Usuba: High-Throughput and Constant-Time Ciphers, by Construction

Darius Mercadier
Pierre-Évariste Dagand
Sorbonne Université
CNRS, Inria, LIP6
Paris, France
firstname.lastname@lip6.fr

Abstract

Cryptographic primitives are subject to diverging imperatives. Functional correctness and auditability pushes for the use of a high-level programming language. Performance and the threat of timing attacks push for using no more abstract than an assembler to exploit (or avoid!) the micro-architectural features of a given machine. We believe that a suitable programming language can reconcile both views and actually improve on the state of the art of both. Usuba is an opinionated dataflow programming language in which block ciphers become so simple as to be "obviously correct" and whose types document and enforce valid parallelization strategies at the granularity of individual bits. Its optimizing compiler, `Usubac`, produces high-throughput, constant-time implementations performing on par with hand-tuned reference implementations. The cornerstone of our approach is a systematization and generalization of *bitslicing*, an implementation trick frequently used by cryptographers.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; *Data flow languages*; • **Security and privacy** → Block and stream ciphers.

Keywords Optimizing Compiler, Bitslicing, Vectorization

ACM Reference Format:
Darius Mercadier and Pierre-Évariste Dagand. 2019. Usuba: High-Throughput and Constant-Time Ciphers, by Construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314636

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314636

1 Introduction

Implementations of cryptographic primitives are subject to stringent requirements. The correctness of a primitive is not judged based solely on its behavior (a hard problem in itself [8, 15]) but within a threat model that may, for example, include timing attacks or any other side-channel. In this paper, we focus on timing attacks because, with the development of practical and remote timing attacks [10, 21], providing constant-time implementations has become standard practice in cryptographic libraries.

To write constant-time code is to fight an uphill battle against compilers [4], which may silently rewrite one's carefully crafted program into one vulnerable to timing attacks, and against the underlying architecture itself [52], whose micro-architectural features may leak secrets through timing or otherwise. The issue is so far-reaching that tools traditionally applied to hardware evaluation are now used to analyze software implementations [68], treating the program and its execution environment as a single black-box.

However, aiming for soundness and security cannot come at the cost of efficiency. These functions are executed repeatedly, processing massive amounts of data. Being able to sustain a high throughput is therefore essential. Bitslicing has been introduced by Biham [17] as an implementation trick to speed up software implementations of DES. Bitslicing achieves high throughput by increasing parallelism. Most record-breaking software implementations of block ciphers exploit this technique [6, 38, 42, 50]. Modern ciphers, meant to be executed in software, are now designed with bitslicing in mind [12, 18, 56, 72]. Their design purposefully allows the programmer to exploit SIMD instruction sets (from Arm's Neon extension to Intel's SSE, AVX and AVX512 extensions). Such ciphers are massively parallel, bit-level programs.

Most cryptographic primitives are likely to stay in software, never graduating to hardware – too expensive and too rigid. Since Intel Core i5, only AES became widely available in hardware thanks to its unique position as a NIST standard. At the other end of the spectrum, the Internet of Things has motivated the development of lightweight ciphers [5, 9, 19, 31] that run reasonably efficiently on low-end devices with power and code size constraints – a niche left

vacant by AES and which is unlikely to be filled by a one-size-fits-all solution. There is therefore a need for supporting a rich portfolio of cryptographic primitives, both on the low-end side (the deployed clients) and the high-end side (the server, interacting with thousands of clients). In this paper, we focus exclusively on high-throughput implementations for high-end servers, for which we can reasonably assume that the threat model is restricted to timing attacks (thus excluding physical interference with the machine). Targeting low-end devices and integrating countermeasures against physical tampering is out of the scope of this paper.

Our language, Usuba [51], has been designed to enable the high-level description of block ciphers that our compiler, Usubac, automatically turns into a high-throughput and constant-time implementation. The language design – its syntax, semantics and type system – guarantees that Usuba programs are *by construction* parallel and constant-time.

Usuba exploits and generalizes the bitslicing trick [17], also known as "SIMD Within A Register" [29] (SWAR) in the compiler community. The key insight of bitslicing is to consider a block cipher as a combinational circuit written using bitwise logic operators such as $x\&y$, $x\,|\,y$, $x\hat{}y$ and $\sim x$. On a 64-bit machine, the bitwise $x\&y$ operation effectively works like 64 parallel Boolean conjunctions, each processing a single bit. High throughput is achieved by parallelism: 64 instances of the cipher can execute in parallel. The implementation is constant-time by design: no data-dependent accesses to memory are made (or, in fact, possible at all).

While restrictive, this programming model is well-suited for the description of block ciphers. Bit-twiddling, typical of symmetric ciphers, is naturally expressed, often at no run-time cost. Their streaming model is compatible with parallel execution: in parallelizable encryption/decryption modes – such as counter mode (CTR) – multiple blocks of the input can be processed in parallel whereas, in non-parallelizable modes – such as output feedback (OFB) – one can still multiplex independent encryptions from several, concurrent clients [30]. Finally, bitsliced implementations can automatically scale to larger registers (*e.g.* SIMD).

On SIMD architectures, we can generalize bitslicing in two variants and thus broaden the scope of the original Usuba language [51]. Rather than considering the bit as the atomic unit of computation, one may use m-bit words as such a basis (m being a word size supported by the SIMD engine). On Intel AVX2, m could for example be 8, 16, 32, 64. We can then exploit *vertical* SIMD operations to perform logic as well as arithmetic in parallel. If we visually represent registers as vertically stacked, vertical operations consist in element-wise computations along this vertical direction. On Intel AVX2, we can apply 32 parallel additions on 8-bit atoms ($m = 8$). We call this transformation "vertical slicing", or "vslicing" for short.

Rather than considering an m-bit atom as a single packed element, we may also dispatch its m bits into m distinct

```
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: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
node Rectangle (plain:u16x4,key:u16x4[26])
       returns (cipher:u16x4)
vars round : u16x4[26]
let round[0] = plain;
    forall i in [0,24] {
      round[i+1] = ShiftRows(SubColumn(round[i]
                                       ^ key[i]))
    }
    cipher = round[25] ^ key[25]
tel
```

Figure 1. Rectangle cipher

packed elements, assuming that m is less or equal to the number of packed elements of the architecture. We lose the ability to perform arithmetic operations but gain the ability to arbitrarily shuffle bits of our m-bit word in a single instruction (using `vpshufb` on Intel AVX2). This style relies on *horizontal* SIMD operations: we can perform element-wise computations within a single register, *i.e.* along an horizontal direction. We call this transformation "horizontal slicing", or "hslicing" for short. Whenever the slicing direction is unimportant, we talk about "m-slicing" (assuming that $m > 1$) and call "slicing" the technique encompassing bitslicing ($m = 1$) and m-slicing ($m > 1$).

As shown in Section 4, the choice between bitslicing, vslicing and hslicing depends on the cipher and the target architecture. On x86-64 (without SIMD extensions), we have no choice but to resort to bitslicing. On AVX, the fastest implementation of the Rectangle cipher is vsliced [72] whereas the fastest implementation of AES is hsliced [38].

Manually writing and maintaining a sliced program is extremely tedious. For the sake of illustration, Kwan's implementation of DES is composed of thousands of lines like the following:

```
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);
```

As we shall demonstrate in Section 2, Usuba enables us to translate almost literally the textbook description of a cipher. For example, Figure 1 shows the (complete) implementation of the Rectangle cipher [72].

Usuba provides domain-specific language constructs, a type-system guaranteeing safety as well as parallelism, and an optimizing compiler striving to exploit SIMD and maximize throughput. By abstracting away parallelism from the implementation, an Usuba program is trivial to write and easy to maintain.

In summary, the main contributions of this paper are:

- We present Usuba, a statically-typed vector-based programming language for specifying block ciphers (Section 2). Its programming model makes the description of cryptographic algorithms intuitive while its type system reconciles the need for abstraction – for code reuse – and specialization – giving access to architecture-specific features ;
- We describe `Usubac` (Section 3), an optimizing compiler[1] translating Usuba to C. It involves a combination of folklore techniques (inlining, unrolling) tailored to the unique problems posed by sliced programs and introduces new techniques (interleaving, sliced scheduling) made possible by our programming model ;
- Finally, we evaluate our Usuba implementation of 5 ciphers (Section 4) in various combinations of slicing modes and target architectures. We show that `Usubac` produces code outperforming hand-tuned implementations written in assembly while scaling gracefully across several generations of SIMD architectures. Thanks to Usuba's expressivity, we are able to compare, at no expense, the performance of all the slicing modes supported by a given cipher, allowing us to carry the first – to our knowledge – performance evaluation of slicing modes across instruction sets.

Aside from the technical challenge of producing "faster than hand-tuned" cryptographic code from high-level descriptions, this paper is an attempt to draw the consequences of the end of Moore's law. To provide sustained performance improvements, hardware designers turn to specialization. Re-compiling old software to target a new architecture does not automatically pay off anymore, if it ever did. A high-performance cryptographic primitive for AVX2 is but a poor cryptographic primitive for AVX512.

Usuba takes the stance that specialized hardware calls for specializable software. The language design reflects this by enabling the programmer to encode the parallelism of a cipher in its type. The compiler takes side with Gustafson [32], targeting faster yet more specialized architectures to tackle larger problems within the same time frame, *i.e.* striving for increased throughput.

2 The Usuba Language

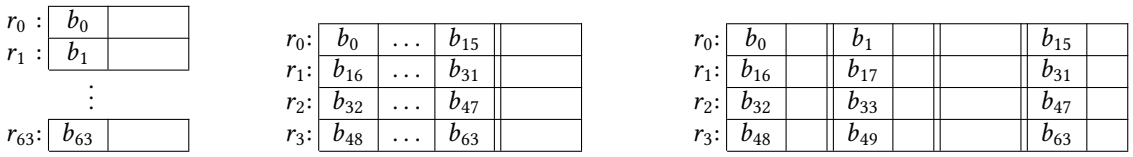
The Usuba language design is driven by a combination of algorithmic and hardware-specific constraints. Algorithmically, implementing a block cipher requires some discipline to achieve high throughput and avoid timing attacks. To get high throughput, we focus exclusively on parallel modes of operation such as counter mode (CTR, most widely used). This excludes feedback modes, such as output feedback (OFB), delivering much lower throughput in software. We translate this constraint by forbidding feedback loops in our designs. An Usuba program can be understood as a stateless combinational circuit, parallelizable by design.

Processing independent blocks in parallel forbids us from using any form of conditional execution. As is standard in GPU programming, we resort to masking for conditionals, effectively wiring the two results to a multiplexer. Usuba programs are thus immune to timing attacks based on branch prediction, by construction. We are further able to guarantee the absence of cache effects by forbidding data-dependent access to memory. For example, we support lookup tables but compile them to Boolean circuits.

Finally, Usuba must be expressive enough to describe hardware-oriented ciphers (such as DES or Trivium, which are specified in terms of Boolean operations) as well as software-oriented ciphers specified in terms of affine transformations (such as AES and, more generally, ciphers exploiting maximum distance separable matrices [28]). To account for the former, Usuba provides abstractions to manipulate vectors, such as extracting a single bit as well as splitting or combining vectors. To account for the latter, 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, similar in spirit to APL [35], 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 addition is a poor candidate for bitslicing: this addition turns into a genuine full-adder 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 quantities ranging from bytes to zwords, 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.

[1] Available from https://github.com/DadaIsCrazy/usuba



(a) Bitslicing: $u_D 1 \times 64$ **(b)** Vertical slicing: $u_V 16 \times 4$ **(c)** Horizontal slicing: $u_H 16 \times 4$

Figure 2. Data layouts

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".

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 $u_D m \times n$ 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 ($D = V$) or horizontal ($D = H$) SIMD instructions. A single m-bit value is thus typed $u_D m \times 1$, abbreviated $u_D m$.

This notation allows us to unambiguously specify the data layout of the blocks processed by the cipher. Consider the 64-bit input block $b_0, b_1, \ldots, b_{63}$ of the Rectangle cipher, targeting 128-bit SSE registers. Bitsliced Rectangle manipulates blocks of type $u_V 1 \times 64$, *i.e.* each bit is dispatched to an individual register (Figure 2a). A 16-bit vertical slicing of Rectangle manipulates blocks of type $u_V 16 \times 4$, *i.e.* each one of the 4 sub-blocks is dispatched to the first 16-bit element of 4 registers (Figure 2b). Horizontal slicing has type $u_H 16 \times 4$, *i.e.* each 16-bit word is horizontally spread across each of the 16 packed elements of 4 SIMD registers (Figure 2c). Note that directions collapse in the case of bitslicing, *i.e.* $u_V 1 \times 64 \cong u_H 1 \times 64$. Both cases amount to the same layout, or put otherwise: vertical and horizontal slicing are two (orthogonal!) generalizations of bitslicing.

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, a vertical, 16-bit SIMD addition (`vpaddw`) in vertical slicing will amount to performing an addition on 8 blocks in parallel. We only need to specify the treatment of a single slice, Usubac then automatically generates code that maximizes register usage. Transposing a sequence of input blocks in a form suitable for parallel processing is fully determined by its type. Usubac automatically synthesizes this function and we evaluate its throughput in Section 4.3.

2.2 Syntax & Semantics

In and of itself, Usuba is an unsurprising dataflow language [1, 25]. We introduce its syntax and semantics by way of an example: our implementation of the Rectangle cipher (Figure 1), whose reference implementation consists in 115 lines of C++. In the following, we shall leave types and typing aside, coming back to this point in Section 2.3.

An Usuba program is composed of a totally ordered set of nodes (here, `SubColumn`, `ShiftRows` and `Rectangle`). The last node plays the role of the `main` entry point: it will be compiled to a C function. A node typically 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 (here, `SubColumn`), useful for specifying S-boxes. Conceptually, a lookup table is an array: the 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 [67]– 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 [23, 44, 61, 70]. 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 with the following node:

```
node SubColumn (a:v4) returns (b:v4)
vars
    t1:v1, t2:v1, t3:v1, t4:v1, t5:v1, t6:v1,
    t7:v1, t8:v1, t9:v1, t10:v1, t11:v1, 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
```

Syntactically, our treatment of vectors stems from the work on hardware synthesis of synchronous dataflow programs [69]. In effect, we are merely describing "software circuits". Many programming idioms found in hardware synthesis, which would be extremely inefficient in general, can be efficiently implemented in our setting. In bitsliced mode, we can for example permute the 4^{th} and 13^{th} bit of a 16-bit word at no run-time cost: this amounts to *statically* swapping the names of the registers holding the 4^{th} and 13^{th} bits throughout the rest of the program.

Given a vector x of size n (thus, of type $\tau[n]$), we can obtain the element $x[k]$ at position $0 \leq k < n$ and the consecutive elements $x[k..l]$ in the range $0 \leq k < l < n$. 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 $y = u[n..m+n]$.

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 inter-dependent equations. However, in practice, the `Usubac` compiler preserves this structure in its pipeline and may generate imperative loops performing destructive updates (Section 3.2).

Some ciphers (*e.g.* Chacha20, Serpent) are defined in a very 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 form (SSA). To simplify such codes, Usuba provides an "imperative assignment" operator `x := 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.

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 of vectors (left `<<` and right `>>`) and rotations of vectors (left `<<<` and right `>>>`). Their availability and exact semantics (esp., run-time cost) depend on the slicing mode and the target architecture, as shown next.

2.3 Type System

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 $\mathsf{u}_D m \times n$, which documents the layout of blocks coming in and out of the cipher.

We also have general vectors whose types are $\tau[n]$, for any type τ. Now, consider the key schedule of vsliced Rectangle: it is presented as an object `key` of type $\mathsf{u}_V 16 \times 4[26]$, which is 26 quadruples of 16-bit words. To obtain the key at round 23, we thus write $key[23]$ while obtaining the 3^{rd} word of that key one must write $key[23][3]$. The notation $\mathsf{u}_D m \times 4[26]$ indicates that accesses must be performed in column-major order, *i.e.* as if we were accessing an object of type $\mathsf{u}_D m[26][4]$ (following C convention). In fact, because vectors are kept flat, the types $\mathsf{u}_D m[26][4]$ and $\mathsf{u}_D m \times 4[26]$ are actually equivalent from the typechecker's standpoint, the compiler collapsing both types to $\mathsf{u}_D m[104]$. This apparent redundancy is explained by the fact that types serve two purposes. In the surface language, matricial types ($\mathsf{u}_D m \times 4$) document the data layout and its SIMDization. In the target language, the matricial structure is irrelevant: an object of type $\mathsf{u}_D m \times 4[26]$ supports exactly the same operations as an object of type $\mathsf{u}_D m[26][4]$. Surface types are thus normalized, after type-checking, into *distilled* types.

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 $\mathsf{u}_D 1$, $\mathsf{u}_D 8$, *etc.* This completes the grammar of distilled types, including type parameters in word size and direction:

$$
\begin{array}{rcll}
\mathsf{n} & \in & \mathbb{N} & \text{(integers)} \\
\langle \tau \rangle & ::= & & \text{(types)} \\
& | & \mathsf{u}_{\langle \mathrm{D} \rangle} \langle \mathrm{m} \rangle & \text{(base type)} \\
& | & \tau[\mathsf{n}] & \text{(vectors)} \\
& | & \mathsf{nat} & \text{(constants)}
\end{array}
$$

$$
\begin{array}{rcll}
\langle \mathrm{m} \rangle & ::= & & \text{(size)} \\
& | & \text{`}m & \text{(parameter)} \\
& | & \mathsf{n} & \text{(fixed size)}
\end{array}
\qquad
\begin{array}{rcll}
\langle \mathrm{D} \rangle & ::= & & \text{(direction)} \\
& | & \text{`}D & \text{(parameter)} \\
& | & \mathsf{V} & \text{(vertical)} \\
& | & \mathsf{H} & \text{(horizontal)}
\end{array}
$$

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 $\mathsf{b}n$ and $\mathsf{u}m$ for the type $\mathsf{u}_{\text{`}D} 1 \times n$ and, respectively, $\mathsf{u}_{\text{`}D} m$ where `D is the direction parameter in the nearest scope. Similarly, we write $\mathsf{v}n$ for $\mathsf{u}_{\text{`}D}\text{`}m \times n$ when `m is the nearest word size parameter.

Ad-hoc polymorphism. In reality, very few programs are defined for *any* word size or *any* direction. Only Boolean circuits are direction polymorphic, bitwise logical operations applying uniformly. Also, no program is absolutely parametric in the word size: we can only compute up to the register size of the underlying architecture.

Table 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 $m \in [1, 64]$	$\geq$ x86-64	and, or, *etc.* (1 instr.)
	$\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [65, 128]$	$\geq$ SSE	
	$\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [129, 256]$	$\geq$ AVX	
	$\mathsf{Logic}(\mathsf{u}_D m)$ for $m \in [257, 512]$	$\geq$ AVX512	
$\mathsf{Arith}(\tau)$	$\mathsf{Arith}(\tau) \Rightarrow \mathsf{Arith}(\tau[n])$ for $n \in \mathbb{N}$	all	homomorphic application (n instr.)
	$\mathsf{Arith}(\mathsf{u}_V 8)$ $\mathsf{Arith}(\mathsf{u}_V 16)$ $\mathsf{Arith}(\mathsf{u}_V 32)$	$\geq$ SSE	vpadd, vpsub, *etc.* (1 instr.)
	$\mathsf{Arith}(\mathsf{u}_V 64)$	$\geq$ AVX2	
$\mathsf{Shift}(\tau)$	$\mathsf{Shift}(\tau[n])$ for $n \in \mathbb{N}$	all	variable renaming (0 instr.)
	$\mathsf{Shift}(\mathsf{u}_V \text{`}m)$, $\mathsf{Shift}(\mathsf{u}_H \text{`}m) \Rightarrow \mathsf{Shift}(\mathsf{u}_D \text{`}m)$	all	depends of instance
	$\mathsf{Shift}(\mathsf{u}_V 16)$ $\mathsf{Shift}(\mathsf{u}_V 32)$	$\geq$ SSE	vpsrl/vpsll ($\leq$ 3 instr.)
	$\mathsf{Shift}(\mathsf{u}_V 64)$	$\geq$ AVX2	
	$\mathsf{Shift}(\mathsf{u}_H 2)$ $\mathsf{Shift}(\mathsf{u}_H 4)$ $\mathsf{Shift}(\mathsf{u}_H 8)$ $\mathsf{Shift}(\mathsf{u}_H 16)$	$\geq$ SSE	vpshuf (1 instr.)
	$\mathsf{Shift}(\mathsf{u}_H 32)$ $\mathsf{Shift}(\mathsf{u}_H 64)$	$\geq$ AVX512	

To capture these invariants, we introduce a form of bounded polymorphism through type-classes [71]. 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 three type-classes for logical, arithmetic and shift/rotate operations:

$\mathsf{Logic}(\tau)$:
$\&: \tau \to \tau \to \tau$
$|: \tau \to \tau \to \tau$
$\hat{}: \tau \to \tau \to \tau$
$\sim: \tau \to \tau$

$\mathsf{Arith}(\tau)$:
$+: \tau \to \tau \to \tau$
$*: \tau \to \tau \to \tau$
$-: \tau \to \tau \to \tau$

$\mathsf{Shift}(\tau)$:
$\gg: \tau \to \mathsf{nat} \to \tau$
$\ll: \tau \to \mathsf{nat} \to \tau$
$\gg>: \tau \to \mathsf{nat} \to \tau$
$\ll<: \tau \to \mathsf{nat} \to \tau$

The implementations of these classes depend on the type considered and the target architecture. For instance, arithmetic on 13-bit words is impossible, even in vertical mode. The generated code also depends on a combination of types and target architecture. For instance, shifting a vector amounts to renaming registers whereas shifting 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.

Our type-class mechanism is not user-extensible. The list of all the possible type-classes instances is summarized in Table 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.

Type-checking. Despite its unusual combination of features, Usuba's type system is rather unsurprising. First, it is applied to a first-order and explicitly-typed language, thus requiring no type inference. Second, vectors – while perhaps intimidating – can only take statically-known sizes: we are far away from the realm of dependent types, merely touching upon phantom types techniques. The combination of prenex polymorphism and type-classes is well understood. We therefore refrain from further expounding our type system.

As a result of these features, our generic implementation of Rectangle type-checks as-is when targeting any instruction set beyond SSE, meaning that it can be sliced vertically or horizontally. In Section 3.1, we shall describe the mechanisms at our disposal to produce vertically or horizontally sliced C code as well as for extracting a bitsliced implementation.

3 The Usubac Compiler

The Usubac compiler consists of two phases: the front-end (Section 3.1), whose role is to distill Usuba's high-level constructs into a minimal core language, and the back-end (Section 3.2), which performs optimizations over the core language. The core language, called Usuba0, amounts to dataflow graphs of Boolean and arithmetic operators. In its final pass, the back-end translate Usuba0 to C code with intrinsics.

3.1 Front-end

The front-end extends the usual compilation pipeline of synchronous dataflow languages [16, 69], namely normalization and scheduling, with domain-specific transformations. Normalization enforces that, aside from nodes, equations are 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 back-end.

Several features of Usuba requires specific treatment by the front-end. First, the language offers domain-specific syntax for specifying cryptographic constructs such as lookup tables or permutations. We boil these constructs down to Boolean circuits. Second, the language provides a limited form of parametric polymorphism while offering an overloading mechanism for logical and arithmetic operations based on ad-hoc polymorphism. We produce monomorphic code, with types and operations being expanded to machine-baked instructions for the target architecture and parallelism discipline.

Monomorphization. The parametric polymorphism offered by Usuba enables the concise description of size-generic or direction-generic algorithms. The choice of execution model can thus be postponed to compile-time. However, the code generated by Usuba must be monomorphic. Striving for performance, we cannot afford to manipulate boxed values at run-time. Also, the underlying SIMD architecture does not allow switching between a vertical or horizontal style of SIMD operations. The `Usubac` compiler provides flags `-H` and `-V` to monomorphize the main node to the corresponding horizontal or vertical direction, while the `-w` m flag monomorphizes the atomic word size to the given positive integer m.

Crucially, we do not implement ad-hoc polymorphism by dictionary passing [71], which would imply a run-time overhead, but resort to static monomorphization [36]. Provided that type-checking was successful, we know precisely which operator instance exists at the given type and macro-expand it.

Flattening: from m-slicing to bitslicing. We may also want to flatten an 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 micro-controllers) do not offer vectorized instruction sets at all. Also, bitsliced algorithms serve as the basis for hardening software implementations against fault attacks [45, 64]. To account for these use-cases, Usuba 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 $\mathsf{u}_D m \times n$ into the type $\mathsf{b}m[n]$. Note that the vectorial direction of the source is irrelevant: it will be collapsed 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, therefore, type-class resolution) succeeds, in which case we have obtained the desired bitsliced implementation, or type-class resolution fails, meaning that the given program exploits operators that are not available in bitsliced form. For instance, short of providing an arithmetic instance on $\mathsf{b}8$, we will not be able to bitslice an algorithm relying on addition on $\mathsf{u}_V 8$.

Once again, we rely on ad-hoc polymorphism to capture (in a single conceptual framework) the fact that a given program may or may not be amenable to bitslicing. The user is thus presented with a meaningful type error, which points out exactly which operator is incompatible with (efficient) bitslicing.

Usuba0. The front-end produces a monomorphic dataflow graph whose nodes correspond to logical and arithmetic operations provided by the target instruction set. This strict subset of the Usuba language is called Usuba0. It translates almost directly to C: in principle, we only need to schedule the code (as described in the following Section) after which each equation turns into a variable assignment of an integer type. Nodes are translated to function definitions whereas node calls translate to function calls. However, the Usuba0 code produced by the front-end is subjected to several optimizations before delivery to a C compiler. We describe these transformations in the following Section.

3.2 Back-end

The back-end exclusively manipulates Usuba0 code, taking advantage of referential transparency as well as the fact that we are dealing with a non Turing-complete language. Optimizations carried out at this level are easier to write but also more precise. The back-end is complementary with the optimizations offered by C compilers. There is a significant semantic gap between the user intents expressed in Usuba and their straightforward sliced translations in C. This results in missed optimization opportunities for the C compiler, of which we shall study a few examples in the following. The back-end authoritatively performs these optimizations at the level of Usuba0.

Interleaving. A first optimization consists in interleaving several executions of the program. Usuba allows us to systematize this folklore programming trick, popularized by Matsui [49]: for a cipher using a small number of registers (for example, strictly below 8 general-purpose registers on Intel), we can increase its instruction-level parallelism (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 parallel programming framework: we can implement it by a straightforward Usuba0-to-Usuba0 translation.

The interleaving heuristic itself is retrospectively straightforward: the number of interleaved instances is set to be the number of CPU registers available on the target architecture divided by the maximal number of live registers in the given algorithm. For example, Serpent and Rectangle use respectively 8 and 7 AVX registers at most, which drives the compiler to pick an interleaving factor of 2 when compiling. Choosing a larger factor would induce spilling, which is highly detrimental to performance.

A second design decision concerns the size of the independent code blocks to be interleaved. We have empirically observed that we can adopt a coarse-grained approach: we chose to alternate between blocks of 10 equations from each of the interleaved instances. The scheduling performed by Usubac and the instruction scheduling later performed by C compilers will do an excellent job at taking advantage of the resulting ILP.

On Serpent, we observe that the throughput of 2 interleaved ciphers is 21.75% higher than the throughput of a single cipher, while increasing the code size by 29.3%. Similarly for Rectangle, the throughput increases by 27.62% at the expense of a 19.2% increase in code size.

Inlining. The decision of inlining nodes is partly justified by the usual reasoning applied by C compilers: a function call implies a significant overhead that, for very frequently called functions (such as S-boxes, in our case) makes it worth the increase in code size. In fact, all the compilers we have tested (excepted GCC in some situations) are able to spot the fact that S-boxes, for example, benefit from being inlined. Usubac is only a helping hand here, making sure that the user does not observe a performance regression because of what could only be characterized as a bug in the C compiler's heuristics.

Bitslicing, however, sends the inlining heuristics of C compilers astray. A bitsliced node compiles to a C function taking hundreds of variables as inputs and outputs. For instance, the round function in 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 but also a growth in code size. C compilers naturally avoid inlining this function because its code is quite large, missing the fact that calling it actually becomes a bottleneck.

On DES, inlining results in a 44.8% improvement in throughput while actually *reducing* code size by 9.1%. Similarly, a bitsliced implementation of AES (automatically obtained from the more efficient hsliced version) is 24.24% more efficient with inlining at the expense of a 24.8% increase in code size. Because the AES round function is significantly larger than its number of input variables, we notice an increase of code size. However, increasing code size is hardly a performance issue in our setting: our code is executed in a straight-line sequence, with few to no control flow instructions. As a result, whatever big the resulting binary is, instruction prefetch allows us to amortize the few cache misses over the whole cache lines.

Besides, inlining offers more opportunities for scheduling. For instance, symmetric cipher are usually composed of a sequence of 10 to 20 applications of a single round function. Inlining this function enables Usubac to schedule instructions across rounds, so as to increase ILP.

Algorithm 1 Bitsliced code scheduling

```
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 next
6:          schedule funCall next
7:          for each variable v defined by funCall do
8:              for each equation eqn of prog using v do
9:                  if eqn is ready to be scheduled then
10:                     schedule eqn next
```

Scheduling bitsliced code. Our scheduling algorithm differs significantly depending on whether we are scheduling bitsliced or *m*-sliced code. In a bitsliced program, the single, major bottleneck is register pressure: a significant portion of the execution time is spent spilling registers to and from the stack. On DES, about 1 in 5 instructions deals with spilling. The role of scheduling is therefore to minimize register pressure as much as possible.

One might expect the C compiler to already minimize register pressure. However, it is well known that combining register allocation and instruction scheduling is NP-hard [55], and modern C compilers use heuristics to get the best possible approximations in most cases. In the bistlicing setting, where hundreds of variables are alive at the same time, those heuristics often prove to be inefficient, and miss opportunities to lower the spilling.

Our scheduling algorithm (Algorithm 1) focuses on reducing the live ranges of function calls arguments and return values –regardless of whether those functions will be inlined or not–. The reasonning behind it is as follows. According to

the x86-64 calling conventions, the first 8 arguments of function calls are passed in registers. It is therefore profitable to schedule the instructions computing those arguments close to the function call, thus removing the need to spill them only to reload them later into registers. As for return values, they are returned by reference in a structure (since x86-64 calling conventions do not allow functions to return more than one value), which amounts to having spilled them already. However, by moving instructions that uses them right after the function call, we increase the potential benefits of inlining: the return values will not need to be spilled but instead will be allowed to stay in registers as their live ranges will now be much shorter.

On bitsliced DES, combining scheduling and inlining increases throughput by 6.77% compared to the inlined code and reduces code size by 9.9% whereas, on bitsliced AES, throughput is increased by 2.49% and code size reduced by 3.4%. This witnesses the fact that the scheduling performed by `Usubac` is able to reduce unnecessary spilling, which was not spotted by C compilers. Overall, combining inlining and scheduling results in a net 45.8% increase in throughput compared to baseline. Similarly, on bitsliced AES, throughput is globally improved by 26.22%.

Scheduling m-sliced code. Unlike bitsliced code, m-sliced programs have much lower register pressure. Spilling is less of an issue, the latency of the few resulting load and store operations being hidden by the CPU pipeline. Instead, the challenge consists in being able to saturate the parallel CPU execution units. To do so, one must increase ILP, taking into account the availability of specialized execution units. For instance, hsliced code will rely on SIMD shuffle instructions (`vpshufb`) that can be executed on a single execution unit on current Skylake architecture. Performing a sequence of shuffles is extremely detrimental to performance: the execution of the shuffles (and their respective dependencies) become serialized, bottlenecking the dedicated execution unit.

One might think that the out-of-order nature of modern CPU would alleviate this issue, allowing the processor to execute independent instructions ahead in the execution streams. C compilers (such as `ICC` and `Clang`) seems to adopt this policy: groups of shuffle in the source code will be scheduled as-is in the resulting assembly. Only `GCC` statically schedules independent (for example, arithmetic) instructions in-between shuffles in the generated assembly. However, due to the bulky nature of sliced code, we observe that the reservation station is quickly saturated, preventing actual out-of-order execution.

We statically increase ILP in `Usubac` by maintaining a look-behind window of the previous 16 instructions (which corresponds to the maximal number of registers available on Intel platforms without AVX512) while scheduling. To schedule an instruction, we search among the remaining instructions one with no data hazard with the instructions in the look-behind window, and that would execute on a different execution unit. If no such instruction can be found, we reduce the size of the look-behind window, and, utlimately just schedule any instruction that is ready.

This scheduling algorithm increased the throughput of hsliced AES by 2.43%, and of vsliced Chacha20 by 9.09%. Inspecting the generated assembly, we notice that, in both cases, the impact of data hazards have been minimized by reorganizing computations within each rounds.

Inlining is instrumental in improving the quality of scheduling in this setting too: it allows instructions to flow freely across node calls. For instance, a round of Chacha20 is specified in terms of 2 half-rounds, which we naturally implement with two nodes. Thanks to inlining, we can afford to define a node, hence increasing readability, knowing that, performance-wise, it is transparent.

Whereas inlining allows scheduling to improve the code quality within a single round, symmetric ciphers usually combine ten or more iterations of a given round. For instance, Chacha20 performs 10 iterations of a double round, and AES performs between 10 and 14 rounds. In Usuba, these iterations are naturally expressed with a grouped definition, using the `forall` declaration.

`Usubac` can choose between expanding the `forall` declaration – unrolling the loop at the expense of code size – or to translate it into a C `for` loop. The loop bounds being statically known, the C compiler could also decide to fully unroll this loop. However, the size of the loop body being significant, compilers prefer to avoid unrolling it, a sound decision in general since code locality ought to be preserved. Since we have perfect locality anyway, code size is not an issue in our setting. We may thus profitably unroll all the rounds into a single straight-line program: scheduling is thus able to re-order instructions across distinct encryption rounds. On AES (resp. Chacha20), this yields a 3.22% (resp. 3.63%) speedup compared to an implementation performing intra-round scheduling only, at the expense of a 31.90% (resp. 19.40%) increase in code size.

Remark. It would be tempting to integrate the above optimizations as a domain-specific backend for, say, LLVM [46]. We conjecture that the resulting compiler would not be radically different nor significantly faster than Usuba. First, our optimizations are complementary to those performed by the C compiler (e.g., inlining). They would be expressed almost as-is over LLVM-IR. Second, Table 2 will show that there is no silver bullet among compilers: we are currently free to pick the absolute best compiler, even if closed source. Third, going the extra mile and targeting C simplifies integration in existing code bases, as witnessed by the incorporation of the C files produced by HACL* [73] into the Firefox and Wireguard projects. This allows the cryptographical primitives to potentially outlive the DSL. In the following, we evaluate the performance of the code generated by our compiler.

Table 2. Optimal configurations

Cipher	Mode	Compiler	Inlining	Unrolling	Interleaving	Scheduling
DES	bitslice	`Clang`	✓	✓		✓
AES	bitslice	`Clang`	✓	✓		✓
	hslice	`Clang`	✓	✓		✓
Rectangle	bitslice	`ICC`	✓	✓		✓
	hslice	`GCC`			✓	✓
	vslice	`Clang`			✓	✓
Chacha20	vslice	`ICC`	✓	✓		✓
Serpent	vslice	`Clang`		✓	✓	

4 Experimental Results

We use the Supercop [13] benchmarking framework to measure the throughput of reference implementations of 3 ciphers (AES, Chacha20 and Serpent) in CTR mode and 2 ciphers in ECB mode (DES and Rectangle). To provide a fair comparison, focused solely on the implementation of the cryptographic primitive, we benchmark our implementation following the same key schedule and CTR increment algorithms. This puts us at a slight disadvantage over some implementations that fuse the runtime into the primitive, so as to further speed up computation. Unlike the hand-tuned, assembly implementations, ours are bound to respect x86 calling conventions, implying a small overhead. It is future work to integrate the cryptographic runtime as part of our model, so as to be able to optimize it as well. We have conducted our benchmarks on a Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz machine running Ubuntu 16.04.5 LTS (Xenial Xerus) with Clang 7.0.0, GCC 8.1.0 and ICC 18.0.2.

By design, our sliced implementations are constant-time: no data-dependent branches or access to memory are ever made. We nonetheless experimentally validated this claim using the `dudect` [68] tool, which statistically checks that the execution time of our primitive is independent of its input. All our implementations have received a green flag, unsurprisingly.

In Table 2, we provide a high-level view of our benchmark results: for each cipher and each of its supported slicing mode, we indicate the combination of C compiler and `Usubac` features (Section 3.2) that led to the code delivering the highest throughput. We observe that no C compiler stands out as producing the most efficient code in all situations. The same goes for `Usubac` optimizations: their potency depends on the algorithm as well as its slicing mode. The combination of inlining, unrolling and scheduling is a requirement on bitsliced code. Interleaving is generally useful in *m*-sliced modes (vertical or horizontal), under the proviso that the cipher uses a small number of register (which is not the case for Chacha20 or AES).

4.1 Benchmarks

In Table 3, we report the benchmark of our Usuba implementations of 5 ciphers against the most efficient publicly available implementations. We provide the SLOC count of the cipher primitive (*i.e.* excluding key schedule and counter management) for every implementation. Usuba codes 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. Latencies are shown for illustrative purposes, and are roughly the same for Usuba and reference implementations. Unsurprisingly, bitslicing exhibit higher latencies than *m*-slicing since more blocks must be loaded before computation can proceed.

Our first benchmark is DES [57] in Electronic Codebook (ECB) mode [58]. Needless to say, DES in general and its ECB mode in particular are 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, which results in publicly-available, optimized C implementations in bitsliced mode for 64-bit general-purpose registers. Our implementation, using the same S-box, is faster, thanks in large part to our careful treatment of register spilling.

Our second benchmark is AES [59], whose fastest hand-tuned SSSE3 and AVX implementations are hsliced. 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 because the latter fuse 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. If we manually modify our implementation to avoid this indirection (and, as a result, violate the calling conventions), our AVX implementation is on par with the corresponding hand-tuned one (*i.e.* within the margin of error).

Table 3. Comparison between Usuba code & reference implementations

Mode	Cipher	Instr. set	Code size (SLOC)		Throughput (cycles/byte)		Speedup	Latency (cycles)
			Ref.	Usuba	Ref.	Usuba		Usuba
bitslicing	DES [43, 44]	x86-64	1053	655	12.01	11.47	+4.50%	5872
16-hslicing	AES [37, 38]	SSSE3	272	218	7.77	7.92	-1.93%	1013
16-hslicing	AES [39, 41]	AVX	339	218	5.59	5.71	-2.15%	730
32-vslicing	Chacha20 [12, 27]	AVX2	20	24	1.03	1.02	+0.97%	522
32-vslicing	Chacha20 [12, 53]	AVX	134	24	2.09	2.07	+0.96%	529
32-vslicing	Chacha20 [12, 53]	SSSE3	134	24	2.72	2.31	+15.07%	591
32-vslicing	Chacha20 [12]	x86-64	26	24	5.64	5.65	-0.18%	361
32-vslicing	Serpent [18, 33]	AVX2	300	214	4.33	4.53	-4.62%	579
32-vslicing	Serpent [18, 34]	AVX	300	214	8.36	8.66	-3.59%	554
32-vslicing	Serpent [18, 40]	SSE2	300	214	11.48	11.29	+1.66%	722
32-vslicing	Serpent [18, 62]	x86-64	300	214	30.37	25.78	+15.11%	412
16-vslicing	Rectangle [72][2]	AVX2	115	31	2.45	2.10	+14.29%	268
16-vslicing	Rectangle [72]	AVX	115	31	4.92	4.21	+14.43%	269
16-vslicing	Rectangle [72]	SSE4.2	115	31	14.51	11.18	+22.95%	715
16-vslicing	Rectangle [72]	x86-64	115	31	28.61	25.88	+9.54%	207

Chacha20 [12] has been designed to be efficiently implementable in software, with an eye toward SIMD. The fastest implementation to date is vsliced. Key to our success has been our ability to do instruction scheduling within each round as well as across successive rounds. Chacha20's round is composed of 4 updates of 4 variables, which, done sequentially, suffer from data hazards. Our scheduling algorithm is able to effectively interleave those updates, thus removing any data hazard.

Serpent [18] was another contender to the AES challenge. It can be implemented with a small number of 64-bit registers (less than 8). Its fastest implementation is written in assembly, exploiting the AVX2 instruction set. It interleaves 2 parallel executions of the cipher. While `Usubac` generates C code using this optimization, the assembly produced by `Clang` contains about 25% more memory `MOV`s than the reference implementation, thus making us fall a few percents behind. On general purpose 32-bit registers, however, the reference C code does not use interleaving, allowing us to be about 15% faster.

Rectangle [72] is a lightweight cipher, designed for relatively fast execution on micro-controllers. It only consumes a handful of registers. The reference implementation of Rectangle is manually vsliced in C++ and fails to take advantage of interleaving: as a result, our straightforward Usuba implementation easily outperforms the reference one.

4.2 Scalability

Figure 3 shows the scaling of our implementations on the main SIMD available on Intel: SSE (SSE4.2), AVX, AVX2 and AVX512. Those benchmarks were compiled using `ICC` 18.0.1, and executed on a Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz running a Scientific Linux 7.5 (Nitrogen).

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 $u_V16 \times 4$ costs 0.09 cycles/byte on AVX512 while the transposition of $u_H16 \times 4$ costs up to 10.76 cycles/byte on SSE.

One could expect the speedup to be linear in the size of registers, but the reality is more complex. Spilling wider registers puts more pressure on the L1 data-cache, leading to more frequent misses. To stay within thermal design power (TDP) envelope of the CPU, wider SIMD execution units (from AVX to AVX512) are throttled, meaning that the processor must perform time-consuming frequency reduction steps. AVX and AVX512 registers need tens of thousands warm-up cycles before being used, since they are not powered when no instruction uses them. SSE instructions take two operands and overwrite one to store the result, while AVX offer 3-operand non destructive instructions, thus reducing register spilling. Also, successive generations of SIMD instructions expand the number of registers (from 8 with SSE, 16 with AVX and 32 with AVX512), further reducing the need for spilling. The latency and throughput of most instructions differ from one micro-architecture to another and from one SIMD to another. For instance, up to 4 general purpose bitwise operations can be computed per cycle, and only 3 SSE/AVX. Finally, newer SIMD extensions tend to have more powerful instructions than older ones. For instance, AVX512 offers, among many useful features, a bitwise ternary logic instruction (`vpternlogq`) that computes any three-operand binary function at once. 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.

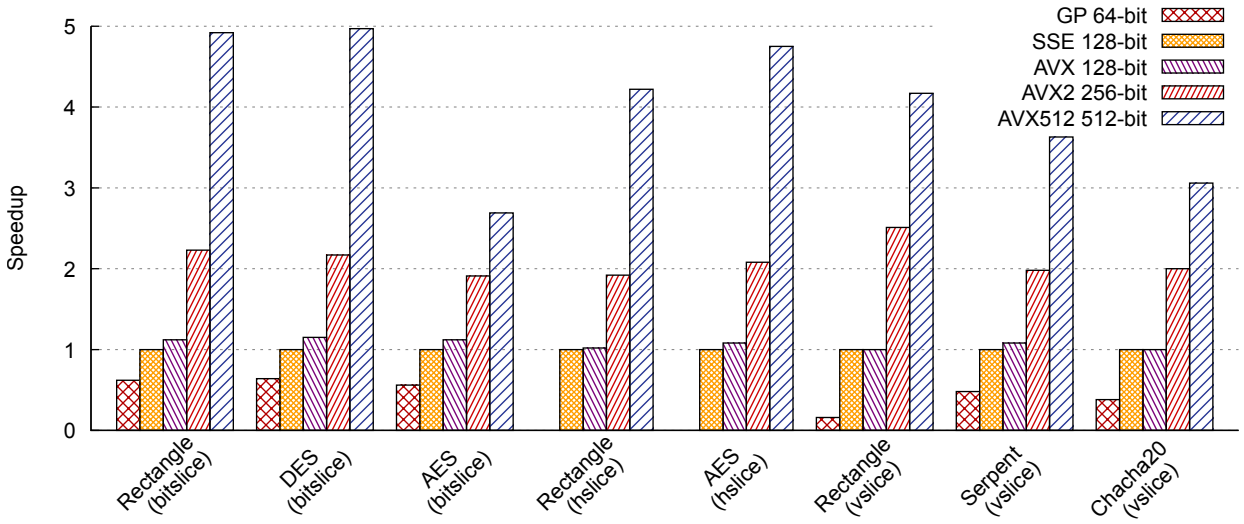


Figure 3. Scalability of SIMD compilation

Bitslice Rectangle and DES both scale nicely. As any bitsliced algorithm, they both suffer from a lot of spilling, and therefore benefit a lot from having more registers available: using AVX instructions instead of SSE allows to use non-destructive 3-operand instructions, therefore yielding a 10 to 15% speedup, without using wider registers. When doubling the word size by using 256-bit AVX2 registers instead of 128-bit, the speedup is unsurprisingly doubled. AVX512 benefits are twofold. First, they provide twice more registers than AVX2, thus reducing the spilling. Second, they offer a bitwise ternary logic instruction, which is exploited by `ICC` to fuse nested logical gates. Overall, this allows Rectangle and DES to run almost 5 times faster on AVX512 than on SSE. Bitslice AES, on the other hand, does not benefit from having wider registers, because it contains many more spilling and memory operations: about 6 times more than Rectangle and DES. Consequently, increasing the register size puts more pressure on the caches: while AES suffers from less than 1% misses in the L1 data cache when using general purpose 64-bit registers, that number grows to 6% on SSE, 14% on AVX2, and up to 50% on AVX512.

Overall, *m*-sliced programs are similar excepted when it comes to shuffle and *m*-bit arithmetic. As a result, they exhibit similar scaling behavior. They tend to use only a handful of variables, and do not suffer from spilling as much as bitsliced implementations, which explains why using AVX instructions yields only a marginal – if any – improvement in throughput. As in bitslicing, AVX2 registers allows the throughput to be doubled. AVX512 is particularly beneficial because of the aforementioned `vpternlogq` instruction. This translates into a good speedup on Rectangle and AES while the effect is reduced on Chacha20, which involves few Boolean operations.

4.3 Monomorphizations

As explained in Section 2.3, we can specialize our polymorphic implementation of Rectangle to all types of slicing and SIMD instruction set: for instance, on AVX2, the vsliced C implementation of Rectangle generated from the Usuba code will have the type

```
void Rectangle (__m256i plain[4], __m256i key[26][4],
                __m256i cipher[4])
```

while the bitsliced SSE implementation has type

```
void Rectangle (__m128i plain[64], __m128i key[26][64],
                __m128i cipher[64])
```

This allows us to present the first – to the best of our knowledge – comparison of vsliced, hsliced and bitsliced implementations of a single cipher (Figure 4).

Overall, vslicing is much more efficient than bitslicing and hslicing. One of the main reason is that it uses a very efficient transposition (about 15 instructions), as it only needs to transpose up to words of 16-bit, unlike in bitslicing where the transposition goes all the way down to single bits. Both *m*-slicing techniques use only a handful of registers, which explains why bitsliced implementations are slower than *m*-sliced ones.

Excluding transposition, vslicing and hslicing exhibits very similar performances. This is not surprising since both codes are similar: their only difference comes from the rotations of Rectangle, each of which is done with a shuffle in hslicing, and 2 shifts and a `or` in vslicing. However, shuffles can be executed only on Port 5 of the Skylake CPU, while shifts and `or`s can be executed on Ports 0, 1 and 5,

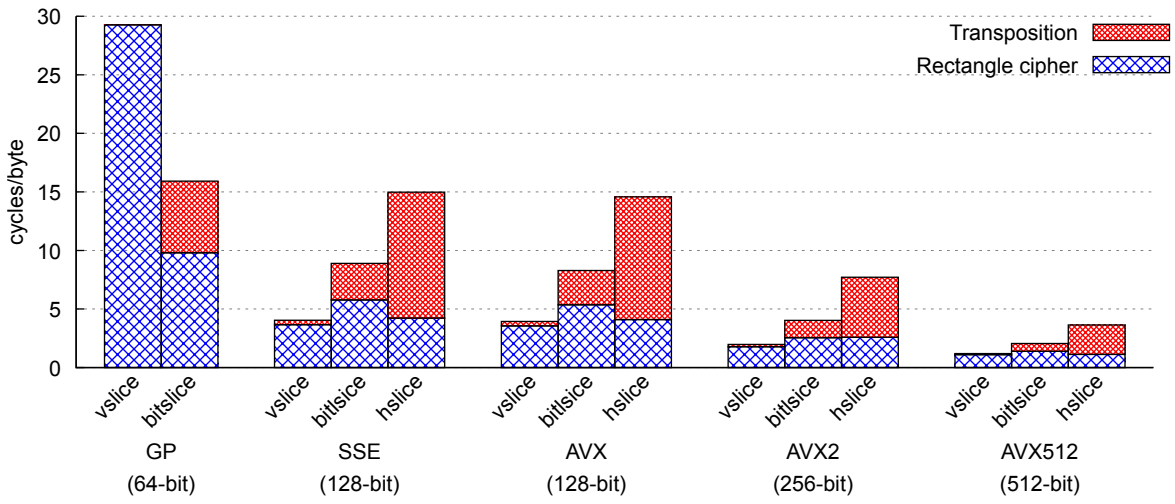


Figure 4. Monomorphizations of Rectangle

which means that both version of the rotations have the same throughput in isolation.

On 64-bit general purpose registers, bitslicing is actually faster than vslicing because the latter processes only one block at a time, as a consequence of the lack of x86-64 instruction to shift 4 16-bit words in a single 64-bit register.

5 Related Work

Bitslicing compilers. The idea of automating the generation of bitsliced primitives was developed by Pornin [67], who designed a proof-of-concept compiler called `bsc`. We borrow `bsc`'s syntax for lookup tables and permutations. As a prototype, it has been used to bitslice DES, for which it produced an excessively large C file (65000 variables for 65000 statements) exhibiting poor performance. To the best of our knowledge, this is the only attempt at mechanically deriving a bitsliced implementation from a high-level algorithmic description: some clues left in bitsliced reference implementations seem to suggest that cryptographers use scripting languages to aid in setting the basis of their programs but the grunge-work of connecting and optimizing the pieces is left to the programmer.

The previous version of Usuba also dealt with bitslicing only. Our earlier work [51] explored the impact of bitslicing on a broad range of SIMD architectures (from SSE to AVX512 but also Arm Neon and IBM AltiVec). Thanks to *m*-slicing, Usuba can be used to implement a broader range of ciphers while enabling new optimizations (m-sliced scheduling, interleaving). In the present work, we also formally account for the front-end (type system and type-class mechanism).

The `swarc` compiler of Fisher [29] can be understood as a slicing compiler of sort. SIMD within a register stems from an attempt at defining a programming model to unify the 14 vendor-specific SIMD instruction sets available at the turn of the millennium. By programming within this model, 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 would provide "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.

Data-parallel languages. The difficulty of writing efficient and future-proof SIMD programs has nonetheless persisted, hence the search for alternative programming model. The Single Program Multiple Data (SPMD) programming model has recently been suggested to address the frustration of C programmers trying to vectorize their inherently sequential programs [66]. Usuba takes a similar stance: to take full advantage of hardware-level parallelism, we must be able to describe the parallelism of our programs. Our scope is much narrower however: 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.

Our work bears some similarity (and is indeed culturally acclimated to) the work on Data Parallel Haskell [48, 65]. The combination of parametric polymorphism, ad-hoc overloading, and equational reasoning is common to both Haskell and Usuba. In principle, we conjecture that one could translate – with minor modifications – our Usuba programs into semantically equivalent Haskell ones. However, a Haskell program comes with a superfluous runtime system for our purposes, while the Turing completeness of the language would impede future efforts on automated verification.

Constant-time cryptography. Several, complementary tools have been proposed to guarantee that cryptographic primitives run in constant time. A first approach consists in verifying *a posteriori* that a given implementation is constant time [3]. Using a formal timing model of an abstract machine, one can use automated theorem provers to show that a program is safe within this model. Taking advantage of such a security verifier, which processes LLVM code, Cauligi et al. [26] side-step the C compiler, substituting it for a domain-specific language allowing the user to indicate whether an input is a secret or not. Thanks to a powerful type system, they can help programmer write constant-time code while the compiler exploits types to avoid leaking secret information through timing while carrying the annotations across to the LLVM code and thus use `ct-verif` [3] for verification.

Portable assemblers. Rather than fight against C compilers, cryptographers tend to develop their own assemblers, aiming at the same time at writing high-performance code while automating much of the boilerplate [11, 60]. The semantics of these "languages" remain largely undocumented, making for a poor foundation to carry verification work. Recent works have been trying to bridge that gap [2, 20] 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 compilers, 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. In the future, we would like to replace our dependency on C compilers by directly targeting one of these portable assembler.

Languages for cryptography. More generally, cryptographers with an inclination for verification have been prone to invest in domain-specific languages. This methodology is exemplified by Cryptol [47], whose primary goal is to support validation and verification of crypto-systems. Similarly, the $F^\star$ language is a key component of the Everest project [15], which provides a verified implementation of HTTPS. CAO [7, 54] is also an inspiration to Usuba: it has demonstrated that some crypto-systems can be described as operations in a given algebraic structure (the finite field $GF(2^8)$ for AES), using a compiler to produce somewhat efficient code from such a high-level description. Usuba aims at closing that gap too, starting from measurably efficient code and now working its way to higher levels of abstractions.

Cryptographical implementation tricks. Historically, bitslicing was born out of Biham's ingenuous software implementation of DES and it quickly became folklore amongst cryptographers, leading to it being used to speed-up other ciphers and applied on SIMD architectures. Interleaving [49] is another technique introduced in the context of the Camellia cipher, which we turn into a systematic program transformation in Usuba. Other techniques, such as counter-caching [63] or natively supporting Galois Fields arithmetics [23, 24], would be natural extensions of the present work.

6 Conclusion

We have presented Usuba, a synchronous dataflow programming language designed to write bitsliced code. Using parametric polymorphism, Usuba allows us to concisely and generically describe block ciphers, which can then be specialized to a given slicing mode and architecture to produce efficient C codes.

We implemented domain-specific optimizations, like interleaving and scheduling, and as a result, `Usubac` produces C code whose performance is similar to hand-tuned assembly. We backed up this claim by benchmarking our codes against some of the best hand-tuned bitsliced (DES), hsliced (AES) and vsliced (Serpent, Rectangle, Chacha20) implementations available.

Future work Usuba is restricted to describing combinational circuits. We would like to extend it with controlled form of stateful computation that would, by typing, be compatible with sliced compilation. For instance, Trivium [22] is a stateful stream cipher in which the bits of the state are only used 64 rounds after their definition. It can therefore be efficiently bitsliced on 64-bit registers.

Usuba currently focuses solely on the cryptographic primitive, at the exclusion of the cryptographic run-time (key schedule, CTR increment). Integrating these elements in Usuba, would enable further optimizations, like counter caching [14]: in CTR-mode, some ciphers, such as AES, exhibit low diffusion during individual rounds, some of the early computations of the cipher can be cached instead of being recomputed for every block, thus yielding a speedup of up to 15%.

Finally, bitslicing has been used to protect cryptographic codes against fault attacks, using temporal and spatial redundancy [45, 64] for instance. While protecting an implementation by hand is a tedious and error-prone task, we could add a front-end to Usuba to automatically add such counter-measures.

Acknowledgments

The original idea of a bitslicing compiler was suggested to us by Xavier Leroy, to whom we are deeply indebted. We are also grateful to Lionel Lacassagne for allowing us to borrow his AVX-512 machine to run our benchmarks. We would like to thank our shepherd, Ayal Zaks, the anonymous reviewers and the artifact reviewers for their thoughtful comments.

This work was partially funded by the Émergence(s) program of the City of Paris and the EDITE doctoral school.

References

[1] . 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (26 2009). https://doi.org/10.1109/IEEESTD.2009.4772740

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. 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*. 1807–1823. https://doi.org/10.1145/3133956.3134078

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[4] Gogul Balakrishnan and Thomas W. Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6 (2010), 23:1–23:84. https://doi.org/10.1145/1749608.1749612

[5] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. 2017. 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*. 321–345. https://doi.org/10.1007/978-3-319-66787-4_16

[6] Zhenzhen Bao, Peng Luo, and Dongdai Lin. 2015. 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*. 18–36. https://doi.org/10.1007/978-3-319-29814-6_3

[7] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. 2011. Type Checking Cryptography Implementations. In *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*. 316–334. https://doi.org/10.1007/978-3-642-29320-7_21

[8] Gilles Barthe. 2015. High-Assurance Cryptography: Cryptographic Software We Can Trust. *IEEE Security & Privacy* 13, 5 (2015), 86–89. https://doi.org/10.1109/MSP.2015.112

[9] Adnan Baysal and Sühap Sahin. 2015. RoadRunneR: A Small and Fast Bitslice Block Cipher for Low Cost 8-Bit Processors. In *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany, September 10-11, 2015, Revised Selected Papers*. 58–76. https://doi.org/10.1007/978-3-319-29078-2_4

[10] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming

[11] Daniel J. Bernstein. 2007. `qhasm` software package. https://cr.yp.to/qhasm.html

[12] Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20. In *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*. https://cr.yp.to/papers.html#chacha

[13] Daniel J. Bernstein and Tanja Lange (editors). 2018. Supercop: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. https://bench.cr.yp.to/supercop.html

[14] Daniel J. Bernstein and Peter Schwabe. 2008. 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*. 322–336. https://doi.org/10.1007/978-3-540-89754-5_25

[15] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

[16] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. 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*. 121–130. https://doi.org/10.1145/1375657.1375674

[17] Eli Biham. 1997. A fast new DES implementation in software. In *FSE*. https://doi.org/10.1007/BFb0052352

[18] Eli Biham, Ross J. Anderson, and Lars R. Knudsen. 1998. Serpent: A New Block Cipher Proposal. In *Fast Software Encryption, 5th International Workshop, FSE '98, Paris, France, March 23-25, 1998, Proceedings*. 222–238. https://doi.org/10.1007/3-540-69710-1_15

[19] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. 2007. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*. 450–466. https://doi.org/10.1007/978-3-540-74735-2_31

[20] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[21] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716. https://doi.org/10.1016/j.comnet.2005.01.010

[22] Christophe De Cannière. 2006. Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. In *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*. 171–186. https://doi.org/10.1007/11836810_13

[23] David Canright. 2005. 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*. 441–455. https://doi.org/10.1007/11545262_32

[24] David Canright and Dag Arne Osvik. 2009. A More Compact AES. In *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*. 157–169. https://doi.org/10.1007/978-3-642-05445-7_10

[25] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 1987)*. ACM Press, Munich, Germany, 178–188.

[26] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. 69–76. https://doi.org/10.1109/SecDev.2017.24

[27] Romain Dolbeau and Daniel J. Bernstein. 2014. chacha20 dolbeau amd64-avx2. https://bench.cr.yp.to/supercop.html

[28] Sébastien Duval and Gaëtan Leurent. 2018. MDS Matrices with Lightweight Circuits. *IACR Trans. Symmetric Cryptol.* 2018, 2 (2018), 48–78. https://doi.org/10.13154/tosc.v2018.i2.48-78

[29] Randall James Fisher. 2003. *General-purpose Simd Within a Register: Parallel Processing on Consumer Microprocessors*. Ph.D. Dissertation. West Lafayette, IN, USA. AAI3108343.

[30] Vinodh Gopal, Jim Guilford, Wajdi Feghali, Erdinc Ozturk, Gil Wolrich, and Martin Dixon. 2010. Processing Multiple Buffers in Parallel to Increase Performance on Intel Architecture Processors. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf

[31] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. 2014. LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*. 18–37. https://doi.org/10.1007/978-3-662-46706-0_2

[32] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (1988), 532–533. https://doi.org/10.1145/42411.42415

[33] Johannes Götzfried. 2012. Serpent AVX2. https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/avx2-16way-1

[34] Johannes Götzfried. 2012. Serpent AVX2. https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/avx-8way-1

[35] Kenneth E. Iverson. 1980. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (1980), 444–465. https://doi.org/10.1145/358896.358899

[36] Stefan Kaes. 1988. Parametric Overloading in Polymorphic Programming Languages. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*. 131–144. https://doi.org/10.1007/3-540-19027-9_9

[37] Emilia Käsper and Peter Schwabe. 2009. AES-CTR, non-constant-time key setup. https://cryptojedi.org/crypto/data/aes-ctr-128-const-intel64-20090611.tar.bz2

[38] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. *CHES* (2009). https://doi.org/10.1007/978-3-540-74735-2_9

[39] Jussi Kivilinna. 2011. Block Ciphers: Fast Implementations on x86-64 Architecture. https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/aes128ctr/avx

[40] Jussi Kivilinna. 2011. Serpent sse2 implementation. https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/sse2-8way

[41] Jussi Kivilinna. 2013. *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.

[42] Robert Könighofer. 2008. 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*. 187–202. https://doi.org/10.1007/978-3-540-79263-5_12

[43] Matthew Kwan. 2000. Bitslice DES. http://www.darkside.com.au/bitslice/

[44] Matthew Kwan. 2000. Reducing the Gate Count of Bitslice DES. *IACR Cryptology ePrint Archive* 2000 (2000), 51. http://eprint.iacr.org/2000/051

[45] Benjamin Lac, Anne Canteaut, Jacques J. A. Fournier, and Renaud Sirdey. 2018. 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*. 1–5. https://doi.org/10.1109/ISCAS.2018.8351693

[46] Chris Lattner and Vikram Adve. 2004. 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.

[47] J. R. Lewis and B. Martin. [n. d.]. Cryptol: high assurance, retargetable crypto development and validation, Vol. 2. 820–825.

[48] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2017. Exploiting vector instructions with generalized stream fusion. *Commun. ACM* 60, 5 (2017), 83–91. https://doi.org/10.1145/3060597

[49] Mitsuru Matsui. 2006. 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*. 341–358. https://doi.org/10.1007/11799313_22

[50] Mitsuru Matsui and Junko Nakajima. 2007. 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*. 121–134. https://doi.org/10.1007/978-3-540-74735-2_9

[51] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. 2018. 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*. 4:1–4:8. https://doi.org/10.1145/3178433.3178437

[52] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. 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*. 21–44. https://doi.org/10.1007/978-3-319-76953-0_2

[53] Andrew Moon. [n. d.]. chacha opt. https://github.com/floodyberry/chacha-opt

[54] Andrew Moss and Dan Page. 2010. Bridging the gap between symbolic and efficient AES implementations. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*. 101–110. https://doi.org/10.1145/1706356.1706376

[55] Rajeev Motwani, Krishna V Palem, Vivek Sarkar, and Salem Reyen. 1995. Combining register allocation and instruction scheduling. *Courant Institute, New York University* (1995).

[56] Samuel Neves and Jean-Philippe Aumasson. 2012. Implementing BLAKE with AVX, AVX2, and XOP. *IACR Cryptology ePrint Archive* 2012 (2012), 275. http://eprint.iacr.org/2012/275

[57] National Bureau of Standards. 1977. *Announcing the Data Encryption Standard*. Technical Report FIPS Publication 46. National Bureau of Standards.

[58] National Bureau of Standards. 1980. *DES modes of operation*. Technical Report FIPS Publication 81. National Bureau of Standards.

[59] National Bureau of Standards. 2001. *Announcing the Advanced Encryption Standard (AES)*. Technical Report. National Bureau of Standards.

[60] OpenSSL. 2019. perlasm. https://github.com/openssl/openssl/tree/master/crypto/perlasm

[61] Dag Arne Osvik. 2000. Speeding up Serpent. In *AES Candidate Conference*. 317–329.

[62] Dag Arne Osvik and Jussi Kivilinna. 2002. Linux serpent. https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/serpent128ctr/linux_c

[63] Jin Park and Dong Lee. 2018. FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 3 (Aug. 2018), 469–499. https://doi.org/10.13154/tches.v2018.i3.469-499

[64] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2016. 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*. 231–244. https://doi.org/10.1007/978-3-319-69453-5_13

[65] Simon L. Peyton Jones. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*. 138. https://doi.org/10.1007/978-3-540-89330-1_10

[66] Matt Pharr and William R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *InPar '12: Innovative Parallel Computing*. 1–13.

[67] Thomas Pornin. 2001. *Implantation et optimisation des primitives cryptographiques*. Ph.D. Dissertation. École Normale Supérieure. http://www.bolet.org/~pornin/2001-phd-pornin.pdf

[68] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is my code constant time?. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. 1697–1702. https://doi.org/10.23919/DATE.2017.7927267

[69] Frédéric Rocheteau. 1992. *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)*. Ph.D. Dissertation. Grenoble Institute of Technology, France. https://tel.archives-ouvertes.fr/tel-00342092

[70] Markus Ullrich, Christophe De Canniere, Sebastiaan Indesteege, Özgül Küçük, Nicky Mouha, and Bart Preneel. 2011. Finding optimal bitsliced implementations of 4× 4-bit S-boxes. In *SKEW 2011 Symmetric Key Encryption Workshop, Copenhagen, Denmark*. 16–17. http://skew2011.mat.dtu.dk/proceedings/Finding%20Optimal%20Bitsliced%20Implementations%20of%204%20to%204-bit%20S-boxes.pdf

[71] Philip Wadler and Stephen Blott. 1989. 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*. 60–76. https://doi.org/10.1145/75277.75283

[72] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. 2014. RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms. *IACR Cryptology ePrint Archive* 2014 (2014), 84. http://eprint.iacr.org/2014/084

[73] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. *IACR Cryptology ePrint Archive* 2017 (2017), 536. http://eprint.iacr.org/2017/536