST contestants are of varying quality: some appear to have been finely tuned for throughput while others focus on simplicity, acting as an executable specification.

Several candidates of the NIST LWC do not provide implementations for ARM devices. However, since they all provide a reference (generic) C implementation, we first benchmarked our implementations against the reference implementations on an Intel i5-6500 @ 3.20GHz, running Linux 4.15.0-54. The implementations were compiled with Clang 7.0.0 with flags `-O3 -fno-slp-vectorize -fno-vectorize`. These flags prevent Clang from trying to produce vectorized code, which would artificially advantage some implementations at the expense of others because of brittle, hard-to-predict vectorization heuristics. At the exception of SUBTERRANEAN (which is bitsliced), the reference implementations are *v*sliced, representing the state of the primitive through a matrix of 32-bit values, or 64-bit in the case of ASCON. To evaluate bitsliced implementations, we simulate a 32-bit architecture, meaning that the throughput we report corresponds to the parallel encryption of 32 independent blocks. The cost of transposing data into a bitslice format (around 9 cycles/bytes to transpose a 32×32 matrix) is excluded.

We disabled nonessential unrolling in Usubac, and used less aggressive inlining than in Section 5.1 (Page 125), in order to use the same settings as when generating masked implementations for ARM. As a consequence, we are slower across the board compared to the results reported in Section 5.1.

The results are shown in Table 6.2. We notice that Usuba often delivers throughputs that are on par or better than the reference implementations. Note that this does not come at the expense of readability: our Usuba implementations are written in a high-level language. The reference implementations of SKINNY, PHOTON and SPONGENT use lookup tables, which do not admit a straightforward, constant-time implementation. As a result, we are unable to implement a constant-time *m*sliced version in Usuba and to mask such an implementation. We now turn our attention specifically to a few implementations that exhibit interesting performance.

SUBTERRANEAN's reference implementation is an order of magnitude slower than in Usuba because its implementation is bit-oriented (each bit is stored in a distinct 8-bit variable) but only a single block is encrypted at a time. Switching to 32-bit variables and encrypting 32 blocks in parallel, as Usuba does, significantly improves throughput.

SPONGENT is slowed done by a prohibitively expensive bit-permutation over its 160-bit state, which is spread across 20 8-bit variables. Thanks to bitslicing, Usuba turns this permutation into a static renaming of variables, which occurs at compile-time. The reference implementation, however, is not bitsliced and cannot apply this trick.

ASCON. Our *m*slice implementation of ASCON is twice slower than the reference implementation. Unlike the reference implementation, we have refrained from performing aggressive function inlining and loop unrolling to keep code size in check, since we target embedded systems. However, if we instruct the Usuba compiler to perform these optimizations, our *m*slice implementation outperforms the reference one, as shown in Section 5.1 (Page 125).

ACE's reference implementation suffers from significant performance issues, relying on an excessive number of temporary variables to store intermediate results. Usuba does not have such issues, and thus easily outperforms it.

GIMLI offers two reference implementations, one being a high-performance SSE implementation with the other serving as an executable specification on general-purpose registers. We chose the general-purpose one here (which had not been subjected to the same level of optimization) because our target architecture (Cortex M) does not provide a vectorized instruction set.

GIFT's *m*slice implementation suffers from severe performance issues because of its expensive linear layer. Using the recent fixslicing technique [8] improves the throughput of Usuba's *m*slice GIFT implementation down to 42 cycles/bytes.

6.3.2 Masking Benchmarks

> **!** The benchmarks of this section are available at:
> `https://github.com/pedagand/usuba-nucleo`

We ran our benchmarks on a Nucleo STM32F401RE offering an ARM Cortex-M4 with 512 kilobytes of Flash memory and 96 kilobytes of SRAM. We compiled our implementations using arm-none-eabi-gcc version 9.2.0 at optimization level `-O3`. We considered two modes regarding the Random Number Generator (RNG):

- **Pooling mode**: The RNG generates random numbers at a rate of 32 bits every 64 clock cycles. Fetching a random number can thus take up to 65 clock cycles.
- **Fast mode:** The RNG has a production rate faster than the consumption rate of the program. The RNG routine thus can simply read a register containing this 32-bit random word without checking for its availability.

These two modes were chosen because they are the ones used in PYJAMASK' LWC submission, which is the only submission addressing the issue of getting random numbers for a masked implementation.

Primitive	Mults /byte	RNG mode	Performance (cycles/byte) (lower is better)						
			$d=0$	$d=3$	$d=7$	$d=15$	$d=31$	$d=63$	$d=127$
ASCON	1.375	fast	49	1.1k	3.1k	11.6k	42.5k	163k	640k
		pooling	49	1.3k	4.6k	20.5k	79.2k	324k	1.3M
XOODOO	1.5	fast	63	889	3.3k	10.8k	39.4k	143k	555k
		pooling	63	1.7k	7.0k	29.1k	113k	448k	1.7M
CLYDE	3	fast	92	961	3.5k	11.8k	41.9k	161k	653k
		pooling	92	1.9k	7.6k	31.4k	121k	483k	1.9M
PYJAMASK	3	fast	994	5.0k	12.8k	38.4k	108k	297k	950k
		pooling	994	5.9k	17.2k	59.7k	194k	646k	2.3M
GIMLI	6	fast	56	1.8k	7.1k	24.7k	95.2k	356k	1.4M
		pooling	56	4.0k	17.4k	73.4k	293k	1.2M	4.6M
GIFT	10	fast	1.1k	12.5k	32.3k	77.6k	285k	819k	2.6M
		pooling	1.1k	15.3k	44.7k	138k	532k	1.8M	6.4M
ACE	19.2	fast	92	3.9k	13.3k	40.1k	190k	746k	2.8M
		pooling	92	7.5k	32.9k	114k	495k	2.0M	7.8M

(a) Cycles per byte

Primitive	Mults	RNG mode	Performance (cycles) (lower is better)						
			$d=0$	$d=3$	$d=7$	$d=15$	$d=31$	$d=63$	$d=127$
CLYDE	48	fast	1.5k	15.4k	56.5k	189.4k	670.1k	2.6M	10.4M
		pooling	1.5k	30.1k	121.1k	502.9k	1.9M	7.7M	29.9M
PYJAMASK	56	fast	15.9k	79.5k	205.4k	614.4k	1.7M	4.8M	15.2M
		pooling	15.9k	94.9k	274.6k	954.6k	3.1M	10.3M	36.3M
ASCON	60	fast	2.0k	42.0k	123.2k	464.4k	1.7M	6.5M	25.6M
		pooling	2.0k	53.6k	182.8k	821.6k	3.2M	13.0M	52.0M
XOODOO	144	fast	3.0k	42.7k	156.5k	520.3k	1.9M	6.9M	26.6M
		pooling	3.0k	82.1k	334.1k	1.4M	5.4M	21.5M	83.0M
GIFT	160	fast	17.9k	200.5k	516.3k	1.2M	6.5M	13.1M	42.2M
		pooling	17.9k	244.3k	714.9k	2.2M	8.5M	29.1M	102.4M
GIMLI	288	fast	2.7k	85.0k	342.7k	1.2M	4.6M	17.1M	67.2M
		pooling	2.7k	190.6k	832.8k	3.5M	14.1M	56.2M	218.9M
ACE	384	fast	3.7k	155.2k	531.6k	1.6M	7.6M	29.8M	113.6M
		pooling	3.7k	302.0k	1.3M	4.6M	19.8M	78.4M	310.8M

(b) Cycles per block

Table 6.3: Performance of Tornado *m*sliced masked implementations

*m*slicing Scaling

Table 6.3a gives the throughputs of the *m*sliced implementations produced by Tornado in terms of cycles per byte. Since masking a multiplication has a quadratic cost in the number of shares, we expect performance at high orders to be mostly proportional with the number of multiplications used by the primitives. We thus report the number of multiplications involved in our implementations of the primitives. We observe that, the higher the masking order, the better ciphers with few multiplications perform (compared to ciphers with more multiplications), confirming this effect. This is less pronounced at small orders since the execution time remains dominated by linear operations. Using the pooling RNG increases the cost of multiplications compared to the fast RNG, which results in performance being proportional to the number of multiplications at smaller order than with the fast RNG.

PYJAMASK illustrates the influence of the number of multiplications on scaling. Because of its use of dense binary matrix multiplications, it involves a significant number of linear operations for only a few multiplications. As a result, it is slower than GIMLI and ACE at order 3, despite the fact that they use respectively 2× and 6× more multiplications. With the fast RNG, the inflection point is reached at order 7 for ACE and order 31 for GIMLI, only to improve afterward. Similarly when compared to CLYDE, PYJAMASK goes from 5× slower at order 3 to 1.5× slower at order 127 with the fast RNG and 1.2× slower at order 127 with the pooling RNG. The same analysis applies to GIFT and ACE, where the linear overhead of GIFT is only dominated at order 63 with the pooling RNG and at order 127 with the fast RNG.

One notable exception is ASCON with the fast RNG, compared in particular to XOODOO and CLYDE. Whereas ASCON uses a smaller number of multiplications, it involves a 64-sliced implementation (Table 6.1), unlike its counterparts that are 32-sliced. Running on our 32-bit Cortex-M4 requires GCC to generate 64-bit emulation code, which induces a significant operational overhead and prevents further optimization by the compiler. When using the pooling RNG however, ASCON is faster than both XOODOO and CLYDE at every order, thanks to its smaller number of multiplications.

For scenarios in which one is not interested in encrypting a lot of data but rather a single block, possibly short, then it makes more sense to look at the performances of a single run of a cipher, rather than its amortized performances over the amount of bytes it encrypts. This is shown in Table 6.3b. The ciphers that use the least amount of multiplications have the upper hand when masking order increases: CLYDE is clearly the fastest primitive at order 127, closely followed by PYJAMASK. ASCON, which is the fastest one when looking at the cycles per bytes actually owns its performances to his low number of multiplications compared to its 320-bit block size. Therefore, when looking at a single run, it is actually 1.7× slower than CLYDE at order 127. Similarly, XOODOO performs well on the cycles per bytes metric, but has a block size of 384 bits, making it 2.5× slower than CLYDE.

Bitslicing Scaling

The key limiting factor to execute bitsliced code on an embedded device is the amount of memory available. Bitsliced programs tend to be large and to consume a significant amount of stack. Masking such implementations at high orders quickly becomes impractical because of the quadratic growth of the stack usage.

To reduce stack usage and allow us to explore high masking orders, our bitsliced programs manipulate 8-bit variables, meaning that 8 independent blocks can be processed in parallel. This trades memory usage for throughput, as we could have used 32-bit variables and improved our throughput by a factor 4. However, doing so puts an unbearable amount of pressure on the stack, which would have prevented us from considering masking orders beyond 7. Besides, it is not clear whether there is a use-case for such a massively parallel (32 independent blocks) encryption primitive in a lightweight setting. As a result of our compilation strategy, we have been able to mask all primitives with up to 16 shares and, additionally, reach 32 shares for PHOTON, SKINNY, SPONGENT and SUBTERRANEAN.

Table 6.4a and 6.4b report the throughput of our bitsliced implementations with the fast and pooling RNGs. As for the *m*sliced implementations, we observe a close match between the asymptotic throughput of the primitive and their number of multiplications per bits, which becomes even more prevalent as order increases and the overhead of linear operations becomes comparatively smaller. PYJAMASK remains a good example to illustrate this phenomenon, the inflection point being reached at order 15 with respect to ACE (which uses 3× more multiplications).

While ASCON with the fast RNG is slowed down by its suboptimal use of 64-bit registers in *m*slicing, its throughput in bitslicing is similar to XOODOO's, which exhibits the same number of multiplications per bits. Finally, we observe that with the pooling RNG, and starting at order 15, the throughput of our implementations is in accord with their relative number of multiplications per bits. In bitslicing (more evidently than in

Primitive	Mults/bits	Throughput (cycles/bytes) (lower is better)				
		$d=0$	$d=3$	$d=7$	$d=15$	$d=31$
SUBTERRANEAN	8	94	2.1k	7.2k	27.2k	95.2k
ASCON	12	101	3.1k	11.4k	42.4k	-
XOODOO	12	112	3.1k	10.5k	38.4k	-
CLYDE	12	177	3.4k	13.6k	45.3k	-
PHOTON	12	193	7.7k	14.3k	45.0k	154k
PYJAMASK	14	1.6k	16.5k	31.8k	97.9k	-
GIMLI	24	127	5.5k	19.1k	76.9k	-
ACE	38	336	8.2k	35.3k	123k	-
GIFT	40	358	11.1k	36.8k	136k	-
SKINNY	48	441	18.2k	61.8k	200k	664k
SPONGENT	80	624	19.4k	64.8k	259k	948k

(a) Fast RNG

Primitive	Mults/bits	Throughput (cycles/bytes) (lower is better)				
		$d=0$	$d=3$	$d=7$	$d=15$	$d=31$
SUBTERRANEAN	8	94	4.46k	19.13k	79.63k	312k
ASCON	12	101	7.33k	30.33k	125k	-
XOODOO	12	112	6.69k	28.79k	120k	-
CLYDE	12	177	7.88k	31.04k	127k	-
PHOTON	12	193	10.47k	31.77k	126k	476k
PYJAMASK	14	1.59k	20.33k	52.81k	193k	-
GIMLI	24	127	12.14k	53.64k	236k	-
ACE	38	336	19.94k	89.12k	395k	-
GIFT	40	358	21.38k	93.92k	405k	-
SKINNY	48	441	34.28k	131k	525k	1.97M
SPONGENT	80	624	44.04k	188k	816k	3.15M

(b) Pooling RNG

Table 6.4: Throughput of Tornado bitsliced masked implementations

*m*slicing), the number of multiplications is performance critical, even at relatively low masking order.

6.3.3 Usuba Against Hand-Tuned Implementations

Of these 11 NIST submissions, only PYJAMASK provides a masked implementation. Our implementation is consistently (at every order, and with both the pooling and fast RNGs) 1.8 times slower than their masked implementation. The main reason is that they manually optimized in assembly the two most used routines.

The first one is the matrix multiplication shown in Figure 6.3 (Page 139). When compiled with arm-none-eabi-gcc 4.9.3, the body of the loop compiles to 6 instructions. The reference implementation, on the other hand, inlines the loop and uses ARM's barrel shifter. As a result, each iteration takes only 3 instructions. Using this hand-tuned matrix multiplication in our generated code offers a speedup of 15% to 52%, depending on the masking order, as shown in Table 6.5a. The slowdown is less important at high masking orders because this matrix multiplication has a linear cost in the masking order, while the masked multiplication has a quadratic cost in the masking order (since it must compute

masking order	3	7	15	31	63	127
speedup	52%	49%	43%	34%	24%	15%

(a) Hand-Tuned `mat_mult` compared to Usuba's `mat_mult`

masking order	3	7	15	31	63	127
speedup	14%	21%	33%	37%	39%	41%

(b) Hand-tuned ISW multiplication compared to the C implementation of Figure 6.2a

Table 6.5: Speedups gained by manually implementing key routines in assembly

all cross products between two shared values).

The second hand-optimized routine is the masked multiplication. The authors of the reference implementation used the same ISW multiplication gadget as we used in Usuba (shown in Figure 6.2a) but manually optimized it in assembly. They unrolled twice the main loop of the multiplication, then fused the two resulting inner loops, and carefully assigned 6 variables to registers. If we note m the masking order, this optimization allows the masked multiplication to require m fewer jumps ($m/2$ for inner loops and $m/2$ outer loops), to use $m/2 \times 10$ less memory accesses thanks to their manual assignment of array values to registers, and to require $m/2 \times 3$ less memory accesses thanks to the merged inner loops. GCC fails to perform this optimization automatically.

Unlike the hand-optimized matrix multiplication, whose benefits diminish as the masking order increases, the optimized masked multiplication is more beneficial at higher masking order, as shown in Table 6.5b, which compares the hand-tuned assembly ISW multiplication against the C implementation of Figure 6.2a.

While the Usuba implementations can be sped up by using the hand-tuned version of the ISW masked multiplication, these two examples (matrix multiplication and masked multiplication) hint at the limits of achieving high performance by going through C. Indeed, while the superscalar nature of high-end CPUs can make up for occasional suboptimal code produced by C compilers, embedded devices are less forgiving. Similarly, Schwabe and Stoffelen [274] designed high-performance implementations of AES on Cortex-M3 and M4 microprocessors, and preferred to write their own register allocator and instruction scheduler (in fewer than 250 lines of Python), suggesting that GCC's register allocator and instruction scheduler may be far from optimal for cryptography on embedded devices.
We leave to future work to directly target ARM assembly in Usuba, and to implement architecture-specific optimizations for embedded microprocessors.

6.4 Conclusion

In this chapter, we extended Usubac to automatically generate probing-secure implementations in the bit-probing and register-probing model. By relying on tightPROVE^{+}, we are able to achieve provable security.

We used this extension of Usubac to compare the performance of 11 ciphers from the ongoing NIST lightweight cryptography competition, at masking orders varying from 3 to 127.

6.4.1 Related Work

Several approaches have been proposed to automatically secure implementations against power-based side-channel attacks.

Bayrak et al. [47] designed one of the first automated tools to protect cryptographic code at the software level. Their approach is purely empirical: they identify sensitive operations by running the cipher while recording power traces. The operations that are found to leak information are then protected using random precharging [289], which consists in adding random instructions before and after the leaking instructions so as to lower the signal-to-noise ratio.

Agosta et al. [12] developed a tool to automatically protect ciphers against DPA using code morphing, which consists in dynamically (at runtime) generating and randomly choosing the code to be executed. To do so, the cipher is broken into fragments of a few instructions. For each fragment, several alternative (but semantically equivalent) sequences of instructions are generated at compile time. At runtime, a polymorphic engine selects one sequence from each of the fragments. The resulting code can leak secret data, but an attacker would not know where in the original cipher the leak is, since the code executing is dynamically randomized.

Moss et al. [221] proposed the first compiler to automate first-order Boolean masking, based on CAO [37]. Developers add `secret` or `public` annotations to the inputs of the program, and the compiler infers for each computation whether the result is `secret` (one of its operands is `secret`) or public (all operands are `public`). The compiler then searches for operations on `secret` values that break the masking (for instance a `xor` where the masks of the operands would cancel out), and fixes them by changing the masks of its operands.

Agosta et al. [13] designed a dataflow analysis to automatically mask ciphers at higher orders. Their analysis tracks the secret dependencies (plaintext or key) of each intermediate variable, and is thus able to detect intermediate results that would leak secret information and need to be masked. They implemented their algorithm in LLVM [196], allowing them to mask ciphers written in C (although they require some annotations in the C code to identify the plaintext and key).

Sleuth [48] is a tool to automatically verify whether an implementation leaks information that is statistically dependent for any secret input and would thus be vulnerable to SCA. Sleuth reduces this verification to a Boolean satisfiability problem, which is solved using a SAT solver. Sleuth is implemented as a pass in the LLVM backend, and thus ensures that the compiler does not break masking by reordering instructions.

Eldib et al. [133] developed a tool similar to Sleuth called SC Sniffer. However, whereas Sleuth verifies that all leakage is statistically independent of any input, SC Sniffer ensures that a cipher is perfectly masked, which is a stronger property: a masked algorithm is perfectly masked at order d if the distribution of any d intermediate results is statistically independent of any secret input [83]. SC Sniffer is also implemented as an LLVM pass and relies on the Yices SMT solver [131] to conduct its analysis. Eldib and Wang [132] then built MC Masker, to perform synthesis of masking countermeasures based on SC Sniffer's analysis. MC Masker generates the masked program using inductive synthesis: it iteratively adds countermeasures until the SMT solver is able to prove that the program is perfectly masked.

Barthe et al. [43] proposed to verify masking using program equivalence. More specifically, their tool (later dubbed maskverif), inspired by Barthe et al. [41], tries to construct a bijection between the distribution of intermediate values of the masked program of interest and a distribution that is independent from the secret inputs. maskverif scales better than previous work (*e.g.,* MC Masker) at higher masking order, even though it is not able to handle a full second-order masked AES. Furthermore, maskverif is not limited to analyze 1-bit variables, unlike Sleuth and SC Sniffer.

Barthe et al. [44] later introduced the notion of *strong non-interference*. Intuitively, a strongly t-non-interfering code is t-probing secure (or *t-non-interfering*) and composable with any other strongly t-non-interfering code. This is a stronger property than t-probing security, since combining t-probing secure gadgets does not always produce a t-probing secure implementation [117]. They developed a tool (later dubbed maskComp) to generate masked C implementations from unmasked ones. Unlike previous work which could only verify the security of an implementation at a given order, maskComp is able to prove the security of an implementation at any order.

Coron [115] designed a tool called CheckMasks to automatically verify the security of higher-order masked implementations, similarly to maskverif [43]. For large implementations, CheckMasks can only verify the security at small orders. However, CheckMasks is also able to verify the t-SNI (or t-NI) property of gadgets at any order.

Chapter 7

Protecting Against Fault Attacks and Power-based Side-channels Attacks

Publication

This work was done in collaboration with Pantea Kiaei (Virginia Tech), Patrick Schaumont (Worcester Polytechnic Institute), Karine Heydemann (LIP6) and Pierre-Évariste Dagand (LIP6). It led to the following publication:

P. Kiaei, D. Mercadier, P. Dagand, K. Heydemann, and P. Schaumont. Custom instruction support for modular defense against side-channel and fault attacks. In *Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE, October 5-7, 2020*. Springer, 2020. URL `https://eprint.iacr.org/2020/466`

In Chapter 6, we showed how to generate DPA-resistant implementations with Usubac using masking countermeasures. In this chapter, we pursue a similar line of work, and present a backend of Usubac that generates implementations that are resilient to both DPA and fault attacks. This is achieved by exploiting the SKIVA architecture, a 32-bit CPU with custom instructions to enable masking and fault countermeasures (Section 7.2). We evaluate the throughput and fault resistance of an AES implementation on SKIVA in Section 7.3, thus showing that SKIVA allows to efficiently enable side-channel countermeasures.

7.1 Fault Attacks

Fault attacks [88, 35, 171, 39] consist in physically tampering with a cryptographic device in order to induce faults in its computations. The most common ways to do so consist in under-powering [139] or overpowering [25, 188], blasting ionizing light (*e.g.*, lasers) [280, 157, 271], inducing clock glitches [20, 28], using electromagnetic (EM) pulses [271, 252], or even heating up or cooling down [279], or using X-rays or ion beams [35].

There are several ways to exploit fault attacks. Algorithm-specific attacks aim at changing the behavior of a cipher at the algorithmic level by modifying either some of its data [108] or alter its control flow by skipping [270] or replacing [28] some instructions. Differential fault analysis (DFA) [76] uses differential cryptanalysis techniques to recover the secret key by comparing correct and faulty ciphertexts produced from the same plaintexts. Safe-error attacks (SEA) [308] simply observes whether a fault has an impact on the output of the cipher, and thus deduce the values of some secret variables. Statistical fault analysis (SFA) [141], unlike SEA and DFA does not require correct (*i.e.,* not faulted) ciphertexts: by injecting a fault at a chosen location and partially decrypting the ciphertext up to that location, SFA can be used to recover secret bits. Ineffective fault attacks (IFA) [109] are similar to SEA: the basic idea is to set an intermediate variable at 0 (using a so-called stuck-at-zero faults), and if the ciphertext is unaffected by this fault (the fault is ineffective), this means that the original value of this intermediate variable was already 0, thus revealing secret information. Finally, based on SFA and IFA, SIFA [127] uses ineffective faults to deduce a bias in the distribution of some secret data.

Various countermeasures have been proposed to prevent fault attacks. Hardware protections can either prevent or detect faults, using for instance active or passive shields [188], integrity checks [162] or other tampering detection mechanisms [5, 298, 106]. However, these techniques tend to be expensive and lack genericity: each countermeasure protects against a known set of attacks and the hardware might still be vulnerable to new attacks [35]. Software countermeasures, on the other hand, are less expensive and easier to adapt to new attacks. Software countermeasures can be either part of the design of the cryptographic algorithm—in the protocol [210] or the primitive [278]—or applied to implementations of existing ciphers, such as using redundant operations, error detection codes or consistency checks [145, 209, 173, 143].

One must be careful when implementing countermeasures against fault attack, since they can increase the vulnerability to other side-channel analysis [256, 257, 112]. For instance, Kiaei et al. [179] showed that using direct redundant bytes leaks more information than using complementary redundant bytes.

Only few works deal with protection against both side-channel analysis and fault injection. Cnudde and Nikova [110, 111] proposed a hardware-oriented approach where redundancy is applied on top of a masked implementation to obtain combined resistance against faults and SCA. ParTI [272] is a protection scheme that combines threshold implementation (TI) [225] –to defend against SCA– and concurrent error detection [204] –to thwart fault attacks. However, ParTI only targets hardware implementations, and the faults it is able to detect are limited in hamming weight. By comparison, CAPA [260] is based on secure multiparty computations protocols (MPC), which allows it to detect more faults than ParTI. Still, CAPA is hardware-oriented and expensive to adapt to software. Finally, Simon et al. [278] proposed the Frit permutation, which is secure against both faults and SCA by design, and efficient both in hardware and software. However, no solution exists to efficiently secure legacy ciphers at the software level.

7.2 SKIVA

SKIVA is a custom 32-bit processor developed by Pantea Kiaei (Virginia Tech) and Patrick Schaumont (Worcester Polytechnic Institute), in collaboration with Karine Heydemann (LIP6), Pierre-Evariste Dagand (LIP6) and myself. It enables a modular approach to countermeasure design, giving programmers the flexibility to protect their ciphers against timing-based and power-based side-channel analysis as well as fault injection at various levels of security.

Using the bitslicing model, SKIVA views its 32-bit processor as 32 parallel 1-bit processors (called *slices*). SKIVA then proposes an *aggregated slice* model, which consists in allocating multiple slices for each bit of data: aggregated slices can be shares of a masked design, redundant data of a fault-protected design, or a combination of both.

SKIVA relies on custom hardware instructions to efficiently and securely compute on aggregated slices (Section 7.2.1). Those new instructions are added to the SPARC V8 instruction set of the open-source LEON3 processor. A patched version of the Leon Bare-C

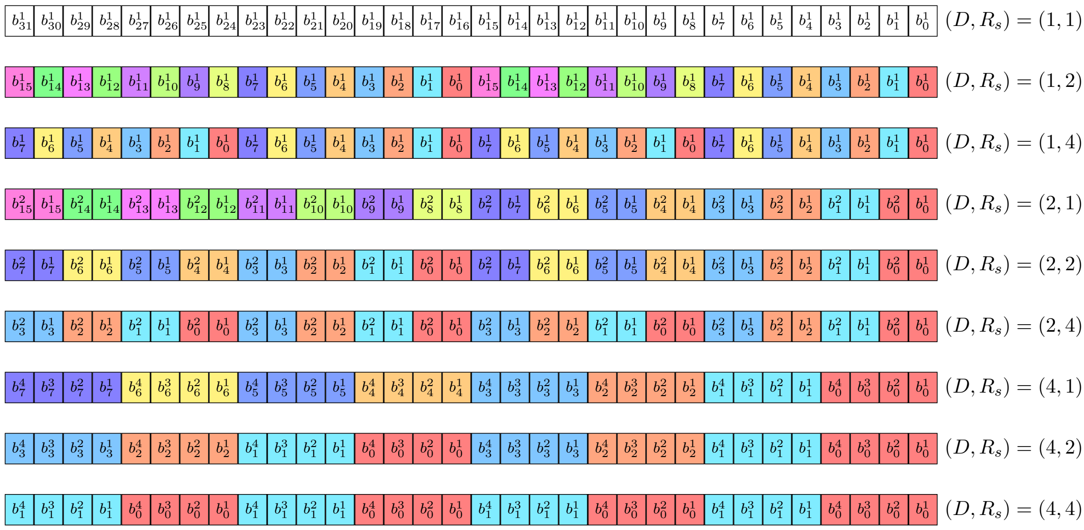


Figure 7.1: Bitslice aggregations on a 32-bit register, depending on (D,R_s).

Cross Compilation System's (BCC) toolchain makes those instructions available in assembly and in C (using inline assembly).

Usuba provides a SKIVA backend, thus freeing developers from the burden of writing low-level assembly code to target SKIVA. Furthermore, SKIVA offers 9 different levels of security through 9 combinations of countermeasures: a single Usuba program can be compiled to either. While Tornado is integrated at multiple stages of Usubac's pipeline (Chapter 6), SKIVA is only integrated as a backend and thus only affects code generation: instead of generating standard C code, Usuba can emit SKIVA-specific instructions.

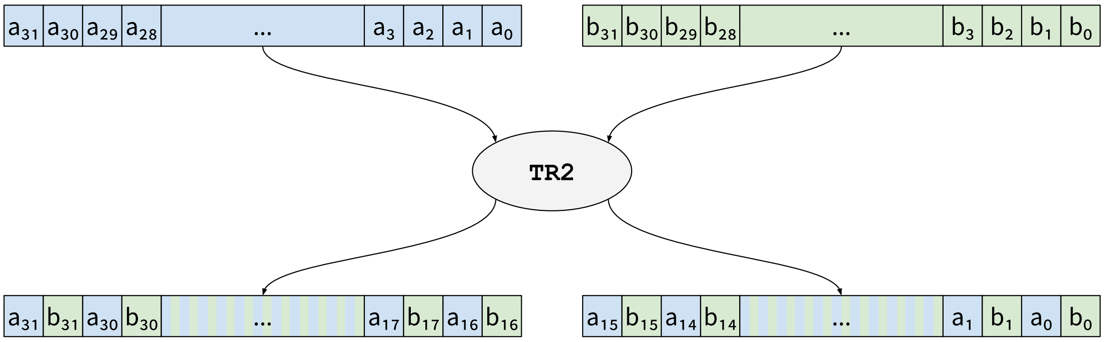
7.2.1 Modular Countermeasures

SKIVA supports four protection mechanisms that can be combined in a modular manner: bitslicing to protect against timing attacks, higher-order masking to protect against power side-channel leakage, intra-instruction redundancy to protect against data faults (faults on the dataflow) and temporal redundancy to protect against control faults (faults on the control flow). We use AES as an example, but the techniques are equally applicable to other bitsliced ciphers.

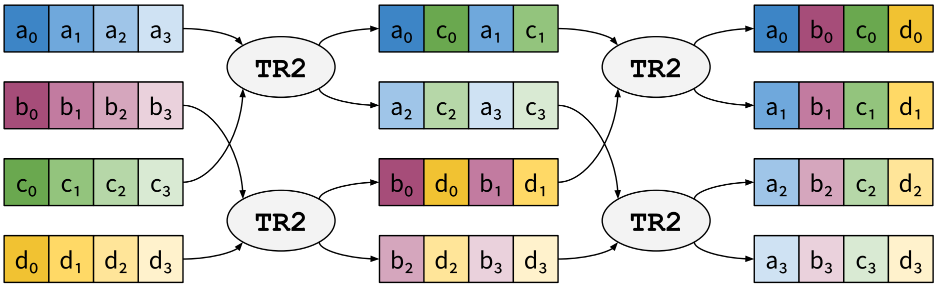
SKIVA requires ciphers to be fully bitsliced. For instance, the 128-bit input of AES is represented with 128 variables. Since each variable stores 32 bits on SKIVA, 32 instances of AES can be computed in a single run of the primitive. The key to compose countermeasures in SKIVA is to use some of those 32 instances to store redundant bits (to protect against fault attacks), or masked shares (to protect against power analysis).

Figure 7.1 shows the organization of the slices for masked and intra-instruction-redundant design. By convention, the letter D is used to denote the number of shares ($D \in \{1, 2, 4\}$) of a given implementation, and R_s to denote the spatial redundancy ($R_s \in \{1, 2, 4\}$).

SKIVA supports masking with 1, 2, and 4 shares leading to respectively unmasked, 1st-order, and 3rd-order masked implementations. Within a machine word, the D shares encoding the i^{th} bit are grouped together, as illustrated by the contiguously colored bits $b_i^{j\in[1,D]}$ in Figure 7.1. SKIVA also supports spatial redundancy by duplicating a single slice into two or four slices. Within a machine word, the R_s duplicates of the i^{th} bit are interspersed every $32/R_s$ bit, as illustrated by the alternation of colored words $b^j_{i\in[1,R_s]}$ in Figure 7.1.



(a) The `tr2` instruction



(b) 4×4 matrix transposition using `tr2`

Figure 7.2: SKIVA's `tr2` instruction

Instructions for Bitslicing

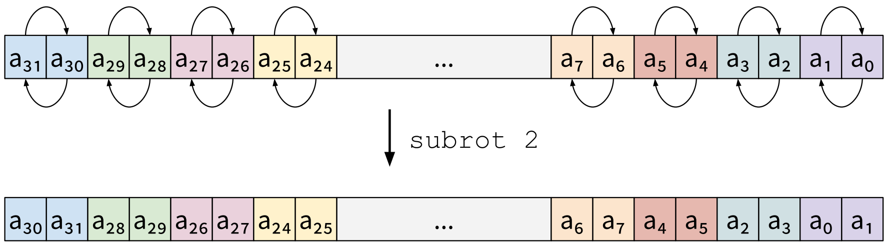
To efficiently transpose data into their bitsliced representation, SKIVA offers the instruction `tr2` (and `invtr2` to perform the inverse transformation), which interleaves two registers, as shown in Figure 7.2a. Transposing a $n \times n$ matrix can be done using $n \times 2^{n-1}$ `tr2` instructions. For instance, Figure 7.2b shows how to transpose a 4×4 matrix using 4 `tr2`. By comparison, transposing the same 4×4 matrix without the `tr2` instruction requires 32 operations (as shown in Section 1.2.2, Page 35): 8 `shifts`, 8 `ors` and 16 `ands`.

7.2.2 Higher-order Masked Computation

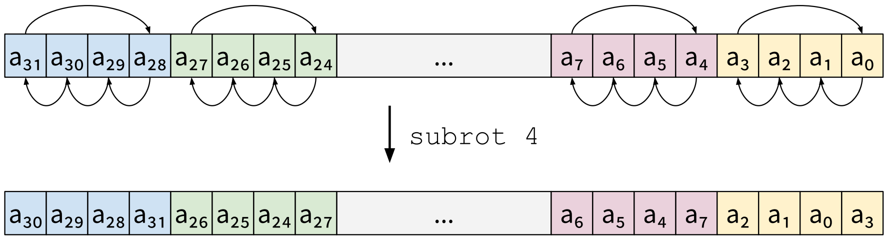
Designing masking schemes is orthogonal to SKIVA, which can be seen as a common platform to evaluate masking designs. SKIVA merely offers hardware support to write efficient software implementations of gadgets, as well as a way to combine masking with other countermeasures. Thus, any masking schemes can be used on SKIVA by just providing a software implementation of its gadgets.

To demonstrate SKIVA on AES, we used the NINA gadgets [125], which provides security against both side-channel attacks and faults using the properties of non-interference (NI) and non-accumulation (NA).

In our data representation, the D shares representing any given bit are stored in the same register, as opposed to being stored in D distinct registers. This choice allows a straightforward composition of masking and redundancy, and is consistent with some previous masked implementations [167].



(a) Two-shares slices rotations



(b) Four-shares slices rotations

Figure 7.3: SKIVA's `subrot` instructions

Instructions for Higher-Order Masking

Computing a masked multiplication between two shared values a and b requires computing their partial share-products. For instance, if a and b are represented by two shares (a_0,a_1) and (b_0,b_1), then the partial products $a_0 \cdot b_0$, $a_0 \cdot b_1$, $a_1 \cdot b_0$ and $a_1 \cdot b_1$ need to be computed. Since all the shares of a given value are stored in several slices of the same register, a single `and` computes n partial products at once (where n is the number of shares). The shares then need to be rotated in order to compute the other partial products. SKIVA offers the `subrot` instruction to perform this rotation on sub-words. Depending on an immediate parameter, `subrot` either rotates two-shares slices (Figure 7.3a) or four-shares slices (Figure 7.3b).

7.2.3 Data-redundant Computation

SKIVA uses intra-instruction redundancy (IIR) [240, 193, 107] to protect implementations against data faults. It supports either a direct redundant implementation, in which the duplicated slices contain the same value, or a complementary redundant implementation, in which the duplicated slices are complemented pairwise. For example, with $R_s = 4$, there can be four exact copies (direct redundancy) or two exact copies and two complementary copies (complementary redundancy).

In practice, complementary redundancy is favored over direct redundancy. First, it is less likely for complemented bits to flip to consistent values during a single fault injection. For instance, timing faults during state transition [311] or memory accesses [28] follow a random word corruption or a stuck-at-0 model. Second, complementary slices ensure a constant Hamming weight for a slice throughout the computation of a cipher. Furthermore, Kiaei et al. [179] showed that complementary redundancy results in reduced power leakage compared to direct redundancy.

Instructions for Fault Redundancy Checking and Generation

In order to generate redundant bytes to counter fault attacks, SKIVA provides the `red` instruction. This instruction can be used to generate either direct redundancy or complementary redundancy, and works for both two-shares and four-shares. An immediate passed as parameter controls which redundancy (direct, complementary, n shares) is generated. For instance, to generate two-shares complementary redundant values, the immediate `0b011` would be passed to `red` (Figure 7.4a), while to generate four-shares complementary redundant values, the immediate `0b101` would be used (Figure 7.4b).

The instruction `ftchk` is used to verify the consistency of redundant data. In its simplest form, it simply computes the `xnor` (a `xor` followed by a `not`) of the complementary redundant shares of its argument. If the result is anything but 0, then both half of the inputs are not (complementary) redundant, which means that a fault was injected. Figure 7.4c illustrates `ftchk` on a two-shares value.

In order to prevent ineffective fault attacks (IFA and SIFA), `ftchk` can perform majority-voting on four-shares redundant values. Figure 7.4d illustrates the behavior of `ftchk` in majority-voting mode on a four-shares complementary redundant value (where `majority` returns the most common of its input).

Instructions for Fault-Redundant Computations

Computations on direct-redundant data can be done using standard bitwise operations. However, for complementary redundant data, the bitwise operations have to be adjusted to complement operations. SKIVA thus offers 6 bitwise instructions to compute on complementary redundant values. `andc16` (resp., `xorc16` and `xnorc16`), illustrated in Figure 7.5a, performs `and` (resp., `xor` and `bxnor`) on the lower half of its two-shares redundant arguments, and a `nand` (resp., `xnor` and `xor`) on the upper half. Similarly, `andc8` (resp., `xorc8` and `xnorc8`), illustrated in Figure 7.5b, work in the same way on four-shares redundant values.

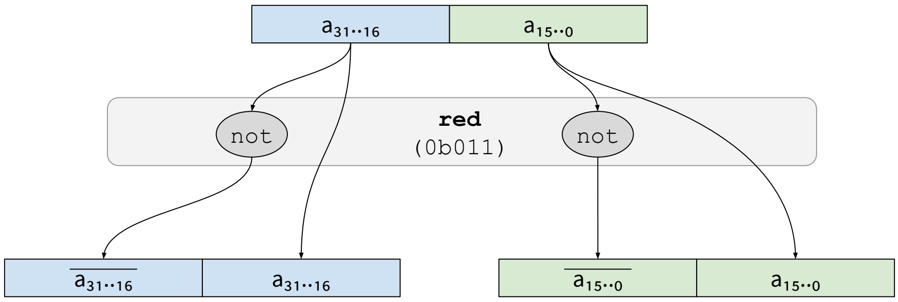
These operations can be simply written in C as follows, but would take 5 instructions each:

```
#define ANDC8(a,b)    ((((a) | (b)) & 0xFF00FF00) |
                       (((a) & (b)) & 0x00FF00FF))
#define XORC8(a,b)    ((~((a) ^ (b)) & 0xFF00FF00) |
                       (((a) ^ (b)) & 0x00FF00FF))
#define ANDC16(a,b)   ((((a) | (b)) & 0xFFFF0000) |
                       (((a) & (b)) & 0x0000FFFF))
#define XORC16(a,b)   ((~((a) ^ (b)) & 0xFFFF0000) |
                       (((a) ^ (b)) & 0x0000FFFF))
#define XNORC8(a,b)   ((((a) ^ (b)) & 0xFF00FF00) |
                       (~((a) ^ (b)) & 0x00FF00FF))
#define XNORC16(a,b)  ((((a) ^ (b)) & 0xFFFF0000) |
                       (~((a) ^ (b)) & 0x0000FFFF))
```

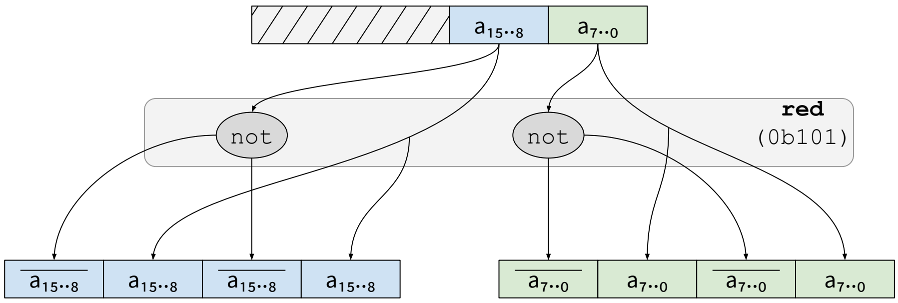
7.2.4 Time-redundant Computation

Data-redundant computation does not protect against control faults such as instruction skip. We protect our implementation against control faults using temporal redundancy (TR) across rounds [240]. We pipeline the execution of 2 consecutive rounds in 2 aggregated slices. By convention, we use the letter R_t to distinguish implementations with temporal redundancy ($R_t = 2$) from implementations without ($R_t = 1$). For $R_t = 2$, half of the slices compute round i while the other half compute round $i - 1$.

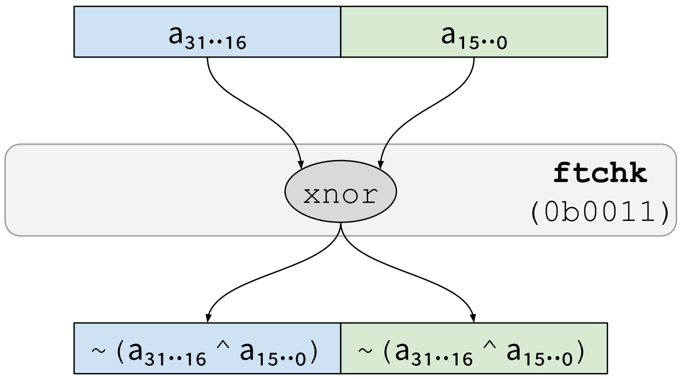
Figure 7.6 illustrates the principle of time-redundant bitslicing as applied to AES computation. The operation initializes the pipeline by filling half of the slices with the output of the first round of AES, and the other half with the output of the initial key whitening. At the end of round $i + 1$, we have recomputed the output of round i (at a later



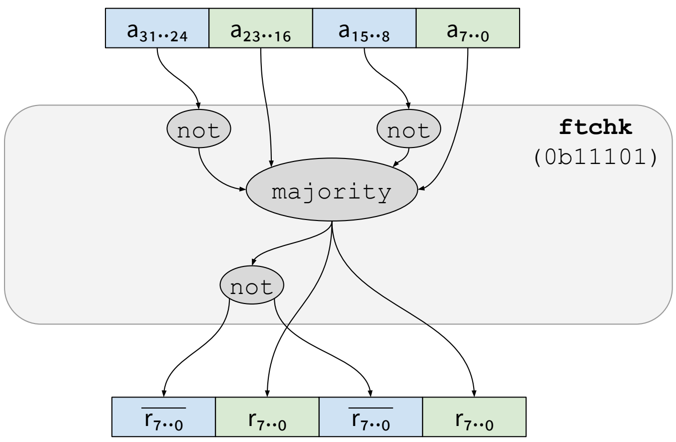
(a) Two-shares complementary redundancy generation



(b) Four-shares complementary redundancy generation

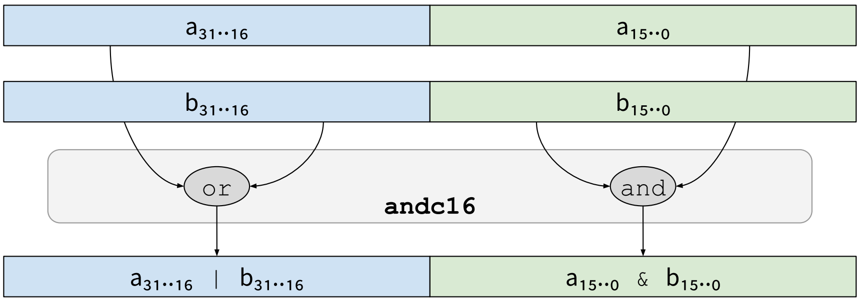


(c) Two-shares complementary redundancy fault checking

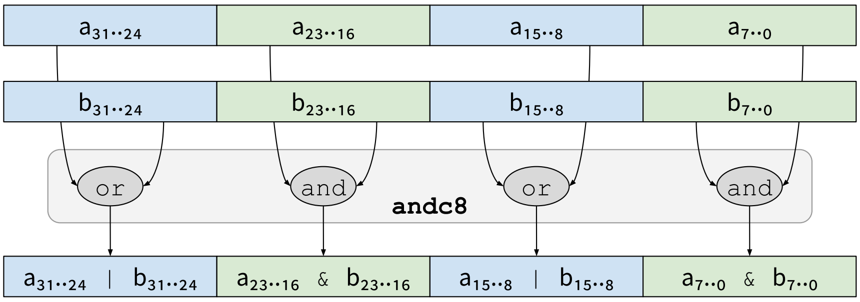


(d) Four-shares complementary redundancy majority voting

Figure 7.4: SKIVA's redundancy-related instructions



(a) Four-shares `andc16`



(b) Four-shares `andc8`

Figure 7.5: SKIVA's redundant `and` instructions

time): we can, therefore, compare the two results and detect control faults based on the different results they may have produced. In contrast to typical temporal-redundancy countermeasures such as instruction duplication [311], this technique does not increase code size: the same instructions compute both rounds at the same time. Only the last AES round, which is distinct from regular rounds, must be computed twice in a non-pipelined fashion.

Whereas pipelining protects the inner round function, faults remain possible on the control path of the loop itself. We protect against these threats through standard loop hardening techniques [161], namely redundant loop counters—packing multiple copies of a counter in a single machine word—and duplication of the loop control structure—producing multiple copies of conditional jumps so as to lower the odds of all of them being skipped through an injected fault.

7.3 Evaluation

We used Usubac to generate the 18 different implementations of AES (all combinations of $D \in \{1, 2, 4\}$, $R_s \in \{1, 2, 4\}$ and $R_t \in \{1, 2\}$) for SKIVA. We present a performance evaluation of these 18 implementations, as well as an experimental validation of the security of our control-flow fault countermeasures. For an analysis of the power leakage of these implementations, as well as a theoretical analysis of the security of SKIVA against data faults, we refer to Kiaei et al. [179].

7.3.1 Performance

Our experimental evaluation has been carried on a prototype of SKIVA deployed on the main FPGA (Cyclone IV EP4CE115) of an Altera DE2-115 board. The processor is clocked at 50MHz and has access to 128 kB of RAM. Our performance results are ob-

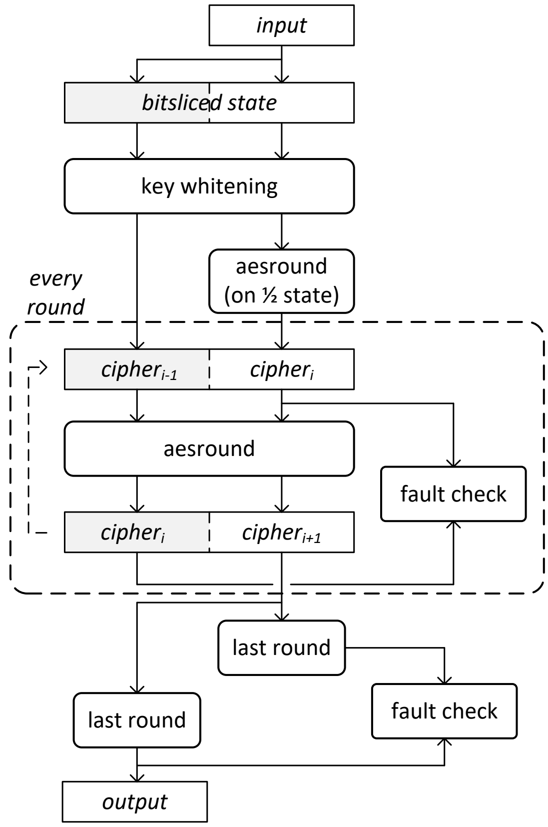


Figure 7.6: Time-Redundant computation of a bitsliced AES

tained by running the desired programs on bare metal. We assume that we have access to a TRNG that frequently fills a register with a fresh 32-bit random string. Several implementations of AES are available on our 32-bit, SPARC-derivative processor, with varying degrees of performance. The $v$sliced implementation (using only 8 variables to represent 128 bits of data) of BearSSL [246] performs at 48 C/B. Our bitsliced implementation (using 128 variables to represent 128 bits of data) performs favorably at 44 C/B while weighing 8060 bytes: despite significant register pressure (128 live variables for 32 machine registers) introducing spilling and slowing down performance, the rotations of `MixColumn` and the `ShiftRows` operations are compiled away, improving performance. This bitsliced implementation serves as our baseline in the following.

Throughput (AES)

We evaluate the throughput of our 18 variants of AES, for each value of $D \in \{1, 2, 4\}$, $R_s \in \{1, 2, 4\}$ and $R_t \in \{1, 2\}$. To remove the influence of the TRNG's throughput from this evaluation, we assume that its refill frequency is strictly higher than the rate at which our implementation consumes random bits. In practice, a refill rate of 10 cycles for 32 bits is enough to meet this requirement.

Table 7.1 shows the result of our benchmarks. For D and R_t fixed, the throughput decreases linearly with R_s. At fixed D, the variant (D, $R_s = 1$, $R_t = 2$) (temporal redundancy by a factor 2) exhibits similar throughput as (D, $R_s = 2$, $R_t = 1$) (spatial redundancy by a factor 2). However, these two implementations are not equivalent from a security standpoint. The former offers weaker security guarantees than the latter. Similarly, at fixed D and R_s, we may be tempted to run twice the implementation (D, R_s, $R_t = 1$) rather than running once the implementation (D, R_s, $R_t = 2$): once again, the security of the former is reduced compared to the latter since temporal redundancy ($R_t = 2$)

$R_t = 1$		D = 1	D = 2	D = 4
R_s	1	44 C/B	176 C/B	579 C/B
	2	89 C/B	413 C/B	1298 C/B
	4	169 C/B	819 C/B	2593 C/B

(a) Reciprocal throughput ($R_t = 1$)

$R_t = 2$		D = 1	D = 2	D = 4
R_s	1	131 C/B	465 C/B	1433 C/B
	2	269 C/B	1065 C/B	3170 C/B
	4	529 C/B	2122 C/B	6327 C/B

(b) Reciprocal throughput ($R_t = 2$)

Table 7.1: Exhaustive evaluation of the AES design space

	With impact		Without impact			
	Detected (1)	Not detected (2)	Detected (3)	Not detected (4)	Crash (5)	# of faults
$R_t = 1$	0.19%	92.34%	0.00%	4.31%	3.15%	12840
$R_t = 2$	78.19%	0.00%	5.22%	12.18%	4.40%	21160

Table 7.2: Experimental results of simulated instruction skips

couples the computation of 2 rounds within each instruction, whereas pure instruction redundancy ($R_t = 1$) does not.

Code Size (AES)

We measure the impact of the hardware and our software design on code size, using our bitsliced implementation of AES as a baseline. SKIVA provides us with native support for spatial, complementary redundancy (`andc`, `xorc` and `xnorc`). Performing these operations through software emulation would result in a $\times 1.3$ (for $D = 2$) to $\times 1.4$ (for $D = 4$) increase in code size. One must nonetheless bear in mind that the security provided by emulation is, *a priori*, not equivalent to the one provided by native support. The temporal redundancy ($R_t = 2$) mechanism comes at the expense of a small increase (less than $\times 1.06$) in code size, due to the loop hardening protections as well as the checks validating results across successive rounds. The higher-order masking comes at a reasonable expense in code size: going from 1 to 2 shares increases code size by $\times 1.5$ whereas going from 2 to 4 shares corresponds to a $\times 1.6$ increase. A fully protected implementation ($D = 4$, $R_s = 4$, $R_t = 2$) thus weighs 13148 bytes.

7.3.2 Experimental Evaluation of Temporal Redundancy

We simulated the impact of faults on our implementation of AES. We focus our attention exclusively on control faults (instruction skips) since we can analytically predict the outcome of data faults [179]. To this end, we implement a fault injection simulator using gdb running through the JTAG interface of the FPGA board. We execute our implementation up to a chosen breakpoint, after which we instruct the processor to skip the current instruction, hence simulating the effect of an instruction skip. In particular, we have exhaustively targeted every instruction of the first and last round as well as the pipelining routine (for $R_t = 2$). Since rounds 2 to 9 use the same code as the first round, the absence of vulnerabilities against instruction skips within the latter means that the former is secure against instruction skip as well. This exposes a total of 1248 injection points for $R_t = 2$ and 1093 injection points for $R_t = 1$. For each such injection point, we perform an instruction skip from 512 random combinations of keys and plaintexts for $R_t = 2$ and 352 random combinations for $R_t = 1$.

The results are summarized in Table 7.2. Injecting a fault had one of five effects. A fault may yield an incorrect ciphertext with (1) or without (2) being detected. A fault may yield a correct ciphertext, with (3) or without (4) being detected. Finally, a fault may cause the program or the board to crash (5). According to our attacker model, only outcome (2) witnesses a vulnerability. In every other outcome, the fault either does not produce a faulty ciphertext or is detected within two rounds. For $R_t = 2$, we verify that every instruction skip was either detected (outcome 1 or 3) or had no effect on the output of the corresponding round (outcome 4) or led to a crash (outcome 5). Comparatively, with $R_t = 1$, nearly 95% of the instruction skips led to an undetected fault impacting the ciphertext. In 0.19% of the cases, the fault actually impacts the fault-detection mechanism itself, thus triggering a false positive.

7.4 Conclusion

In this chapter, we showed how, by targeting SKIVA, Usubac is able to generate code that withstands power-based side-channel attacks as well as fault attacks.
We generated a bitsliced AES for SKIVA and evaluated its performance at various security level (in terms of redundancy and masking), showing that SKIVA's custom instructions for redundant and masked computation allow for relatively cheap countermeasures against side-channel attacks.

Chapter 8

Conclusion

Writing, optimizing and protecting cryptographic implementations by hand are tedious tasks, requiring knowledge in cryptography, CPU microarchitectures and side-channel attacks. The resulting programs tend to be hard to maintain due to their high complexity. To overcome these issues, we propose Usuba, a high-level domain-specific language to write symmetric cryptographic primitives. Usuba programs act as specifications, enabling high-level reasoning about ciphers.

The cornerstone of Usuba is a programming technique commonly used in cryptography called bitslicing, which is used to implement high-throughput and constant-time ciphers. Drawing inspiration from existing variations of bitslicing, we proposed a generalization of bitslicing that we dubbed *m*slicing, which overcomes some drawbacks of bitslicing (*e.g.,* high register pressure, lack of efficient arithmetic) to increase throughput even further.

Usuba allows developers to write sliced code without worrying about the actual parallelization: an Usuba program is a scalar description of a cipher, from which the Usubac compiler automatically produces vectorized code. Furthermore, Usuba provides polymorphic types and constructions that are then automatically specialized for a given slicing type (*e.g.,* bitslicing, *m*slicing), thus abstracting implementation details (*i.e.,* slicing) from the source code.

Usubac incorporates several domain-specific optimizations to increase throughput. Our bitslice scheduler reduces spilling in bitsliced code, which translates into speedups of up to 35%. Our *m*slice scheduler aims at maximizing instruction-level parallelism in *m*sliced code and increases throughput by up to 39%. Finally, Usubac can automatically interleave several independent instances of a cipher to maximize instruction-level parallelism when scheduling fails to do so, which offers speedups of up to 35% as well. Overall, on modern Intel CPUs, Usuba is 3 to 5% slower than hand-tuned assembly implementation of AES, but performs on par with hand-tuned assembly or C implementations of CHACHA20, SERPENT and GIMLI and outperforms manually optimized C implementations of ASCON, ACE, RECTANGLE and CLYDE.

Usubac can also automatically generate masking countermeasures against side-channel attacks. By integrating tightPROVE$^+$ in Usubac's pipeline, we are able to guarantee that the masked implementations generated by Usubac are provably secure against probing attacks. We developed several optimizations tailored for masked programs (constant propagation, loop fusion) to improve the performance of Usuba's masked implementations. Using this automated masking generation, we were able to compare masked implementations (at orders up to 127) of 11 ciphers from the recent NIST Lightweight Cryptography Competition.

We also wrote a backend for SKIVA, automatically securing ciphers against both power-based side-channel attacks and fault injections. We evaluated the throughput of Usuba's AES on the SKIVA platform, and empirically showed that this implementation is resilient to instruction skip attacks.

Some numbers. Usubac contains slightly over 10.000 lines of OCaml code. Its regression tests amount to 2.200 lines of C and 1.000 lines of Perl. The ciphers we implemented in Usuba amount to more than 2.500 lines of code. Finally, the benchmarks we mentioned throughout this thesis count more 2.300 lines of Perl scripts, which generates hundreds of millions of lines of C code.

8.1 Future Work

8.1.1 Fine-Grained Autotuning

Rather than considering nodes (resp., loops) one by one for inlining (resp., unrolling), Usubac's autotuner (Section 4.2.1, Page 96) evaluates the impact of inlining all nodes and unrolling all loops. While this may lead to suboptimal code, the space of all possible combinations of inlining, unrolling, scheduling and interleaving is often too large to be explored in reasonable time. This is a known issue that autotuners must overcome, for instance by using higher-level heuristics to guide the search [253, 49].

In Usuba, we could also improve the efficiency of the autotuner by combining its empirical evaluation with static analysis. For instance, a node with fewer than 2 instructions will always be more efficient inlined (to remove the overhead of calling a function). Similarly, instructions per cycle (IPC) can be statically estimated to guide optimizations: interleaving, for instance, will never improve the performance of a code whose IPC is 4.

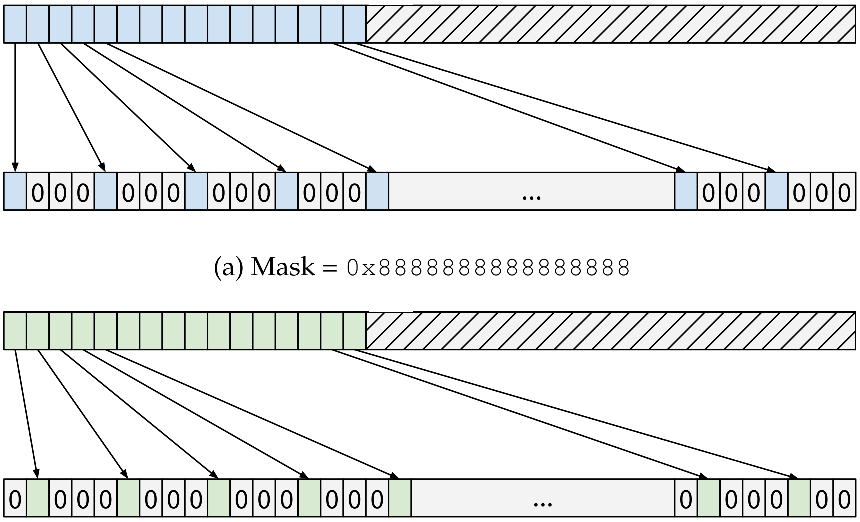
8.1.2 *m*slicing on General-Purpose Registers

In Section 5.3 (Page 132), we mentioned that the lack of x86-64 instruction to shift 4 16-bit words in a single 64-bit register prevents us from parallelizing RECTANGLE's *v*sliced implementation on general-purpose registers.

However, in practice, the `_pdep_u64` intrinsic can be used to interleave 4 independent 64-bit inputs of RECTANGLE in 4 64-bit registers. This instruction takes a register `a` and an integer `mask` as parameters, and dispatches the bits of `a` to a new register following the pattern specified in `mask`.

For instance, using `_pdep_u64` with the mask `0x8888888888888888` would dispatch a 16-bit input of RECTANGLE into a 64-bit register as shown in Figure 8.1a. A second input of RECTANGLE could be dispatched into the next bits using the mask `0x4444444444444444` (Figure 8.1b). Repeating the process with two more inputs using the masks `0x2222222222222222` and `0x1111111111111111`, and then combining the results (using a simple `or`) would produce a 64-bit register containing 4 interleaved 16-bit inputs of RECTANGLE. Since RECTANGLE's whole input is made of 64 bits, this process needs to be repeated 4 times. Then, regular `shift` and `rotate` instructions can be used to independently rotate each input. A left-rotation by 2 can now be done by a simple left-rotation by 8, as shown in Figure 8.2.

In essence, this technique relies on a data-layout inspired by *h*slicing (bits of the input are split along the horizontal direction of the registers), while offering access to a limited set of *v*slice instructions: bitwise instructions and shifts/rotates can be used, but arithmetic instructions cannot. This mode could be incorporated within Usuba in order to increase throughput on general-purpose registers. Gottschlag et al. [149] showed, for instance, that because of the CPU frequency reduction triggered by AVX and AVX512, a single process using AVX or AVX512 instructions can slow down other workloads running in parallel. Implementing this efficient *m*slicing on general-purpose register might therefore enable systemwide performance improvements.



(a) Mask = `0x8888888888888888`

(b) Mask = `0x4444444444444444`

Figure 8.1: Intel `_pdef_u64` instruction

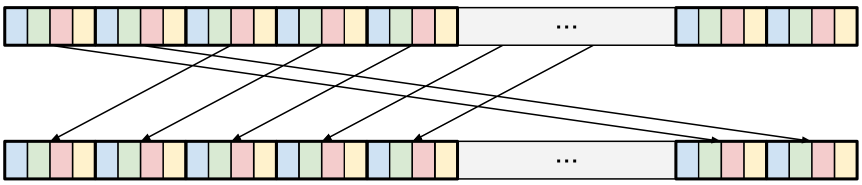


Figure 8.2: Left-rotation by 2 after packing data with `_pdef_u64`

8.1.3 Hybrid *m*slicing

Recall the main computing nodes of CHACHA20:

```
node QR (a:u32, b:u32, c:u32, d:u32)
     returns (aR:u32, bR:u32, cR:u32, dR:u32)
let
    a := a + b;
    d := (d ^ a) <<< 16;
    c := c + d;
    b := (b ^ c) <<< 12;
    aR = a + b;
    dR = (d ^ aR) <<< 8;
    cR = c + dR;
    bR = (b ^ cR) <<< 7;  tel

node DR (state:u32x16) returns (stateR:u32x16)
let
    state[0,4,8,12]  := QR(state[0,4,8,12]);
    state[1,5,9,13]  := QR(state[1,5,9,13]);
    state[2,6,10,14] := QR(state[2,6,10,14]);
    state[3,7,11,15] := QR(state[3,7,11,15]);

    stateR[0,5,10,15] = QR(state[0,5,10,15]);
    stateR[1,6,11,12] = QR(state[1,6,11,12]);
    stateR[2,7,8,13]  = QR(state[2,7,8,13]);
    stateR[3,4,9,14]  = QR(state[3,4,9,14]);  tel
```

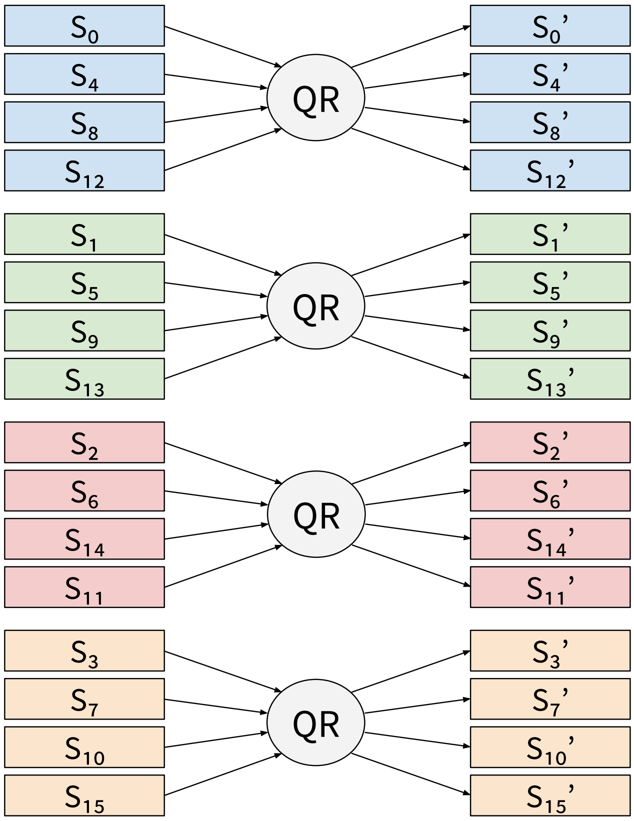


Figure 8.3: CHACHA20's round

Within the node DR, the node QR is called 4 times on independent inputs, and then 4 times again on independent inputs. Figure 8.3 provides a visual representation of the first 4 calls to DR.

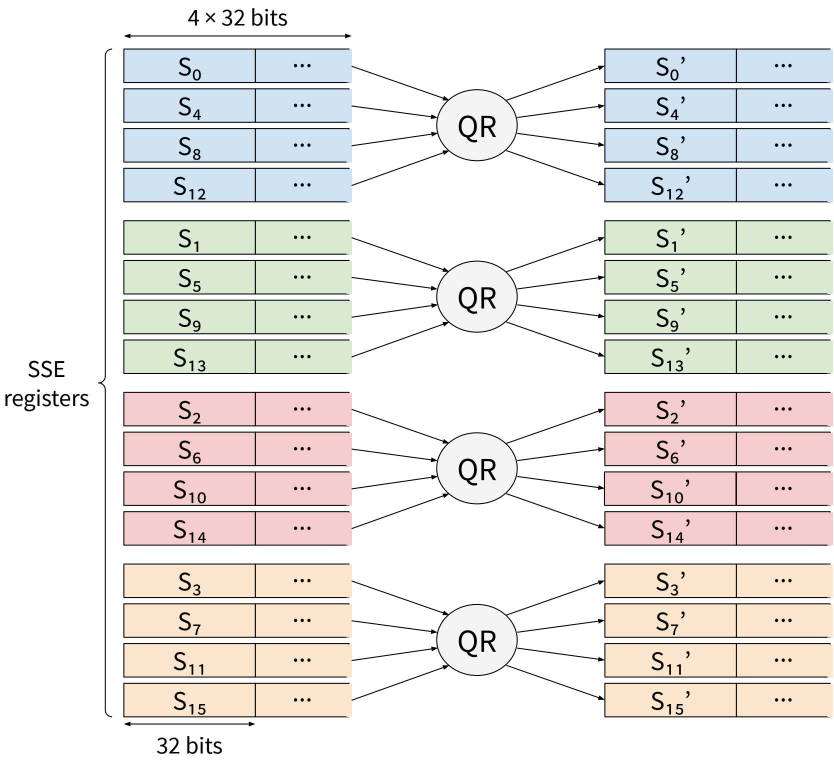
In *v*slicing, each 32-bit word of the state is mapped to a SIMD register, and these registers are filled with independent inputs. Figure 8.4a illustrates the first 4 calls to QR (first half of DR) using this *v*slice representation, and Figure 8.4b illustrate the last 4 calls to QR (second half of DR).

Using pure *v*slicing may be suboptimal because 16 registers are required, which leaves no register for temporary variables on SSE and AVX. In practice, at least one register of the state is spilled to memory.

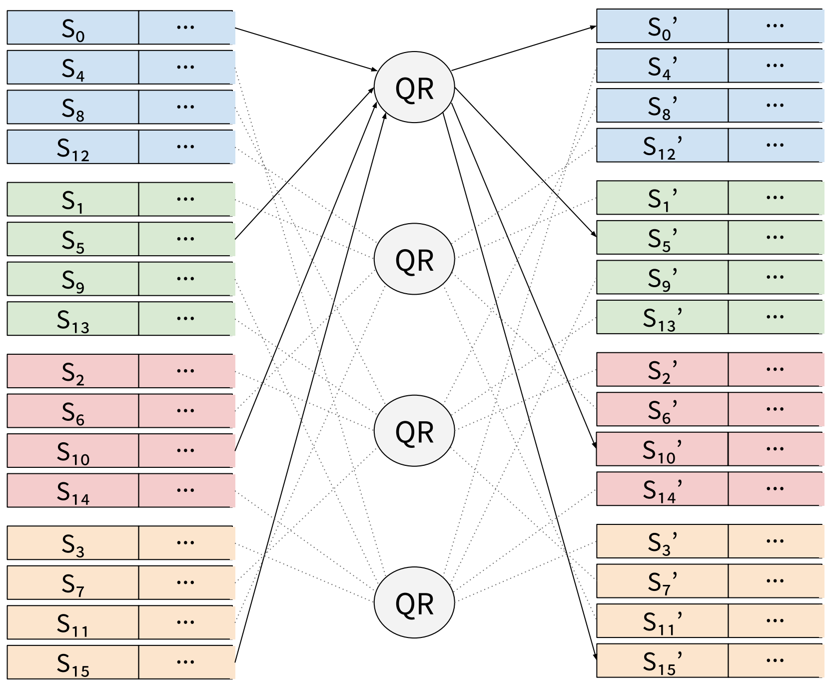
Rather than parallelizing CHACHA20 by filling SSE and AVX register with independent inputs, it is possible to do the parallelization of a single instance on SSE registers (respectively, two instances on AVX2). Since the first 4 calls to QR operate on independent values, we can pack the 16 32-bit words of the state within 4 SSE (or AVX) registers, and a single call to QR will compute it four times on a single input, as illustrated by Figure 8.5a.

Three rotations are then needed to reorganize the data for the final 4 calls to QR (Figure 8.5b). On SSE or AVX registers, those rotations would be done using a `shuffle` instruction. Finally, a single call to QR computes the last 4 calls, as illustrated in Figure 8.5c.

Compared to *v*slicing, this technique requires fewer independent inputs: on SSE (resp., AVX), it requires only 1 (resp., 2) independent inputs to reach the maximal throughput, while in *v*slicing, 4 (resp., 8) independent inputs are required. Also, the latency is divided by 4 compared to *v*slicing.

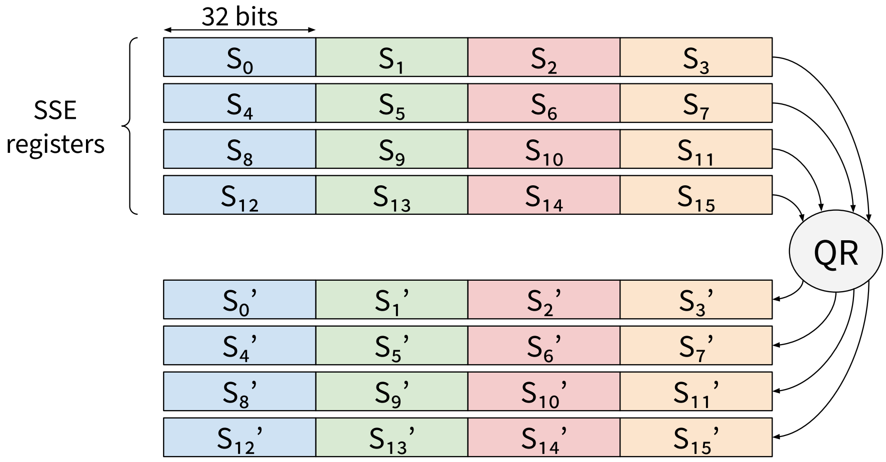


(a) First half

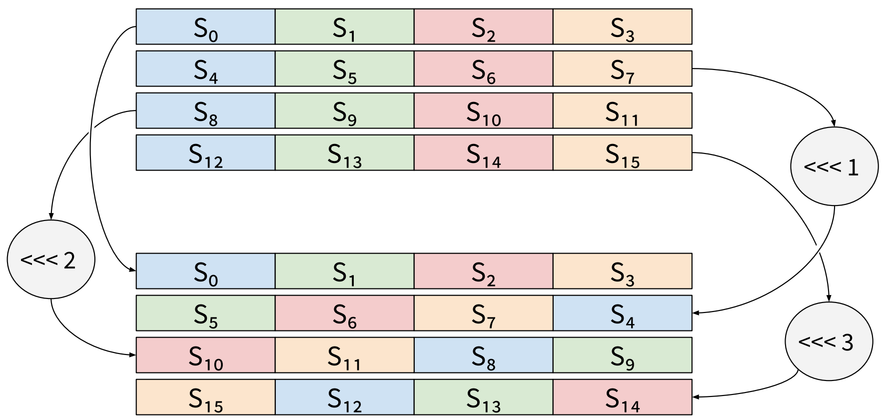


(b) Second half

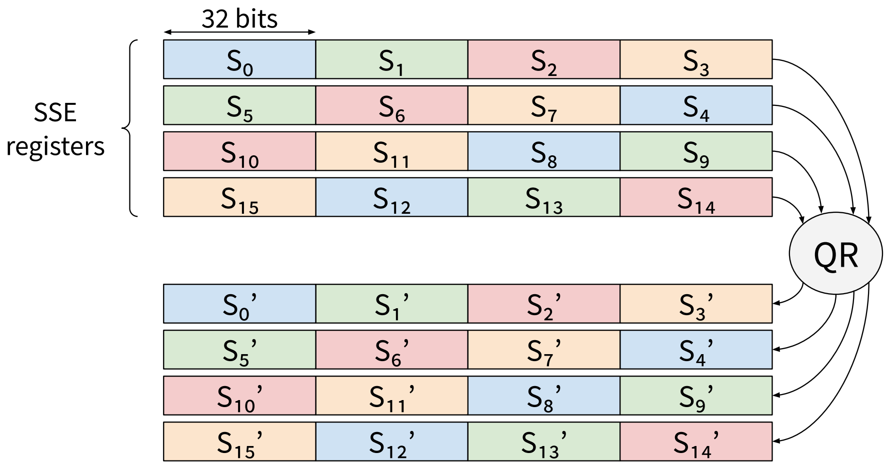
Figure 8.4: *v*slice CHACHA20's double round



(a) First half of `DR`



(b) Rotation between the first and second halves of `DR`



(c) Second half of `DR`

Figure 8.5: CHACHA20 in hybrid slicing mode

The downside is that it requires additional shuffles to reorganize the data within each round and at the end of each round, which incurs an overhead compared to *v*slicing. Furthermore, recall from Section 4.2.5 (Page 100) that CHACHA20's quarter round (`QR`) is bound by data dependencies. In *v*slicing, Usubac's scheduling algorithm is able to interleave 4 quarter rounds, thus removing any stalls related to data dependencies. This is not possible with this hybrid sliced form, since only two calls to `QR` remain, both of which cannot be computed simultaneously.

However, a lower register pressure allows such an implementation of CHACHA20 to be implemented without any spilling, which improves performance. Furthermore, because this implementation only uses 5 registers (4 for the state + 1 temporary), it can be interleaved 3 times without introducing any spilling. This interleaving would remove the data hazards from `QR`. As mentioned in Section 5.1 (Page 125), the fastest implementation of GIMLI on AVX2 uses this technique, and is faster than Usuba's *v*sliced implementation.

We can understand this technique as an intermediary stage between vertical and horizontal slicing. One way to incorporate it to Usuba would be to represent atomic types with a word size m and a vector size V ($\mathtt{u}m^V$), instead of a direction D and a word size m ($\mathtt{u}_D m$). The word size would correspond to the size of the packed elements within SIMD registers, and the vector size would represent how many elements of a given input are packed within the same register. For instance, CHACHA20's hybrid implementation would manipulate a state of 4 values of type $\mathtt{u}32^4$.

Using this representation, a *v*slice type $\mathtt{u}_V m$ corresponds to $\mathtt{u}m^1$, while a *h*slice type $\mathtt{u}_H m$ corresponds to $\mathtt{u}1^m$, and a bitslice type $\mathtt{u}_D 1$ would unsurprisingly correspond to $\mathtt{u}1^1$.

A type $\mathtt{u}m^V$ is valid only if the target architecture offers SIMD vectors able to contain V words of size m. Arithmetic instructions are possible between two $\mathtt{u}m^V$ values (provided that the architecture offers instruction to perform m-bit arithmetic) and shuffles are allowed if the architecture offers instructions to shuffle m-bit words.

8.1.4 Mode of Operation

One of the commonly used mode of operation is counter mode (CTR). In this mode (illustrated in Figure 1.7b, Page 30), a counter is encrypted by the primitive (rather than encrypting the plaintext directly), and the output of the primitive is `xor`ed with a block of plaintext to produce the ciphertext. The counter is incremented by one for each subsequent block of the plaintext to encrypt.

For 256 consecutive block to encrypt, only the last byte of the counter changes, and the others remain constant. Hongjun Wu and later Bernstein and Schwabe [66] observed that the last byte of AES's input only impacts 4 bytes during the first round, as illustrated by Figure 8.6. The results of the first round of `AddRoundKey`, `SubBytes` and `ShiftRows` on the first 15 bytes, as well as `MixColumns` on 12 bytes can thus be cached and reused for 256 encryptions. Concretely, the first round of AES only requires computing `AddRoundKey` and `SubBytes` on a single byte (`ShiftRows` does not require computation), and `MixColumns` on 4 bytes instead of 16. Similarly, 12 of the 16 `AddRoundKey` and `SubBytes` of the second round can be cached. Additionally, Park and Lee [239] showed that some values can be precomputed to speed up computation even further (while keeping the constant-time property of bitslicing). Only 4 bytes of the output of the first round depends on the last byte, which can take 256 values. Thus, it is possible to precompute all 256 possible values for these 4 bytes (which can be stored a $4 \times 256 = 1$KB table, and reuse them until incrementing the counter past the 6th least significant byte of the counter (b_{10} on the figure above, which is an input of the same MixColumn as b_{15}), that is, once every 1 trillion block.

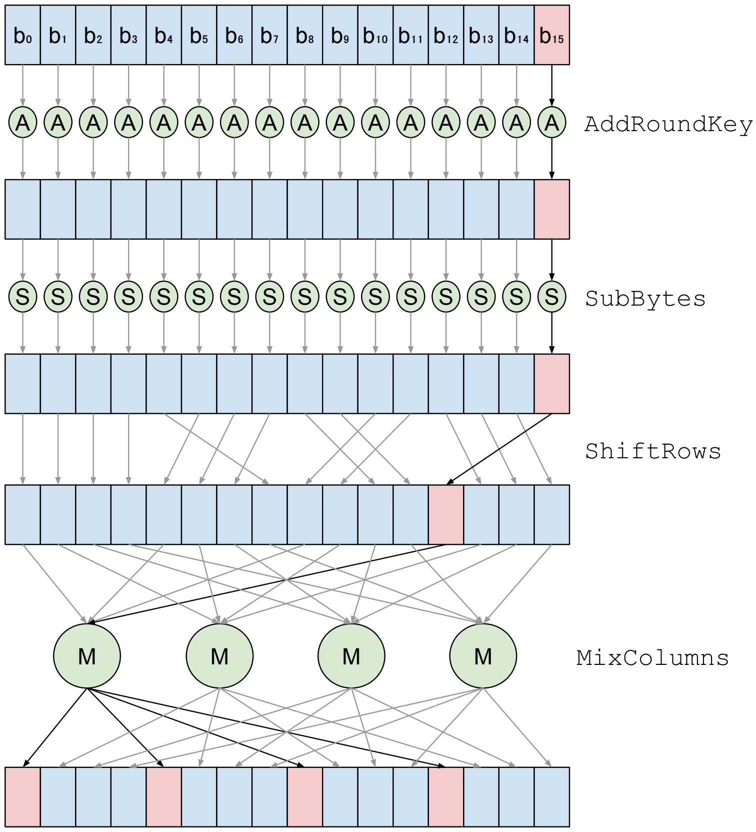


Figure 8.6: AES's first round in CTR mode

To integrate such optimizations in Usuba, we will have to broaden its scope to support modes of operation: Usuba currently does not offer support for stateful computations and forces loops to have static bounds. We would then need to extend the optimizer to exploit the specificities of the Usuba programming model.

8.1.5 Systematic Evaluation on Diverse Vectorized Architectures

Publication

This preliminary work was done in collaboration with Pierre-Évariste Dagand (LIP6), Lionel Lacassagne (LIP6) and Gilles Muller (LIP6). It led to the following publication:

D. Mercadier, P. Dagand, L. Lacassagne, and G. Muller. Usuba: Optimizing & trustworthy bitslicing compiler. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 4:1–4:8, 2018. doi: 10.1145/3178433.3178437

We focused our evaluation of vectorization to Intel architectures, because of their wide availability. Other architectures, such as AltiVec on PowerPC and Neon on ARM, offer similar instructions as SSE and AVX. In particular, both AltiVec and Neon provide 128-bit registers supporting 8/16/32/64-bit arithmetic and shift instructions (used in *v*slicing), shuffles (used in *h*slicing) and 128-bit bitwise instructions (used in all slic-

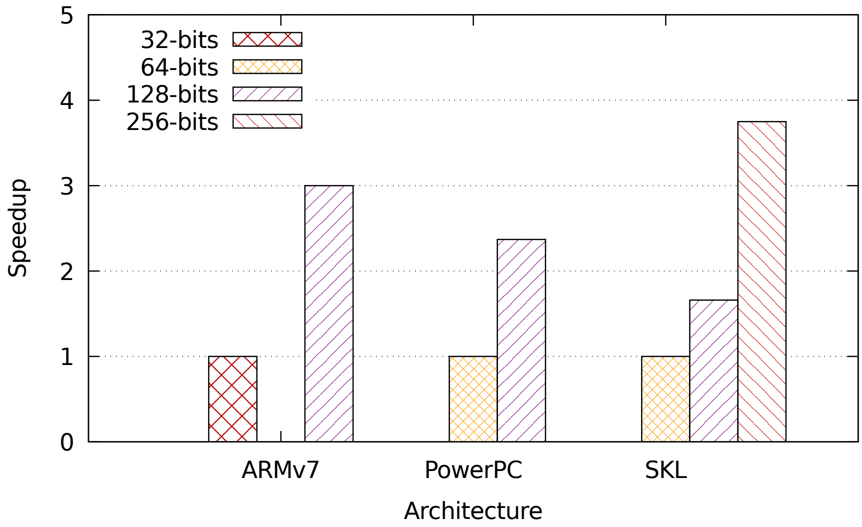


Figure 8.7: Bitsliced DES scaling on Neon, AltiVec and SSE/AVX2/AVX512

ing types). As a proof-of-concept, we thus implemented AltiVec and Neon backends in Usubac.

Figure 8.7 reports the speedup offered by vector extensions on a bitsliced DES (including the transposition) on several architectures: SKL is our Intel Xeon W-2155 (with 64-bit general-purpose registers, 128-bit SSE, 256-bit AVX2, 512-bit AVX512), PowerPC is a PPC 970MP (with 64-bit general-purpose registers and 128-bit AltiVec SIMD), and ARMv7 is an ARMv7 Raspberry Pi3 (with 32-bit general-purpose registers and 128-bit Neon SIMD). Throughputs have been normalized on each architecture to evaluate the speedups of vector extensions rather than the raw throughputs.

The speedups offered by SIMD extensions vary from one architecture to the other. PowerPC's AltiVec offer 32 registers and 3-operand instructions, and thus expectedly perform better than Intel's SSE comparatively to a 64-bit baseline. ARM's Neon extensions, on the other hand, only offer 16 registers, resulting in a similar speedup as Intel's AVX2 (note that ARMv7 has 32-bit registers whereas Skylake has 64-bit registers).

This benchmark only deals with bitslicing, and thus does not exploit arithmetic instructions, shuffles, or other architecture-specific SIMD instructions.

8.1.6 Verification

Bugs in implementations of cryptographic primitives can have devastating consequences [94, 152, 84, 56]. To alleviate the risk of errors, several recent projects aim at generating cryptographic code while ensuring the functional correctness of the executable code [310, 18, 87].

Figure 8.8 illustrates how end-to-end functional correctness could be added to the Usubac compiler. We propose to prove the correctness of the frontend by formally proving that each normalization passes preserves the semantics of the input program. Several works already showed how to prove the correctness of normalization of dataflow languages [89, 24, 23].

For the backend (optimizations), translation validation [244] would certainly be more practical, in the sense that it does not require proving that each optimization preserves semantics. The principle of translation validation is to check whether the program obtained after optimization has the same behavior as the program before optimization. This does not guarantee that the compiler is correct for all possible input programs, but it is

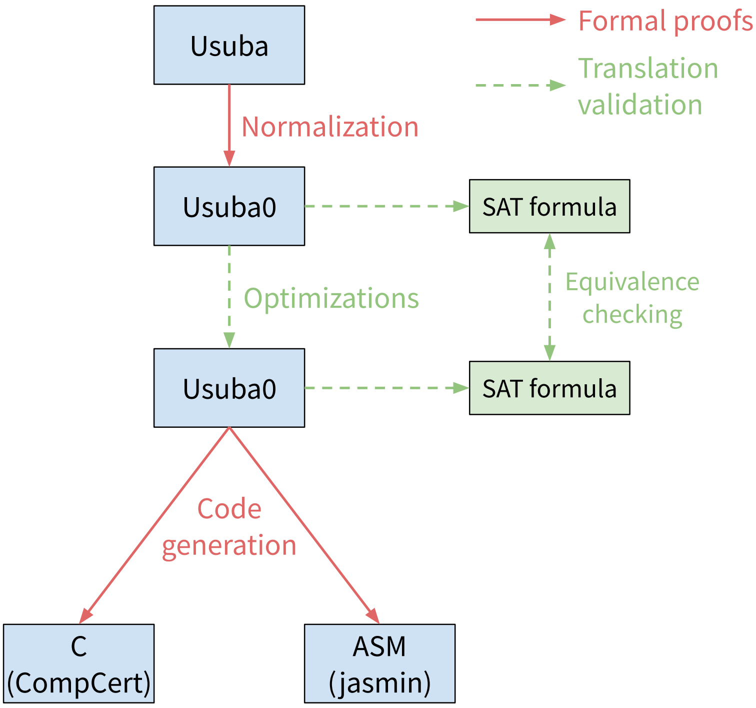


Figure 8.8: Pipeline of a semantics-preserving Usuba compiler

sufficient to guarantee that the compilation of a given program is correct. Translation validation is a tried-and-trusted approach to develop certified optimizations [292].

We implemented a proof-of-concept translation validation for the optimizations of the Usubac compiler. The principle is simple: we extract two SAT formulas from the pipeline, one before the optimizations, and one after. We then feed them to the Boolector SAT solver [224], which checks whether they are equivalent (or, more precisely, whether an input exists such that feeding it to both formulas produces two distinct outputs). Extraction of a SAT formula from an Usuba0 program is straightforward thanks to our dataflow semantics: an Usuba0 program is simply a set of equations.

We used this approach to verify the correctness of our optimizations for RECTANGLE (bitsliced and *v*sliced), DES (bitsliced), AES (bitsliced), CHACHA20 (*v*sliced) and SERPENT (*v*sliced). In all cases, Boolector was able to check the equivalence of the programs pre and post-optimizations in less than a second. This approach is thus practical (*i.e.*, it can verify equivalence of Usuba0 programs in a reasonable amount of time) and requires little to no investment to be implemented.

For additional confidence in the translation validation approach, a certified SAT solver, such as SMTCoq [22], could be used to make sure that the SAT encoding faithfully captures the Usuba semantics.

Finally, translation from Usuba0 to imperative code can be formally verified using existing techniques of the dataflow community [89]. We could either target CompCert [82, 199]. Or, if we decide to generate assembly code ourselves (to improve performance, as motivated in Section 6.3.3, Page 146), we can generate Jasmin assembly [18] and benefit from its mechanized semantics. In both cases (CompCert and Jasmin), further work is required to support SIMD instruction sets.

8.1.7 Targeting GPUs

A lot of implementations of cryptographic algorithms on graphics processing units (GPU) have been proposed in the last 15 years [205, 287, 236, 159, 113], including some implementation of bitsliced DES [307, 228, 11] and bitsliced AES [203, 226, 158, 241, 307]. Throughputs of more than 1 Tbits per seconds are reported for AES by Hajihassani et al. [158]. The applications of such high-throughput implementations include, among others, password cracking [282, 26], random number generation [218, 198] or disk encryption [10, 285].

Both bitsliced [158, 241, 307] and *m*sliced [203, 226] implementations of AES have been demonstrated on GPU. Additionally, Nishikawa et al. [226] showed that GPUs offer a large design space: a computation is broken down into many threads, registers are a shared resource between threads, a limited amount of thread can be executed simultaneously, *etc.* Nishikawa et al. [226] thus proposed an evaluation of the impact of some of those parameters on an *m*sliced AES. Usuba could provide an opportunity to systematically explore this design space across a wide range of ciphers.

Bibliography

[1] libsodium. URL https://github.com/jedisct1/libsodium.

[2] Linux kernel crypto api. URL https://github.com/torvalds/linux/tree/master/crypto.

[3] OpenSSL, cryptography and SSL/TLS tooklit. URL https://github.com/openssl.

[4] Perf wiki. URL https://perf.wiki.kernel.org/index.php/Main_Page.

[5] *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, 1997. URL https://web.archive.org/web/20121105102002/http://www-03.ibm.com/security/cryptocards/pdfs/bs330.pdf.

[6] IEEE standard VHDL language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 26 2009. doi: 10.1109/IEEESTD.2009.4772740.

[7] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, and R. Rohit. Ace: An authenticated encryption and hash algorithm. *Submission to NIST-LWC*, 2019. URL https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ace-spec.pdf.

[8] A. Adomnicai, Z. Najm, and T. Peyrin. Fixslicing: A new gift representation. Cryptology ePrint Archive, Report 2020/412, 2020. https://eprint.iacr.org/2020/412.

[9] G. Agosta and G. Pelosi. A domain specific language for cryptography. In *Forum on specification and Design Languages, FDL 2007, September 18-20, 2007, Barcelona, Spain, Proceedings*, pages 159–164. ECSI, 2007. URL http://www.ecsi-association.org/ecsi/main.asp?l1=library&fn=def&id=251.

[10] G. Agosta, A. Barenghi, F. D. Santis, A. D. Biagio, and G. Pelosi. Fast disk encryption through GPGPU acceleration. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2009, Higashi Hiroshima, Japan, 8-11 December 2009*, pages 102–109. IEEE Computer Society, 2009. ISBN 978-0-7695-3914-0. doi: 10.1109/PDCAT.2009.72. URL https://doi.org/10.1109/PDCAT.2009.72.

[11] G. Agosta, A. Barenghi, F. D. Santis, and G. Pelosi. Record setting software implementation of DES using CUDA. In S. Latifi, editor, *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010*, pages 748–755. IEEE Computer Society, 2010. ISBN 978-0-7695-3984-3. doi: 10.1109/ITNG.2010.43. URL https://doi.org/10.1109/ITNG.2010.43.

[12] G. Agosta, A. Barenghi, and G. Pelosi. A code morphing methodology to automate power analysis countermeasures. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 77–82. ACM, 2012. doi: 10.1145/2228360.2228376.

[13] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 81:1–81:6. ACM, 2013. doi: 10.1145/2463209.2488833.

[14] M. Akkar and C. Giraud. An implementation of DES and aes, secure against some attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, number Generators, pages 309–318, 2001. doi: 10.1007/3-540-44709-1_26.

[15] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013. doi: 10.1109/SP.2013.42.

[16] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, 2013. doi: 10.1016/j.scico.2011.10.008.

[17] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 53–70, 2016. URL `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida`.

[18] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823, 2017. doi: 10.1145/3133956.3134078.

[19] F. Amiel, K. Villegas, B. Feix, and L. Marcel. Passive and active combined attacks: Combining fault attacks and side channel analysis. In L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 92–102. IEEE Computer Society, 2007. doi: 10.1109/FDTC.2007.4318989.

[20] R. J. Anderson and M. G. Kuhn. Low cost attacks on tamper resistant devices. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997. doi: 10.1007/BFb0028165.

[21] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, Waterloo, Ontario, Canada, August 14-15, 2000, Proceedings*, pages 39–56, 2000. doi: 10.1007/3-540-44983-3_4.

[22] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, pages 135–150, 2011. doi: 10.1007/978-3-642-25379-9_12.

[23] C. Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, 2013. URL `https://tel.archives-ouvertes.fr/tel-00818169/file/VD2_AUGER_CEDRIC_07022013.pdf`.

[24] C. Auger, J. Colaço, G. Hamon, and M. Pouzet. A formalization and proof of a modular lustre compiler. In *accompaning paper of LCTES'08*, 2010. URL `https://pdfs.semanticscholar.org/86e0/df88edcd0eda0bf38ccd41e537cb7154173a.pdf`.

[25] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J. Seifert. Fault attacks on RSA with CRT: concrete results and practical countermeasures. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2002. doi: 10.1007/3-540-36400-5_20.

[26] M. Bakker and R. van der Jagt. GPU-based password cracking. Master's thesis, University of Amsterdam, 2010. `https://delaat.net/rp/2009-2010/p34/report.pdf`.

[27] G. Balakrishnan and T. W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010. doi: 10.1145/1749608.1749612.

[28] J. Balasch, B. Gierlichs, and I. Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In L. Breveglieri, S. Guilley, I. Koren, D. Naccache, and J. Takahashi, editors, *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 105–114. IEEE Computer Society, 2011. doi: 10.1109/FDTC.2011.9.

[29] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 321–345, 2017. doi: 10.1007/978-3-319-66787-4_16.

[30] S. Banik, A. Bogdanov, T. Peyrin, Y. Sasaki, S. M. Sim, E. Tischhauser, and Y. Todo. SUNDAE-GIFT. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/SUNDAE-GIFT-spec-round2.pdf`.

[31] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT-COFB. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gift-cofb-spec-round2.pdf`.

[32] F. Bao, R. H. Deng, Y. Han, A. B. Jeng, A. D. Narasimhalu, and T. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, volume 1361 of *Lecture Notes in Computer Science*, pages 115–124. Springer, 1997. doi: 10.1007/BFb0028164.

[33] Z. Bao, P. Luo, and D. Lin. Bitsliced implementations of the prince, LED and RECTANGLE block ciphers on AVR 8-bit microcontrollers. In *Information and Communications Security - 17th International Conference, ICICS 2015, Beijing, China, December 9-11, 2015, Revised Selected Papers*, pages 18–36, 2015. doi: 10.1007/978-3-319-29814-6_3.

[34] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. PHOTON-beetle authenticated encryption and hash family. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/PHOTON-Beetle-spec.pdf`.

[35] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006. doi: 10.1109/JPROC.2005.862424.

[36] M. Barbosa, R. Noad, D. Page, and N. P. Smart. First steps toward a cryptography-aware language and compiler. *IACR Cryptol. ePrint Arch.*, 2005:160, 2005. URL `http://eprint.iacr.org/2005/160`.

[37] M. Barbosa, A. Moss, D. Page, N. F. Rodrigues, and P. F. Silva. Type checking cryptography implementations. In *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*, pages 316–334, 2011. doi: 10.1007/978-3-642-29320-7_21.

[38] M. Barbosa, D. Castro, and P. F. Silva. Compiling CAO: from cryptographic specifications to C implementations. In M. Abadi and S. Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 240–244. Springer, 2014. doi: 10.1007/978-3-642-54792-8_13.

[39] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012. doi: 10.1109/JPROC.2012.2188769.

[40] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, 2006. doi: 10.1016/j.entcs.2005.10.031.

[41] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101. ACM, 2009. doi: 10.1145/1480881.1480894.

[42] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1267–1279. ACM, 2014. doi: 10.1145/2660267.2660283.

[43] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015. doi: 10.1007/978-3-662-46800-5_18.

[44] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016. doi: 10.1145/2976749.2978427.

[45] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020. doi: 10.1145/3371075.

[46] A. Battistello, J. Coron, E. Prouff, and R. Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 23–39, 2016. doi: 10.1007/978-3-662-53140-2_2.

[47] A. G. Bayrak, F. Regazzoni, P. Brisk, F. Standaert, and P. Ienne. A first step towards automatic application of power analysis countermeasures. In L. Stok, N. D. Dutt, and S. Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 230–235. ACM, 2011. doi: 10.1145/2024724.2024778.

[48] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In G. Bertoni and J. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013. doi: 10.1007/978-3-642-40349-1_17.

[49] U. Beaugnon. *Efficient Code Generation for Hardware Accelerators by Refining Partially Specified Implementations. (Génération de code efficace pour accélérateurs matériels par raffinement d'implémentations partiellement spécifiées)*. PhD thesis, École Normale Supérieure, Paris, France, 2019. URL `https://tel.archives-ouvertes.fr/tel-02385303`.

[50] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 123–153, 2016. doi: 10.1007/978-3-662-53008-5_5.

[51] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. SKINNY-AEAD and SKINNY-Hash. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/SKINNY-spec.pdf`.

[52] S. Belaïd, D. Goudarzi, and M. Rivain. Tight private circuits: Achieving probing security with the least refreshing. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, pages 343–372, 2018. doi: 10.1007/978-3-030-03329-3_12.

[53] S. Belaïd, P. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, pages 311–341, 2020. doi: 10.1007/978-3-030-45727-3_11.

[54] M. Bellare and T. Kohno. A theoretical treatment of related-key attacks: Rka-prps, rka-prfs, and applications. In E. Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2003. doi: 10.1007/3-540-39200-9_31. URL `https://doi.org/10.1007/3-540-39200-9_31`.

[55] D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F. Standaert, B. Udvarhelyi, and F. Wiemer. Spook: Sponge-based leakage-resilient authenticatedencryption with a masked tweakable block cipher. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Spook-spec.pdf`.

[56] D. Benjamin. poly1305-x86.pl produces incorrect output, 2016. URL `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006293.html`.

[57] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In D. S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 241–255. ACM, 1991. doi: 10.1145/113445.113466.

[58] D. J. Bernstein. Cache-timing attacks on aes, 2005. URL `http://cr.yp.to/papers.html#cachetiming`.

[59] D. J. Bernstein. Curve25519: New diffie-hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. doi: 10.1007/11745853_14.

[60] D. J. Bernstein. `qhasm` software package, 2007. URL `https://cr.yp.to/qhasm.html`.

[61] D. J. Bernstein. Chacha, a variant of salsa20. In *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*, 2008. URL `https://cr.yp.to/papers.html#chacha`.

[62] D. J. Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008. URL `https://cr.yp.to/snuffle/salsafamily-20071225.pdf`.

[63] D. J. Bernstein. Caesar: Competition for authenticated encryption: Security, applicability, and robustness, 2014.

[64] D. J. Bernstein. The death of optimizing compilers, 2015. URL `https://cr.yp.to/talks/2015.04.16/slides-djb-20150416-a4.pdf`.

[65] D. J. Bernstein and T. L. (editors). Supercop: System for unified performance evaluation related to cryptographic operations and primitives, 2018. URL `https://bench.cr.yp.to/supercop.html`.

[66] D. J. Bernstein and P. Schwabe. New AES software speed records. In *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings*, pages 322–336, 2008. doi: 10.1007/978-3-540-89754-5_25.

[67] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F. Standaert, Y. Todo, and B. Viguier. Gimli : A cross-platform permutation. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 299–320, 2017. doi: 10.1007/978-3-319-66787-4_15.

[68] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier. Gimli. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gimli-spec-round2.pdf`.

[69] T. Beyne, Y. L. Chen, C. Dobraunig, and B. Mennink. Elephant v1. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/elephant-spec.pdf`.

[70] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin. Proving the TLS handshake secure (as it is). In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 235–255. Springer, 2014. doi: 10.1007/978-3-662-44381-1_14.

[71] D. Biernacki, J. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, pages 121–130, 2008. doi: 10.1145/1375657.1375674.

[72] E. Biham. A fast new DES implementation in software. In *FSE*, 1997. doi: 10.1007/BFb0052352.

[73] E. Biham and A. Shamir. Differential cryptanalysis of des-like cryptosystems. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990. doi: 10.1007/3-540-38424-3_1.

[74] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993. ISBN 978-1-4613-9316-0. doi: 10.1007/978-1-4613-9314-6.

[75] E. Biham and A. Shamir. A new cryptanalytic attack on des. *preprint*, 18:10–96, 1996. URL `https://cryptome.org/jya/dfa.htm`.

[76] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. K. Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. doi: 10.1007/BFb0052259.

[77] E. Biham and A. Shamir. Power analysis of the key scheduling of the aes candidates. In *Proceedings of the second AES Candidate Conference*, pages 115–121, 1999. URL `http://www.cs.technion.ac.il/~biham/Reports/aes-power.ps.gz`.

[78] E. Biham, R. J. Anderson, and L. R. Knudsen. Serpent: A new block cipher proposal. In *Fast Software Encryption, 5th International Workshop, FSE '98, Paris, France, March 23-25, 1998, Proceedings*, pages 222–238, 1998. doi: 10.1007/3-540-69710-1_15.

[79] A. Biryukov and D. Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In M. Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009. doi: 10.1007/978-3-642-10366-7_1.

[80] G. R. Blakley. Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318. IEEE, 1979.

[81] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009. doi: 10.1007/s10817-009-9148-3.

[82] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 460–475, 2006. doi: 10.1007/11813040_31.

[83] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, pages 69–83, 2004. doi: 10.1007/978-3-540-30564-4_5.

[84] H. Boeck. Wrong results with poly1305 functions, 2016. URL `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413.html`.

[85] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 450–466, 2007. doi: 10.1007/978-3-540-74735-2_31.

[86] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. spongent: A lightweight hash function. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 312–325, 2011. doi: 10.1007/978-3-642-23951-9_21.

[87] B. Bond, C. Hawblitzel, M. Kapritsos, R. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 917–934, 2017. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond`.

[88] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. doi: 10.1007/3-540-69053-0_4.

[89] T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 586–601, 2017. doi: 10.1145/3062341.3062358.

[90] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, pages 178–189, 2010. doi: 10.1007/978-3-642-13193-6_16.

[91] J. Boyar and R. Peralta. A small depth-16 circuit for the AES s-box. In *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, pages 287–298, 2012. doi: 10.1007/978-3-642-30436-1_24.

[92] J. Boyar, M. Dworkin, R. Peralta, M. Turan, C. Calik, and L. Brandao. Circuit minimization work, 2018. URL `http://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html`.

[93] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In R. L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pages 275–284. ACM, 1989. doi: 10.1145/73141.74843.

[94] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, pages 171–186, 2012. doi: 10.1007/978-3-642-27954-6_11.

[95] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. doi: 10.1016/j.comnet.2005.01.010.

[96] D. Canright. A very compact s-box for AES. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, pages 441–455, 2005. doi: 10.1007/11545262_32. URL `https://doi.org/10.1007/11545262_32`.

[97] C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain. Higher-order masking schemes for s-boxes. In *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, pages 366–384, 2012. doi: 10.1007/978-3-642-34047-5_21.

[98] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 1987)*, pages 178–188, Munich, Germany, Jan. 1987. ACM Press.

[99] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 69–76, 2017. doi: 10.1109/SecDev.2017.24.

[100] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. Fact: a DSL for timing-sensitive computation. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019. doi: 10.1145/3314221.3314605.

[101] J. M. Cebrian, M. Jahre, and L. Natvig. Optimized hardware for suboptimal software: The case for simd-aware benchmarks. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 66–75. IEEE Computer Society, 2014. doi: 10.1109/ISPASS.2014.6844462.

[102] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981. doi: 10.1016/0096-0551(81)90048-5.

[103] A. Chakraborti, N. Datta, A. Jha, and M. Nandi. HyENA. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/hyena-spec-round2.pdf`.

[104] B. Chakraborty and M. Nandi. ORANGE. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/orange-spec.pdf`.

[105] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 398–412, 1999. doi: 10.1007/3-540-48405-1_26.

[106] D. Chaum. Design concepts for tamper responding systems. In D. Chaum, editor, *Advances in Cryptology, Proceedings of CRYPTO '83, Santa Barbara, California, USA, August 21-24, 1983*, pages 387–392. Plenum Press, New York, 1983.

[107] Z. Chen, J. Shen, A. Nicolau, A. V. Veidenbaum, N. F. Ghalaty, and R. Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced simdization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 57–64, 2017. doi: 10.1109/FDTC.2017.10. URL `https://doi.org/10.1109/FDTC.2017.10`.

[108] M. Ciet and M. Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Des. Codes Cryptogr.*, 36(1):33–43, 2005. doi: 10.1007/s10623-003-1160-8.

[109] C. Clavier. Secret external encodings do not prevent transient fault analysis. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2007. doi: 10.1007/978-3-540-74735-2_13.

[110] T. D. Cnudde and S. Nikova. More efficient private circuits II through threshold implementations. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 114–124. IEEE Computer Society, 2016. doi: 10.1109/FDTC.2016.15.

[111] T. D. Cnudde and S. Nikova. Securing the PRESENT block cipher against combined side-channel analysis and fault attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 25(12):3291–3301, 2017. doi: 10.1109/TVLSI.2017.2713483.

[112] L. Cojocar, K. Papagiannopoulos, and N. Timmers. Instruction duplication: Leaky and not too fault-tolerant! In T. Eisenbarth and Y. Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2017. doi: 10.1007/978-3-319-75208-2_10.

[113] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. Cryptographics: Secret key cryptography using graphics cards. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 334–350, 2005. doi: 10.1007/978-3-540-30574-3_23.

[114] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 45–60, 2009. doi: 10.1109/SP.2009.19.

[115] J. Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In B. Preneel and F. Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2018. doi: 10.1007/978-3-319-93387-0_4.

[116] J. Coron, E. Prouff, and M. Rivain. Side channel cryptanalysis of a higher order masking scheme. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 28–44, 2007. doi: 10.1007/978-3-540-74735-2_3.

[117] J. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, pages 410–424, 2013. doi: 10.1007/978-3-662-43933-3_21.

[118] J. Daemen and V. Rijmen. Resistance against implementation attacks: A comparative study of the aes proposals. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, volume 82, 1999. URL `https://www.esat.kuleuven.be/cosic/publications/article-362.ps`.

[119] J. Daemen and V. Rijmen. Aes proposal: Rijndael, 1999. URL `http://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf`.

[120] J. Daemen, S. Hoffert, G. V. Assche, and R. V. Keer. Xoodoo cookbook. *IACR Cryptol. ePrint Arch.*, 2018:767, 2018. URL `https://eprint.iacr.org/2018/767`.

[121] J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer. Xoodyak, a lightweight cryptographic scheme. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf`.

[122] J. Daemen, P. Maat, C. Massolino, and Y. Rotella. The subterranean 2.0 cipher suite. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/subterranean-spec-round2.pdf`.

[123] W. Dai. Crypto++ library. URL `https://github.com/weidai11/cryptopp`.

[124] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[125] S. Dhooghe and S. Nikova. My gadget just cares for me - how NINA can prove security against combined attacks. *IACR Cryptology ePrint Archive*, 2019:615, 2019. URL `https://eprint.iacr.org/2019/615`.

[126] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1. 2. submission to the caesar competition (2016), 2016. URL `https://competitions.cr.yp.to/round3/asconv12.pdf`.

[127] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018 (3):547–572, 2018. doi: 10.13154/tches.v2018.i3.547-572.

[128] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf`.

[129] R. Dolbeau and D. J. Bernstein. chacha20 dolbeau amd64-avx2, 2014. URL `https://bench.cr.yp.to/supercop.html`.

[130] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446, 2013. URL `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev`.

[131] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006. doi: 10.1007/11817963_11.

[132] H. Eldib and C. Wang. Synthesis of masking countermeasures against side channel attacks. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014. doi: 10.1007/978-3-319-08867-9_8.

[133] H. Eldib, C. Wang, and P. Schaumont. Smt-based verification of software countermeasures against side-channel attacks. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2014. doi: 10.1007/978-3-642-54862-8_5.

[134] P. Estérie, M. Gaunard, J. Falcou, J. Lapresté, and B. Rozoy. Boost.simd: generic programming for portable simdization. In P. Yew, S. Cho, L. DeRose, and D. J. Lilja, editors, *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 431–432. ACM, 2012. doi: 10.1145/2370816.2370881.

[135] J. Fan, B. Gierlichs, and F. Vercauteren. To infinity and beyond: Combined attack on ECC using points of low order. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2011. doi: 10.1007/978-3-642-23951-9_10.

[136] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3, 2010.

[137] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. In S. Chatterjee, J. F. Prins, L. Carter, J. Ferrante, Z. Li, D. C. Sehr, and P. Yew, editors, *Languages and Compilers for Parallel Computing, 11th International Workshop, LCPC'98, Chapel Hill, NC, USA, August 7-9, 1998, Proceedings*, volume 1656 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 1998. doi: 10.1007/3-540-48319-5_19.

[138] A. Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, pages 02–29, 2012.

[139] J. J. A. Fournier, S. W. Moore, H. Li, R. D. Mullins, and G. S. Taylor. Security evaluation of asynchronous circuits. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2003. doi: 10.1007/978-3-540-45238-6_12.

[140] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2): 216–231, 2005. doi: 10.1109/JPROC.2004.840301.

[141] T. Fuhr, É. Jaulmes, V. Lomné, and A. Thillard. Fault attacks on AES with faulty ciphertexts only. In W. Fischer and J. Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 108–118. IEEE Computer Society, 2013. doi: 10.1109/FDTC.2013.18.

[142] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, number Generators, pages 251–261, 2001. doi: 10.1007/3-540-44709-1_21.

[143] G. Gaubatz and B. Sunar. Robust finite field arithmetic for fault-tolerant public-key cryptography. In L. Breveglieri, I. Koren, D. Naccache, and J. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2006. doi: 10.1007/11889700_18.

[144] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In R. L. Wexelblat, editor, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, pages 11–16. ACM, 1986. doi: 10.1145/12276.13312.

[145] C. Giraud. An RSA implementation resistant to fault attacks and to simple power analysis. *IEEE Trans. Computers*, 55(9):1116–1120, 2006. doi: 10.1109/TC.2006.135.

[146] J. D. Golic and C. Tymen. Multiplicative masking and power analysis of AES. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 198–212, 2002. doi: 10.1007/3-540-36400-5_16.

[147] J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In J. Lenfant, editor, *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*, pages 442–452. ACM, 1988. doi: 10.1145/55364.55407.

[148] V. Gopal, J. Guilford, W. Feghali, E. Ozturk, G. Wolrich, and M. Dixon. Processing multiple buffers in parallel to increase performance on intel architecture processors, 2010. URL `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf`.

[149] M. Gottschlag, T. Schmidt, and F. Bellosa. AVX overhead profiling: how much does your fast code slow you down? In T. Kim and P. P. C. Lee, editors, *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, August 24-25, 2020*, pages 59–66. ACM, 2020. doi: 10.1145/3409963.3410488.

[150] L. Goubin and J. Patarin. DES and differential power analysis (the "duplication" method). In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 158–172, 1999. doi: 10.1007/3-540-48059-5_15.

[151] D. Goudarzi, J. Jean, S. Kölbl, T. Peyrin, M. Rivain, Y. Sasaki, and S. M. Sim. Pyjamask. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/pyjamask-spec-round2.pdf`.

[152] S. Gueron and V. Krasnov. The fragility of AES-GCM authentication algorithm. In *11th International Conference on Information Technology: New Generations, ITNG 2014, Las Vegas, NV, USA, April 7-9, 2014*, pages 333–337, 2014. doi: 10.1109/ITNG.2014.31.

[153] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 222–239, 2011. doi: 10.1007/978-3-642-22792-9_13.

[154] J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, pages 116–145, 2005. doi: 10.1007/11580850_8.

[155] J. Götzfried. Serpent avx2, 2012. URL `https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/avx-8way-1`.

[156] J. Götzfried. Serpent avx2, 2012. URL `https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/avx2-16way-1`.

[157] D. H. Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 12(5):91–100, 1965. URL `https://www.osti.gov/servlets/purl/4609524`.

[158] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin. Fast AES implementation: A high-throughput bitsliced approach. *IEEE Trans. Parallel Distrib. Syst.*, 30(10):2211–2222, 2019. doi: 10.1109/TPDS.2019.2911278.

[159] O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, pages 350–367, 2009. doi: 10.1007/978-3-642-02384-2_22.

[160] S. He, M. Emmi, and G. F. Ciocarlie. ct-fuzz: Fuzzing for timing leaks. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 466–471, 2020. doi: 10.1109/ICST46399.2020.00063.

[161] K. Heydemann. *Sécurité et performance des applications : analyses et optimisations multi-niveaux*. Habilitation, Université Pierre et Marie Curie (UPMC), 2017.

[162] N. M. Huu, B. Robisson, M. Agoyan, and N. Drach. Low-cost recovery for the code integrity protection in secure embedded processors. In *HOST 2011, Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 5-6 June 2011, San Diego, California, USA*, pages 99–104. IEEE Computer Society, 2011. doi: 10.1109/HST.2011.5955004.

[163] T. in Soft. TIS-CT. URL `https://trust-in-soft.com/tis-ct/`.

[164] Intel. Intel® vtune™ profiler. URL `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html`.

[165] Y. Ishai, A. Sahai, and D. A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 463–481, 2003. doi: 10.1007/978-3-540-45146-4_27.

[166] T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus. *Submission to NIST-LWC*, 2019. URL `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Romulus-spec.pdf`.

[167] A. Journault and F. Standaert. Very high order masking: Efficient implementation and security evaluation. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 623–643, 2017. doi: 10.1007/978-3-319-66787-4_30.

[168] M. Joye, P. Paillier, and B. Schoenmakers. On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, pages 293–308, 2005. doi: 10.1007/11545262_22.

[169] S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, pages 131–144, 1988. doi: 10.1007/3-540-19027-9_9.

[170] D. Kaloper-Mersinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 223–238. USENIX Association, 2015. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kaloper-mersinjak`.

[171] D. Karaklajic, J. Schmidt, and I. Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(12):2295–2306, 2013. doi: 10.1109/TVLSI.2012.2231707.

[172] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[173] R. Karri, K. Wu, P. Mishra, and Y. Kim. Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 579–585. ACM, 2001. doi: 10.1145/378239.379027.

[174] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. *CHES*, 2009. doi: 10.1007/978-3-540-74735-2_9.

[175] E. Käsper and P. Schwabe. Aes-ctr, non-constant-time key setup, 2009. URL `https://cryptojedi.org/crypto/data/aes-ctr-128-const-intel64-20090611.tar.bz2`.

[176] J. Kelsey, B. Schneier, and D. A. Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and TEA. In Y. Han, T. Okamoto, and S. Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11-14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1997. doi: 10.1007/BFb0028479.

[177] J. Kelsey, B. Schneier, D. A. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*, pages 97–110, 1998. doi: 10.1007/BFb0055858.

[178] P. Kiaei and P. Schaumont. Synthesis of parallel synchronous software. *CoRR*, abs/2005.02562, 2020. URL `https://arxiv.org/abs/2005.02562`.

[179] P. Kiaei, D. Mercadier, P. Dagand, K. Heydemann, and P. Schaumont. Custom instruction support for modular defense against side-channel and fault attacks. In *Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE, October 5-7, 2020*. Springer, 2020. URL `https://eprint.iacr.org/2020/466`.

[180] O. Kiselyov. Implementing, and understanding type classes. *updated November*, 2014. URL `http://okmij.org/ftp/Computation/typeclass.html`.

[181] J. Kivilinna. Block ciphers: Fast implementations on x86-64 architecture, 2011. URL `https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/aes128ctr/avx`.

[182] J. Kivilinna. Serpent sse2 implementation, 2011. URL `https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/sse2-8way`.

[183] J. Kivilinna. Block ciphers: fast implementations on x86-64 architecture. Master's thesis, University of Oulu, Faculty of Science, Department of Information Processing Science, Information Processing Science, 2013.

[184] D. E. Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN 0201896850. URL `http://www.worldcat.org/oclc/312994415`.

[185] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. doi: 10.1007/3-540-68697-5_9.

[186] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. doi: 10.1007/3-540-48405-1_25.

[187] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011. doi: 10.1007/s13389-011-0006-y.

[188] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In S. B. Guthery and P. Honeyman, editors, *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. USENIX Association, 1999. URL `https://www.usenix.org/conference/usenix-workshop-smartcard-technology/design-principles-tamper-resistant-smartcard`.

[189] R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pages 187–202, 2008. doi: 10.1007/978-3-540-79263-5_12.

[190] M. Kretz and V. Lindenstruth. Vc: A C++ library for explicit vectorization. *Softw. Pract. Exp.*, 42(11): 1409–1430, 2012. doi: 10.1002/spe.1149.

[191] M. Kwan. Reducing the gate count of bitslice DES. *IACR Cryptology ePrint Archive*, 2000:51, 2000. URL `http://eprint.iacr.org/2000/051`.

[192] M. Kwan. Bitslice des, 2000. URL `http://www.darkside.com.au/bitslice/`.

[193] B. Lac, A. Canteaut, J. J. A. Fournier, and R. Sirdey. Thwarting fault attacks against lightweight cryptography using SIMD instructions. In *IEEE International Symposium on Circuits and Systems, ISCAS 2018, 27-30 May 2018, Florence, Italy*, pages 1–5, 2018. doi: 10.1109/ISCAS.2018.8351693.

[194] X. Lai, J. L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer, 1991. doi: 10.1007/3-540-46416-6_2.

[195] A. Langley. ctgrind, 2010. URL `https://github.com/agl/ctgrind/`.

[196] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[197] C.-Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[198] W. Lee, H. Cheong, R. C. Phan, and B. Goi. Fast implementation of block ciphers and PRNGs in Maxwell GPU architecture. *Cluster Computing*, 19(1):335–347, 2016. doi: 10.1007/s10586-016-0536-2.

[199] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814.

[200] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008. doi: 10.1007/s10817-008-9099-0.

[201] J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. volume 2, pages 820–825, 2003.

[202] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 289–300, 2013. doi: 10.1145/2463676.2465322.

[203] R. K. Lim, L. R. Petzold, and Ç. K. Koç. Bitsliced high-performance AES-ECB on GPUs. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, pages 125–133, 2016. doi: 10.1007/978-3-662-49301-4_8.

[204] F. J. MacWilliams and N. J. A. Sloane. *The theory of error correcting codes*, volume 16. Elsevier, 1977.

[205] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinform.*, 9(S-2), 2008. doi: 10.1186/1471-2105-9-S2-S10.

[206] M. Matsui. How far can we go on the x64 processors? In *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, pages 341–358, 2006. doi: 10.1007/11799313_22.

[207] M. Matsui and J. Nakajima. On the power of bitslice implementation on intel core2 processor. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 121–134, 2007. doi: 10.1007/978-3-540-74735-2_9.

[208] P. M. Maurer and W. J. Schilp. Software bit-slicing: A technique for improving simulation performance. In *1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany*, pages 786–787, 1999. doi: 10.1109/DATE.1999.761231.

[209] M. Medwed and J. Schmidt. Coding schemes for arithmetic and logic operations - how robust are they? In H. Y. Youm and M. Yung, editors, *Information Security Applications, 10th International Workshop, WISA 2009, Busan, Korea, August 25-27, 2009, Revised Selected Papers*, volume 5932 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2009. doi: 10.1007/978-3-642-10838-9_5.

[210] M. Medwed, F. Standaert, J. Großschädl, and F. Regazzoni. Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In D. J. Bernstein and T. Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010. doi: 10.1007/978-3-642-12678-9_17.

[211] D. Mercadier and P. Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 157–173. ACM, 2019. doi: 10.1145/3314221.3314636.

[212] D. Mercadier, P. Dagand, L. Lacassagne, and G. Muller. Usuba: Optimizing & trustworthy bitslicing compiler. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 4:1–4:8, 2018. doi: 10.1145/3178433.3178437.

[213] T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000. doi: 10.1007/3-540-44499-8_19.

[214] T. S. Messerges. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, pages 150–164, 2000. doi: 10.1007/3-540-44706-7_11.

[215] J. Mick and J. Brick. *Bit-slice microprocessor design*. McGraw-Hill, Inc., 1980.

[216] A. Moghimi, T. Eisenbarth, and B. Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 21–44, 2018. doi: 10.1007/978-3-319-76953-0_2.

[217] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, pages 156–168, 2005. doi: 10.1007/11734727_14.

[218] S. K. Monfared, O. Hajihassani, S. M. Zanjani, S. Kiarostami, D. Rahmati, and S. Gorgin. High-performance cryptographically secure pseudo-random number generation via bitslicing. *CoRR*, abs/1909.04750, 2019. `http://arxiv.org/abs/1909.04750`.

[219] A. Moon. chacha opt. URL `https://github.com/floodyberry/chacha-opt`.

[220] S. Morioka and A. Satoh. An optimized s-box circuit architecture for low power AES design. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 172–186, 2002. doi: 10.1007/3-540-36400-5_14.

[221] A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2012. doi: 10.1007/978-3-642-33027-8_4.

[222] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. *Courant Institute, New York University*, 1995. URL `http://i.stanford.edu/pub/cstr/reports/cs/tn/95/22/CS-TN-95-22.pdf`.

[223] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 710–728. IEEE Computer Society, 2017. doi: 10.1109/SP.2017.53.

[224] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[225] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security*, pages 529–545, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-49497-3.

[226] N. Nishikawa, H. Amano, and K. Iwai. Implementation of bitsliced AES encryption on cuda-enabled GPU. In *Network and System Security - 11th International Conference, NSS 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, pages 273–287, 2017. doi: 10.1007/978-3-319-64701-2_20.

[227] NIST. Lightweight cryptography (lwc) standartization, 2019. URL `https://csrc.nist.gov/Projects/lightweight-cryptography`.

[228] D. Noer, A. P. Engsig-Karup, and E. Zenner. Improved software implementation of des using cuda and opencl. In *Western European Workshop on Research in Cryptology*, 2011. `https://orbit.dtu.dk/files/5637699/abstract.pdf`.

[229] N. B. of Standards. Announcing the data encryption standard. Technical Report FIPS Publication 46, National Bureau of Standards, Jan. 1977.

[230] N. B. of Standards. Announcing the Advanced Encryption Standard (AES). Technical report, National Bureau of Standards, 2001.

[231] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 38–49, 1997. doi: 10.1145/253260.253268.

[232] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013. URL `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[233] OpenSSL. perlasm, 2019. URL `https://github.com/openssl/openssl/tree/master/crypto/perlasm`.

[234] D. A. Osvik. Speeding up serpent. In *AES Candidate Conference*, pages 317–329, 2000.

[235] D. A. Osvik and J. Kivilinna. Linux serpent, 2002. URL `https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/linux_c`.

[236] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, pages 75–93, 2010. doi: 10.1007/978-3-642-13858-4_5.

[237] E. Oswald and M. J. Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2001. doi: 10.1007/3-540-44709-1_5.

[238] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptol. ePrint Arch.*, 2002:169, 2002. URL `http://eprint.iacr.org/2002/169`.

[239] J. Park and D. Lee. Face: Fast aes ctr mode encryption techniques based on the reuse of repetitive data. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):469–499, Aug. 2018. doi: 10.13154/tches.v2018.i3.469-499.

[240] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pages 231–244, 2016. doi: 10.1007/978-3-319-69453-5_13.

[241] B. Peccerillo, S. Bartolini, and Ç. K. Koç. Parallel bitsliced AES through PHAST: a single-source high-performance library for multi-cores and GPUs. *J. Cryptographic Engineering*, 9(2):159–171, 2019. doi: 10.1007/s13389-017-0175-4.

[242] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *InPar '12: Innovative Parallel Computing*, pages 1–13, May 2012.

[243] S. S. Pinter. Register allocation with instruction scheduling: A new approach. In R. Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 248–257. ACM, 1993. doi: 10.1145/155090.155114.

[244] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pages 151–166, 1998. doi: 10.1007/BFb0054170.

[245] T. Popp and S. Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, pages 172–186, 2005. doi: 10.1007/11545262_13.

[246] T. Pornin. BearSSL, a smaller SSL/TLS library, . URL `https://bearssl.org/`.

[247] T. Pornin. Constant-time mul. `https://www.bearssl.org/ctmul.html`, . Accessed: 2019-12-01.

[248] T. Pornin. *Implantation et optimisation des primitives cryptographiques*. PhD thesis, École Normale Supérieure, 2001. URL `http://www.bolet.org/~pornin/2001-phd-pornin.pdf`.

[249] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.*, 1(ICFP):17:1–17:29, 2017. doi: 10.1145/3110261.

[250] E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 142–159, 2013. doi: 10.1007/978-3-642-38348-9_9.

[251] J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, pages 200–210, 2001. doi: 10.1007/3-540-45418-7_17.

[252] J.-J. Quisquater. Eddy current for magnetic analysis with active sensor. *Proceedings of Esmart, 2002*, pages 185–194, 2002.

[253] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013. doi: 10.1145/2491956.2462176.

[254] J. Rao, X. Zou, and K. Dai. dscadl: A data flow based symmetric cryptographic algorithm description language. In *2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET)*, pages 84–89. IEEE, 2019. doi: 10.1109/CCET48361.2019.8989331.

[255] J. R. Rao and P. Rohatgi. Empowering side-channel attacks. *IACR Cryptol. ePrint Arch.*, 2001:37, 2001. URL `http://eprint.iacr.org/2001/037`.

[256] F. Regazzoni, T. Eisenbarth, J. Großschädl, L. Breveglieri, P. Ienne, I. Koren, and C. Paar. Power attacks resistance of cryptographic s-boxes with added error detection circuits. In C. Bolchini, Y. Kim, A. Salsano, and N. A. Touba, editors, *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007), 26-28 September 2007, Rome, Italy*, pages 508–516. IEEE Computer Society, 2007. doi: 10.1109/DFT.2007.61.

[257] F. Regazzoni, L. Breveglieri, P. Ienne, and I. Koren. Interaction between fault attack countermeasures and the resistance against power analysis attacks. In M. Joye and M. Tunstall, editors, *Fault Analysis in Cryptography*, Information Security and Cryptography, pages 257–272. Springer, 2012. doi: 10.1007/978-3-642-29656-7_15.

[258] G. Ren. *Compiling Vector Programs for SIMD Devices*. PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[259] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1697–1702, 2017. doi: 10.23919/DATE.2017.7927267.

[260] O. Reparaz, L. De Meyer, B. Bilgin, V. Arribas, S. Nikova, V. Nikov, and N. P. Smart. CAPA: the spirit of beaver against physical attacks. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 121–151, 2018. doi: 10.1007/978-3-319-96884-1_5.

[261] V. Rijmen. Efficient implementation of the rijndael s-box. *Katholieke Universiteit Leuven, Dept. ESAT. Belgium*, 2000.

[262] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 413–427, 2010. doi: 10.1007/978-3-642-15031-9_28.

[263] M. Rivain, E. Dottax, and E. Prouff. Block ciphers implementations provably secure against second order side channel analysis. In *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, pages 127–143, 2008. doi: 10.1007/978-3-540-71039-4_8.

[264] R. L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992. doi: 10.17487/RFC1321.

[265] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. doi: 10.1145/359340.359342.

[266] A. D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[267] T. Roche, V. Lomné, and K. Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In E. Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*, pages 65–83. Springer, 2011. doi: 10.1007/978-3-642-27257-8_5.

[268] F. Rocheteau. *Extension du langage LUSTRE et application à la conception de circuits : le langage LUSTRE-V4 et le système POLLUX. (Extension of the lustre language and application to hardware design : the lustre-v4 language and the pollux system)*. PhD thesis, Grenoble Institute of Technology, France, 1992. URL `https://tel.archives-ouvertes.fr/tel-00342092`.

[269] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In A. Zaks and M. V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120. ACM, 2016. doi: 10.1145/2892208.2892230.

[270] J. Schmidt and C. Herbst. A practical fault attack on square and multiply. In L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. Seifert, editors, *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*, pages 53–58. IEEE Computer Society, 2008. doi: 10.1109/FDTC.2008.10.

[271] J.-M. Schmidt and M. Hutter. Optical and fault-attacks on CRT-based RSA: Concrete results. 2007. URL `http://mhutter.org/papers/Schmidt2007OpticalandEM.pdf`.

[272] T. Schneider, A. Moradi, and T. Güneysu. Parti: Towards combined hardware countermeasures against side-channeland fault-injection attacks. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 39, 2016. doi: 10.1145/2996366.2996427.

[273] K. Schramm and C. Paar. Higher order masking of the AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, pages 208–225, 2006. doi: 10.1007/11605805_14.

[274] P. Schwabe and K. Stoffelen. All the AES you need on cortex-m3 and M4. In R. Avanzi and H. M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016. doi: 10.1007/978-3-319-69453-5_10.

[275] R. Sethi. Complete register allocation problems. In A. V. Aho, A. Borodin, R. L. Constable, R. W. Floyd, M. A. Harrison, R. M. Karp, and H. R. Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 182–195. ACM, 1973. doi: 10.1145/800125.804049.

[276] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. doi: 10.1145/359168.359176.

[277] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai. Piccolo: An ultra-lightweight blockcipher. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 342–357, 2011. doi: 10.1007/978-3-642-23951-9_23.

[278] T. Simon, L. Batina, J. Daemen, V. Grosso, P. M. C. Massolino, K. Papagiannopoulos, F. Regazzoni, and N. Samwel. Towards lightweight cryptographic primitives with built-in fault-detection. *IACR Cryptology ePrint Archive*, 2018:729, 2018. URL `https://eprint.iacr.org/2018/729`.

[279] S. Skorobogatov. Low temperature data remanence in static ram. Technical report, University of Cambridge, Computer Laboratory, 2002. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf`.

[280] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002. doi: 10.1007/3-540-36400-5_2.

[281] M. D. Smith and G. Holloway. Graph-coloring register allocation for irregular architectures, 2000. URL `https://www.cs.tufts.edu/comp/150FP/archive/mike-smith/irreg.pdf`.

[282] M. Sprengers. GPU-based password cracking. Master's thesis, Radboud Universiteit Nijmegen, 2011. `https://www.ru.nl/publish/pages/769526/thesis.pdf`.

[283] S. H. Standard. Fips pub 180-1. Technical report, 1995.

[284] K. Stoffelen. Optimizing s-box implementations for several criteria using SAT solvers. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2016. doi: 10.1007/978-3-662-52993-5_8.

[285] W. Sun, R. Ricci, and M. L. Curry. Gpustore: harnessing GPU computing for storage systems in the OS kernel. In *The 5th Annual International Systems and Storage Conference, SYSTOR '12, Haifa, Israel, June 4-6, 2012*, page 6, 2012. doi: 10.1145/2367589.2367595.

[286] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi: 10.1145/2837614.2837655.

[287] R. Szerwinski and T. Güneysu. Exploiting the power of gpus for asymmetric cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, pages 79–99, 2008. doi: 10.1007/978-3-540-85053-3_6.

[288] G. Tang, K. Takata, M. Tanaka, A. Fujimaki, K. Takagi, and N. Takagi. 4-bit bit-slice arithmetic logic unit for 32-bit rsfq microprocessors. *IEEE Transactions on Applied Superconductivity*, 26(1):1–6, 2016. doi: 10.1109/TASC.2015.2507125.

[289] S. Tillich and J. Großschädl. Power analysis resistant AES implementation with instruction set extensions. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2007. doi: 10.1007/978-3-540-74735-2_21.

[290] K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European solid-state circuits conference*, pages 403–406. IEEE, 2002. URL `https://securewww.esat.kuleuven.be/cosic/publications/article-705.pdf`.

[291] R. F. Touzeau. A fortran compiler for the FPS-164 scientific computer. In M. S. V. Deusen and S. L. Graham, editors, *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, pages 48–57. ACM, 1984. doi: 10.1145/502874.502879.

[292] J. Tristan and X. Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 17–27, 2008. doi: 10.1145/1328438.1328444.

[293] M. Ullrich, C. De Canniere, S. Indesteege, Ö. Küçük, N. Mouha, and B. Preneel. Finding optimal bitsliced implementations of 4× 4-bit s-boxes. In *SKEW 2011 Symmetric Key Encryption Workshop, Copenhagen, Denmark*, pages 16–17, 2011. URL `http://skew2011.mat.dtu.dk/proceedings/Finding%20Optimal%20Bitsliced%20Implementations%20of%204%20to%204-bit%20S-boxes.pdf`.

[294] W. van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Comput. Secur.*, 4(4):269–286, 1985. doi: 10.1016/0167-4048(85)90046-X.

[295] J. Waddle and D. A. Wagner. Towards efficient second-order power analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 1–15, 2004. doi: 10.1007/978-3-540-28632-5_1.

[296] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989. doi: 10.1145/75277.75283.

[297] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, 2019. doi: 10.1145/3290390.

[298] S. H. Weingart. Physical security for the μabyss system. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pages 52–59. IEEE Computer Society, 1987. doi: 10.1109/SP.1987.10019.

[299] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*, page 38, 1998. doi: 10.1109/SC.1998.10004.

[300] F. Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992. doi: 10.2307\%2F3001968.

[301] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, 2009. doi: 10.14778/1687627.1687671.

[302] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES sboxes. In *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, pages 67–78, 2002. doi: 10.1007/3-540-45760-7_6.

[303] P. Wright. *Spycatcher*. 1987.

[304] M. Wu, S. Guo, P. Schaumont, and C. Wang. Eliminating timing side-channel leaks using program repair. In F. Tip and E. Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 15–26. ACM, 2018. doi: 10.1145/3213846.3213851.

[305] S. Xu and D. Gregg. Bitslice vectors: A software approach to customizable data precision on processors with SIMD extensions. In *46th International Conference on Parallel Processing, ICPP 2017, Bristol, United Kingdom, August 14-17, 2017*, pages 442–451, 2017. doi: 10.1109/ICPP.2017.53.

[306] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong. The simeck family of lightweight block ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 307–329, 2015. doi: 10.1007/978-3-662-48324-4_16.

[307] J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. In *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, pages 249–264, 2007. doi: 10.1007/978-3-540-76900-2_15.

[308] S. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Trans. Computers*, 49(9):967–970, 2000. doi: 10.1109/12.869328.

[309] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede. RECTANGLE: A bit-slice ultra-lightweight block cipher suitable for multiple platforms. *IACR Cryptology ePrint Archive*, 2014:84, 2014. URL `http://eprint.iacr.org/2014/084`.

[310] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. *IACR Cryptology ePrint Archive*, 2017:536, 2017. URL `http://eprint.iacr.org/2017/536`.

[311] L. Zussa, J. Dutertre, J. Clédière, and A. Tria. Power supply glitch induced faults on FPGA: an in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*, pages 110–115. IEEE, 2013. doi: 10.1109/IOLTS.2013.6604060.

Appendix A

Backend optimizations

In order to give more visual intuitions of the impact of Usubac's optimizations, we reported the results by the mean of figures in Section 4.2. In this appendix, in order to give a more precise understanding of the impact of these optimizations, we provide the more precise (but less readable) results.

Cipher	Inlining speedup			
	Clang		GCC	
	x86	**AVX2**	**x86**	**AVX2**
ACE	1.54	1.33	1.64	1.01
AES	-	1.01	-	1.43
ASCON	1.20	1.01	1.89	1.15
CHACHA20	1.25	1.11	1.23	1.20
CLYDE	1.16	1.02	1.16	1.22
GIFT	1.69	0.93	1.37	1.05
GIMLI	0.97	0.99	1.23	1.33
PYJAMASK	1.35	0.99	1.08	1.11
RECTANGLE (*h*slice)	-	0.96	-	0.97
RECTANGLE (*v*slice)	1.00	0.99	0.97	0.96
SERPENT	1.01	0.99	1.27	1.27
XOODOO	1.25	0.98	1.61	1.39

Table A.1: Impact of fully inlining msliced ciphers

Cipher	**Mslice scheduling speedup**	
	x86	**AVX2**
ACE	1.35	1.39
AES	-	1.04
ASCON	1.10	1.04
CHACHA20	1.05	1.10
CLYDE	1.02	1.04
GIFT	1.01	1.00
GIMLI	0.98	1.01
RECTANGLE(*v*sliced)	0.97	1.00
RECTANGLE(*h*sliced)	-	1.02
SERPENT	0.97	1.00
XOODOO	1.03	1.00

Table A.2: Performance of the *m*slice scheduling algorithm (compiled with Clang 7.0.0)

Cipher	Inlining speedup			
	Clang		GCC	
	x86	**AVX2**	**x86**	**AVX2**
ACE	1.16	1.54	1.75	3.57
AES	1.28	1.64	1.27	1.43
ASCON	1.20	2.50	1.45	2.56
CLYDE	1.08	1.79	1.02	1.35
DES	1.41	1.96	1.30	1.72
GIFT	3.70	5.88	3.45	8.33
GIMLI	1.41	2.00	1.79	3.03
PHOTON	1.18	1.75	2.00	2.44
PRESENT	1.23	1.08	1.05	0.97
PYJAMASK	5.26	1.20	8.33	8.33
RECTANGLE	1.72	2.33	1.59	2.44
SKINNY	2.63	4.00	2.78	4.76
SPONGENT	1.52	3.12	1.49	3.03
SUBTERRANEAN	2.00	3.03	1.96	2.86
XOODOO	1.37	2.33	1.47	2.08

Table A.3: Impact of fully inlining bitsliced ciphers

Cipher	**Bitslice scheduling speedup**					
	GCC -Os		GCC -O3		Clang -O3	
	x86	**AVX2**	**x86**	**AVX2**	**x86**	**AVX2**
AES	1.04	0.99	1.06	0.98	1.08	1.04
ASCON	1.35	1.32	1.25	1.27	1.04	1.19
CLYDE	1.06	1.06	1.06	1.04	1.01	0.99
DES	1.16	1.19	1.16	1.23	1.01	1.02
GIFT	1.16	1.18	1.12	1.16	1.04	1.09
PHOTON	1.05	1.14	0.97	0.93	0.96	0.97
PRESENT	1.30	1.10	1.16	1.16	1.00	1.00
PYJAMASK	1.19	1.35	1.04	1.04	0.99	1.00
RECTANGLE	1.28	1.20	1.15	1.15	1.00	0.99
SERPENT	1.18	1.20	1.20	1.20	1.04	1.00
SKINNY	1.14	1.16	1.18	1.18	1.03	1.14

Table A.4: Performance of the bitslice scheduling algorithm