MaskSIMD-lib: On the Performance Gap of a Generic C Optimized Assembly and Wide Vector Extensions for Masked Software with an Ascon-p test case

Dor Salomon and Itamar Levi

Bar-Ilan University, BIU, Israel {dor.salomon;itamar.levi}@biu.ac.il

Abstract. Efficient implementations of software masked designs constitute both an important goal and a significant challenge to Side Channel Analysis attack (SCA) security. In this paper we discuss the shortfall between generic C implementations and optimized (inline-) assembly versions while providing a large spectrum of efficient and generic masked implementations for any order, and demonstrate cryptographic algorithms and masking gadgets with reference to the state of the art. Our main goal is to show the prime performance gaps we can expect between different implementations and suggest how to harness the underlying hardware efficiently, a daunting task for various masking-orders or masking algorithm (multiplications, refreshing etc.).

This paper focuses on implementations targeting wide vector bitsliced designs such as the ISAP algorithm. We explore concrete instances of implementations utilizing processors enabled by wide-vector capability extensions of the AMD64 Instruction Set Architecture (ISA); namely, the SSE2/3/4.1, AVX-2 and AVX-512 Streaming Single Instruction Multiple Data (SIMD) extensions. These extensions mainly enable efficient memory level parallelism and provide a gradual reduction in computation-time as a function of the level of extension and the hardware support for instruction-level parallelism. For the first time we provide a complete open-source repository of such gadgets tailored for these extensions, various gadgets types and for all orders.

We evaluate the disparities between *generic* high-level language masking implementations for optimized (inline-) assembly and conventional single execution path data-path architectures such as the ARM architecture. We underscore the crucial trade-off between state storage in the data-memory as compared to keeping it in the register-file (RF). This relates specifically to masked designs, and is particularly difficult to resolve because it requires inline-assembly manipulations and is not natively supported by compilers. Moreover, as the masking order (d) increases and the state gets larger, there must be an increase in data memory read/write accesses for state handling since the RF is simply not large enough. This requires careful optimization which depends to a considerable extent on the underlying algorithm to implement.

We discuss how full utilization of SSE extensions is not always possible; i.e. when d is not a power of two, and pin-point the optimal d values and very sub-optimal values of d which aggressively under-utilize the hardware. More generally, this paper presents several different fully generic masked implementations for any order or multiple highly optimized (inline-) assembly instances which are quite generic (for a wide spectrum of ISAs and extensions), and provide very specific implementations targeting specific extensions. The goal is to promote open-source availability, research, improvement and implementations relating to SCA security and masked designs. The building blocks and methodologies provided here are portable and can be easily adapted to other algorithms.

Keywords: AVX · Countermeasures · Code-Size · Low-Cost · Masking · Side-Channel Analysis · Security Order · SIMD · SSE

Introduction

Side-channel protection by masking countermeasures has quadratic cost factors associated with the desired security level which are dominated by vector-multiplications [CGLS20, MMSS18, CS21, GM18, DFS15, BDMD+20]. Masking implementations are also quite expensive and complicated due to randomness handling (refreshes) and volume (generation) [BDMD+20, Pap18]. However, all inherent masking assumptions theoretically provide exponential security at "only" a polynomial (quadratic) cost.

Generic; i.e., high-level software implementations of masked algorithms provide portability and are designed to be hardware/processor-agnostic. However, these designs do not necessarily utilize the underlying hardware resources efficiently. Specifically, the memory-level parallelism (MLP) and instruction-level parallelism (ILP) with vectorized processors can be sub-optimal. These effects are amplified as the state size in masked algorithms increases. This paper provides an in-depth discussion of the shortfall between generic C implementations and optimized (inline-) assembly versions which optimally utilize MLP and ILP, although they require much more expertise. Our main focus is the performance gap in terms of cycle count, code size and randomness requirements of different implementations and flavors of masked designs at all masking orders (d), with different masking multiplications and refresh primitives, over a large spectrum of ISAs extensions. As an example we focus an efficient cryptographic sponge permutation targeting a wide vector bitsliced implementation, the Ascon-p [DEMS16], which is used by algorithms such as ISAP [DEM+20] and can be generalized to Keccak as used by SHA3 and other sponges. We also explore the differences between generic C masking implementations and optimized (inline-) assembly over conventional single execution path data-path architectures such as the ARM architecture.

One of the crucial features we explore is trading-off state storage in the data-memory as compared to keeping it in the register-file (RF) and algorithmic chunking into independent blocks and spacing instructions to maximally utilize ILP and reduce the impact of read-after-write (RaW) and write-after-read (WaR). Specifically, when targeting masked designs, these are hard challenges for experienced designers because they require inline-assembly manipulations and are not natively supported by compilers. Furthermore, as the masking order (d) increases as the state gets larger, data memory accesses (reads and writes) for state handling needs to be increased since the RF is simply not large enough. This requires careful optimization which depends to a great extent on the underlying algorithm. One of the main issues we evaluate in detail is how full utilization of SSE extensions is not always possible; i.e., when d is not a a power of two. We pinpoint the optimal d values and draw attention to the very sub-optimal values of d which aggressively under-utilize the hardware.

One of the comparison points of our generic-C implementations is the SOTA masked bit-sliced compiled code developed by the Usuba team [MD19, BDM+20]. We show that the proposed generic-C implementation outperforms the Usuba compiled code in several cases in terms of cycle count but also (importantly) in code size. Our generic code does not require any additional auxiliary tools, additional formats, languages, or effort from the user. We put forward that our work is not aimed at providing a masked implementation behavioural security verification tool. However, it is aimed at discussing: (1) the gap between generic and optimized codes and, (2) the gap between codes which utilize such tools, and as such are abstracted, which may result in some performance degradation. We compare such an implementation (generated by the Usuba compiler) to both, general generic C codes we have developed and optimized assembly codes over extensive range of different ISAs (ARM and Intel and the possible extensions).

In terms of the masking gadgets explored here, we report results while considering several masked-multiplications: ISW- and UMA-based algorithms (Usuba only supports ISW-multiplication). We also report results when implementing single-input refreshes which

are not ISW-based (as supported by the Usuba tool). Note that masked-multiplication input refreshes were not used in the reports from [MD19, BDM+20] after verification that refreshes were not needed in this specific implementation (with Tornado and a SAT-solver tool). Here, we decided not to assume this type of scenario, since in the general case refreshes may be required or at least their need cannot be easily falsified with dedicated tools such as [BBD+16], FullVerif [CGLS20, CS21], MaskVerif [BBC+19]. Therefore, we provide results with and without a varying level of refreshes in our designs to better understand their impact on performance. In terms of refresh implementations, we first consider naive ISW-multiplication based refreshes [ISW03] which set one input to a logical '1' and then turn to the far more efficient refresh variants which were explored in [CGLS20] for the hardware implementation case. We provide a generic-C implementation by trading-off the randomness cost of the ISW-based refresh with $\lfloor (d-1)d/2 \rfloor$ [ISW03] and the more randomness efficient variant from HPC [CGLS20]. The randomness-cost of the UMA-based masked multiplication variant we explore is $\lceil (d-1)/4 \rceil \cdot d$ in bits [BDF+17, GMK18].

High performance masked software implementations are attracting growing interest: significant improvements and advances have been reported in [BGRV15, BS12, GR17, JS17, WVGX15]. However, many of these previous works do not evaluate very high order masked design efficiency over ISAs extensions, are focused on (ARM) NEON architectures, or only provide specific rather than generic implementations. Furthermore, most of these reports only provide results for AES or utilize less efficient primitives than the ones evaluated in this work. Vectorized ARM-based processors utilizing NEON were also evaluated in [GPSS18] targeting masked AES with specific d values with tailored inline-assembly constructs. In this work we evaluate Intel-based architectures and architectures without NEON extensions targeting simple (low-end, IoT) ARM architectures. Nevertheless, our generic-C implementations, which are evaluated over x86-64 based extensions, can be utilized to evaluate NEON-supported parallelism, thus extending the results from [GPSS18]. The building blocks and methodologies for our inline assembly optimized designs can be easily extended and evaluated on NEON architectures with our implementations, gadgets and refresh mechanisms. In Section 3.5 we relate to the structure of the provided open-source repository for the reader to be able to embed our primitives in other instances.

Both the generic-C implementations and the proposed optimized assembly codes reduce the utilization of conditional branches, jumps and function calls considerably. The goal is to improve performance while preserving a reasonable trade-off with increased code size, although these reduce the generality and ease of reading.

Contributions:

1. The generic-C codes were carefully designed to support various ISAs and extensions and be easily ported. They are mainly used as a comparison to illustrate the gap with more optimized designs, which constitute a contribution *per se*.

2. Full-fledged comparison with generic-C, through slightly more optimized-C flavors which are still generic (for all ds). We then present an optimized assembly and specially crafted assembly versions for ISA-extensions.

3. The versions of the optimized assembly depend on the specific ISA-extension used and d. These flavors result in significant gains as compared to the literature.

4. We show that for some d values and some ISAs, software masking is actually free in terms of performance; i.e., there is a performance loss of $O(1)$ as compared to the unprotected design.

5. We provide real-life test-cases on concrete, relevant cryptographic instances.

By all these contributions and designs we also support the open-source community and state of the art knowledge. In addition we emphasize the goal of promoting future research with or based on the designs developed.

Paper organization. This paper starts with a short background introduction to Intel architectures, SIMD extensions, and optimization trade-offs. It then discusses the bitsliced permutation used as an example in this paper and the masking gadgets used in Section 1. In Section 2 we detail the implementation aspects of our designs and the comparison designs. In Section 3 we provide a detailed comparison of Intel architectures, several ARM architectures, optimized assembly implementations and finally the cost of different refresh mechanisms. Section 4 discuses the main conclusions that can be drawn from this work.

1 Background

Modern x86 designs are pipelined, superscalar, and are also capable of out of order and speculative execution (via branch prediction, register renaming, and memory dependence prediction). This means they can execute multiple (partial or complete) x86 instructions simultaneously, and not necessarily in the same order as given in the instruction stream (instruction level parallelism, ILP). The out of order execution unit includes load/store buffers for committing memory reads/writes, thus reducing the penalty for memory writes to almost zero. In all x86 based processors there is a level 1 cache (L1) containing a few thousand bytes (at least), with a very fast access time of a few clock cycles. The size of the cipher; e.g., the Ascon-p [DEMS16] permutation in ISAP is 40 bytes ($40 \cdot d$ in d^{th}-order masked design), so it can easily fit inside an L1 cache, even for high order masking designs. All in all, x86 architecture offers excellent instruction level parallelism, and very good Memory Level Parallelism (MLP), which was exploited extensively in our implementations (when possible). However, in architectures that do not fully support ILP, MLP (e.g. simple or low-end/energy architectures), we also expect a significant impact in terms of the cipher's performance. In our implementations, we aimed to maximize the amount of independent instructions by splitting the operations among registers and reordering the independent operations (to be fair, by design, Ascon-p already has a great deal of instruction parallelism).

In the following, we briefly recapitulate the basic terminology used for SIMD extensions,

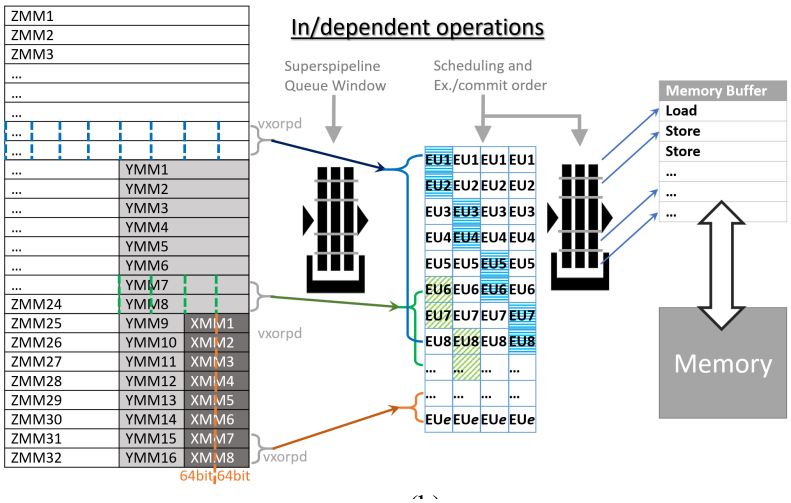


Figure 1: Schematic illustration: abstract view and utilization of X-, Y- and Z-MM registers enabling MLP and multiple execution units (EU) with ILP

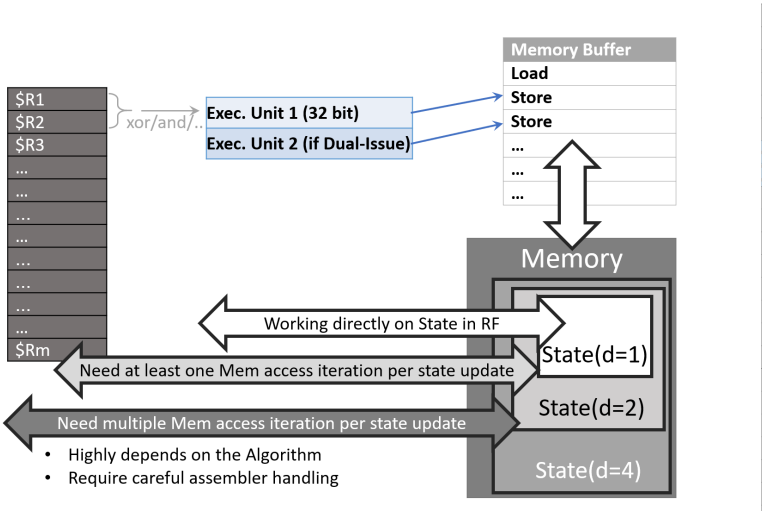


Figure 2: Schematic illustration: data movement between cache, Register-File and Memory buffer following execution, over low-end/ no-MLP architectures

the basic trade-offs at the heart of software optimization of bitsliced algorithms, and present some of the basic building blocks used in this research.

1.1 SIMD extensions and trade-offs

In most x86 based processors, at least one of the following SIMD extensions are present (illustrated in Fig. 1):

- SSE2/SSE3 - 8/16 (for 32/64 bit modes, respectively) 128-bit registers, accessible via XMM register names
- AVX2 - 8/16 (for 32/64 bit modes, respectively) 256-bit registers, accessible via YMM register names, in addition to the 16 registers described in SSE2.
- AVX-512 - 32 512-bit registers, accessible via ZMM register name; this extension also adds 16 128, 256 bits registers to the 16 existing registers described above.

In the following, we relate to some persistent challenges that are likely to exist for years to come for within-processor cryptographic computation in general. Modern workloads, such as big data searches, deep neural networks, graph and image processing, and high data-volume cryptographic applications are memory-bound. The limited memory provided by low-level caches (of any type) eventually may cause a bottleneck in the memory. Data movement between the main memory (and in some cases even from the L1 cache) and CPU cores impose a significant overhead in terms of both latency and energy, especially for low-end/ no-MLP architectures, as illustrated in Fig. 2. As the state size within a cryptographic algorithm increases (due to key size and security level, along with encoding and/or masking due to fault-injection or side-channel attacks), communication takes place through narrow buses with high latency and limited bandwidth. *The low data reuse in cryptographic algorithms cannot amortize the memory access cost in many cases.*

Working with data memory vs. different registers with SIMD extensions

In the following we discuss how to fetch data from the data-memory with different SIMD technologies in the context of the size of the available state (d dependent).

Fig. 3(a) shows the results from a benchmark of copying an array, using various SIMD technologies, measured in CPU clock cycles, as a function of the array size. It is clear that

the SIMD technologies (XMM, YMM, ZMM) outperform the native Register File, but there was little or no improvement among the SIMD registers. This can be explained in two ways:

1. The memory bus width is 256 bits, so that there is no additional performance gain when copying 256/512 bits at a time.

2. In larger registers (XMM → YMM → ZMM) the opcode sizes and the number of cycles per instruction (x86 is pipelined, so that the effect is reduced) increase and cancel out the effect of larger reads/writes to the Register File.

Nevertheless, SIMD still provides a 2x-4x performance gain compared to the native registers: for XMM: ∼2.20x gain. For YMM: ∼3.46x gain. For ZMM ∼4x gain.

Another issue relates to the under-utilization of the bandwidth to get data from the memory; i.e., whether it efficiently utilizes the memory-bus with d=2 for SSE2 or d=8 for AVX512. *This is directly related to the selection of the masking order (d) which corresponds to a specific ISA extension with out inline-assembly* implementations.

Computation time with SIMD extensions as the state size increases

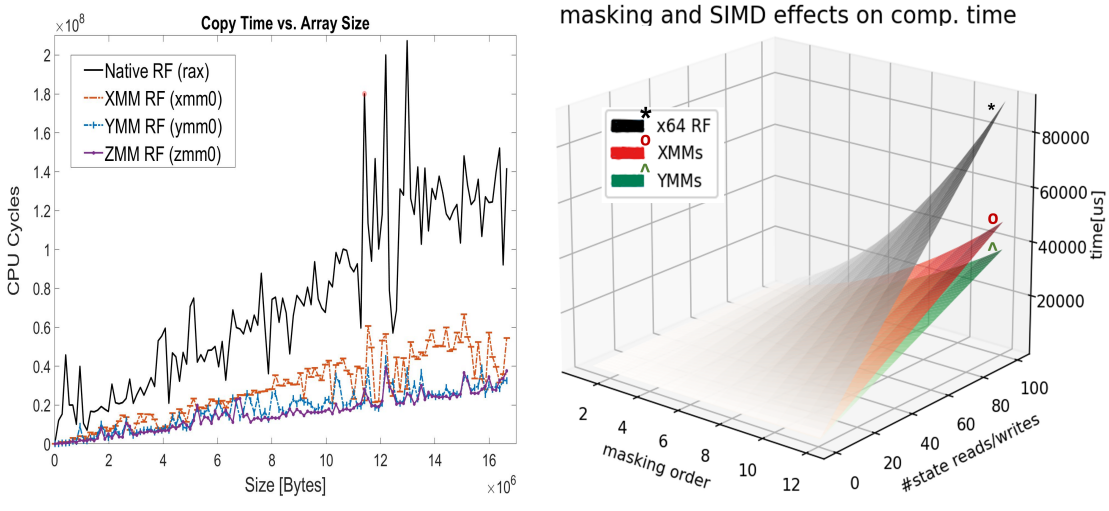
This final comment and example relate to storing states in the IMM/RF when possible, instead of always accessing the memory. Fig. 3(b) illustrates the results of a benchmark that worked exclusively with the RF, the XMM or the YMM registers. The figure shows how the computation time increases as the state size increases and depends on the number of reads/writes needed by the algorithm. As the number or reads/writes increases (with advances in SIMD technology) the computation time becomes dominated by memory access. This is clearly a function of the overall state size which, depending on the size of the registers, can or cannot be handled in registers enforcing memory-accesses. *This is another factor we carefully optimized in our inline-assembly versions.*

The level of Instruction Level Parallelism in Intel's architecture (i.e. the super-scalar/multi issue) is also an important factor. That is, even if we can enjoy processing parallelism equivalent to the memory access parallelism, it is not always possible given the *dependencies* between subsequent instructions, where the improvement can be cancelled out by scheduling, register-renaming, stalls, etc. Therefore, in the following sections, we also compare a "limited parallelism" architecture; e.g., non-NEON ARM low-cost /low-energy architectures.

In our implementations, we made no attempt to minimize the memory reads/writes as a criterion, but only where an impact was observed. This was done for several reasons: (1) The state is still rather small and fits easily into the L1 cache, (2) Memory Level Parallelism and register renaming essentially eliminate the penalty of a memory write, (3) *Owing to* rearrangements of instructions in our implementations the distance between writing-after-read of a certain memory address could be at least a dozen instructions, enabling the processor to execute almost non-stop.

1.2 Highly parallel-able bitsliced ciphers

The candidate we chose to illustrate results with in this paper is the Ascon-p [DEMS16]. It is the core permutation of Ascon, as proposed in the CAESAR lightweight competition. In fact, it is also the main building block of ISAP, an AEAD scheme which is one of finalist in the NIST lightweight cryptography standardization competition. However, the main reason this primitive was chosen is that it is a nice overall representative of a large class of sponge-based constructions (such as Keccak as used in SHA3). ISAP is more oriented towards providing protection against a fairly large class of implementation attacks (e.g. SCA and FIA) and is entirely based on the concept of *mode-level* security.



(a) (b)

Figure 3: (a) Working with data memory vs. different registers with SIMD extensions (b) Computation time with SIMD extensions as the state size/d increases

Therefore, Ascon-p can be used to realize a wide range of cryptographic computations such as pseudorandom number generation, authentication, encryption, authenticated-encryption and hashing. All of these can be appended with implementation security requirements or not. These properties make it a promising candidate for lightweight cryptography.

Note, however, that in this paper we are interested in the typical client/server asymmetry: although one party to the communication might be forced to run on an embedded device, the other might be very strong computationally. In other words, we were interested in implementations that are efficient *on both low-end and high-end devices*. Furthermore, we also aimed at providing efficient implementations over various architectures (supporting advanced ISAs extensions or not), and various security levels; i.e., with any security order (d) in the context of masking efficiency.

Ascon operates on a 320-bits state that is organized into 5×64 bit lanes, and is updated by the permutation Ascon-p. It consists of 3 steps that are applied consecutively on the state in each round: a constant round addition, a substitution layer, and a linear layer. The Ascon substitution-box (Sbox) is in fact very similar to the Keccak Sbox with the exception of several linear operations; namely, six XORs and one Invert, which in the context of masking are low cost, especially as d increases.

The Ascon-p permutation is organized in a sponge construction, which expresses it in terms of rate r and capacity c where $320 = r + c$. The rate in the sponge construction corresponds to the block size, whereas the capacity affects the security level. In this work we implemented the Ascon-p based instances used by Isap-A-128a. Ascon-p is built by default to support bitsliced, high parallelism implementations where the 64 5-bit Sboxes can be sliced and efficiently arranged as bit-operations between 64-bit words.

1.3 Masking gadgets

1.3.1 SOTA single input refreshes

- **Naive:** Standard ISW input refreshes can be implemented with Algorithm 1 which asserts one of the masked multiplication inputs to be refreshed and the other input as '1' (illustrated in Fig. 4(a)).

- **HPC:** A more efficient flavor in terms of randomness and latency was proposed

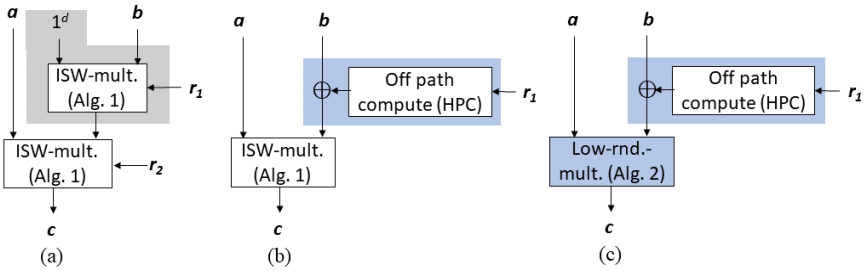


Figure 4: Schematic illustration: (a) refresh by ISW-multiplication gadget (b) refresh by HPC (c) refresh by HPC and multiplication by UMA

in [CGLS20] for all d's by utilizing the concepts of *on-path* and *off-path* randomness handling. Though it is more efficient for hardware scenarios, it is also efficient for software implementations, as implemented here, and illustrated in Fig. 4(b).

1.3.2 SOTA multiplication gadgets

Below we discuss generic multiplications, i.e., ones that their code implementation is quite d-independent, as well as the internal refresh mechanisms are rather easily coded with an high-level description.

- **ISW:** The baseline and generic multiplication gadget which is utilized here is the well-known ISW multiplication gadget (Algorithm 1). Its implementation is simple, is less platform/ISA-dependent and requires a small code size; however, it is quite randomness-hungry, as discussed above.
- **UMA:** The Unified Masking Approach, UMA, algorithm we embedded is more randomness-efficient. However, it is a more complex algorithm that require more operations (more energy hungry) and given in Algorithm 2. It is based on a series of optimizations starting from the parallel masking multiplication algorithm first proposed by Barth et al. [BDF+17], later optimized by Belai'd to reduce the randomness cost for specific ds and finally illustrated in a rather condensed form in [GMK18]. Although there are some randomness utilization optimizations for specific ds (e.g., for d={4,7} in [GPSS18]), our goal with the generic-C implementation was to implement a generic (d independent) code.

Algorithm 1 ISW multiplication.

Input: shares a_i and b_i, s.t. $\Sigma_i a_i = a$ and $\Sigma_i b_i = b$.
Output: shares c_i, s.t. $\Sigma_i c_i = a \otimes b$.

```
for i = 0 to d − 1 do
    c_i ← a_i ⊗ b_i;
end for
for i = 0 to d − 1 do
    for j = i + 1 to d − 1 do
        s ←$ F_{2^n};
        s' ← (s ⊕ (a_i ⊗ b_j)) ⊕ (a_j ⊗ b_i);
        c_i ← c_i ⊕ s;
        c_j ← c_j ⊕ s';
    end for
end for
return c_1, ..., c_d;
```

In Algorithm 2 a boldface lowercase letter denotes a vector of shares (i.e. $\boldsymbol{a} = (a_0, ..., a_{d-1})$). A subscript pre-pended with a $>$ symbol denotes a circular rotation of a vector and a superscript denotes an index to a subsection of a vector. In other words, in the case of $\boldsymbol{r}$, $\boldsymbol{r}^i_{>1}$ denotes the i^{th} subsection of size d, circularly shifted by 1 position. Operations are performed within each assignment from left to right to prevent recombinations.

2 Implementations

Various implementations were designed and tested, and fell into 3 main groups:

- Optimized assembly implementations: These implementations were written with inline assembly, using various SIMD instructions (see Section 1.1), for specific masking orders; e.g., SSE2 for 2^{nd} order masking, AVX2 for 4^{th} order masking, AVX-512 for 8^{th} order masking.

- Generic C implementations: These are generic implementations for every masking order, written in C. An optional optimization for 32 bits was added, to support 64 bit data operations (using SSE2).

- 3^{rd} party / open Source implementations: These implementations were developed / compiled by 3^{rd} party, and were included in our comparisons to evaluate our implementations; specifically, the Usuba compiled generic C implementation for any masking order [MD19, BDM+20], and the non-masked C implementation provided by the creators of the cipher [DEMS16].

Algorithm 2 UMA based generic SW masked multiplication.

Input: shares a_i and b_i, s.t. $\Sigma_i a_i = a$ and $\Sigma_i b_i = b$, and a uniformly drawn at random vector $\boldsymbol{r}$ chunked to $\lceil \frac{d-1}{4} \rceil$ vectors of d bits each.
Output: shares x_i, s.t. $\Sigma_i x_i = a \otimes b$.

$\boldsymbol{x} \leftarrow \boldsymbol{a} \otimes \boldsymbol{b}$;
for $i = 0 < \lfloor d/4 \rfloor$ **do**
  $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (((((\boldsymbol{a} \otimes \boldsymbol{b}_{>2i+1}) \oplus \boldsymbol{r}^i) \oplus (\boldsymbol{a}_{>2i+1} \otimes \boldsymbol{b}) \oplus \boldsymbol{r}^i_{>1}) \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2i+2})) \oplus (\boldsymbol{a}_{>2i+2} \otimes \boldsymbol{b}))$;
end for
$l \leftarrow \lfloor d/4 \rfloor$;
if $d = 3 \bmod 4$ **then**
  $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus ((((\boldsymbol{r}^l \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2i+1})) \oplus (\boldsymbol{a}_{>2i+1} \otimes \boldsymbol{b})) \oplus \boldsymbol{r}^l_{>1}) \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2i+2}))$;
else if $d = 2 \bmod 4$ **then**
  if $d = 2$ **then**
    $\boldsymbol{z} \leftarrow \{r^l_0, r^l_1, r^l_0 \oplus r^l_1\}$;
    $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus ((\boldsymbol{z} \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2l+1})) \oplus (\boldsymbol{a}_{>2l+2} \otimes \boldsymbol{b}))$;
  else
    $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (((\boldsymbol{r}^l \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2l+1})) \oplus \boldsymbol{r}^l_{>2l+2}) \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2l+2}))$;
  end if
else if $d = 1 \bmod 4$ **then**
  $\boldsymbol{z} \leftarrow \{r^l, r^l\}$;
  $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (\boldsymbol{z} \oplus (\boldsymbol{a} \otimes \boldsymbol{b}_{>2l+1}))$;
end if
return $\boldsymbol{x}$;

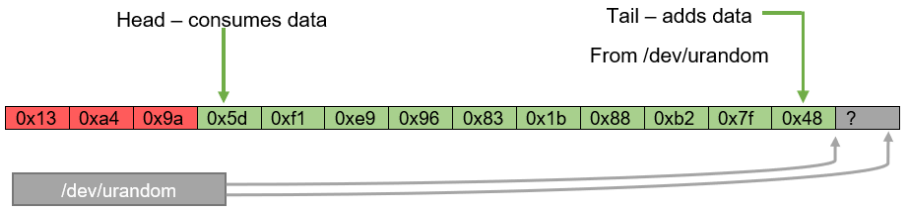


Figure 5: Random buffer implementation

A fair comparison

Efficient randomness throughput handling was carried out by an efficient randomness buffer to obtain the true performance of each implementation, we had to eliminate the biggest bottleneck in our system; namely, randomness bandwidth (e.g., it takes time to generate random bytes and read them from a file). Therefore, we implemented a class which manages a ring buffer that fills up with random bytes (taken in our case from /dev/urandom) whenever needed as illustrated in Fig. 5. The one exception was Usuba, which does not come with a built-in solution for dealing with the randomness bandwidth (there is only a place holder that uses srand(time(NULL)); rand() to get 8 random bytes, which is far from an efficient placeholder). Thus, we integrated our system into the Usuba compiled code for a fair comparison. This was the only change we made to their code, to ensure the best possible performance from Usuba and guarantee comparisons on equal grounds.

Inline optimizations:

We next list the main inline optimizations done by our methodological optimization flow which are algorithmic dependent. That is, they require code/algorithmic understanding, and partitioning to RaW and RaW/WB independent blocks (these procedural steps are recalled below when we discuss the optimization flow-chart):

1. Minimizing Read After Write dependencies which is done with re-ordering instructions and maximizing register usage.

2. Minimizing memory access, by loading the entire state to the register file for as long as possible.

3. Inlining functions by using mostly macros (guarantied to inline the code) or by using the “inline” keyword (depends on compiler’s optimization policy). On the other hand, inlining comes at the cost of increased code size, so as higher masking orders are used / functions get longer, a function’s definition is altered from a macro to an inline functions. (examples: optimized assembly for masking order 4 versus masking order 8, generic C for any masking order).

Fig. 6 of our code illustrates a major design pattern found in our assembly implementations relating to points 1 and 2 above. The chart to the right of the figure shows the usage of the state with the registers (YMM0 to YMM4). It reveals a considerable gap of at least 4 cycles between writing to the registers and reading / writing afterwards. Note that modern compilers can optimize code with the methods mentioned above, but with our experience, GCC does not handle inline assembly optimizations well (the code generated a SEG fault, so we have added the ‘volatile’ keyword).

In Fig. 7 we depict the usage of the entire Register File, while inline-ing each sub-function in Ascon-p (in this case, the linear diffusion layer, LDL). For example, the three highlighted routines here are a section of the full LDL, in this case it corresponds to a 4^{th}

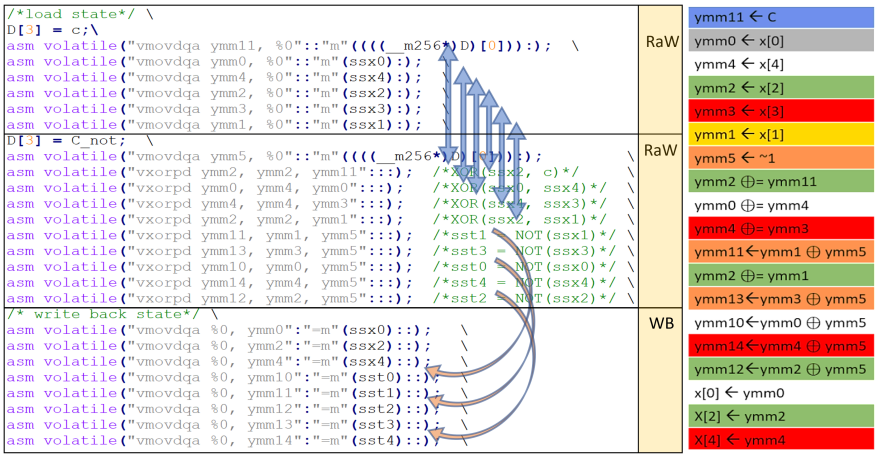


Figure 6: Minimizing RaW and memory-access

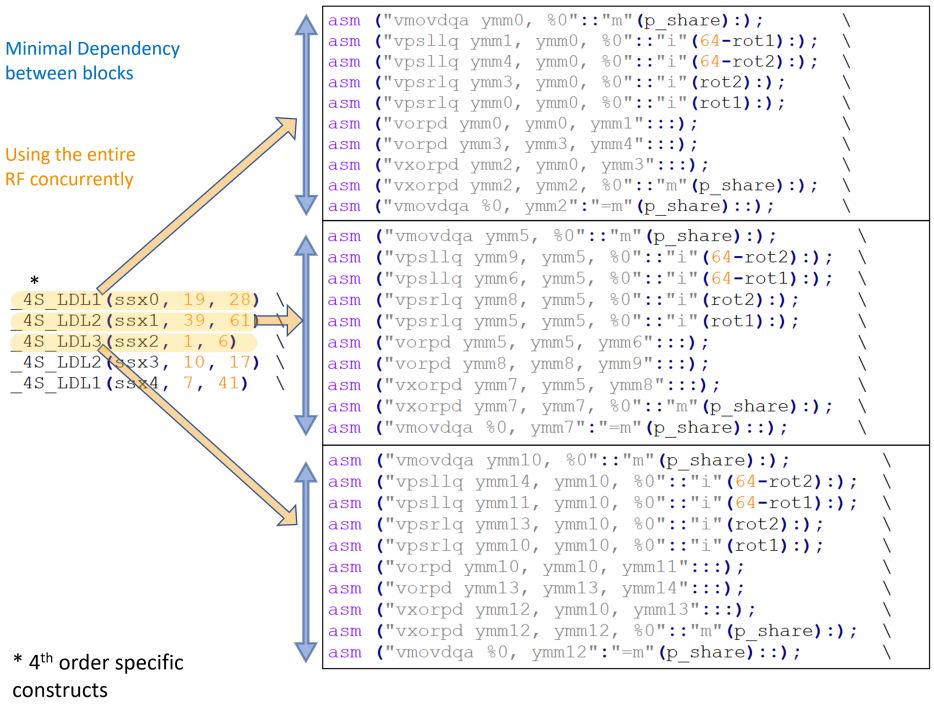


Figure 7: Usage of the entire register file in the linear layer and minimal block dependency

order masking and AVX-2 (i.e., were implemented per extension flavour and d pairs). As stated above, we minimized the Read after Write dependencies in our code as much as possible by reordering instructions and using as much as possible registers, as can be seen from each code section. Note that we have maximized the register usage and removed the 'volatile' keyword so GCC could reorder the instructions.

In Fig. 8 we illustrate optimizations relating to the third optimization method regarding inlining functions by macros: Code-size versus Cycle-count optimization by functions versus macros (or inlined functions) - The benefit of using a macro is the elimination of branching / stack operations overhead, the downside is the codesize increases for each usage of the macro. The benefit of using functions is reduction of overall code-size if the function is

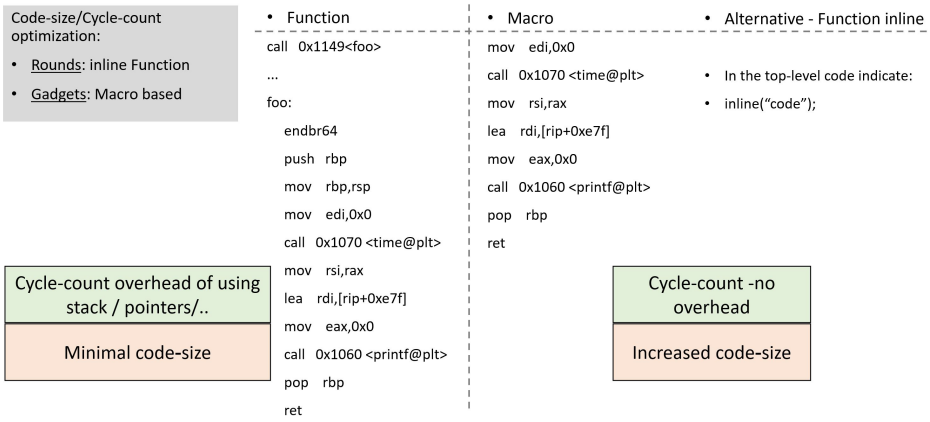


Figure 8: Code-size versus Cycles-count optimization - Functions versus Macros (or inlined functions). A disassembled piece of code is provided.

called frequently. Specifically in our library, as permutation rounds' code is huge, using each round as a macro will not be efficient in terms of code size (10 rounds inlined) and will only cost little in cycle count overhead due to branching/stack handling. However, at the gadgets level, as the macros code is small and only repeats a small number of times (usually a dozen times or less) in a bit-slice vectorial representation in a round, it is efficient to use macros and save cycles even if it slightly increase code-size.

Additional optimizations which we have performed are listed below. Admittedly, these are more trivial/standard steps for assembly optimizations. However, they constitute a huge engineering effort and are indeed important to enumerate. The following list contains key ingredients to improved code-size efficiency and some performance (cycle-count) is lost if not properly embedded:

1. Constant reusing by defining file-wide static variables (memory constants) and using them throughout the program's lifetime (a keen eyed reader would observe the header consts.h used in every file, This optimization both reduces code size and data section size.

2. Loop unrolling - done manually for all optimized assembly implementations, but sometimes also for C code (e.g., paralleling independent operations).

3. Vectorizing loops (done by the compiler by enabling optimizations).

These optimizations are merely examples and are extensively used by different SIMD flavors in our codes and different parts of the algorithm. All are available in our fully public GitHub[1] repository: this repository is constructed especially for embedding the developed optimized gadgets in all ISAs and in additional algorithms, optimizing masked designs. It is packed with supporting examples and supplementary content for evaluation, testing, profiling etc. as discussed in Subsection 3.5, for easily porting.

Optimization process and comparative example:

In this subsection we give a high-level overview of utilizing the macros developed through an example of an LDL implementation. Fig. 9 illustrates in a comparative view of different implementations for one LDL macro listing: (a) pseudocode, (b) SSE2/3, (c) AVX2, (d) AVX512, and (e) C-code. As the C-code indicates, the LDL macro performs two

[1]https://github.com/dorsal1464/ascon-p

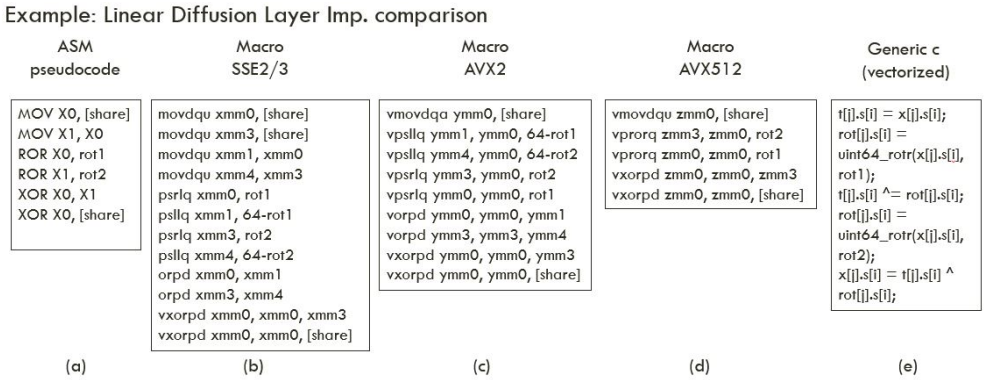


Figure 9: Example of ascon-p's Linear Diffusion Layer implementation for different implementations - (a) Pseudocode (b) SSE2/3 (c) AVX2 (d) AVX512 (e) c-code

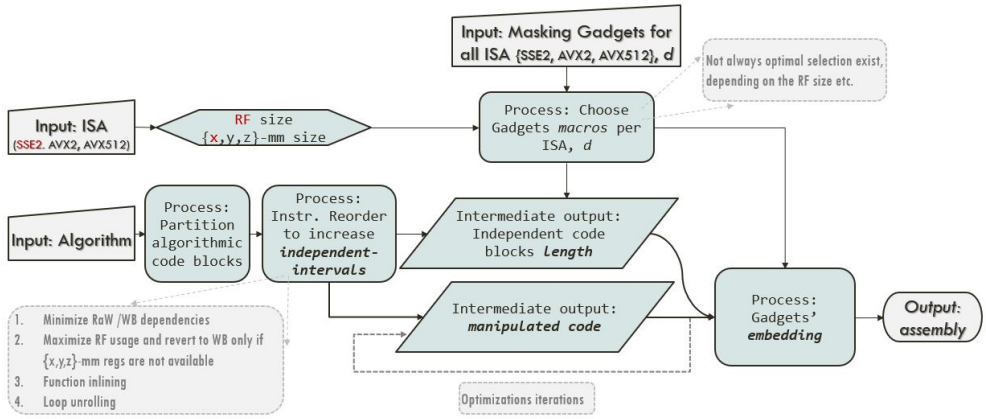


Figure 10: Optimization Flow-chart- Masked Assembly optimization process

rotations on the input and sums them all up with the original value (using xor). As can be seen from the comparison, the advanced architectures implements more sophisticated and extensive instructions that allow smaller codesize and faster (overall) runtime. A keen eyed observer may notice that only AVX-512 supports rotate-right (ROR) as an instruction, this is no mistake and it emphasizes the fact that advanced ISAs can improve computation time / codesize in more than one way (i.e., bigger vectors for faster processing).

The optimization flow-chart, indicating the methodological framework developed, and is provided in Fig. 10. It illustrates the cryptographic engineering optimization process which was taken in this research: first a primitive (algorithmic gadget) is chosen and implemented on all specific ISAs' and extensions and for all d values. Then the process receives as inputs the supporting hardware selection (i.e., ISA and extension), and the high level description cryptographic algorithm. Then specific gadgets are chosen per d and ISA/extension following an instruction reordering of the algorithm (minimizing RaW, WB, loop unrolling and functions, function inlining) which enables identification of independent code-blocks length. Then, this intermediate output is given a few optimization rounds to provide the best gadgets for embedding, and the final assembly is exported. These optimizations are described in Section 2.

3 Evaluation

In all the evaluation benchmarks discussed below we implemented $10^5/d$ repeated Ascon-p blocks with random inputs; i.e., $10^5/d$ calls for each d, design flavor, architecture flavor and optimization mode. This is because as d increases, the input size increases and cannot be stored in memory as is (a typical limit is around 500MB). On the one hand, to obtain solid statistics and robust measurements for metrics such as cycles/bit and randomness usage we need a significant sample space, but on the other, it is impossible to do so when using large d values to generate random inputs or getting them directly from the randomness source, since this would detract from the validity of our measurements. Therefore, in our experiments we limited the number of experiments by normalizing them to the maximum number of inputs that could fit in the cache/buffer (which was easily achieved by dividing by d). In all the experiments, and most importantly for large d values, the number of tests were always far more than enough to achieve convergence on our evaluated parameters.

3.1 Generic Implementation Efficiency vs. d for x86 and x86-64 Intel architectures

Fig. 11 depicts various performance results for the proposed generic C implementations on the Ascon-p permutation block, tested on both the x86, and x86-64 Intel architectures. For each architecture we illustrate the cycles per bit, randomness usage (in units of bytes), and code size of each implementation (as loaded into memory). Note that all the randomness usage figures in this work were divided/normalized by a factor of d, which enable easy visualization of trends and differences between curves. "Proposed (generic C)" refers to our efficient implementation (with and without UMA AND gates), "Usuba compiled" refers to the C code generated for Ascon-p by the Usuba compiler. In all the implementations, optimization flags were set (gcc -O3) to provide the best possible results,

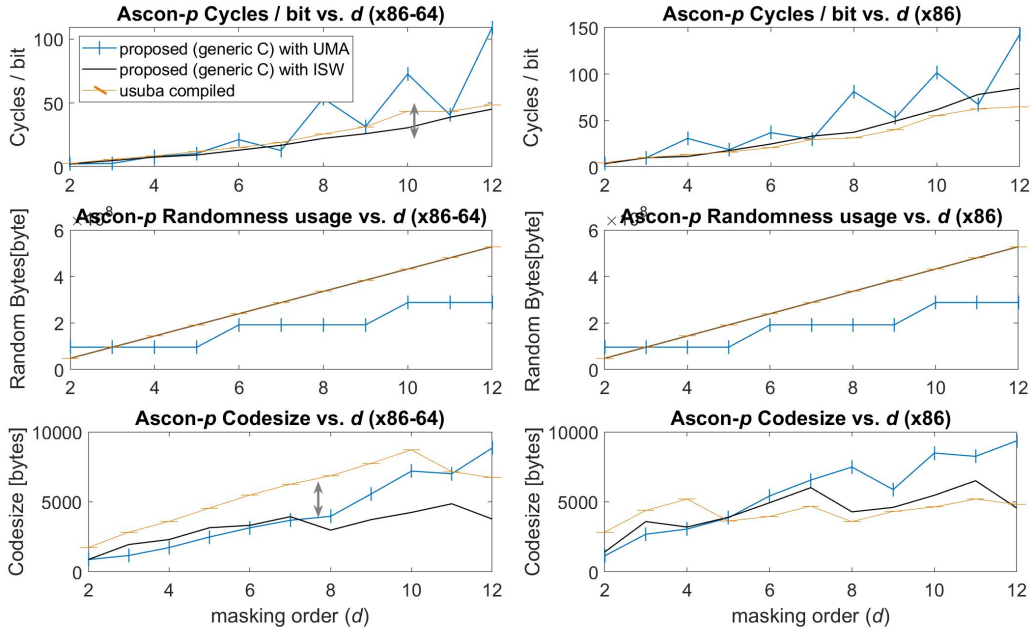


Figure 11: Generic Implementation Efficiency vs. d for x86 and x86-64 Intel architectures

including vectorization of the code. The proposed implementation achieved somewhat better performance in x86-64 (in the range of 5-10%), and slightly worse in x86 compared to the Usuba compiled code, with a maximum 7% degradation. Even though our proposed code was not considerably faster, it used considerably less code than the Usuba code size in x86-64; namely, up to a 50% code-size reduction (in our experience, the Usuba compiled code looked bloated with redundant variables and state copies). As discussed in the introduction, one of the goals of generic C codes is to promote transparency, reduce the need for additional auxiliary tools, formats, languages, or effort from the user except for the official and public ISA. Therefore, exceeding the results from the highly professional Usuba tool and remaining on a par with it in other scenarios constitute a very nice additional added value. In terms of x86, as shown on the right side of Fig. 11, we were not able to detect a clear trend from the code size graph since neither code size exhibited a consistent trend (recall that Ascon-p is optimized for 64-bit registers). As stated above, the findings must be taken with a grain of salt, since an increased code size could imply more vectorization / loop unrolling, which is not a bad thing in cycle counts. However, while trying to find a good balance while maintaining superb cycle counts we observed that the product metric of cycle count and code size was much more efficient in x86-64 for the proposed generic C. Note as well that despite the generality of the code and its independence in terms of auxiliary tools, it was mainly aimed at pinpointing the gap with highly inline assembly optimized codes, as discussed below in details.

The blue curves in Fig. 11 which correspond to the proposed code with UMA AND gates reveals that:

1. The cycles per bit had a "zigzag" pattern, for several reasons. The UMA AND gate is much more complex / long compared to its ISW counterpart; thus, it has more branches and fewer loops, making it harder for the compiler to optimize and vectorize the code for the masking orders in the best way possible.

2. The randomness usage plot has a step-like shape that increases every four masking orders, but always uses fewer random bytes than the other implementations, which is its main added value and motivation. This stems from the design of the UMA AND gates.

3. The code size of the proposed code (with UMA) is considerably larger than other implementations. As stated above, this is due to the fact that the UMA AND gate is much more complex.

Interestingly, an examination of the code of the UMA AND gate showed that for certain masking orders, the cycles per bit was (slightly) better (as observed in the figure), whereas for others, the cycles per bit were much higher. This is related to the branches, since for masking orders that do not divide by 4, we eventually end up with a much larger function that corresponds to the existence of complete, incomplete and pseudo-complete branches [GMK18]).

3.2 Generic Implementation Efficiency vs. d for 32 and 64 bit ARM architectures

Fig. 12 illustrates the performance metrics of the generic C implementations on the Ascon-p permutation block, emulated on the ARMv7 (32-bit), and aarch64 (64-bit) ARM architectures. For each architecture we calculated the cycles per bit, randomness usage, and code size of each implementation (as loaded into memory). The proposed implementation achieved somewhat better performance for some masking orders, and up to 17% worse for others in aarch64. On the other hand in ARMv7, the performance was identical compared to the Usuba compiled code. Again, cycles per bit was emulated and should not be taken as a complete case. Regarding code size,the proposed code was significantly smaller in

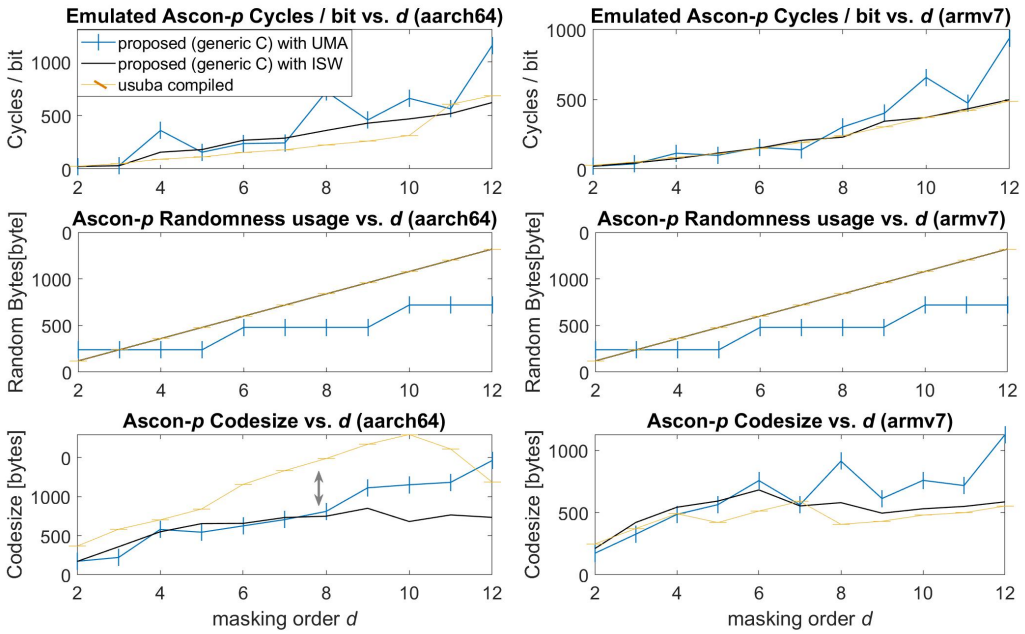


Figure 12: Generic Implementation Efficiency vs. d for 32 and 64 bit ARM architectures

aarch64, and since QEMU's ARM emulation has no special SIMD, there could be direct correlation between increased code size and better performance (because increased code size could imply more loop unrolling). In ARMv7, both graphs followed the basic trend. In terms of the proposed code with UMA AND gate, similar trends as observed for the x86/64 architectures emerged.

3.3 Tailored Implementation Efficiency vs. d and extension type, and the gap from generic-designs

As discussed above, the x86 architecture has three major SIMD extensions: SSE2/3/4 is accessible via XMM register names, AVX2 is accessible via YMM register names, and AVX-512 is accessible via ZMM register names (this extension is only available in the 64-bit mode). Given the constraints mentioned above,in our x86 (32-bit) benchmark, there was no optimized assembly for masking order 8 (usage of AVX-512 is not supported), and the code was slower and longer due to smaller register files.

The performance metric results of the generic C implementations and our optimized assembly on the Ascon-p permutation block are illustrated in Fig. 13, which was tested on both the x86, and the x86-64 Intel architectures. As discussed above, for each architecture we calculated the cycles per bit and the code size. The optimized assembly legend entry refers to the assembly implementations we wrote for masking orders 2, 4, 8 for x86-64. For x86 there is no optimized assembly for masking order 8 as mentioned above. The baseline (no masking) implementation results from the official inventors of Ascon-p also appear on the plots for comparison, as indicated by the gray diamond mark in the $d = 1$ entry. In all implementations, optimization flags were set (gcc -O3) to provide the best possible results including vectorization of the code, although for our optimized assembly, it had little effect. Even though we optimized the proposed generic C to the limit, it still struggled to keep up with our optimized assembly. The shortfall in this case reached 50% in cycles per bit which is very high. Note that in all our assembly implementations, we also included input

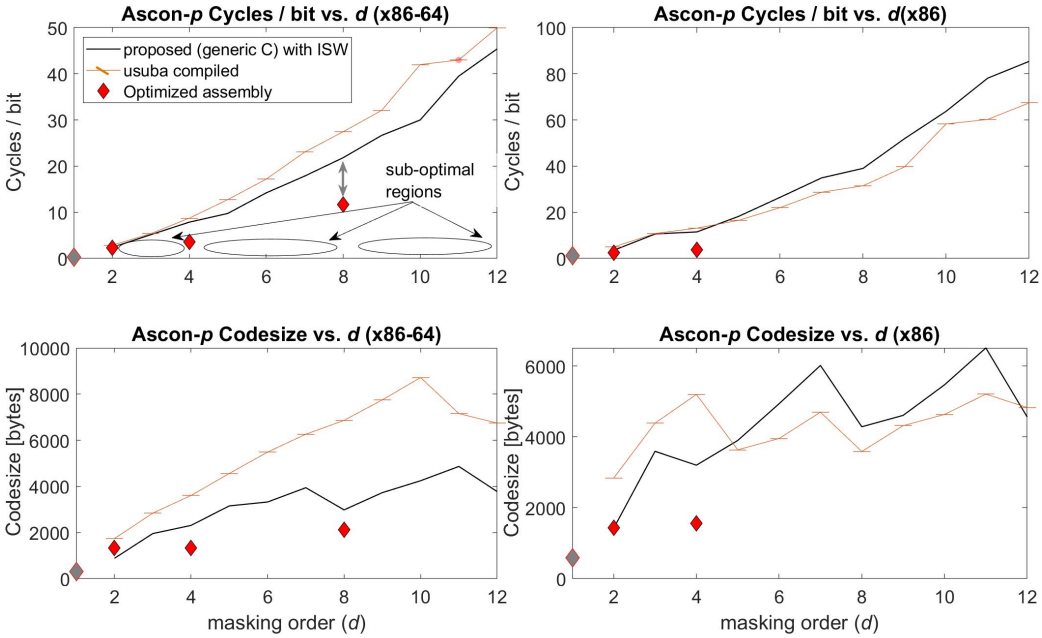


Figure 13: Tailored Implementation Efficiency vs. d and extension type and the differences with generic designs

refreshes (unlike the proposed generic C and Usuba), so real raw performance would be slightly better. Furthermore, in our proposed generic C code, in 64 bits we were faster than in 32 bits for all ds, as we would expect, but in Usuba the situation was sometimes the other way around. This may hint that the Usuba compiler does not generate code aiming for good optimizations for 64 bit architectures. In terms of code size, our assembly code was 100% unrolled, whereas the proposed C code was not (mainly AND gates unrolling), but in almost all cases our assembly implementation was still better, providing even joint and considerable added value for cycles/bit and code-size.

The final and important highlight from the figure is that full utilization of SSE extensions is not always possible. For instance, when d values are not a power of two, the effort will strain the memory access and under-utilize the memory interface hardware by creating vacant information traffic. The need to work with (e.g.) ZMM with d values in the range 5 to 7 will reduce the cycles per bit since these accesses are slower (Fig. 2 and 1) thus pinpointing the distinct optimal d values and very sub-optimal progressively improving values of d which aggressively under-utilize the hardware (these are illustrated with ellipses in the figure). All in all, ideally we would have hoped that the SIMD progressive extensions would give us masking for "free" (at least in terms of throughput). In practice, we were not far from this in terms of cycles per bit since with d=2 and 4 we were very close to the unmasked official design. However, with d=8 we began to observe an impact. As hinted in Sub-section 1.1 this is due to the limited memory-bus width and the payload of the ZMM registers opcodes. However, the results still indicate very significant gains as compared to the generic C implementations, thus justifying the use of such techniques for similar cryptographic primitives.

3.4 The cost of Refreshes

Finally, in Fig. 14 we illustrate the performance of generic C implementations with various refresh levels/implementations on the Ascon-p permutation block AND gates

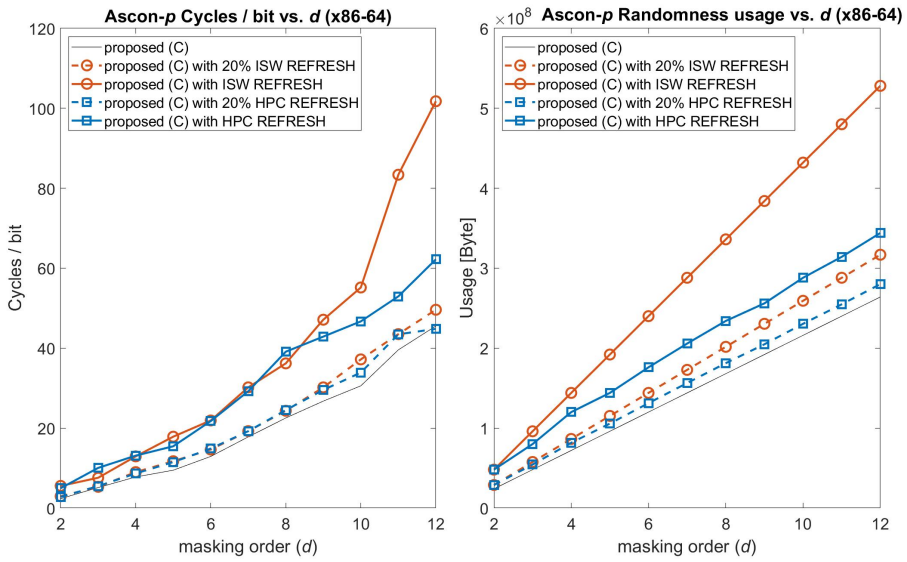


Figure 14: Performance of generic C implementations with various refresh levels/implementations

single-input, as tested on the x86-64 architecture. In each architecture we calculated the cycles per bit and randomness usage.

Clearly, adding 20% refreshes (one out of five ANDs in an Sbox) with either method yielded very good performance, almost on a par with the original proposed generic C. However, at full (100%) input refreshes there was a considerable difference in performance in both refresh methods compared to no refreshes, as expected. For larger masking orders, the ISW became much more expensive in terms of performance. With respect to randomness usage, HPC used considerably fewer random bytes, which justifies its utilization in cases where refreshes are needed or their need is hard to falsify.

3.5 Designed for generalization and embedding

Our tailored repository was optimized for constructive reuse, generalization, embedding in new projects and envisioned by open-source spirit. Gadgets/Macros' repository structure and design examples illustrate this property. It is generic in the sense that they are rather easily applicable to all bit-sliced designs. New or other bit-sliced designs (SPN-based and others) can be easily ported. The flow for embedding new algorithms is shortly discussed.

As illustrated in Fig. 15, which visualizes part of the structure of our MaskSIMD-lib, it is partitioned to ciphers examples, gadgets libraries state handling and auxiliary tools, in addition, more branches exist for test benching evaluating efficiency metrics, randomness handling etc. For a new algorithm to be implemented by the methodological framework developed, first a selection of primitives is done by branching from levels 2 to 4 as highlighted in the figure, these are set owing predefined architectural parameters of the platform. Then the designer turns to perform optimizations steps as listed in Optimization Flow-chart (Fig. 10) and partitioning to independent algorithmic code blocks (recall Fig. 10). Next a designer may utilize the test-benches and profiling tools developed specifically for the evaluation of Masked permutations: located in "scripts/", state handling in memory examples are located in "src/gadgets/state", efficient randomness buffer is located in "src/random" and utilization examples are provided in the test-benches. Finally, evaluation can be performed.

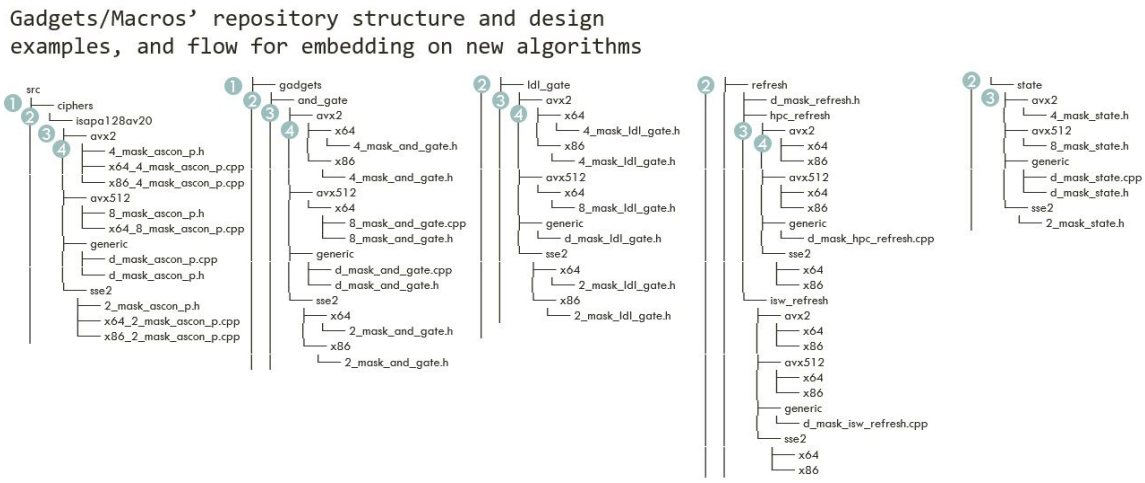


Figure 15: Gadgets/Macros' repository structure and design examples, and flow for embedding on new algorithms. A user may choose a specific gadget macro (2) from a specific genre (1), then choose the architecture (3 & 4).

4 Discussion and Conclusion

High performance masked software implementations are attracting significant interest. In this work we evaluated the efficiency of high order masked designs over different ISAs extensions, specifically targeting Intel x86/-64 architectures and SSE3/AVX-2/AVX-512 extensions. We evaluated non-NEON ARM architectures and provided ultra-specific assembly optimized implementations (different d to optimally match the level of extension) and fully ported and flexible generic C implementations for all d values with several levels of optimizations and parametric natures for gadget selection. Whereas most previous studies on software masking have focused on the AES algorithm, NEON architectures or provide some specific d values results, in this work we provide a complementary view with the Ascon-p sponge permutation and a variety of architectures ranging from low-cost to ultra high-performance, and for all d values. We concretely embed state of the art masking gadgets in this evaluation to better understand the differences between utilizing specific multiplication algorithm or a specific refresh gadget. To the best of our knowledge, this constitutes the first published results on some of these combinations.

Both the generic-C implementations and the proposed optimized assembly codes considerably reduced the utilization of conditional branches, jumps and utilization of function calls in an attempt to improve performance while preserving a reasonable balance with increased code size. Although some of these features reduce generality and ease of reading and are tedious, the shortfall we observed with our evaluation metrics was notable. The major design patterns found in our optimized assembly implementations for each SIMD extension flavor are as follows: (1) Minimization of read after write and write after read dependencies, achieved by instruction re-ordering and with maximum register usage (2) Minimizing memory access, by loading the entire state to the register file for as long as possible and, (3) Usage of the entire register file, while inline-ing each sub-function.

The generic-C codes were carefully designed to support various ISAs and extensions and to be easily ported. They were mainly used as a comparison to identify differences from more optimized designs, but constitute a contribution in their own right. As compared to the Usuba compiler results, the proposed generic C implementation achieved better performance in cycles/bit for the x86-64 architecture (in the range of 5-10%), and on both x86-64/x86 the code size of our generic C was up to 50% smaller (in fact, we can trade

off and achieve fewer cycles/bit for slightly more code area). One of the highlights of this work is that the generic C codes promote design transparency, reduce the need for additional auxiliary tools, additional formats, languages, and effort from the user. Therefore, outperforming the results of the highly professional Usuba compiler and remaining on a par with it in other scenarios is a very nice additional added value.

The optimized inline assembly versions demonstrate a gap of up to 50% in cycles per bit, which is considerable, between the most optimized generic C design (and Usuba), and also concurrently provide far more code size efficiency. We discuss the fact that full utilization of SSE extensions is not always possible; for instance when d values are not a power of two (with a 64 bit word size in Ascon), which places excessive strain on memory access that under-utilizes the memory bandwidth. This underscores the distinct optimal d values and very sub-optimal progressively improving values of d. Finally, we reported on the relative gap of several refresh gadgets and discussed its wide range in terms of cycle counts and randomness usage, thus highlighting that although different types of gadgets reduce generality and require different building blocks for different parts of the implementation, the use of more advanced refresh mechanisms is worthwhile.

Finally, we believe that providing real-life test cases with concrete and relevant cryptographic instances such as sponges (which propagate to various other cryptographic primitives) can contribute to the open-source community and the sharing of knowledge and expertise, thus overall promoting future research with or based on these designs.

Acknowledgments

This research was funded by Israel Science Foundation (ISF) grant number 2569/21.

References

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. Maskverif: Automated verification of higher-order masking in presence of physical defaults. In *European Symposium on Research in Computer Security*, pages 300–318. Springer, 2019.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 535–566. Springer, 2017.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Eurocrypt 2020-39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 12107, pages 311–341. Springer, 2020.

[BDMD+20] Begül Bilgin, Lauren De Meyer, Sébastien Duval, Itamar Levi, and François-Xavier Standaert. Low and depth and efficient inverses: a guide on s-boxes for low-latency masking. *IACR Transactions on Symmetric Cryptology*, 2020(1):144–184, 2020.

[BGRV15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. Dpa, bitslicing and masking at 1 ghz. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 599–619. Springer, 2015.

[BS12] Daniel J Bernstein and Peter Schwabe. Neon crypto. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 320–339. Springer, 2012.

[CGLS20] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Transactions on Computers*, 2020.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition-and glitch-robust probing model: Better safe than sorry. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 136–158, 2021.

[DEM+20] CE Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2. 0. 2020.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 401–429. Springer, 2015.

[GM18] Hannes Gross and Stefan Mangard. A unified masking approach. *Journal of cryptographic engineering*, 8(2):109–124, 2018.

[GMK18] Hannes Gross, Stefan Mangard, and Thomas Korak. *Domain-Oriented Masking–*. PhD thesis, Graz University of Technology, Austria, 2018.

[GPSS18] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. Vectorizing higher-order masking. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 23–43. Springer, 2018.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 567–597. Springer, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 623–643. Springer, 2017.

[MD19] Darius Mercadier and Pierre-Évariste Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–173, 2019.

[MMSS18] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited-or why proofs in the robust probing model are needed. *Cryptology ePrint Archive*, 2018.

[Pap18] Kostas Papagiannopoulos. Low randomness masking and shuffling: An evaluation using mutual information. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 524–546, 2018.

[WVGX15] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. Higher-order masking in practice: A vector implementation of masked aes for arm neon. In *Cryptographers' Track at the RSA Conference*, pages 181–198. Springer, 2015.